Constant Time Queries on Uniformly Distributed Points on a Hemisphere

Mel Slater
Department of Computer Science
University College London, UK

Abstract

A set of uniformly distributed points on a hemisphere is generated using a popular method based on triangle subdivision. In applications, each data point (for example, representing a direction from a point on a surface) is typically associated with additional information (for example, a radiance value). Given an arbitrary query point on the hemisphere we require the nearest data point from the given distribution. An algorithm is presented that finds the data point in constant time, independently of the number of original points in the distribution. A portion of the hemisphere is rendered such that each point in the distribution has an associated set of quadrilaterals rendered with a unique color index for that point. The frame-buffer for the rendered hemisphere portion can be stored in off-screen memory. Any query point can be 'rendered' into this off-screen frame buffer, projected to a 'pixel' location, and the color index stored at this pixel location found. This color index is a lookup into an array of the original data points. This algorithm is presented in detail, and an illustrative implementation in OpenGL is described.

Keywords
Uniform distribution, sphere, hemisphere, queries, query-point

1. INTRODUCTION

A 'uniform' distribution of points on a sphere may be used to represent a set of directions from a point on a surface. This type of representation is important, for example, in global illumination methods such as radiosity, and also light field rendering, for example [1][2]. In general data points are generated on a hemisphere, but associated with each point is additional information - such as a radiance value. Now given some arbitrary query point on the hemisphere we need to retrieve the nearest data point and its associated information. Ideally such queries should be executed in constant time independent of the original number of data points.

There is no standard definition of what constitutes a 'uniform' distribution on a sphere, and this paper is not concerned with that issue. Methods are typically based on subdivisions of Platonic solids as discussed for example in [3] where a unit sphere is initially approximated by an icosahedron, and each constituent equilateral triangle is subdivided and projected to the surface of the sphere, recursively to a given depth. We adopt the method described [1] which is concerned with points distributed on a hemisphere. We approximate the hemisphere by the 'top' ($z \geq 0$) half of a regular octahedron, and again subdivide each constituent equilateral triangle, into four smaller triangles, project the vertices to the surface of the sphere, and repeat the operation recursively for each new triangle. At each level of recursion the triangle vertices will be 'uniformly' distributed over the hemisphere. We call these vertices the *data points* on the hemisphere. Now given an arbitrary query point on the hemisphere we require the nearest data point, in fact a pointer to information that includes the data point, since it is not usually the point itself that is of interest, but other data associated with it.

Although the points are uniformly distributed over the sphere there is no obvious way in which to quickly find the nearest to the query point. The points are not uniformly distributed when projected to the $z = 0$ plane, so that partitioning $-1 \leq x, y \leq 1$ into a regular grid and storing the data points in a 2-dimensional array corresponding to this grid would not be suitable. The data points either in 3D or in projected 2D could be stored in a K-D tree, but then of course there is no constant time search. In [3] a quad-tree type of data structure was used for searching, since each triangle is subdivided into 4 and therefore may be considered as the parent of its 4 sub-triangles. Again the

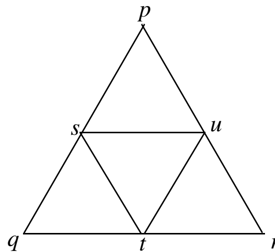
search time is dependent on the number of data points. [1] used the hemisphere subdivision as an alternative to a hemicube in a proposed radiosity solution. However, in order to find the directions corresponding to a patch, all relevant directions had to be compared with a patch bounding volume. Using the constant-time method described in this paper, a patch could be projected to a hemisphere in approximately the same time as projecting it to a hemicube - i.e., making use of a rasterisation operation with constant time lookup for the hemisphere elements.

In the next Section we describe the method for generating the data points in more detail. In Section 3 we describe a constant-time query method, and provide implementation details in Section 4.

2. Uniform Points on a Sphere

The vertices of the initial half octahedron in $z \geq 0$ space are the apex $(0, 0, 1)$ and $(1, 0, 0)$, $(0, 1, 0)$, $(-1, 0, 0)$ and $(0, -1, 0)$. All points are on the unit sphere centered at the origin. We consider the first quadrant only for the moment, with vertices $p(0, 0, 1)$, $q(1, 0, 0)$ and $r(0, 1, 0)$. Subdivide this triangle by bisecting each of the three sides, to form a set of 4 triangles, as shown in Figure 1, and project the mid-points to the surface of the sphere (in other words normalize them). Now treat each new triangle in the same way recursively.

FIGURE 1. Triangle Subdivision




```
triangle(Point3D p,Point3D q,Point3D r,int depth)
{
        Point3D pq,pr,qr,s,t,u;

        if(depth < MaxDepth){
                pq = (p+q)/2;
                s = normalize(pq);
                pr = (p+r)/2;
                u = normalize(pr);
                qr = (q+r)/2;
                t = normalize(qr);
                triangle(p,s,u,depth+1);
                triangle(s,q,t,depth+1);
                triangle(u,t,r,depth+1);
                triangle(s,t,u,depth+1);
        }
        else{
                report(p);
                report(q);
                report(r);
        }
}
```

A call of `triangle(p,q,r,0)` will produce the required points. Note that some points may be reported multiple times. The algorithm can be organized into a non-recursive one to avoid this, but this is not relevant to the problem of queries.

It is easy to calculate the number of distinct points that will be generated. Call the initial pqr triangle level 0. Then this has two rows, the first with one point (p) and the second with two points $(q \text{ and } r)$. Level 1 is shown in Figure 1 - this has three rows, each row with one more vertex than the next. Each time we go to another level we double the number of intervals on a triangle edge

(such as pq), and the number of points is one plus the number of intervals. Hence if at level l there are d points then at level $l+1$ there will be $2(d-1)+1$ points on each edge. Since at level 0 there are 2 points on each edge, at level l there will be 2^l+1 points. Another way to think of this is that the overall triangle consists of 2^l+1 rows of points, where the ith row consists of i points. Hence the total number of points at level l is:

$$1 + 2 + 3 + \ldots + (2^l + 1) = \frac{(2^l + 1)(2^l + 2)}{2} \tag{1}$$

Now this formulation will generate points on that 'one quarter' of the hemisphere bounded by the original pqr triangle. It is easy to cover all of the hemisphere (in fact the whole sphere) by rotations. We can partition the xy plane into five regions:

$$\begin{aligned}
&\text{Region 0: } x = 0 \text{ and } y = 0 \\
&\text{Region 1: } x > 0 \text{ and } y \ge 0 \\
&\text{Region 2: } x \le 0 \text{ and } y > 0 \\
&\text{Region 3: } x < 0 \text{ and } y \le 0 \\
&\text{Region 4: } x \ge 0 \text{ and } y < 0
\end{aligned} \tag{2}$$

The algorithm above only generates points in Regions 0,1 and the first column of Region 2. However, given a generated data point in Region 1 we can find equivalent points in regions 2, 3 and 4 by rotating it by multiples of 90 degrees around the z-axis.

Next consider storing each unique point in an array for later lookup or other processing. We represent the generated points as if in a triangular array. For example, the points in Regions 0, 1 and the first column of Region 2 would be represented as follows:

$$\begin{aligned}
&p_{00} \\
&p_{10}, p_{11} \\
&p_{20}, p_{21}, p_{22} \\
&\ldots \\
&p_{m0}, p_{m1}, \ldots, p_{mm}
\end{aligned} \tag{3}$$

where $m = 2^l$ and l is the number of levels (or depth of recursion). If we were to generate points to depth l then the initial three points would be: $p = p_{00}, q = p_{m0}, r = p_{mm}$. Now suppose that p_{ij} and p_{kl} are two points on an edge that is to be bisected, then the mid-point of the edge projected onto the sphere would be:

$$p_{\frac{i+k}{2}, \frac{j+l}{2}} = \text{normalize}\left(\frac{p_{ij} + p_{kl}}{2}\right) \tag{4}$$

where 'normalize' projects the point to the surface of the unit sphere.

According to (2) the data points are assigned to the regions are as follows:

$$\begin{aligned}
&\text{Region 0: } p_{00} \\
&\text{Region 1: } p_{ij},\ i \ge 1,\ 0 \le j < i \\
&\text{Region 2: } p_{ij},\ i \ge 1,\ i \le j < 2i \\
&\text{Region 3: } p_{ij},\ i \ge 1,\ 2i \le j < 3i \\
&\text{Region 4: } p_{ij},\ i \ge 1,\ 3i \le j < 4i
\end{aligned} \tag{5}$$

These triangular array coordinates are flattened into a linear array $D_k, k = 0, 1, \ldots$ which stores the data points generated for all 5 regions. We assign $D_0 = p_{00}$, and then the points in Region 1 are assigned to the next contiguous block of array positions, followed by all the points of Region 2 in the next block, and so on. An example for a level 2 subdivision is shown in Figure 2, where the triangular array indices are shown in the lighter font, and the corresponding position in the flattened

array in bold. In this scheme, the number of data points assigned to Region 1 is therefore 1 for p_{10}, two for the next row p_{20}, p_{21}, three for the next row p_{30}, p_{31}, p_{32}, and overall:

$$M = 1 + 2 + \ldots + m = \frac{m(m+1)}{2} \tag{6}$$

The point p_{ij} in Regions 0 or 1 will have:

$$1 + 1 + 2 + \ldots + (i-1) = 1 + \frac{i(i-1)}{2} \tag{7}$$

elements in the rows 'above' it, the first is for p_{00}, then for p_{11}, and so on for each row up to row $(i-1)$. Hence for any point p_{ij} in Region 1, we have

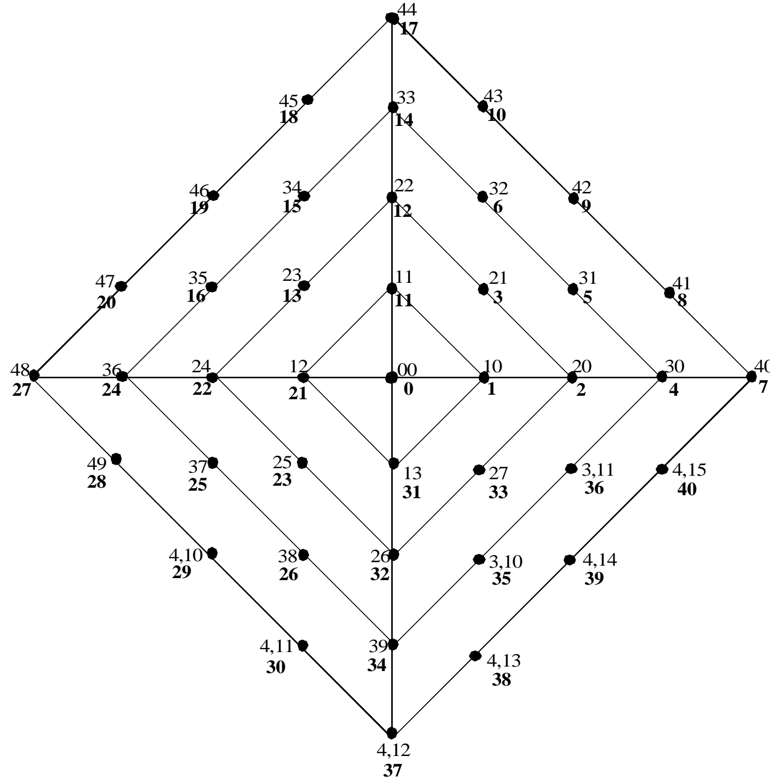
$$D_{1 + \frac{i(i-1)}{2} + j} = p_{ij} \tag{8}$$

For example, $p_{32} = D_6$ as shown in Figure 2. Given the array position k of a point in Region 1, the equivalent points in Regions 2, 3 and 4 would, using (6), be allocated to array positions $k + R \cdot M$ for $R = 1, 2, 3$. For example, in Figure 2, $M = 10$, and the equivalent (rotated) points to $p_{42} = D_9$ are D_{19}, D_{29}, D_{39} respectively for Regions 2,3 and 4. However, in order to compute the position corresponding to points with $i = j$, we must use:

$$D_{1 + \frac{i(i-1)}{2} + M} = p_{ii} \tag{9}$$

since p_{ii} is equivalent to p_{i0} in Region 1.

FIGURE 2. Assignment of Triangular Array Elements to a Flattened Array



The new version of `triangle` will be as follows:

```
triangle(Point3D p, Address ap, Point3D q,Address aq,
        Point3D r, Address ar, int depth)
{
       Point3D pq,pr,qr,s,t,u;
       Address as,au,at;

       if(depth < MaxDepth){
             pq = (p+q)/2; as = (ap+aq)/2;
             s = normalize(pq);
             pr = (p+r)/2; au = (ap+ar)/2;
             u = normalize(pr);
             qr = (q+r)/2; at = (ap+ar)/2;
             t = normalize(qr);
             triangle(p,ap,s,as,u,au,depth+1);
             triangle(s,as,q,aq,t,at,depth+1);
             triangle(u,au,t,at,r,ar,depth+1);
             triangle(s,as,t,at,u,au,depth+1);
       }
       else{
             putDirection(p,ap);
             putDirection(q,aq);
             putDirection(r,ar);
       }
}

void putDirection(Point3D p, Address ap)
{
       int j = getIndex(ap);

       /*shown only for p in Region 1*/
       D[j]          = p;
       D[j+M]        = rotate90(p);
       D[j+2*M]      = rotate180(p);
       D[j+3*M]      = rotate270(p);
}
```

Here `Address` is a tuple consisting of the `i,j` indices of the conceptual triangular array. The function `getIndex` takes an address and returns the position in the array using (8) and (9). The function `putDirection` is responsible for allocating points to the array `D`. What is shown is simplified since it does not take into account boundary conditions such as `i=j=0`, points with `i=j` and so on. The full implementation is shown in the code accompanying the paper[1].

3. Querying the Points on the Hemisphere

Now given the distribution of points, how would we execute a query? - i.e., given an arbitrary point q on the hemisphere, find the array index of the nearest point amongst the distribution of data points. Restrict q to be in Region 0 or 1 for the moment. The fundamental idea is to render the hemisphere ‘quarter’ of Region 1 into the frame buffer, such that each generated data point has an associated set of polygons representing all the points on the hemisphere closest to that point. Each such polygon is rendered with a ‘color index’ which corresponds exactly to the array index of the data point. The frame buffer is read back into a 2-dimensional array. Now given any point q we compute the pixel position to which it would be projected. We use that to look up the color index at that position in the buffer, which gives us the closest corresponding generated point by way of the array look-up.

Figure 3 shows an example of how a triangle would be rendered. c is the mid-point of the triangle, and i_p, i_q, i_r are the array indices of the vertices p, q and r. Hence associated with each vertex is a quadrilateral, and this is rendered with color index corresponding to the array index of that vertex. After rendering, the frame buffer is read back into an array called, say, `Pixel`, where `Pixel[WindowSize*y+x]` is the color index for pixel position (`x,y`), and assuming a square window of width

1. http://www.cs.ucl.ac.uk/staff/m.slater/Papers/Sphere

`WindowSize`. Now any query point is projected to the display in the sense that its (`x`,`y`) pixel position is computed (of course using the same viewing parameters as those used to create the framebuffer), and then its color index can be looked up in the `Pixel` array.

If the point is not in Region 0 or 1, then it may be rotated back to Region 1, the color index for Region 1 found, and then the appropriate multiple of M (6) used to find the array position appropriate to its region.

The quarter hemisphere must, of course, be rendered such that each data point and its associated quadrilateral is visible. This can be achieved for example, with a view direction through the origin and the centre of the initial triangle $(1/3, 1/3, 1/3)$, looking at the inside of the hemisphere quadrant. This is shown in Figure 4 for the case $l = 5$.

FIGURE 3. A Triangle is Partitioned into Quadrilaterals

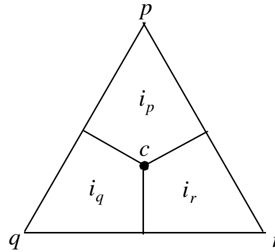
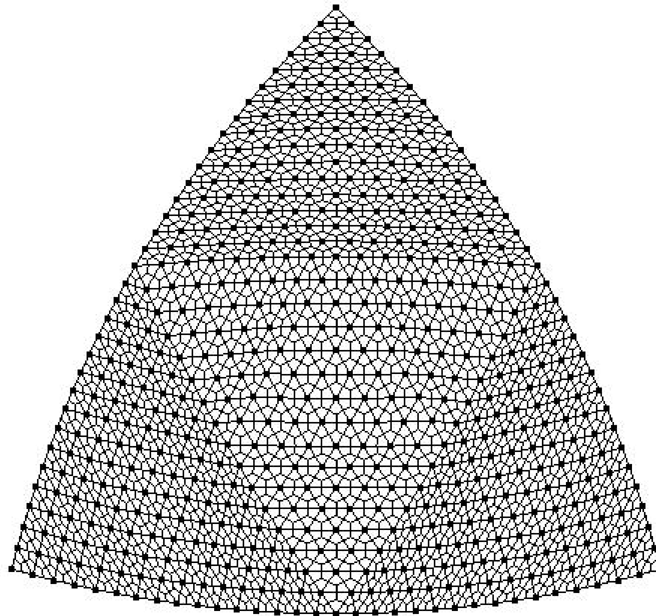


FIGURE 4. The Quadrant of the Hemisphere



4. Implementation Issues

This algorithm can be implemented in OpenGL, and an illustrative implementation is provided. When the quadrant of the sphere is first rendered the modelview and projection matrices are immediately recorded and multiplied together and the result stored for later use with query points. Given a query point, this matrix is used to multiply the point delivering a point in homogeneous coordinates, which is divided by its w coordinate to perform the projection. The projected point is then transformed to window coordinates. Color index 0 should be reserved for black, and all positive

color indices may be used to index points, hence the actual color index should be 1 plus the array index.

When the quadrant is rendered the framebuffer is immediately read into memory using glReadPixels. This is then the two-dimensional array of color index values. Using the OpenGL color index, however, may not always be appropriate since the supported number of color index values may be less than what is required for a given level of recursion. The total number generated data points is $1 + 2m(m + 1)$ (using (6)). However, the maximum color index required is that corresponding to the point p_{mm} which is $1 + m^2$. For 6 levels of recursion the number of points generated on the sphere is 8321. On the SGI O2 on which the algorithm was implemented the maximum color index was 4096. On other systems it might be considerably less than this. Instead of using the color index as such, the array indices can be stored in RGB values, packed into the red, green and blue fields.

The algorithm may not report correct results in two circumstances. First, a point not actually on the hemisphere may nevertheless return an index value - simply because it is projected into the framebuffer at a point where there is a non-zero color index. This should not be a problem provided that the calling program always provides points known to be on the hemisphere, which would normally be the case. Second, the rendered quadrilaterals are not exactly on the sphere of course. Hence it might be the case that a point on the hemisphere near the edges of Region 1 might project to just outside the rendered region and pick up a zero color index. In order to overcome this problem, if it occurs, the neighboring pixel positions are examined, and the one containing the data point which has the highest correlation with the query point is returned.

The implementation tested 1,000,000 pseudo-randomly generated query points at various levels of recursion. The results are shown in Table 1. The ‘non results’ column refers to the number of times out of 1,000,000 that the query point was projected to a black region of the framebuffer, and all its surrounding pixels were also black. As can be seen, this event is rare, but it is expected that this would occur more often for the lowest levels of recursion, since the approximation of the sphere by the quadrilaterals is at its worst. The correlation is the minimum correlation between query point and returned data point amongst all the valid results. Of course the time to execute the queries was the same for all levels of recursion.

Table 1: Results for 1,000,000 Randomly Selected Query Points at Various Levels of Recursion

level	No. of Data Points	No. of Non-Results	Minimum Correlation
3	145	48	0.9841
4	545	0	0.9958
5	2113	0	0.9988
6	8321	0	0.9996
7	33025	0	0.9999

References

[1] Gatenby, Neil; Hewitt, Terry. Radiosity in Computer Graphics: A Proposed Alternative to the Hemi-Cube Algorithm. In Second Eurographics Workshop on Rendering, Barcelona, 1991. Also reprinted in "Photorealistic Rendering in Computer Graphics", Springer-Verlag, pp104-111.

[2] Camahort, E., Lerios, A., Fussell, D. (1998) Uniformly Sampled Light Fields, Rendering Techniques'98, 117-130.

[3] Giraldo, F.X. (1997) Lagrange-Galerkin Methods on Spherical Geodesic Grids, Journal of Computational Physics, 136, 107-213.

Acknowledgements

This research is part of work funded by the UK EPSRC Senior Research Fellowship of the author. Thanks to Franco Tecchia, Yiorgos Chrysanthou, Jesper Mortensen, Pankaj Khanna, and Insu Yu for helpful comments and suggestions.