Solar Shading

And the analysis of building environments

De Vora Ariel - 23.08.23

Université de Strasbourg

cemosis
Centre de modélisation et de simulation de Strasbourg

iloomi

HiDALGO2
CENTRE OF EXCELLENCE

The Solar Shading Project

Goals :

Environment:

- Cut energy use by 10% by the close of 2024
- Reduce Greenhouse gas emissions (building sector representing 23% of the total outputs)

Application:

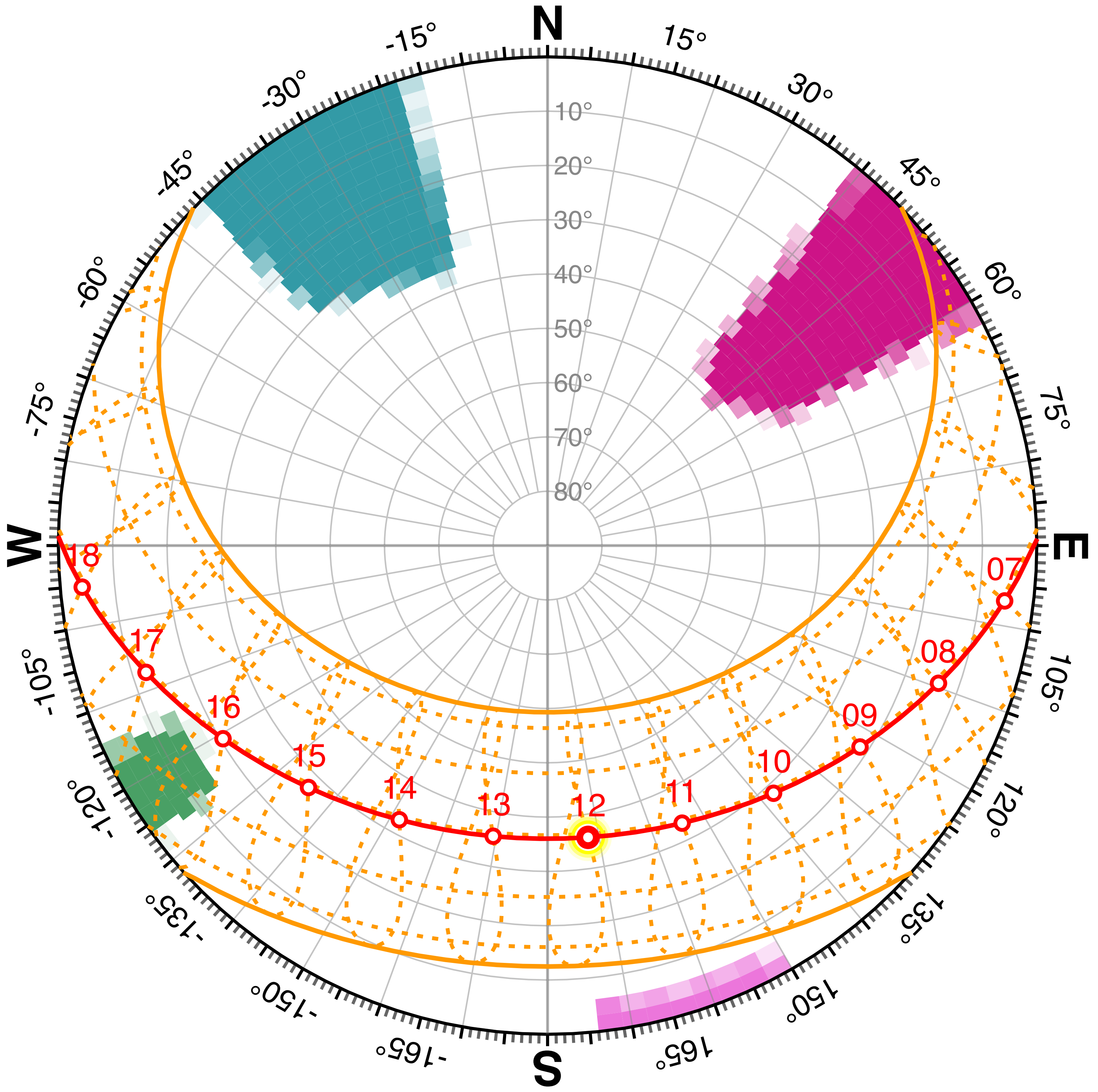
- Enable large scale computations
- Application with Accurate & Verified Results

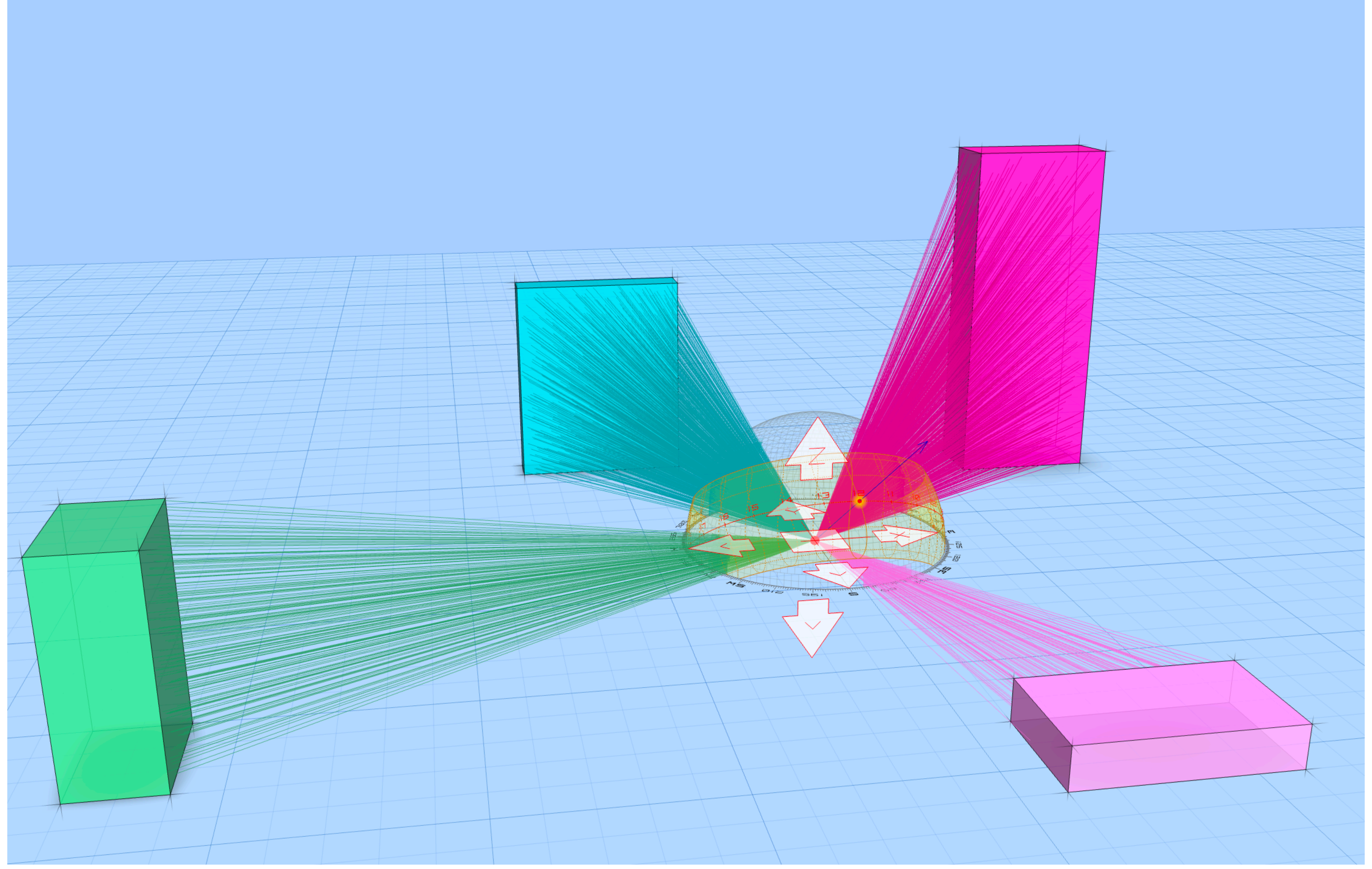
My Role and Objectives

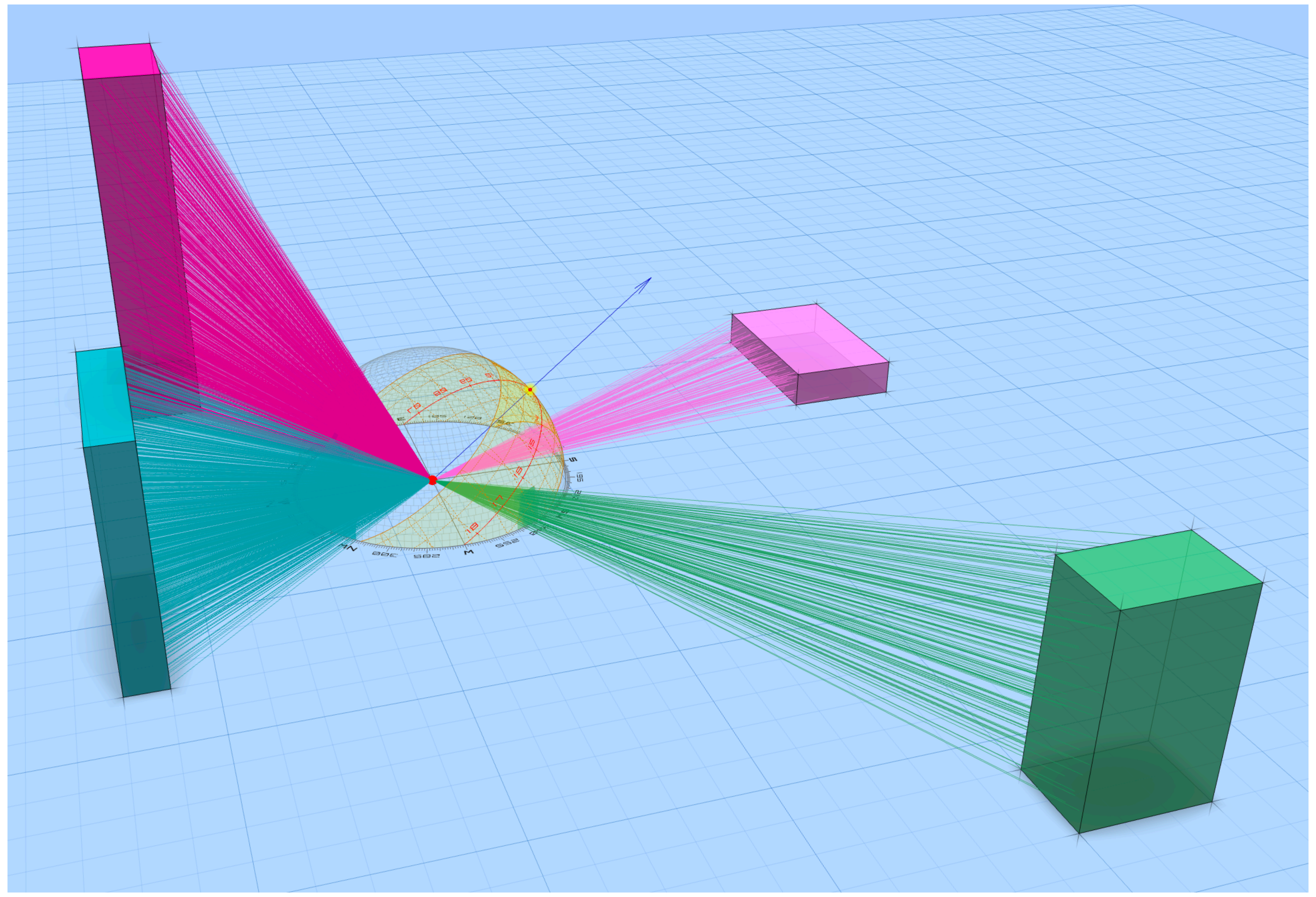
Goals

Optimization:

- Vectorization
- Benchmarking (Speed, qualityRNG, techniques)
- Ray Tracing Parallelization







My Role and Objectives

Goals

Implementation:

- Solar Radiation
- Automated Retrieval of Meteorological Data
- Perez all-Weather Sky Distribution Model
- GPU implementation

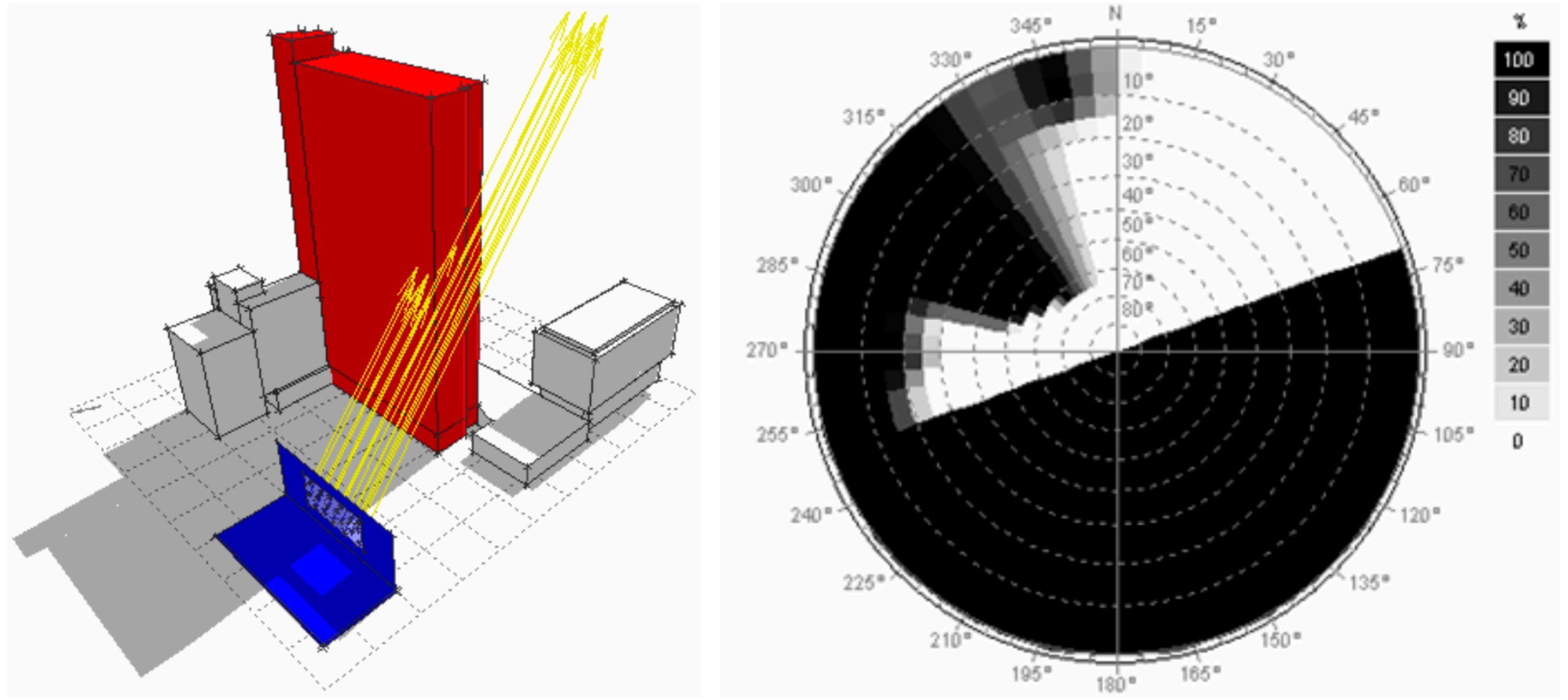


Figure 3 Shading masks for planar surfaces must store fractional shading information.

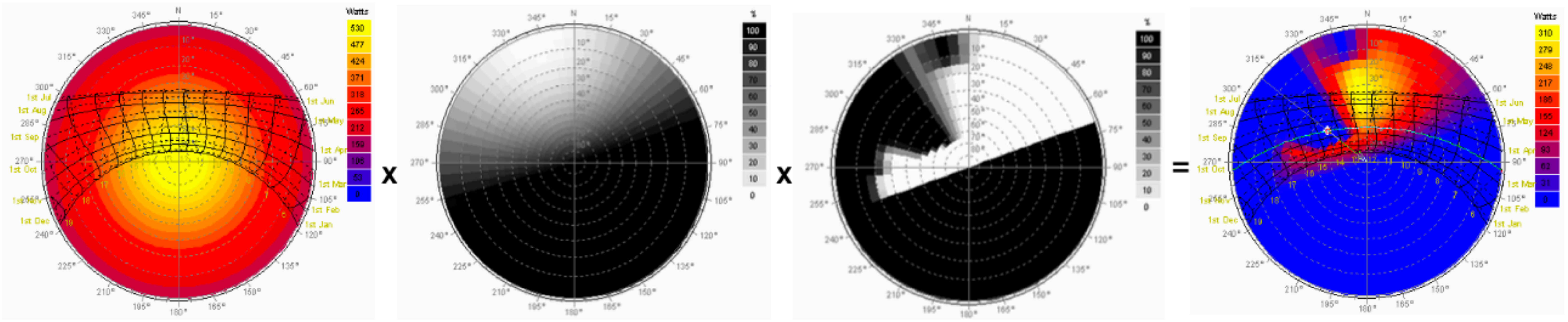


Figure 5 A diffuse radiation factor can be determined by multiplying the diffuse sky distribution by both incidence angle and shading factors for each sky segment.

Image from the article: The Application of Shading Masks in Building Applications

My Role and Objectives

Tools

Benchmarking:

- Shell
- Reframe
- GoogleTest

Optimization:

- **CUDA** (to HPC convertor)**, NVIDIA's OptiX API**
- **Compilation Flags**
- **Github for seemless integration in the Feel++ project**

Outline of the presentation

Mathematical Framework:

- **Monte Carlo Integration**
- **Implications of Random Number Generators**

Bounding Volume Hierarchy:

- **Construction** (Context & Previous Work)
- **Optimization**
- **Traversal** (Context & Previous Work)
- **Optimizations** (Speculative Traversal)

Shading Masks:

- **Context & Previous Work**
- **Optimizations**
- **Compilation Flags**

Perez Sky Distribution Model:

- **Implementations**
- **Warnings**
- **Optimizations**

Mathematical Framework

Monte Carlo Integration

If $\{\bar{x}_i\}_{i \in \mathbf{N}}$ are N samples chosen from the distribution $p(\bar{x})$

$$I \approx Q_N \approx \frac{1}{N} \sum_{i=1}^{N} \frac{f(\bar{x}_i)}{p(\bar{x}_i)}$$

But if the $\{\bar{x}_i\}_{i \in \mathbf{N}}$ are uniformly chosen

$$V = \int_{\Omega} d\bar{x}$$

$$I \approx Q_N \approx V \frac{1}{N} \sum_{i=1}^{N} f(\bar{x}_i)$$

Mathematical Framework

Monte Carlo Integration

Due to the law of large numbers

$$\lim_{N \to \infty} Q_N = I$$

Since

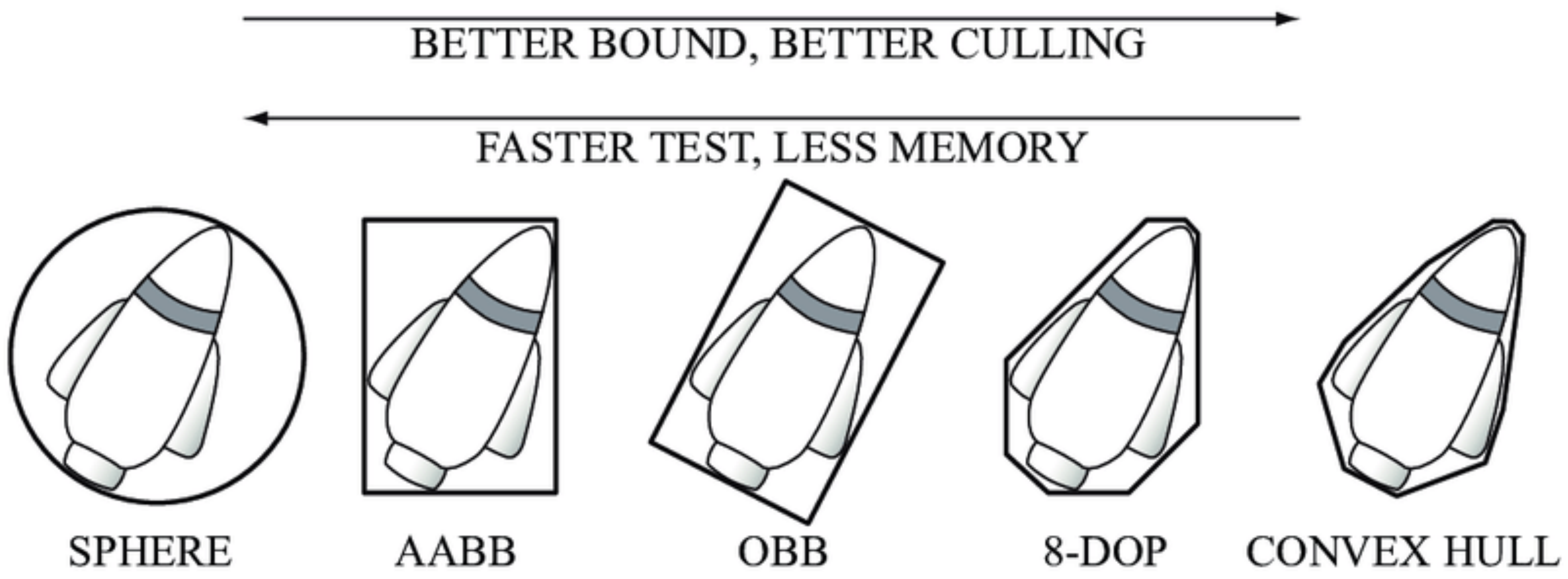
$$Var(f) \cong \sigma^2 = \frac{1}{N-1} \sum_{i=1}^{N} (f(\bar{x}_i) - \mu)^2$$

$$\lim_{N \to \infty} \sqrt{Var(Q_N)} = \lim_{N \to \infty} V \frac{\sigma_N}{\sqrt{N}} \to 0$$

As long as $\{\sigma_i^2\}$ is bounded

Bounding Volumes

Context / Previous Work




```cpp
for(auto &elt : elem)
{
    auto const& e = unwrap_ref( elt );
    Eigen::VectorXd M_bound_min(mesh->realDimension()),M_bound_max(mesh->realDimension());
    auto v1 = e.point(0).node();
    double* ptr_data = &v1[0];
    M_bound_min = Eigen::Map<Eigen::VectorXd, Eigen::Unaligned>(ptr_data, v1.size());
    M_bound_max = Eigen::Map<Eigen::VectorXd, Eigen::Unaligned>(ptr_data, v1.size());

    for(int j=1;j<e.nPoints();j++)
    {
        for(int k=0;k<mesh->realDimension();k++)
        {
            M_bound_min[k] = std::min(M_bound_min[k],e.point(j).node()[k]);
            M_bound_max[k] = std::max(M_bound_max[k],e.point(j).node()[k]);
        }
    }
    for(int k=0;k<mesh->realDimension();k++)
    {
        M_bound_min[k] = M_bound_min[k] - 2*FLT_MIN;
        M_bound_max[k] = M_bound_max[k] + 2*FLT_MIN;
    }
    BVHPrimitiveInfo primitive_i(e.id(),M_bound_min,M_bound_max);
    M_primitiveInfo.push_back(primitive_i);
    index_Primitive++;
}
```

Image from Real-Time Collision Detection

Bounding Volume Hierarchy

Context / Previous Work

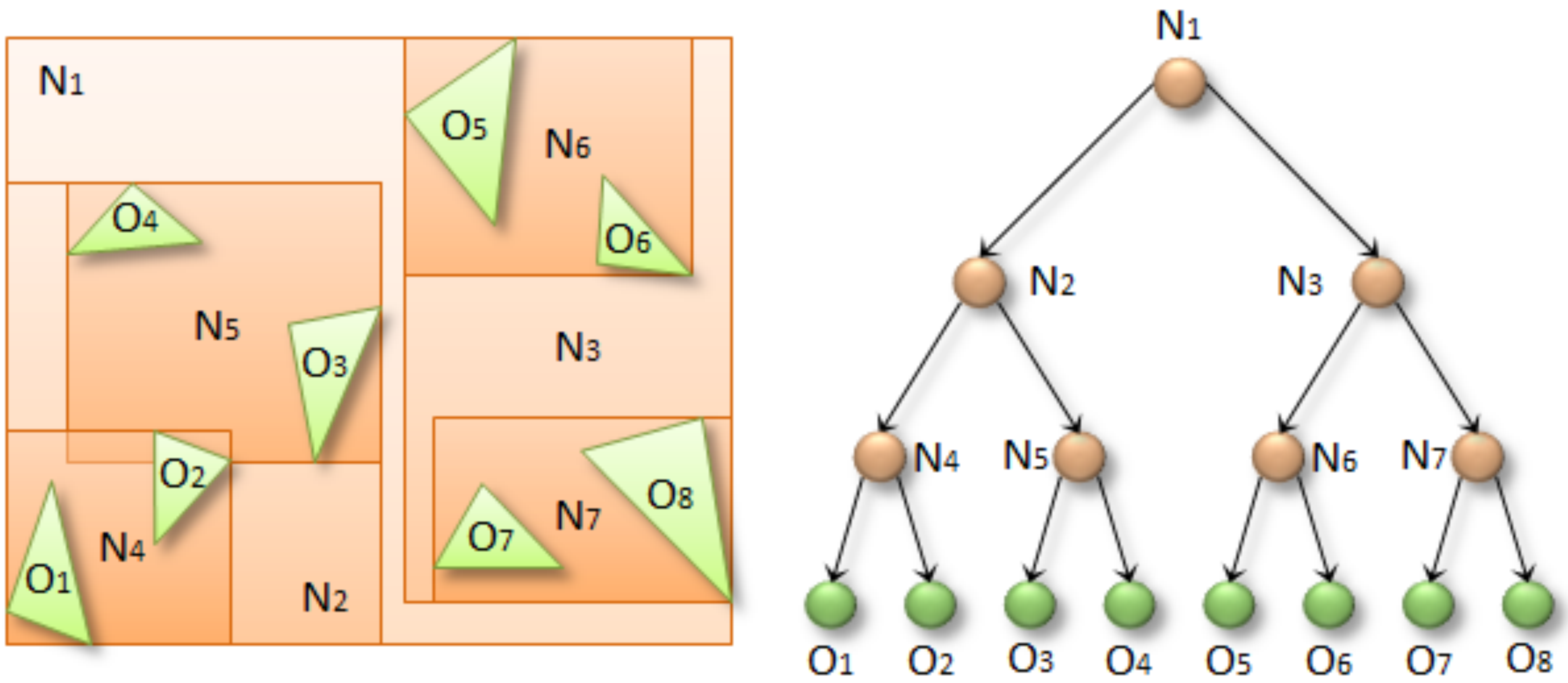


Image from NVIDIA's website

Bounding Volume Hierarchy

Context / Previous Work

Bounding Box covering all given Primitives

```cpp
for (int i = start_index_primitive+1; i < end_index_primitive; ++i)
{
    M_bound_min_node = node->newBoundsMin(M_bound_min_node,M_primitiveInfo[i].M_bound_min);
    M_bound_max_node = node->newBoundsMax(M_bound_max_node,M_primitiveInfo[i].M_bound_max);
}
```

```cpp
BVHNode * recursiveBuild(BVHNode * current_parent, int cut_dimension, int start_index_primitive, int end_index_primitive,
                        std::vector<int> &orderedPrims)
{
    LOG(INFO) <<fmt::format("cut dimension {}, start index primitive {}, end index primitive {}",
                            cut_dimension,start_index_primitive,end_index_primitive);
    Eigen::VectorXd M_bound_min_node(nDim),M_bound_max_node(nDim);
    BVHNode * node = new BVHTree::BVHNode();
    M_bound_min_node = M_primitiveInfo[start_index_primitive].M_bound_min;
    M_bound_max_node = M_primitiveInfo[start_index_primitive].M_bound_max;
    for (int i = start_index_primitive+1; i < end_index_primitive; ++i)
    {
        M_bound_min_node = node->newBoundsMin(M_bound_min_node,M_primitiveInfo[i].M_bound_min);
        M_bound_max_node = node->newBoundsMax(M_bound_max_node,M_primitiveInfo[i].M_bound_max);
    }
    auto mid = (start_index_primitive + end_index_primitive) / 2;
    std::nth_element(&M_primitiveInfo[start_index_primitive], &M_primitiveInfo[mid],
            &M_primitiveInfo[end_index_primitive-1]+1,
                [cut_dimension](const BVHPrimitiveInfo &a, const BVHPrimitiveInfo &b) {
                    return a.M_centroid[cut_dimension] < b.M_centroid[cut_dimension];
                });
    int nPrimitives = end_index_primitive - start_index_primitive;
    if (nPrimitives == 1)
    {
        // Create a leaf, since there is only one primitive in the list
        int firstPrimOffset = orderedPrims.size();
        for (int i = start_index_primitive; i < end_index_primitive; ++i)
        {
        int primNum = M_primitiveInfo[i].M_primitiveNumber;
        orderedPrims.push_back(primNum);
        }
        node->buildLeaf(current_parent,firstPrimOffset, nPrimitives, M_bound_min_node,M_bound_max_node);
        return node;
    }
    else{
        // Create a node, since there are at least two primitives in the list
        node->buildInternalNode(current_parent,(cut_dimension+1)%nDim,
                            recursiveBuild( node, (cut_dimension+1)%nDim, start_index_primitive, mid, orderedPrims),
                            recursiveBuild( node, (cut_dimension+1)%nDim, mid, end_index_primitive, orderedPrims));
    }

    return node;
}
```

Bounding Volume Hierarchy

Context / Previous Work

Partition of the Primitives in the specified cutting dimension

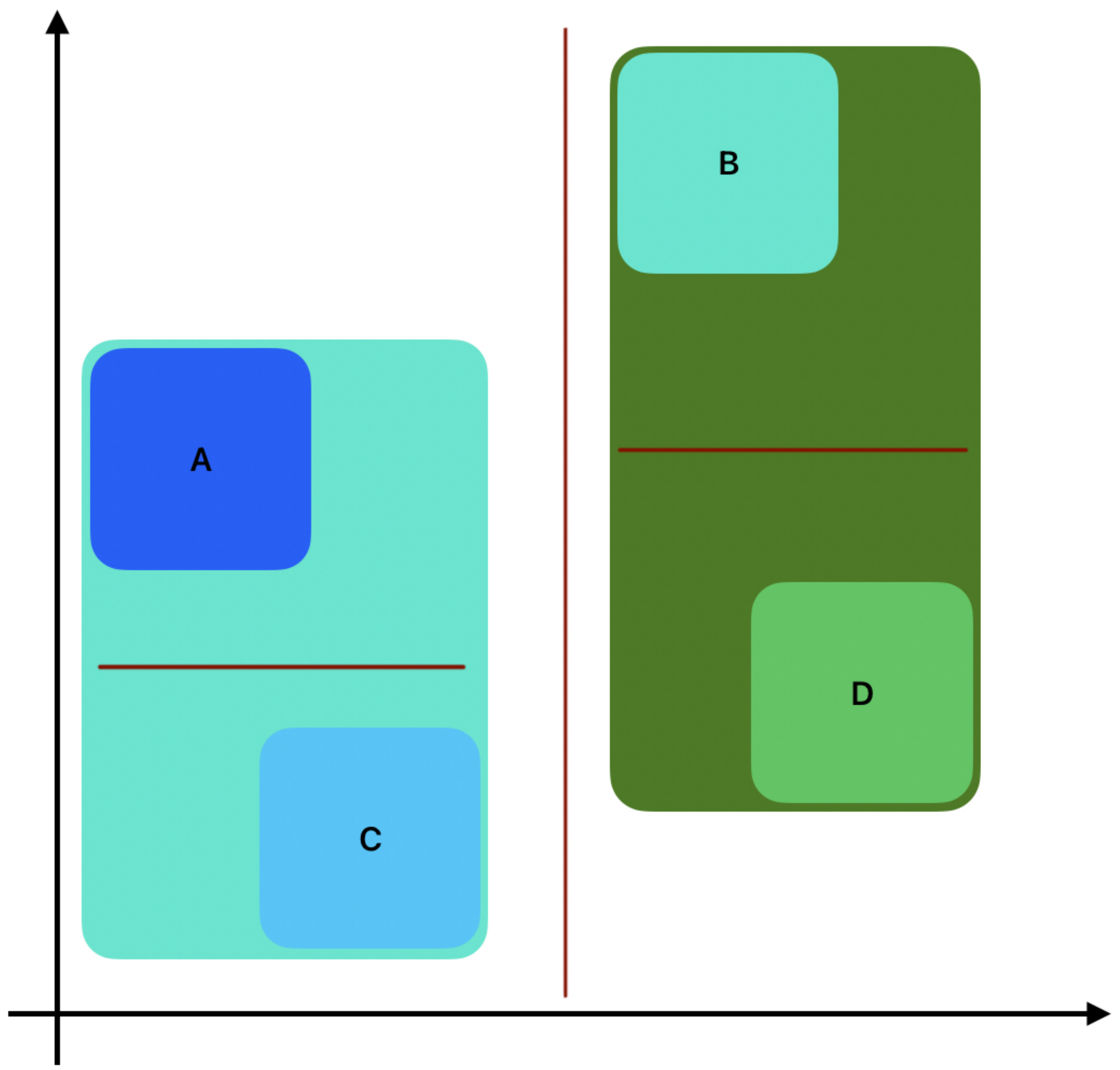
```
auto mid = (start_index_primitive + end_index_primitive) / 2;
std::nth_element(&M_primitiveInfo[start_index_primitive], &M_primitiveInfo[mid],
        &M_primitiveInfo[end_index_primitive-1]+1,
            [cut_dimension](const BVHPrimitiveInfo &a, const BVHPrimitiveInfo &b) {
                return a.M_centroid[cut_dimension] < b.M_centroid[cut_dimension];
            });
```

```
BVHNode * recursiveBuild(BVHNode * current_parent, int cut_dimension, int start_index_primitive, int end_index_primitive,
                        std::vector<int> &orderedPrims)
{
    LOG(INFO) <<fmt::format("cut dimension {}, start index primitive {}, end index primitive {}",
                        cut_dimension,start_index_primitive,end_index_primitive);
    Eigen::VectorXd M_bound_min_node(nDim),M_bound_max_node(nDim);
    BVHNode * node = new BVHTree::BVHNode();
    M_bound_min_node = M_primitiveInfo[start_index_primitive].M_bound_min;
    M_bound_max_node = M_primitiveInfo[start_index_primitive].M_bound_max;
    for (int i = start_index_primitive+1; i < end_index_primitive; ++i)
    {
        M_bound_min_node = node->newBoundsMin(M_bound_min_node,M_primitiveInfo[i].M_bound_min);
        M_bound_max_node = node->newBoundsMax(M_bound_max_node,M_primitiveInfo[i].M_bound_max);
    }
    auto mid = (start_index_primitive + end_index_primitive) / 2;
    std::nth_element(&M_primitiveInfo[start_index_primitive], &M_primitiveInfo[mid],
            &M_primitiveInfo[end_index_primitive-1]+1,
                [cut_dimension](const BVHPrimitiveInfo &a, const BVHPrimitiveInfo &b) {
                    return a.M_centroid[cut_dimension] < b.M_centroid[cut_dimension];
                });
    int nPrimitives = end_index_primitive - start_index_primitive;
    if (nPrimitives == 1)
    {
        // Create a leaf, since there is only one primitive in the list
        int firstPrimOffset = orderedPrims.size();
        for (int i = start_index_primitive; i < end_index_primitive; ++i)
        {
        int primNum = M_primitiveInfo[i].M_primitiveNumber;
        orderedPrims.push_back(primNum);
        }
        node->buildLeaf(current_parent,firstPrimOffset, nPrimitives, M_bound_min_node,M_bound_max_node);
        return node;
    }
    else{
        // Create a node, since there are at least two primitives in the list
        node->buildInternalNode(current_parent,(cut_dimension+1)%nDim,
                    recursiveBuild( node, (cut_dimension+1)%nDim, start_index_primitive, mid, orderedPrims),
                    recursiveBuild( node, (cut_dimension+1)%nDim, mid, end_index_primitive, orderedPrims));
    }

    return node;
}
```

Bounding Volume Hierarchy

Context / Previous Work

Partition of the Primitives in the specified cutting dimension




```cpp
BVHNode * recursiveBuild(BVHNode * current_parent, int cut_dimension, int start_index_primitive, int end_index_primitive,
                        std::vector<int> &orderedPrims)
{
    LOG(INFO) <<fmt::format("cut dimension {}, start index primitive {}, end index primitive {}",
                            cut_dimension,start_index_primitive,end_index_primitive);
    Eigen::VectorXd M_bound_min_node(nDim),M_bound_max_node(nDim);
    BVHNode * node = new BVHTree::BVHNode();
    M_bound_min_node = M_primitiveInfo[start_index_primitive].M_bound_min;
    M_bound_max_node = M_primitiveInfo[start_index_primitive].M_bound_max;
    for (int i = start_index_primitive+1; i < end_index_primitive; ++i)
    {
        M_bound_min_node = node->newBoundsMin(M_bound_min_node,M_primitiveInfo[i].M_bound_min);
        M_bound_max_node = node->newBoundsMax(M_bound_max_node,M_primitiveInfo[i].M_bound_max);
    }
    auto mid = (start_index_primitive + end_index_primitive) / 2;
    std::nth_element(&M_primitiveInfo[start_index_primitive], &M_primitiveInfo[mid],
            &M_primitiveInfo[end_index_primitive-1]+1,
                [cut_dimension](const BVHPrimitiveInfo &a, const BVHPrimitiveInfo &b) {
                    return a.M_centroid[cut_dimension] < b.M_centroid[cut_dimension];
                });
    int nPrimitives = end_index_primitive - start_index_primitive;
    if (nPrimitives == 1)
    {
        // Create a leaf, since there is only one primitive in the list
        int firstPrimOffset = orderedPrims.size();
        for (int i = start_index_primitive; i < end_index_primitive; ++i)
        {
        int primNum = M_primitiveInfo[i].M_primitiveNumber;
        orderedPrims.push_back(primNum);
        }
        node->buildLeaf(current_parent,firstPrimOffset, nPrimitives, M_bound_min_node,M_bound_max_node);
        return node;
    }
    else{
        // Create a node, since there are at least two primitives in the list
        node->buildInternalNode(current_parent,(cut_dimension+1)%nDim,
                            recursiveBuild( node, (cut_dimension+1)%nDim, start_index_primitive, mid, orderedPrims),
                            recursiveBuild( node, (cut_dimension+1)%nDim, mid, end_index_primitive, orderedPrims));
    }

    return node;
}
```

Image from NVIDIA's website

Bounding Volume Hierarchy

Context / Previous Work

Creation of a leaf, or split into clusters

```cpp
int nPrimitives = end_index_primitive - start_index_primitive;
if (nPrimitives == 1)
{
    // Create a leaf, since there is only one primitive in the list
    int firstPrimOffset = orderedPrims.size();
    for (int i = start_index_primitive; i < end_index_primitive; ++i)
    {
    int primNum = M_primitiveInfo[i].M_primitiveNumber;
    orderedPrims.push_back(primNum);
    }
    node->buildLeaf(current_parent,firstPrimOffset, nPrimitives, M_bound_min_node,M_bound_max_node);
    return node;
}
else{
    // Create a node, since there are at least two primitives in the list
    node->buildInternalNode(current_parent,(cut_dimension+1)%nDim,
                        recursiveBuild( node, (cut_dimension+1)%nDim, start_index_primitive, mid, orderedPrims),
                        recursiveBuild( node, (cut_dimension+1)%nDim, mid, end_index_primitive, orderedPrims));
}
```

```cpp
BVHNode * recursiveBuild(BVHNode * current_parent, int cut_dimension, int start_index_primitive, int end_index_primitive,
                        std::vector<int> &orderedPrims)
{
    LOG(INFO) <<fmt::format("cut dimension {}, start index primitive {}, end index primitive {}",
                        cut_dimension,start_index_primitive,end_index_primitive);
    Eigen::VectorXd M_bound_min_node(nDim),M_bound_max_node(nDim);
    BVHNode * node = new BVHTree::BVHNode();
    M_bound_min_node = M_primitiveInfo[start_index_primitive].M_bound_min;
    M_bound_max_node = M_primitiveInfo[start_index_primitive].M_bound_max;
    for (int i = start_index_primitive+1; i < end_index_primitive; ++i)
    {
        M_bound_min_node = node->newBoundsMin(M_bound_min_node,M_primitiveInfo[i].M_bound_min);
        M_bound_max_node = node->newBoundsMax(M_bound_max_node,M_primitiveInfo[i].M_bound_max);
    }
    auto mid = (start_index_primitive + end_index_primitive) / 2;
    std::nth_element(&M_primitiveInfo[start_index_primitive], &M_primitiveInfo[mid],
        &M_primitiveInfo[end_index_primitive-1]+1,
            [cut_dimension](const BVHPrimitiveInfo &a, const BVHPrimitiveInfo &b) {
                return a.M_centroid[cut_dimension] < b.M_centroid[cut_dimension];
            });
    int nPrimitives = end_index_primitive - start_index_primitive;
    if (nPrimitives == 1)
    {
        // Create a leaf, since there is only one primitive in the list
        int firstPrimOffset = orderedPrims.size();
        for (int i = start_index_primitive; i < end_index_primitive; ++i)
        {
        int primNum = M_primitiveInfo[i].M_primitiveNumber;
        orderedPrims.push_back(primNum);
        }
        node->buildLeaf(current_parent,firstPrimOffset, nPrimitives, M_bound_min_node,M_bound_max_node);
        return node;
    }
    else{
        // Create a node, since there are at least two primitives in the list
        node->buildInternalNode(current_parent,(cut_dimension+1)%nDim,
                            recursiveBuild( node, (cut_dimension+1)%nDim, start_index_primitive, mid, orderedPrims),
                            recursiveBuild( node, (cut_dimension+1)%nDim, mid, end_index_primitive, orderedPrims));
    }

    return node;
}
```

Bounding Volume Hierarchy

Optimizations

Agglomerative Clustering

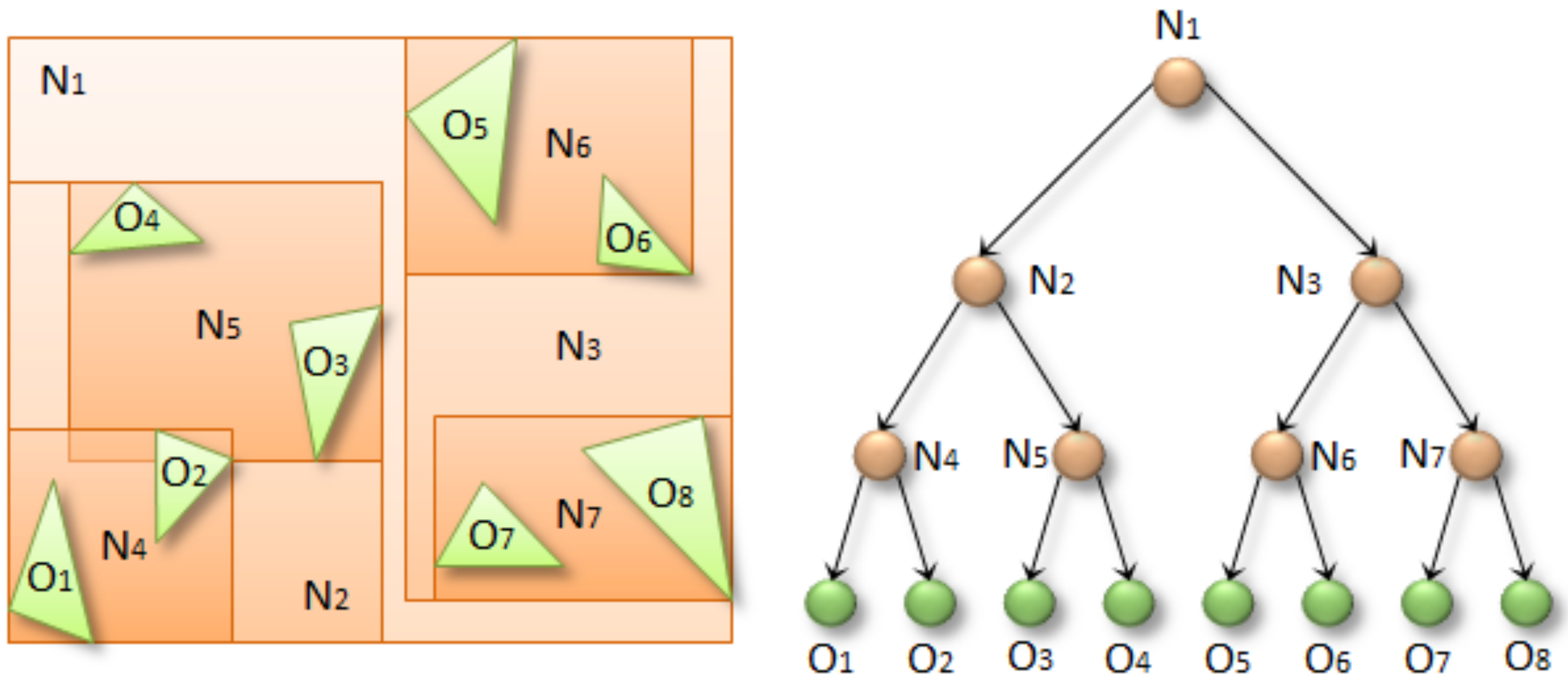


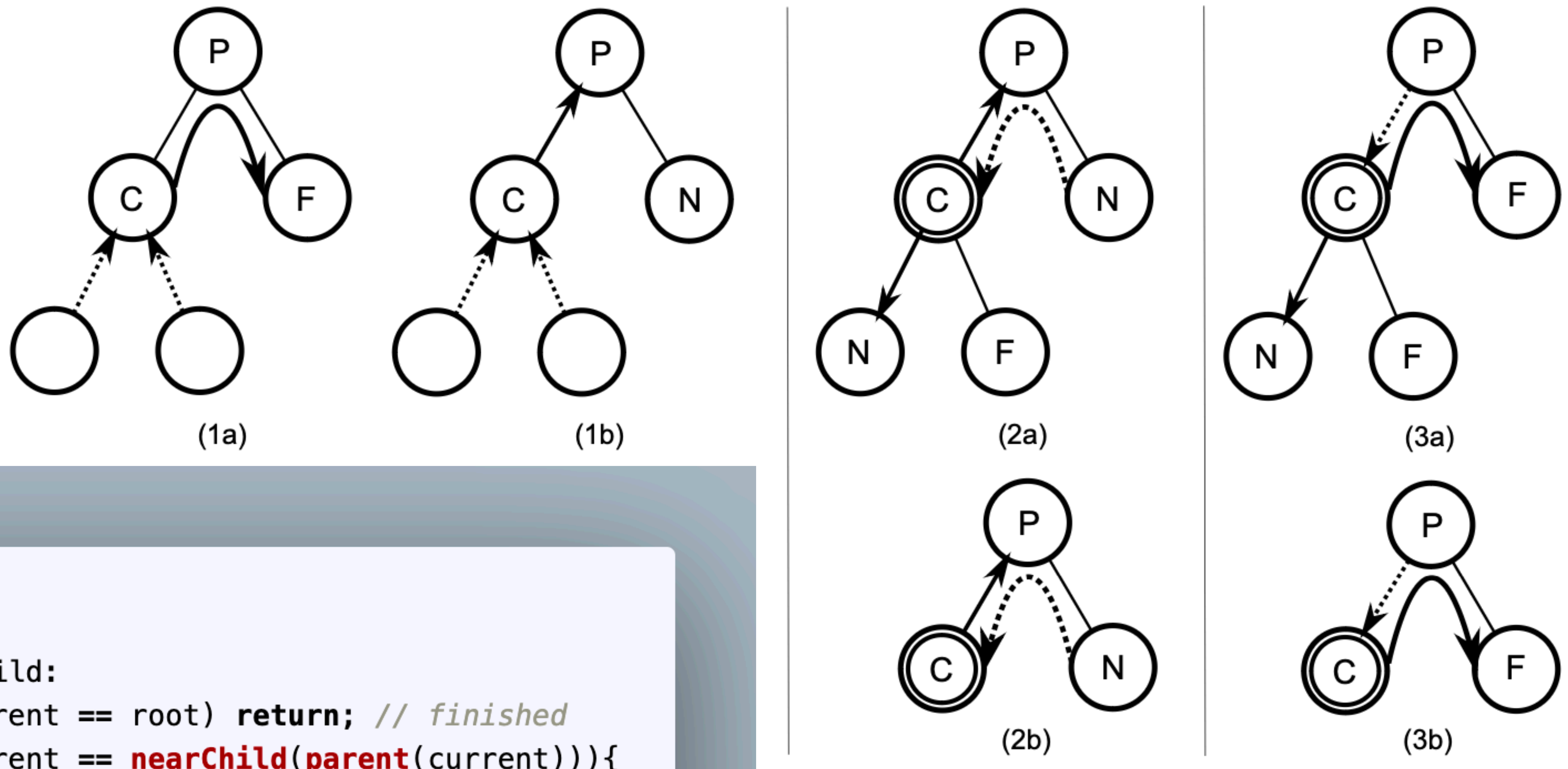
Image from NVIDIA's website

```cpp
template<int nDim>
typename BVHTree<nDim>::BVHNode* BVHTree<nDim>::agglomerativeBuild(std::list<BVHNode*>& nodes,
                                                                    std::vector<int>& orderedPrims)
{
    // Stop condition: returns the root node when there is only one node left
    if (nodes.size() == 1) {
        return nodes.front();
    }

    BVHNode* node1 = nullptr;
    BVHNode* node2 = nullptr;
    double minDistance = std::numeric_limits<double>::max();

    // Find the pair of nodes with the smallest centroid distance
    for (auto it1 = nodes.begin(); it1 != nodes.end(); ++it1) {
        for (auto it2 = std::next(it1); it2 != nodes.end(); ++it2) {
            double dist = ((*it1)->M_centroid - (*it2)->M_centroid).norm();
            if (dist < minDistance) {
                minDistance = dist;
                node1 = *it1;
                node2 = *it2;
            }
        }
    }

    // Create a new internal node that groups the closest pair of nodes
    BVHNode* newCluster = new BVHTree::BVHNode();
    newCluster->aggbuildInternalNode(nullptr, node1, node2);

    // Remove the merged nodes and add the new cluster to the list
    nodes.remove(node1);
    nodes.remove(node2);
    nodes.push_back(newCluster);

    // Recursively continue the clustering
    return agglomerativeBuild(nodes, orderedPrims);
}
```

Bounding Volume Hierarchy

Context / Previous Work

```
case fromChild:
    if (current == root) return; // finished
    if (current == nearChild(parent(current))){
        current = sibling(current);
        state = fromSibling; // (1a)
    }
    else {
        current = parent(current);
        state = fromChild; // (1b)
    }
    break;
```



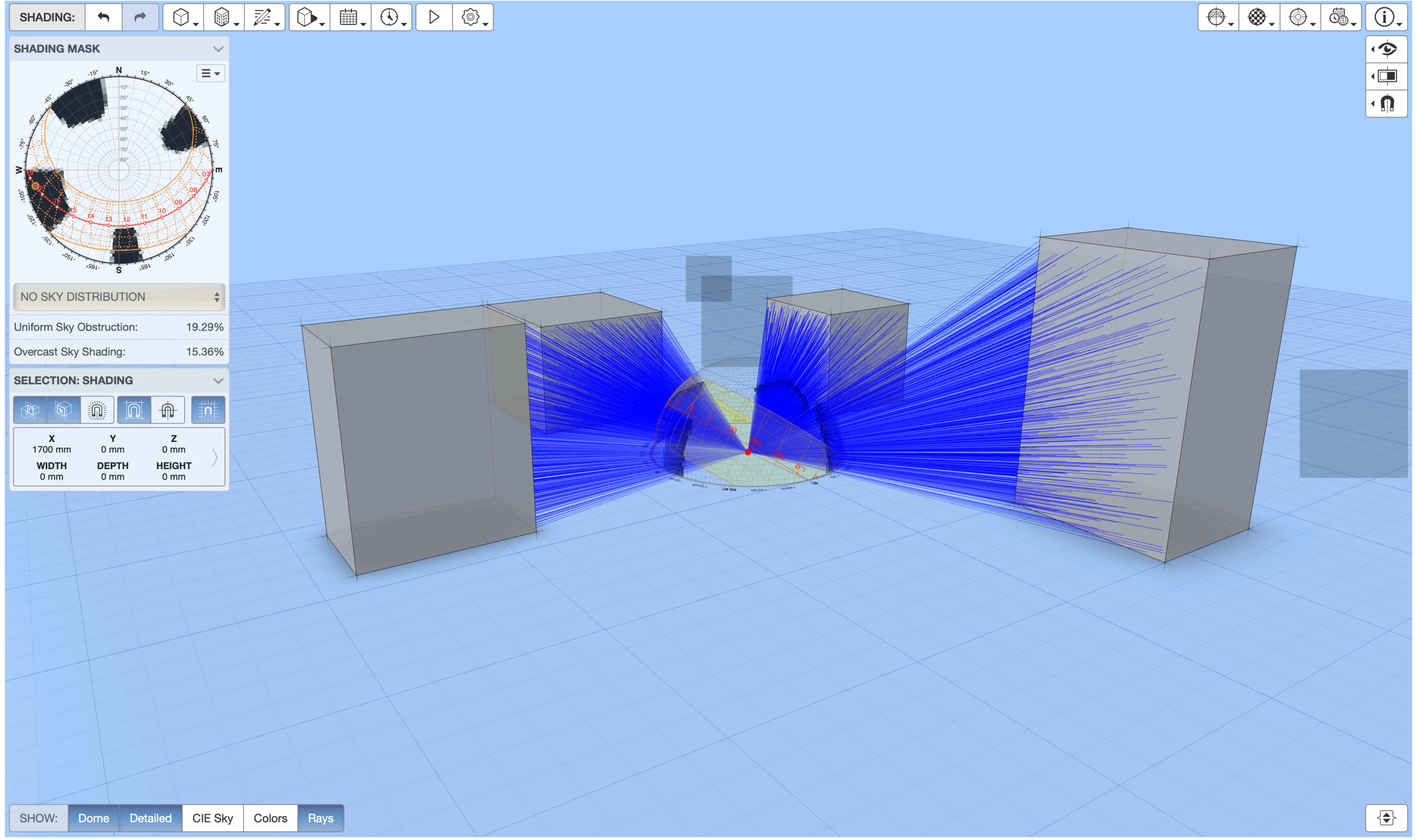

```
case fromParent:
    if (boxtest(ray, current) == MISSED) {
        current = sibling(current);
        sttate. = fromSibling; // (3a)
    }
    else if (isLeaf(current)) {
        // ray-primitive intersection tests
        processLeaf(current);
        current = sibling(current);
        state = fromSibling; // (3b)
    }
    else {
        current = nearChild(current);
        state = fromParent; // (3a)
    }
    break;
```

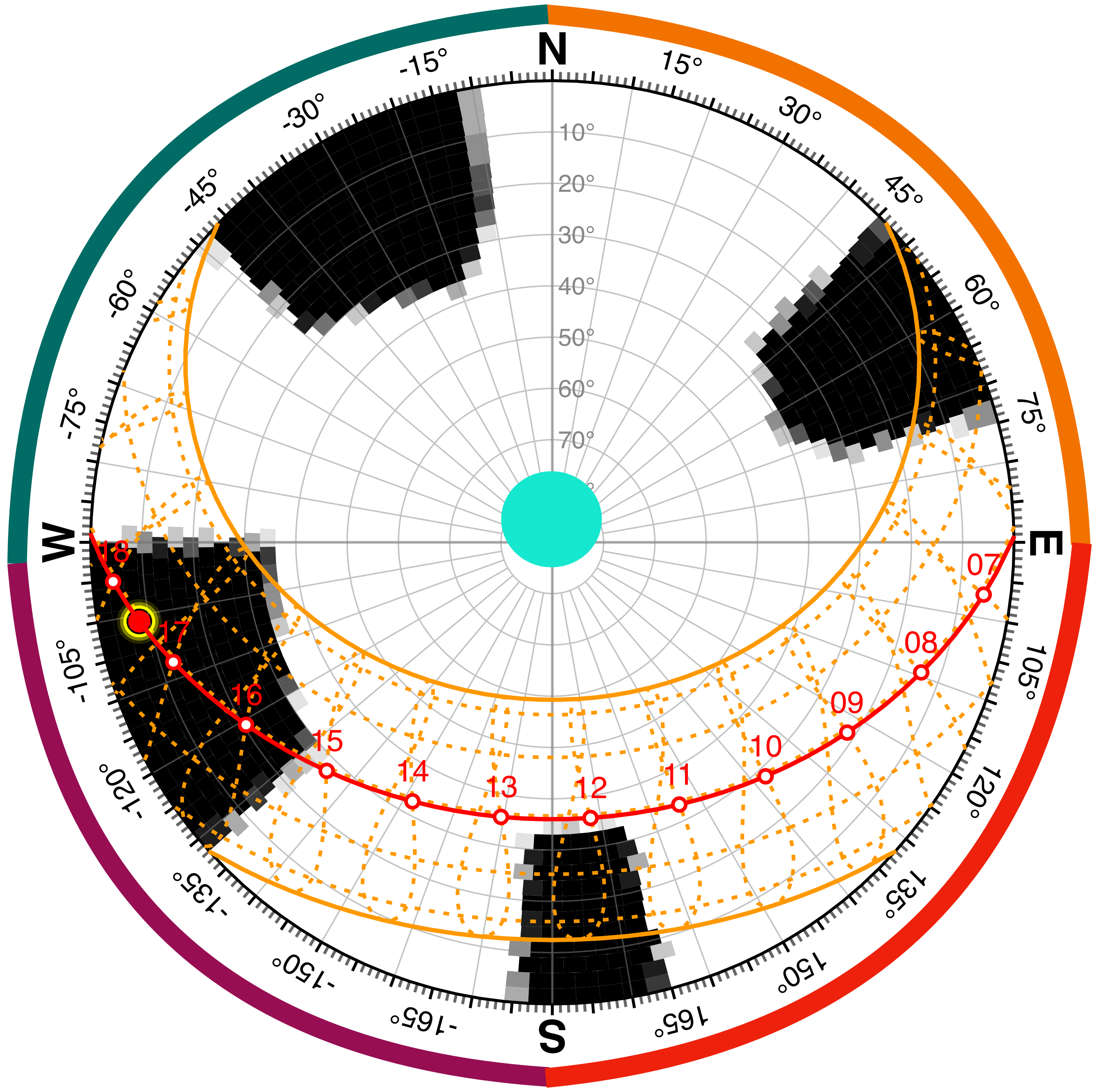
```
case fromSibling:
    if (boxtest(ray, current) == MISSED) {
        current = parent(current);
        state = fromChild; // (2a)
    }
    else if (isLeaf(current)) {
        // ray-primite intersection tests
        processLeaf(ray, current);
        current = parent(current);
        state = fromChild; // (2b)
    }
    else {
        current = nearChild(current);
        state = fromParent; //2a
    }
    break;
```

Image and Pseudo-Code from Efficient Stack-less BVH Traversal for Ray Tracing

Shading Masks

Context / Previous Work





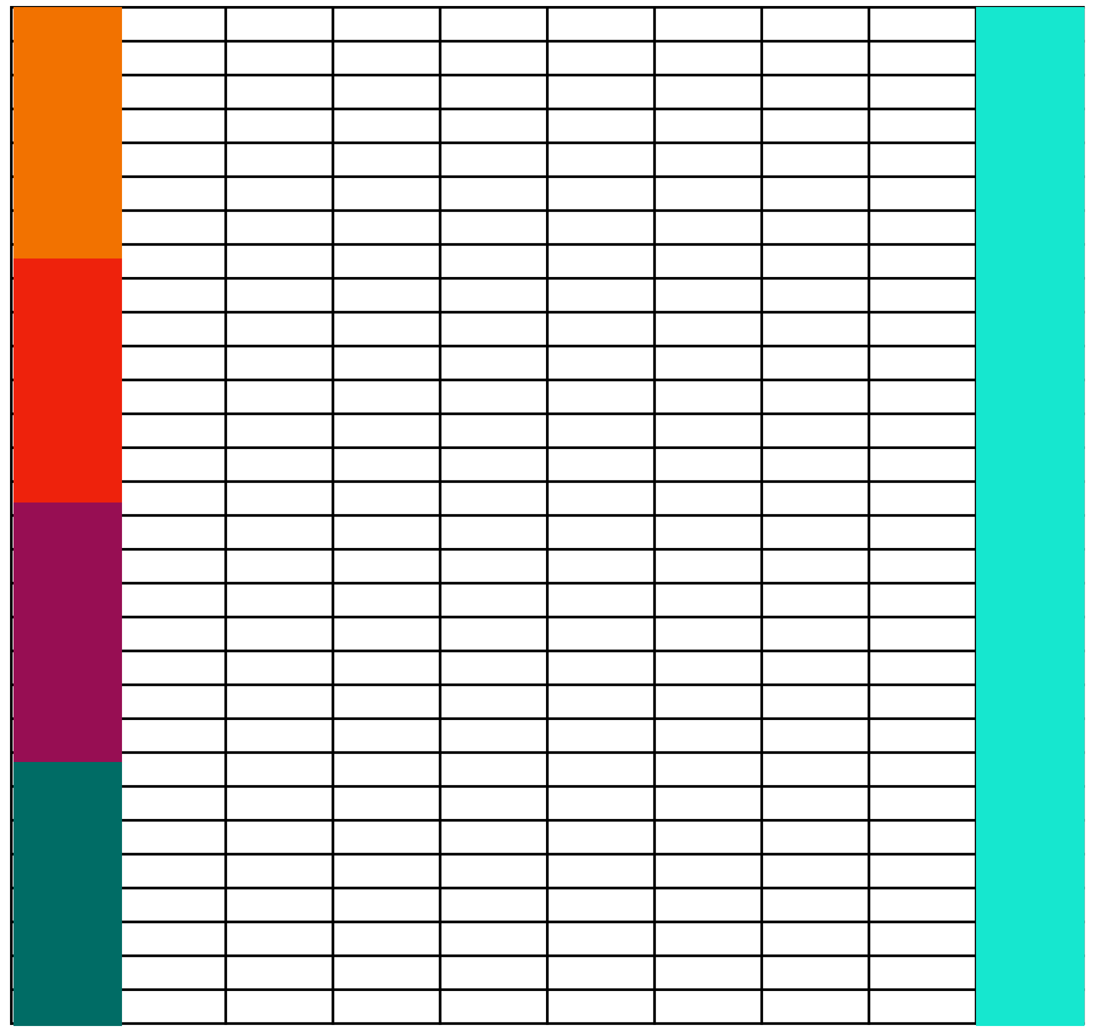


Image from Andrew Marsh's WebApp

Shading Masks

Context / Previous Work

Creation of a ray

```cpp
auto random_origin = get_random_point(el.second.vertices());

Eigen::VectorXd rand_dir(dim);
Eigen::VectorXd p1(dim),p2(dim),p3(dim),origin(3);
bool inward_ray=false;
if(dim==3)
{
    for(int i=0;i<dim;i++)
    {
        p1(i)=column(el.second.vertices(), 0)[i];
        p2(i)=column(el.second.vertices(), 1)[i];
        p3(i)=column(el.second.vertices(), 2)[i];
        origin(i) = random_origin[i];
        rand_dir(i) = random_direction[i];
    }
    auto element_normal = ((p3-p1).head<3>()).cross((p2-p1).head<3>());
    element_normal.normalize();

    // Choose the direction randomly among the latitude and azimuth
    getRandomDirectionSM(random_direction,M_gen,M_gen2,index_azimuth,index_altitude);
    for(int i=0;i<dim;i++)
    {
        rand_dir(i) = random_direction[i];
    }
    if(rand_dir.dot(element_normal)>=0)
    {
        inward_ray=true;
    }

}

BVHRay ray(origin,rand_dir);
```

```cpp
for(auto  [marker,marker_id] : M_submeshes[building_name]->markerNames())
{

    SM_table_marker.setZero();
    Angle_table_marker.setZero();
    auto ray_submesh = createSubmesh(_mesh=M_submeshes[building_name],_range=markedelements(M_submeshes[building_name],marker));

    // Launch Nrays from each triangle of each marker
    for(auto const &el : ray_submesh->elements() ) // from each element of the submesh, launch M_Nrays randomly oriented
    {

            auto rays_from_element = [&,marker=marker](int n_rays_thread){

                Eigen::MatrixXd SM_table(M_azimuthSize,M_altitudeSize);
                SM_table.setZero();

                Eigen::MatrixXd Angle_table(M_azimuthSize,M_altitudeSize);
                Angle_table.setZero();

                int index_altitude;
                int index_azimuth;
                for(int i=0;i<n_rays_thread;i++)
                {

                    // Construct the ray emitting from a random point of the element
                    auto random_origin = get_random_point(el.second.vertices());

                    Eigen::VectorXd rand_dir(dim);
                    Eigen::VectorXd p1(dim),p2(dim),p3(dim),origin(3);
                    bool inward_ray=false;
                    if(dim==3)
                    {
                        for(int i=0;i<dim;i++)
                        {
                            p1(i)=column(el.second.vertices(), 0)[i];
                            p2(i)=column(el.second.vertices(), 1)[i];
                            p3(i)=column(el.second.vertices(), 2)[i];
                            origin(i) = random_origin[i];
                            rand_dir(i) = random_direction[i];
                        }
                        auto element_normal = ((p3-p1).head<3>()).cross((p2-p1).head<3>());
                        element_normal.normalize();

                        // Choose the direction randomly among the latitude and azimuth
                        getRandomDirectionSM(random_direction,M_gen,M_gen2,index_azimuth,index_altitude);
                        for(int i=0;i<dim;i++)
                        {
                            rand_dir(i) = random_direction[i];
                        }
                        if(rand_dir.dot(element_normal)>=0)
                        {
                            inward_ray=true;
                        }

                    }

                    BVHRay ray(origin,rand_dir);

                    int closer_intersection_element = -1;
                    if(inward_ray)
                    {
                        closer_intersection_element = 1;
                    }
                    else
                    {
                        for(auto& [building_name,bvh_building_tree] : M_bvh_tree_vector)
                        {
                            closer_intersection_element = bvh_building_tree.raySearch(ray,"") ;
                            if (closer_intersection_element >=0 )
                                break;
                        }
                    }
                    // If there is an intersection, increase the shading mask table entry by 1 and augment the angle table by 1 as well
                    if ( closer_intersection_element >=0 )
                    {
                        SM_table(index_azimuth,index_altitude)++;
                        Angle_table(index_azimuth,index_altitude)++;
                    }
                    else
                    {
                        Angle_table(index_azimuth,index_altitude)++;
                    }
                }

                return std::make_pair(SM_table,Angle_table);
            };

        // Execute the lambda function on multiple threads using
        // std::async and std::future to collect the results
        std::vector<int> n_rays_thread;
        n_rays_thread.push_back(M_Nrays - (M_Nthreads-1) * (int)(M_Nrays / M_Nthreads));
        for(int t= 1; t < M_Nthreads; ++t){
            n_rays_thread.push_back( M_Nrays / M_Nthreads);
        }

        // Used to store the future results
        std::vector< std::future< std::pair<Eigen::MatrixXd, Eigen::MatrixXd > > > futures;

        for(int t = 0; t < M_Nthreads; ++t){

            // Start a new asynchronous task
            futures.emplace_back(std::async(std::launch::async, rays_from_element, n_rays_thread[t]));
        }

        for( auto& f : futures){
            // Wait for the result to be ready
            auto two_tables =  f.get();

            // Add the tables obtained in threads
            SM_table_marker +=two_tables.first;
            Angle_table_marker += two_tables.second;

        }
    }
    // Divide the shading mask by the corresponding value of the angle table
    // If an angle combination has not been selected, suppose there is no shadow
    auto shadingMatrix = SM_table_marker.array().binaryExpr( Angle_table_marker.array() , [](auto x, auto y) { return y==0 ? 0 : x/y; });

    // Shading mask value 0 means that the surface is not shadowed, value 1 it is fully shadowed
    // Save the shading mask table to a csv file
    saveShadingMask(building_name,marker,shadingMatrix.matrix());

}
```

Shading Masks

Context / Previous Work

Intersection Test

```
int closer_intersection_element = -1;
if(inward_ray)
{
    closer_intersection_element = 1;
}
else
{
    for(auto& [building_name,bvh_building_tree] : M_bvh_tree_vector)
    {
        closer_intersection_element = bvh_building_tree.raySearch(ray,"") ;
        if (closer_intersection_element >=0 )
            break;
    }
}
// If there is an intersection, increase the shading mask table entry by 1 and augment the angle table by 1 as well
if ( closer_intersection_element >=0 )
{
    SM_table(index_azimuth,index_altitude)++;
    Angle_table(index_azimuth,index_altitude)++;
}
else
{
    Angle_table(index_azimuth,index_altitude)++;
}
```

```
for(auto  [marker,marker_id] : M_submeshes[building_name]->markerNames())
{

    SM_table_marker.setZero();
    Angle_table_marker.setZero();
    auto ray_submesh = createSubmesh(_mesh=M_submeshes[building_name],_range=markedelements(M_submeshes[building_name],marker));

    // Launch Nrays from each triangle of each marker
    for(auto const &el : ray_submesh->elements() ) // from each element of the submesh, launch M_Nrays randomly oriented
    {

            auto rays_from_element = [&,marker=marker](int n_rays_thread){

                Eigen::MatrixXd SM_table(M_azimuthSize,M_altitudeSize);
                SM_table.setZero();

                Eigen::MatrixXd Angle_table(M_azimuthSize,M_altitudeSize);
                Angle_table.setZero();

                int index_altitude;
                int index_azimuth;
                for(int i=0;i<n_rays_thread;i++)
                {

                    // Construct the ray emitting from a random point of the element
                    auto random_origin = get_random_point(el.second.vertices());

                    Eigen::VectorXd rand_dir(dim);
                    Eigen::VectorXd p1(dim),p2(dim),p3(dim),origin(3);
                    bool inward_ray=false;
                    if(dim==3)
                    {
                        for(int i=0;i<dim;i++)
                        {
                            p1(i)=column(el.second.vertices(), 0)[i];
                            p2(i)=column(el.second.vertices(), 1)[i];
                            p3(i)=column(el.second.vertices(), 2)[i];
                            origin(i) = random_origin[i];
                            rand_dir(i) = random_direction[i];
                        }
                        auto element_normal = ((p3-p1).head<3>()).cross((p2-p1).head<3>());
                        element_normal.normalize();

                        // Choose the direction randomly among the latitude and azimuth
                        getRandomDirectionSM(random_direction,M_gen,M_gen2,index_azimuth,index_altitude);
                        for(int i=0;i<dim;i++)
                        {
                            rand_dir(i) = random_direction[i];
                        }
                        if(rand_dir.dot(element_normal)>=0)
                        {
                            inward_ray=true;
                        }

                    }

                    BVHRay ray(origin,rand_dir);

                    int closer_intersection_element = -1;
                    if(inward_ray)
                    {
                        closer_intersection_element = 1;
                    }
                    else
                    {
                        for(auto& [building_name,bvh_building_tree] : M_bvh_tree_vector)
                        {
                            closer_intersection_element = bvh_building_tree.raySearch(ray,"") ;
                            if (closer_intersection_element >=0 )
                                break;
                        }
                    }
                    // If there is an intersection, increase the shading mask table entry by 1 and augment the angle table by 1 as well
                    if ( closer_intersection_element >=0 )
                    {
                        SM_table(index_azimuth,index_altitude)++;
                        Angle_table(index_azimuth,index_altitude)++;
                    }
                    else
                    {
                        Angle_table(index_azimuth,index_altitude)++;
                    }
                }

                return std::make_pair(SM_table,Angle_table);
            };

        // Execute the lambda function on multiple threads using
        // std::async and std::future to collect the results
        std::vector<int> n_rays_thread;
        n_rays_thread.push_back(M_Nrays - (M_Nthreads-1) * (int)(M_Nrays / M_Nthreads));
        for(int t= 1; t < M_Nthreads; ++t){
            n_rays_thread.push_back( M_Nrays / M_Nthreads);
        }

        // Used to store the future results
        std::vector< std::future< std::pair<Eigen::MatrixXd, Eigen::MatrixXd > > > futures;

        for(int t = 0; t < M_Nthreads; ++t){

            // Start a new asynchronous task
            futures.emplace_back(std::async(std::launch::async, rays_from_element, n_rays_thread[t]));
        }

        for( auto& f : futures){
            // Wait for the result to be ready
            auto two_tables =  f.get();

            // Add the tables obtained in threads
            SM_table_marker +=two_tables.first;
            Angle_table_marker += two_tables.second;

        }
    }
    // Divide the shading mask by the corresponding value of the angle table
    // If an angle combination has not been selected, suppose there is no shadow
    auto shadingMatrix = SM_table_marker.array().binaryExpr( Angle_table_marker.array() , [](auto x, auto y) { return y==0 ? 0 : x/y; });

    // Shading mask value 0 means that the surface is not shadowed, value 1 it is fully shadowed
    // Save the shading mask table to a csv file
    saveShadingMask(building_name,marker,shadingMatrix.matrix());

}
```

Shading Masks

Context / Previous Work

Iteration over the lambda function

```cpp
std::vector<int> n_rays_thread;
 n_rays_thread.push_back(M_Nrays - (M_Nthreads-1) * (int)(M_Nrays / M_Nthreads));
 for(int t= 1; t < M_Nthreads; ++t){
     n_rays_thread.push_back( M_Nrays / M_Nthreads);
 }

 // Used to store the future results
 std::vector< std::future< std::pair<Eigen::MatrixXd, Eigen::MatrixXd > > > futures;

 for(int t = 0; t < M_Nthreads; ++t){

     // Start a new asynchronous task
     futures.emplace_back(std::async(std::launch::async, rays_from_element, n_rays_thread[t]));
 }

 for( auto& f : futures){
     // Wait for the result to be ready
     auto two_tables =  f.get();

     // Add the tables obtained in threads
     SM_table_marker +=two_tables.first;
     Angle_table_marker += two_tables.second;

 }
```

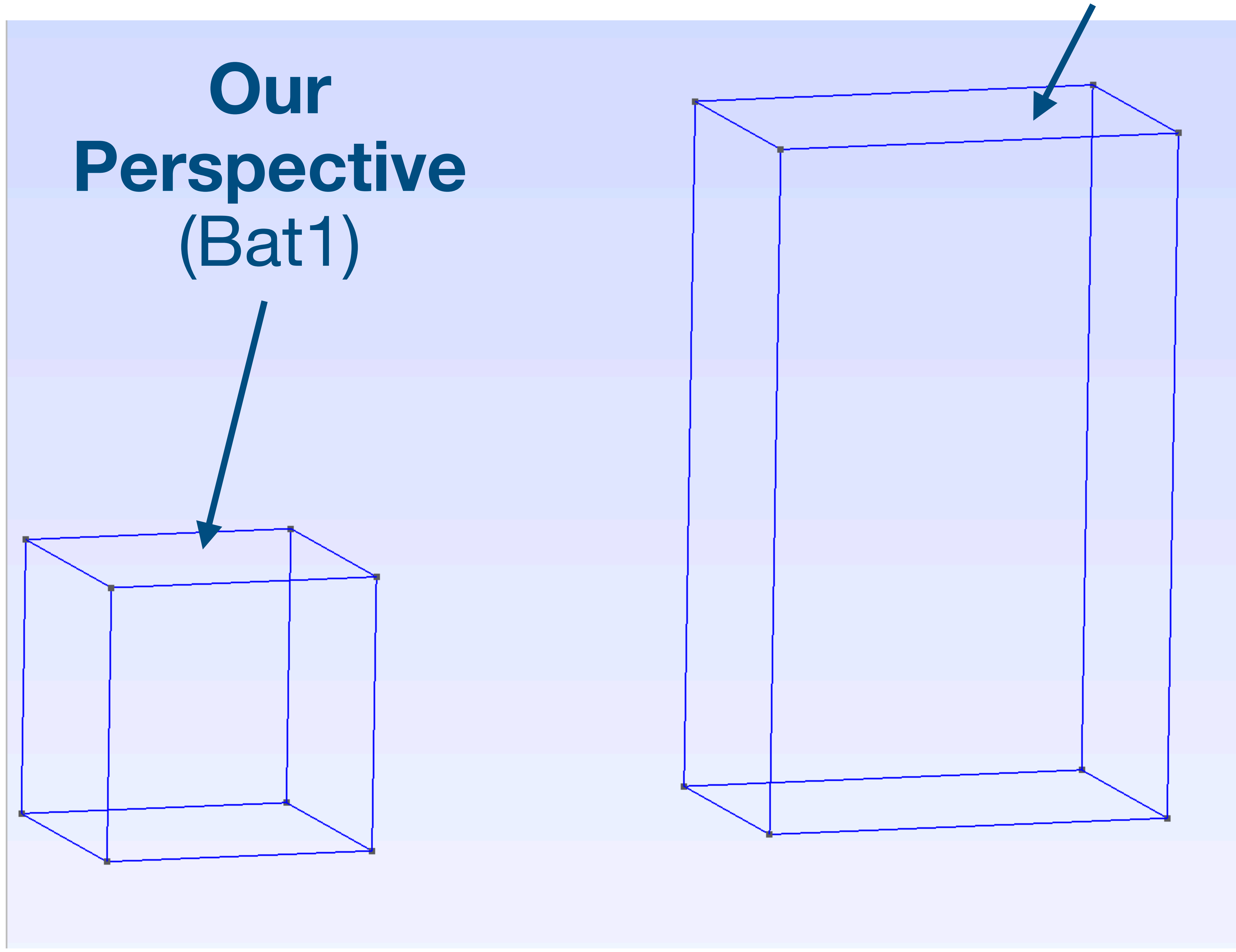
```cpp
for(auto  [marker,marker_id] : M_submeshes[building_name]->markerNames())
{

    SM_table_marker.setZero();
    Angle_table_marker.setZero();
    auto ray_submesh = createSubmesh(_mesh=M_submeshes[building_name],_range=markedelements(M_submeshes[building_name],marker));

    // Launch Nrays from each triangle of each marker
    for(auto const &el : ray_submesh->elements() ) // from each element of the submesh, launch M_Nrays randomly oriented
    {

            auto rays_from_element = [&,marker=marker](int n_rays_thread){

                Eigen::MatrixXd SM_table(M_azimuthSize,M_altitudeSize);
                SM_table.setZero();

                Eigen::MatrixXd Angle_table(M_azimuthSize,M_altitudeSize);
                Angle_table.setZero();

                int index_altitude;
                int index_azimuth;
                for(int i=0;i<n_rays_thread;i++)
                {

                    // Construct the ray emitting from a random point of the element
                    auto random_origin = get_random_point(el.second.vertices());

                    Eigen::VectorXd rand_dir(dim);
                    Eigen::VectorXd p1(dim),p2(dim),p3(dim),origin(3);
                    bool inward_ray=false;
                    if(dim==3)
                    {
                        for(int i=0;i<dim;i++)
                        {
                            p1(i)=column(el.second.vertices(), 0)[i];
                            p2(i)=column(el.second.vertices(), 1)[i];
                            p3(i)=column(el.second.vertices(), 2)[i];
                            origin(i) = random_origin[i];
                            rand_dir(i) = random_direction[i];
                        }
                        auto element_normal = ((p3-p1).head<3>()).cross((p2-p1).head<3>());
                        element_normal.normalize();

                        // Choose the direction randomly among the latitude and azimuth
                        getRandomDirectionSM(random_direction,M_gen,M_gen2,index_azimuth,index_altitude);
                        for(int i=0;i<dim;i++)
                        {
                            rand_dir(i) = random_direction[i];
                        }
                        if(rand_dir.dot(element_normal)>=0)
                        {
                            inward_ray=true;
                        }

                    }

                    BVHRay ray(origin,rand_dir);

                    int closer_intersection_element = -1;
                    if(inward_ray)
                    {
                        closer_intersection_element = 1;
                    }
                    else
                    {
                        for(auto& [building_name,bvh_building_tree] : M_bvh_tree_vector)
                        {
                            closer_intersection_element = bvh_building_tree.raySearch(ray,"") ;
                            if (closer_intersection_element >=0 )
                                break;
                        }
                    }
                    // If there is an intersection, increase the shading mask table entry by 1 and augment the angle table by 1 as well
                    if ( closer_intersection_element >=0 )
                    {
                        SM_table(index_azimuth,index_altitude)++;
                        Angle_table(index_azimuth,index_altitude)++;
                    }
                    else
                    {
                        Angle_table(index_azimuth,index_altitude)++;
                    }
                }

                return std::make_pair(SM_table,Angle_table);
            };

        // Execute the lambda function on multiple threads using
        // std::async and std::future to collect the results
        std::vector<int> n_rays_thread;
        n_rays_thread.push_back(M_Nrays - (M_Nthreads-1) * (int)(M_Nrays / M_Nthreads));
        for(int t= 1; t < M_Nthreads; ++t){
            n_rays_thread.push_back( M_Nrays / M_Nthreads);
        }

        // Used to store the future results
        std::vector< std::future< std::pair<Eigen::MatrixXd, Eigen::MatrixXd > > > futures;

        for(int t = 0; t < M_Nthreads; ++t){

            // Start a new asynchronous task
            futures.emplace_back(std::async(std::launch::async, rays_from_element, n_rays_thread[t]));
        }

        for( auto& f : futures){
            // Wait for the result to be ready
            auto two_tables =  f.get();

            // Add the tables obtained in threads
            SM_table_marker +=two_tables.first;
            Angle_table_marker += two_tables.second;

        }
    }
    // Divide the shading mask by the corresponding value of the angle table
    // If an angle combination has not been selected, suppose there is no shadow
    auto shadingMatrix = SM_table_marker.array().binaryExpr( Angle_table_marker.array() , [](auto x, auto y) { return y==0 ? 0 : x/y; });

    // Shading mask value 0 means that the surface is not shadowed, value 1 it is fully shadowed
    // Save the shading mask table to a csv file
    saveShadingMask(building_name,marker,shadingMatrix.matrix());

}
```

Shading Masks

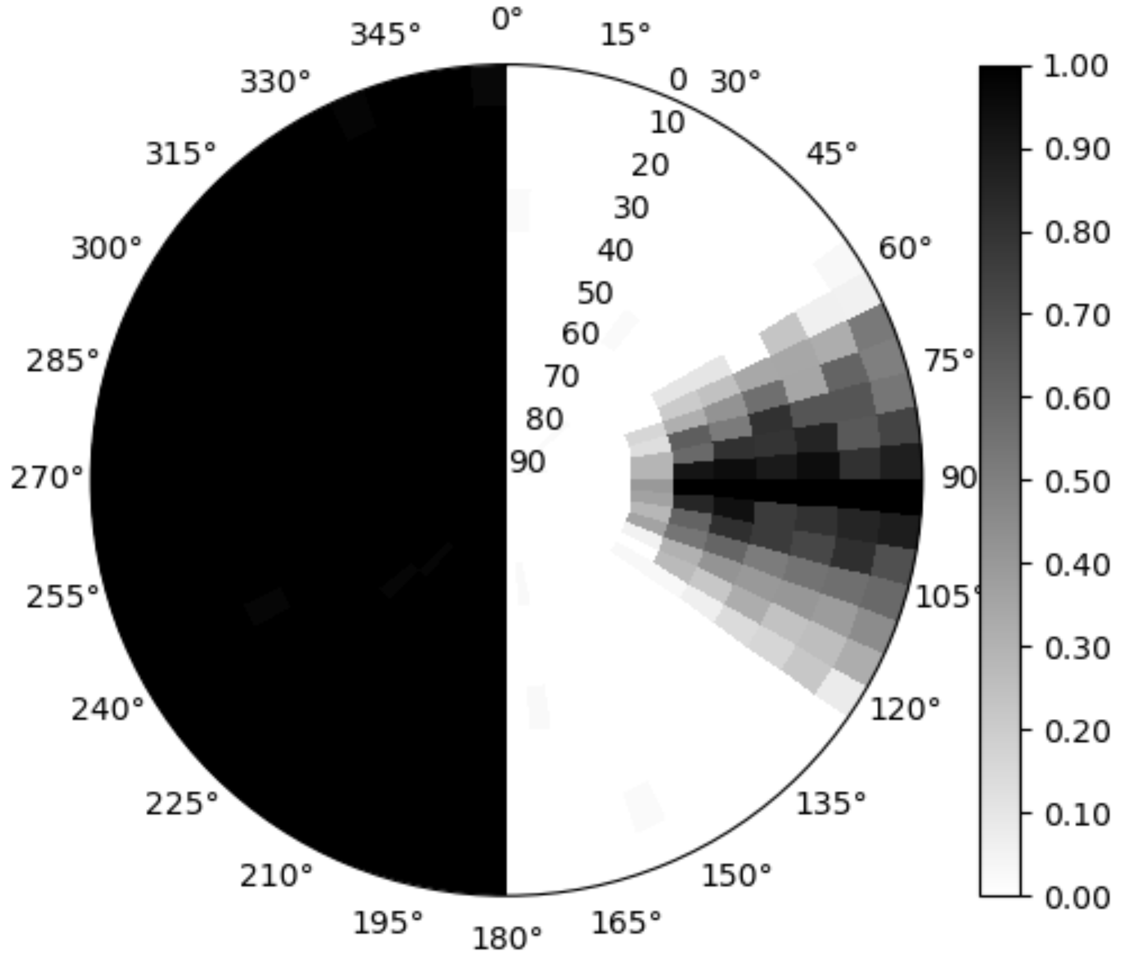
Context / Previous Work

Test Case: 2 Buildings

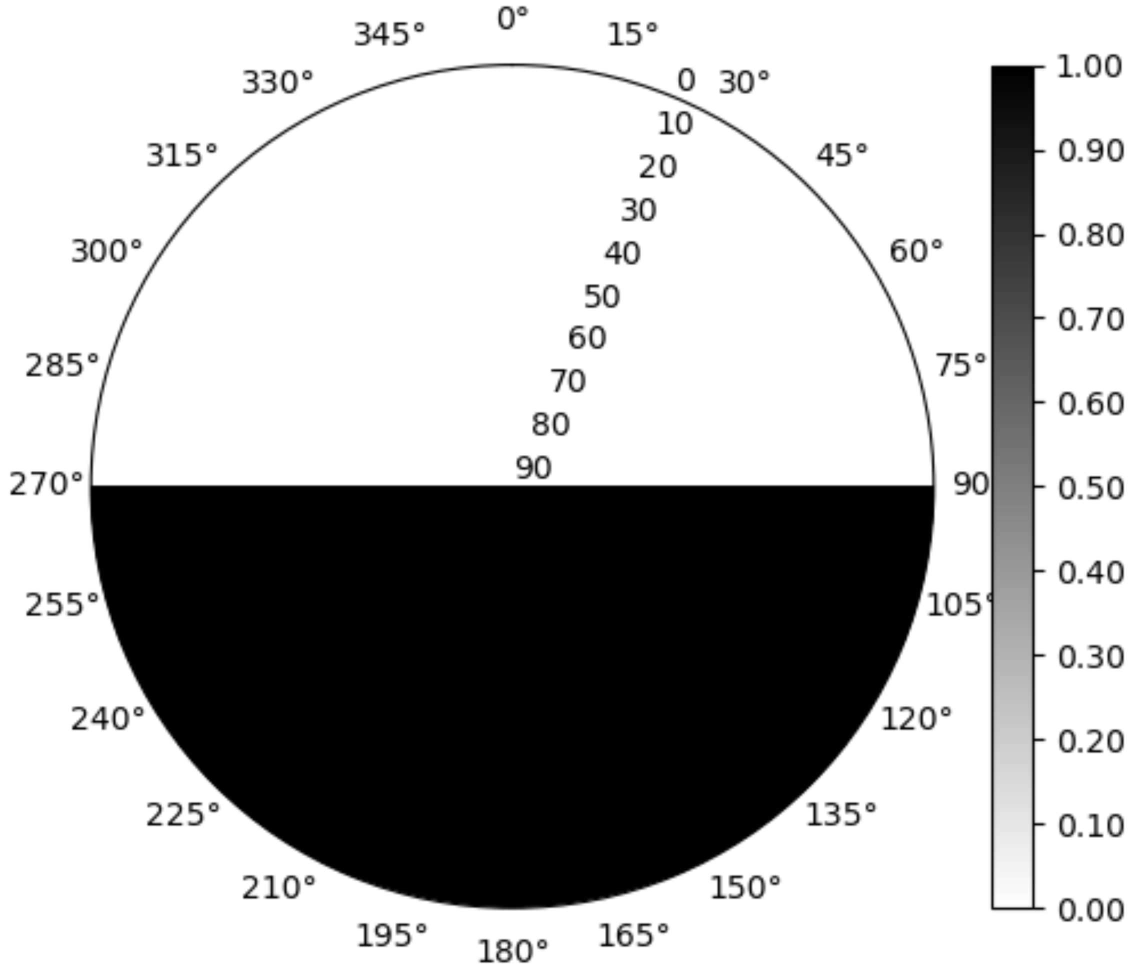
Casted Shadows
(Bat2)



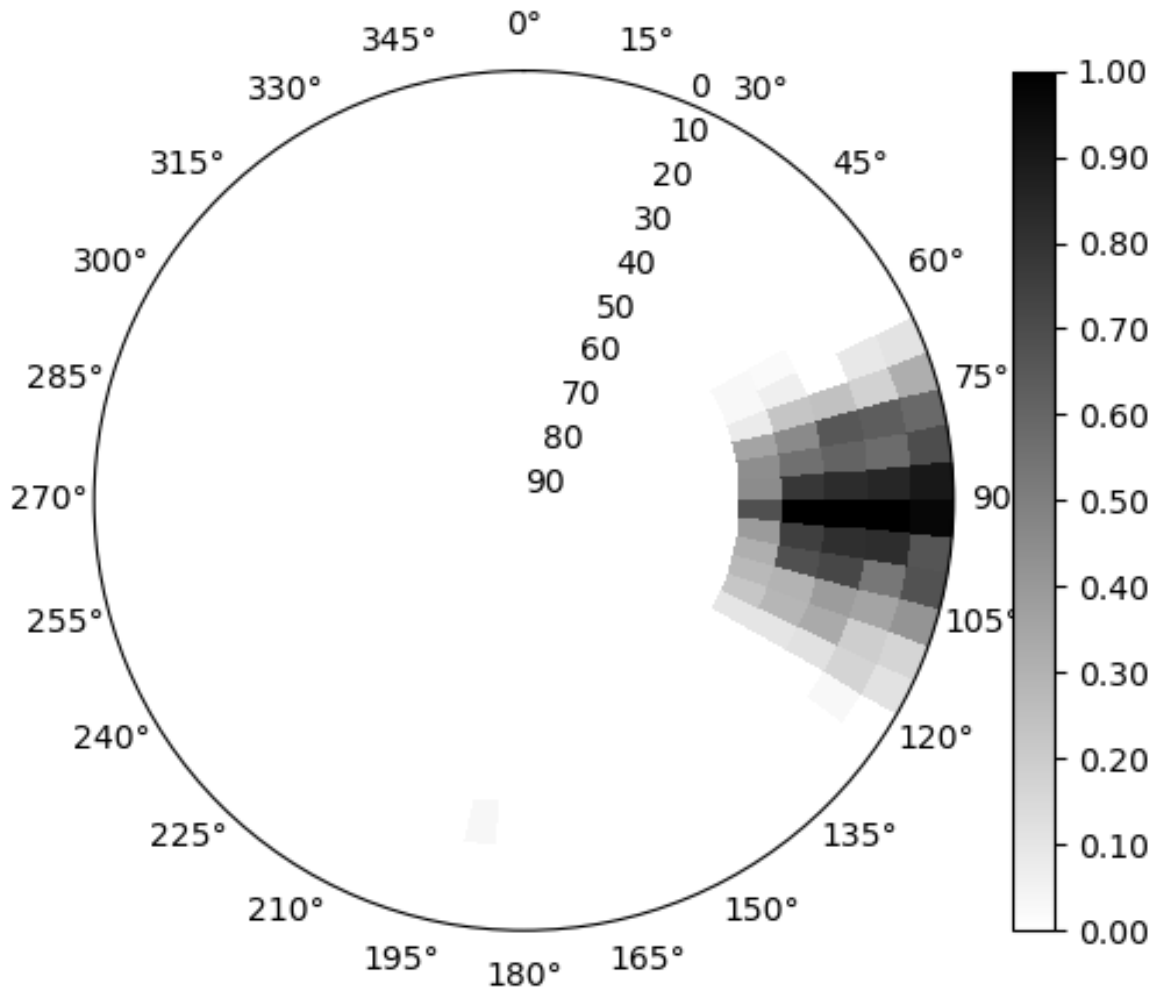
Est



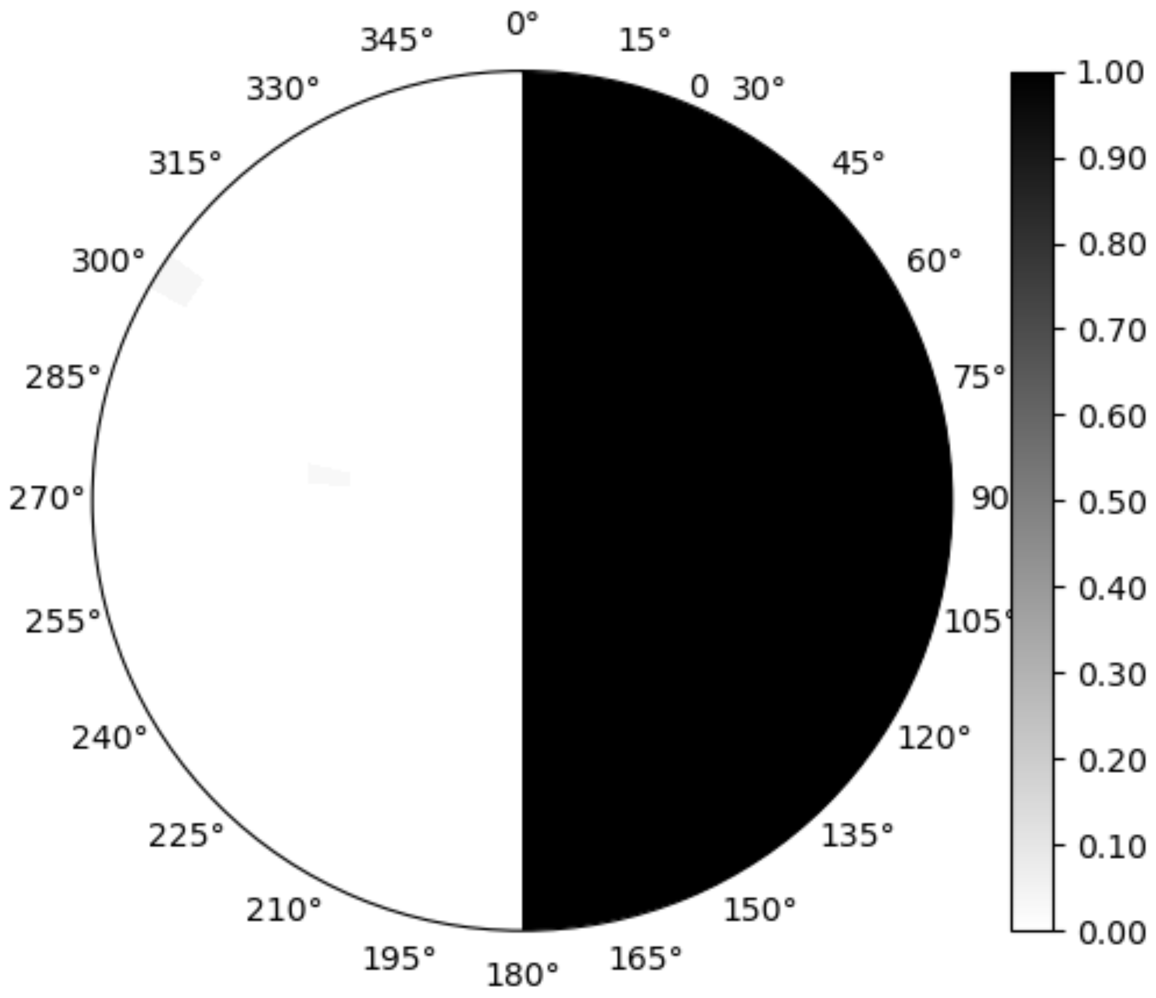
Nord



Top



Ouest



Shading Masks

Optimizations

EigenRand's **Thread Safety** and **Vectorization** Capabilities

```cpp
auto rays_from_element = [&,marker=marker](int n_rays_thread){

    Eigen::Rand::UniformIntGen<int> unif_gen_azi(0,M_azimuthSize-1);
    Eigen::Rand::UniformIntGen<int> unif_gen_alti(0,M_altitudeSize-1);
    Eigen::Rand::P8_mt19937_64 urng_azi{ std::random_device{}() };
    Eigen::Rand::P8_mt19937_64 urng_alti{ std::random_device{}() };

    Eigen::Rand::UniformRealGen<double> unif_gen_real1(0.,1.);
    Eigen::Rand::UniformRealGen<double> unif_gen_real2(0.,1.);
    Eigen::Rand::P8_mt19937_64 urng_real1{ std::random_device{}() };
    Eigen::Rand::P8_mt19937_64 urng_real2{ std::random_device{}() };

    Eigen::MatrixXd SM_table(M_azimuthSize,M_altitudeSize);
    SM_table.setZero();

    Eigen::MatrixXd Angle_table(M_azimuthSize,M_altitudeSize);
    Angle_table.setZero();

    Eigen::VectorXi index_altitude(n_rays_thread);
    Eigen::VectorXi index_azimuth(n_rays_thread);

    Eigen::MatrixXd random_directions(n_rays_thread,3);
    Eigen::MatrixXd random_origins(n_rays_thread,3);

    random_origins = get_random_points(el.second.vertices(),n_rays_thread, unif_gen_real1,
                                       unif_gen_real2, urng_real1, urng_real2);
    random_directions = get_random_directions(index_azimuth, index_altitude, n_rays_thread,
                                       unif_gen_azi, unif_gen_alti, urng_azi, urng_alti);
```

```cpp
Eigen::VectorXd S = unif_gen_real1.generate<Eigen::VectorXd>(n_rays_thread,1,urng_real1);
Eigen::VectorXd T = unif_gen_real2.generate<Eigen::VectorXd>(n_rays_thread,1,urng_real2);
```

Shading Masks

Optimizations

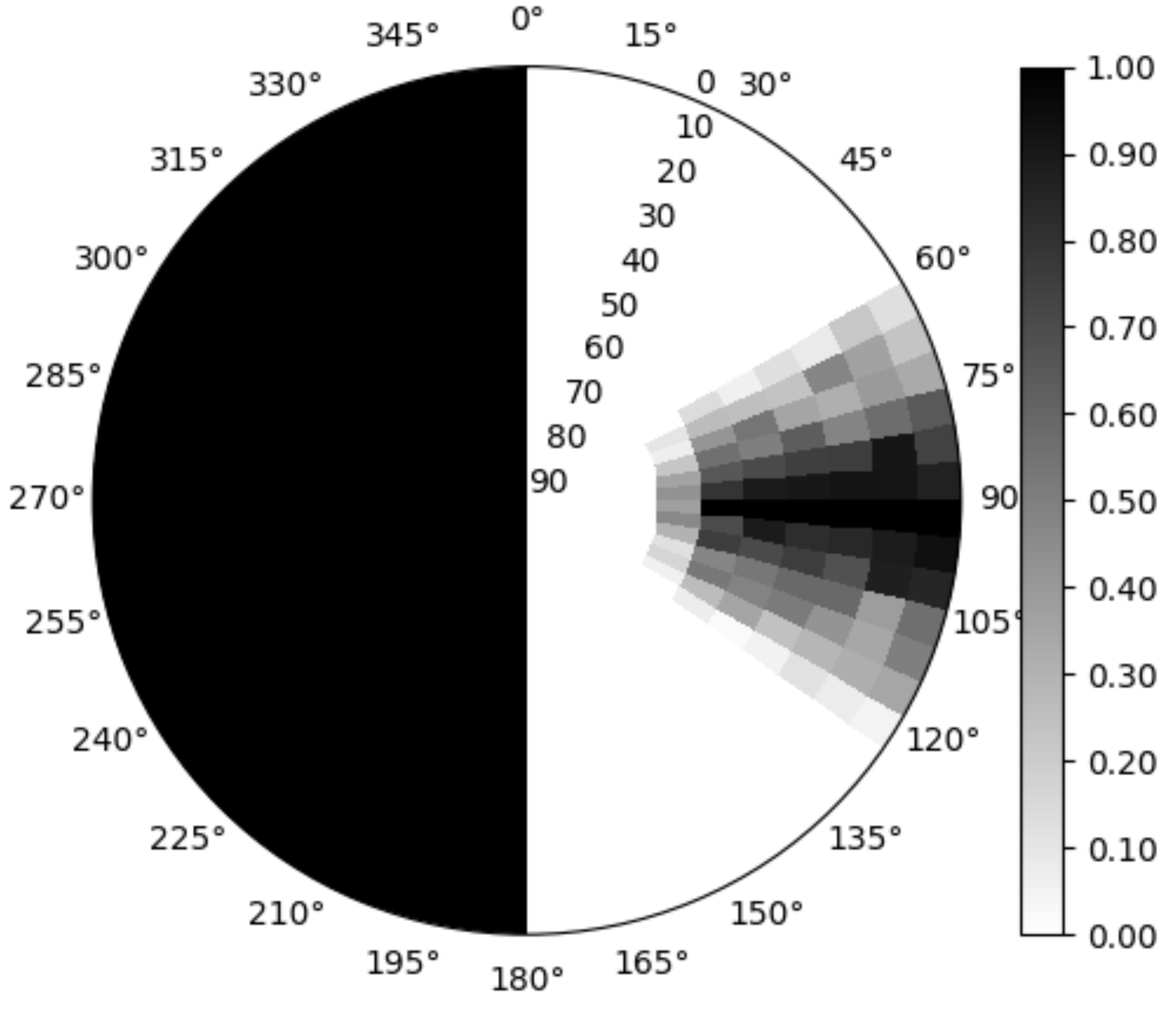
Validation of different RNGs

Case Bat1 Est

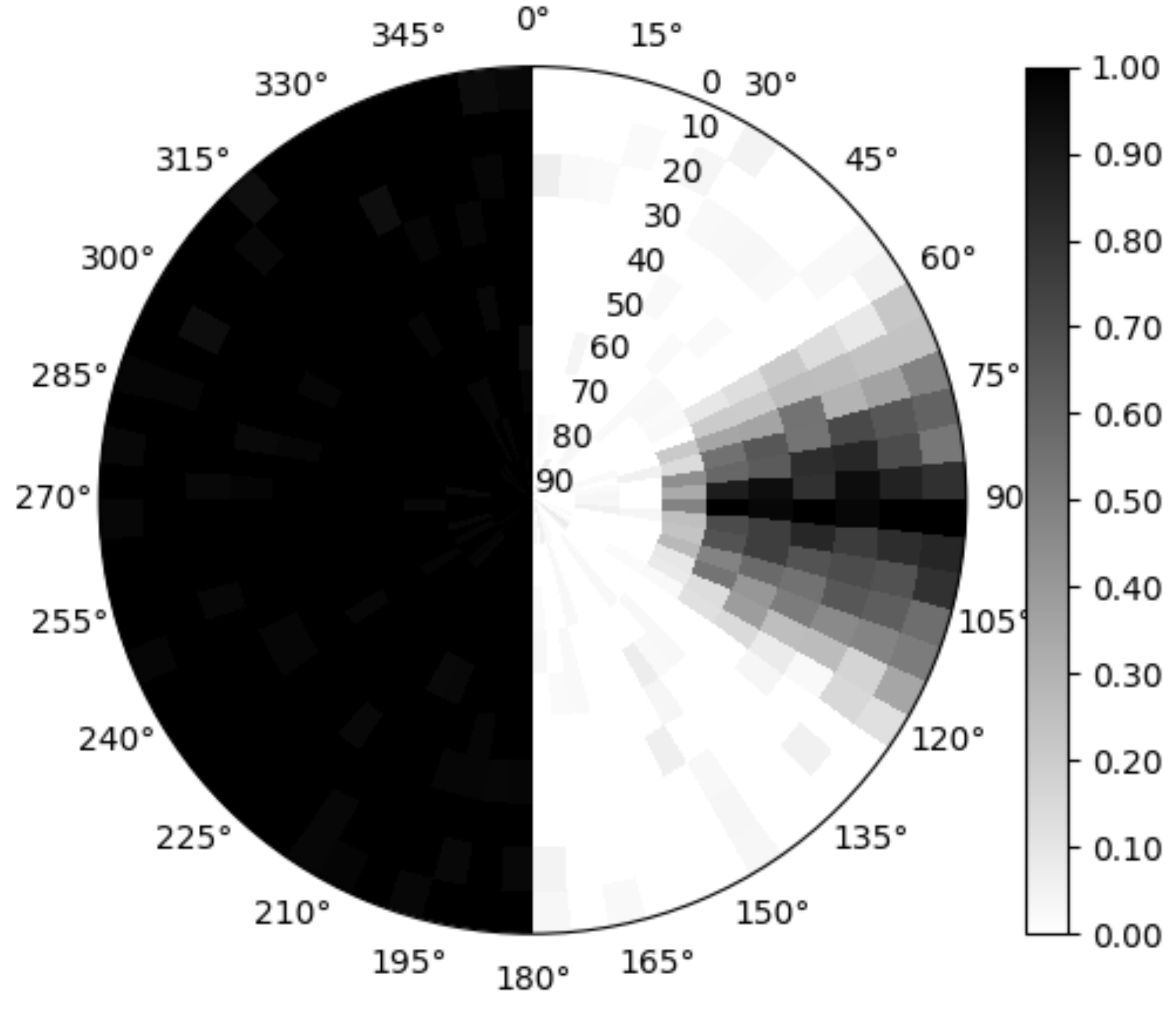
```python
for matrix in matrix_files:
    # compare the firs part of the name and check if it corresponds
    # to the name of the base matrix appended with something
    if str(matrix.name)[:len(base_matrix.stem)] == base_matrix.stem:
        comparison_matrices.append(matrix)

for matrix in comparison_matrices:
    # Load comparison matrix
    comparison_data = np.genfromtxt(matrix, delimiter=',')

    # Compute difference
    diff_data = base_data - comparison_data
```

Way more problems to map back during the asynchronous task

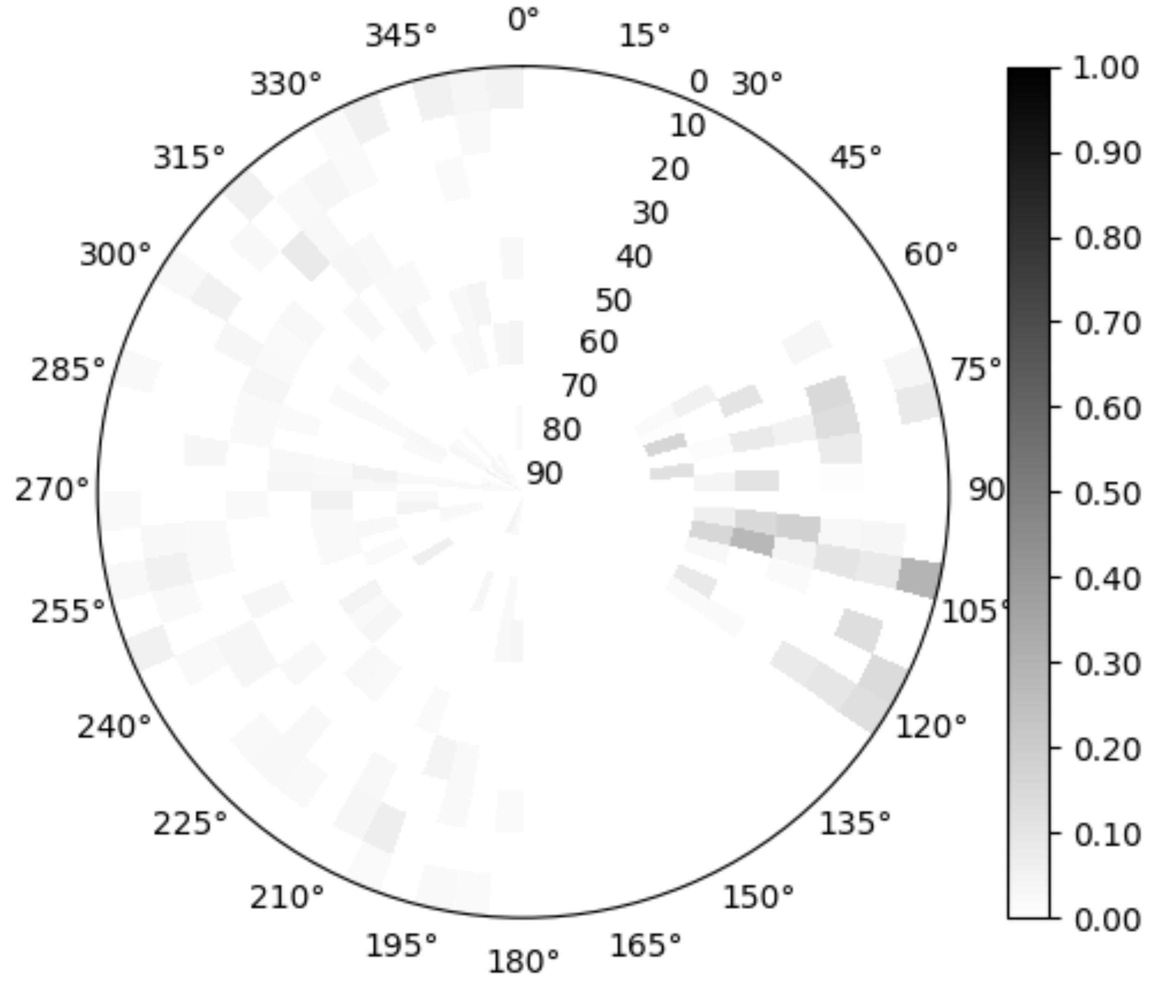
EigenRand



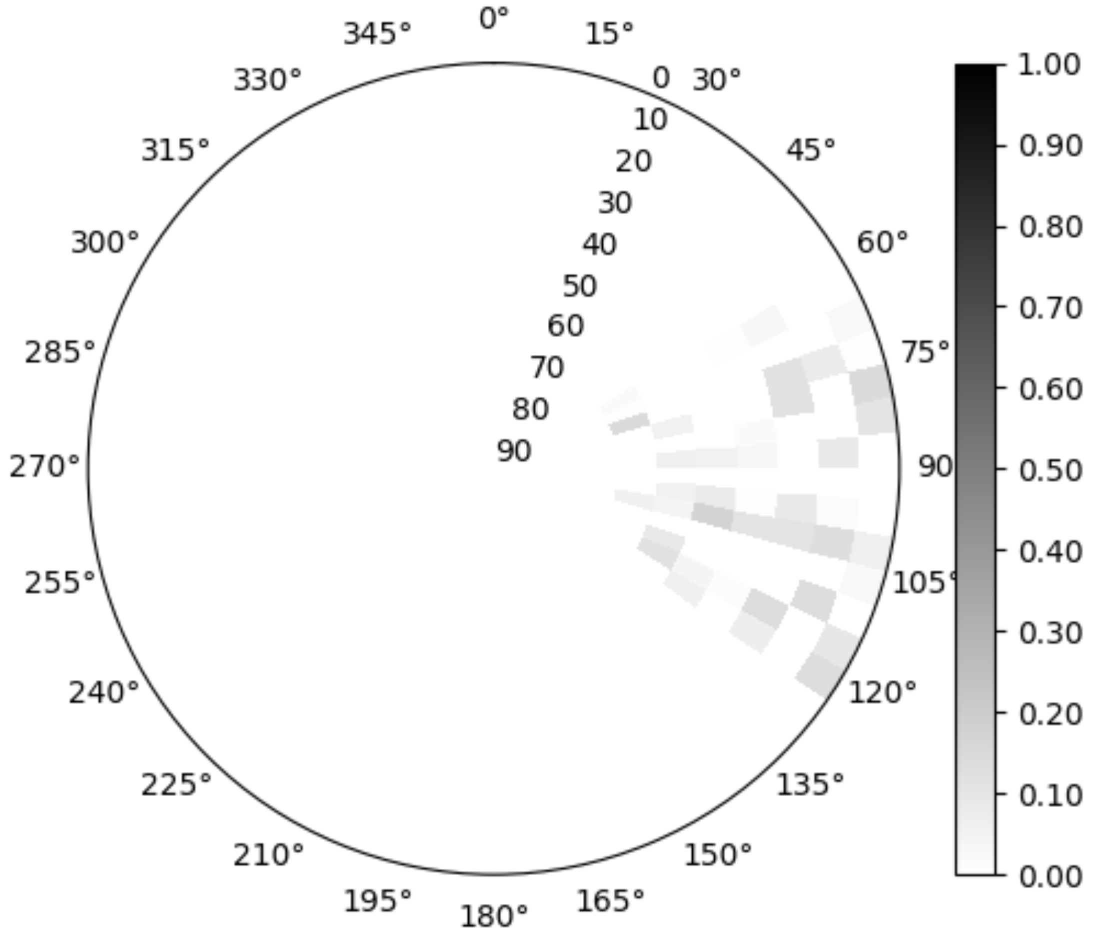
PCG



PCG



EigenRand

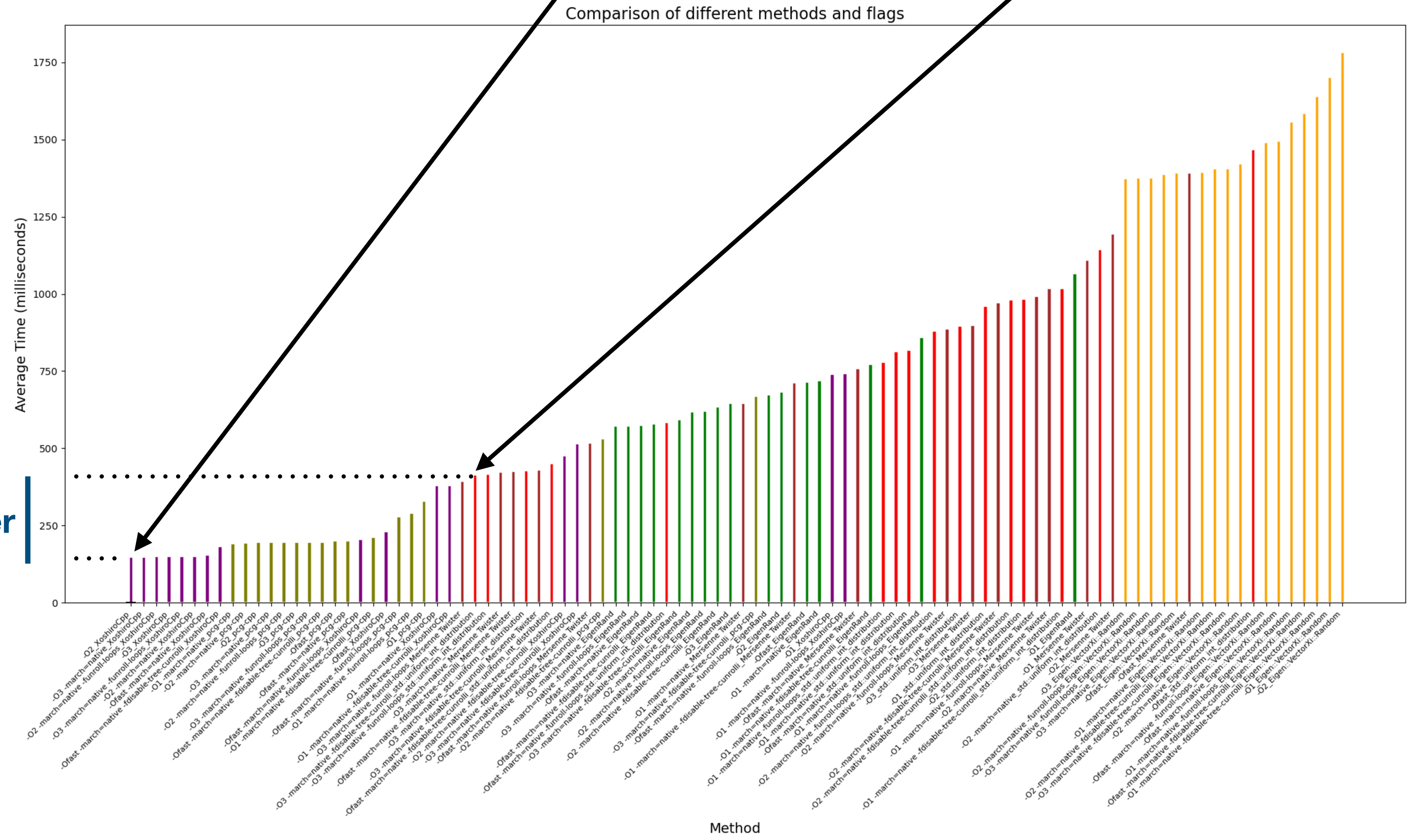


Shading Masks

Optimizations

Xoshiro-Cpp : 125 ms

std::uniform : 400 ms



3.2 x faster

Shading Masks

Optimizations

Up to **8.63%** performance boost

Randomness causes **inconsistent results**

```
==============================================================================
METHOD                    TIME (seconds) FLAGS
==============================================================================
RNG STD:                  9.389095158 s   '-O2'
RNG EIGEN:                9.121100925 s   '-O2'
RNG EIGEN CAST:           9.19422145 s    '-O2 -funroll-loops'
RNG PCG:                  9.382495989 s   '-Ofast -funroll-loops'
RNG XOSHIRO:              8.578198413 s   '-Ofast -funroll-loops'
RNG EIGEN VEC:            9.554856035 s   '-Ofast -funroll-loops'
RNG MIX:                  9.412993456 s   '-O2'
==============================================================================
```

```
==============================================================================
METHOD                    TIME (seconds) FLAGS
==============================================================================
RNG STD:                  8.241940879 s   '-Ofast -ftlo'
RNG EIGEN:                7.984007652000001 s '-Ofast'
RNG EIGEN CAST:           8.00745842 s    '-O2 -funroll-loops'
RNG PCG:                  7.531147481 s   '-Ofast -funroll-loops
                                           -funsafe-math-optimizations'
RNG XOSHIRO:              7.919334421 s   '-O2 -funroll-loops'
RNG EIGEN VEC:            7.894269027 s   '-O2 -funroll-loops'
RNG MIX:                  7.855635813 s   '-Ofast -ftlo'
==============================================================================
```

Perez all-Weather Sky Distribution Model

Implementations

Element-wise Product for the **diffuse radiation factor**

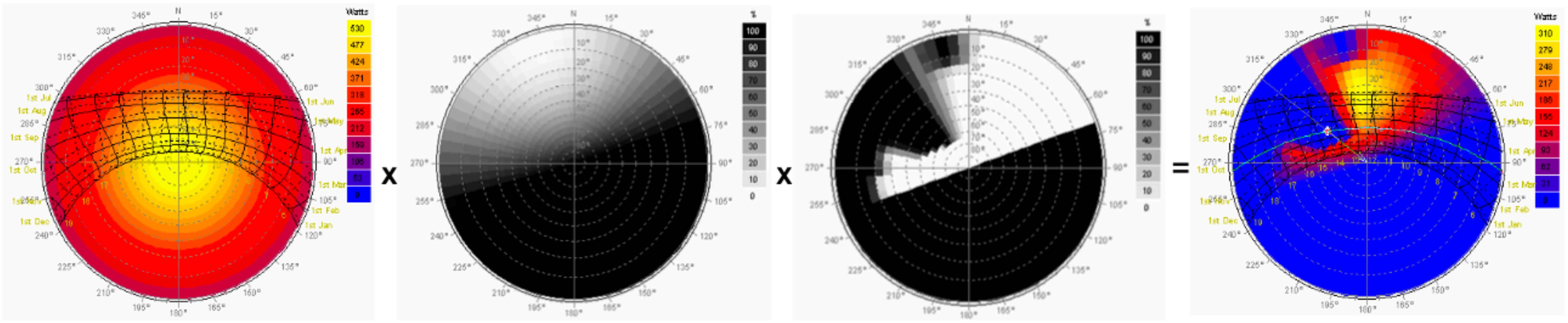


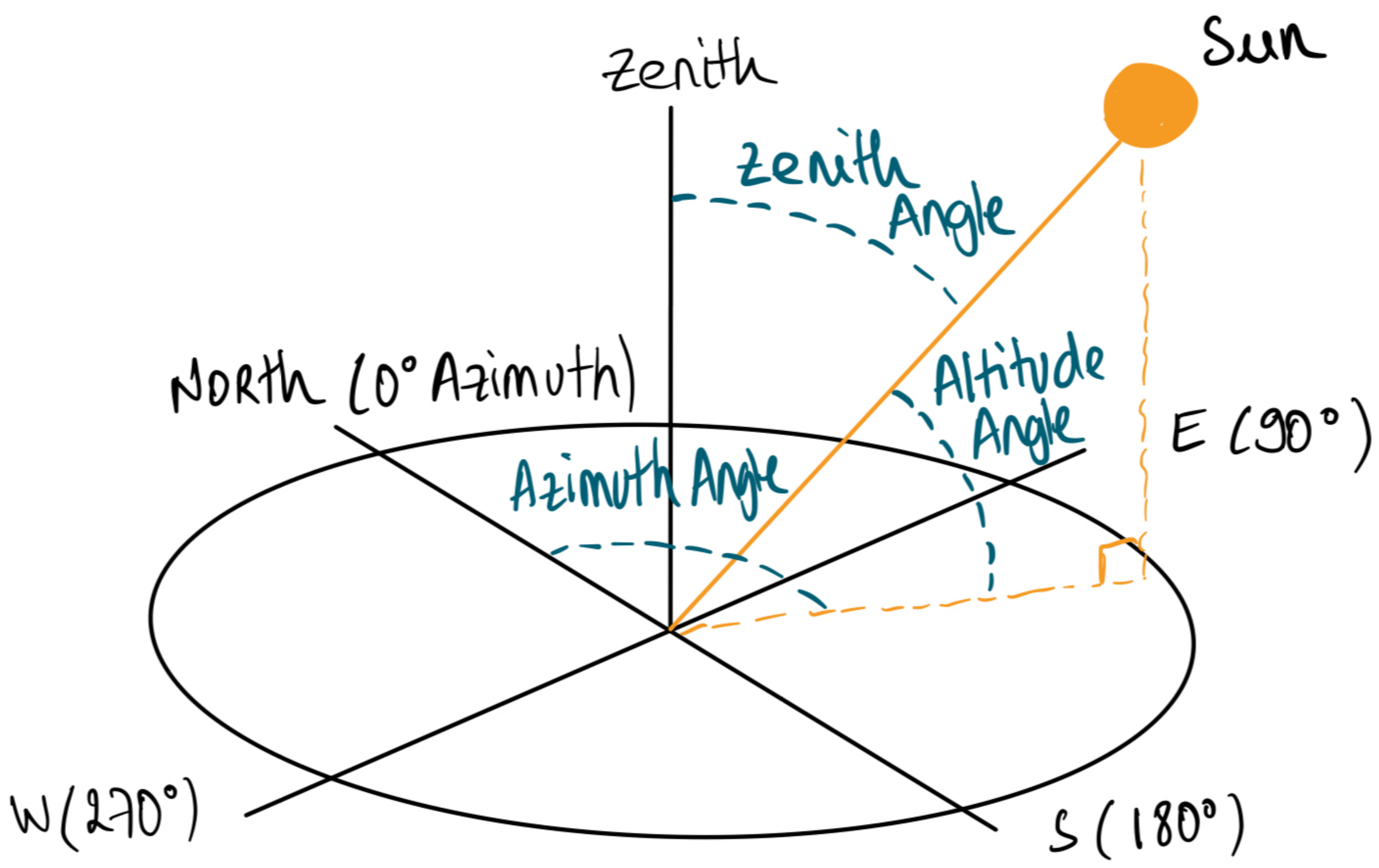
Figure 5 A diffuse radiation factor can be determined by multiplying the diffuse sky distribution by both incidence angle and shading factors for each sky segment.

General **Multi-purpose** **Robust** **Adaptive**

Image from the article: The Application of Shading Masks in Building Applications

Perez all-Weather Sky Distribution Model

Implementation



$$lv = f(\zeta, \gamma) = (1 + a\exp(\frac{b}{\cos(\zeta)}))(1 + c\exp(d\gamma) + e\cos^2(\gamma))$$

$$x = x_1(\epsilon) + x_2(\epsilon)Z + \Delta(x_3(\epsilon) + x_4(\epsilon)Z)$$

where $x \in \{a, b, c, d, e\}$

$$\cos(\gamma) = \cos(Z)\cos(z) + \sin(Z)\sin(z)\cos(\ \ A - a\ \ )$$

$$Lv = \frac{lv \times Evd}{\int lv(\gamma, \zeta)\cos(\zeta)\, d\omega}$$

Perez all-Weather Sky Distribution Model

Implementation

Hourly computation of the integral using a **deterministic method** (and solid angle)

Leveraging **Monte Carlo**

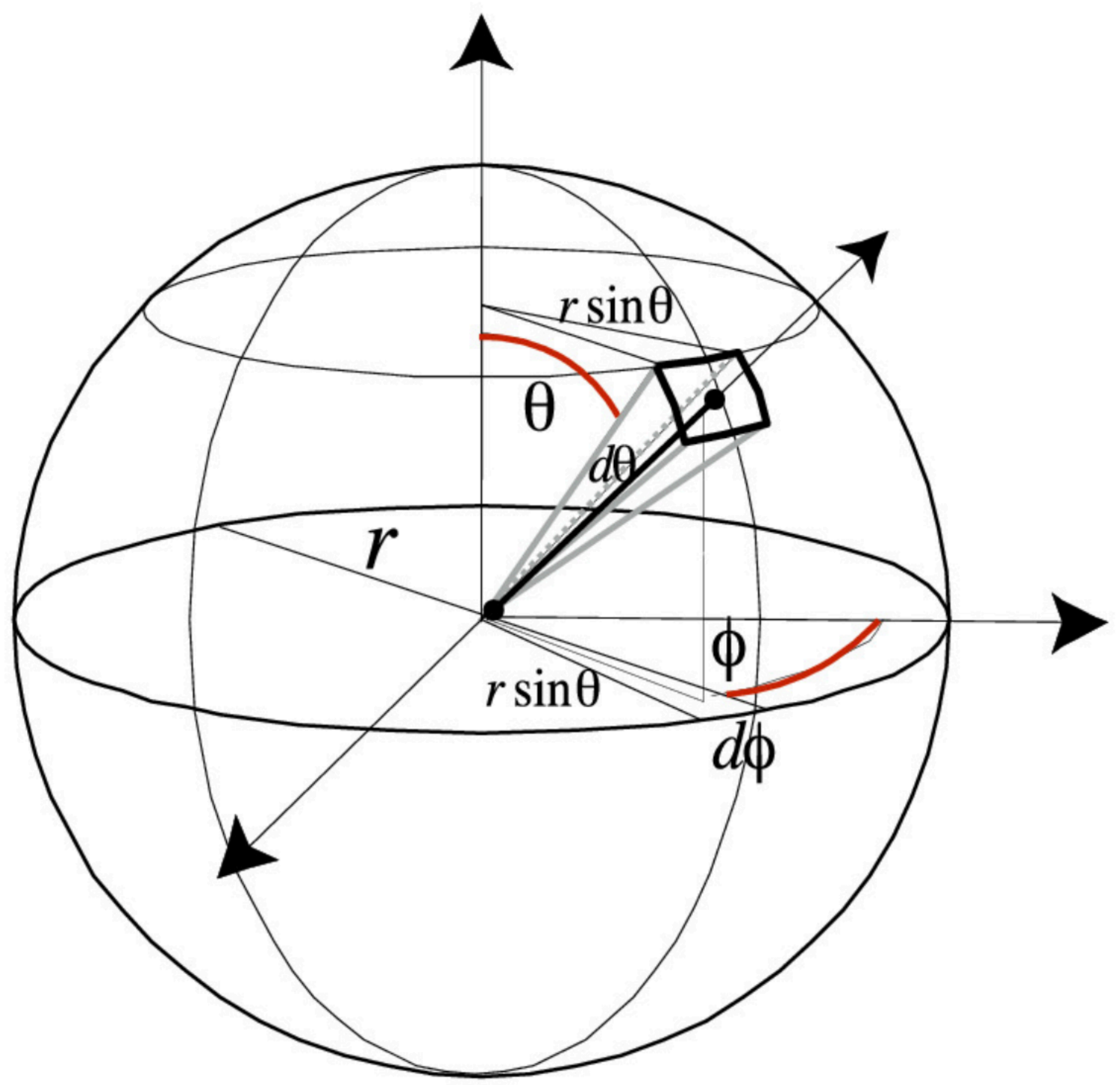


Image from eecs.berkeley

```cpp
for (int i = 0; i < M_Time.size(); i++)
{
    M_SkyModel.setZero();
    int hour = M_Time[i];
    double delta = M_Brightness[i];
    double epsilon = M_Clearness[i];
    double A = M_SolarAzimuth[i];
    double Z = M_ZenithAngles[i];
    setPerezParameters(epsilon, delta, Z);
    double integral_value = compute_integral(A, Z);

    for (int j = 0; j < AltitudeSize; j++)
    {
        for (int k = 0; k < AzimuthSize; k++)
        {
            double z = M_PI / 2.0 - (double)j / (double)AltitudeSize * M_PI / 2.0;  // zenith of the sky element
            double a = (double)k / (double)AzimuthSize * 2.0 * M_PI;  // azimuth of the sky element
            if ( j == 0 )
                z = M_PI / 2.0 - 1e-6; // avoid division by zero

            // Calculate gamma with error checking
            double gamma;
            double cos_gamma = std::cos(Z) * std::cos(z) + std::sin(Z) * std::sin(z) * std::cos(std::abs(A - a));
            if (cos_gamma > 1 && cos_gamma < 1.1)
                gamma = 0;
            else if (cos_gamma > 1.1)
            {
                throw std::logic_error ("Error in calculation of gamma (angle between point and sun)");
            }
            else
            {
                gamma = std::acos(cos_gamma);
            }
            M_SkyModel(k, j) += getNormalizedLuminance(z, gamma, M_DiffuseIlluminance[i], A, Z, integral_value);
        }
    }
    saveSkyModel(hour);
}
```

Perez all-Weather Sky Distribution Model

Implementation

Computation of **Normalized Luminance** over the Sky Hemisphere

$$\cos(\gamma) = \cos(Z)\cos(z) + \sin(Z)\sin(z)\cos(\ A - a\ )$$

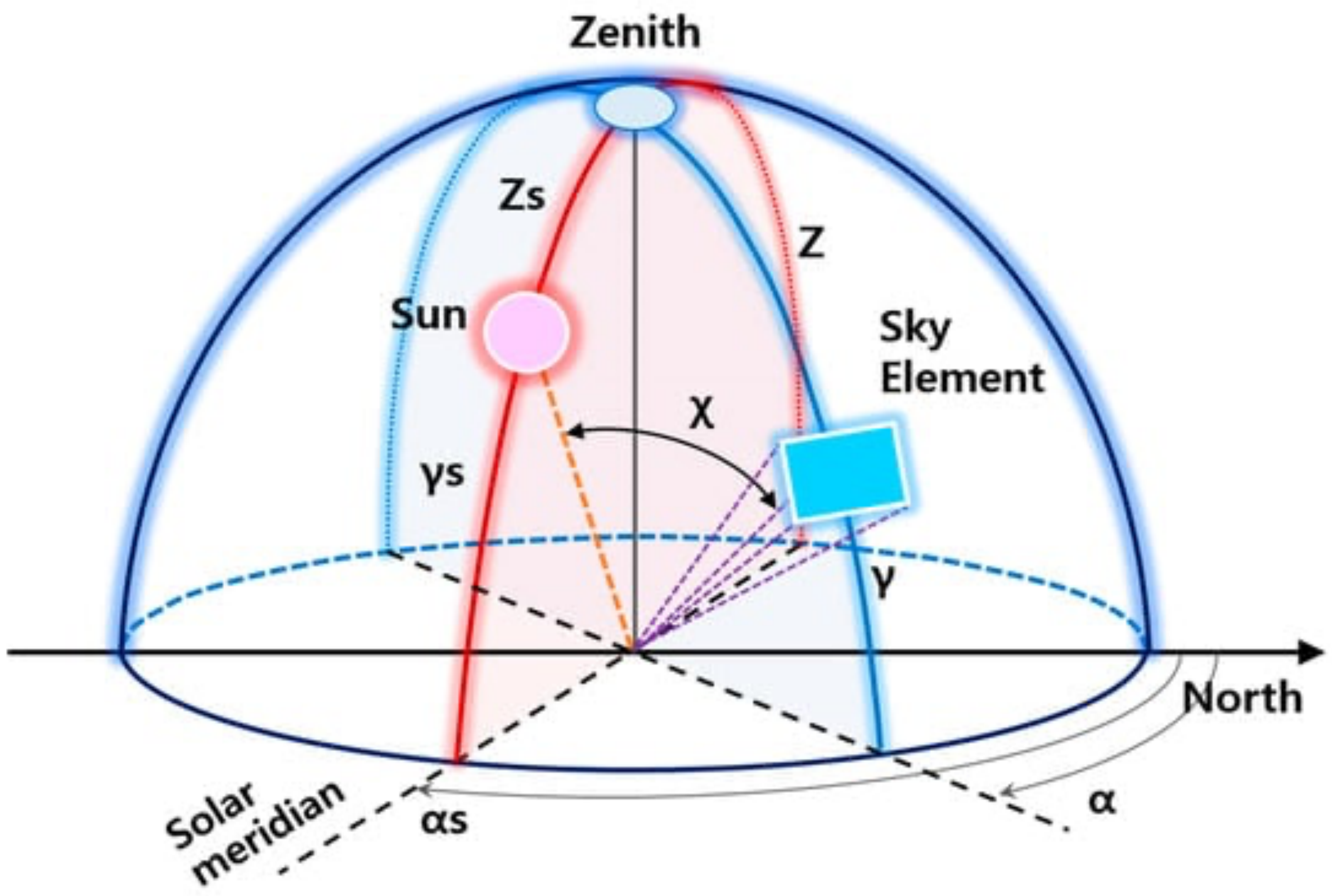


Image from ResearchGate

```
for (int i = 0; i < M_Time.size(); i++)
{
    M_SkyModel.setZero();
    int hour = M_Time[i];
    double delta = M_Brightness[i];
    double epsilon = M_Clearness[i];
    double A = M_SolarAzimuth[i];
    double Z = M_ZenithAngles[i];
    setPerezParameters(epsilon, delta, Z);
    double integral_value = compute_integral(A, Z);

    for (int j = 0; j < AltitudeSize; j++)
    {
        for (int k = 0; k < AzimuthSize; k++)
        {
            double z = M_PI / 2.0 - (double)j / (double)AltitudeSize * M_PI / 2.0;  // zenith of the sky element
            double a = (double)k / (double)AzimuthSize * 2.0 * M_PI;  // azimuth of the sky element
            if ( j == 0 )
                z = M_PI / 2.0 - 1e-6; // avoid division by zero

            // Calculate gamma with error checking
            double gamma;
            double cos_gamma = std::cos(Z) * std::cos(z) + std::sin(Z) * std::sin(z) * std::cos(std::abs(A - a));
            if (cos_gamma > 1 && cos_gamma < 1.1)
                gamma = 0;
            else if (cos_gamma > 1.1)
            {
                throw std::logic_error ("Error in calculation of gamma (angle between point and sun)");
            }
            else
            {
                gamma = std::acos(cos_gamma);
            }
            M_SkyModel(k, j) += getNormalizedLuminance(z, gamma, M_DiffuseIlluminance[i], A, Z, integral_value);
        }
    }
    saveSkyModel(hour);
}
```

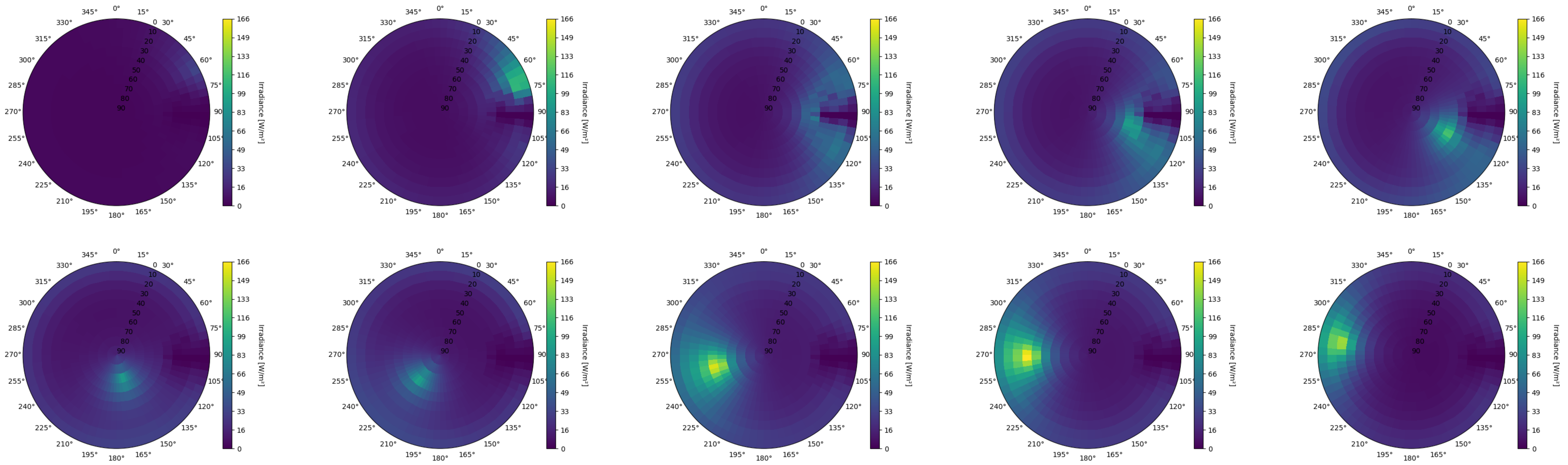
Perez all-Weather Sky Distribution Model

Implementation

Strasbourg - 20.07.21

Test case : example Buildings

Without Interpolation algorithm



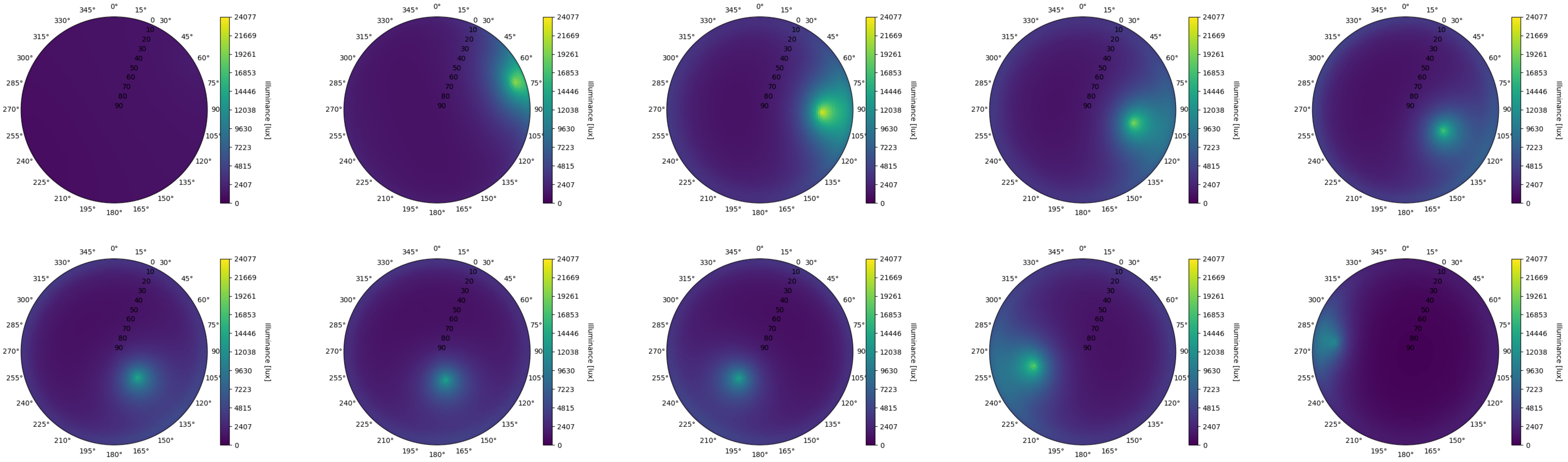
Perez all-Weather Sky Distribution Model

Implementation

Strasbourg - 20.07.21

Test case : example Buildings

With Interpolation algorithm



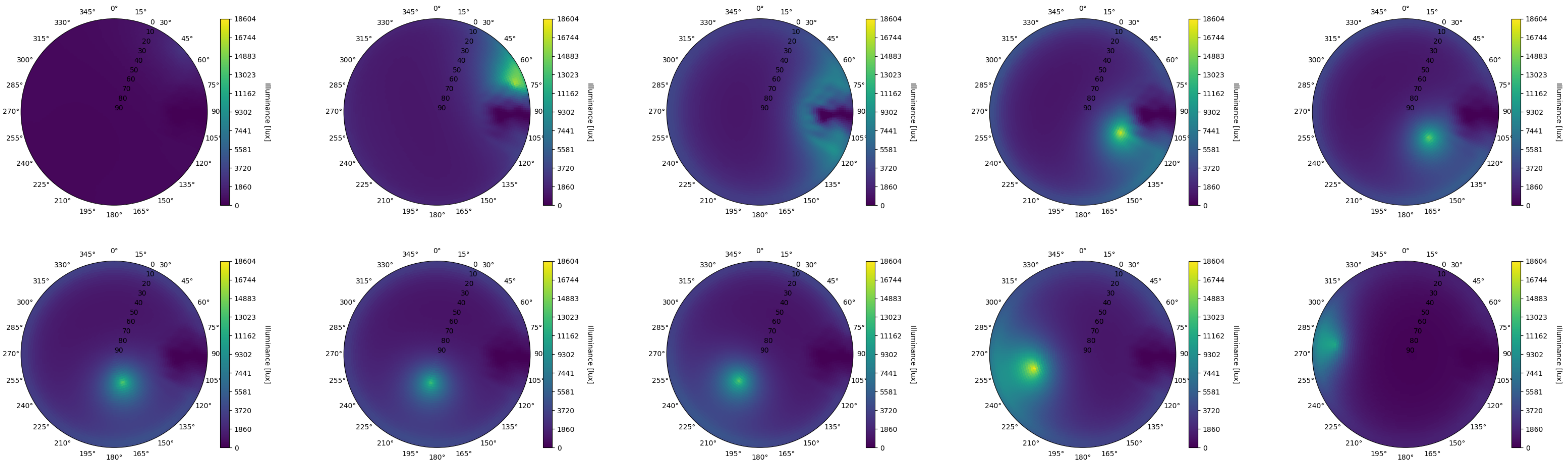
Perez all-Weather Sky Distribution Model

Implementation

Strasbourg - 20.07.21

Test case : example Buildings

With Interpolation algorithm

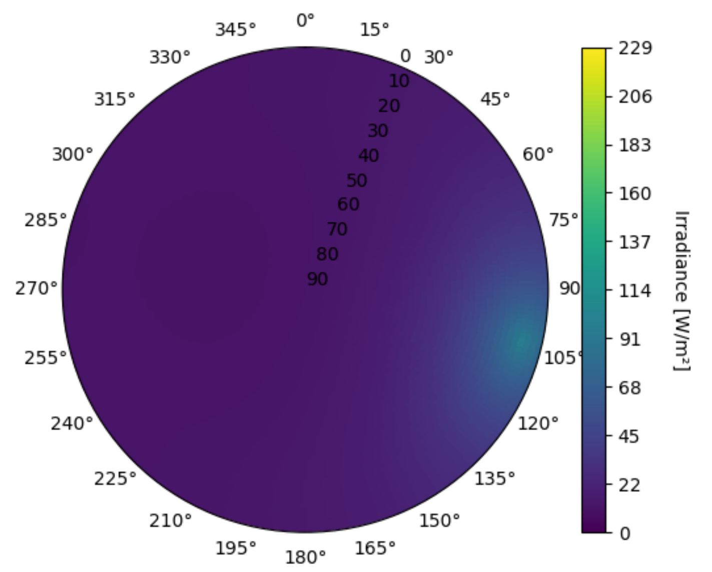


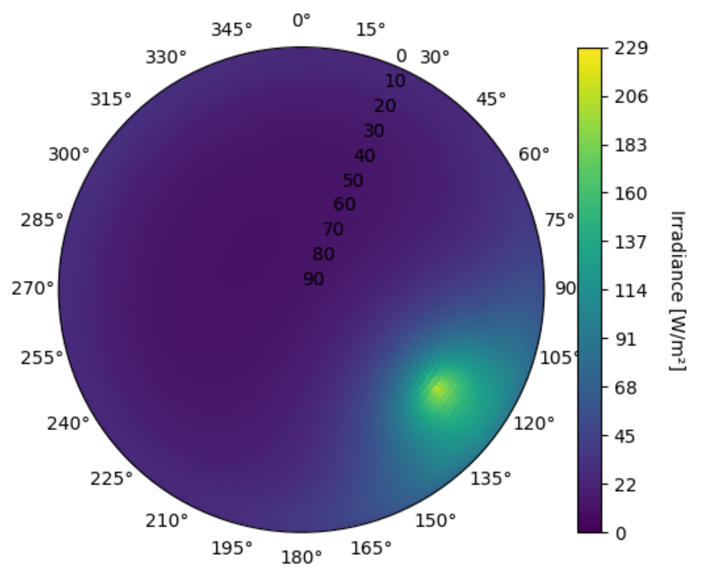
Perez all-Weather Sky Distribution Model

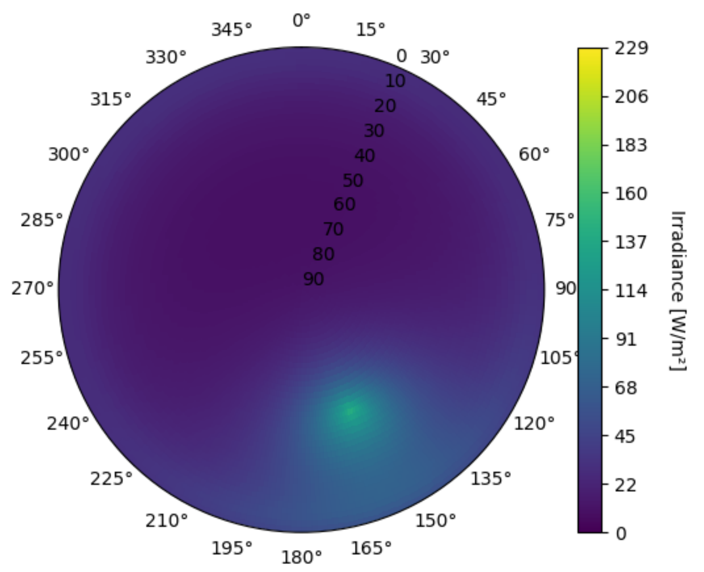
Implementation

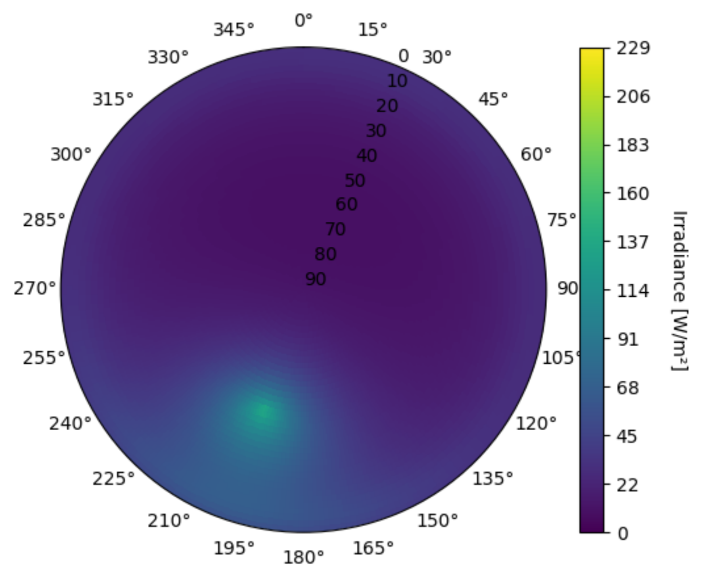
New-York - 03.03.20

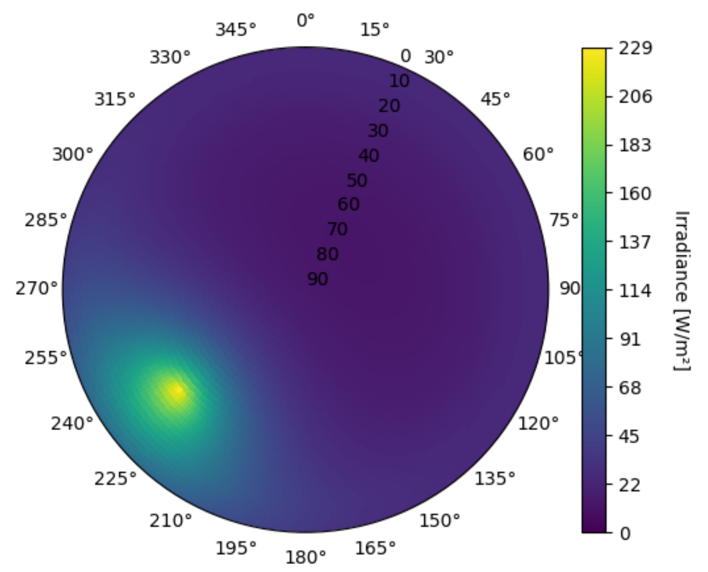
Test case : example Buildings

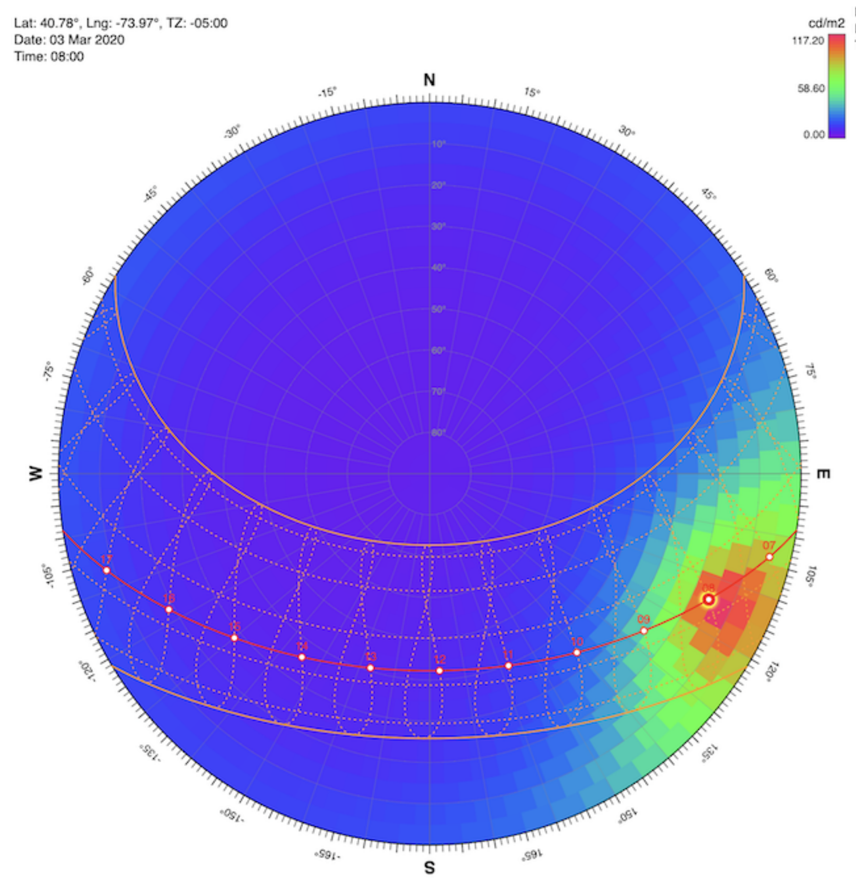


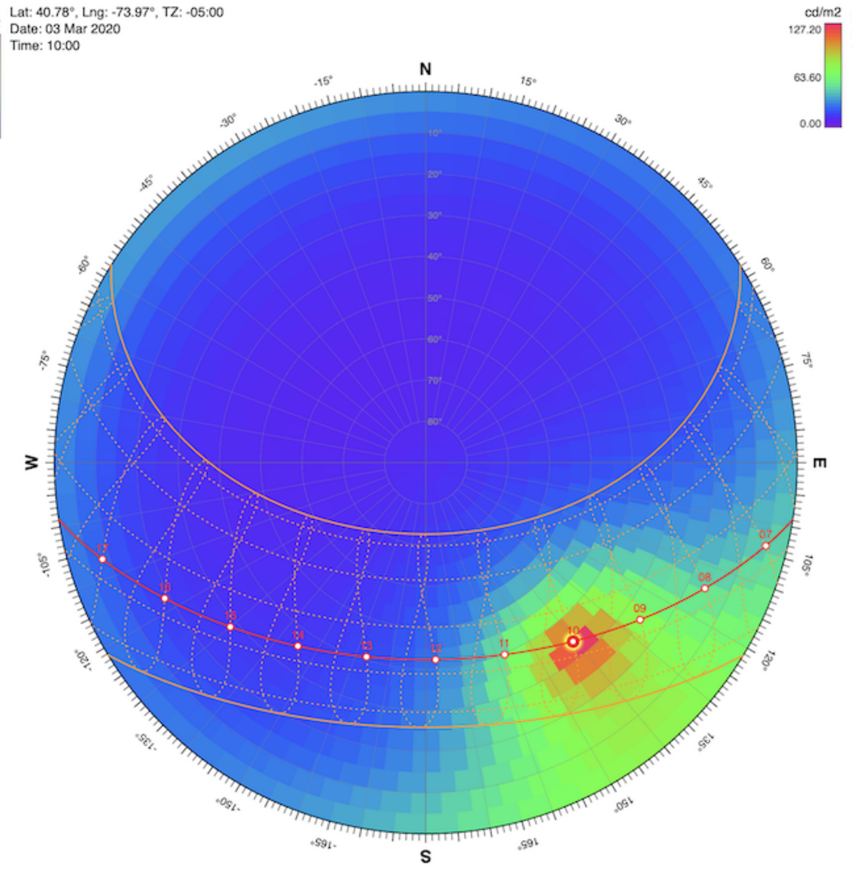


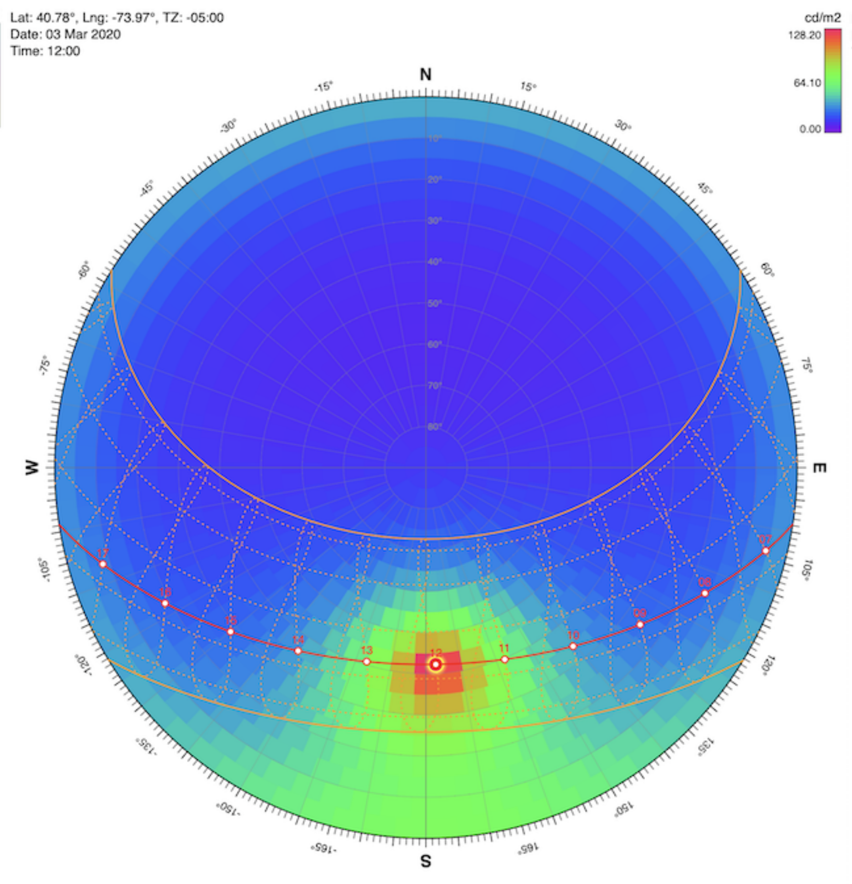


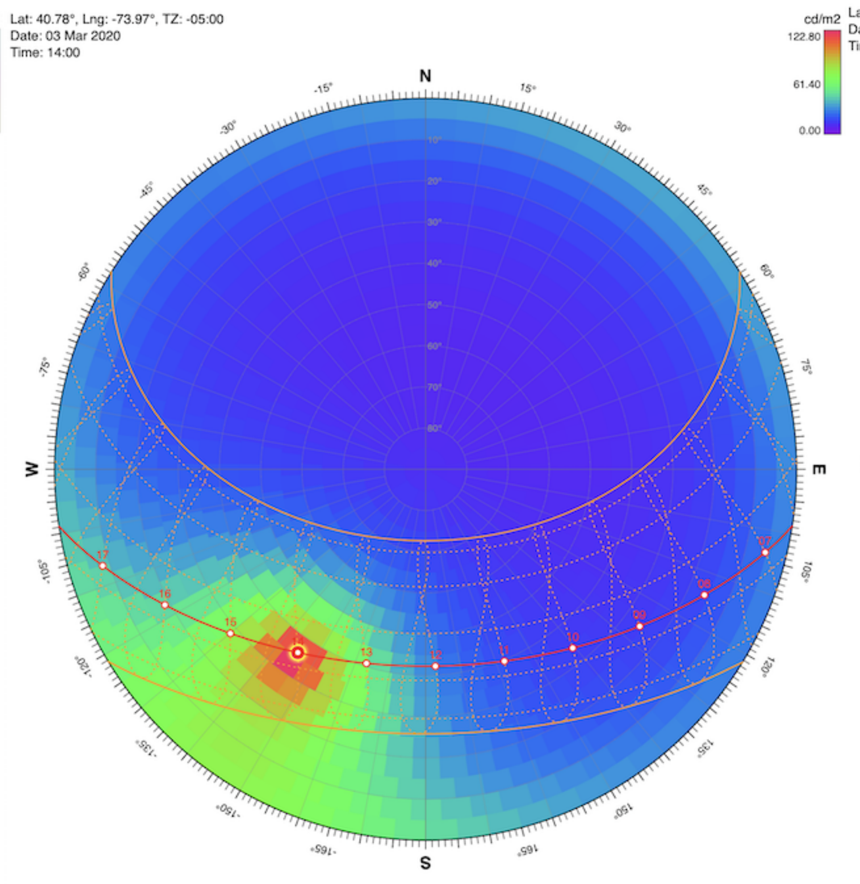












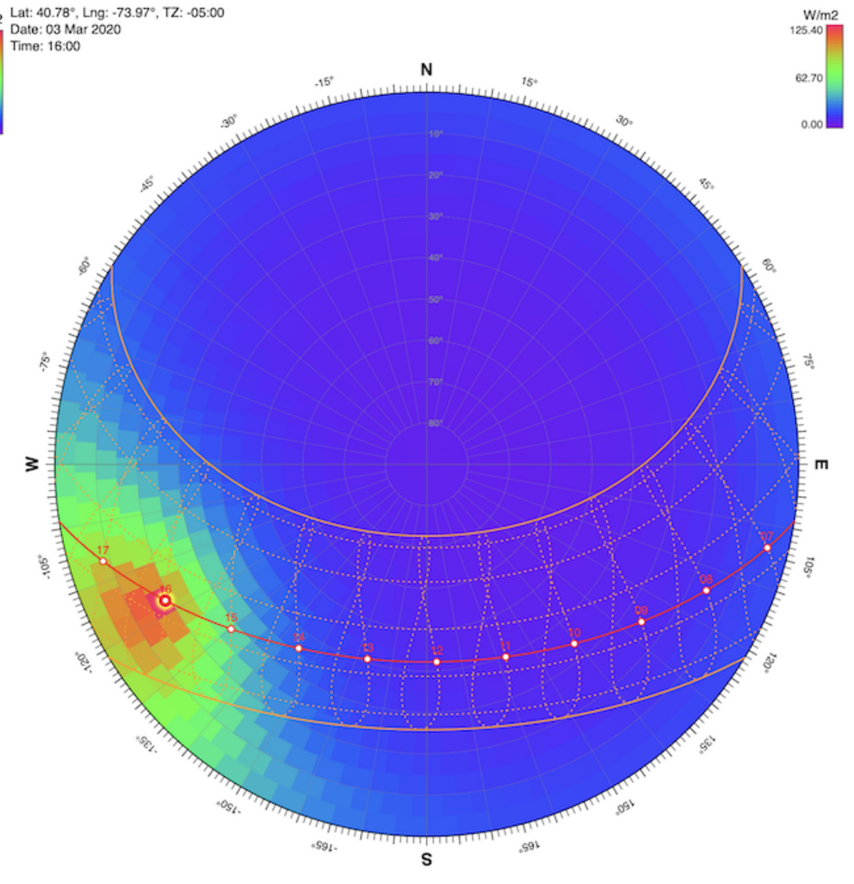
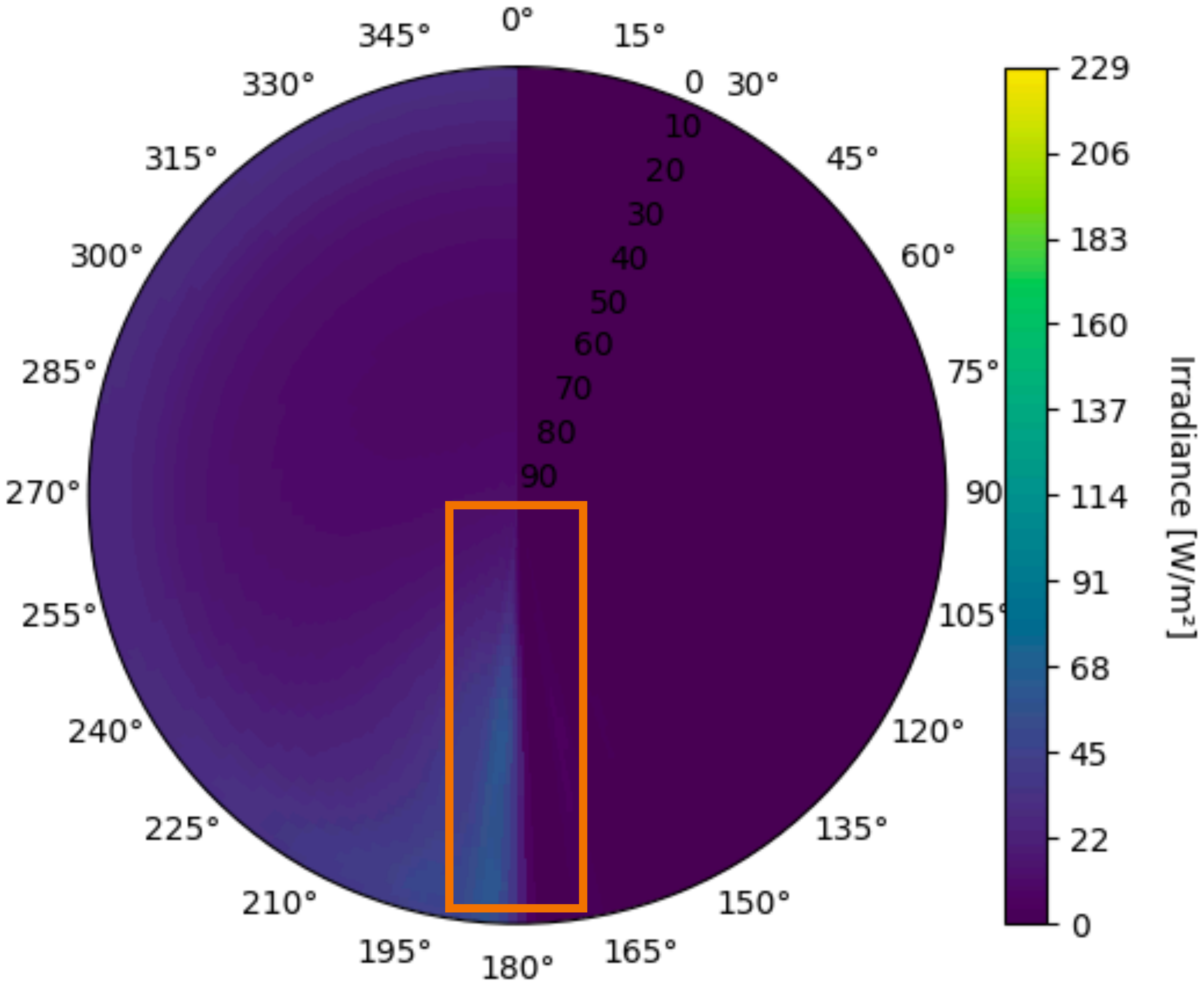
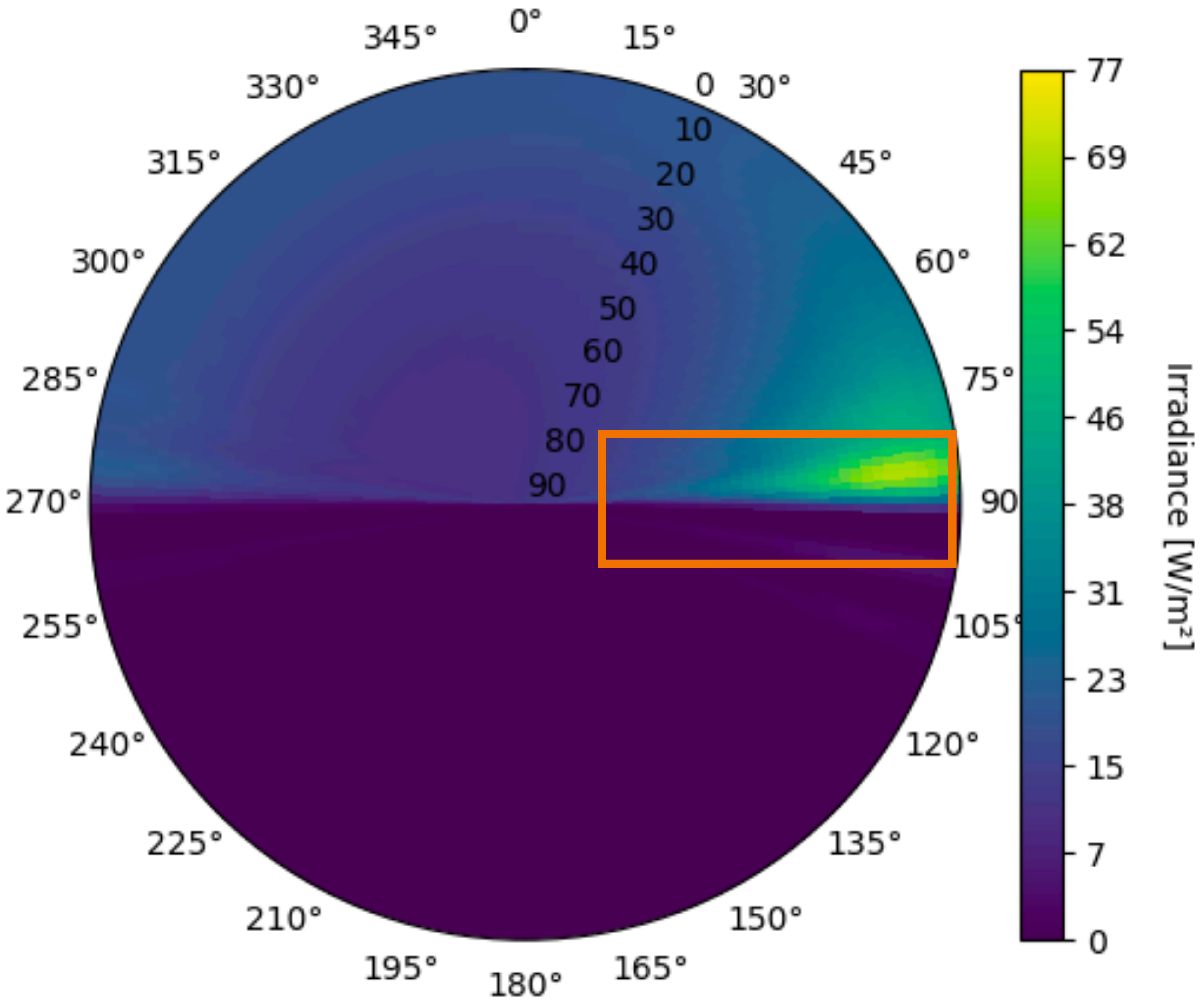


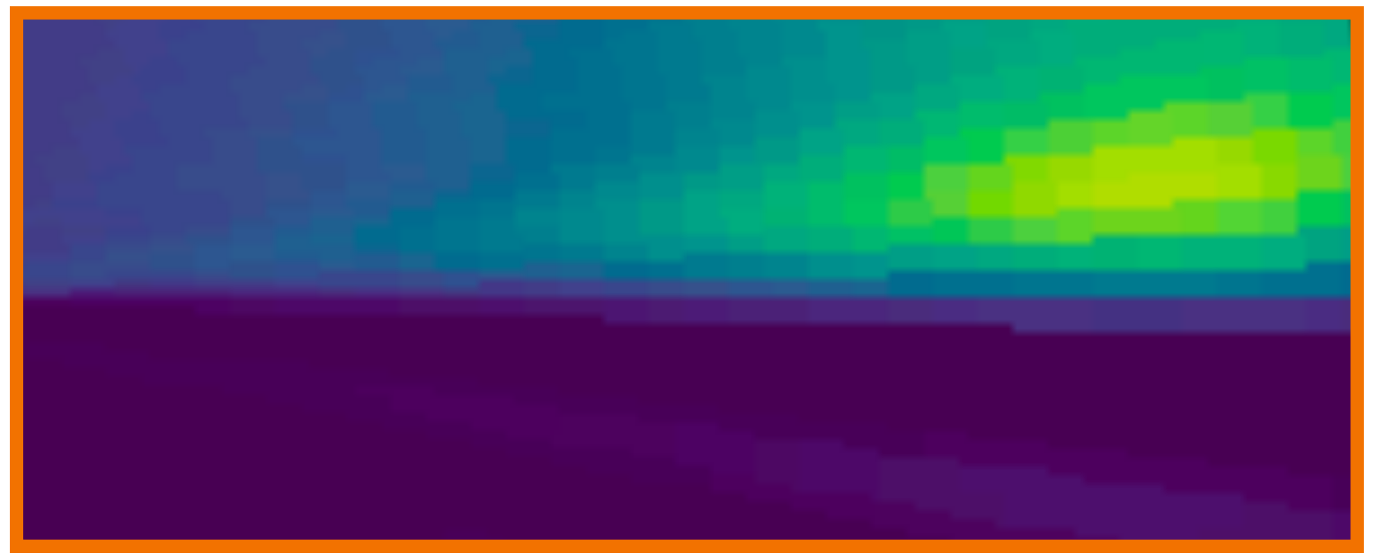
Image from Andrew Marsh' WebApp

Perez all-Weather Sky Distribution Model

Warnings







Conclusion and Summary of the Internship

Acknowledgments

Special thanks to my referent, L. Berti, who guided and helped me through this journey,

Of course to C. Prud’homme, who gave me the opportunity to flourish in a nurturing and enriching environment

And to P. LeMoine, dedicating hours of his time explaining Specx to me and the goals behind it

END

Shading Masks

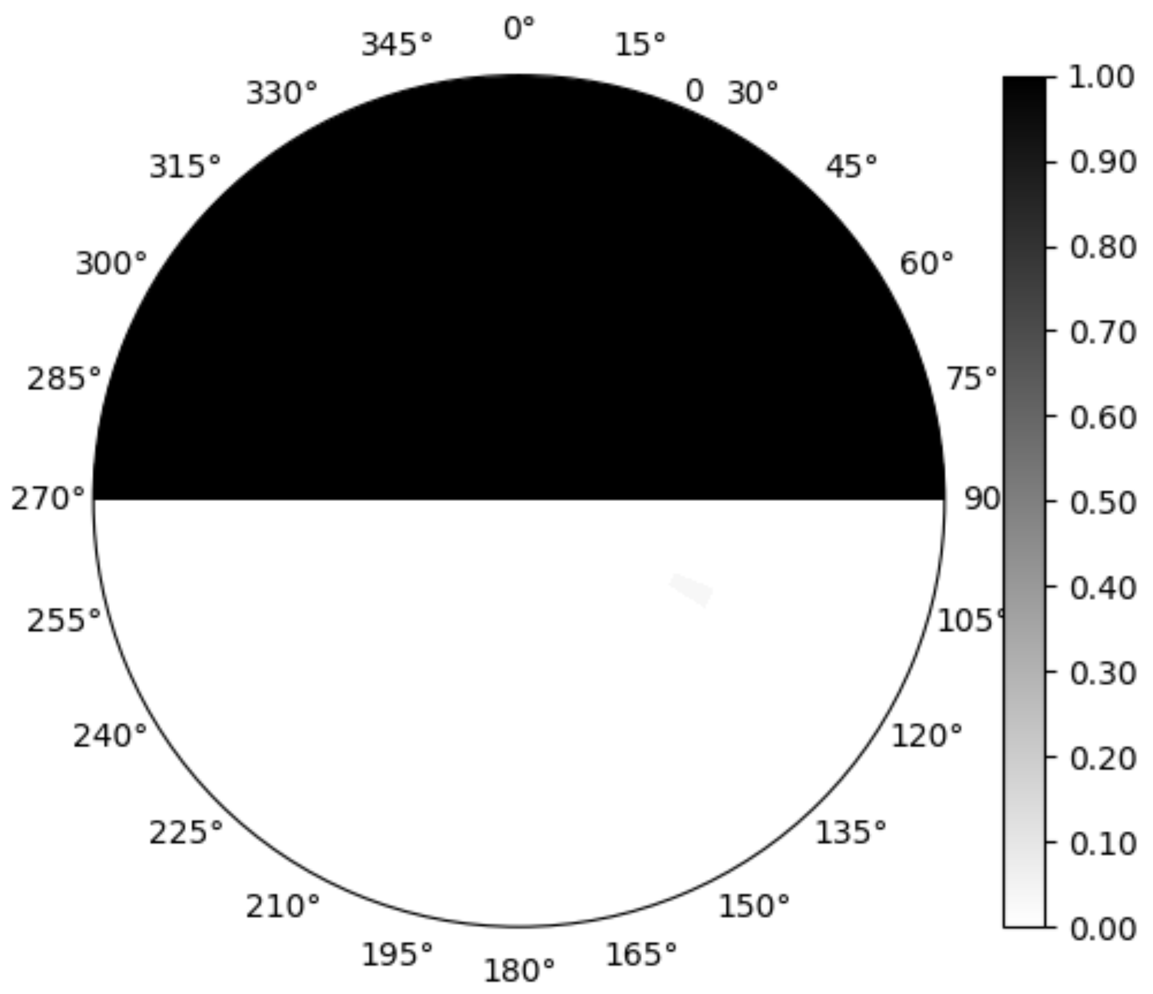
Optimizations

Validation of different RNGs

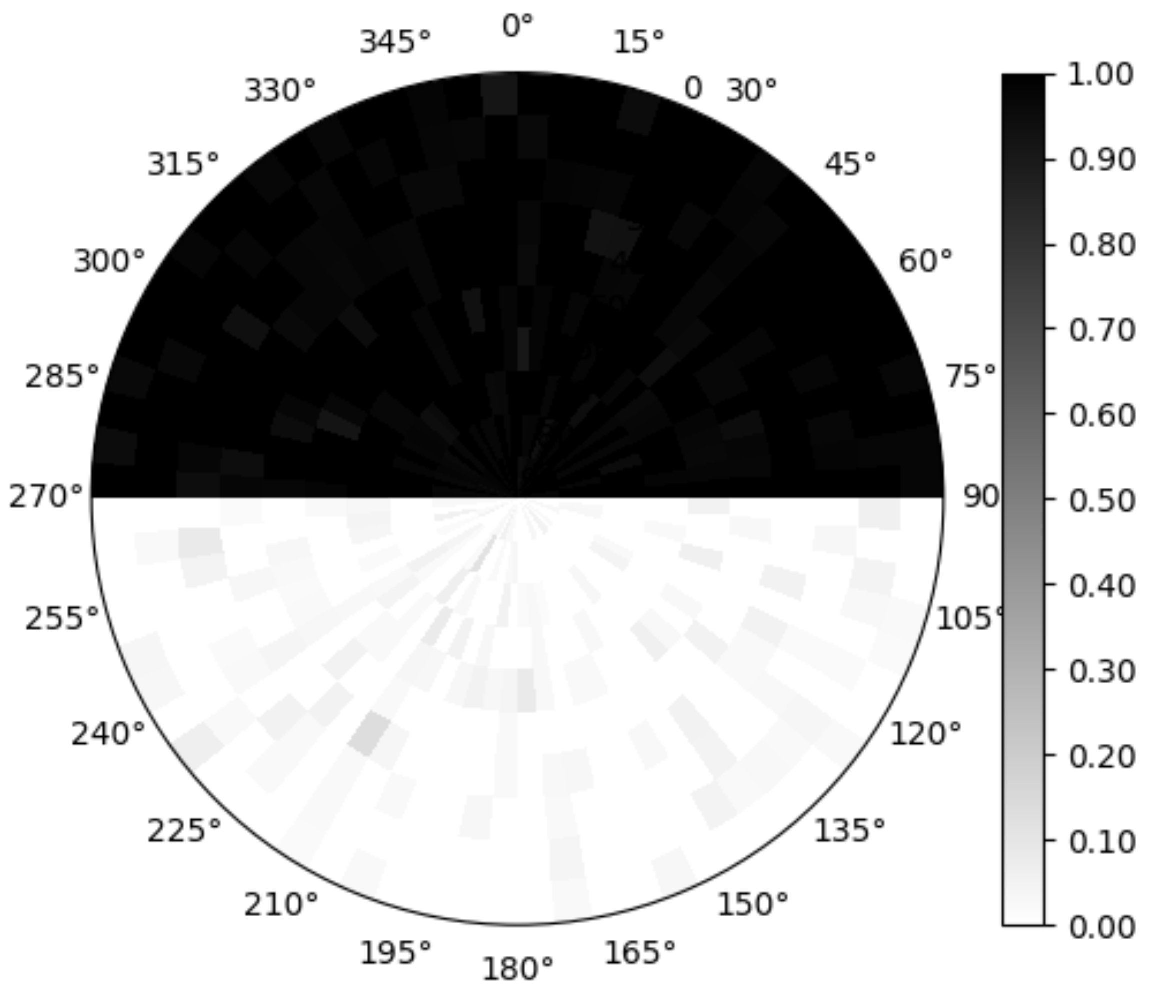
Case Bat1 South

```python
for matrix in matrix_files:
    # compare the firs part of the name and check if it corresponds
    # to the name of the base matrix appended with something
    if str(matrix.name)[:len(base_matrix.stem)] == base_matrix.stem:
        comparison_matrices.append(matrix)

for matrix in comparison_matrices:
    # Load comparison matrix
    comparison_data = np.genfromtxt(matrix, delimiter=',')

    # Compute difference
    diff_data = base_data - comparison_data
```

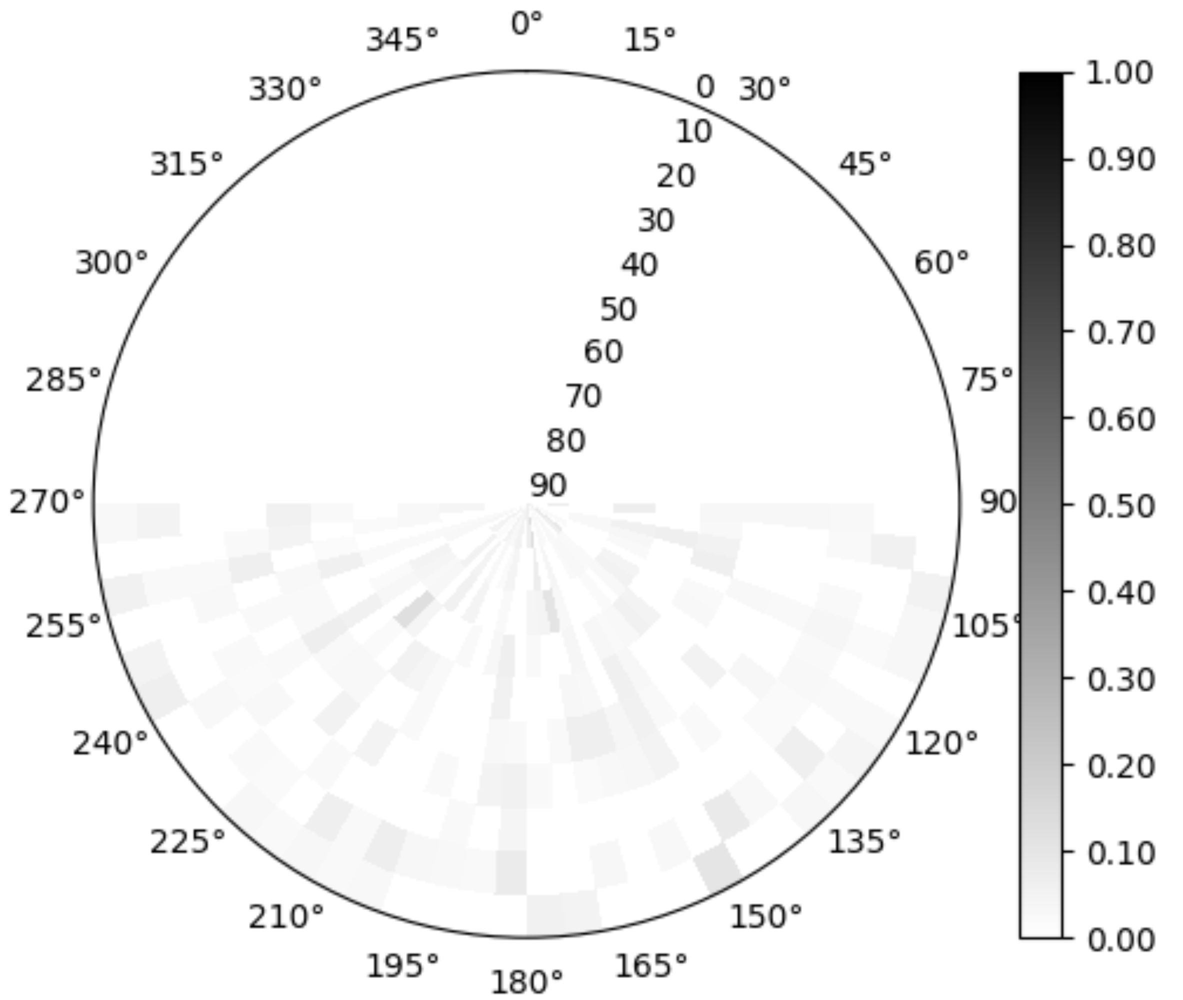
EigenRand



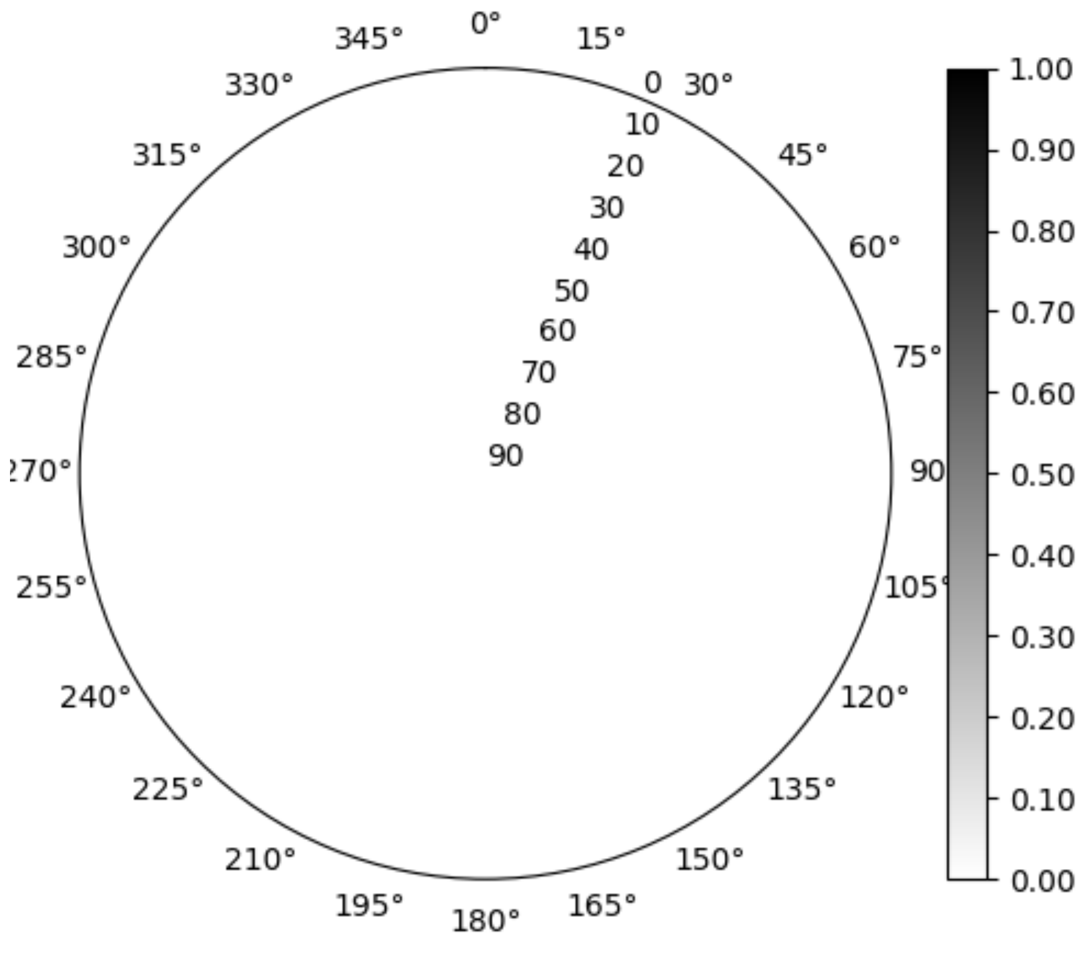
PCG



PCG



EigenRand



Perez all-Weather Sky Distribution Model

Implementations

$$lv = f(\zeta, \gamma) = (1 + a\exp(\frac{b}{\cos(\zeta)}))(1 + c\exp(d\gamma) + e\cos^2(\gamma))$$

$$x = x_1(\epsilon) + x_2(\epsilon)Z + \Delta(x_3(\epsilon) + x_4(\epsilon)Z)$$

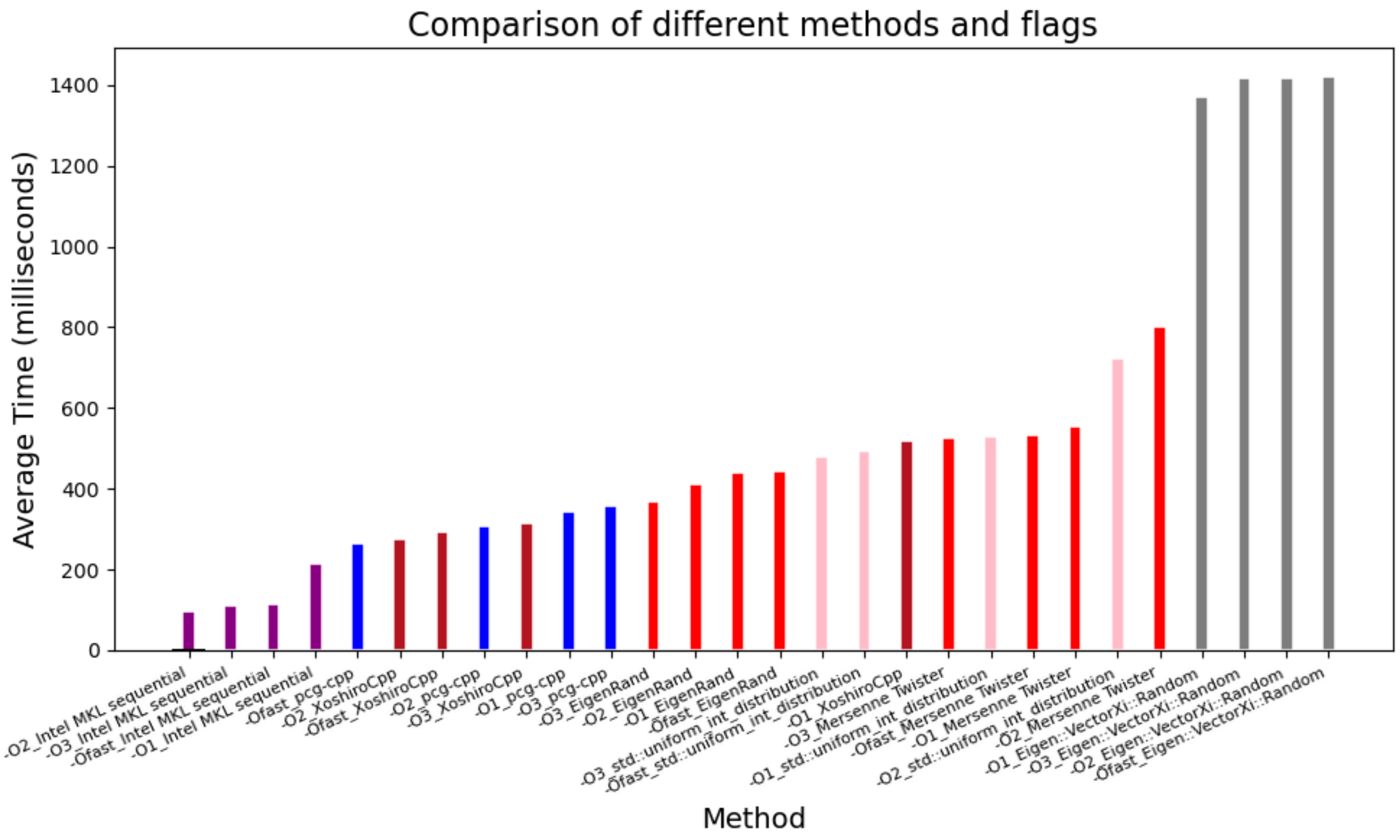
where $x \in \{a, b, c, d, e\}$

Where γ **is the angle between the sky element and the sun**

$$\cos(\gamma) = \cos(Z)\cos(z) + \sin(Z)\sin(z)\cos(\ A - a\ )$$

Shading Masks

Optimizations



Bounding Volume Hierarchy

Context / Previous Work

Speculative Traversal

Image and Pseudo-Code from Efficient Stack-less BVH Traversal for Ray Tracing