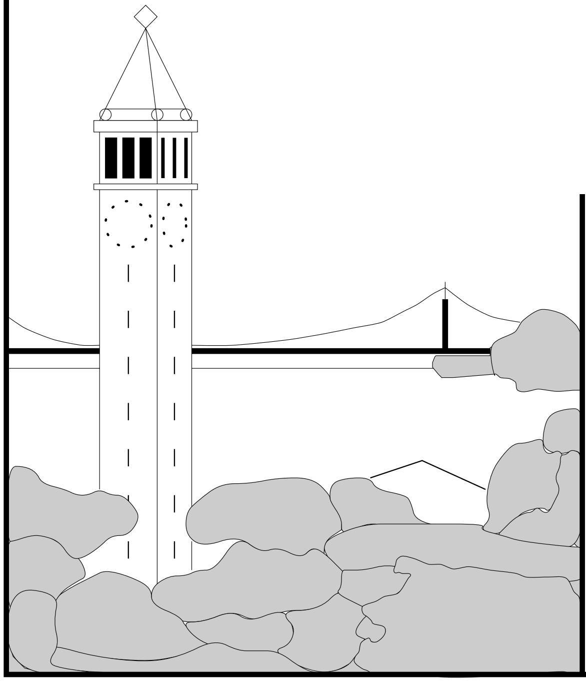
A Dynamic Mesh Display Representation for the Holodeck Ray Cache System

Maryann Simmons



Report No. UCB/CSD-00-1090

January 2000

Computer Science Division (EECS)
University of California
Berkeley, California 94720

A Dynamic Mesh Display Representation for the Holodeck Ray Cache System

Technical Report UCB/CSD-00-1090

Maryann Simmons [†]
simmons@cs.berkeley.edu

January 2000

This report presents a dynamic mesh representation that is designed to display the results of interactively sampling a 3D environment. The mesh generator takes samples comprising a world space location, RGB color value, and sampled ray direction, and assembles them into a 3D triangle mesh. From this representation a complete image can be constructed and displayed, both from the initial view, and from subsequent views as the user moves through the environment. The representation exploits the fact that from a fixed vantage point there is a one-to-one mapping between visible world space points and their projection onto a sphere centered at that viewpoint. A Delaunay triangulation constructed on the sphere provides the mesh topology and the the vertex coordinates are derived from the input samples. The resulting mesh is used as the 2.5D display representation. The mesh is dynamic and ephemeral: it is updated as samples are added and deleted, and reconstructed after significant viewpoint changes.

The dynamic mesh representation is described in the context of an interactive rendering system based on the *holodeck*, a 4-dimensional ray-caching data structure. In the holodeck system, a display driver makes requests for ray samples based on the user's current view. The display driver must then quickly construct a coherent image based on these samples. This report introduces the dynamic mesh representation as a solution to the reconstruction problem, describes its implementation, and presents the results of utilizing this representation in the holodeck environment.

Keywords: surface reconstruction, image reconstruction, Delaunay mesh, ray tracing

[†] This work was done at Silicon Graphics in collaboration with Greg Ward and at U.C. Berkeley under the guidance of Carlo H. Séquin. This work was partially supported by ONR MURI grant #FDN00014-96-1-1200.

Contents

1 Introduction

The interactive, realistic rendering of complex environments, real and virtual, is a longstanding goal in the field of computer graphics. We have the ability to calculate full global illumination solutions, producing the diffuse and view-dependent effects necessary to impart a realistic percept of an environment. Real-world environments can be captured via scanners and photographs utilizing computer vision techniques. Visibility algorithms and graphics hardware advances allow the interactive exploration of complex virtual environments. Incorporating all of these capabilities into a single system, however, continues to pose a challenge.

Interactive ray-tracers offer one class of solutions to this problem. One such method is based on the *holodeck* [6, 17], a four-dimensional ray-caching data structure which serves as a rendering target and caching mechanism for interactive walk-throughs of non-diffuse environments with full global illumination. In the holodeck system, a display driver makes requests for ray samples based on the user's current view. A server gathers the relevant samples, first sending those that are cached in the holodeck from previous views, and then requesting additional rays to be traced for the current view. Ray samples are generated by the *Radiance* lighting simulation system [8, 16]. The display driver takes the requested samples from the server and converts them into a suitable display representation. This requires mapping world floating-point colors to displayable RGBs and constructing a coherent image. The driver must continue to update this image during progressive refinement as new values are computed by the sample generator and passed on by the server. The system expects a certain interactive rhythm from the user, in the form of slow, inertial view changes, with stationary observation phases in between. During view motion, no new samples are provided by the server and the display driver must make do with the existing representation to provide sufficient visual feedback.

This report presents a dynamic mesh representation as a solution to the reconstruction problem, describes its implementation, and presents the results of utilizing this representation in an interactive rendering system based on the holodeck ray-cache. The mesh generator takes samples comprising a world space location, RGB color value, and sampled ray direction, and assembles them into a 3D triangle mesh. From this representation a complete image can be constructed and displayed, both from the initial view, and from subsequent views as the user moves through the environment. Gouraud-shaded triangles are displayed both during motion and afterwards during progressive refinement. The advantage of such a representation is threefold. Firstly, triangles are rendered and Gouraud-shaded by the graphics hardware, providing barycentric interpolation of radiance between spatially adjacent samples. Secondly, since the representation explicitly contains the 3D information for each sample, and not just a representation of the projections for a particular view, the rendering representation can be re-used between frames for small view motions. Finally, since the generator utilizes the sample data exclusively to build the representation, it can be used without additional or a priori knowledge of the scene geometry.

The representation exploits the fact that from a fixed vantage point there is a one-to-one mapping between visible world space points and their projection onto a sphere centered at that viewpoint. In this sense the samples form a height field, and we can reduce the 3D meshing problem to a 2D triangulation on the sphere. A Delaunay mesh constructed on the sphere provides the mesh topology and the vertex coordinates are derived from the input samples. The resulting mesh is used as the 2.5D display representation. We maintain the Delaunay condition on the mesh to improve both image quality and robustness of the meshing code. Such a triangulation provides a reasonable interpolation and maximizes the minimum angle and therefore minimizes rendering artifacts caused by long thin triangles. Such triangles could also prove problematic during the computation and manipulation of the mesh as they are prone to producing topological inconsistencies due to round-off errors in the calculations.

The mesh contains sufficient information to render the scene from any viewing configuration based at the initial or "canonical" viewpoint from which the mesh is constructed. If the observer moves slightly off the viewpoint, the same mesh is maintained and used as the rendering representation. The sample points are re-projected as the mesh is transformed and rendered by the graphics hardware. In the case of small view motions without significant visibility changes, the resulting image will not suffer noticeable artifacts. Once the observer moves a significant distance from the canonical viewpoint, the current mesh is discarded and a new mesh is constructed with the current sample set projected relative to the new canonical viewpoint. To make this reconstruction faster, only those samples that fall into the current view frustum are added to the new mesh. The mesh is therefore dynamic and ephemeral: it is updated as samples are added and deleted, and reconstructed after significant viewpoint changes.

The remainder of this section describes the format of the input samples that are received from the driver and from which the display representation must be progressively constructed. The rest of the paper describes the dynamic mesh representation, and algorithms for its construction and display. Results of utilizing the dynamic mesh representation in the holodeck environment are presented, and the representation is evaluated in this context. The report concludes with ideas for improvements and future work. Appendix A includes a specification of the display driver interface for the holodeck system.

1.1 Input

As the user moves about in an environment visualized using the holodeck system, samples are fetched from cache for each new view and sent to the display driver. New rays are then generated by any number of ray tracing processes and also passed on to the display driver. In this system, the display driver has no control over how many, or what samples it receives at any given time. The display generation process must be able to assimilate whatever samples it is given as quickly as possible into a coherent image for display, which should progressively refine as more samples arrive.

For each sample in a given view, the driver provides the 3D coordinates, a 4 component (RGBE [15]) color value, and a ray direction. The coordinate value is the intersection point of the ray with the environment, and can represent a world space point or a direction, in the case that the intersection occurs at infinity (or effectively so). A tone-mapping operation [7] is periodically applied to the color value to generate a renderable RGB value based on the current set of samples, allowing input samples with high dynamic range to be displayed without perceived loss of visual information. The ray direction is the direction from which the intersection point was calculated. Because the holodeck server utilizes a cache and re-uses rays for alternate views, this information is not redundant. Samples returned by the server may not pass exactly through the current eye point. The ray direction can be used to determine how close the sample ray passed to the actual ray from the current viewpoint as a measure of relative sample quality.

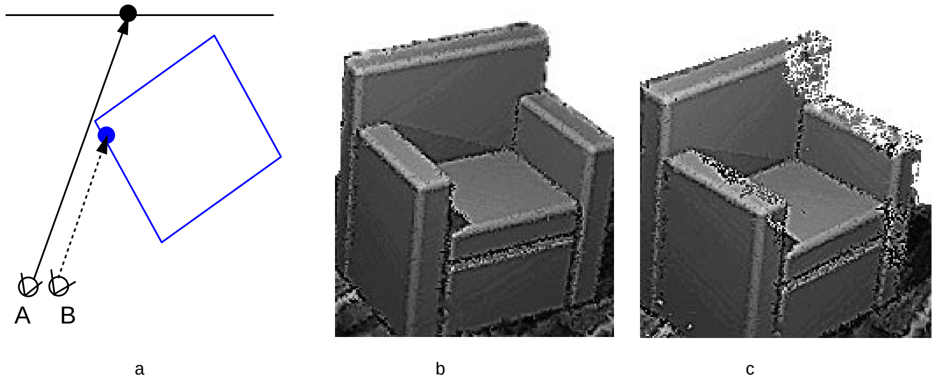


Figure 1: Multi-depth values: a) A viewing situation in plan view. The solid ray returns the correct visible sample for view A. If the same ray is re-used for view B, it can return a point that should be occluded. b) An example reconstruction using only rays from the current view. c) The view is pivoted about the chair, and rays are re-used. Multi-depth artifacts can be seen along the chair's silhouette.

In the case of a sample with an intersection point at infinity, the direction value passed is set to invalid, and the coordinate value contains the ray direction. These *directional* samples represent background scenery. In the rest of this report such samples are referred to as *background* points to differentiate them from the world space, or *foreground* points. Background samples are incorporated directly into the mesh with the world space points, but require special processing during re-projection and rendering.

When the ray sample is re-projected to the expected position in a new view, we may get a sample that is actually behind a foreground object when we re-project its location (see Figure 1a). Such *multi-depth* samples occur near object silhouettes and can produce visible artifacts in the rendering, appearing as small holes piercing through the object's visible boundary (see Figure 1c). Even when the multi-depth sampling is minimal, the artifact can still be marked, given the human visual system's sensitivity to silhouette boundaries. As more rays are traced during progressive refinement, sufficiently many correct samples will eventually be generated and will replace the incorrect ones in the representation.

The following sections discuss techniques for generating and rendering meshes for display based upon the type of input described above.

2 Mesh Construction

Given the holodeck environment, we need a mesh data structure that supports the following operations:

- Fast insertion of a new sample into the mesh: requires point location.
- Deletion of samples: not as common - but also must be fast
- View frustum culling

Given a specified view and a set of samples, the goal of representation generation is to quickly display a coherent image that is a viable representation of the potentially sparse set of sample points. The representation should support progressive refinement while the view is stationary and provide reasonable interactivity during motion. There is clearly a tradeoff between the quality of the resulting image and the time spent during reconstruction. We have chosen a 3D Gouraud-shaded mesh as the base display representation. With such a primitive we can exploit existing graphics hardware to do simple and fast interpolation of sample values, rendering of the mesh, and view transformations during motion, re-using the same representation between adjacent views.

Given a single view, other researchers have chosen to build a two-dimensional Delaunay triangulation of the projected samples on the image plane as a solution to the reconstruction problem [2, 10, 13]. While this basic approach provides a reasonable barycentric interpolation of the radiance values, the samples must be re-projected and the mesh reconstructed each frame to provide a coherent image during viewer motion. Alternate approaches utilize the projected points to determine the mesh topology in two dimensions, but retain the original 3D information in the resulting vertices, resulting in a 2.5D mesh representation, which can be rendered from alternate viewpoints [14, 3]. This allows only limited view motion; artifacts will appear as soon as the viewer moves enough to reveal new areas in the scene.

In the dynamic mesh representation, we also exploit the 2.5D nature of the data in the construction of the mesh. In the holodeck environment, the sample data (excluding multi-depth samples) is a height field relative to a given view location. We define a unit sphere centered at the eye point and project the samples onto the spherical surface – reducing the problem to triangulation on the sphere. A Delaunay mesh constructed on the sphere provides the mesh topology, and the vertices are derived from the input samples. Given the initial view, the view sphere is centered at the eye point. This initial point is called the "canonical" viewpoint and may or may not coincide with the current eye location as the simulation progresses. A base mesh is created that covers the surface of the sphere. The base mesh is represented in Figure 2a. In Figure 2b, an example sample set is projected onto the view sphere, and the corresponding spherical mesh is shown in 2c.

Given a new sample, we perform point location to find the existing spherical mesh triangle that encloses its projection. If the projection of the new sample coincides with that of an existing sample, the

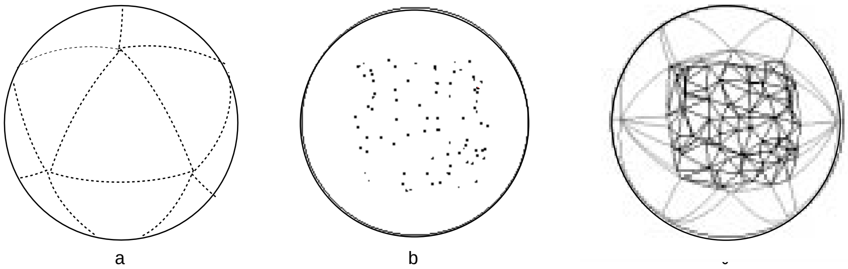


Figure 2: Spherical mesh a) Base icosahedral mesh b) Projected set of samples c) Resulting spherical mesh

sample whose direction passes closest to the current view direction is retained. Otherwise, the sample point is inserted into the triangle, creating three new triangles if the new sample falls interior to an existing triangle, or four if it falls on an edge (see Figure 3). The Delaunay condition is tested for each new triangle, and reasserted if necessary. We have adapted a planar Delaunay triangulation algorithm [5, 9] to work on the sphere. Sample points may also be deleted from the mesh. In this case, the sample is removed, as well as all of its adjacent triangles. The resulting hole is re-triangulated, and the Delaunay condition is reasserted for all new triangles.

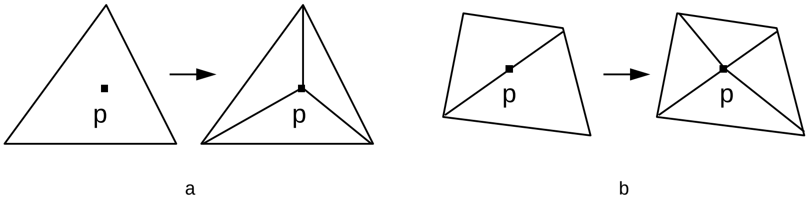


Figure 3: Point Insertion: a) Sample p falls inside existing triangle: 3 new triangles are created. b) Sample p falls on edge : 4 new triangles are created.

In the following sections we discuss the basic mesh structure, the implementation of the tasks of sample insertion and sample deletion, and the use of a point location data structure to make these tasks more efficient. We conclude the section with a discussion of robustness issues.

2.1 Mesh structure

Our implementation maintains the mesh as a list of triangles with vertex and neighbor pointers. This is the rendering representation. A second, spatial data structure is maintained to support the efficient execution of operations on the mesh. Samples are stored in a fixed size 1D array whose maximum size is specified at startup. The mesh vertices are derived from these samples. A fixed size triangle list is also created at initialization. The maximum number of triangles in the spherical mesh is bounded by the number of samples and number of triangles in the base mesh. If b is the number of triangles in the base mesh and s is the maximum sample count, then the maximum number of triangles is $(b + 2s)$. The insertion of each sample deletes an existing triangle and creates 3 new triangles, or deletes 2 existing triangles and creates 4 new ones. The triangles store pointers to their vertices, and to the 3 neighboring triangles. Samples contain a pointer to one of their adjacent triangles; the rest can be found by triangle neighbor traversal.

Existing in parallel with the 3D mesh is the notion of a spherical mesh. This mesh is the projection of the 3D mesh onto a unit sphere centered at a particular viewpoint, the canonical viewpoint, from which the mesh is constructed. This exists more as a concept then a data structure in its own right, as the only stored information for this mesh is the canonical viewpoint. The rest of the data is shared with or derived from the 3D mesh.

At initialization time a base spherical mesh is created that tiles the view sphere. The base mesh is an icosahedral subdivision of the sphere. Figure 2a shows the base mesh and Section 2.4 discusses the motivation for such a choice in detail. This mesh is created as a starting point, so sample insertion will have something to insert into. The vertices and triangles comprising the base mesh are referred to as *base vertices* and *base triangles*. These are really just place-holders that are necessary to maintain consistent mesh topology. These vertices and triangles are stored separately at the end of the mesh sample and triangle array to identify them as being extraordinary entities. The base points are calculated to lie on the sphere centered at the canonical viewpoint specified at initialization time.

2.2 Sample insertion

Once the base mesh has been initialized, samples can be incrementally inserted into the mesh. There is a one-to-one mapping of visible world space samples and their projection on the view sphere. The possibility of multi-depth values requires special checking at insertion time, but it is possible to choose one of the samples as being preferable if two projections coincide (this test is covered in Section 2.2.2). All mesh insertion and deletion operations are applied at the conceptual level to the spherical mesh. In this way, we can simplify the problem of 3D mesh construction by reducing the dimensionality to a spherical surface. Because of the height-field characteristic of the data, the mesh topology can be calculated on the sphere and then utilized for the 3D mesh merely by replacing the projected vertex coordinates with their original world space coordinates.

Given a new sample point, we first must determine which spherical mesh triangle encloses the sample's spherical projection relative to the canonical view. All points inside the space enclosed by the pyramid with apex at the canonical viewpoint and defined by the world space coordinates of a triangle will project to that spherical triangle. The process of point location is discussed in detail in Section 2.2.1. Once we find the appropriate triangle, we first test if the sample should be added to the mesh. This test is based on the density and quality of samples in the nearby area and is discussed in Section 2.2.2.

2.2.1 Point Location

We maintain a separate triangle-based quadtree data structure to accelerate point location. Associated with the view sphere is an octahedron in canonical form (origin at the canonical viewpoint, aligned with coordinate axes) that subdivides the sphere surface into eight uniform spherical triangles. A triangular quadtree is created on each octahedral face (see Figure 4a,b). Each quadtree cell contains a list of the samples that fall in that cell. The task of point location is to locate the mesh triangle that a new sample falls in. To perform point location, we first find which octant the point falls in, thereby identifying the appropriate quadtree root node. The quadtree is then traversed until the appropriate leaf is located. Once the leaf is located, a sample is extracted from the set and the new sample is tested for inclusion

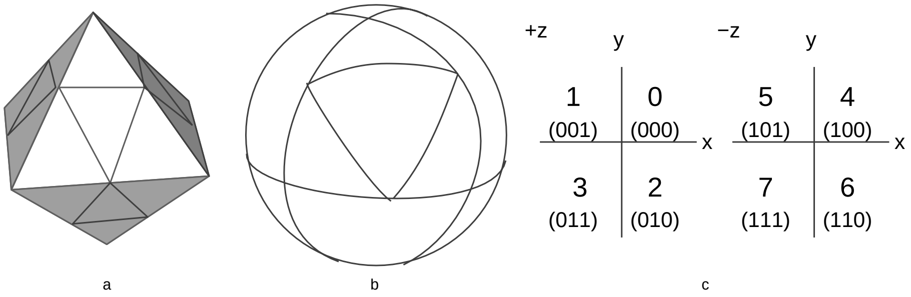


Figure 4: Point location structure a) Quadtree roots on the octahedron b) Corresponding spherical representation c) Octant labeling.

with the adjacent triangle. If the new sample does not fall in the triangle, a walk is performed along the mesh surface until the desired triangle is located. We describe these steps in more detail below.

Given a new sample, we first subtract off the canonical viewpoint such that the coordinate point is defined relative to the view sphere. This operation is unnecessary in the case of a directional point. Next the spherical octant containing the point is identified. Due to the alignment of the octree, this operation is trivial. The octants are numbered from 0-7 as indicated in Figure 4c. The octants are labeled with a 3 bit identifier, where the high order bit is set if the octant is in the negative side of the z=0 plane, the second order bit is set if the octant is on the negative side of the y=0 plane, and the lowest order bit set similarly for x. Given a 3D point p, the octant identifier I is calculated as in the following pseudo-code:

```
I =(p[2] > 0.0?0:4) | (p[1] > 0.0?0:2) | (p[0] > 0.0?0:1)
```

Once the octant is located, the point is converted into integer barycentric coordinates (a, b, c) relative to the triangle $q_0 q_1 q_2$ forming the quadtree root for that octant. The point v is projected into the plane of the quadtree root triangle. Since the octant plane equations are all of the form $Ax + By + Cz = D$, where $A, B, C, D = \pm 1$, we can first normalize the point to the $x + y + z = 1$ frame by scaling its coordinates by A, B, C of the plane equation, resulting in v'. The barycentric coordinates relative to this frame are calculated by taking the intersection of the ray v' with $x + y + z = 1$ or $p = \frac{v'}{s}$, $s = v'[0] + v'[1] + v'[2]$. The values of the coordinates will be in $[0.0, 1.0]$. In Figure 5a, the point v projects to point p in the quadtree plane. Figure 5b shows the local barycentric coordinate system for the quadtree root with vertices q_0, q_1, q_2. In Figure 5c, the barycentric coordinate p_b, has been calculated at level 0 in the quadtree.

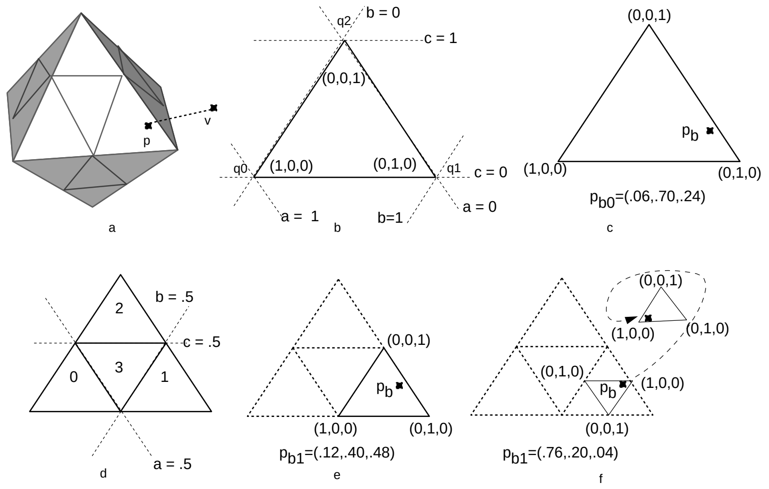


Figure 5: Quadtree traversal a) Identifying sample projection on quadtree b) Barycentric coordinate system of quadtree root c,d,e) Barycentric coordinates for point p at level 0,1,2, respectively

The barycentric coordinates are then converted into integer values for calculation efficiency. The range $[0.0, 1.0]$ is mapped to $[0, B]$, where B is set such that the range fits into an unsigned long on the target architecture with one bit to spare to prevent overflow during addition of two valid numbers in the range.

The quadtree is then traversed until the appropriate leaf is located. This operation is efficient, requiring only integer shifts and adds. Given the barycentric coordinate of the point at level i, $p_{b(i)}$, and a quadtree node, we can calculate which child $p_{b(i)}$ falls in at the next level in the quadtree. At the same time we adjust the coordinates of $p_{b(i)}$ relative to the new coordinate frame of the child, producing $p_{b(i+1)}$. Figure 5d shows the numbering of the children and the defining half-spaces. For expository purposes, the examples use the floating point values of the coordinates; in practice the integer values are used.

The following pseudo-code implements the child identification and coordinate adjustment.

```
int bary_child(b)
        if(b[0] > B/2) / * IN child 0 */
             b[0] = (b[0] << 1) - B; b[1] <<= 1; b[2] <<=1; return(0)
        if(b[1] > B/2)  /* IN child 1 */
             b[0] <<= 1; b[1] = (b[1] << 1) - B; b[2] <<= 1; return(1)
        if(b[2] > B/2)   /* IN child 2 */
             b[0] <<= 1; b[1] <<= 1; b[2] = (b[2] << 1) - B; return(2)
        /* IN child 3 */
        b[0] = B - (b[0] << 1);b[1] = B - (b[1] << 1); b[2]= B - (b[2] << 1); return(3)
```

In Figure 5e, the coordinates relative to child 1 have been calculated. Figure 5f shows the results at level 2. Note that when the center, or number 3 child is traversed, the orientation of the triangle is flipped so that the coordinates correspond to the same defining half-spaces.

This traversal continues until the leaf level is reached. Once the leaf is located, we select the first sample in the set. We could alternatively choose the sample that is closest to the new point, but in practice we find that we get good performance with choosing the first sample (average of 8 triangles visited on the walk), and it is therefore not necessary to perform the expensive test of comparing the new sample to each one in the set to find the closest. It may be possible that the leaf cell contains no samples. In this case we recursively pop back up in the quadtree traversal until a leaf is found that does contain some samples.

Each sample stores a pointer to one of its adjacent triangles. This triangle is retrieved and tested to see if it contains the new point. To test if the projection of the point onto the sphere lies within a particular spherical triangle, we can test the position of the point relative to each of the planes forming the pyramid with apex at the canonical viewpoint, and passing through the world space triangle, or equivalently against the great circles forming the spherical triangle. If the point lies inside of all three planes, it is accepted as being inside the triangle. If any one of the tests fail, the test traverses to the triangle adjacent to the current one across the plane (edge) for which the test failed.

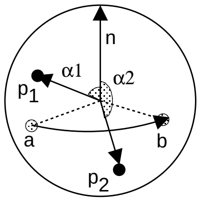


Figure 6: Orientation test: if $(p \cdot n) > 0$ (as with $p1$), the point is inside edge ab. The angle distance between the point p and edge ab is $(\frac{\pi}{2} - \cos^{-1} \alpha)$

We also want to know how close the input point lies to the spherical edge, to ensure robustness in the test (see Section 2.4). Given a new sample, we calculate the angle separation between the spherical point p and the spherical edge ab (see Figure 6). If n is the normal to the great circle defined by the edge ab, the dot product $(p \cdot n)$ gives the cosine of the angle between the normal n and the point p. If the point lies within $[0, \frac{\pi}{2}]$ above the edge, $0 \leq (p \cdot n) \leq 1$. If the point p lies within $(0, \frac{\pi}{2}]$ below the edge, $-1 \leq (p \cdot n) < 0$. The angle distance α between the point p and the edge ab is $\alpha = \frac{\pi}{2} - \cos^{-1}(p \cdot n)$. The derivative of $y = cos\theta$, $\frac{\delta y}{\delta x} = sin\theta$, has an absolute minimum at $\theta = 0$ and maximum at $\theta = \frac{\pi}{2}$. Our angle test is therefore less sensitive to small changes in $(p \cdot n)$ when p is very close to ab, i.e. $cos^{-1}(p \cdot n) \approx \frac{\pi}{2}$. This makes the test less susceptible to errors caused by round-off error in the calculation of $\cos\theta$.

We are guaranteed to find the appropriate triangle. Firstly, there will be a triangle, since the surface of the view sphere is completely tiled with spherical triangles. Secondly, because of the spherical Delaunay topology, we are guaranteed to converge upon the correct triangle, without re-visiting any mesh triangles along the way. Section 2.4 discusses how the geometric predicates are implemented in this algorithm to ensure robustness.

Initially, we had implemented the point location based on storing triangles in the quadtree structure. With this implementation, the mesh walk was not required, and the sample was instead tested against each triangle in the cell. Each cell was guaranteed to have at least one triangle. In practice, we found that this approach suffered in two aspects. Firstly, it was costly to insert the triangles into the data structure: requiring a test to determine all of the quadtree cells that the triangle intersected. Secondly, we found it difficult to ensure a robust implementation. As a result we have implemented the sample quadtree structure as discussed above.

2.2.2 Testing sample quality

Not all samples are added to the mesh, and samples may be deleted from the mesh as the result of tests performed at insertion time based upon the density and quality of samples in the mesh. In an attempt to minimize artifacts due to visibility errors from approximate sampling, samples are also rejected once the sampling gets relatively dense if their addition would cause a "puncture" (e.g. a smooth surface interrupted by a single sample with large depth discontinuity) in the mesh.

If the spherical projection of two samples is closer than some defined resolution epsilon, ϵ_v, only one of the samples will ultimately be part of the mesh. We choose an epsilon corresponding to approximately 0.028° of visual angle from the canonical view. This is done both to avoid instabilities in the mesh construction (see Section 2.4) but also because it corresponds to reasonable rendering resolution: with a $60°(\frac{\pi}{3})$ view frustum angle and a screen resolution of 1024x1280 pixels, there are approximately 19.35 pixels/degree (measured from the center of the frustum where the pixel/degree ratio is densest). A mesh with 0.028° or .0005 radians separation between samples gives approximately 1.8 samples per pixel. A mesh constructed with detail beyond this would not add significant visual information to the final image, but would incur more rendering overhead. We therefore choose $\epsilon_v = 5 \times 10^{-4}$.

To determine which of the two samples to keep, we first examine the types of the samples. If the new sample is a directional point, it is discarded: it is assumed that foreground points are always preferable to background points, so if the new sample is a directional sample and the existing one is a world space point, the new sample is discarded; if both the existing and the new sample are directional points, they must correspond to the same world space direction, so it is sufficient to keep the existing one in place. If the existing point is a directional point and the new sample is a world space point, the directional sample is replaced with the foreground sample. If both samples are world space points, we compare the sample directions. The sample that was calculated from the direction closest to the direction relative to the current view is chosen. This is based on the assumption that a sample that was calculated from a view direction closer to the current view is less likely to produce a visibility error. In the limit, all of the samples will be sent from the current view and there will be no visibility errors due to approximate sampling. Let s_n be the direction that the new sample was calculated from and s_e the sampled direction of the existing sample, and d_n and d_e the corresponding directions to the samples from the current viewpoint: if $(d_e \cdot s_e) >= (d_n \cdot s_n)$, the existing sample is considered better and the new sample is discarded. If the opposite is true, the old sample is deleted from the mesh and the new sample is then added.

For efficiency, the test occurs after point location has been performed to determine which mesh triangle the new sample falls in. The sample is only tested against the three vertices of the triangle. The actual closest sample is not necessarily one of these three. This approximate test is sufficient however, because since we maintain a Delaunay condition on our mesh, the triangles are in general well-formed. The only way that a vertex outside the triangle could be closer is if there was a long, thin triangle adjacent, and as this would be swapped out in the Delaunay verification, it is not likely to happen. We perform a conservative test to eliminate possibly problematic samples. If the new sample falls within the circumcircle (see Section 2.2.3) of the triangle, there cannot be any vertex closer to it than one of the triangle vertices (by definition, no other samples are contained in the circumcircle). If the circumcircle test fails, the new sample is rejected. In practice, this situation has never occurred.

Due to the re-use of samples, the re-projection of a cached sample to a new view can produce incorrect visibility results. A sample that was visible in a previous view, may no longer be visible in the current view. The correct sample for that image location will eventually be traced and replace the current sample in the mesh, but right after a view motion that sample may not yet be available. In this case, the old sample is presented by the driver to the display generation. If no additional checking is done, this point will get added to the mesh, and can create the artifact of false "punctures" in the mesh, where a continuous foreground surface is interrupted by a sharp depth discontinuity at a single point. With small view motions this artifact is most likely to occur near object silhouettes. We apply a simple heuristic to minimize these artifacts: if the addition of a sample point will cause a sharp depth discontinuity between the new point and its neighbors, and the new point is behind the existing points relative to the current view, the sample is not added to the mesh. Note that such a puncture point could be valid: for example a fine screen surface in front of a background. By first testing the direction of the sample to see if it coincides with the current view, we can differentiate between these cases.

When we are inserting a sample, we must also check if there is room in the existing sample array. The array is fixed in size at initialization time. If we request a new sample and determine that the sample array is full, we must first delete an existing sample before the new one can be added. We choose a simple, approximate Least Recently Used (LRU) sample replacement scheme. Our replacement strategy is based on a clock or use bit. When a sample is first allocated its use bit is set. At rendering time all triangles in the view frustum are marked as active. At this time the samples corresponding to the adjacent vertices of active triangles have their use bits set. When the sample array is full a circular traversal is made from the location of the last allocated sample. The traversal visits the samples in the array in order, testing the use bits. If the bit is set, it is cleared, and the traversal moves on to the next sample. If the bit is clear, then that sample is chosen for removal from the sample array and the corresponding vertex is removed from the mesh. With this algorithm, samples that have been rendered recently will be retained, and those that have not taken part in the most recent renderings will be subject to removal. In general, the size of the sample array is chosen to the desired screen resolution, so it is more likely that samples will be replaced by the proximity tests as described at the beginning of this section when the view is stationary. If the user is moving about, the mesh will be periodically rebuilt with only the visible samples included. In an interaction scenario where the user spins about a fixed viewpoint and pauses at various locations, enabling the progressive refinement of multiple sections of the spherical mesh, it is possible to run out of samples in the array.

If the sample is accepted, the triangle is subdivided and the neighbors are updated appropriately. If the triangle falls on an existing sample, it will be handled by the quality testing described above. In order to maintain a "well-behaved" mesh that minimizes rendering artifacts and numerical errors, we maintain a Delaunay condition on the spherical mesh triangles. This is discussed in the following section.

2.2.3 Delaunay condition

After any changes to the mesh topology, a test is performed to verify if the Delaunay condition still holds. A triangle satisfies the Delaunay condition if the circumcircle of the triangle contains no other sample points [11]. In two dimensions, this condition can be verified with a point-in-circle test (see Figure 7a).

In our spherical mesh environment, given a triangle t_i with vertices v_0, v_1, v_2 and its adjacent neighbor t_j with vertices v_2, v_1, v_3, the test is whether vertex v_3 lies within the circumcone formed by the canonical viewpoint and v_0,v_1,v_2 In the spherical environment, we utilize a point-in-cone test (see Figure 7b). The triangle vertices v_0, v_1, v_2 and opposite vertex in question v_3 are projected onto the view sphere giving a, b, c, p. A point p lies in the cone defined by the canonical viewpoint and a, b, c if the angle between the cone center ray z and p is less than the angle between z and one (any) of the vertices a, b, c (see Figure

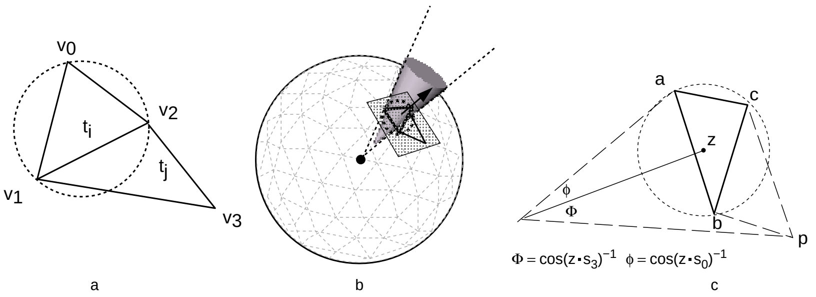


Figure 7: Delaunay test: a) Point in circumcircle test in the plane b) Point in circumcone test in spherical environment c) Vertices are projected onto the sphere. The angle between the cone center z and the point in question p is compared against the radius angle of the cone (e.g. the angle between z and a)

7c). The ray z defining the cone center is equivalent to the normal to the plane defined by a, b, c. The dot product of any normalized point p and z is proportional to the cosine of the angle between p and the cone center ray. If the cosine of the angle between z and p is greater than that between a and z, the corresponding angle is smaller, and p lies within the cone. The test is implemented as follows:

$$\begin{aligned} z &= (b-a) \times (c-a) \\ in &= (p \cdot z) > (a \cdot z) \end{aligned}$$

After the addition of each new sample, the Delaunay test is performed against the sample point and all adjacent triangles. If the test fails, an edge swap is performed in the quadrilateral formed by the two adjacent triangles (see Figure 8). The existing triangles are deleted and two new triangles formed with the neighbor relationships derived from the previous triangles. These new triangles are then added to the list of triangles that must be tested against their neighbors to determine if the Delaunay condition is maintained. The number of possible swaps performed at each insertion is $O(n)$ for a mesh with n samples. In practice, however, we have found the number of edge swaps performed per sample insertion is 2.8 on average over a number of representative runs of the mesh generation.

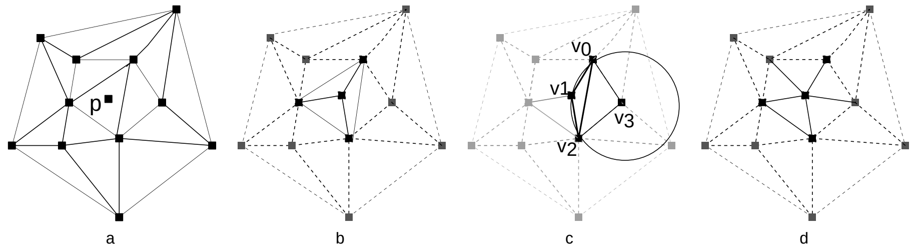


Figure 8: Point insertion: a) Point Location to find triangle containing p b) 3 new triangles created c) New triangle $v_0 v_1 v_2$ with point v_3 does not pass circumcone test d) After swapping 2 edges to restore Delaunay condition

The Delaunay test is performed on the spherical mesh using the projection of the 3D sample coordinates. This projection is done so the mesh topology can be constructed more efficiently than if the problem was addressed in the full 3D space. In addition, by producing a mesh that is Delaunay when projected from the canonical view, we hope to generate a set of 3D triangles that will be less susceptible to shading artifacts, both from the canonical viewpoint and from neighboring locations. It is important to remember that the world-space coordinates are actually utilized as the mesh vertices. This allows the re-use of the mesh from viewpoints slightly off the canonical view simply by having the rendering hardware do the appropriate transformations upon the 3D triangles.

2.3 Sample Deletion

It is possible for samples to be deleted from as well as inserted into the mesh. Samples are deleted by marking the sample as free, removing all adjacent triangles, re-triangulating the hole, and re-asserting the Delaunay condition (see Figure 9 for an example in the plane). We discuss relevant details below.

After the adjacent triangles have been deleted, the chain of edges forming the boundary of the hole is constructed by traversing the neighbors counterclockwise until all adjacent triangles have been visited. The resulting spherical polygon is then re-triangulated. This is not a general triangulation routine, as we are able to exploit the fact that the spherical polygon formed by the removal of the vertex p from the mesh is a star relative to p. The re-triangulation proceeds by traversing the boundary list and cutting off ears formed by a consecutive triple of vertices s_0, s_1, s_2. A cut is is accepted iff the following conditions are true:

1. The edge $s_2 s_0$ does not intersect any boundary edges of the remaining spherical polygon.
2. The angle formed by $s_0 s_1 s_2$ is less than π (i.e. convex).
3. The resulting new triangle passes the Delaunay test (i.e. none of the remaining vertices of the spherical polygon lie in the cone defined by $s_0 s_1 s_2$ and the canonical view point).

The Delaunay condition (3) guarantees that no edge intersections occur, but the implementation of the Delaunay test is more expensive than the intersection test (1), and the convexity test (2) can use the results of this test, so it is implemented as constraint 1. The resulting triangulation, due to the 3rd constraint, satisfies the Delaunay condition on the sphere.

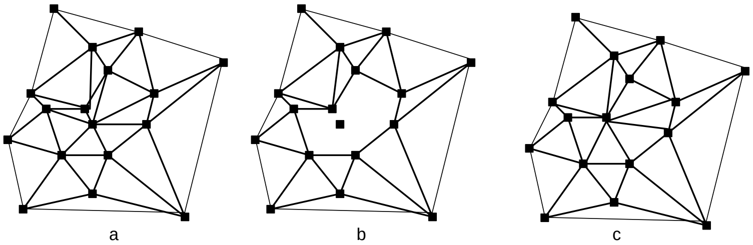


Figure 9: Sample Deletion in the plane: a) Initial mesh b) Adjacent triangles are deleted, creating a hole c) Re-triangulation of hole with Delaunay condition asserted

Given the projection of a sequential triple of vertices, s_0, s_1, s_2, onto the view sphere, we first perform the test for edge intersection. To determine if any intersecting edges would result, a test is performed to find which side of the new edge the original (deleted) center sample lies on. Because the initial configuration was a star formation around the center vertex, an edge formed between two vertices that forms a polygon disjoint from the one containing the center point cannot intersect any other edges.

The implementation of the test calculates the normal to the plane of the great circle defining the spherical edge $s_2 s_0$: $n = s_0 \times s_2$. If the center point p lies outside of this edge, the edge cannot intersect any of the remaining edges in the spherical polygon. This can be determined by taking the dot product with the projected point p with the resulting cross product (see Figure 10a). The sign of the dot product determines which case is true: The test is calculated as $t = (n \cdot p)$ where p is the deleted sample. If t is positive, the center point lies on the inside of the proposed triangle, and the triple is rejected (see Figure 10b). If the test passes, the test for convexity is next performed. If the remaining spherical vertex s_1 lies in the negative half-space defined by n, the angle is convex.

Finally the Delaunay condition is tested. Each remaining point on the spherical polygon boundary is tested for inclusion in the cone defined by s_0, s_1, s_2 and the canonical viewpoint (see Section 2.2.3). If no point lies within the cone, then the cut is accepted. Points outside of the boundary need not be tested.

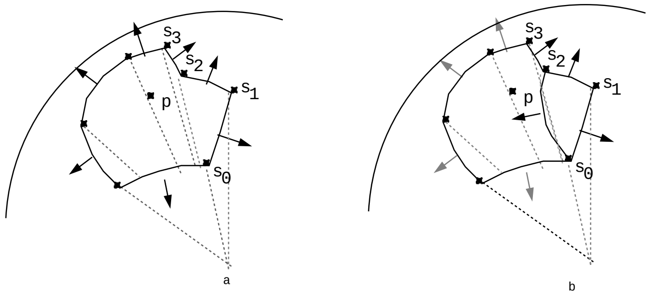


Figure 10: Re-triangulation after deletion a) Bounding spherical polygon after deletion of sample projected as p. Angle $s_0 s_1 s_2$ is convex, and $s_1 s_2 s_3$ concave relative to the spherical polygon interior. Great circle normals are shown. b) Testing edge $s_2 s_0$ point p lies to the outside of the triangle $s_0 s_1 s_2$ and the Delaunay condition is upheld, so the split is accepted.

Once all convex angles have been trimmed off as triangles, a single triangle or a spherical quadrilateral will remain (it can have more than four edges, but those additional edges are coincident with the same great circle). If the remaining polygon is a triangle, the triangulation is complete. If the polygon is a quadrilateral, a trivial triangulation is performed.

2.4 Robustness

It is very important for our application that the incremental mesh construction be fast. It also must be robust, however. We accomplish this through a combination of performing inexact tests wherever possible, and imposing constraints on the mesh based on the input and computing environment to ensure that round-off error is not a problem. We also take advantage of the fact that we do not need to add every sample. Those for which we cannot make guarantees about the resulting mesh are discarded. As this happens very infrequently, and only when the sampling is dense, no adverse effects are observed.

The algorithms manipulating the mesh depend on a clean 2D topology. In our context this means that the mesh elements only intersect at mesh vertices, and each edge is adjacent to exactly two triangles. For efficiency purposes, there are no run-time tests to detect corrupted topology; we instead ensure clean topology by construction. There are two major geometric operations involved in mesh construction : point location, and testing for the Delaunay condition. In this section we discuss an implementation of each geometric operation that produces clean mesh topology at each step.

In the following, edge lengths are represented by the angle measured in radians subtended by the two spherical end points. Angles interior to the triangle are measured as the angle between the two planes defined by the great circles associated with the spherical edges.

2.4.1 Point Location

The primary and most expensive task in mesh construction is point location: given a new spherical point, we must determine which existing triangle the point falls in. We first utilize the spherical quadtree to put us in the neighborhood of the triangle. We then initiate a walk to find the correct triangle. We do not require that the spherical quadtree data structure be completely robust: due to round-off error, a sample near a cell boundary could be inserted into a cell adjacent to its correct location. This sort of inconsistency is tolerated for efficiency at insertion time, and because it does not lead to catastrophic topology errors. If the walk starts in the incorrect, nearby cell, this error will at most degrade the efficiency slightly; even in the case where a point has been inserted exactly, the search is often started only in the neighborhood of the correct triangle. In order that this inconsistency in the point location not cause problems when searching for specific points to remove from the data structure during sample deletion, each sample maintains a pointer to its quadtree cell, so it can be removed directly without a traversal.

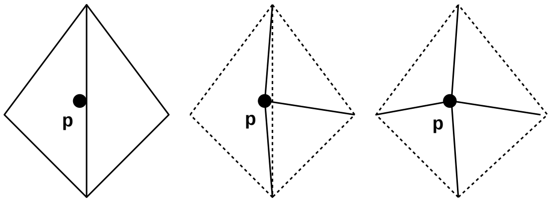


Figure 11: Round-off error can lead to the invalid topology in the middle figure if the orientation test reports that the point is on the right side of the edge. An edge split would produce the valid construction shown in the right figure. In both examples, the new edges are shown in solid line, and the edges retained from the previous configuration are shown dotted.

During the walk, we compare the edges of the current triangle to the point, to determine if the point lies interior to the triangle, and if not, which edge to cross to continue the walk. Problems can occur when a point is close to one or more edges. Figure 11 illustrates one potential problem. In the left figure, the round-off error is such that the test determines that the point is on the right side of the edge, incorrectly splitting the triangle, resulting in the topology shown in the middle figure. To prevent this problem, if a point is within some epsilon, ϵ_e, of an edge, it is treated as if it is on the edge and a four-way split is performed on the two adjacent triangles. The result is shown in the right figure.

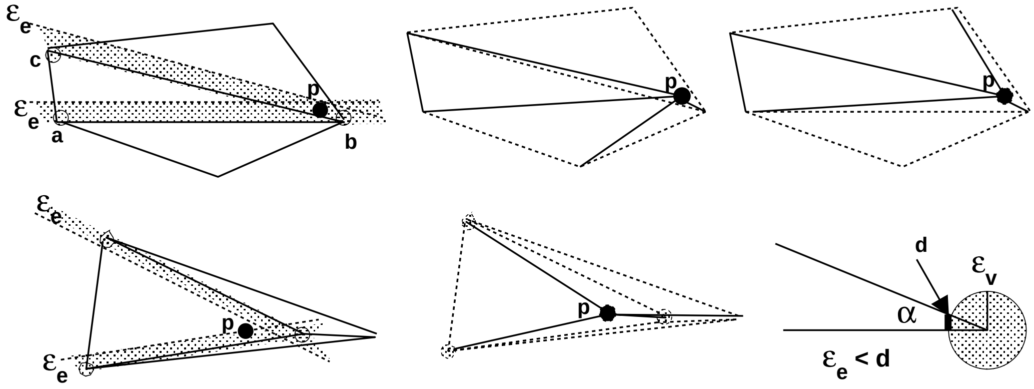


Figure 12: If ϵ_e is too large and adjacent edges overlap, the wrong edge can be split as shown in the top figure. Invalid topology can also occur with too large values, as shown in the bottom figure, even if the ϵ regions do not overlap. We choose ϵ_e to be less than edge separation for the smallest angle α a distance ϵ_v from the vertex as shown in the lower right figure

The next example in Figure 12 illustrates topological errors that can occur if ϵ_e is chosen too large. The point p is within ϵ_e of two edges (ba and cb). If the edge ba is tested first, then the topological error shown in the middle figure will occur if the two triangles are split into four. The desired answer is shown on the right. Finding the closest edge within ϵ_e would avoid this error, but we would prefer not to have to test for this. Instead we choose ϵ_e small enough such that a point cannot be within ϵ_e of more than one edge. Lastly, the bottom of Figure 12 shows another potential problem if ϵ_e is chosen too large. The split is shown in the adjacent figure. The split causes invalid topology because ϵ_e is larger than the edge separation for the smallest angle, a distance ϵ_v away from the vertex (ϵ_v is the minimum angle distance between vertices and therefore the minimum edge angle). We choose $\epsilon_e < \frac{d}{2}$ to avoid this kind of error (see lower right Figure 12).

We first utilize constraints on our environment to determine what ϵ_e to choose such that the above problems cannot occur. This analysis assumes that calculations are done with exact arithmetic. We then do forward error analysis on the angle calculation code to make sure that round-off errors in the machine arithmetic calculations are not large enough to introduce inconsistencies given our chosen constraints.

2.4.2 Selecting Epsilon

We choose $\epsilon_e < \frac{d}{2}$, where d is the angle separation between two spherical edges, a distance ϵ_v away from the vertex. We can calculate a lower bound on this distance d, given the initial mesh and the constraints imposed by the triangulation algorithm.

The initial base triangulation is an icosahedral subdivision of the sphere. An icosahedron can be constructed from three mutually perpendicular golden rectangles, i.e. rectangles whose length and width are related by the golden ratio $\varphi \approx 1.61803$. [1] This is illustrated in Figure 13. The initial spherical edge length is $cos^{-1}(\frac{p_0 \cdot p_1}{||p_0||||p_1||}) \approx 1.10715$.

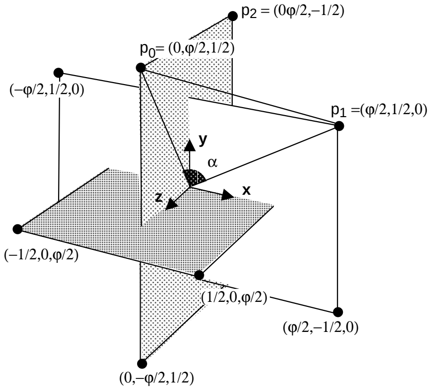


Figure 13: Icosahedron construction and edge angle

The initial triangulation defines the largest circumcone that can exist in the mesh. Let a, b, c be the spherical points corresponding to p_0, p_1, p_2. We can calculate the center ray of this cone, z, as the normal to the spherical triangle: $z = \frac{(a-c)\times(b-c)}{||(a-c)\times(b-c)||}$. The cone radius angle $\theta = cos^{-1}(z \cdot a) \approx 0.652358$. This is illustrated on the left in Figure 14.

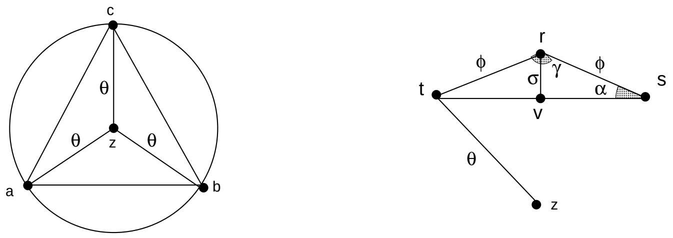


Figure 14: Calculating minimum (α) and maximum (γ) possible angle bounds for the spherical mesh

In the mesh construction, no two points are allowed to be closer than ϵ_v, therefore the minimum edge length possible is $\phi = \epsilon_v$. Given these bounds, we can calculate the minimum possible angle. The minimum angle occurs for a triangle with circumcone of radius θ, and two edges of length ϕ (see Figure 14). Given $\phi = \epsilon_v = 5 \times 10^{-4}$ (see Section 2.2.2), we calculate the maximum angle γ ($\alpha = \frac{\pi-\gamma}{2}$).

For this calculation, we choose the cone axis to be $z = (0.0, 0.0, 1.0)$, and choose $r_x = 0.0$. We can solve for r, s based on the following constraints:

$$\begin{array}{llll} r \cdot z = \cos\theta & s \cdot z = \cos\theta & ||r|| = ||s|| = 1 & r \cdot s = \cos\phi \\ r = (0.0, \sin\theta, \cos\theta) & s_z = \cos\theta & s_y = \frac{(\cos\phi - \cos^2\theta)}{\sin\theta} & s_x = \sqrt{1.0 - s_y^2 - s_z^2} \end{array}$$

Given r, s we can calculate the maximum angle γ and minimum angle α:

[1] The golden ratio is the value of x such that $\frac{x}{1} = \frac{1}{x-1}$. Choosing the positive root, $\varphi = x = \frac{1+\sqrt{(5)}}{2} \approx 1.61803$

$$\gamma = 2\left(\cos^{-1}\left(\frac{(r \times z)\cdot(r \times s)}{||(r \times z)||||(r \times s)||}\right)\right) \approx 3.140938$$

$$\alpha = \frac{\gamma - \pi}{2} \approx 0.000327$$

Given the minimum and maximum angles, we can calculate what the minimum edge separation d is, a distance ϕ from the vertex forming the minimum angle. We will assign $\epsilon_e = \frac{d}{2}$. Again, looking at Figure 14, we wish to calculate σ. We use the following constraints to first calculate t, then the halfway vector v:

$r = (0.0, 0.0, 1.0) \quad r \cdot s = \cos\phi \quad r \cdot t = \cos\phi \quad ||s|| = ||t|| = 1 \quad s = (0, \sin\phi, \cos\phi)$

$t_x^2 + t_y^2 = 1 - \cos^2\phi = \sin^2\phi \quad \sin\gamma = \left|\left|\frac{(s-r)\times(t-r)}{||(s-r)||||(t-r)||}\right|\right|$

From the resulting value of t (see Appendix B), we calculate the halfway vector v between s, t. The result of $v \cdot r$ is the cosine of the minimum edge separation. Since $r = (0, 0, 1)$, $\cos\sigma = r \cdot v = v_z = 9.9999 \times 10^{-1}$ and the minimum angle $\sigma = 1.056 \times 10^{-7}$. We choose the edge epsilon to be less than half the value of σ: $\epsilon_e = 5 \times 10^{-8}$. The following section evaluates the possible effects of round-off error from utilizing inexact arithmetic to perform the angle calculation.

Mesh topology is also altered in order to maintain the Delaunay condition. We must ensure that the mesh remains topologically sound through these operations as well. We verify in the Section 2.4.4 that the algorithm described in this section also satisfies the constraints of the Delaunay predicate.

2.4.3 Forward Error Analysis

In this section, we analyze the possible round-off error that could be incurred in the angle calculation. This value must not be significant enough to cause problems based on the given choices of ϵ_e and ϵ_v. We apply forward error analysis [12] to derive error bounds for the angle calculation. In the following, t_i refers to the true value of sub-computation i, and x_i is the computed value of that calculation including cumulative round-off error. In the IEEE standard for binary floating point arithmetic (ANSI/IEEE 754-1985) [4], a floating point calculation with exact round-off can be in error as much as $\frac{1}{2}$ units in the last place (ulp). With base $\beta = 2$, and p-bit significand, the error in rounding t to $x = \underbrace{d.dd\ldots d}_{p} \times 2^e$ is $\leq \underbrace{0.00\ldots0}_{p} 1 \times 2^e = \epsilon \times 2^e$. For double precision floating point the machine epsilon, $\epsilon = 2^{-53}$. Therefore:

$$|t - x| \leq \frac{1}{2} ulp \leq \epsilon \times 2^e \leq \epsilon|x|$$

In the error analysis, we use the following upper bound on the round-off error:

$$t = x \pm \epsilon|x|$$

In the following, the symbols $\oplus, \ominus, \otimes, \oslash, SQRT$ represent double floating point addition, subtraction, multiplication,division, and square root with exact rounding. For the above operations, the IEEE standard requires the answer to be exactly rounded: therefore, with $\odot$ representing the stated binary operations, we use the following expression of the error analysis:

$$t = a \odot b \pm \epsilon|a \odot b|$$

For square root, $t = SQRT(a) \pm \epsilon SQRT(a)$.

For addition/subtraction of two terms $(x \pm \alpha)$ and $(y \pm \beta)$, where x, y are the calculated subcomponents and α, β the accumulated error, the true value t is:

$$t = x + y \pm (\alpha + \beta) = x \oplus y \pm (\epsilon|x \oplus y| + \alpha + \beta)$$

Similarly for multiplication of two terms $(x \pm \alpha)(y \pm \beta)$:

$$t = xy \pm (\alpha\beta + x\beta + y\alpha) = x \otimes y \pm (\epsilon|x \otimes y| + \alpha\beta + x\beta + y\alpha)$$

For square root (see Appendix C.1):

$$t = SQRT(x) \pm (\epsilon|SQRT(x)| + \frac{1}{2}|\alpha x^{-\frac{1}{2}}| + + \frac{1}{2}|\alpha^2 x^{-\frac{3}{2}}|)$$

For division of terms $(x \pm \alpha)$ and $(y \pm \beta)$ (see Appendix C.2):

$$t = x \oslash y \pm (\epsilon|x \oslash y|) + \left|\frac{|y\alpha| + |x\beta|}{y(|y| - |\beta|)}\right|$$

For spherical edge ab and normalized point p, we want to calculate the angle between ab and p. We first calculate the normal to the great circle, or plane through a, b and the origin, $n = \frac{a \times b}{||a \times b||}$. For θ, the angle between the normal and the ray to the point, $(n \cdot p) = cos\theta$. The calculations can be represented as follows:

$$\begin{array}{lcllcllcl}
cp_x & = & a_y b_z - a_z b_y & cp_y & = & a_z b_x - a_x b_z & cp_z & = & a_x b_y - a_y b_x \\
||cp|| & = & (cp_x^2 + cp_y^2 + cp_z^2)^{\frac{1}{2}} & & & & & & \\
n_x & = & \frac{cp_x}{||cp||} & n_y & = & \frac{cp_y}{||cp||} & n_z & = & \frac{cp_z}{||cp||} \\
\cos\theta & = & n_x p_x + n_y p_y + n_z p_z & & & & & &
\end{array}$$

Our angle computation can be expressed as follows:

$$\begin{array}{lcllcl|lcllcl}
t_1 & = & a_y b_z & x_1 & = & a_y \otimes b_z & t_{12} & = & t_9 t_9 & x_{12} & = & x_9 \otimes x_9 \\
t_2 & = & a_z b_y & x_2 & = & a_z \otimes b_y & t_{13} & = & t_{10} + t_{11} & x_{13} & = & x_{10} \oplus x_{11} \\
t_3 & = & t_1 - t_2 & x_3 & = & x_1 \ominus x_2 & t_{14} & = & t_{13} + t_{12} & x_{14} & = & x_{13} \oplus x_{12} \\
t_4 & = & a_z b_x & x_4 & = & a_z \otimes b_x & t_{15} & = & (t_{14})^{\frac{1}{2}} & x_{15} & = & SQRT(x_{14}) \\
t_5 & = & a_x b_z & x_5 & = & a_x \otimes b_z & t_{16} & = & \frac{t_3}{t_{15}} & x_{16} & = & x_3 \oslash x_{15} \\
t_6 & = & t_4 - t_5 & x_6 & = & x_4 \ominus x_5 & t_{17} & = & \frac{t_6}{t_{15}} & x_{17} & = & x_6 \oslash x_{15} \\
t_7 & = & a_x b_y & x_7 & = & a_x \otimes b_y & t_{18} & = & \frac{t_9}{t_{15}} & x_{18} & = & x_9 \oslash x_{15} \\
t_8 & = & a_y b_x & x_8 & = & a_y \otimes b_x & t_{19} & = & t_{16} p_x & x_{19} & = & x_{16} \otimes p_x \\
t_9 & = & t_7 - t_8 & x_9 & = & x_7 \ominus x_8 & t_{20} & = & t_{17} p_y & x_{20} & = & x_{17} \otimes p_y \\
t_{10} & = & t_3 t_3 & x_{10} & = & x_3 \otimes x_3 & t_{21} & = & t_{18} p_z & x_{21} & = & x_{18} \otimes p_z \\
t_{11} & = & t_6 t_6 & x_{11} & = & x_6 \otimes x_6 & t_{22} & = & t_{19} + t_{20} & x_{22} & = & x_{19} \oplus x_{20} \\
t_A & = & t_{22} + t_{21} & A & = & x_{22} \oplus x_{21} & & & & & &
\end{array}$$

We now derive a bounds for the worse case error. Since the input points lie on the unit sphere, $|a_x|, |a_y|, |a_z|, |b_x|, |b_y|, |b_z|, |c_x|, |c_y|, |c_z| <= 1$. The magnitude of the cross-product is equal to $\sin\theta$, where θ is the edge length. We can bound this value ($|x_{15}|$, and its square root $|x_{14}|$) given the initial, and minimum edge lengths: $2.5e - 7 \leq |x_{14}| \leq .930856$ and $5e - 4 \leq |x_{15}| \leq .96480899$

Given these upper bounds, we calculate the error by implementing the above forward error calculations using a multi-precision arithmetic package (MPFUN [1]) and giving the bounded values as input. Given this approximation the upper bound on the round-off error, $e_r <= 1.1 \times 10^{-11}$.

A change in $cos\theta$, $\delta(\cos\theta) = 1.1 \times 10^{-11}$ when $\theta \approx \frac{\pi}{2}$, corresponds approximately to a linear change in θ, $\delta\theta \approx 1.1 \times 10^{-11}$. Since this value is well within our choice of ϵ_e, the test will not be troubled by round-off error. The forward error analysis was a conservative estimate. We calculate the round-off error

Table 1: Round-off Error: Experimental Results

Test	Minimum	Maximum	Average	Std. deviation $\sigma = \sqrt{\frac{n\sum_{i=1}^{n} x^2 - (\sum_{i=1}^{n} x)^2}{n^2}}$
Orientation	0.0	5.15×10^{-19}	1.86×10^{-23}	1.8×10^{-21}
Point-in-Cone	0.0	1.75×10^{-19}	1.5×10^{-24}	5.09×10^{-22}

over various runs of the program, by comparing the computed value to that calculated using MPFUN. Approximately 500,000 angle calculations were performed each run. A small portion of the screen was selected for progressive refinement during the course of the run in order that the mesh reached maximum density during the test. The results from this test are shown in Table 1 in the row labeled "Orientation".

2.4.4 Delaunay predicate

For the Delaunay point-in-cone test, for triangle a, b, c and point p, we calculate the center axis z of the cone circumscribing triangle a, b, c (see Figure 7): $z = (b - a) \times (c - a)$ To test if p lies within the cone, we compare the cosines of the angle between p and a, and the cone axis, z. If p lies within the cone, $(p \cdot z) > (a \cdot v)$ Figure 15 shows a potential problem that could occur due to round-off error. If cumulative error in the calculation causes the test to return true in the situation illustrated in Figure 15a, the invalid topology shown in Figure 15b will occur after the edge flip. When the triangle interior angles are very large(small), the magnitude of z approaches zero. We have shown that the maximum angle $\gamma \approx 3.140938$, with $\sin\gamma \approx 6.5 \times 10^{-4}$.Since the minimum edge separation between a, p is $\epsilon_v = 5 \times 10^{-4}$, the round-off error in the dot product calculation would need to be greater than $\frac{\epsilon_v}{2} = 2.5 \times 10^{-4}$ to cause problems. Our worst case analysis (see previous section) yields a round-off error of $e_d < 2.5 \times 10^{-15}$. Run-time calculations produce the round-off errors shown in the second row of Table 1.

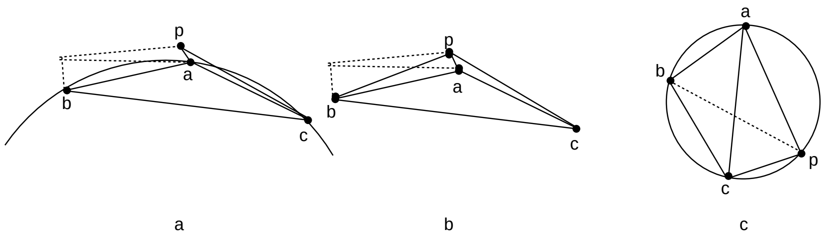


Figure 15: Delaunay test errors

We must also address the degenerate case where all four points a, b, c, p lie on a circle (see Figure 15c). Depending on the algorithm, in this instance the Delaunay test could go into an infinite loop, repeatedly flipping the diagonal of the quadrilateral formed by a, b, c, p. We address this case algorithmically. All swaps are initiated by the addition of a new vertex a. The test is performed always using the triangle a, b, c, for each triangle adjacent to a, to form the circumcone, and p is the remaining point in the opposite triangle pcb. In the degenerate case, the flip will only occur once, each time a, b, c or p is manipulated.

3 Rendering

One of the advantages of the spherical mesh display representation is that it is a valid 3D triangle mesh that can be transformed with new views and rendered directly by the hardware using OpenGL. After each view change, the current mesh is first rendered. As more sample points are received for the same view, the mesh and the display are incrementally updated. The mesh constructed from a canonical viewpoint is valid for all viewing configurations with the same eye position. For small view motions relative to the distance to the viewed sample points, the mesh is re-used. With this approximation, errors will appear in the image where portions of the environment are occluded relative to the canonical viewpoint, but visible to the current eye point (or vice versa). We minimize these errors by constructing a new mesh once the viewer moves a substantial distance away from the canonical viewpoint.

In Section 3.1 we first describe how the mesh is rendered for a single view, and how view frustum culling is utilized to optimize the rendering performance. We then describe how the rendering algorithm handles the following interaction scenarios:

- The observer is stationary, and new samples are being generated (Incremental Rendering, Section 3.2).
- The viewer is moving (Rendering During View Motion, Section 3.4).
- The viewer has moved a substantial distance away from the canonical viewpoint (Mesh Rebuilding, Section 3.5).

3.1 Basic Rendering Algorithm

Because the representation is a triangle mesh with world space coordinates and RGB color values for all vertices, for the most part rendering is very straightforward using OpenGL. Points at infinity, if they are present, must be handled separately. A point at infinity comes in from the driver as a direction vector. This is represented as a directional point in the mesh by adding the canonical viewpoint to the direction vector and creating a point on the view sphere. This mesh point can then be treated identically to the world space samples for mesh manipulation. For rendering, the directional points are processed first. The depth buffer is disabled, and all triangles made up entirely of directional points are translated to the origin, scaled if necessary to ensure they are not clipped to the frustum, and then translated relative to the current view point. If a triangle is "mixed", or contains one or two directional points, it is processed with the background triangles. The world space points are projected onto the current view sphere, and then scaled if necessary to fit in the frustum.

Once all background triangles are rendered, depth testing is enabled and the remaining triangles are rendered in the standard way. The only other consideration is the handling of *base* points. These resemble a directional point in that they are stored as a unit vector relative to the canonical viewpoint. They do not have a valid RGB value. If a base point is part of a background triangle, its coordinate is rendered as a background point would be. For a foreground triangle, the direction vector is scaled out by the average distance to the foreground points from the current view. For both foreground and background triangles the color used is the average of the non-base vertices of the triangle.

3.2 Incremental Rendering

When the observer is stationary, and new samples are being generated, incremental updates are made to the mesh and to the display. Each batch of new triangles will overwrite the image of the parent triangles. We utilize a painter's approach in the incremental display algorithm: the depth buffer is disabled, the new triangles are depth-sorted, then rendered back-to-front. As in the general case, background and mixed triangles are rendered first, then foreground triangles. When the canonical viewpoint corresponds to the current eye point, depth testing is unnecessary (and not performed), but as we will discuss below, these two points do not always coincide.

3.3 View Frustum Culling

To optimize rendering and mesh reconstruction, we utilize the spherical quadtree data structure to perform view frustum culling. Each of the 6 faces of the view frustum is split into two triangles (see Figure 16a). All nodes of the spherical quadtree that the projection of a frustum triangle overlaps are marked. All samples stored in the marked nodes, and all triangles adjacent to those samples, are considered visible.

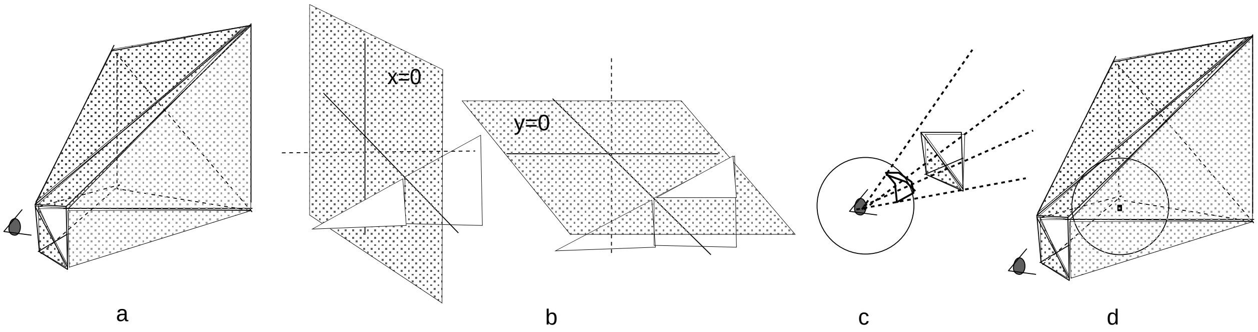


Figure 16: View Frustum culling a) Frustum divided into triangular facets b) Clipping frustum triangles against the octants c) When viewpoint coincides with canonical viewpoint: Only the front frustum face needs to be inserted d) When the viewpoint is inside the view frustum, the projection encompasses the whole frustum and does not help.

The frustum culling is implemented utilizing the spherical quadtree. Each quadtree root that a frustum triangle overlaps is traversed. At each level, the child branches are visited for those children that the triangle cannot be trivially rejected as not intersecting that node. At the leaf level, a triangle-triangle test is performed. All samples, and their adjacent triangles, contained in any cell that the triangle intersects are marked as visible. For each quadtree root that the frustum triangle's projection intersects, the frustum triangle must be clipped against the planes forming the quadtree root.

The clipping proceeds by clipping the initial triangle against each of the coordinate planes. A list of vertices is maintained: one for those that fall above the plane and one for those that fall below it. Since we are testing against the coordinate=0 plane, the intersection test is trivial. Starting with the first plane, for example x=0, we test each triangle edge against the plane. We form two lists, those vertices that lie below the plane (i.e. $v[0] < 0.0$) and those that lie above it (i.e. $v[0] > 0.0$). Points that lie on the plane are added to each list. Where an edge spans the plane, the intersection is found (again an easy computation since we are dealing with coordinate planes). This point is added to each list (see Figure 16b). After the first set of tests, at most two convex polygons are formed. The tests with the next coordinate plane are then performed on these polygons.

After the intersection tests, we will have a maximum of 8 vertex lists, each comprising a convex polygon. Each of these lists is trivially triangulated by forming a triangle from every sequential triple of vertices. The resulting triangles are then processed by the insertion algorithm.

For each quadtree root, we first convert the projected triangle to barycentric coordinates. We traverse to the leaf levels, following each child path where we cannot perform a trivial reject. At each level we pass down the barycentric coordinates of the current quadtree node, the triangle barycentric coordinates, and a scale value. The coordinates of the node are calculated at each level from the parent's coordinates, the child number, and the scale value. The triangle vertex coordinates remain unchanged for better numerical stability. We determine what children the triangle could possibly intersect by a simple test. If all three vertex coordinates have first coordinates greater than a, for example, the triangle must lie in the zeroth child. If this is not the case for any of the sides a,b, or c, we then check to see if the vertices are outside all edges (less than), indicating that it must lie entirely in child 3. Otherwise, the traversal continues down any child path that has any vertex in its positive space. The procedure requires at most 3 additions, 3 shifts, and 9 compares at each level. Figure 17 includes an explanation of the variables and an example. Note that in the example floating point values are used for expository purposes. The variables sa, sb, sc store the result of the test against each half-space respectively for each vertex $t0, t1, t2$.

In the case where the traversal continues down the middle child, child 3, the orientation of the triangle reverses with respect to a,b,c. (see Figure 17d). The maximum barycentric value for a coordinate is no longer at the vertex, but on the opposite edge. While traversing such a branch, the testing operations

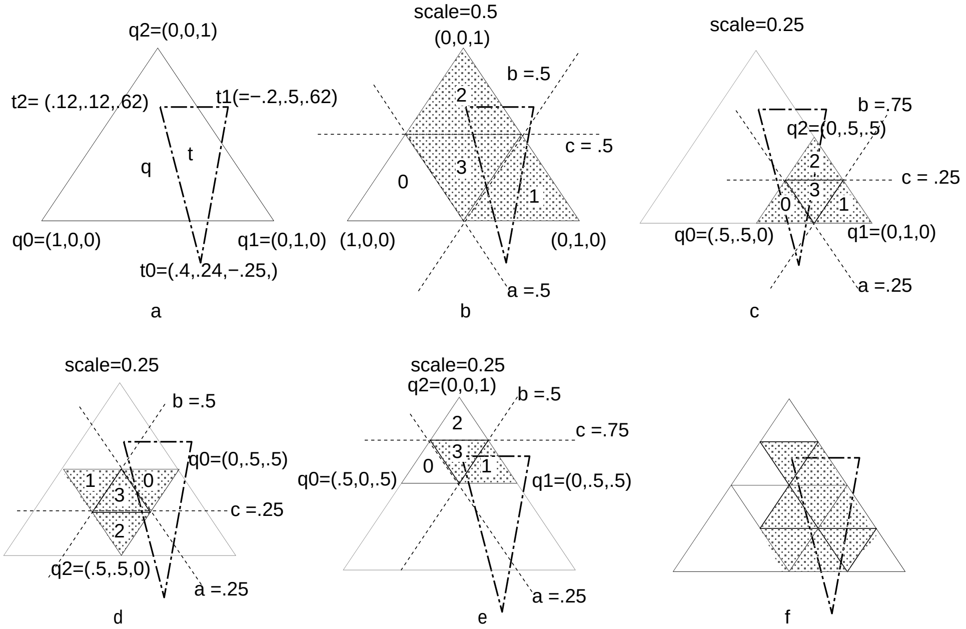


Figure 17: Triangle intersection. Shaded triangles indicate children visited on traversal a) Triangle t and its barycentric coordinates relative to quadtree root q. b) At first level, child 0 does not need to be traversed. c) Child 1 at level 2: all children are traversed: Note that triangle does not intersect child 1 but this is not revealed by trivial reject test. d) Child 3 at level 2: all children must be traversed e) Child 2 at level 2: Child 2 can be rejected f) Resulting set of cells after triangle-triangle test performed at leaf level.

must be adapted accordingly: addition turns to subtraction, and the greater-than test is replaced with a less-than test.

By the time that we have reached a leaf node, we have determined only that the triangle could not be trivially rejected. To determine if the triangle does indeed intersect the cell we must perform a definitive intersection test between the triangle and the quadtree cell at the leaf. To perform this test efficiently we maintain the information about the relative position of the vertices calculated in the previous steps during the traversal. We first check if the triangle can be trivially accepted. This is true if any of the triangle vertices lie in the cell. This is implemented as a bitwise AND: $(sa\&sb\&sc)$ and will be nonzero if any vertex lies in the cell. If no vertex lies in the cell, we next attempt to see if the triangle can be accepted if any of the vertices lie ON a cell edge, giving a conservative answer: if the triangle is just touching the cell it will be considered as IN. This will be true if for any of the vertices, $v[0] = qa$,$v[1] = qb$, or $v[2] = qc$. (see Figure 18).

If the triangle cannot be trivially accepted, we then test for edge crossings. We first test if the edge can be trivially rejected from crossing using the bits calculated earlier. If both vertices on the triangle edge lie to one side of the line defining the quadtree edge, no intersection test is necessary. We also test against the 3 lines, bounding the cell $a = q1[0], b = q0[1], c = q0[2]$. The 6 test lines are shown in Figure 18a. If we must perform the intersection test, it is done parametrically with a double calculation. If is not possible to do the calculation in integer space because we can not guarantee that the intermediate result will not overflow. If the intersection parameter lies between q_i and q_{i+1}, the edges intersect within the triangle, and it is accepted. If not, we keep track of where the intersection occurred, below the low end or above the high end. This test is performed against each triangle edge and each cell edge until a crossing is found, or until we see that we have already visited the cell edge with a crossing once in each direction (low and high). If no such event occurs, the triangle is rejected and the routine terminates. Otherwise, the triangle is added to the quadtree leaf node.

The interval marking operation is done for efficiency as it will potentially find an accept before an

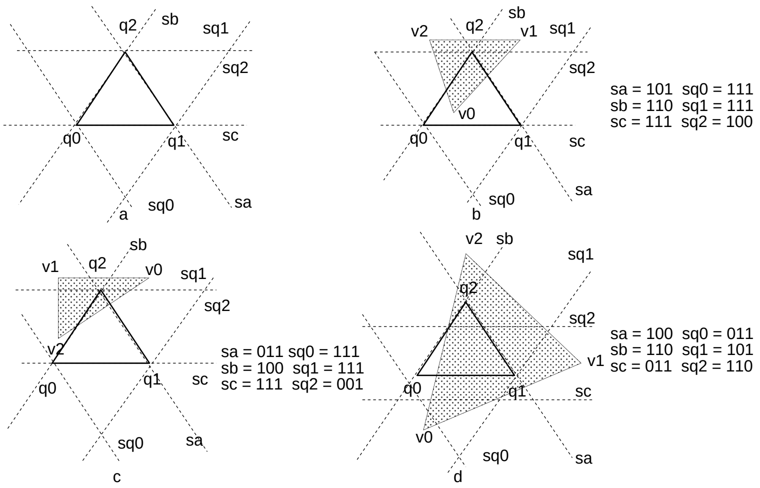


Figure 18: Triangle intersection test a) Quadtree q0,q1,q2. Triangle vertices are tested against lines shown. Lines are labeled with corresponding bit test variables, sa,sb,sc,sq0,sq1,sq2. b) $(sa\&sb\&sc) = 100$, vertex v0 is interior and triangle is accepted. c) Edge v0v1 is trivially rejected because of test sq1. Likewise for v1v2 for test sb. After parametric intersection test, triangle is accepted. d) Edge v0v1 intersects below the range on edge q1q2. Edge v1v2 intersects above the range on edge q1q2. Therefore the triangle is accepted without having to do further testing.

intersection in the interval is found. Figure 18c shows an example. Triangle edge t0t1 crosses quadtree edge q0q1 at the lower end of the boundary. So the triangle is not accepted. Triangle edge t1t2 crosses the same edge at the upper end of the boundary. With this information alone the triangle cannot be accepted. The triangle, however, must intersect the cell if the intersection occurs above and below the range for any triangle edge pair and any cell edge. The triangle can therefore be accepted at this point without any further testing. In the worst case on an accept, we will have to do 2 intersection tests with 3 additions/subtractions, one multiply and divide with double precision math. We pre-calculate the triangle edge differences $(t1 - t0, t2 - t1, t0 - t2)$ and pass them with the triangle, so only a single add, multiply, and divide are required for each intersection test. In the case of an eventual reject, it is again a worst case of 2 intersection tests.

If the current viewpoint coincides with the canonical view, only the projection of the near (or far) frustum face triangles need be tested (see Figure 16c). This is an efficient test in general, but in some cases is too conservative to be useful. If, for example, the user backs up, the projection of the frustum covers the entire sphere (see Figure 16d).

3.4 Rendering During View Motion

When the viewer is moving, no new samples are sent to the display driver. The general algorithm described in Section 3.1 is applied during motion. Because the mesh is constructed relative to the canonical viewpoint, all views that share that viewpoint will get a “correct” image, to the resolution of the provided samples just by rendering the 3D mesh. Once the observer moves off of the canonical viewpoint, artifacts will become visible when previously occluded areas come into view. We accept these artifacts during motion. Once the viewer stops, however, if the view has moved far beyond the canonical view, the mesh is rebuilt from the new view (this is described in more detail in section 3.5).

To optimize the rendering during motion, we first apply view frustum culling and only draw those triangles that are marked as visible . In addition, we utilize the spherical quadtree and OpenGL display lists to improve rendering performance. When the view starts changing, we first mark all quadtree nodes that are visible within the current view, as described in the previous section. The quadtree is traversed only to some predefined level. For each visible node at that level, the triangles are rendered into a display list and displayed. The display list is stored with the node. For each new view, the quadtree is traversed, and the nodes within the new view are visited: if a display list exists already, it is simply called; if this is the first time the node has been visited, a display list is created and stored. This rendering is actually performed in two passes, and two corresponding sets of display lists are potentially stored per node, one for background triangles and one for foreground triangles. This is necessary in order to get the overall rendering order correct. It also makes it possible to perform the necessary translate and scale operations of the background triangles once in the outer loop as matrix transformation calls. This information is view dependent and therefore these operations must be pushed to the outer loop, not only for efficiency, but also to allow the re-use of display lists between views.

The traversal progresses only to the specified level. When a display list is created for a node, all of the nodes children are visited and the associated triangles rendered. Because triangles can span multiple nodes, they may be visited multiple times in the traversal of a subtree. Within the creation of a single display list, the triangles are marked when they are first visited, and therefore only drawn once. Triangles can also span nodes at the level that the display lists are being created. In this case, they must be rendered in both display lists, because it is not known a priori if both nodes will be visible in a particular view.

There is a tradeoff between how many levels to traverse in the tree and how many display lists to store, and how many non-visible triangles get rendered because the node size is coarser than the frustum size, and how many redundant triangles get drawn because of overlap across display lists. We have found traversing two levels to work well in practice. Because the number of nodes is small, the corresponding display lists are stored in a fixed array whose ordering corresponds to a breadth first traversal of the quadtree. If significantly more levels were to be traversed on average, some sort of hashing scheme would be preferable.

3.4.1 Approximate rendering

For lower end machines and complex scenes, we have implemented an approximate rendering scheme that is invoked if the frame rate drops below a set rate during motion. This approach renders an approximation to the current mesh based on the spherical quadtree subdivision. The quadtree is traversed to an adaptively specified level depending on a given *quality* based on the current frame rate and desired interaction speed. The lower the quality, the fewer quadtree levels visited and the coarser, and more quickly rendered, the approximation. The triangles forming the quadtree subdivision are drawn relative to the current viewpoint, with a depth and color averaged from all of the mesh triangles that lie beneath that node. Figure 20 shows an example scene on the left, and the same scene on the right rendered with the approximation.

3.5 Mesh Rebuilding

If the view has moved a relatively large amount off the current view, we must rebuild the mesh with a new canonical view to avoid rendering artifacts. When the mesh is being constructed, we keep track of the average distance of the sample points to the canonical view. Once the view moves a percentage of this distance off of the canonical view, the mesh is rebuilt. We have found a value of 10% to work well in practice. With this simple heuristic, larger view motions are allowed when the scene geometry lies far from the viewpoint, and therefore artifacts are less noticeable, than when the viewed objects are close in the foreground. This process can be slow if there are many samples, and therefore we first cull the samples to the new view frustum and only re-insert those samples that are relevant to the current view.

4 Results

We have implemented the dynamic mesh algorithms and the display driver interface for the holodeck ray cache system. We evaluate our representation in this context according to the following criteria:

- **Image quality**: The representation should provide a reasonable image given a sparse set of samples, and progressively refine as more samples are made available.

- **Update Speed**: The display should be able to update the view as fast as the server can provide samples, either from the cache, or from the ray-tracer(s) performing the sample generation.

- **Feedback during Motion**: The representation should give the viewer reasonable interactive feedback during motion, although the display need not be at full quality.

In terms of image quality, the mesh representation provides a reasonable interpolation of the sparse sample set available at start-up, and the image is progressively refined as more information is received. In the limit, the image converges to the screen resolution as our mesh representation will refine to sub-pixel triangle sizes. Figure 19 shows some example images. The most notable artifact in our images is the lack of sharp edges. The human visual system is very sensitive to discontinuities in the image, especially at the silhouette boundaries of recognizable objects in the scene. Even in a higher resolution image as in the lower right, Figure 20, the lack of clean edges is noticeable.

Running on a low-end SGI O2 workstation, about 10,000 samples can be added to the mesh per second. This rate is sufficient to keep up with a single ray-trace process during progressive refinement, but is not able to update the display and representation as fast as the server can deliver samples in the case of cached rays. When run on an Onyx2 with 21 processors dedicated to the task, the sample generation achieves a near linear speed-up. Unfortunately, the display generation is not parallelized, and while the representation generation and rendering is certainly faster with the benefit of a better CPU and IR graphics, it cannot flush the sample queue fast enough to keep up with the ray-tracers.

On the O2 system, while the interaction speed is sufficient to move about the environment, the motion is jerky. Often during motion, the approximate rendering scheme is activated because the rendering of the triangles is too slow to keep the frame rate. The most noticeable performance deficiency occurs when the viewer has paused after motion, and the entire representation must be re-generated. This lag can be on the order of a couple minutes for a scene in which we have 10s of thousands of samples cached for the now visible portion of the environment. On the higher end configuration, we see the interaction performance rise to a usable level. With the display list rendering, we get fairly smooth interaction during motion.

The image quality during motion varies according to where the observer moves relative to the current view. If the motion is relatively small, the re-projection of the mesh produces a reasonable image. One can zoom in on a point, for example, and get the desired effect of that portion of the environment enlarging, without producing gaps at closer inspection. If the viewer translates or pivots about a point, the appropriate parallax effect will be generated because we maintain the 3D information. Once the view strays too far in this case, areas that have not been previously seen will become exposed. If the viewer then pauses, this information will be filled in.

5 Conclusion

In evaluating the results of this prototype of the dynamic mesh display representation, we feel that the representation succeeds in dynamically providing a reasonable reconstruction of a constantly changing sample set, which at times is very sparse. The current implementation, however, falls short of our goals, most notably in the area of performance. We feel that this first attempt could be improved upon in several ways.

The use of the rendering hardware to perform barycentric interpolation of the sample values provides a quick image reconstruction. This approach, however, does not capture the important color and depth discontinuities in the environment that result in sharp edges in the image. We intend to explore inserting edges at such discontinuities, forming a constrained Delaunay triangulation. We will also consider creating a layered spherical mesh representation where the presence of discontinuities in the input would cause the set of far away points to be put in another mesh layer forming an onion-like structure based around the canonical viewpoint.

In regards to performance, the rebuilding of the mesh after large view motions is currently a major bottleneck. We plan to experiment with an *evolving* mesh where the spherical topology can be continuously updated and re-used between adjacent views, removing the current lag that now occurs whenever the entire mesh needs to be re-built.

We also plan to investigate the use of the dynamic mesh in other environments, with sampled data coming from both real-world and synthetic environments.

Appendix

A Holodeck System Display Driver Interface

The basic tasks specified in the interface with the display driver are the following: incrementally create a display representation from a given set of samples; render the representation for a specified view. We categorize these tasks as mesh construction, and rendering. The specific interface is listed below and each task is described in more detail in the following sections.

- Mesh Construction
  - **Init(n)**: Initialize representation for at least *n* samples. If *n* is 0, clear data structures. Return number allocated.
  - **NewSamp(c,p,d)**: Add new sample with color (RGBE) *c*, world intersection point *p*, and ray direction *d* to representation; remove old samples as necessary. Return identifier to associate with sample. Sample should be output in next call to *Update*.
- Rendering
  - **Update(vp,quality)**: Draw the display representation using OpenGL calls. Assume that current view specified by *vp* has been set up and that the frame buffer has been selected for drawing. The *quality* level is on a linear scale where 100% is full (final) quality. It is not necessary to redraw geometry that has been output since the last call to Clean().
  - **Clean()**: This is called after the display has been effectively cleared, meaning that all geometry must be resent down the pipeline in the next call to Update().

A.1 Implementation of Display driver interface

In the following we summarize what operations are invoked by calls to the display driver interface.

- **Init(n)**: Causes the creation of the sample and triangle data structures of size at least *n* samples. If $n <= 0$, the mesh is cleared and the data structures re-initialized.
- **NewSamp(c,p,d)**: The first time this routine is called, the base mesh is created with the current viewpoint. This is the canonical viewpoint. Until the viewpoint changes, all new samples will be added to this mesh. On each subsequent call to NewSamp, the following steps are carried out:
  1. Point location returns the triangle containing the sample projection.
  2. Sample testing determines if the sample should be included in the mesh.
  3. Sample allocation finds a free sample. Existing samples may need to be deleted to fill this request.
  4. Sample insertion creates new triangles and adds them to the point location data structure. The involves possible swapping and deletion of triangles to maintain the Delaunay condition on the spherical mesh
- **Clean(tmflag)**: Sets a flag indicating that everything should be rendered on the next call to Update. If *tmflag* is set, a flag indicating that tone-mapping should be performed is also set.
- **Update(view,quality)**:
  - **Viewer is moving**: If *quality* is set to **high**, render display lists, otherwise render using quadtree approximation.

- **Viewer is stationary**: If viewer has just stopped and current viewpoint is epsilon from canonical view, rebuild the mesh, re-tonemap and render full result. Otherwise if *Clean* has been called, render the display lists, re-tone-mapping if specified. If *Clean* has not been called do an incremental render update with the newly created triangles that haven't been rendered yet.

B Calculating Minimum Edge Separation

Given the minimum and maximum angles, we can calculate what the minimum edge separation d is, a distance ϕ from the vertex forming the minimum angle. We will assign $\epsilon_e = \frac{d}{2}$. Again, looking at Figure 5, we wish to calculate σ. We use the following constraints to first calculate t, then the halfway vector v:

$$r = (0.0, 0.0, 1.0) \qquad r \cdot s = \cos\phi \qquad r \cdot t = \cos\phi$$

$$||s|| = ||t|| = 1 \qquad s = (0, \sin\phi, \cos\phi) \quad t_x^2 + t_y^2 = 1 - \cos^2\phi = \sin^2\phi$$

$$\sin\gamma = \left|\left|\frac{(s-r)\times(t-r)}{||(s-r)||||(t-r)||}\right|\right| = \left|\left|\frac{(0, \sin\phi, \cos\phi - 1)\times(t_x, t_y, \cos\phi - 1)}{\sqrt{\sin^2\phi + (\cos\phi - 1)^2}\sqrt{t_x^2 + t_y^2 + (\cos\phi - 1)^2}}\right|\right|$$

$$\sin\gamma = \left|\left|\frac{(\sin\phi(\cos\phi - 1) - (\cos\phi - 1)t_y, (\cos\phi - 1)t_x, -\sin\phi t_x)}{\sqrt{2 - 2\cos\phi}\sqrt{2 - 2\cos\phi}}\right|\right|$$

Let $B = \cos\phi - 1, A = \sin\phi B, D = 2 - 2\cos\phi$:

$$\sin\gamma = \left|\left|\frac{(A - Bt_y, Bt_x, -\sin\phi t_x)}{D}\right|\right|$$

$$\sin^2\gamma = \frac{(A - Bt_y)^2 + B^2t_x^2 + \sin^2\phi t_x^2}{D^2} = \frac{A^2 + B^2t_y^2 - 2ABt_y + B^2t_x^2 + \sin^2\phi t_x^2}{D^2}$$

$$D^2\sin^2\gamma = -\sin^2\phi t_y^2 - 2ABt_y + (A^2 + B^2\sin^2\phi + \sin^4\phi)$$

For quadratic equation, $a = -\sin^2\phi, b = -2AB, c = (A^2 + B^2\sin^2\phi + \sin^4\phi - \sin^2\gamma D^2)$ Therefore $t = (2.11 \times 10^{-7}, -4.99 \times 10^{-4}, 9.99 \times 10^{-1})$. The halfway vector, $\frac{t+s}{2}$, normalized: $v = (1.05 \times 10^{-7}, 2.23 \times 10^{-11}, 9.99 \times 10^{-1})$.

C Forward Error Analysis

C.1 Square root round-off error

For the square root computation of number x with accumulated round-off error α, we derive an expression for $\sqrt{x+\alpha}$ in terms of $\sqrt{x}$:

$$(x+\alpha)^{\frac{1}{2}} = \left(x\left(1 + \frac{\alpha}{x}\right)\right)^{\frac{1}{2}} = x^{\frac{1}{2}}\left(1 + \frac{\alpha}{x}\right)^{\frac{1}{2}}$$

The second term can be expressed as a binomial series: $\left(1+\frac{\alpha}{x}\right)^{\frac{1}{2}}=\sum_{n=0}^{\infty}\binom{\frac{1}{2}}{n}\left(\frac{\alpha}{x}\right)^{n}$ where $\binom{\frac{1}{2}}{0}=1$ and $\binom{\frac{1}{2}}{n}=\frac{\frac{1}{2}(\frac{1}{2}-1)(\frac{1}{2}-2)\ldots(\frac{1}{2}-(n-1))}{n!}$ if $n \geq 1$. Therefore:

$$(x+\alpha)^{\frac{1}{2}}=x^{\frac{1}{2}}\sum_{n=0}^{\infty}\binom{\frac{1}{2}}{n}\left(\frac{\alpha}{x}\right)^{n}$$

If we assume $\alpha < x$, the series converges.

$$\lim_{n\to\infty}\left|\frac{a_{n+1}}{a_n}\right|=\lim_{n\to\infty}\left|\frac{(\frac{1}{2}-n)\frac{\alpha}{x}}{n+1}\right|=\left|\frac{\alpha}{x}\right|$$

We first look at the case where the error term $\alpha > 0$. This is an alternating series. For an alternating series, with partial sum indicated by s_j, $|s-s_j| \leq |a_{j+1}|$. Therefore

$$(x+\alpha)^{\frac{1}{2}}=x^{\frac{1}{2}}\left(1+\frac{1}{2}\frac{\alpha}{x}+R_n\right)$$

where $|R_n(\frac{\alpha}{x})| <= |\frac{1}{8}(\frac{\alpha}{x})^2|$
The true value t

$$t=x^{\frac{1}{2}}\pm\left(\frac{1}{2}|\alpha x^{-\frac{1}{2}}|+\frac{1}{8}|\alpha^2 x^{-\frac{3}{2}}|\right)=SQRT(x)\pm\left(\epsilon SQRT(x)+\frac{1}{2}|\alpha x^{-\frac{1}{2}}|+\frac{1}{8}|\alpha^2 x^{-\frac{3}{2}}|\right)$$

We now consider when $\alpha < 0$. In this case, all terms after the first in the geometric series are negative. Since $\left|\binom{\frac{1}{2}}{n}\right| \geq \left|\binom{\frac{1}{2}}{n+1}\right|$ then the remainder $R_n(\frac{\alpha}{x})$ of approximating the square root with only the first n terms satisfies the following:

$$\left|R_n\left(\frac{\alpha}{x}\right)\right| \leq \sum_{n+1}^{\infty}\left|\binom{\frac{1}{2}}{n}\right|\left|\frac{\alpha}{x}\right|^n \leq \sum_{n+1}^{\infty}\left|\binom{\frac{1}{2}}{1}\right|\left|\frac{\alpha}{x}\right|^n$$

The last expression is a geometric series, $\sum_{n=m}^{\infty}\frac{1}{2}\left(\frac{\alpha}{x}\right)^n=\frac{\frac{1}{2}\left(\frac{\alpha}{x}\right)^m}{1-\frac{\alpha}{x}}$, therefore

$$\left|R_n\left(\frac{\alpha}{x}\right)\right| \leq \frac{\frac{1}{2}\left|\frac{\alpha}{x}\right|^{n+1}}{1-\left|\frac{\alpha}{x}\right|}=\frac{|\alpha^{n+1}|}{2x^{n+1}-2|\alpha x^n|}$$

$$t=x^{\frac{1}{2}}\pm\left(\frac{1}{2}|\alpha x^{-\frac{1}{2}}|+\frac{|\alpha^2 x^{\frac{1}{2}}|}{2x^2-2|\alpha x|}\right)=SQRT(x)\pm\left(\epsilon|SQRT(x)|+\frac{1}{2}|\alpha x^{-\frac{1}{2}}|+\left|\frac{\alpha^2}{2|x^{\frac{3}{2}}|-2|\alpha x^{\frac{1}{2}}|}\right|\right)$$

As a worse case, we will take a simpler expression which is the upper bound of the two cases. Both cases share the first two terms, so we will only consider the 3rd term:

$$\frac{1}{8}\left(\frac{\alpha}{x}\right)^2 \leq \frac{\alpha^2}{|2x^2-2|\alpha x||} \leq \frac{\alpha^2}{2x^2}$$

if $\alpha \leq \frac{x}{2}$. We will use the following for square root computation:

$$t=SQRT(x)\pm\left(\epsilon|SQRT(x)|+\frac{1}{2}|\alpha x^{-\frac{1}{2}}|++\frac{1}{2}|\alpha^2 x^{-\frac{3}{2}}|\right)$$

C.2 Division round-off error

For division of two terms x, y with accumulated round-off error, α, β: $\frac{x+\alpha}{y+\beta}$, we derive an expression in terms of $\frac{x}{y}$:

$$\frac{x+\alpha}{y+\beta} = \frac{x}{y+\beta} + \frac{\alpha}{y+\beta}$$

$$\frac{x}{y+\beta} - \frac{x}{y} = \frac{xy}{y(y+\beta)} - \frac{x(y+\beta)}{y(y+\beta)} = \frac{-x\beta}{y(y+\beta)}$$

$$\frac{x}{y+\beta} = \frac{x}{y} + \frac{-x\beta}{y(y+\beta)}$$

$$t = \frac{x}{y} \pm (\frac{|\alpha|}{(|y| - |\beta|)} + \frac{|x\beta|}{|y|(|y| - |\beta|)}$$

$$t = x \oslash y \pm (\epsilon |x \oslash y|) + \left| \frac{|y\alpha| + |x\beta|}{y(|y| - |\beta|)} \right|$$

References

[1] D. H. Bailey. Mpfun: A portable high performance multiprecision package. Technical Report RNR-90-022, NASA Ames Research Center, October 1990.

[2] L. Darsa and B. Costa. Multi-resolution representation and reconstruction of adaptively sampled images. In *SIBGRAPI*, pages 321–328, 1996.

[3] L. Darsa, B. Costa, and A. Varshney. Navigating static environments using image-space simplification and morphing. In *ACM Symposium on Interactive 3D Graphics*, pages 25–34, 1997.

[4] D. Goldberg. What every computer scientist should know about floating-point arithmetic. In *ACM Computing Surveys*, volume 23(1), pages 5–48, March 1991.

[5] L. Guibas and J. Stolfi. Primitives for the manipulation of general subdivisions and the computation of voronoi diagrams. *ACM Transactions on Graphics*, 42:2:74–123, 1985.

[6] G. W. Larson. The holodeck: A parallel ray-caching rendering system. In *Proceedings Eurographics Workshop on Parallel Graphics and Visualization*, September 1998.

[7] G. W. Larson, H. Rushmeier, and C. Piatko. Visibility matching tone reproduction. *IEEE Transactions on Visualization and Computer Graphics*, 25:4:291–306, December 1997.

[8] G. W. Larson and R. Shakespeare. *Rendering with Radiance.* Morgan Kaufmann Publishers, San Francisco, CA, 1998.

[9] D. Lischinski. *Incremental Delaunay Triangulation*, pages 47–49. A P Professional, San Diego, CA, 1994.

[10] F. Pighin, D. Lischinski, and D. Salesin. Progressive previewing of ray-traced images using image-plane discontinuity meshing. In *Proceedings 8th Eurographics Workshop on Rendering*, pages 115–124, June 1997.

[11] F. P. Preparata and M.I. Shamos. *Proximity: Fundamental Algorithms*, pages 204–223. Springer-Verlag, New York, NY, 1985.

[12] J. R. Shewchuk. Robust Adaptive Floating-Point Geometric Predicates. In *Proceedings of the Twelfth Annual Symposium on Computational Geometry*, pages 141–150. Association for Computing Machinery, May 1996.

[13] P. Shirley, B. Wade, P. M. Hubbard, D. Zareski, B. Walter, and D. P. Greenberg. Global illumination via density estimation. In *Rendering Techniques (Proceedings of the Eurographics Workshop)*, pages 219–230, 1995.

[14] F. Sillion, G. Drettakis, and B. Bodelet. Efficient imposter manipulation for real-time visualization of urban scenery. In *Computer Graphics Forum (Proceedings Eurographics)*, pages 207–218, 1997.

[15] G. Ward. *Real Pixels*, chapter Image Processing, pages 80–83. A P Professional, 1991.

[16] G. Ward. The RADIANCE lighting simulation and rendering system. In *Computer Graphics (Proceedings ACM SIGGRAPH)*, pages 459–472, 1994.

[17] G. Ward and M. Simmons. The holodeck interactive ray cache. *to appear: ACM Transactions on Graphics*, ?:?:?–?, 2000.

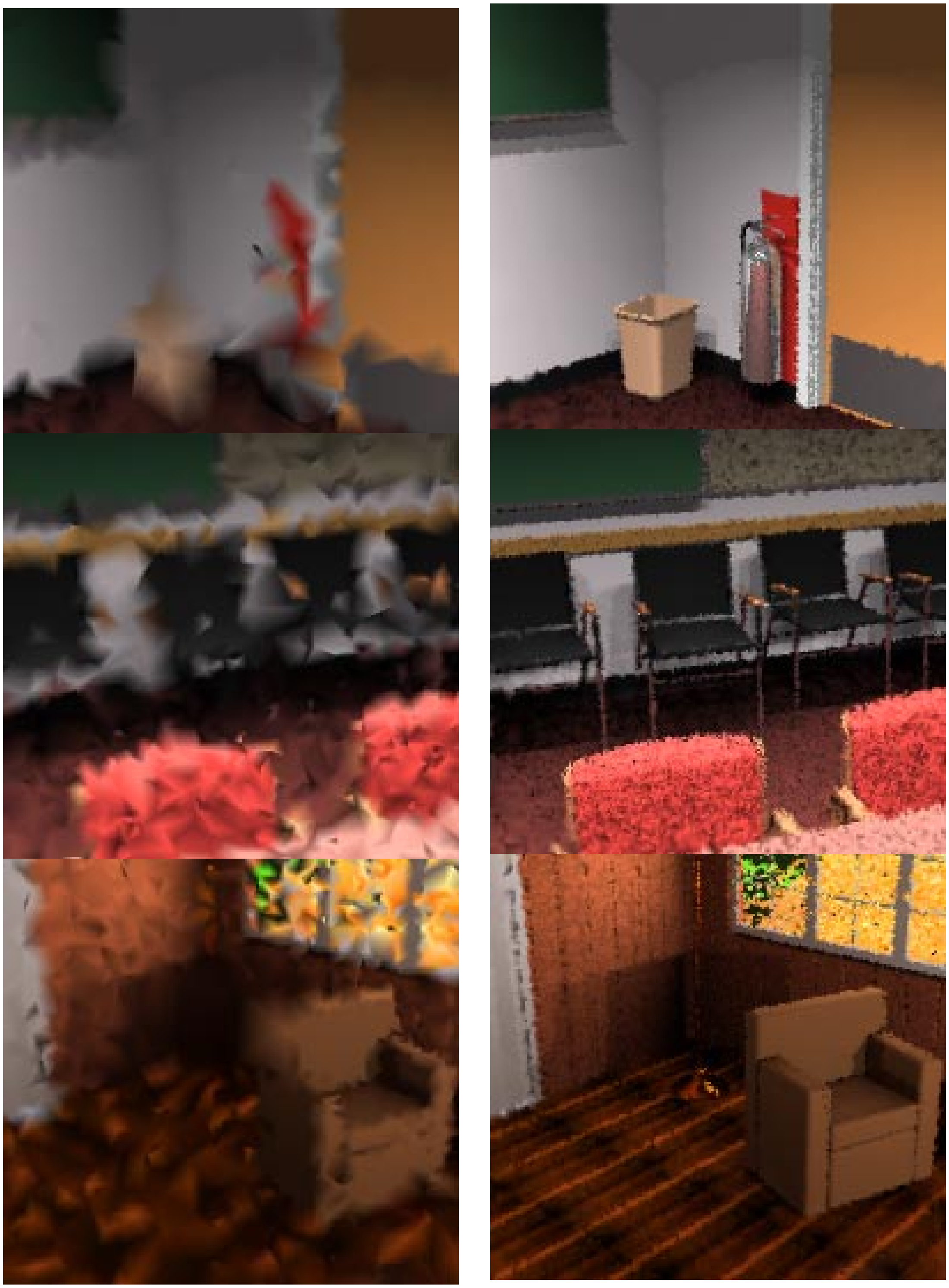


Figure 19: Selected images: In each case the left image shows the display when the view is first seen, and the right image after the viewer pauses a few minutes at the same location.

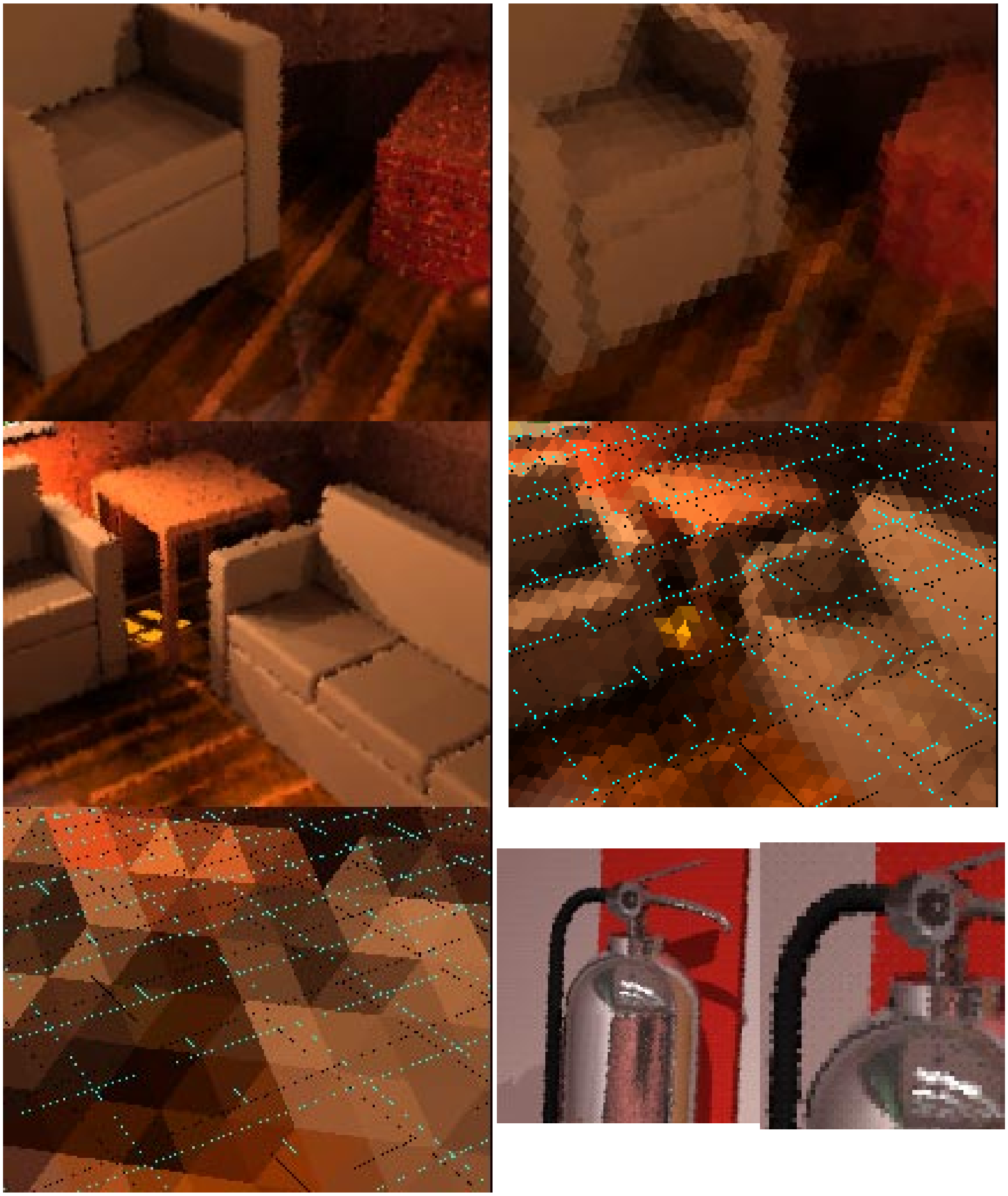


Figure 20: Approximate Rendering: Mesh rendering is shown, and then example approximate renderings used in motion. Extinquisher image after 5 minutes on an O2, zoomed version on right.