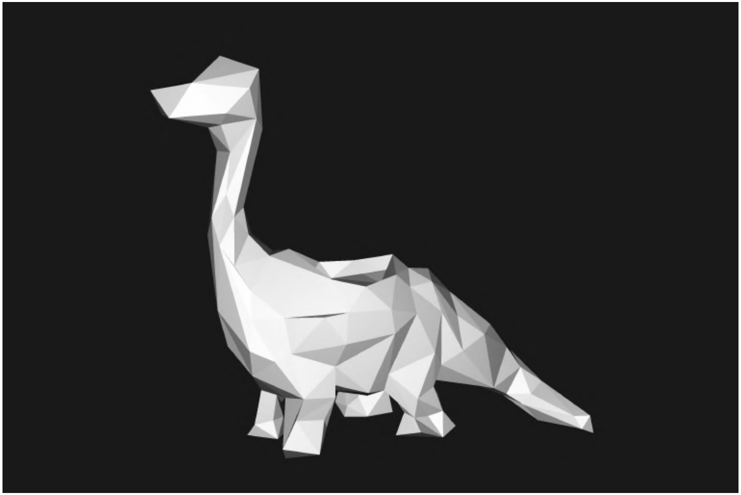
Accélération du ray-tracing

Bounding Volume Hierarchies

Computer Graphics : le rendu d'image



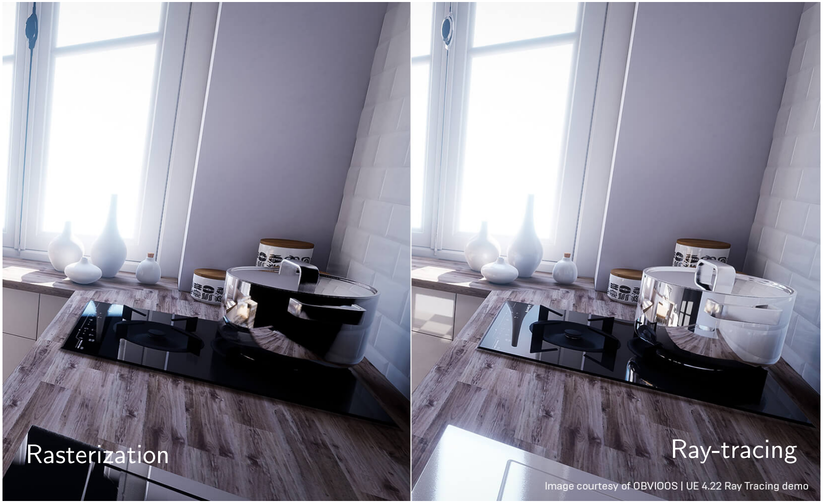
Problématique

Pb : mesurer l'influence de la géométrie des scènes sur la qualité d'un BVH simple et proposer une méthode de construction plus sophistiquée sans détériorer le temps de construction de l'arbre

Plan

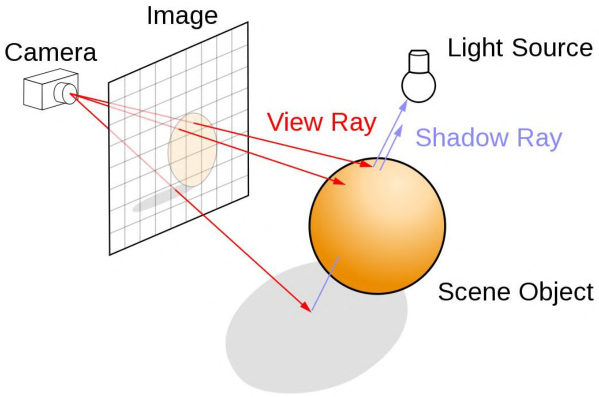
- Présentation du problème : qu'est-ce que le ray tracing ?
- Une structure d'accélération : les Bounding Volume Hierarchies (BVH)
- Influence de la géométrie de la scène sur la qualité du BVH
- Perfectionnement : les Spatial Splits
- Résultats
- Conclusion

Ray-tracing vs. Rasterization



Source : unrealengine.com

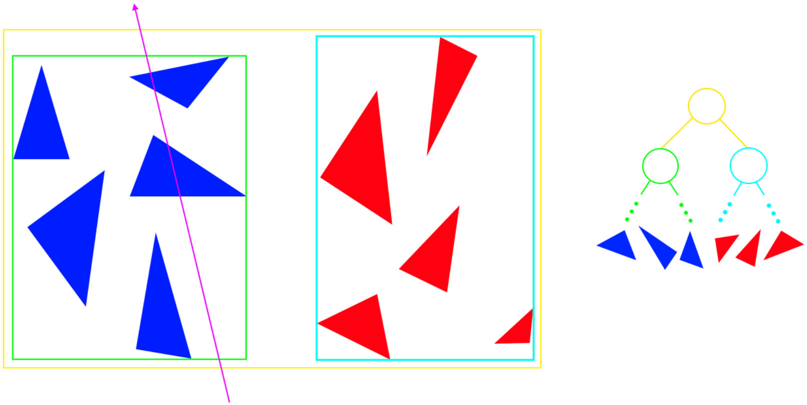
Principe du ray-tracing



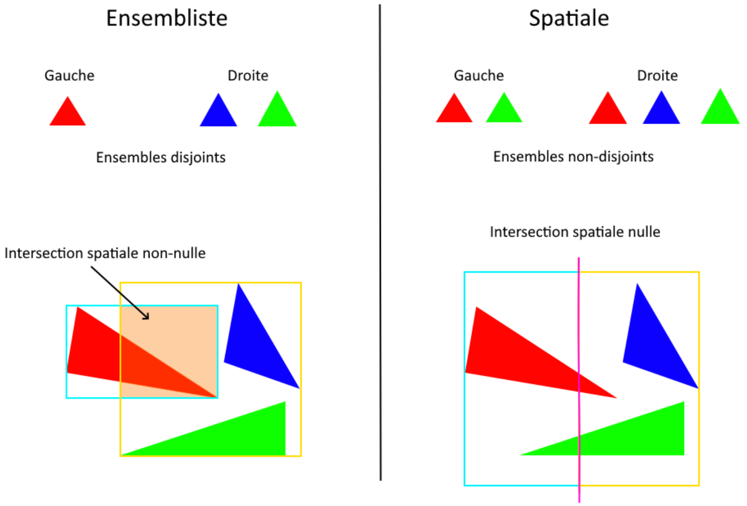
Source : developer.nvidia.com

Nb pixels : ≈ 1 000 000
Nb triangles : ≈ 100 000 - 10 000 000

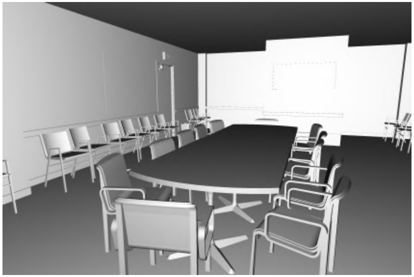
Structures d'accélération

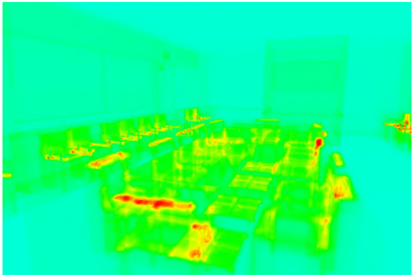


Construction de l'arbre



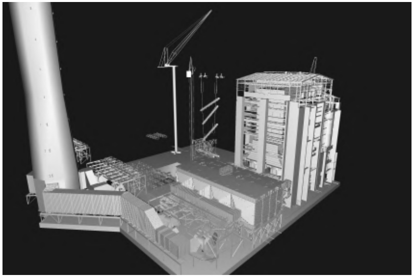
Influence de la géométrie de la scène

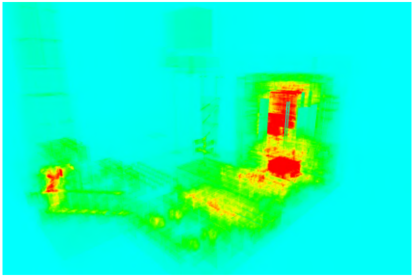




Conference ($\approx$ 120 000 triangles)
9/10 max hits (rouge) : $\approx$ 350 hits
Ref hits (vert) : $\approx$ 133 hits

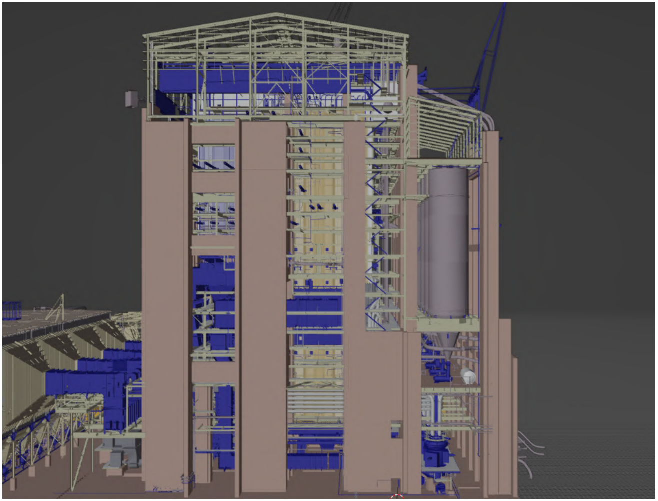
Influence de la géométrie de la scène



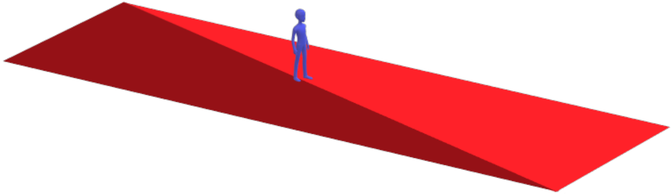


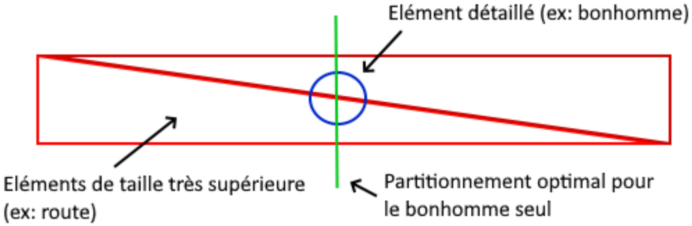
Powerplant ($\approx$ 12 000 000 triangles)
9/10 max hits (rouge) : $\approx$ 3000 hits
Ref hits (vert) : $\approx$ 500 hits

Influence de la géométrie de la scène

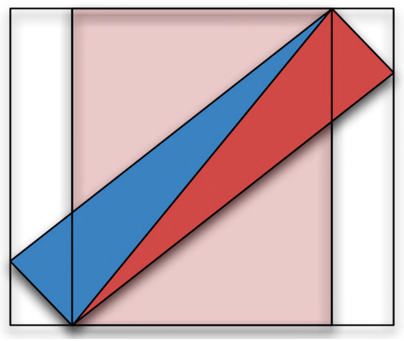


Influence de la géométrie de la scène

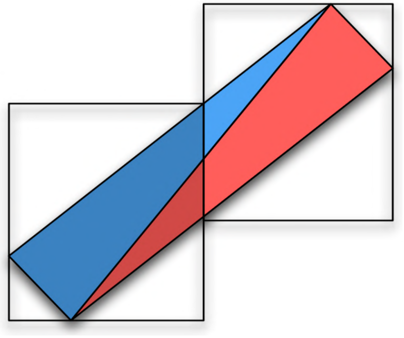



Elément détaillé (ex: bonhomme)
Eléments de taille très supérieure (ex: route)
Partitionnement optimal pour le bonhomme seul

Perfectionnement : les Spatial Splits



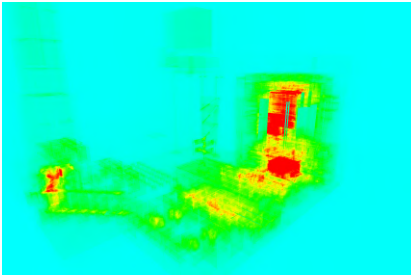
BVH classique



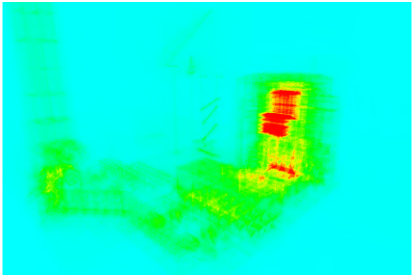
BVH avec Spatial Splits (SBVH)

Source : https://www.nvidia.in/docs/IO/77714/sbvh.pdf

Résultats



Rendu BVH : 26.04s
9/10 max hits (rouge) : $\approx$ 3000 hits

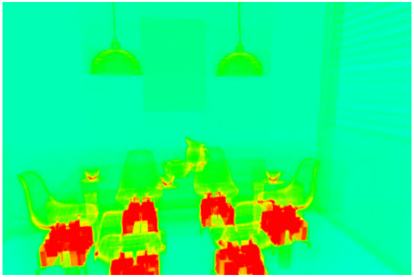


Rendu SBVH : 20.17s
9/10 max hits (rouge) : $\approx$ 1800 hits

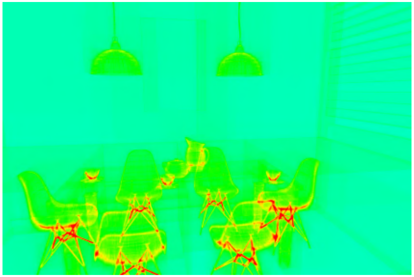
Powerplant ($\approx$ 12 000 000 triangles)
Ref hits (vert) : $\approx$ 500 hits

Gain de temps : $\approx$ 23%

Résultats



Rendu BVH : 16.58s
9/10 max hits (rouge) : $\approx$ 1000 hits

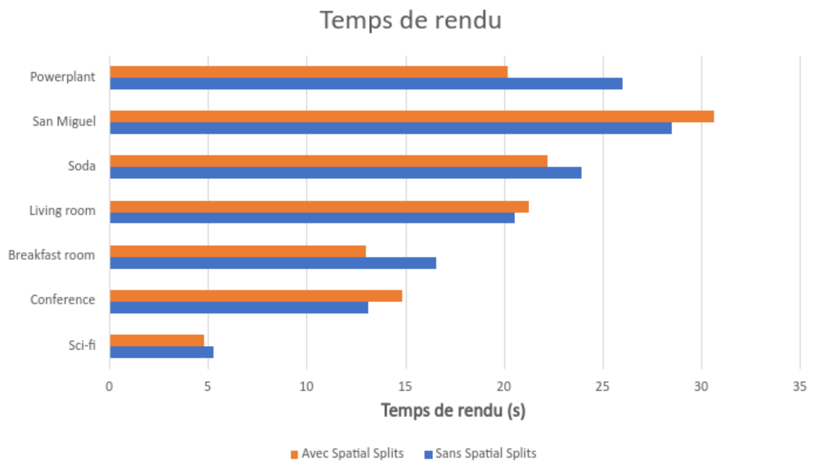


Rendu SBVH : 13.02s
9/10 max hits (rouge) : $\approx$ 450 hits

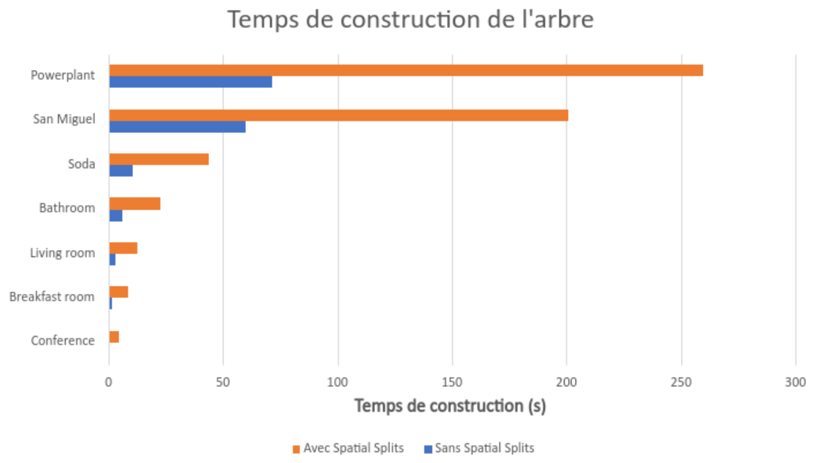
Breakfast room ($\approx$ 270 000 triangles)
Ref hits (vert) : $\approx$ 133 hits

Gain de temps : $\approx$ 21%

Résultats


Temps de rendu
Powerplant
San Miguel
Soda
Living room
Breakfast room
Conference
Sci-fi
0
5
10
15
20
25
30
35
Temps de rendu (s)
Avec Spatial Splits
Sans Spatial Splits

Conclusion


Temps de construction de l'arbre
Powerplant
San Miguel
Soda
Bathroom
Living room
Breakfast room
Conference
0
50
100
150
200
250
300
Temps de construction (s)
Avec Spatial Splits
Sans Spatial Splits

Annexe

Code

```c
#include <stdlib.h>
#include <float.h>
#include <assert.h>
#include <stdatomic.h>
#include <pthread.h>
#include <unistd.h>

#include "bvh.h"
#include "vector.h"
#include "aabb.h"
#include "render.h"
#include "deque.h"
#include "misc.h"

#define EPSILON 0.00000001

Split computeObjectSplit(int* nodeRefs, int nFrags, AABB bounds, AABB centroidBounds, Fragment* fragments) {
    Split split = {.type = 0, .cost = INFINITY, .leftBounds = aabbNewEmpty(), .rightBounds = aabbNewEmpty()};

    for (int dim = 0; dim < 3; dim++) {
        double boundsMin = vcomp(centroidBounds.pMin, dim);
        double boundsMax = vcomp(centroidBounds.pMax, dim) + EPSILON;
        double binStep = (boundsMax - boundsMin) / OBJECT_BIN_COUNT;

        if (binStep <= EPSILON) continue;

        double invBoundsExt = 1 / (boundsMax - boundsMin);

        ObjectBin bins[OBJECT_BIN_COUNT];

        // Initialize bins
        for (int i = 0; i < OBJECT_BIN_COUNT; i++) {
            bins[i].bounds = aabbNewEmpty();
            bins[i].count = 0;
        }

        // Assign primitives to a bin by centroid
        for (int i = 0; i < nFrags; i++) {
```

```
        Fragment fragment = fragments[nodeRefs[i]];
        int binIndex = invBoundsExt * OBJECT_BIN_COUNT * (vcomp(fragment.centroid, dim) - boundsMin);
        bins[binIndex].bounds = aabbUnion(bins[binIndex].bounds, fragment.bounds);
        bins[binIndex].count++;
    }

    // Compute partial costs
    double costs[OBJECT_BIN_COUNT - 1];

    // First pass computes half of the SAH
    AABB leftBounds = aabbNewEmpty();
    int leftCount = 0;
    for (int i = 0; i < OBJECT_BIN_COUNT - 1; i++) {
        leftBounds = aabbUnion(leftBounds, bins[i].bounds);
        leftCount += bins[i].count;
        costs[i] = aabbSA(leftBounds) * leftCount;
    }

    // Second pass computes the second half of the SAH and finds the best split
    double minPartialCost = INFINITY;
    int minSplitIndex = -1;

    AABB rightBounds = aabbNewEmpty();
    int rightCount = 0;
    for (int j = OBJECT_BIN_COUNT - 1; j > 0; j--) {
        rightBounds = aabbUnion(rightBounds, bins[j].bounds);
        rightCount += bins[j].count;

        double partialCost = costs[j - 1] + aabbSA(rightBounds) * rightCount;
        if (partialCost < minPartialCost) {
            minSplitIndex = j - 1;
            minPartialCost = partialCost;
        }
    }

    assert(minSplitIndex >= 0);

    // Calculate actual cost and update the split object
```

```
        double cost = TRAVERSAL_COST + minPartialCost / aabbSA(bounds);
        if (cost < split.cost) {
            split.splitIndex = minSplitIndex;
            split.cost = cost;
            split.dim = dim;

            // Recompute children bounds and primitive counts
            split.leftCount = 0;
            split.rightCount = 0;
            split.leftBounds = aabbNewEmpty();
            split.rightBounds = aabbNewEmpty();

            for (int i = 0; i <= minSplitIndex; i++) {
                split.leftCount += bins[i].count;
                split.leftBounds = aabbUnion(split.leftBounds, bins[i].bounds);
            }
            for (int j = minSplitIndex + 1; j < OBJECT_BIN_COUNT; j++) {
                split.rightCount += bins[j].count;
                split.rightBounds = aabbUnion(split.rightBounds, bins[j].bounds);
            }
        }
    }

    return split;
}

int* partitionObject(int* nodeRefs, int nFrags, Split split, int* leftChildRefs, int* rightChildRefs, AABB centroidBounds, Frag
    double boundsMin = vcomp(centroidBounds.pMin, split.dim);
    double boundsMax = vcomp(centroidBounds.pMax, split.dim) + EPSILON;
    double binStep = (boundsMax - boundsMin) / OBJECT_BIN_COUNT;
    double invBoundsExt = 1 / (boundsMax - boundsMin);

    int leftI = 0;
    int rightI = 0;
    for (int i = 0; i < nFrags; i++) {
        Fragment fragment = fragments[nodeRefs[i]];
        int binIndex = invBoundsExt * OBJECT_BIN_COUNT * (vcomp(fragment.centroid, split.dim) - boundsMin);
        if (binIndex <= split.splitIndex) {
            leftChildRefs[leftI++] = nodeRefs[i];
        }
        else {
            rightChildRefs[rightI++] = nodeRefs[i];
        }
    }
}

Split computeSpatialSplit(int* nodeRefs, int nFrags, AABB bounds, triangle* triangles, Fragment* fragments) {
    Split split = {.type = 1, .cost = INFINITY};

    for (int dim = 0; dim < 3; dim++) {
        double boundsMin = vcomp(bounds.pMin, dim);
        double boundsMax = vcomp(bounds.pMax, dim) + EPSILON;
        double binStep = (boundsMax - boundsMin) / SPATIAL_BIN_COUNT;

        if (binStep <= EPSILON) continue;

        double invBoundsExt = 1 / (boundsMax - boundsMin);

        SpatialBin bins[SPATIAL_BIN_COUNT];

        // Initialize bins
        for (int i = 0; i < OBJECT_BIN_COUNT; i++) {
            bins[i].bounds = aabbNewEmpty();
            bins[i].entries = 0;
            bins[i].exits = 0;
        }

        // Assign primitives to the bins they overlap
        for (int i = 0; i < nFrags; i++) {
            Fragment fragment = fragments[nodeRefs[i]];
            int binMin = invBoundsExt * SPATIAL_BIN_COUNT * (vcomp(fragment.bounds.pMin, dim) - boundsMin);
            int binMax = invBoundsExt * SPATIAL_BIN_COUNT * (vcomp(fragment.bounds.pMax, dim) - boundsMin);

            bins[binMin].entries++;
            bins[binMax].exits++;
            for (int j = binMin; j <= binMax; j++) {
                double leftPlanePos = boundsMin + j * binStep;
                double rightPlanePos = boundsMin + (j + 1) * binStep;
                AABB clippedTriangleBounds = triangleChoppedBounds(triangles[fragment.primIndex], dim, leftPlanePos, rightPlane

                // Restrict the clipped triangle bounds to the actual fragment
                AABB clippedBounds = aabbInter(clippedTriangleBounds, fragment.bounds);
                bins[j].bounds = aabbUnion(bins[j].bounds, clippedBounds);
            }
        }

        // Compute partial costs
        double costs[SPATIAL_BIN_COUNT - 1];

        // First pass computes half of the SAH
        AABB leftBounds = aabbNewEmpty();
        int leftCount = 0;
        for (int i = 0; i < OBJECT_BIN_COUNT - 1; i++) {
            leftBounds = aabbUnion(leftBounds, bins[i].bounds);
            leftCount += bins[i].entries;
            costs[i] = aabbSA(leftBounds) * leftCount;
        }

        // Second pass computes the second half of the SAH and finds the best split
        double minPartialCost = INFINITY;
        int minSplitIndex = -1;

        AABB rightBounds = aabbNewEmpty();
        int rightCount = 0;
        for (int j = OBJECT_BIN_COUNT - 1; j > 0; j--) {
            rightBounds = aabbUnion(rightBounds, bins[j].bounds);
            rightCount += bins[j].exits;

            double partialCost = costs[j - 1] + aabbSA(rightBounds) * rightCount;
            if (partialCost < minPartialCost) {
                minSplitIndex = j - 1;
                minPartialCost = partialCost;
            }
        }

        assert(minSplitIndex >= 0);

        // Calculate actual cost and update the split object
        double cost = TRAVERSAL_COST + minPartialCost / aabbSA(bounds);
        if (cost < split.cost) {
            split.splitIndex = minSplitIndex;
            split.cost = cost;
            split.dim = dim;

            // Recompute primitive counts
            split.leftCount = 0;
            split.rightCount = 0;

            for (int i = 0; i <= minSplitIndex; i++) {
                split.leftCount += bins[i].entries;
            }
            for (int j = minSplitIndex + 1; j < SPATIAL_BIN_COUNT; j++) {
                split.rightCount += bins[j].exits;
            }
        }
    }

    return split;
}

void partitionSpatial(int* nodeRefs, int nFrags, Split split, int* leftChildRefs, int* rightChildRefs, AABB bounds, atomic_int*
    double boundsMin = vcomp(bounds.pMin, split.dim);
    double boundsMax = vcomp(bounds.pMax, split.dim) + EPSILON;
    double binStep = (boundsMax - boundsMin) / SPATIAL_BIN_COUNT;
    double invBoundsExt = 1 / (boundsMax - boundsMin);
    double planePos = boundsMin + binStep * (split.splitIndex + 1);

    int leftI = 0;
    int rightI = 0;
    for (int i = 0; i < nFrags; i++) {
        Fragment fragment = fragments[nodeRefs[i]];
        int binMin = invBoundsExt * SPATIAL_BIN_COUNT * (vcomp(fragment.bounds.pMin, split.dim) - boundsMin);
        int binMax = invBoundsExt * SPATIAL_BIN_COUNT * (vcomp(fragment.bounds.pMax, split.dim) - boundsMin);

        if (binMax <= split.splitIndex) {
            leftChildRefs[leftI++] = nodeRefs[i];
        }
        else if (binMin > split.splitIndex) {
            rightChildRefs[rightI++] = nodeRefs[i];
        }
        else { // Duplicated ref
            AABB leftTriangleBounds;
            AABB rightTriangleBounds;

            triangleSplitBounds(triangles[fragment.primIndex], split.dim, planePos, &leftTriangleBounds, &rightTriangleBounds)

            // Restrict the split triangle bounds to the actual fragment bounds
            AABB leftBounds = aabbInter(leftTriangleBounds, fragment.bounds);
            AABB rightBounds = aabbInter(rightTriangleBounds, fragment.bounds);

            // Update left prim info
            fragments[nodeRefs[i]].bounds = leftBounds;
            fragments[nodeRefs[i]].centroid = aabbCentroid(leftBounds);

            // Update right prim info
            int newFragmentI = atomic_fetch_add(nextFragment, 1);
            fragments[newFragmentI].primIndex = fragment.primIndex;
            fragments[newFragmentI].bounds = rightBounds;
            fragments[newFragmentI].centroid = aabbCentroid(rightBounds);

            leftChildRefs[leftI++] = nodeRefs[i];
            rightChildRefs[rightI++] = newFragmentI;
        }
    }
}

void outputNodeBounds(AABB bounds, FILE* f) {
    vect pMin = bounds.pMin;
    vect pMax = bounds.pMax;
    fprintf(f, "v %f %f %f\n", pMin.x, pMin.y, pMin.z);
    fprintf(f, "v %f %f %f\n", pMax.x, pMin.y, pMin.z);
    fprintf(f, "v %f %f %f\n", pMin.x, pMin.y, pMax.z);
    fprintf(f, "v %f %f %f\n", pMax.x, pMin.y, pMax.z);
    fprintf(f, "v %f %f %f\n", pMin.x, pMax.y, pMin.z);
    fprintf(f, "v %f %f %f\n", pMax.x, pMax.y, pMin.z);
    fprintf(f, "v %f %f %f\n", pMin.x, pMax.y, pMax.z);
    fprintf(f, "v %f %f %f\n", pMax.x, pMax.y, pMax.z);
    fprintf(f, "f %d %d %d %d\n", -8, -7, -5, -6);
    fprintf(f, "f %d %d %d %d\n", -8, -6, -2, -4);
    fprintf(f, "f %d %d %d %d\n", -8, -7, -3, -4);
    fprintf(f, "f %d %d %d %d\n", -4, -3, -1, -2);
    fprintf(f, "f %d %d %d %d\n", -7, -5, -1, -3);
    fprintf(f, "f %d %d %d %d\n", -6, -5, -1, -2);
}

void outputBVH(BVH* bvh) {
    FILE* f = fopen("./bvh.obj", "w");

    for (int i = 0; i < bvh->nNodes; i++) {
        BVHNode node = bvh->nodes[i];
        outputNodeBounds(node.bounds, f);
    }

    fclose(f);
}

void makeLeafNode(int* nodeRefs, int nFrags, AABB bounds, int nodeIndex, BVHNode* nodes, atomic_int* nextRef, Fragment* fragmen
    int offset = atomic_fetch_add(nextRef, nFrags);
    nodes[nodeIndex].bounds = bounds;
    nodes[nodeIndex].nPrims = nFrags;
    nodes[nodeIndex].refsOffset = offset;

    for (int i = 0; i < nFrags; i++) {
        references[offset + i] = fragments[nodeRefs[i]].primIndex;
    }
}
```

```
void buildRec(buildWorkerArgs* args, taskArgs task) {
    int* nodeRefs = task.nodeRefs;
    int nFrags = task.nFrags;
    int nodeIndex = task.nodeIndex;
    BVHNode* nodes = task.nodes;
    Fragment* fragments = task.fragments;
    atomic_int* nextNode = task.nextNode;
    atomic_int* nextFragment = task.nextFragment;
    atomic_int* nextRef = task.nextRef;
    int* references = task.references;

    assert(nFrags > 0);

    // Compute node bounds and centroid bounds
    AABB bounds = aabbNewEmpty();
    AABB centroidBounds = aabbNewEmpty();

    for (int i = 0; i < nFrags; i++) {
        Fragment fragment = fragments[nodeRefs[i]];
        bounds = aabbUnion(bounds, fragment.bounds);
        centroidBounds = aabbExpand(centroidBounds, fragment.centroid);
    }

    if (nFrags <= 1) {
        makeLeafNode(nodeRefs, nFrags, bounds, nodeIndex, nodes, nextRef, fragments, references);
        (*args->nTasks)--;
        free(nodeRefs);
        return;
    }

    // Compute splits
    Split bestSplit = computeObjectSplit(nodeRefs, nFrags, bounds, centroidBounds, fragments);

    if (args->options->useSpatialSplits) {
        double lambda = aabbSA(aabbInter(bestSplit.leftBounds, bestSplit.rightBounds)) * args->invRootSA;
        assert(lambda >= 0 && lambda <= 1);
        if (lambda >= args->options->alpha) {
            Split spatialSplit = computeSpatialSplit(nodeRefs, nFrags, bounds, args->triangles, fragments);
            if (spatialSplit.cost < bestSplit.cost) bestSplit = spatialSplit;
        }
    }

    // Partition primitives according to the best split
    double leafCost = nFrags;
    if (leafCost <= bestSplit.cost) {
        makeLeafNode(nodeRefs, nFrags, bounds, nodeIndex, nodes, nextRef, fragments, references);
        (*args->nTasks)--;
        free(nodeRefs);
        return;
    }
    else {
        assert(bestSplit.leftCount + bestSplit.rightCount >= nFrags);

        int leftChildIndex = atomic_fetch_add(nextNode, 2);
        nodes[nodeIndex].bounds = bounds;
        nodes[nodeIndex].nPrims = 0;
        nodes[nodeIndex].axis = bestSplit.dim;
        nodes[nodeIndex].leftChild = leftChildIndex;
        nodes[nodeIndex].rightChild = leftChildIndex + 1; // Useless, but easier for traversal

        int* leftChildRefs = malloc(bestSplit.leftCount * sizeof(int));
        int* rightChildRefs = malloc(bestSplit.rightCount * sizeof(int));

        if (bestSplit.type == 0) partitionObject(nodeRefs, nFrags, bestSplit, leftChildRefs, rightChildRefs, centroidBounds, fr
        else partitionSpatial(nodeRefs, nFrags, bestSplit, leftChildRefs, rightChildRefs, bounds, nextFragment, args->triangles

        taskArgs leftTask = {.nodeRefs = leftChildRefs, .nFrags = bestSplit.leftCount, .nodeIndex = leftChildIndex, .nextNode =
        taskArgs rightTask = {.nodeRefs = rightChildRefs, .nFrags = bestSplit.rightCount, .nodeIndex = leftChildIndex + 1, .nex

        free(nodeRefs);

        (*args->nTasks)++;
        if (leftTask.nFrags > rightTask.nFrags) {
            pushRight(args->queues[args->index], leftTask);
            buildRec(args, rightTask);
        }
        else {
            pushRight(args->queues[args->index], rightTask);
            buildRec(args, leftTask);
        }
    }
}

static bool trySteal(buildWorkerArgs* data, taskArgs* t){
    int index = data->index;
    for (int i = (index + 1) % N_THREADS; i != index; i = (i + 1) % N_THREADS) {
        if (tryPopLeft(data->queues[i], t)) return true;
    }
    return false;
}

static void* buildWorker(void* args) {
    buildWorkerArgs data = *(buildWorkerArgs*) args;
    int index = data.index;
    deque* q = data.queues[index];
    while (*data.nTasks > 0) {
        taskArgs t;
        if (tryPopRight(q, &t)) {
            buildRec(&data, t);
        } else if (trySteal(&data, &t)){
            buildRec(&data, t);
        }
    }
}

BVH* buildBVH(triangle* triangles, int n, BVHOptions* options) {
    // Initialize construction variables
    BVH* bvh = (BVH*) malloc(sizeof(BVH));
    Fragment* fragments = (Fragment*) malloc(n * MAX_REFS_FACT * sizeof(Fragment));
    int* references = (int*) malloc(n * MAX_REFS_FACT * sizeof(int));
    BVHNode* nodes = (BVHNode*) malloc(n * MAX_REFS_FACT * 2 * sizeof(BVHNode));
    atomic_int* nextNode = malloc(sizeof(atomic_int));
```

```c
atomic_int* nextFragment = malloc(sizeof(atomic_int));
atomic_int* nextRef = malloc(sizeof(atomic_int));
*nextNode = 1;
*nextFragment = n;
*nextRef = 0;

// Compute primitives' centroids and root bounds
int* rootRefs = malloc(n * sizeof(int));
AABB rootBounds = aabbNewEmpty();

for (int i = 0; i < n; i++) {
    triangle t = triangles[i];
    AABB bounds = triangleBounds(t);
    fragments[i].primIndex = i;
    fragments[i].bounds = bounds;
    fragments[i].centroid = aabbCentroid(bounds);

    rootRefs[i] = i;

    rootBounds = aabbUnion(rootBounds, bounds);
}

// Initialize work stealing
pthread_t* threads = (pthread_t*) malloc(N_THREADS * sizeof(pthread_t));
buildWorkerArgs* args = (buildWorkerArgs*) malloc(N_THREADS * sizeof(buildWorkerArgs));
deque** queues = (deque**) malloc(N_THREADS * sizeof(deque*));

for (int i = 0; i < N_THREADS; i++) {
    queues[i] = newDeque();
}

taskArgs rootTask = {.nodeRefs = rootRefs, .nFrags = n, .nodeIndex = 0, .nextNode = nextNode, .nextFragment = nextFragment,
pushLeft(queues[0], rootTask);
atomic_int* nTasks = malloc(sizeof(atomic_int));
*nTasks = 1;

for (int i = 0; i < N_THREADS; i++) {
    args[i].triangles = triangles;
```

```
        args[i].options = options;
        args[i].invRootSA = 1 / aabbSA(rootBounds);
        args[i].queues = queues;
        args[i].nTasks = nTasks;
        args[i].index = i;
        pthread_create(&threads[i], NULL, buildWorker, &args[i]);
    }
    for (int i = 0; i < N_THREADS; i++) {
        pthread_join(threads[i], NULL);
    }

    // Cleanup and result
    bvh->nodes = nodes;
    bvh->references = references;
    bvh->nNodes = *nextNode;
    bvh->nRefs = *nextRef;

    free(fragments);
    free(nextNode);
    free(nextFragment);
    free(nextRef);
    free(threads);
    free(args);
    for (int i = 0; i < N_THREADS; i++) freeDeque(queues[i]);
    free(queues);
    free(nTasks);

    return bvh;
}

void intersectBVH(triangle* triangles, BVH* bvh, Ray* r, int* hits) {
    vect invDir = vinv(r->dir);
    int dirIsNeg[3] = { invDir.x < 0, invDir.y < 0, invDir.z < 0 };

    int stackSize = 1;
    int stack[64];
    stack[0] = 0;
    *hits = 0;

    while (stackSize > 0) {
        BVHNode node = bvh->nodes[stack[--stackSize]];
        (*hits)++;
        if (aabbRayIntersects(node.bounds, *r, invDir)) {
            if (node.nPrims > 0) {
                for (int i = node.refsOffset; i < node.refsOffset + node.nPrims; i++) {
                    triangle tri = triangles[bvh->references[i]];
                    triangleRayIntersection(tri, r);
                    (*hits)++;
                }
            }
            else {
                if (dirIsNeg[node.axis]) {
                    stack[stackSize++] = node.rightChild;
                    stack[stackSize++] = node.leftChild;
                }
                else {
                    stack[stackSize++] = node.leftChild;
                    stack[stackSize++] = node.rightChild;
                }
            }
        }
    }
}


#ifndef BVH_H
#define BVH_H

#include "deque.h"
#include "vector.h"
#include "aabb.h"
#include "render.h"

struct Fragment {
    int primIndex;
    AABB bounds;
    vect centroid;
};

typedef struct Fragment Fragment;

struct BVHNode {
    AABB bounds;
    int nPrims;
    int refsOffset;
    int axis;
    int leftChild;
    int rightChild;
};

typedef struct BVHNode BVHNode;

struct BVH {
    BVHNode* nodes;
    int* references;
    int nNodes;
    int nRefs;
};

typedef struct BVH BVH;

struct Split {
    int splitIndex;
    double cost;
    int dim;

    int type; // Object = 0, Spatial = 1

    int leftCount;
    int rightCount;
    AABB leftBounds;
    AABB rightBounds;
};
```

```
typedef struct Split Split;

struct ObjectBin {
    AABB bounds;
    int count;
};

typedef struct ObjectBin ObjectBin;

struct SpatialBin {
    AABB bounds;
    int entries;
    int exits;
};

typedef struct SpatialBin SpatialBin;

struct BVHOptions {
    bool useSpatialSplits;
    double alpha;
};

typedef struct BVHOptions BVHOptions;

struct buildWorkerArgs {
    triangle* triangles;
    BVHOptions* options;
    double invRootSA;
    deque** queues;
    atomic_int* nTasks;
    int index;
};

typedef struct buildWorkerArgs buildWorkerArgs;

BVH* buildBVH(triangle* triangles, int n, BVHOptions* options);

void intersectBVH(triangle* triangles, BVH* bvh, Ray* r, int* hits);
```

```c
void outputBVH(BVH* bvh);

#endif


#include <float.h>

#include "vector.h"
#include "triangle.h"
#include "ray.h"
#include "aabb.h"

void swapVects(vect* arr, int i, int j) {
    vect tmp = arr[i];
    arr[i] = arr[j];
    arr[j] = tmp;
}

bool triangleRayIntersection(triangle tri, Ray* r) {
    vect v0 = tri.vertices[tri.v[0]],
        v1 = tri.vertices[tri.v[1]],
        v2 = tri.vertices[tri.v[2]];

    vect v0v1 = vsub(v1, v0);
    vect v0v2 = vsub(v2, v0);
    vect pVec = vcross(r->dir, v0v2);
    double det = vdot(v0v1, pVec);

    if (fabsl(det) < DBL_EPSILON) return false;

    double invDet = (double) 1 / det;
    vect tVec = vsub(r->orig, v0);
    double u = vdot(tVec, pVec) * invDet;

    if (u < 0 || u > 1) return false;

    vect qVec = vcross(tVec, v0v1);
    double v = vdot(r->dir, qVec) * invDet;
    if (v < 0 || u + v > 1) return false;

    double t = vdot(v0v2, qVec) * invDet;
    if (t < r->tMax) {
        r->hitInfo.n[0] = tri.n[0];
        r->hitInfo.n[1] = tri.n[1];
        r->hitInfo.n[2] = tri.n[2];
        r->hitInfo.u = u;
        r->hitInfo.v = v;
        r->tMax = t;
        r->hasHit = true;
        return true;
    }
}

int trianglePlaneIntersection(vect vertices[3], int dim, double planePos, vect intersections[2]) {
    // If all vertices lie on the same side of the plane, no intersection
    if (vcomp(vertices[2], dim) < planePos || vcomp(vertices[0], dim) >= planePos) {
        return 0;
    }

    // Lerp vertices to find the 2 intersections
    int isxCount = 0;
    for (int i = 0; i < 2; i++) {
        vect from = vertices[i];
        for (int j = i + 1; j < 3; j++) {
            vect to = vertices[j];
            if (vcomp(from, dim) < planePos && vcomp(to, dim) >= planePos) {
                double delta = vcomp(to, dim) - vcomp(from, dim);
                double t = (planePos - vcomp(from, dim)) / delta;
                intersections[isxCount++] = vadd(vmult(from, 1 - t), vmult(to, t));
            }
        }
    }

    assert(isxCount == 2);
    return 2;
}

AABB triangleChoppedBounds(triangle tri, int dim, double leftPlanePos, double rightPlanePos) {
    vect vertices[3] = { tri.vertices[tri.v[0]], tri.vertices[tri.v[1]], tri.vertices[tri.v[2]] };

    // Sort vertices along current dim
    if (vcomp(vertices[1], dim) < vcomp(vertices[0], dim)) swapVects(vertices, 0, 1);
    if (vcomp(vertices[2], dim) < vcomp(vertices[1], dim)) swapVects(vertices, 1, 2);
    if (vcomp(vertices[1], dim) < vcomp(vertices[0], dim)) swapVects(vertices, 0, 1);

    // Compute intersections
    vect leftIsx[2];
    vect rightIsx[2];
    int leftIsxCount = trianglePlaneIntersection(vertices, dim, leftPlanePos, leftIsx);
    int rightIsxCount = trianglePlaneIntersection(vertices, dim, rightPlanePos, rightIsx);

    AABB clipped = aabbUnion(aabbFromPoints(leftIsx, leftIsxCount), aabbFromPoints(rightIsx, rightIsxCount));
    for (int i = 0; i < 3; i++) {
        if (vcomp(vertices[i], dim) >= leftPlanePos && vcomp(vertices[i], dim) < rightPlanePos) {
            clipped = aabbExpand(clipped, vertices[i]);
        }
    }

    return clipped;
}

void triangleSplitBounds(triangle tri, int dim, double planePos, AABB* leftBounds, AABB* rightBounds) {
    vect vertices[3] = { tri.vertices[tri.v[0]], tri.vertices[tri.v[1]], tri.vertices[tri.v[2]] };

    // Sort vertices along current dim
    if (vcomp(vertices[1], dim) < vcomp(vertices[0], dim)) swapVects(vertices, 0, 1);
    if (vcomp(vertices[2], dim) < vcomp(vertices[1], dim)) swapVects(vertices, 1, 2);
    if (vcomp(vertices[1], dim) < vcomp(vertices[0], dim)) swapVects(vertices, 0, 1);

    vect isx[2];
    int isxCount = trianglePlaneIntersection(vertices, dim, planePos, isx);
    *leftBounds = *rightBounds = aabbFromPoints(isx, isxCount);
    for (int i = 0; i < 3; i++) {
        if (vcomp(vertices[i], dim) >= planePos) {
            *rightBounds = aabbExpand(*rightBounds, vertices[i]);
        }
        else {
            *leftBounds = aabbExpand(*leftBounds, vertices[i]);
        }
    }
}

AABB triangleBounds(triangle t) {
    return aabbExpand(aabbExpand(aabbNewPoint(t.vertices[t.v[0]]), t.vertices[t.v[1]]), t.vertices[t.v[2]]);
}


#ifndef TRIANGLE_H
#define TRIANGLE_H

#include "ray.h"
#include "aabb.h"

struct triangle {
    int v[3];
    int n[3];
    vect* vertices;
};

typedef struct triangle triangle;

bool triangleRayIntersection(triangle tri, Ray* r);

AABB triangleChoppedBounds(triangle tri, int dim, double leftPlanePos, double rightPlanePos);

void triangleSplitBounds(triangle tri, int dim, double planePos, AABB* leftBounds, AABB* rightBounds);

AABB triangleBounds(triangle t);

#endif
```

```c
#include <float.h>
#include <stdbool.h>

#include "vector.h"
#include "aabb.h"
#include "misc.h"
#include "ray.h"

AABB aabbNewEmpty() {
    AABB b;
    b.pMin = vnew(INFINITY, INFINITY, INFINITY);
    b.pMax = vnew(-INFINITY, -INFINITY, -INFINITY);
    return b;
}

AABB aabbNewPoint(vect p) {
    AABB b = {.pMin = p, .pMax = p};
    return b;
}

AABB aabbNewPoints(vect pMin, vect pMax) {
    AABB b = {.pMin = pMin, .pMax = pMax};
    return b;
}

AABB aabbFromPoints(vect* points, int n) {
    AABB b = aabbNewEmpty();
    for (int i = 0; i < n; i++) {
        b = aabbExpand(b, points[i]);
    }
    return b;
}

bool aabbIsEmpty(AABB b) {
    return b.pMin.x > b.pMax.x || b.pMin.y > b.pMax.y || b.pMin.z > b.pMax.z;
}

AABB aabbExpand(AABB b, vect p) {
    return aabbNewPoints(vmin(b.pMin, p), vmax(b.pMax, p));
}

AABB aabbUnion(AABB b1, AABB b2) {
    return aabbNewPoints(vmin(b1.pMin, b2.pMin), vmax(b1.pMax, b2.pMax));
}

AABB aabbInter(AABB b1, AABB b2) {
    return aabbNewPoints(vmax(b1.pMin, b2.pMin), vmin(b1.pMax, b2.pMax));
}

vect aabbDiag(AABB b) {
    return vsub(b.pMax, b.pMin);
}

double aabbSA(AABB b) {
    if (aabbIsEmpty(b)) return 0;

    vect d = aabbDiag(b);
    return 2 * (d.x * d.y + d.y * d.z + d.x * d.z);
}

vect aabbCentroid(AABB b) {
    return vadd(vmult(b.pMin, 0.5), vmult(b.pMax, 0.5));
}

bool aabbIsPointInside(vect p, AABB a) {
    return (p.x >= a.pMin.x && p.x <= a.pMax.x)
        && (p.y >= a.pMin.y && p.y <= a.pMax.y)
        && (p.z >= a.pMin.z && p.z <= a.pMax.z);
}

bool aabbIsWithin(AABB a, AABB b) {
    return aabbIsPointInside(a.pMin, b) && aabbIsPointInside(a.pMax, b);
}

bool aabbRayIntersects(AABB b, Ray r, vect invDir) {
    if (aabbIsEmpty(b)) return false;
```

```c
    double t0 = 0;
    double t1 = r.tMax;

    for (int i = 0; i < 3; i++) {
        double tNear = (vcomp(b.pMin, i) - vcomp(r.orig, i)) * vcomp(invDir, i);
        double tFar = (vcomp(b.pMax, i) - vcomp(r.orig, i)) * vcomp(invDir, i);

        if (tNear > tFar) {
            double tmp = tNear;
            tNear = tFar;
            tFar = tmp;
        }

        t0 = max(tNear, t0);
        t1 = min(tFar, t1);

        if (t0 > t1) return false;
    }

    return true;
}

#ifndef AABB_H
#define AABB_H

#include <float.h>

#include "vector.h"
#include "ray.h"

struct AABB {
    vect pMin;
    vect pMax;
};

typedef struct AABB AABB;
```

```c
AABB aabbNewEmpty();

AABB aabbNewPoint(vect p) ;

AABB aabbFromPoints(vect* points, int n);

AABB aabbExpand(AABB b, vect p);

AABB aabbUnion(AABB b1, AABB b2);

AABB aabbInter(AABB b1, AABB b2);

vect aabbDiag(AABB b);

double aabbSA(AABB b);

vect aabbCentroid(AABB b);

bool aabbRayIntersects(AABB b, Ray r, vect invDir);

bool aabbIsPointInside(vect p, AABB a);

bool aabbIsWithin(AABB a, AABB b);

#endif


#include <time.h>

#include "clock.h"

double deltaT(timestamp t0, timestamp t1) {
    return (t1.tv_sec - t0.tv_sec) + 1e-9 * (t1.tv_nsec - t0.tv_nsec);
}

timestamp getTime(void) {
    timestamp t;
    clock_gettime(CLOCK_REALTIME, &t);
    return t;
}


#ifndef CLOCK_H
#define CLOCK_H

#include <time.h>

typedef struct timespec timestamp;

timestamp getTime(void);

double deltaT(timestamp t0, timestamp t1);

#endif
```

```c
#include <stdlib.h>
#include <assert.h>

#include "color.h"

color cnew(double r, double g, double b) {
    color c = {.r = r, .g = g, .b = b};
        return c;
}

color cmult(color a, double k) {
    return cnew(a.r * k, a.g * k, a.b * k);
}

color cdiv(color a, double k) {
    assert(k != 0);
    return cmult(a, (double) 1 / k);
}

color cneg(color a) {
    return cnew(-a.r, -a.g, -a.b);
}
```

```c
color cadd(color a, color b) {
    return cnew(a.r + b.r, a.g + b.g, a.b + b.b);
}

color csub(color a, color b) {
    return cadd(a, cneg(b));
}

void printColor(color a) {
    printf("{%.3f, %.3f, %.3f}\n", a.r, a.g, a.b);
}
```

```c
#ifndef COLOR_H
#define COLOR_H

#include <stdio.h>

struct color {
    double r;
    double g;
    double b;
};

typedef struct color color;

color cnew(double r, double g, double b);

color cmult(color a, double k);

color cdiv(color a, double k);

color cneg(color a);

color cadd(color a, color b);

color csub(color a, color b);
```

```c
void printColor(color a);

#endif


#include <stdlib.h>
#include <stdio.h>
#include <assert.h>
#include <pthread.h>
#include <stdbool.h>

#include "deque.h"

const taskArgs DEFAULT = {};

node* newNode(taskArgs data) {
    node* n = malloc(sizeof(node));
    n->next = NULL;
    n->prev = NULL;
    n->data = data;
    return n;
}

deque* newDeque(void) {
    deque* q = malloc(sizeof(deque));
    q->sentinel = newNode(DEFAULT);
    q->sentinel->next = q->sentinel;
    q->sentinel->prev = q->sentinel;
    pthread_mutex_init(&q->lock, NULL);
    return q;
}

bool tryPopLeft(deque* q, taskArgs* res) {
    pthread_mutex_lock(&q->lock);
    node* head = q->sentinel->next;
    if (head == q->sentinel) {
        pthread_mutex_unlock(&q->lock);
        return false;
    }
    head->prev->next = head->next;
    head->next->prev = head->prev;
    *res = head->data;
    free(head);
    pthread_mutex_unlock(&q->lock);
    return true;
}

bool tryPopRight(deque* q, taskArgs* res) {
    pthread_mutex_lock(&q->lock);
    node* tail = q->sentinel->prev;
    if (tail == q->sentinel) {
        pthread_mutex_unlock(&q->lock);
        return false;
    }
    tail->next->prev = tail->prev;
    tail->prev->next = tail->next;
    *res = tail->data;
    free(tail);
    pthread_mutex_unlock(&q->lock);
    return true;
}

void pushLeft(deque* q, taskArgs data) {
    pthread_mutex_lock(&q->lock);
    node* n = newNode(data);
    n->prev = q->sentinel;
    n->next = q->sentinel->next;
    n->prev->next = n;
    n->next->prev = n;
    pthread_mutex_unlock(&q->lock);
}

void pushRight(deque* q, taskArgs data) {
    pthread_mutex_lock(&q->lock);
    node* n = newNode(data);
    n->next = q->sentinel;
    n->prev = q->sentinel->prev;
    n->prev->next = n;
    n->next->prev = n;
    pthread_mutex_unlock(&q->lock);
}

void freeDeque(deque* q) {
    pthread_mutex_lock(&q->lock);
    node* n = q->sentinel->next;
    while (n != q->sentinel) {
        node* tmp = n->next;
        free(n);
        n = tmp;
    }
    free(q->sentinel);
    pthread_mutex_unlock(&q->lock);
    pthread_mutex_destroy(&q->lock);
    free(q);
}


#ifndef DEQUE_H
#define DEQUE_H

#include <stdbool.h>
#include <stdatomic.h>

struct taskArgs {
    int* nodeRefs;
    int nFrags;
    int nodeIndex;
    atomic_int* nextNode;
    atomic_int* nextFragment;
    atomic_int* nextRef;
    atomic_int* nObj;
    atomic_int* nSpat;
    struct BVHNode* nodes;
    struct Fragment* fragments;
    int* references;
};
```

```c
typedef struct taskArgs taskArgs;

struct node {
    taskArgs data;
    struct node* next;
    struct node* prev;
};

typedef struct node node;

struct deque {
    node* sentinel;
    pthread_mutex_t lock;
};

typedef struct deque deque;

deque* newDeque(void);

bool tryPopLeft(deque* q, taskArgs* res);

bool tryPopRight(deque* q, taskArgs* res);

void pushLeft(deque* q, taskArgs data);

void pushRight(deque* q, taskArgs data);

void freeDeque(deque* q);

#endif
```

```c
#ifndef MISC_H
#define MISC_H

#define min(a,b) (((a) < (b)) ? (a) : (b))
#define max(a,b) (((a) > (b)) ? (a) : (b))
```

```c
#define MAX_VERTICES 10000000
#define MAX_TRIANGLES 20000000
#define MAX_NORMALS 3 * MAX_TRIANGLES
#define N_THREADS 8
#define PIXEL_CHUNCK_SIZE 1000
#define MAX_REFS_FACT 3
#define OBJECT_BIN_COUNT 16
#define SPATIAL_BIN_COUNT 16
#define TRAVERSAL_COST 0.125
#define DEFAULT_ALPHA 0.00001

#endif


#include <stdbool.h>
#include <float.h>

#include "ray.h"
#include "vector.h"
#include "color.h"

Ray pointAt(vect orig, vect to) {
    Ray r = {
        .orig = orig,
        .dir = vunit(vsub(to, orig)),
        .hasHit = false,
        .tMax = INFINITY
    };
    return r;
}


#ifndef RAY_H
#define RAY_H

#include <stdbool.h>

#include "vector.h"
#include "color.h"
```

```c
struct HitInfo {
    int n[3];
    double u, v;
};

typedef struct HitInfo HitInfo;

struct Ray {
    vect orig;
    vect dir;

    bool hasHit;
    double tMax;
    HitInfo hitInfo;
};

typedef struct Ray Ray;

Ray pointAt(vect orig, vect to);

#endif
```

```c
#include <stdlib.h>
#include <string.h>
#include <stdbool.h>
#include <math.h>
#include <pthread.h>
#include <stdatomic.h>

#include "render.h"
#include "bvh.h"
#include "vector.h"
#include "ray.h"
#include "color.h"
#include "clock.h"
#include "misc.h"
#include "triangle.h"
```

```c
const color bgColor = {.r = 0.1, .g = 0.1, .b = 0.1};

color hueToRgb(double hue) { // 0 = Turquoise, >= 180 = Red
    if (hue >= 180) return cnew(1, 0, 0);
    else if (hue >= 120) return cnew(1, (180 - hue) / 60, 0);
    else if (hue >= 60) return cnew((hue - 60) / 60, 1, 0);
    else return cnew(0, 1, (60 - hue) / 60);
}

color rayColor(rtx* rt, renderOptions* opt, Ray* r, int* hits) {
    // Intersection
    if (opt->useBVH) {
        intersectBVH(rt->triangles, rt->bvh, r, hits);
    }
    else {
        for (int i = 0; i < rt->nT; i++) {
            triangleRayIntersection(rt->triangles[i], r);
        }
    }

    if (!r->hasHit) return bgColor;

    // Shading
    vect n0 = rt->normals[r->hitInfo.n[0]];
    vect n1 = rt->normals[r->hitInfo.n[1]];
    vect n2 = rt->normals[r->hitInfo.n[2]];
    double u = r->hitInfo.u;
    double v = r->hitInfo.v;

    vect n = vadd(vmult(n0, 1 - u - v), vadd(vmult(n1, u), vmult(n2, v)));
    double shade = fabsl(vdot(n, r->dir));
    return cnew(shade, shade, shade);
}

static void* renderWorker(void* args) {
    renderWorkerArgs data = *(renderWorkerArgs*)args;
    rtx* rt = data.rt;
    renderOptions* opt = data.options;
    color** image = data.image;
    color** heatmap = data.heatmap;

    int pixelTot = opt->imW * opt->imH;

    while (true) {
        int chunkId = atomic_fetch_add(data.nextChunk, 1);
        int start = chunkId * PIXEL_CHUNCK_SIZE;
        int end = min((chunkId + 1) * PIXEL_CHUNCK_SIZE, pixelTot);

        if (start >= end) break;

        for (int k = start; k < end; k++) {
            int i = k / opt->imW;
            int j = k % opt->imW;
            double u = (double) j / opt->imW;
            double v = (double) i / opt->imH;

            int hits;
            Ray r = pointAt(data.orig, vadd(data.corner, vadd(vmult(data.horiz, u), vmult(data.vert, v))));
            image[opt->imH - i - 1][j] = rayColor(data.rt, opt, &r, &hits);

            if (opt->heatmap) {
                heatmap[opt->imH - i - 1][j] = hueToRgb((double) hits * 180 / opt->redThreshold);
            }
        }
    }
}

void render(rtx* rt, renderOptions* opt, color** image, color** heatmap) {
    // Calculate a few useful vectors
    vect dir = vneg(vnew(sin(opt->theta) * sin(opt->phi), cos(opt->theta), sin(opt->theta) * cos(opt->phi)));
    vect orig = vadd(opt->focus, vmult(dir, -opt->r));
    vect horiz = vmult(vnew(cos(opt->phi), 0, -sin(opt->phi)), opt->vpW);
    vect vert = vneg(vmult(vnew(cos(opt->theta) * sin(opt->phi), -sin(opt->theta), cos(opt->theta) * cos(opt->phi)), opt->vpH)
    vect corner = vsub(vadd(orig, vmult(dir, opt->focL)), vadd(vdiv(horiz, 2), vdiv(vert, 2)));
    // Init render parallelization
    pthread_t* threads = malloc(N_THREADS * sizeof(pthread_t));

    atomic_int* nextChunk = malloc(sizeof(atomic_int));
    *nextChunk = 0;
    renderWorkerArgs args = { .rt = rt, .options = opt, .image = image, .heatmap = heatmap, .orig = orig, .corner = corner, .ho

    for (int i = 0; i < N_THREADS; i++) {
        pthread_create(&threads[i], NULL, renderWorker, &args);
    }
    for (int i = 0; i < N_THREADS; i++) {
        pthread_join(threads[i], NULL);
    }

    free(threads);
    free(nextChunk);
}


#ifndef RTX_H
#define RTX_H

#include "vector.h"
#include "color.h"
#include "ray.h"
#include "misc.h"
#include "triangle.h"

struct rtx {
    bool useSpatialSplits;
    double alpha;

    vect vertices[MAX_VERTICES];
    vect normals[MAX_NORMALS];
    triangle triangles[MAX_TRIANGLES];

    struct BVH* bvh;

    int nV, nT;
};

typedef struct rtx rtx;

struct renderOptions {
    vect focus;
    double r;
    double theta;
    double phi;

    int imW;
    int imH;
    double vpW;
    double vpH;
    double focL;

    bool useBVH;
    bool heatmap;
    int redThreshold;
};

typedef struct renderOptions renderOptions;

struct renderWorkerArgs {
    rtx* rt;
    renderOptions* options;
    color** image;
    color** heatmap;
    atomic_int* nextChunk;
    vect orig, corner, horiz, vert;
};

typedef struct renderWorkerArgs renderWorkerArgs;

void render(rtx* rt, renderOptions* opt, color** image, color** heatmap);

#endif
```

```c
#include <stdlib.h>
#include <math.h>
#include <assert.h>
#include <stdio.h>
#include <stdbool.h>

#include "vector.h"
#include "misc.h"

vect vnew(double x, double y, double z) {
    vect v = {.x = x, .y = y, .z = z};
        return v;
}

vect vmult(vect a, double k) {
    return vnew(a.x * k, a.y * k, a.z * k);
}

vect vdiv(vect a, double k) {
    assert(k != 0);
    return vmult(a, (double) 1 / k);
}

vect vneg(vect a) {
    return vnew(-a.x, -a.y, -a.z);
}

vect vadd(vect a, vect b) {
    return vnew(a.x + b.x, a.y + b.y, a.z + b.z);
}

vect vsub(vect a, vect b) {
    return vadd(a, vneg(b));
}

double vdot(vect a, vect b) {
    return a.x * b.x + a.y * b.y + a.z * b.z;
}
```

```c
vect vcross(vect a, vect b) {
    return vnew(a.y * b.z - a.z * b.y, a.z * b.x - a.x * b.z, a.x * b.y - a.y * b.x);
}

double vmag2(vect a) {
    return vdot(a, a);
}

double vmag(vect a) {
    return sqrtl(vmag2(a));
}

vect vunit(vect a) {
    return vdiv(a, vmag(a));
}

vect vmin(vect a, vect b) {
    return vnew(min(a.x, b.x), min(a.y, b.y), min(a.z, b.z));
}

vect vmax(vect a, vect b) {
    return vnew(max(a.x, b.x), max(a.y, b.y), max(a.z, b.z));
}

vect vinv(vect a) {
    assert(a.x != 0 || a.y != 0 || a.z != 0);
    return vnew(1 / a.x, 1 / a.y, 1 / a.z);
}

double vcomp(vect a, int comp) {
    if (comp == 0) return a.x;
    if (comp == 1) return a.y;
    if (comp == 2) return a.z;
    assert(false);
}

void print_vect(vect a) {
    printf("{%.6f, %.6f, %.6f}\n", a.x, a.y, a.z);
}

#ifndef VECTOR_H
#define VECTOR_H

#include <stdlib.h>
#include <math.h>
#include <assert.h>
#include <stdio.h>

struct vect {
    double x;
    double y;
    double z;
};

typedef struct vect vect;

vect vnew(double x, double y, double z);

vect vmult(vect a, double k);

vect vdiv(vect a, double k);

vect vneg(vect a);

vect vadd(vect a, vect b);

vect vsub(vect a, vect b);

double vdot(vect a, vect b);

vect vcross(vect a, vect b);

double vmag2(vect a);

double vmag(vect a);
```

```c
vect vunit(vect a);

vect vmin(vect a, vect b);

vect vmax(vect a, vect b);

vect vinv(vect a);

double vcomp(vect a, int comp);

void print_vect(vect a);

#endif
```