

D:\MAL23-P7-G5-Main\Engine\Source\GameEngine\Utilities\Structures\Octree.hpp

```
1  #pragma once
2
3  #include <vector>
4  #include <shared_mutex>
5  #include <concepts>
6
7  #include <CommonUtilities/Math/AABB.hpp>
8  #include <CommonUtilities/Math/Vector3.hpp>
9  #include <CommonUtilities/Math/Frustum.hpp>
10 #include <CommonUtilities/Math/Intersection.hpp>
11 #include <CommonUtilities/Math/Sphere.hpp>
12
13 #include <CommonUtilities/Structures/FreeVector.hpp>
14
15 /// Octree based upon: https://stackoverflow.com/questions/41946007/efficient-and-well-explained-implementation-of-a-quadtree-for-2d-collision-det
16 ///
17 template<std::equality_comparable T>
18 class Octree
19 {
20 public:
21     using ElementType    = T;
22     using ValueType      = std::remove_const_t<T>;
23     using SizeType       = int;
24
25     static constexpr SizeType ourChildCount = 8;
26
27     struct Element
28     {
29         cu::AABBf    aabb;    // aabb encompassing the item
30         ValueType    item;
31     };
32
33     Octree() = default;
34
35     Octree(const cu::AABBf& aRootAABB, int aMaxElements = 16, int aMaxDepth = 16);
36
37     int ElementCount() const;
38
39     const cu::AABBf& GetRootAABB() const;
40
41     void SetRootAABB(const cu::AABBf& aRootAABB);
42
43     /// Inserts given element into the quadtree.
44     ///
45     /// \param AABB: Rectangle encompassing item
46     /// \param Args: Constructor parameters for item
47     ///
48     /// \returns Index to element, can be used to directly access it when, e.g., erasing it from the tree
49     ///
50     template<typename... Args> requires std::constructible_from<T, Args...>
51     auto Insert(const cu::AABBf& aAABB, Args&&... someArgs) -> SizeType;
52
53     /// Attempts to erase element from tree.
54     ///
55     /// \param Index: Index to element to erase
56     ///
57     /// \returns True if successfully removed the element, otherwise false
```

```

58     ///
59     bool Erase(SizeType aIndex);
60
61     /// Updates the given element with new data.
62     ///
63     /// \param Index: index to element
64     /// \param Args: Data to update the current element
65     ///
66     /// \returns True if successfully updated the element, otherwise false
67     ///
68     template<typename... Args> requires std::constructible_from<T, Args...>
69     bool Update(SizeType aIndex, Args&&... someArgs);
70
71     /// Retrieves an element.
72     ///
73     /// \param Index: index to element
74     ///
75     NODISC auto Get(SizeType aIndex) -> ValueType&;
76
77     /// Retrieves an element.
78     ///
79     /// \param Index: index to element
80     ///
81     NODISC auto Get(SizeType aIndex) const -> const ValueType&;
82
83     /// Retrieves the aabb encompassing item
84     ///
85     /// \param Index: index to item.
86     ///
87     NODISC auto GetAABB(SizeType aIndex) const -> const cu::AABBf&;
88
89     /// Queries the tree for elements.
90     ///
91     /// \param Frustum: Frustum to search for overlapping elements
92     ///
93     /// \returns List of entities intersecting the frustum
94     ///
95     NODISC void Query(const cu::Frustumf& aFrustum, std::vector<SizeType>& outResult) const;
96
97     /// Queries the tree for elements.
98     ///
99     /// \param Frustum: Frustum to search for overlapping elements
100    ///
101    /// \returns List of entities intersecting the frustum
102    ///
103    NODISC void QueryNoDepth(const cu::Frustumf& aFrustum, std::vector<SizeType>& outResult) const;
104
105    /// Queries the tree for elements.
106    ///
107    /// \param Frustum: Frustum to search for overlapping elements
108    ///
109    /// \returns List of entities intersecting the frustum
110    ///
111    NODISC void Query(const cu::Vector3f& aStartPos, const cu::Vector3f& aEndPos, std::vector<SizeType>& outResult) const;
112
113    /// Queries the tree for elements.
114    ///
115    /// \param AABB: Bounding box to search for overlapping elements
116    ///
117    /// \returns List of entities intersecting the aabb

```

```

118     ///
119     NODISC void Query(const cu::AABBf& aAABB, std::vector<SizeType>& outResult) const;
120
121     /// Queries the tree for elements.
122     ///
123     /// \param Point: Point to search for overlapping elements
124     ///
125     /// \returns List of entities intersecting the point
126     ///
127     NODISC void Query(const cu::Spheref& aSphere, std::vector<SizeType>& outResult) const;
128
129     /// Queries the tree for elements.
130     ///
131     /// \param Point: Point to search for overlapping elements
132     ///
133     /// \returns List of entities intersecting the point
134     ///
135     NODISC void Query(const cu::Vector3f& aPoint, std::vector<SizeType>& outResult) const;
136
137     /// Performs a lazy cleanup of the tree, should be called after items have been erased.
138     ///
139     void Cleanup();
140
141     /// Clears the tree
142     ///
143     void Clear();
144
145     std::vector<cu::AABBf> GetBranchAABBs() const;
146
147 private:
148     struct Node
149     {
150         SizeType firstChild {-1}; // points to first sub-branch or first element ptr index
151         SizeType count      {0};  // -1 for branch or it's a leaf and count means number of elements
152     };
153
154     struct ElementPtr
155     {
156         SizeType element {-1}; // points to item in elements, not sure if even needed, seems to always be aligned anyways
157         SizeType next     {-1}; // points to next elt ptr, or -1 means end of items
158     };
159
160     struct NodeReg
161     {
162         cu::AABBf  aabb;
163         SizeType   index {0};
164         SizeType   depth {0};
165     };
166
167     struct NodeQuery
168     {
169         SizeType index {0};
170     };
171
172     struct NodeRegQuery
173     {
174         cu::AABBf  aabb;
175         SizeType   index {0};
176     };
177

```

```

178 void NodeInsert(const NodeReg& aNode, SizeType aEltIndex);
179 void LeafInsert(const NodeReg& aNode, SizeType aEltIndex);
180
181 auto FindLeaves(const NodeReg& aNode, const cu::AABBf& aAABB) const -> std::vector<NodeReg>;
182
183 auto QFindLeaves(const NodeRegQuery& aNode, const cu::Vector3f& aStartPos, const cu::Vector3f& aEndPos) const -> std::vector<NodeQuery>;
184 auto QFindLeaves(const NodeRegQuery& aNode, const cu::Frustumf& aFrustum) const -> std::vector<NodeQuery>;
185 auto QFindLeavesNoDepth(const NodeRegQuery& aNode, const cu::Frustumf& aFrustum) const -> std::vector<NodeQuery>;
186 auto QFindLeaves(const NodeRegQuery& aNode, const cu::AABBf& aAABB) const -> std::vector<NodeQuery>;
187 auto QFindLeaves(const NodeRegQuery& aNode, const cu::Spheref& aSphere) const -> std::vector<NodeQuery>;
188
189 void GetNodeLeaves(SizeType aNodeIndex, std::vector<NodeQuery>& outLeaves, std::vector<SizeType>& aProcessNodes) const;
190
191 static bool IsLeaf(const Node& aNode);
192 static bool IsBranch(const Node& aNode);
193
194 cu::FreeVector<Element>      myElements;      // all the elements
195 cu::FreeVector<ElementPtr>   myElementsPtr;   // all the element ptrs
196 cu::FreeVector<Node>        myNodes;
197
198 cu::AABBf    myRootAABB;
199
200 SizeType    myMaxElements    {16}; // max elements before subdivision
201 SizeType    myMaxDepth      {8}; // max depth before no more leaves will be created
202
203 mutable std::vector<std::uint8_t> myVisited;
204 mutable std::shared_mutex myMutex;
205 };
206
207 template<std::equality_comparable T>
208 inline Octree<T>::Octree(const cu::AABBf& aRootAABB, int aMaxElements, int aMaxDepth)
209     : myRootAABB(aRootAABB), myMaxElements(aMaxElements), myMaxDepth(aMaxDepth)
210 {
211     myNodes.emplace();
212 }
213
214 template<std::equality_comparable T>
215 inline int Octree<T>::ElementCount() const
216 {
217     return (int)myElements.count();
218 }
219
220 template<std::equality_comparable T>
221 inline const cu::AABBf& Octree<T>::GetRootAABB() const
222 {
223     std::shared_lock lock(myMutex);
224     return myRootAABB;
225 }
226
227 template<std::equality_comparable T>
228 inline void Octree<T>::SetRootAABB(const cu::AABBf& aRootAABB)
229 {
230     std::scoped_lock lock(myMutex);
231
232     if (myRootAABB == aRootAABB)
233         return;
234
235     myRootAABB = aRootAABB;
236
237     if (myElements.empty())

```

```

238         return;
239
240         std::vector<NodeQuery> leaves;
241         std::vector<SizeType> toProcess;
242
243         leaves.reserve(ourChildCount * myMaxDepth / 2);
244         toProcess.reserve(ourChildCount * myMaxDepth / 2);
245
246         GetNodeLeaves(0, leaves, toProcess);
247
248         assert(!leaves.empty() && "You have elements but no leaves?");
249
250         myVisited.resize(myElements.size());
251
252         std::vector<SizeType> elements;
253         elements.reserve(myMaxElements * myMaxDepth * ourChildCount / 2);
254
255         for (const auto& leaf : leaves)
256         {
257             const auto nodeIndex = leaf.index;
258
259             for (auto child = myNodes[nodeIndex].firstChild; child != -1;)
260             {
261                 const auto eltIndex = myElementsPtr[child].element;
262
263                 if (!myVisited[eltIndex])
264                 {
265                     elements.emplace_back(eltIndex);
266                     myVisited[eltIndex] = true;
267                 }
268
269                 child = myElementsPtr[child].next;
270             }
271         }
272
273         for (const auto eltIndex : elements)
274         {
275             myVisited[eltIndex] = false;
276         }
277
278         myElementsPtr.clear();
279         myNodes.clear();
280
281         myNodes.emplace();
282
283         for (auto elt : elements)
284         {
285             NodeInsert({ myRootAABB, 0, 0 }, elt);
286         }
287     }
288
289     template<std::equality_comparable T>
290     template<typename... Args> requires std::constructible_from<T, Args...>
291     inline auto Octree<T>::Insert(const cu::AABBf& aAABB, Args&&... someArgs) -> SizeType
292     {
293         std::scoped_lock lock(myMutex);
294
295         if (!myRootAABB.Overlaps(aAABB)) // dont attempt to add if outside boundary
296             return -1;
297

```

```

298     const auto aIndex = (SizeType)myElements.emplace(aAABB, std::forward<Args>(someArgs)...);
299     NodeInsert({ myRootAABB, 0, 0 }, aIndex);
300
301     return aIndex;
302 }
303
304 template<std::equality_comparable T>
305 inline bool Octree<T>::Erase(SizeType aIndex)
306 {
307     std::scoped_lock lock(myMutex);
308
309     const cu::AABBf& aabb = myElements[aIndex].aabb;
310     const auto leaves = QFindLeaves({ myRootAABB, 0 }, aabb);
311
312     if (leaves.empty())
313         return false;
314
315     for (const auto& leaf : leaves)
316     {
317         Node& node = myNodes[leaf.index];
318
319         auto ptrIndex = node.firstChild;
320         auto prvIdx = -1;
321
322         while (ptrIndex != -1 && myElementsPtr[ptrIndex].element != aIndex)
323         {
324             prvIdx = ptrIndex;
325             ptrIndex = myElementsPtr[ptrIndex].next;
326         }
327
328         if (ptrIndex != -1)
329         {
330             const auto nextIndex = myElementsPtr[ptrIndex].next;
331
332             if (prvIdx == -1)
333             {
334                 node.firstChild = nextIndex;
335             }
336             else
337             {
338                 myElementsPtr[prvIdx].next = nextIndex;
339             }
340
341             myElementsPtr.erase(ptrIndex);
342
343             --node.count;
344
345             assert(node.count >= 0 && "Node cannot have a negative count of elements");
346         }
347     }
348
349     myElements.erase(aIndex);
350
351     return true;
352 }
353
354 template<std::equality_comparable T>
355 template<typename... Args> requires std::constructible_from<T, Args...>
356 inline bool Octree<T>::Update(SizeType aIndex, Args&&... someArgs)
357 {

```

```

358     std::shared_lock lock(myMutex);
359
360     if (aIndex >= myElements.size() || !myElements.valid(aIndex))
361         return false;
362
363     myElements[aIndex].item = T{std::forward<Args>(someArgs)...};
364
365     return true;
366 }
367
368 template<std::equality_comparable T>
369 auto Octree<T>::Get(SizeType aIndex) -> ValueType&
370 {
371     std::shared_lock lock(myMutex);
372     return myElements[aIndex].item;
373 }
374
375 template<std::equality_comparable T>
376 auto Octree<T>::Get(SizeType aIndex) const -> const ValueType&
377 {
378     std::shared_lock lock(myMutex);
379     return myElements[aIndex].item;
380 }
381
382 template<std::equality_comparable T>
383 inline auto Octree<T>::GetAABB(SizeType aIndex) const -> const cu::AABBf&
384 {
385     std::shared_lock lock(myMutex);
386     return myElements[aIndex].aabb;
387 }
388
389 template<std::equality_comparable T>
390 inline void Octree<T>::Query(const cu::Frustumf& aFrustum, std::vector<SizeType>& outResult) const
391 {
392     std::scoped_lock lock(myMutex);
393
394     outResult.clear();
395
396     myVisited.resize(myElements.size());
397
398     for (const auto& leaf : QFindLeaves({ myRootAABB, 0 }, aFrustum))
399     {
400         const auto nodeIndex = leaf.index;
401
402         for (auto child = myNodes[nodeIndex].firstChild; child != -1;)
403         {
404             const auto eltIndex = myElementsPtr[child].element;
405             const auto& elt = myElements[eltIndex];
406
407             if (!myVisited[eltIndex] && aFrustum.IsInside(elt.aabb)) // TODO: if leaf was detected as inside frustum, IsInside step can be skipped
408             {
409                 outResult.emplace_back(eltIndex);
410                 myVisited[eltIndex] = true;
411             }
412
413             child = myElementsPtr[child].next;
414         }
415     }
416
417     for (const auto eltIndex : outResult)

```

```

418     {
419         myVisited[eltIndex] = false;
420     }
421 }
422
423 template<std::equality_comparable T>
424 inline void Octree<T>::QueryNoDepth(const cu::Frustumf& aFrustum, std::vector<SizeType>& outResult) const
425 {
426     std::scoped_lock lock(myMutex);
427
428     outResult.clear();
429
430     myVisited.resize(myElements.size());
431
432     for (const auto& leaf : QFindLeavesNoDepth({ myRootAABB, 0 }, aFrustum))
433     {
434         const auto nodeIndex = leaf.index;
435
436         for (auto child = myNodes[nodeIndex].firstChild; child != -1;)
437         {
438             const auto eltIndex = myElementsPtr[child].element;
439             const auto& elt = myElements[eltIndex];
440
441             if (!myVisited[eltIndex] && aFrustum.IsInsideNoDepth(elt.aabb))
442             {
443                 outResult.emplace_back(eltIndex);
444                 myVisited[eltIndex] = true;
445             }
446
447             child = myElementsPtr[child].next;
448         }
449     }
450
451     for (const auto eltIndex : outResult)
452     {
453         myVisited[eltIndex] = false;
454     }
455 }
456
457 template<std::equality_comparable T>
458 inline void Octree<T>::Query(const cu::Vector3f& aStartPos, const cu::Vector3f& aEndPos, std::vector<SizeType>& outResult) const
459 {
460     std::scoped_lock lock(myMutex);
461
462     outResult.clear();
463
464     myVisited.resize(myElements.size());
465
466     for (const auto& leaf : QFindLeaves({ myRootAABB, 0 }, aStartPos, aEndPos))
467     {
468         const auto nodeIndex = leaf.index;
469
470         for (auto child = myNodes[nodeIndex].firstChild; child != -1;)
471         {
472             const auto eltIndex = myElementsPtr[child].element;
473             const auto& elt      = myElements[eltIndex];
474
475             if (!myVisited[eltIndex] && cu::IntersectionAABBSegment(elt.aabb, aStartPos, aEndPos))
476             {
477                 outResult.emplace_back(eltIndex);

```

```

478         myVisited[eltIndex] = true;
479     }
480
481     child = myElementsPtr[child].next;
482 }
483 }
484
485 for (const auto eltIndex : outResult)
486 {
487     myVisited[eltIndex] = false;
488 }
489 }
490
491 template<std::equality_comparable T>
492 inline void Octree<T>::Query(const cu::AABBf& aAABB, std::vector<SizeType>& outResult) const
493 {
494     std::scoped_lock lock(myMutex);
495
496     outResult.clear();
497
498     myVisited.resize(myElements.size());
499
500     for (const auto& leaf : QFindLeaves({ myRootAABB, 0 }, aAABB))
501     {
502         const auto nodeIndex = leaf.index;
503
504         for (auto child = myNodes[nodeIndex].firstChild; child != -1;)
505         {
506             const auto eltIndex = myElementsPtr[child].element;
507             const auto& elt      = myElements[eltIndex];
508
509             if (!myVisited[eltIndex] && elt.aabb.Overlaps(aAABB))
510             {
511                 outResult.emplace_back(eltIndex);
512                 myVisited[eltIndex] = true;
513             }
514
515             child = myElementsPtr[child].next;
516         }
517     }
518
519     for (const auto eltIndex : outResult)
520     {
521         myVisited[eltIndex] = false;
522     }
523 }
524
525 template<std::equality_comparable T>
526 inline void Octree<T>::Query(const cu::Spheref& aSphere, std::vector<SizeType>& outResult) const
527 {
528     std::scoped_lock lock(myMutex);
529
530     outResult.clear();
531
532     myVisited.resize(myElements.size());
533
534     for (const auto& leaf : QFindLeaves({ myRootAABB, 0 }, aSphere))
535     {
536         const auto nodeIndex = leaf.index;
537

```

```

538     for (auto child = myNodes[nodeIndex].firstChild; child != -1;)
539     {
540         const auto eltIndex = myElementsPtr[child].element;
541         const auto& elt      = myElements[eltIndex];
542
543         if (!myVisited[eltIndex] && cu::IntersectionSphereAABB(aSphere, elt.aabb))
544         {
545             outResult.emplace_back(eltIndex);
546             myVisited[eltIndex] = true;
547         }
548
549         child = myElementsPtr[child].next;
550     }
551 }
552
553 for (const auto eltIndex : outResult)
554 {
555     myVisited[eltIndex] = false;
556 }
557 }
558
559 template<std::equality_comparable T>
560 inline void Octree<T>::Query(const cu::Vector3f& aPoint, std::vector<SizeType>& outResult) const
561 {
562     Query(cu::AABBf(aPoint, aPoint), outResult);
563 }
564
565 template<std::equality_comparable T>
566 inline void Octree<T>::Cleanup()
567 {
568     std::scoped_lock lock(myMutex);
569
570     if (myNodes.empty())
571         return;
572
573     std::vector<SizeType> toProcess;
574     toProcess.reserve(ourChildCount * myMaxDepth / 2);
575
576     if (IsBranch(myNodes[0]))
577     {
578         toProcess.emplace_back(0); // push root
579     }
580
581     while (!toProcess.empty())
582     {
583         Node& node = myNodes[toProcess.back()];
584         toProcess.pop_back();
585
586         int numEmpty = 0;
587         for (int i = 0; i < ourChildCount; ++i)
588         {
589             const int childIndex = node.firstChild + i;
590             const Node& child     = myNodes[childIndex];
591
592             if (IsBranch(child))
593             {
594                 toProcess.emplace_back(childIndex);
595             }
596             else if (child.count == 0)
597             {

```

```

598         ++numEmpty;
599     }
600 }
601
602     if (numEmpty == ourChildCount)
603     {
604         for (int i = ourChildCount - 1; i >= 0; --i)
605         {
606             myNodes.erase(node.firstChild + i);
607         }
608
609         node = {}; // reset
610     }
611 }
612 }
613
614 template<std::equality_comparable T>
615 inline void Octree<T>::Clear()
616 {
617     std::scoped_lock lock(myMutex);
618
619     myElements.clear();
620     myElementsPtr.clear();
621     myNodes.clear();
622 }
623
624 template<std::equality_comparable T>
625 inline std::vector<cu::AABBf> Octree<T>::GetBranchAABBs() const
626 {
627     std::shared_lock lock(myMutex);
628
629     std::vector<cu::AABBf> result;
630
631     if (myNodes.empty())
632         return result;
633
634     std::vector<NodeRegQuery> toProcess;
635     toProcess.reserve(ourChildCount * myMaxDepth / 2);
636
637     toProcess.emplace_back(myRootAABB, 0);
638
639     while (!toProcess.empty())
640     {
641         const auto nd = toProcess.back();
642         toProcess.pop_back();
643
644         result.emplace_back(nd.aabb);
645
646         if (IsBranch(myNodes[nd.index]))
647         {
648             const auto fc = myNodes[nd.index].firstChild;
649
650             const auto c    = nd.aabb.GetCenter();
651             const auto hs    = nd.aabb.GetExtends();
652             const auto l     = nd.aabb.GetMin().x;
653             const auto b     = nd.aabb.GetMin().y;
654             const auto h     = nd.aabb.GetMin().z;
655             const auto r     = l + hs.x;
656             const auto t     = b + hs.y;
657             const auto f     = h + hs.z;

```

```

658
659     auto b1 = cu::AABBf(r, t, f, r + hs.x, t + hs.y, f + hs.z);
660     auto b2 = cu::AABBf(l, t, f, l + hs.x, t + hs.y, f + hs.z);
661     auto b3 = cu::AABBf(r, b, f, r + hs.x, b + hs.y, f + hs.z);
662     auto b4 = cu::AABBf(l, b, f, l + hs.x, b + hs.y, f + hs.z);
663     auto b5 = cu::AABBf(r, t, h, r + hs.x, t + hs.y, h + hs.z);
664     auto b6 = cu::AABBf(l, t, h, l + hs.x, t + hs.y, h + hs.z);
665     auto b7 = cu::AABBf(r, b, h, r + hs.x, b + hs.y, h + hs.z);
666     auto b8 = cu::AABBf(l, b, h, l + hs.x, b + hs.y, h + hs.z);
667
668     toProcess.emplace_back(b1, fc + 0);
669     toProcess.emplace_back(b2, fc + 1);
670     toProcess.emplace_back(b3, fc + 2);
671     toProcess.emplace_back(b4, fc + 3);
672     toProcess.emplace_back(b5, fc + 4);
673     toProcess.emplace_back(b6, fc + 5);
674     toProcess.emplace_back(b7, fc + 6);
675     toProcess.emplace_back(b8, fc + 7);
676 }
677 }
678
679 return result;
680 }
681
682 template<std::equality_comparable T>
683 inline void Octree<T>::NodeInsert(const NodeReg& aNode, SizeType aEltIndex)
684 {
685     const cu::AABBf& aabb = myElements[aEltIndex].aabb;
686     auto leaves = FindLeaves(aNode, aabb);
687
688     for (const auto& leaf : leaves)
689     {
690         LeafInsert(leaf, aEltIndex);
691     }
692 }
693
694 template<std::equality_comparable T>
695 inline void Octree<T>::LeafInsert(const NodeReg& aNode, SizeType aEltIndex)
696 {
697     Node* node = &myNodes[aNode.index];
698
699     node->firstChild = (SizeType)myElementsPtr.emplace(aEltIndex, node->firstChild);
700
701     if (node->count == myMaxElements && aNode.depth < myMaxDepth && aNode.aabb.Contains(myElements[aEltIndex].aabb)) // nodes that are smaller than the elements aabb cannot get divided
702     {
703         std::vector<SizeType> elements;
704         elements.reserve(myMaxElements);
705
706         while (node->firstChild != -1)
707         {
708             const auto index = node->firstChild;
709
710             const auto next = myElementsPtr[index].next;
711             const auto elt = myElementsPtr[index].element;
712
713             node->firstChild = next;
714             myElementsPtr.erase(index);
715
716             elements.emplace_back(elt);
717         }

```

```

718
719     const auto fc = myNodes.emplace();
720     for (int i = 0; i < ourChildCount - 1; ++i)
721     {
722         myNodes.emplace();
723     }
724
725     node = &myNodes[aNode.index];
726
727     node->firstChild    = (SizeType)fc;
728     node->count         = -1; // set as branch
729
730     for (const auto elt : elements)
731     {
732         NodeInsert(aNode, elt);
733     }
734 }
735 else
736     ++node->count;
737 }
738
739 template<std::equality_comparable T>
740 inline auto Octree<T>::FindLeaves(const NodeReg& aNode, const cu::AABBf& aAABB) const -> std::vector<NodeReg>
741 {
742     std::vector<NodeReg> leaves;
743     std::vector<NodeReg> toProcess;
744
745     leaves.reserve(ourChildCount / 2);
746     toProcess.reserve(ourChildCount * myMaxDepth / 2);
747
748     toProcess.emplace_back(aNode);
749
750     while (!toProcess.empty())
751     {
752         const auto nd = toProcess.back();
753         toProcess.pop_back();
754
755         if (IsLeaf(myNodes[nd.index]))
756         {
757             leaves.emplace_back(nd);
758         }
759         else
760         {
761             const auto fc    = myNodes[nd.index].firstChild;
762
763             const auto c      = nd.aabb.GetCenter();
764             const auto hs     = nd.aabb.GetExtends();
765             const auto l      = nd.aabb.GetMin().x;
766             const auto b      = nd.aabb.GetMin().y;
767             const auto h      = nd.aabb.GetMin().z;
768             const auto r      = l + hs.x;
769             const auto t      = b + hs.y;
770             const auto f      = h + hs.z;
771
772             if (aAABB.GetMax().z >= c.z)
773             {
774                 if (aAABB.GetMax().y >= c.y)
775                 {
776                     if (aAABB.GetMax().x >= c.x)
777                     {

```

```

778         toProcess.emplace_back(cu::AABBf(r, t, f, r + hs.x, t + hs.y, f + hs.z), fc + 0, nd.depth + 1);
779     }
780     if (aAABB.GetMin().x < c.x)
781     {
782         toProcess.emplace_back(cu::AABBf(1, t, f, 1 + hs.x, t + hs.y, f + hs.z), fc + 1, nd.depth + 1);
783     }
784 }
785 if (aAABB.GetMin().y < c.y)
786 {
787     if (aAABB.GetMax().x >= c.x)
788     {
789         toProcess.emplace_back(cu::AABBf(r, b, f, r + hs.x, b + hs.y, f + hs.z), fc + 2, nd.depth + 1);
790     }
791     if (aAABB.GetMin().x < c.x)
792     {
793         toProcess.emplace_back(cu::AABBf(1, b, f, 1 + hs.x, b + hs.y, f + hs.z), fc + 3, nd.depth + 1);
794     }
795 }
796 }
797 if (aAABB.GetMin().z < c.z)
798 {
799     if (aAABB.GetMax().y >= c.y)
800     {
801         if (aAABB.GetMax().x >= c.x)
802         {
803             toProcess.emplace_back(cu::AABBf(r, t, h, r + hs.x, t + hs.y, h + hs.z), fc + 4, nd.depth + 1);
804         }
805         if (aAABB.GetMin().x < c.x)
806         {
807             toProcess.emplace_back(cu::AABBf(1, t, h, 1 + hs.x, t + hs.y, h + hs.z), fc + 5, nd.depth + 1);
808         }
809     }
810     if (aAABB.GetMin().y < c.y)
811     {
812         if (aAABB.GetMax().x >= c.x)
813         {
814             toProcess.emplace_back(cu::AABBf(r, b, h, r + hs.x, b + hs.y, h + hs.z), fc + 6, nd.depth + 1);
815         }
816         if (aAABB.GetMin().x < c.x)
817         {
818             toProcess.emplace_back(cu::AABBf(1, b, h, 1 + hs.x, b + hs.y, h + hs.z), fc + 7, nd.depth + 1);
819         }
820     }
821 }
822 }
823 }
824
825 return leaves;
826 }
827
828 template<std::equality_comparable T>
829 inline auto Octree<T>::QFindLeaves(const NodeRegQuery& aNode, const cu::Vector3f& aStartPos, const cu::Vector3f& aEndPos) const -> std::vector<NodeQuery>
830 {
831     std::vector<NodeQuery> leaves;
832     std::vector<NodeRegQuery> toProcess;
833
834     leaves.reserve(ourChildCount / 2);
835     toProcess.reserve(ourChildCount * myMaxDepth / 2);
836
837     toProcess.emplace_back(aNode);

```

```

838
839     while (!toProcess.empty())
840     {
841         const auto nd = toProcess.back();
842         toProcess.pop_back();
843
844         if (IsLeaf(myNodes[nd.index]))
845         {
846             leaves.emplace_back(nd.index);
847         }
848         else
849         {
850             const auto fc    = myNodes[nd.index].firstChild;
851
852             const auto c     = nd.aabb.GetCenter();
853             const auto hs    = nd.aabb.GetExtends();
854             const auto l     = nd.aabb.GetMin().x;
855             const auto b     = nd.aabb.GetMin().y;
856             const auto h     = nd.aabb.GetMin().z;
857             const auto r     = l + hs.x;
858             const auto t     = b + hs.y;
859             const auto f     = h + hs.z;
860
861             const auto b1 = cu::AABBf(r, t, f, r + hs.x, t + hs.y, f + hs.z);
862             const auto b2 = cu::AABBf(l, t, f, l + hs.x, t + hs.y, f + hs.z);
863             const auto b3 = cu::AABBf(r, b, f, r + hs.x, b + hs.y, f + hs.z);
864             const auto b4 = cu::AABBf(l, b, f, l + hs.x, b + hs.y, f + hs.z);
865             const auto b5 = cu::AABBf(r, t, h, r + hs.x, t + hs.y, h + hs.z);
866             const auto b6 = cu::AABBf(l, t, h, l + hs.x, t + hs.y, h + hs.z);
867             const auto b7 = cu::AABBf(r, b, h, r + hs.x, b + hs.y, h + hs.z);
868             const auto b8 = cu::AABBf(l, b, h, l + hs.x, b + hs.y, h + hs.z);
869
870             if (cu::IntersectionAABBSegment(b1, aStartPos, aEndPos)) toProcess.emplace_back(b1, fc + 0);
871             if (cu::IntersectionAABBSegment(b2, aStartPos, aEndPos)) toProcess.emplace_back(b2, fc + 1);
872             if (cu::IntersectionAABBSegment(b3, aStartPos, aEndPos)) toProcess.emplace_back(b3, fc + 2);
873             if (cu::IntersectionAABBSegment(b4, aStartPos, aEndPos)) toProcess.emplace_back(b4, fc + 3);
874             if (cu::IntersectionAABBSegment(b5, aStartPos, aEndPos)) toProcess.emplace_back(b5, fc + 4);
875             if (cu::IntersectionAABBSegment(b6, aStartPos, aEndPos)) toProcess.emplace_back(b6, fc + 5);
876             if (cu::IntersectionAABBSegment(b7, aStartPos, aEndPos)) toProcess.emplace_back(b7, fc + 6);
877             if (cu::IntersectionAABBSegment(b8, aStartPos, aEndPos)) toProcess.emplace_back(b8, fc + 7);
878         }
879     }
880
881     return leaves;
882 }
883
884 template<std::equality_comparable T>
885 inline auto Octree<T>::QFindLeaves(const NodeRegQuery& aNode, const cu::Frustumf& aFrustum) const -> std::vector<NodeQuery>
886 {
887     std::vector<NodeQuery>    leaves;
888     std::vector<NodeRegQuery> toProcess;
889     std::vector<SizeType>     toProcessContained;
890
891     leaves.reserve(ourChildCount / 2);
892     toProcess.reserve(ourChildCount * myMaxDepth / 2);
893     toProcessContained.reserve(ourChildCount * myMaxDepth / 4);
894
895     toProcess.emplace_back(aNode);
896
897     while (!toProcess.empty())

```

```

898 {
899     const auto nd = toProcess.back();
900     toProcess.pop_back();
901
902     if (IsLeaf(myNodes[nd.index]))
903     {
904         leaves.emplace_back(nd.index);
905     }
906     else // TODO: optimize further if this becomes a bottleneck
907     {
908         if (aFrustum.Contains(nd.aabb))
909         {
910             GetNodeLeaves(nd.index, leaves, toProcessContained);
911         }
912         else
913         {
914             const auto fc = myNodes[nd.index].firstChild;
915
916             const auto c    = nd.aabb.GetCenter();
917             const auto hs   = nd.aabb.GetExtends();
918             const auto l    = nd.aabb.GetMin().x;
919             const auto b    = nd.aabb.GetMin().y;
920             const auto h    = nd.aabb.GetMin().z;
921             const auto r    = l + hs.x;
922             const auto t    = b + hs.y;
923             const auto f    = h + hs.z;
924
925             const auto b1 = cu::AABBf(r, t, f, r + hs.x, t + hs.y, f + hs.z);
926             const auto b2 = cu::AABBf(l, t, f, l + hs.x, t + hs.y, f + hs.z);
927             const auto b3 = cu::AABBf(r, b, f, r + hs.x, b + hs.y, f + hs.z);
928             const auto b4 = cu::AABBf(l, b, f, l + hs.x, b + hs.y, f + hs.z);
929             const auto b5 = cu::AABBf(r, t, h, r + hs.x, t + hs.y, h + hs.z);
930             const auto b6 = cu::AABBf(l, t, h, l + hs.x, t + hs.y, h + hs.z);
931             const auto b7 = cu::AABBf(r, b, h, r + hs.x, b + hs.y, h + hs.z);
932             const auto b8 = cu::AABBf(l, b, h, l + hs.x, b + hs.y, h + hs.z);
933
934             if (aFrustum.IsInside(b1)) toProcess.emplace_back(b1, fc + 0);
935             if (aFrustum.IsInside(b2)) toProcess.emplace_back(b2, fc + 1);
936             if (aFrustum.IsInside(b3)) toProcess.emplace_back(b3, fc + 2);
937             if (aFrustum.IsInside(b4)) toProcess.emplace_back(b4, fc + 3);
938             if (aFrustum.IsInside(b5)) toProcess.emplace_back(b5, fc + 4);
939             if (aFrustum.IsInside(b6)) toProcess.emplace_back(b6, fc + 5);
940             if (aFrustum.IsInside(b7)) toProcess.emplace_back(b7, fc + 6);
941             if (aFrustum.IsInside(b8)) toProcess.emplace_back(b8, fc + 7);
942         }
943     }
944 }
945
946 return leaves;
947 }
948 template<std::equality_comparable T>
949 inline auto Octree<T>::QFindLeavesNoDepth(const NodeRegQuery& aNode, const cu::Frustumf& aFrustum) const -> std::vector<NodeQuery>
950 {
951     std::vector<NodeQuery>    leaves;
952     std::vector<NodeRegQuery> toProcess;
953     std::vector<SizeType>    toProcessContained;
954
955     leaves.reserve(ourChildCount / 2);
956     toProcess.reserve(ourChildCount * myMaxDepth / 2);
957     toProcessContained.reserve(ourChildCount * myMaxDepth / 4);

```

```

958
959     toProcess.emplace_back(aNode);
960
961     while (!toProcess.empty())
962     {
963         const auto nd = toProcess.back();
964         toProcess.pop_back();
965
966         if (IsLeaf(myNodes[nd.index]))
967         {
968             leaves.emplace_back(nd.index);
969         }
970         else // TODO: optimize further if this becomes a bottleneck
971         {
972             if (aFrustum.Contains(nd.aabb))
973             {
974                 GetNodeLeaves(nd.index, leaves, toProcessContained);
975             }
976             else
977             {
978                 const auto fc = myNodes[nd.index].firstChild;
979
980                 const auto c    = nd.aabb.GetCenter();
981                 const auto hs   = nd.aabb.GetExtends();
982                 const auto l    = nd.aabb.GetMin().x;
983                 const auto b    = nd.aabb.GetMin().y;
984                 const auto h    = nd.aabb.GetMin().z;
985                 const auto r    = l + hs.x;
986                 const auto t    = b + hs.y;
987                 const auto f    = h + hs.z;
988
989                 const auto b1 = cu::AABBf(r, t, f, r + hs.x, t + hs.y, f + hs.z);
990                 const auto b2 = cu::AABBf(l, t, f, l + hs.x, t + hs.y, f + hs.z);
991                 const auto b3 = cu::AABBf(r, b, f, r + hs.x, b + hs.y, f + hs.z);
992                 const auto b4 = cu::AABBf(l, b, f, l + hs.x, b + hs.y, f + hs.z);
993                 const auto b5 = cu::AABBf(r, t, h, r + hs.x, t + hs.y, h + hs.z);
994                 const auto b6 = cu::AABBf(l, t, h, l + hs.x, t + hs.y, h + hs.z);
995                 const auto b7 = cu::AABBf(r, b, h, r + hs.x, b + hs.y, h + hs.z);
996                 const auto b8 = cu::AABBf(l, b, h, l + hs.x, b + hs.y, h + hs.z);
997
998                 if (aFrustum.IsInsideNoDepth(b1)) toProcess.emplace_back(b1, fc + 0);
999                 if (aFrustum.IsInsideNoDepth(b2)) toProcess.emplace_back(b2, fc + 1);
1000                 if (aFrustum.IsInsideNoDepth(b3)) toProcess.emplace_back(b3, fc + 2);
1001                 if (aFrustum.IsInsideNoDepth(b4)) toProcess.emplace_back(b4, fc + 3);
1002                 if (aFrustum.IsInsideNoDepth(b5)) toProcess.emplace_back(b5, fc + 4);
1003                 if (aFrustum.IsInsideNoDepth(b6)) toProcess.emplace_back(b6, fc + 5);
1004                 if (aFrustum.IsInsideNoDepth(b7)) toProcess.emplace_back(b7, fc + 6);
1005                 if (aFrustum.IsInsideNoDepth(b8)) toProcess.emplace_back(b8, fc + 7);
1006             }
1007         }
1008     }
1009
1010     return leaves;
1011 }
1012 template<std::equality_comparable T>
1013 inline auto Octree<T>::QFindLeaves(const NodeRegQuery& aNode, const cu::AABBf& aAABB) const -> std::vector<NodeQuery>
1014 {
1015     std::vector<NodeQuery> leaves;
1016     std::vector<NodeRegQuery> toProcess;
1017

```

```

1018 leaves.reserve(ourChildCount / 2);
1019 toProcess.reserve(ourChildCount * myMaxDepth / 2);
1020
1021 toProcess.emplace_back(aNode);
1022
1023 while (!toProcess.empty())
1024 {
1025     const auto nd = toProcess.back();
1026     toProcess.pop_back();
1027
1028     if (IsLeaf(myNodes[nd.index]))
1029     {
1030         leaves.emplace_back(nd.index);
1031     }
1032     else
1033     {
1034         const auto fc    = myNodes[nd.index].firstChild;
1035
1036         const auto c      = nd.aabb.GetCenter();
1037         const auto hs      = nd.aabb.GetExtends();
1038         const auto l       = nd.aabb.GetMin().x;
1039         const auto b       = nd.aabb.GetMin().y;
1040         const auto h       = nd.aabb.GetMin().z;
1041         const auto r       = l + hs.x;
1042         const auto t       = b + hs.y;
1043         const auto f       = h + hs.z;
1044
1045         if (aAABB.GetMax().z >= c.z)
1046         {
1047             if (aAABB.GetMax().y >= c.y)
1048             {
1049                 if (aAABB.GetMax().x >= c.x)
1050                 {
1051                     toProcess.emplace_back(cu::AABBf(r, t, f, r + hs.x, t + hs.y, f + hs.z), fc + 0);
1052                 }
1053                 if (aAABB.GetMin().x < c.x)
1054                 {
1055                     toProcess.emplace_back(cu::AABBf(l, t, f, l + hs.x, t + hs.y, f + hs.z), fc + 1);
1056                 }
1057             }
1058             if (aAABB.GetMin().y < c.y)
1059             {
1060                 if (aAABB.GetMax().x >= c.x)
1061                 {
1062                     toProcess.emplace_back(cu::AABBf(r, b, f, r + hs.x, b + hs.y, f + hs.z), fc + 2);
1063                 }
1064                 if (aAABB.GetMin().x < c.x)
1065                 {
1066                     toProcess.emplace_back(cu::AABBf(l, b, f, l + hs.x, b + hs.y, f + hs.z), fc + 3);
1067                 }
1068             }
1069         }
1070         if (aAABB.GetMin().z < c.z)
1071         {
1072             if (aAABB.GetMax().y >= c.y)
1073             {
1074                 if (aAABB.GetMax().x >= c.x)
1075                 {
1076                     toProcess.emplace_back(cu::AABBf(r, t, h, r + hs.x, t + hs.y, h + hs.z), fc + 4);
1077                 }

```

```

1078         if (aAABB.GetMin().x < c.x)
1079         {
1080             toProcess.emplace_back(cu::AABBf(1, t, h, 1 + hs.x, t + hs.y, h + hs.z), fc + 5);
1081         }
1082     }
1083     if (aAABB.GetMin().y < c.y)
1084     {
1085         if (aAABB.GetMax().x >= c.x)
1086         {
1087             toProcess.emplace_back(cu::AABBf(r, b, h, r + hs.x, b + hs.y, h + hs.z), fc + 6);
1088         }
1089         if (aAABB.GetMin().x < c.x)
1090         {
1091             toProcess.emplace_back(cu::AABBf(1, b, h, 1 + hs.x, b + hs.y, h + hs.z), fc + 7);
1092         }
1093     }
1094 }
1095 }
1096 }
1097
1098 return leaves;
1099 }
1100
1101 template<std::equality_comparable T>
1102 inline auto Octree<T>::QFindLeaves(const NodeRegQuery& aNode, const cu::Spheref& aSphere) const -> std::vector<NodeQuery>
1103 {
1104     std::vector<NodeQuery> leaves;
1105     std::vector<NodeRegQuery> toProcess;
1106
1107     leaves.reserve(ourChildCount / 2);
1108     toProcess.reserve(ourChildCount * myMaxDepth / 2);
1109
1110     toProcess.emplace_back(aNode);
1111
1112     while (!toProcess.empty())
1113     {
1114         const auto nd = toProcess.back();
1115         toProcess.pop_back();
1116
1117         if (IsLeaf(myNodes[nd.index]))
1118         {
1119             leaves.emplace_back(nd.index);
1120         }
1121         else
1122         {
1123             const auto fc = myNodes[nd.index].firstChild;
1124
1125             const auto c    = nd.aabb.GetCenter();
1126             const auto hs    = nd.aabb.GetExtends();
1127             const auto l    = nd.aabb.GetMin().x;
1128             const auto b    = nd.aabb.GetMin().y;
1129             const auto h    = nd.aabb.GetMin().z;
1130             const auto r    = l + hs.x;
1131             const auto t    = b + hs.y;
1132             const auto f    = h + hs.z;
1133
1134             const auto b1 = cu::AABBf(r, t, f, r + hs.x, t + hs.y, f + hs.z);
1135             const auto b2 = cu::AABBf(1, t, f, 1 + hs.x, t + hs.y, f + hs.z);
1136             const auto b3 = cu::AABBf(r, b, f, r + hs.x, b + hs.y, f + hs.z);
1137             const auto b4 = cu::AABBf(1, b, f, 1 + hs.x, b + hs.y, f + hs.z);

```

```

1138         const auto b5 = cu::AABBf(r, t, h, r + hs.x, t + hs.y, h + hs.z);
1139         const auto b6 = cu::AABBf(l, t, h, l + hs.x, t + hs.y, h + hs.z);
1140         const auto b7 = cu::AABBf(r, b, h, r + hs.x, b + hs.y, h + hs.z);
1141         const auto b8 = cu::AABBf(l, b, h, l + hs.x, b + hs.y, h + hs.z);
1142
1143         if (cu::IntersectionSphereAABB(aSphere, b1)) toProcess.emplace_back(b1, fc + 0);
1144         if (cu::IntersectionSphereAABB(aSphere, b2)) toProcess.emplace_back(b2, fc + 1);
1145         if (cu::IntersectionSphereAABB(aSphere, b3)) toProcess.emplace_back(b3, fc + 2);
1146         if (cu::IntersectionSphereAABB(aSphere, b4)) toProcess.emplace_back(b4, fc + 3);
1147         if (cu::IntersectionSphereAABB(aSphere, b5)) toProcess.emplace_back(b5, fc + 4);
1148         if (cu::IntersectionSphereAABB(aSphere, b6)) toProcess.emplace_back(b6, fc + 5);
1149         if (cu::IntersectionSphereAABB(aSphere, b7)) toProcess.emplace_back(b7, fc + 6);
1150         if (cu::IntersectionSphereAABB(aSphere, b8)) toProcess.emplace_back(b8, fc + 7);
1151     }
1152 }
1153
1154 return leaves;
1155 }
1156
1157 template<std::equality_comparable T>
1158 inline void Octree<T>::GetNodeLeaves(SizeType aNodeIndex, std::vector<NodeQuery>& outLeaves, std::vector<SizeType>& aProcessNodes) const
1159 {
1160     aProcessNodes.clear();
1161
1162     aProcessNodes.emplace_back(aNodeIndex);
1163
1164     while (!aProcessNodes.empty())
1165     {
1166         const auto index = aProcessNodes.back();
1167         aProcessNodes.pop_back();
1168
1169         if (IsLeaf(myNodes[index]))
1170         {
1171             outLeaves.emplace_back(index);
1172         }
1173         else
1174         {
1175             const auto fc = myNodes[index].firstChild;
1176
1177             aProcessNodes.emplace_back(fc + 0);
1178             aProcessNodes.emplace_back(fc + 1);
1179             aProcessNodes.emplace_back(fc + 2);
1180             aProcessNodes.emplace_back(fc + 3);
1181             aProcessNodes.emplace_back(fc + 4);
1182             aProcessNodes.emplace_back(fc + 5);
1183             aProcessNodes.emplace_back(fc + 6);
1184             aProcessNodes.emplace_back(fc + 7);
1185         }
1186     }
1187 }
1188
1189 template<std::equality_comparable T>
1190 inline bool Octree<T>::IsLeaf(const Node& aNode)
1191 {
1192     return aNode.count != -1;
1193 }
1194 template<std::equality_comparable T>
1195 inline bool Octree<T>::IsBranch(const Node& aNode)
1196 {
1197     return aNode.count == -1;

```

