Path Guiding & SD-Tree

1.SD-Tree

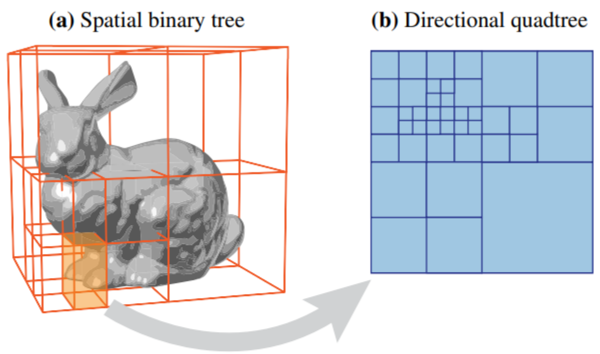

Figure 2: *The spatio-directional subdivision scheme of our SD-tree. Space is adaptively partitioned by a binary tree* ***(a)*** *that alternates between splitting the x, y, and z dimension in half. Each leaf node of the spatial binary tree contains a quadtree* ***(b)****, which approximates the spherical radiance field as an adaptively refined piecewise-constant function.*

1.1 Spatial Binary Tree

```

```

1.2 Directional Quadtree

```cpp
class QuadTreeNode {
public:
    QuadTreeNode() {
        m_children = {};
        for (size_t i = 0; i < m_sum.size(); ++i) {
            m_sum[i].store(0, std::memory_order_relaxed);
        }
    }

    void setSum(int index, Float val) {
        m_sum[index].store(val, std::memory_order_relaxed);
    }

    Float sum(int index) const {
        return m_sum[index].load(std::memory_order_relaxed);
    }

    void copyFrom(const QuadTreeNode& arg) {
        for (int i = 0; i < 4; ++i) {
            setSum(i, arg.sum(i));
            m_children[i] = arg.m_children[i];
        }
```

```
    }

    QuadTreeNode(const QuadTreeNode& arg) {
        copyFrom(arg);
    }

    QuadTreeNode& operator=(const QuadTreeNode& arg) {
        copyFrom(arg);
        return *this;
    }
    // val is the index of child node in the global vector<QuadTreeNode>
    //note that leaf nodes dont have an idx in the vector<QuadTreeNode>
    void setChild(int idx, uint16_t val) {
        m_children[idx] = val;
    }

    uint16_t child(int idx) const {
        return m_children[idx];
    }

    void setSum(Float val) {
        for (int i = 0; i < 4; ++i) {
            setSum(i, val);
        }
    }

    //Intresting Impl.
    int childIndex(Point2& p) const {
        int res = 0;
        for (int i = 0; i < Point2::dim; ++i) {
            if (p[i] < 0.5f) {
                p[i] *= 2;
            } else {
                p[i] = (p[i] - 0.5f) * 2;
                res |= 1 << i;
            }
        }

        return res;
    }

    // Evaluates the directional irradiance *sum density* (i.e. sum / area) at a given location p.
    // To obtain radiance, the sum density (result of this function) must be divided
    // by the total statistical weight of the estimates that were summed up.
    Float eval(Point2& p, const std::vector<QuadTreeNode>& nodes) const {
        SAssert(p.x >= 0 && p.x <= 1 && p.y >= 0 && p.y <= 1);
```

```
        const int index = childIndex(p);
        if (isLeaf(index)) {
            return 4 * sum(index);
        } else {
            return 4 * nodes[child(index)].eval(p, nodes);
        }
    }

    Float pdf(Point2& p, const std::vector<QuadTreeNode>& nodes) const {
        SAssert(p.x >= 0 && p.x <= 1 && p.y >= 0 && p.y <= 1);
        const int index = childIndex(p);
        if (!(sum(index) > 0)) {
            return 0;
        }

        const Float factor = 4 * sum(index) / (sum(0) + sum(1) + sum(2) + sum(3));
        if (isLeaf(index)) {
            return factor;
        } else {
            return factor * nodes[child(index)].pdf(p, nodes);
        }
    }
    //iteratively calculating the depth. Easy to understand.
    int depthAt(Point2& p, const std::vector<QuadTreeNode>& nodes) const {
        SAssert(p.x >= 0 && p.x <= 1 && p.y >= 0 && p.y <= 1);
        const int index = childIndex(p);
        if (isLeaf(index)) {
            return 1;
        } else {
            return 1 + nodes[child(index)].depthAt(p, nodes);
        }
    }
    //Very Interesting Impl.
    Point2 sample(Sampler* sampler, const std::vector<QuadTreeNode>& nodes) const {
        int index = 0;

        Float topLeft = sum(0);
        Float topRight = sum(1);
        Float partial = topLeft + sum(2);
        Float total = partial + topRight + sum(3);

        // Should only happen when there are numerical instabilities.
        if (!(total > 0.0f)) {
            return sampler->next2D();
        }

        Float boundary = partial / total;
        Point2 origin = Point2{0.0f, 0.0f};
```

```
        Float sample = sampler->next1D();

        if (sample < boundary) {
            SAssert(partial > 0);
            sample /= boundary;
            boundary = topLeft / partial;
        } else {
            partial = total - partial;
            SAssert(partial > 0);
            origin.x = 0.5f;
            sample = (sample - boundary) / (1.0f - boundary);
            boundary = topRight / partial;
            index |= 1 << 0;
        }

        if (sample < boundary) {
            sample /= boundary;
        } else {
            origin.y = 0.5f;
            sample = (sample - boundary) / (1.0f - boundary);
            index |= 1 << 1;
        }

        if (isLeaf(index)) {
            return origin + 0.5f * sampler->next2D();
        } else {
            return origin + 0.5f * nodes[child(index)].sample(sampler, nodes);
        }
    }

    void record(Point2& p, Float irradiance, std::vector<QuadTreeNode>& nodes) {
        SAssert(p.x >= 0 && p.x <= 1 && p.y >= 0 && p.y <= 1);
        int index = childIndex(p);

        if (isLeaf(index)) {
            addToAtomicFloat(m_sum[index], irradiance);
        } else {
            nodes[child(index)].record(p, irradiance, nodes);
        }
    }

    Float computeOverlappingArea(const Point2& min1, const Point2& max1, const Point2& min2, const Point2& max2) {
        Float lengths[2];
        for (int i = 0; i < 2; ++i) {
            lengths[i] = std::max(std::min(max1[i], max2[i]) - std::max(min1[i], min2[i]), 0.0f);
        }
        return lengths[0] * lengths[1];
```

```
    }

    void record(const Point2& origin, Float size, Point2 nodeOrigin, Float nodeSize, Float value, std::vector<QuadTreeNode>& nodes) {
        Float childSize = nodeSize / 2;
        for (int i = 0; i < 4; ++i) {
            Point2 childOrigin = nodeOrigin;
            if (i & 1) { childOrigin[0] += childSize; }
            if (i & 2) { childOrigin[1] += childSize; }

            Float w = computeOverlappingArea(origin, origin + Point2(size), childOrigin, childOrigin + Point2(childSize));
            if (w > 0.0f) {
                if (isLeaf(i)) {
                    addToAtomicFloat(m_sum[i], value * w);
                } else {
                    nodes[child(i)].record(origin, size, childOrigin, childSize, value, nodes);
                }
            }
        }
    }

    bool isLeaf(int index) const {
        return child(index) == 0;
    }

    // Ensure that each quadtree node's sum of irradiance estimates
    // equals that of all its children.
    void build(std::vector<QuadTreeNode>& nodes) {
        for (int i = 0; i < 4; ++i) {
            // During sampling, all irradiance estimates are accumulated in
            // the leaves, so the leaves are built by definition.
            if (isLeaf(i)) {
                continue;
            }

            QuadTreeNode& c = nodes[child(i)];

            // Recursively build each child such that their sum becomes valid...
            c.build(nodes);

            // ...then sum up the children's sums.
            Float sum = 0;
            for (int j = 0; j < 4; ++j) {
                sum += c.sum(j);
            }
            setSum(i, sum);
        }
    }
```

```
private:
    std::array<std::atomic<Float>, 4> m_sum;
    std::array<uint16_t, 4> m_children;
};
```

DTree

```
class DTree {
public:
    DTree() {
        m_atomic.sum.store(0, std::memory_order_relaxed);
        m_maxDepth = 0;
        m_nodes.emplace_back();
        m_nodes.front().setSum(0.0f);
    }

    const QuadTreeNode& node(size_t i) const {
        return m_nodes[i];
    }

    Float mean() const {
        if (m_atomic.statisticalWeight == 0) {
            return 0;
        }
        const Float factor = 1 / (M_PI * 4 * m_atomic.statisticalWeight);
        return factor * m_atomic.sum;
    }

    void recordIrradiance(Point2 p, Float irradiance, Float statisticalWeight, EDirectionalFilter directionalFilter) {
        if (std::isfinite(statisticalWeight) && statisticalWeight > 0) {
            addToAtomicFloat(m_atomic.statisticalWeight, statisticalWeight);

            if (std::isfinite(irradiance) && irradiance > 0) {
                if (directionalFilter == EDirectionalFilter::ENearest) {
                    m_nodes[0].record(p, irradiance * statisticalWeight, m_nodes);
                } else {
                    int depth = depthAt(p);
                    Float size = std::pow(0.5f, depth);

                    Point2 origin = p;
                    origin.x -= size / 2;
                    origin.y -= size / 2;
                    m_nodes[0].record(origin, size, Point2(0.0f), 1.0f, irradiance * statisticalWeight / (size * size), m_nodes);
```

```
                }
            }
        }
    }

    Float pdf(Point2 p) const {
        if (!(mean() > 0)) {
            return 1 / (4 * M_PI);
        }

        return m_nodes[0].pdf(p, m_nodes) / (4 * M_PI);
    }

    int depthAt(Point2 p) const {
        return m_nodes[0].depthAt(p, m_nodes);
    }

    int depth() const {
        return m_maxDepth;
    }

    Point2 sample(Sampler* sampler) const {
        if (!(mean() > 0)) {
            return sampler->next2D();
        }

        Point2 res = m_nodes[0].sample(sampler, m_nodes);

        res.x = math::clamp(res.x, 0.0f, 1.0f);
        res.y = math::clamp(res.y, 0.0f, 1.0f);

        return res;
    }

    size_t numNodes() const {
        return m_nodes.size();
    }

    Float statisticalWeight() const {
        return m_atomic.statisticalWeight;
    }

    void setStatisticalWeight(Float statisticalWeight) {
        m_atomic.statisticalWeight = statisticalWeight;
    }

    void reset(const DTree& previousDTree, int newMaxDepth, Float subdivisionThreshold) {
        m_atomic = Atomic{};
        m_maxDepth = 0;
        m_nodes.clear();
        m_nodes.emplace_back();
```

```

        struct StackNode {
            size_t nodeIndex;
            size_t otherNodeIndex;
            const DTree* otherDTree;
            int depth;
        };

        std::stack<StackNode> nodeIndices;
        nodeIndices.push({0, 0, &previousDTree, 1});

        const Float total = previousDTree.m_atomic.sum;

        // Create the topology of the new DTree to be the refined version
        // of the previous DTree. Subdivision is recursive if enough energy is there.
        while (!nodeIndices.empty()) {
            StackNode sNode = nodeIndices.top();
            nodeIndices.pop();

            m_maxDepth = std::max(m_maxDepth, sNode.depth);

            for (int i = 0; i < 4; ++i) {
                const QuadTreeNode& otherNode = sNode.otherDTree->m_nodes[sNode.otherNodeIndex];
                const Float fraction = total > 0 ? (otherNode.sum(i) / total) : std::pow(0.25f, sNode.depth);
                SAssert(fraction <= 1.0f + Epsilon);

                if (sNode.depth < newMaxDepth && fraction > subdivisionThreshold) {
                    if (!otherNode.isLeaf(i)) {
                        SAssert(sNode.otherDTree == &previousDTree);
                        nodeIndices.push({m_nodes.size(), otherNode.child(i), &previousDTree, sNode.depth + 1});
                    } else {
                        nodeIndices.push({m_nodes.size(), m_nodes.size(), this, sNode.depth + 1});
                    }

                    m_nodes[sNode.nodeIndex].setChild(i, static_cast<uint16_t>(m_nodes.size()));
                    m_nodes.emplace_back();
                    m_nodes.back().setSum(otherNode.sum(i) / 4);

                    if (m_nodes.size() > std::numeric_limits<uint16_t>::max()) {
                        SLog(EWarn, "DTreeWrapper hit maximum children count.");
                        nodeIndices = std::stack<StackNode>();
                        break;
```

```
                    }
                }
            }
        }

        // Uncomment once memory becomes an issue.
        //m_nodes.shrink_to_fit();

        for (auto& node : m_nodes) {
            node.setSum(0);
        }
    }

    size_t approxMemoryFootprint() const {
        return m_nodes.capacity() * sizeof(QuadTreeNode) + sizeof(*this);
    }

    void build() {
        auto& root = m_nodes[0];

        // Build the quadtree recursively, starting from its root.
        root.build(m_nodes);

        // Ensure that the overall sum of irradiance estimates equals
        // the sum of irradiance estimates found in the quadtree.
        Float sum = 0;
        for (int i = 0; i < 4; ++i) {
            sum += root.sum(i);
        }
        m_atomic.sum.store(sum);
    }

private:
    std::vector<QuadTreeNode> m_nodes;

    struct Atomic {
        Atomic() {
            sum.store(0, std::memory_order_relaxed);
            statisticalWeight.store(0, std::memory_order_relaxed);
        }

        Atomic(const Atomic& arg) {
            *this = arg;
        }

        Atomic& operator=(const Atomic& arg) {
            sum.store(arg.sum.load(std::memory_order_relaxed), std::memory_order_relaxed);
            statisticalWeight.store(arg.statisticalWeight.load(std::memory_order_relaxed), std::memory_order_relaxed);
```

```
                return *this;
            }

            std::atomic<Float> sum;
            std::atomic<Float> statisticalWeight;

        } m_atomic;

        int m_maxDepth;
};
```