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

```cpp
#pragma once

#include <vector>
#include <shared_mutex>
#include <concepts>

#include <CommonUtilities/Math/AABB.hpp>
#include <CommonUtilities/Math/Vector3.hpp>
#include <CommonUtilities/Math/Frustum.hpp>
#include <CommonUtilities/Math/Intersection.hpp>
#include <CommonUtilities/Math/Sphere.hpp>

#include <CommonUtilities/Structures/FreeVector.hpp>

/// Octree based upon: https://stackoverflow.com/questions/41946007/efficient-and-well-explained-implementation-of-a-quadtree-for-2d-collision-det
///
template<std::equality_comparable T>
class Octree
{
public:
    using ElementType   = T;
    using ValueType     = std::remove_const_t<T>;
    using SizeType      = int;

    static constexpr SizeType ourChildCount = 8;

    struct Element
    {
        cu::AABBf   aabb;   // aabb encompassing the item
        ValueType   item;
    };

    Octree() = default;

    Octree(const cu::AABBf& aRootAABB, int aMaxElements = 16, int aMaxDepth = 16);

    int ElementCount() const;

    const cu::AABBf& GetRootAABB() const;

    void SetRootAABB(const cu::AABBf& aRootAABB);

    /// Inserts given element into the quadtree.
    ///
    /// \param AABB: Rectangle encompassing item
    /// \param Args: Constructor parameters for item
    ///
    /// \returns Index to element, can be used to directly access it when, e.g., erasing it from the tree
    ///
    template<typename... Args> requires std::constructible_from<T, Args...>
    auto Insert(const cu::AABBf& aAABB, Args&&... someArgs) -> SizeType;

    /// Attempts to erase element from tree.
    ///
    /// \param Index: Index to element to erase
    ///
    /// \returns True if successfully removed the element, otherwise false
```

```
    ///
    bool Erase(SizeType aIndex);

    /// Updates the given element with new data.
    ///
    /// \param Index: index to element
    /// \param Args: Data to update the current element
    ///
    /// \returns True if successfully updated the element, otherwise false
    ///
    template<typename... Args> requires std::constructible_from<T, Args...>
    bool Update(SizeType aIndex, Args&&... someArgs);

    /// Retrieves an element.
    ///
    /// \param Index: index to element
    ///
    NODISC auto Get(SizeType aIndex) -> ValueType&;

    /// Retrieves an element.
    ///
    /// \param Index: index to element
    ///
    NODISC auto Get(SizeType aIndex) const -> const ValueType&;

    /// Retrieves the aabb encompassing item
    ///
    /// \param Index: index to item.
    ///
    NODISC auto GetAABB(SizeType aIndex) const -> const cu::AABBf&;

    /// Queries the tree for elements.
    ///
    /// \param Frustum: Frustum to search for overlapping elements
    ///
    /// \returns List of entities intersecting the frustum
    ///
    NODISC void Query(const cu::Frustumf& aFrustum, std::vector<SizeType>& outResult) const;

    /// Queries the tree for elements.
    ///
    /// \param Frustum: Frustum to search for overlapping elements
    ///
    /// \returns List of entities intersecting the frustum
    ///
    NODISC void QueryNoDepth(const cu::Frustumf& aFrustum, std::vector<SizeType>& outResult) const;

    /// Queries the tree for elements.
    ///
    /// \param Frustum: Frustum to search for overlapping elements
    ///
    /// \returns List of entities intersecting the frustum
    ///
    NODISC void Query(const cu::Vector3f& aStartPos, const cu::Vector3f& aEndPos, std::vector<SizeType>& outResult) const;

    /// Queries the tree for elements.
    ///
    /// \param AABB: Bounding box to search for overlapping elements
    ///
    /// \returns List of entities intersecting the aabb
```

```
    ///
    NODISC void Query(const cu::AABBf& aAABB, std::vector<SizeType>& outResult) const;

    /// Queries the tree for elements.
    ///
    /// \param Point: Point to search for overlapping elements
    ///
    /// \returns List of entities intersecting the point
    ///
    NODISC void Query(const cu::Spheref& aSphere, std::vector<SizeType>& outResult) const;

    /// Queries the tree for elements.
    ///
    /// \param Point: Point to search for overlapping elements
    ///
    /// \returns List of entities intersecting the point
    ///
    NODISC void Query(const cu::Vector3f& aPoint, std::vector<SizeType>& outResult) const;

    /// Performs a lazy cleanup of the tree, should be called after items have been erased.
    ///
    void Cleanup();

    /// Clears the tree
    ///
    void Clear();

    std::vector<cu::AABBf> GetBranchAABBs() const;

private:
    struct Node
    {
        SizeType firstChild  {-1};  // points to first sub-branch or first element ptr index
        SizeType count       {0};   // -1 for branch or it's a leaf and count means number of elements
    };

    struct ElementPtr
    {
        SizeType element {-1};  // points to item in elements, not sure if even needed, seems to always be aligned anyways
        SizeType next    {-1};  // points to next elt ptr, or -1 means end of items
    };

    struct NodeReg
    {
        cu::AABBf   aabb;
        SizeType    index {0};
        SizeType    depth {0};
    };

    struct NodeQuery
    {
        SizeType index {0};
    };

    struct NodeRegQuery
    {
        cu::AABBf   aabb;
        SizeType    index {0};
    };

```

```
    void NodeInsert(const NodeReg& aNode, SizeType aEltIndex);
    void LeafInsert(const NodeReg& aNode, SizeType aEltIndex);

    auto FindLeaves(const NodeReg& aNode, const cu::AABBf& aAABB) const -> std::vector<NodeReg>;

    auto QFindLeaves(const NodeRegQuery& aNode, const cu::Vector3f& aStartPos, const cu::Vector3f& aEndPos) const -> std::vector<NodeQuery>;
    auto QFindLeaves(const NodeRegQuery& aNode, const cu::Frustumf& aFrustum) const -> std::vector<NodeQuery>;
    auto QFindLeavesNoDepth(const NodeRegQuery& aNode, const cu::Frustumf& aFrustum) const -> std::vector<NodeQuery>;
    auto QFindLeaves(const NodeRegQuery& aNode, const cu::AABBf& aAABB) const -> std::vector<NodeQuery>;
    auto QFindLeaves(const NodeRegQuery& aNode, const cu::Spheref& aSphere) const -> std::vector<NodeQuery>;

    void GetNodeLeaves(SizeType aNodeIndex, std::vector<NodeQuery>& outLeaves, std::vector<SizeType>& aProcessNodes) const;

    static bool IsLeaf(const Node& aNode);
    static bool IsBranch(const Node& aNode);

    cu::FreeVector<Element>     myElements;     // all the elements
    cu::FreeVector<ElementPtr>  myElementsPtr;  // all the element ptrs
    cu::FreeVector<Node>        myNodes;

    cu::AABBf   myRootAABB;

    SizeType    myMaxElements   {16}; // max elements before subdivision
    SizeType    myMaxDepth      {8}; // max depth before no more leaves will be created

    mutable std::vector<std::uint8_t> myVisited;
    mutable std::shared_mutex myMutex;
};

template<std::equality_comparable T>
inline Octree<T>::Octree(const cu::AABBf& aRootAABB, int aMaxElements, int aMaxDepth)
    : myRootAABB(aRootAABB), myMaxElements(aMaxElements), myMaxDepth(aMaxDepth)
{
    myNodes.emplace();
}

template<std::equality_comparable T>
inline int Octree<T>::ElementCount() const
{
    return (int)myElements.count();
}

template<std::equality_comparable T>
inline const cu::AABBf& Octree<T>::GetRootAABB() const
{
    std::shared_lock lock(myMutex);
    return myRootAABB;
}

template<std::equality_comparable T>
inline void Octree<T>::SetRootAABB(const cu::AABBf& aRootAABB)
{
    std::scoped_lock lock(myMutex);

    if (myRootAABB == aRootAABB)
        return;

    myRootAABB = aRootAABB;

    if (myElements.empty())
```

```
        return;

    std::vector<NodeQuery> leaves;
    std::vector<SizeType> toProcess;

    leaves.reserve(ourChildCount * myMaxDepth / 2);
    toProcess.reserve(ourChildCount * myMaxDepth / 2);

    GetNodeLeaves(0, leaves, toProcess);

    assert(!leaves.empty() && "You have elements but no leaves?");

    myVisited.resize(myElements.size());

    std::vector<SizeType> elements;
    elements.reserve(myMaxElements * myMaxDepth * ourChildCount / 2);

    for (const auto& leaf : leaves)
    {
        const auto nodeIndex = leaf.index;

        for (auto child = myNodes[nodeIndex].firstChild; child != -1;)
        {
            const auto eltIndex = myElementsPtr[child].element;

            if (!myVisited[eltIndex])
            {
                elements.emplace_back(eltIndex);
                myVisited[eltIndex] = true;
            }

            child = myElementsPtr[child].next;
        }
    }

    for (const auto eltIndex : elements)
    {
        myVisited[eltIndex] = false;
    }

    myElementsPtr.clear();
    myNodes.clear();

    myNodes.emplace();

    for (auto elt : elements)
    {
        NodeInsert({ myRootAABB, 0, 0 }, elt);
    }
}

template<std::equality_comparable T>
template<typename... Args> requires std::constructible_from<T, Args...>
inline auto Octree<T>::Insert(const cu::AABBf& aAABB, Args&&... someArgs) -> SizeType
{
    std::scoped_lock lock(myMutex);

    if (!myRootAABB.Overlaps(aAABB)) // dont attempt to add if outside boundary
        return -1;

```

```
    const auto aIndex = (SizeType)myElements.emplace(aAABB, std::forward<Args>(someArgs)...);
    NodeInsert({ myRootAABB, 0, 0 }, aIndex);

    return aIndex;
}

template<std::equality_comparable T>
inline bool Octree<T>::Erase(SizeType aIndex)
{
    std::scoped_lock lock(myMutex);

    const cu::AABBf& aabb   = myElements[aIndex].aabb;
    const auto leaves       = QFindLeaves({ myRootAABB, 0 }, aabb);

    if (leaves.empty())
        return false;

    for (const auto& leaf : leaves)
    {
        Node& node = myNodes[leaf.index];

        auto ptrIndex   = node.firstChild;
        auto prvIdx     = -1;

        while (ptrIndex != -1 && myElementsPtr[ptrIndex].element != aIndex)
        {
            prvIdx      = ptrIndex;
            ptrIndex    = myElementsPtr[ptrIndex].next;
        }

        if (ptrIndex != -1)
        {
            const auto nextIndex = myElementsPtr[ptrIndex].next;

            if (prvIdx == -1)
            {
                node.firstChild = nextIndex;
            }
            else
            {
                myElementsPtr[prvIdx].next = nextIndex;
            }

            myElementsPtr.erase(ptrIndex);

            --node.count;

            assert(node.count >= 0 && "Node cannot have a negative count of elements");
        }
    }

    myElements.erase(aIndex);

    return true;
}

template<std::equality_comparable T>
template<typename... Args> requires std::constructible_from<T, Args...>
inline bool Octree<T>::Update(SizeType aIndex, Args&&... someArgs)
{
```

```
    std::shared_lock lock(myMutex);

    if (aIndex >= myElements.size() || !myElements.valid(aIndex))
        return false;

    myElements[aIndex].item = T{std::forward<Args>(someArgs)...};

    return true;
}

template<std::equality_comparable T>
auto Octree<T>::Get(SizeType aIndex) -> ValueType&
{
    std::shared_lock lock(myMutex);
    return myElements[aIndex].item;
}

template<std::equality_comparable T>
auto Octree<T>::Get(SizeType aIndex) const -> const ValueType&
{
    std::shared_lock lock(myMutex);
    return myElements[aIndex].item;
}

template<std::equality_comparable T>
inline auto Octree<T>::GetAABB(SizeType aIndex) const -> const cu::AABBf&
{
    std::shared_lock lock(myMutex);
    return myElements[aIndex].aabb;
}

template<std::equality_comparable T>
inline void Octree<T>::Query(const cu::Frustumf& aFrustum, std::vector<SizeType>& outResult) const
{
    std::scoped_lock lock(myMutex);

    outResult.clear();

    myVisited.resize(myElements.size());

    for (const auto& leaf : QFindLeaves({ myRootAABB, 0 }, aFrustum))
    {
        const auto nodeIndex = leaf.index;

        for (auto child = myNodes[nodeIndex].firstChild; child != -1;)
        {
            const auto eltIndex = myElementsPtr[child].element;
            const auto& elt = myElements[eltIndex];

            if (!myVisited[eltIndex] && aFrustum.IsInside(elt.aabb)) // TODO: if leaf was detected as inside frustum, IsInside step can be skipped
            {
                outResult.emplace_back(eltIndex);
                myVisited[eltIndex] = true;
            }

            child = myElementsPtr[child].next;
        }
    }

    for (const auto eltIndex : outResult)
```

```
    {
        myVisited[eltIndex] = false;
    }
}

template<std::equality_comparable T>
inline void Octree<T>::QueryNoDepth(const cu::Frustumf& aFrustum, std::vector<SizeType>& outResult) const
{
    std::scoped_lock lock(myMutex);

    outResult.clear();

    myVisited.resize(myElements.size());

    for (const auto& leaf : QFindLeavesNoDepth({ myRootAABB, 0 }, aFrustum))
    {
        const auto nodeIndex = leaf.index;

        for (auto child = myNodes[nodeIndex].firstChild; child != -1;)
        {
            const auto eltIndex = myElementsPtr[child].element;
            const auto& elt = myElements[eltIndex];

            if (!myVisited[eltIndex] && aFrustum.IsInsideNoDepth(elt.aabb))
            {
                outResult.emplace_back(eltIndex);
                myVisited[eltIndex] = true;
            }

            child = myElementsPtr[child].next;
        }
    }

    for (const auto eltIndex : outResult)
    {
        myVisited[eltIndex] = false;
    }
}

template<std::equality_comparable T>
inline void Octree<T>::Query(const cu::Vector3f& aStartPos, const cu::Vector3f& aEndPos, std::vector<SizeType>& outResult) const
{
    std::scoped_lock lock(myMutex);

    outResult.clear();

    myVisited.resize(myElements.size());

    for (const auto& leaf : QFindLeaves({ myRootAABB, 0 }, aStartPos, aEndPos))
    {
        const auto nodeIndex = leaf.index;

        for (auto child = myNodes[nodeIndex].firstChild; child != -1;)
        {
            const auto eltIndex = myElementsPtr[child].element;
            const auto& elt     = myElements[eltIndex];

            if (!myVisited[eltIndex] && cu::IntersectionAABBSegment(elt.aabb, aStartPos, aEndPos))
            {
                outResult.emplace_back(eltIndex);
```

```
                myVisited[eltIndex] = true;
            }

            child = myElementsPtr[child].next;
        }
    }

    for (const auto eltIndex : outResult)
    {
        myVisited[eltIndex] = false;
    }
}

template<std::equality_comparable T>
inline void Octree<T>::Query(const cu::AABBf& aAABB, std::vector<SizeType>& outResult) const
{
    std::scoped_lock lock(myMutex);

    outResult.clear();

    myVisited.resize(myElements.size());

    for (const auto& leaf : QFindLeaves({ myRootAABB, 0 }, aAABB))
    {
        const auto nodeIndex = leaf.index;

        for (auto child = myNodes[nodeIndex].firstChild; child != -1;)
        {
            const auto eltIndex = myElementsPtr[child].element;
            const auto& elt     = myElements[eltIndex];

            if (!myVisited[eltIndex] && elt.aabb.Overlaps(aAABB))
            {
                outResult.emplace_back(eltIndex);
                myVisited[eltIndex] = true;
            }

            child = myElementsPtr[child].next;
        }
    }

    for (const auto eltIndex : outResult)
    {
        myVisited[eltIndex] = false;
    }
}

template<std::equality_comparable T>
inline void Octree<T>::Query(const cu::Sphere& aSphere, std::vector<SizeType>& outResult) const
{
    std::scoped_lock lock(myMutex);

    outResult.clear();

    myVisited.resize(myElements.size());

    for (const auto& leaf : QFindLeaves({ myRootAABB, 0 }, aSphere))
    {
        const auto nodeIndex = leaf.index;

```

```
        for (auto child = myNodes[nodeIndex].firstChild; child != -1;)
        {
            const auto eltIndex = myElementsPtr[child].element;
            const auto& elt     = myElements[eltIndex];

            if (!myVisited[eltIndex] && cu::IntersectionSphereAABB(aSphere, elt.aabb))
            {
                outResult.emplace_back(eltIndex);
                myVisited[eltIndex] = true;
            }

            child = myElementsPtr[child].next;
        }
    }

    for (const auto eltIndex : outResult)
    {
        myVisited[eltIndex] = false;
    }
}

template<std::equality_comparable T>
inline void Octree<T>::Query(const cu::Vector3f& aPoint, std::vector<SizeType>& outResult) const
{
    Query(cu::AABBf(aPoint, aPoint), outResult);
}

template<std::equality_comparable T>
inline void Octree<T>::Cleanup()
{
    std::scoped_lock lock(myMutex);

    if (myNodes.empty())
        return;

    std::vector<SizeType> toProcess;
    toProcess.reserve(ourChildCount * myMaxDepth / 2);

    if (IsBranch(myNodes[0]))
    {
        toProcess.emplace_back(0); // push root
    }

    while (!toProcess.empty())
    {
        Node& node = myNodes[toProcess.back()];
        toProcess.pop_back();

        int numEmpty = 0;
        for (int i = 0; i < ourChildCount; ++i)
        {
            const int childIndex    = node.firstChild + i;
            const Node& child       = myNodes[childIndex];

            if (IsBranch(child))
            {
                toProcess.emplace_back(childIndex);
            }
            else if (child.count == 0)
            {
```

```
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                     {
```

```
                        toProcess.emplace_back(cu::AABBf(r, t, f, r + hs.x, t + hs.y, f + hs.z), fc + 0, nd.depth + 1);
                    }
                    if (aAABB.GetMin().x < c.x)
                    {
                        toProcess.emplace_back(cu::AABBf(l, t, f, l + hs.x, t + hs.y, f + hs.z), fc + 1, nd.depth + 1);
                    }
                }
                if (aAABB.GetMin().y < c.y)
                {
                    if (aAABB.GetMax().x >= c.x)
                    {
                        toProcess.emplace_back(cu::AABBf(r, b, f, r + hs.x, b + hs.y, f + hs.z), fc + 2, nd.depth + 1);
                    }
                    if (aAABB.GetMin().x < c.x)
                    {
                        toProcess.emplace_back(cu::AABBf(l, b, f, l + hs.x, b + hs.y, f + hs.z), fc + 3, nd.depth + 1);
                    }
                }
            }
            if (aAABB.GetMin().z < c.z)
            {
                if (aAABB.GetMax().y >= c.y)
                {
                    if (aAABB.GetMax().x >= c.x)
                    {
                        toProcess.emplace_back(cu::AABBf(r, t, h, r + hs.x, t + hs.y, h + hs.z), fc + 4, nd.depth + 1);
                    }
                    if (aAABB.GetMin().x < c.x)
                    {
                        toProcess.emplace_back(cu::AABBf(l, t, h, l + hs.x, t + hs.y, h + hs.z), fc + 5, nd.depth + 1);
                    }
                }
                if (aAABB.GetMin().y < c.y)
                {
                    if (aAABB.GetMax().x >= c.x)
                    {
                        toProcess.emplace_back(cu::AABBf(r, b, h, r + hs.x, b + hs.y, h + hs.z), fc + 6, nd.depth + 1);
                    }
                    if (aAABB.GetMin().x < c.x)
                    {
                        toProcess.emplace_back(cu::AABBf(l, b, h, l + hs.x, b + hs.y, h + hs.z), fc + 7, nd.depth + 1);
                    }
                }
            }
        }
    }

    return leaves;
}

template<std::equality_comparable T>
inline auto Octree<T>::QFindLeaves(const NodeRegQuery& aNode, const cu::Vector3f& aStartPos, const cu::Vector3f& aEndPos) const -> std::vector<NodeQuery>
{
    std::vector<NodeQuery> leaves;
    std::vector<NodeRegQuery> toProcess;

    leaves.reserve(ourChildCount / 2);
    toProcess.reserve(ourChildCount * myMaxDepth / 2);

    toProcess.emplace_back(aNode);
```

```

    while (!toProcess.empty())
    {
        const auto nd = toProcess.back();
        toProcess.pop_back();

        if (IsLeaf(myNodes[nd.index]))
        {
            leaves.emplace_back(nd.index);
        }
        else
        {
            const auto fc   = myNodes[nd.index].firstChild;

            const auto c    = nd.aabb.GetCenter();
            const auto hs   = nd.aabb.GetExtends();
            const auto l    = nd.aabb.GetMin().x;
            const auto b    = nd.aabb.GetMin().y;
            const auto h    = nd.aabb.GetMin().z;
            const auto r    = l + hs.x;
            const auto t    = b + hs.y;
            const auto f    = h + hs.z;

            const auto b1 = cu::AABBf(r, t, f, r + hs.x, t + hs.y, f + hs.z);
            const auto b2 = cu::AABBf(l, t, f, l + hs.x, t + hs.y, f + hs.z);
            const auto b3 = cu::AABBf(r, b, f, r + hs.x, b + hs.y, f + hs.z);
            const auto b4 = cu::AABBf(l, b, f, l + hs.x, b + hs.y, f + hs.z);
            const auto b5 = cu::AABBf(r, t, h, r + hs.x, t + hs.y, h + hs.z);
            const auto b6 = cu::AABBf(l, t, h, l + hs.x, t + hs.y, h + hs.z);
            const auto b7 = cu::AABBf(r, b, h, r + hs.x, b + hs.y, h + hs.z);
            const auto b8 = cu::AABBf(l, b, h, l + hs.x, b + hs.y, h + hs.z);

            if (cu::IntersectionAABBSegment(b1, aStartPos, aEndPos)) toProcess.emplace_back(b1, fc + 0);
            if (cu::IntersectionAABBSegment(b2, aStartPos, aEndPos)) toProcess.emplace_back(b2, fc + 1);
            if (cu::IntersectionAABBSegment(b3, aStartPos, aEndPos)) toProcess.emplace_back(b3, fc + 2);
            if (cu::IntersectionAABBSegment(b4, aStartPos, aEndPos)) toProcess.emplace_back(b4, fc + 3);
            if (cu::IntersectionAABBSegment(b5, aStartPos, aEndPos)) toProcess.emplace_back(b5, fc + 4);
            if (cu::IntersectionAABBSegment(b6, aStartPos, aEndPos)) toProcess.emplace_back(b6, fc + 5);
            if (cu::IntersectionAABBSegment(b7, aStartPos, aEndPos)) toProcess.emplace_back(b7, fc + 6);
            if (cu::IntersectionAABBSegment(b8, aStartPos, aEndPos)) toProcess.emplace_back(b8, fc + 7);
        }
    }

    return leaves;
}

template<std::equality_comparable T>
inline auto Octree<T>::QFindLeaves(const NodeRegQuery& aNode, const cu::Frustumf& aFrustum) const -> std::vector<NodeQuery>
{
    std::vector<NodeQuery>      leaves;
    std::vector<NodeRegQuery>   toProcess;
    std::vector<SizeType>       toProcessContained;

    leaves.reserve(ourChildCount / 2);
    toProcess.reserve(ourChildCount * myMaxDepth / 2);
    toProcessContained.reserve(ourChildCount * myMaxDepth / 4);

    toProcess.emplace_back(aNode);

    while (!toProcess.empty())
```

```
    {
        const auto nd = toProcess.back();
        toProcess.pop_back();

        if (IsLeaf(myNodes[nd.index]))
        {
            leaves.emplace_back(nd.index);
        }
        else // TODO: optimize further if this becomes a bottleneck
        {
            if (aFrustum.Contains(nd.aabb))
            {
                GetNodeLeaves(nd.index, leaves, toProcessContained);
            }
            else
            {
                const auto fc = myNodes[nd.index].firstChild;

                const auto c    = nd.aabb.GetCenter();
                const auto hs   = nd.aabb.GetExtends();
                const auto l    = nd.aabb.GetMin().x;
                const auto b    = nd.aabb.GetMin().y;
                const auto h    = nd.aabb.GetMin().z;
                const auto r    = l + hs.x;
                const auto t    = b + hs.y;
                const auto f    = h + hs.z;

                const auto b1 = cu::AABBf(r, t, f, r + hs.x, t + hs.y, f + hs.z);
                const auto b2 = cu::AABBf(l, t, f, l + hs.x, t + hs.y, f + hs.z);
                const auto b3 = cu::AABBf(r, b, f, r + hs.x, b + hs.y, f + hs.z);
                const auto b4 = cu::AABBf(l, b, f, l + hs.x, b + hs.y, f + hs.z);
                const auto b5 = cu::AABBf(r, t, h, r + hs.x, t + hs.y, h + hs.z);
                const auto b6 = cu::AABBf(l, t, h, l + hs.x, t + hs.y, h + hs.z);
                const auto b7 = cu::AABBf(r, b, h, r + hs.x, b + hs.y, h + hs.z);
                const auto b8 = cu::AABBf(l, b, h, l + hs.x, b + hs.y, h + hs.z);

                if (aFrustum.IsInside(b1)) toProcess.emplace_back(b1, fc + 0);
                if (aFrustum.IsInside(b2)) toProcess.emplace_back(b2, fc + 1);
                if (aFrustum.IsInside(b3)) toProcess.emplace_back(b3, fc + 2);
                if (aFrustum.IsInside(b4)) toProcess.emplace_back(b4, fc + 3);
                if (aFrustum.IsInside(b5)) toProcess.emplace_back(b5, fc + 4);
                if (aFrustum.IsInside(b6)) toProcess.emplace_back(b6, fc + 5);
                if (aFrustum.IsInside(b7)) toProcess.emplace_back(b7, fc + 6);
                if (aFrustum.IsInside(b8)) toProcess.emplace_back(b8, fc + 7);
            }
        }
    }

    return leaves;
}
template<std::equality_comparable T>
inline auto Octree<T>::QFindLeavesNoDepth(const NodeRegQuery& aNode, const cu::Frustumf& aFrustum) const -> std::vector<NodeQuery>
{
    std::vector<NodeQuery>      leaves;
    std::vector<NodeRegQuery>   toProcess;
    std::vector<SizeType>       toProcessContained;

    leaves.reserve(ourChildCount / 2);
    toProcess.reserve(ourChildCount * myMaxDepth / 2);
    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}
1012template<std::equality_comparable T>
1013inline 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
```

```
    leaves.reserve(ourChildCount / 2);
    toProcess.reserve(ourChildCount * myMaxDepth / 2);

    toProcess.emplace_back(aNode);

    while (!toProcess.empty())
    {
        const auto nd = toProcess.back();
        toProcess.pop_back();

        if (IsLeaf(myNodes[nd.index]))
        {
            leaves.emplace_back(nd.index);
        }
        else
        {
            const auto fc   = myNodes[nd.index].firstChild;

            const auto c    = nd.aabb.GetCenter();
            const auto hs   = nd.aabb.GetExtends();
            const auto l    = nd.aabb.GetMin().x;
            const auto b    = nd.aabb.GetMin().y;
            const auto h    = nd.aabb.GetMin().z;
            const auto r    = l + hs.x;
            const auto t    = b + hs.y;
            const auto f    = h + hs.z;

            if (aAABB.GetMax().z >= c.z)
            {
                if (aAABB.GetMax().y >= c.y)
                {
                    if (aAABB.GetMax().x >= c.x)
                    {
                        toProcess.emplace_back(cu::AABBf(r, t, f, r + hs.x, t + hs.y, f + hs.z), fc + 0);
                    }
                    if (aAABB.GetMin().x < c.x)
                    {
                        toProcess.emplace_back(cu::AABBf(l, t, f, l + hs.x, t + hs.y, f + hs.z), fc + 1);
                    }
                }
                if (aAABB.GetMin().y < c.y)
                {
                    if (aAABB.GetMax().x >= c.x)
                    {
                        toProcess.emplace_back(cu::AABBf(r, b, f, r + hs.x, b + hs.y, f + hs.z), fc + 2);
                    }
                    if (aAABB.GetMin().x < c.x)
                    {
                        toProcess.emplace_back(cu::AABBf(l, b, f, l + hs.x, b + hs.y, f + hs.z), fc + 3);
                    }
                }
            }
            if (aAABB.GetMin().z < c.z)
            {
                if (aAABB.GetMax().y >= c.y)
                {
                    if (aAABB.GetMax().x >= c.x)
                    {
                        toProcess.emplace_back(cu::AABBf(r, t, h, r + hs.x, t + hs.y, h + hs.z), fc + 4);
                    }
```

```
                    if (aAABB.GetMin().x < c.x)
                    {
                        toProcess.emplace_back(cu::AABBf(l, t, h, l + hs.x, t + hs.y, h + hs.z), fc + 5);
                    }
                }
                if (aAABB.GetMin().y < c.y)
                {
                    if (aAABB.GetMax().x >= c.x)
                    {
                        toProcess.emplace_back(cu::AABBf(r, b, h, r + hs.x, b + hs.y, h + hs.z), fc + 6);
                    }
                    if (aAABB.GetMin().x < c.x)
                    {
                        toProcess.emplace_back(cu::AABBf(l, b, h, l + hs.x, b + hs.y, h + hs.z), fc + 7);
                    }
                }
            }
        }
    }

    return leaves;
}

template<std::equality_comparable T>
inline auto Octree<T>::QFindLeaves(const NodeRegQuery& aNode, const cu::Spheref& aSphere) const -> std::vector<NodeQuery>
{
    std::vector<NodeQuery> leaves;
    std::vector<NodeRegQuery> toProcess;

    leaves.reserve(ourChildCount / 2);
    toProcess.reserve(ourChildCount * myMaxDepth / 2);

    toProcess.emplace_back(aNode);

    while (!toProcess.empty())
    {
        const auto nd = toProcess.back();
        toProcess.pop_back();

        if (IsLeaf(myNodes[nd.index]))
        {
            leaves.emplace_back(nd.index);
        }
        else
        {
            const auto fc = myNodes[nd.index].firstChild;

            const auto c    = nd.aabb.GetCenter();
            const auto hs   = nd.aabb.GetExtends();
            const auto l    = nd.aabb.GetMin().x;
            const auto b    = nd.aabb.GetMin().y;
            const auto h    = nd.aabb.GetMin().z;
            const auto r    = l + hs.x;
            const auto t    = b + hs.y;
            const auto f    = h + hs.z;

            const auto b1 = cu::AABBf(r, t, f, r + hs.x, t + hs.y, f + hs.z);
            const auto b2 = cu::AABBf(l, t, f, l + hs.x, t + hs.y, f + hs.z);
            const auto b3 = cu::AABBf(r, b, f, r + hs.x, b + hs.y, f + hs.z);
            const auto b4 = cu::AABBf(l, b, f, l + hs.x, b + hs.y, f + hs.z);
```

```
            const auto b5 = cu::AABBf(r, t, h, r + hs.x, t + hs.y, h + hs.z);
            const auto b6 = cu::AABBf(l, t, h, l + hs.x, t + hs.y, h + hs.z);
            const auto b7 = cu::AABBf(r, b, h, r + hs.x, b + hs.y, h + hs.z);
            const auto b8 = cu::AABBf(l, b, h, l + hs.x, b + hs.y, h + hs.z);

            if (cu::IntersectionSphereAABB(aSphere, b1)) toProcess.emplace_back(b1, fc + 0);
            if (cu::IntersectionSphereAABB(aSphere, b2)) toProcess.emplace_back(b2, fc + 1);
            if (cu::IntersectionSphereAABB(aSphere, b3)) toProcess.emplace_back(b3, fc + 2);
            if (cu::IntersectionSphereAABB(aSphere, b4)) toProcess.emplace_back(b4, fc + 3);
            if (cu::IntersectionSphereAABB(aSphere, b5)) toProcess.emplace_back(b5, fc + 4);
            if (cu::IntersectionSphereAABB(aSphere, b6)) toProcess.emplace_back(b6, fc + 5);
            if (cu::IntersectionSphereAABB(aSphere, b7)) toProcess.emplace_back(b7, fc + 6);
            if (cu::IntersectionSphereAABB(aSphere, b8)) toProcess.emplace_back(b8, fc + 7);
        }
    }

    return leaves;
}

template<std::equality_comparable T>
inline void Octree<T>::GetNodeLeaves(SizeType aNodeIndex, std::vector<NodeQuery>& outLeaves, std::vector<SizeType>& aProcessNodes) const
{
    aProcessNodes.clear();

    aProcessNodes.emplace_back(aNodeIndex);

    while (!aProcessNodes.empty())
    {
        const auto index = aProcessNodes.back();
        aProcessNodes.pop_back();

        if (IsLeaf(myNodes[index]))
        {
            outLeaves.emplace_back(index);
        }
        else
        {
            const auto fc = myNodes[index].firstChild;

            aProcessNodes.emplace_back(fc + 0);
            aProcessNodes.emplace_back(fc + 1);
            aProcessNodes.emplace_back(fc + 2);
            aProcessNodes.emplace_back(fc + 3);
            aProcessNodes.emplace_back(fc + 4);
            aProcessNodes.emplace_back(fc + 5);
            aProcessNodes.emplace_back(fc + 6);
            aProcessNodes.emplace_back(fc + 7);
        }
    }
}

template<std::equality_comparable T>
inline bool Octree<T>::IsLeaf(const Node& aNode)
{
    return aNode.count != -1;
}
template<std::equality_comparable T>
inline bool Octree<T>::IsBranch(const Node& aNode)
{
    return aNode.count == -1;
```

```
}
```