# Real-time Realistic Rendering and Lighting of Forests: Appendix

Eric Bruneton and Fabrice Neyret

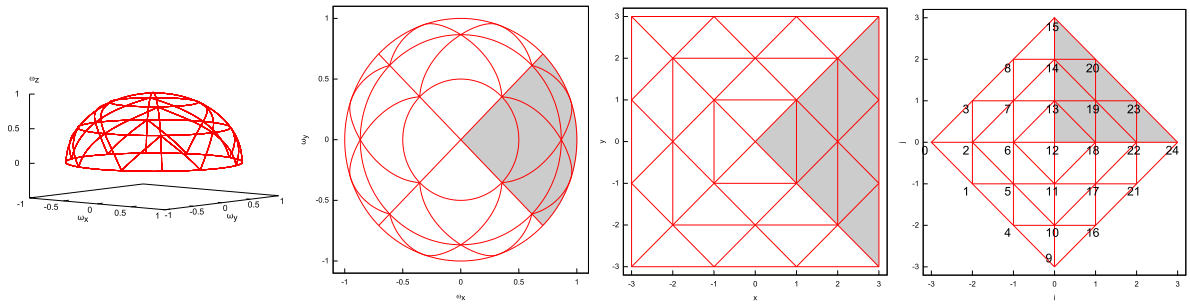


***Figure 1:*** *Partition of the hemisphere used for our precomputed views, for $n = 3$ (each vertex corresponds to a precomputed view direction).* From left to right*: hemisphere in perspective, viewed from top, after mapping to a square, and after scaling and rotation of this square.*

# A View interpolation

This appendix describes how we compute the 3 nearest precomputed views and the corresponding weights, in step 1 of our runtime shape reconstruction algorithm (see Section 5).

The idea of our method is to map the upper hemisphere (see Fig 1 left) to a square (see Fig. 1 right), and to compute the nearest views and the interpolation weights in this deformed space instead of on the hemisphere. Indeed, we can then decompose this square into a regular grid of triangles, so that computing the nearest view directions (i.e. the nearest grid vertices) and the corresponding interpolation weights (i.e. the barycentric coordinates of a point in a triangle) becomes trivial.

In practice, in order to get precomputed view directions that are almost uniformly distributed on the hemisphere, we do not decompose its square mapping into one regular triangle grid, but into four symetrical grids, one per quadrand (see Fig. 1 right). The four cases are very similar, and for clarity we present only one case here (in gray in Fig. 1. The full algorithm is presented in Section A.3).

The following sections define our mapping between the hemisphere and the square, and the partition of the square. The interpolation algorithm then follows naturally.

## A.1 Mapping and partition

Let $\boldsymbol{\omega} = [\omega_x, \omega_y, \omega_z]$ be a unit direction vector in the upper hemisphere ($\omega_z \geq 0$). We map the quadrand defined by $\omega_x > |\omega_y|$ into the square quadrant defined by (see Fig. 1):

$$x = \frac{2n}{\pi} \arccos(\omega_z) \tag{1}$$

$$y = \frac{\omega_y}{\omega_x} \frac{2n}{\pi} \arccos(\omega_z) \tag{2}$$

where $n$ is a natural number controlling the number of precomputed views (we use $n = 9$, yielding 181 views). We then scale and rotate this square quadrant to get a more convenient, axis aligned quadrant defined by:

$$i = \frac{x - y}{2} = n \left(1 - \frac{\omega_y}{\omega_x}\right) \frac{\arccos(\omega_z)}{\pi} \tag{3}$$

$$j = \frac{x + y}{2} = n \left(1 + \frac{\omega_y}{\omega_x}\right) \frac{\arccos(\omega_z)}{\pi} \tag{4}$$

## A.2 View interpolation

The 3 nearest views corresponding to $\boldsymbol{\omega}$, and the corresponding interpolation weights, are then easy

to compute. Let

$$w_i = i - \lfloor i \rfloor \tag{5}$$
$$w_j = j - \lfloor j \rfloor \tag{6}$$

then the nearest views are

$$(\lfloor i \rfloor, \lfloor j \rfloor) \text{ with weight } 1 - w_i - w_j \tag{7}$$
$$(\lfloor i \rfloor + 1, \lfloor j \rfloor) \text{ with weight } w_i \tag{8}$$
$$(\lfloor i \rfloor, \lfloor j \rfloor + 1) \text{ with weight } w_j \tag{9}$$

if $w_i + w_j < 1$, or

$$(\lfloor i \rfloor + 1, \lfloor j \rfloor + 1) \text{ with weight } w_i + w_j - 1 \tag{10}$$
$$(\lfloor i \rfloor + 1, \lfloor j \rfloor) \text{ with weight } 1 - w_j \tag{11}$$
$$(\lfloor i \rfloor, \lfloor j \rfloor + 1) \text{ with weight } 1 - w_i \tag{12}$$

otherwise.

## A.3 Full algorithm

The formulas for the other quadrants are similar. The four cases can thus be handled in a unified way, as shown below (GLSL code whose input is `W`=$\omega$ and whose outputs are the views `v` and the weights `w` – and where `N`=$n$):

```
bool q = abs(W.x) > abs(W.y);
float s = q ? W.y / W.x : -W.x / W.y;
float t = acos(W.z) * (N / PI);
float i = (1.0 - s) * t;
float j = (1.0 + s) * t;
int fi = int(floor(i));
int fj = int(floor(j));
float wi = i - fi;
float wj = j - fj;
float wk = 1.0 - wi - wj;
bool b = wk > 0.0;
int Q = int(sign(W.x + W.y));
ivec3 I = Q * ivec3(b ? fi : fi + 1, fi + 1, fi);
ivec3 J = Q * ivec3(b ? fj : fj + 1, fj, fj + 1);
v = q ? view(I,J) : view(-J,I);
w = b ? vec3(wk, wi, wj)
      : vec3(-wk, 1.0 - wj, 1.0 - wi);
```

where `view(i,j)` computes a view number between 0 and $2n(n+1)$ inclusive, as follows (see Fig. 1 right):

```
ivec3 view(ivec3 i, ivec3 j) {
  return i*(ivec3(2*N+1)-abs(i))+j+ivec3(N*(N+1));
}
```

These view numbers can then be used to access the precomputed view images in a 2D texture array.