Chapter 6

# Implementation Details

The algorithms described in the previous chapters have been implemented in a radiosity code base I call “radiator”. In this section I will discuss some of the more interesting implementation details of my system. We start with a basic overview of the component parts of the system, and then in turn look at details of the face cluster system, and the radiosity system built on top of it.

## 6.1. System Architecture

The basic architecture of the face-cluster radiosity system is shown in **Figure 81**. All components share a common library, “gcl”, which provides rendering and scene-graph services, scene-file handling, and other basic graphics utilities. In particular the library has support for multiresolution models as one of its scene-file types—both the geometry-simplification edge-collapse models, and the geometry-aggregation face-cluster models. Both types of model share the same basic multiresolution data structure. (Indeed, both kinds of model can be merged into a meta model containing basic geometry, a vertex hierarchy, and a cluster hierarchy.)

An original base model file can be processed and an ascii log-format cluster hierarchy output by the “make-fch” utility. (File formats supported include AutoDesk’s 3DS, Cyberware’s PLY, MGF, and the ascii Wavefront OBJ format.) The gcl library can then translate this file into a convenient memory-mappable

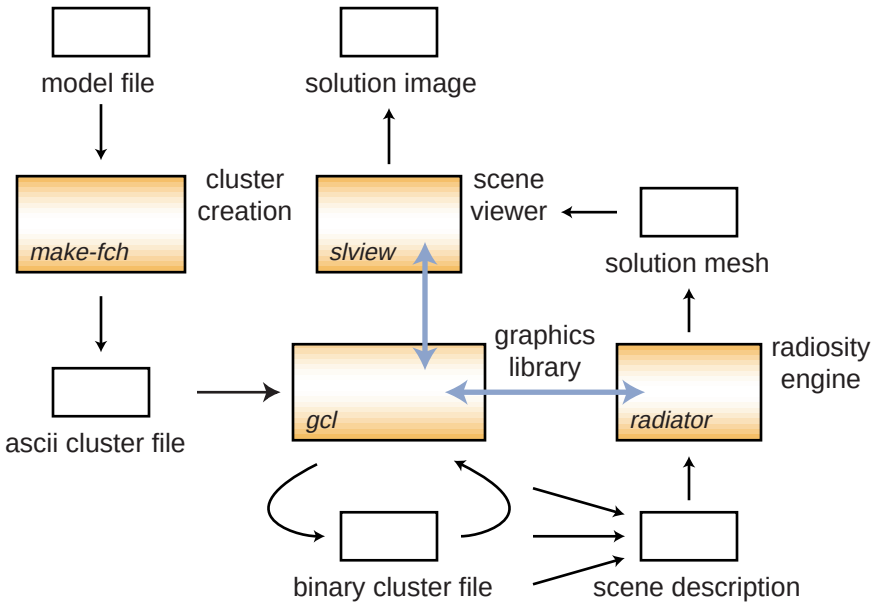


**Figure 81:** Interaction between various components of the radiosity system.

binary format. (An ascii format is used for the initial cluster file both for flexibility and because it can be moved between machines with different byte orders, whereas the binary format cannot.)

The radiosity system, “radiator”, uses gcl to read scene descriptions that reference such models. (These descriptions describe the scene hierarchy with embedded geometric transforms and material properties.) Where necessary, it creates radiosity elements from face clusters on demand. (See **Section 6.3.5**.) Once it has calculated a radiosity solution, it writes out the corresponding solution mesh with vertex colours. If no input polygon refinement has taken place for a particular scene model, only a list of vertex colours needs to be written out; the geometry of the model stays as is.

These view-independent solution meshes can then be viewed with “slview”, and images generated from various vantage points. They can also be

fed to an animation program for generating fly-throughs. The entire system is available on-line at http://www.cs.cmu.edu/~ajw/thesis-code/.

## 6.2. Face Clusters: Algorithms and Data Structures

In this section we describe generic face cluster algorithms and data structures. In the next section we will specialise them for radiosity.

### 6.2.1. A Clustered Model

A clustered model is stored using a number of contiguous arrays, as in **Listing 3**. The base model is stored in the **`points`** and **`faces`** arrays, containing a list of vertices and indexes into that list for each model triangle respectively. Other attributes such as vertex or face colours or texture coordinates can also be stored, but we will consider only geometry here.

```
struct ClusteredModel                              array length
{
     PointList     points;                             v
     FaceList      faces;                              f
     ClusterList   clusters;                           f - r
     IndexList     rootClusters;                       r
     ActiveList    clustersActive;                     2f - r
};
```

**Listing 3:** A face-clustered model's data structure.

The face clusters themselves are stored in the **`clusters`** array. Only internal nodes are stored; there are no clusters corresponding to the model's faces, in the interest of saving storage space. Each cluster stores information about its bounding box, sum area normal, and two child IDs, as we shall see in the next section.

If there are $f$ total faces in the model, then there are a total possible number of $f-1$ face clusters, assuming that the model is completely connected and thus there is only one root node. This is not always the case, however, as each separate

connected surface component corresponds to a single hierarchy, and a model may contain more than one such component. In the more general case, if there are $r$ root nodes, then there are $f - r$ face clusters. The **`rootClusters`** array holds the IDs of all root clusters in the model.

It is convenient to assign a universal ID to all nodes in the cluster hierarchy, both faces and face clusters, as follows. The faces are numbered 0 to $f - 1$, according to their occurrence in the faces array, and the clusters from $f$ upwards, according to their order of creation by the clustering program. This numbering corresponds to a so-called *log format*; it is the order we get if we start at the model leaves and write each clustering operation to a log file as we successively merge clusters. This numbering has the following advantages:

- The child IDs of any cluster are always smaller than its own ID. This means that by iterating linearly through the IDs we can accomplish a preorder (forwards) or postorder (backwards) traversal of the hierarchy, as illustrated in **Listing 4**[1]. Many common operations on the cluster hierarchy can be implemented in this way.
- We can detect when we are at a leaf by checking the child ID of a particular cluster. If it is less than $f$, the child is a leaf, and the index of the corresponding face in the faces array is just the child ID.

The resulting set up is shown in **Figure 82**. The face clusters are generally stored in order of their ID in the clusters array, to make it easy to map between the ID and their position in the array. To make truncation of the hierarchy easier, we can store them in reverse order, so that discarding clusters below a certain point in the hierarchy is simply a matter of shortening the cluster array.

Finally, the last field, **`clustersActive`**, is used by visualisation software to track a cut through the cluster hierarchy to be displayed. Any such cut is guaranteed to completely cover the model's surface without redundancy; we are merely varying the number and size of clusters used to do so. The field holds mark bits corresponding to every ID in the cluster hierarchy. The clusters or faces in the current cut have their mark bits set, and the rest are clear. The cut can be moved lower in the hierarchy (and thus the number of active clusters increased) by doing a preorder traversal, and marking the children of any currently active cluster. Likewise, the active cut can be moved higher by doing a postorder traver-

1. These are not strict pre and postorder traversals, as the traversal does not visit child and parent nodes in consecutive order; it is only guaranteed that the parent will be visited before the children and vice-versa. For most purposes this suffices, however.

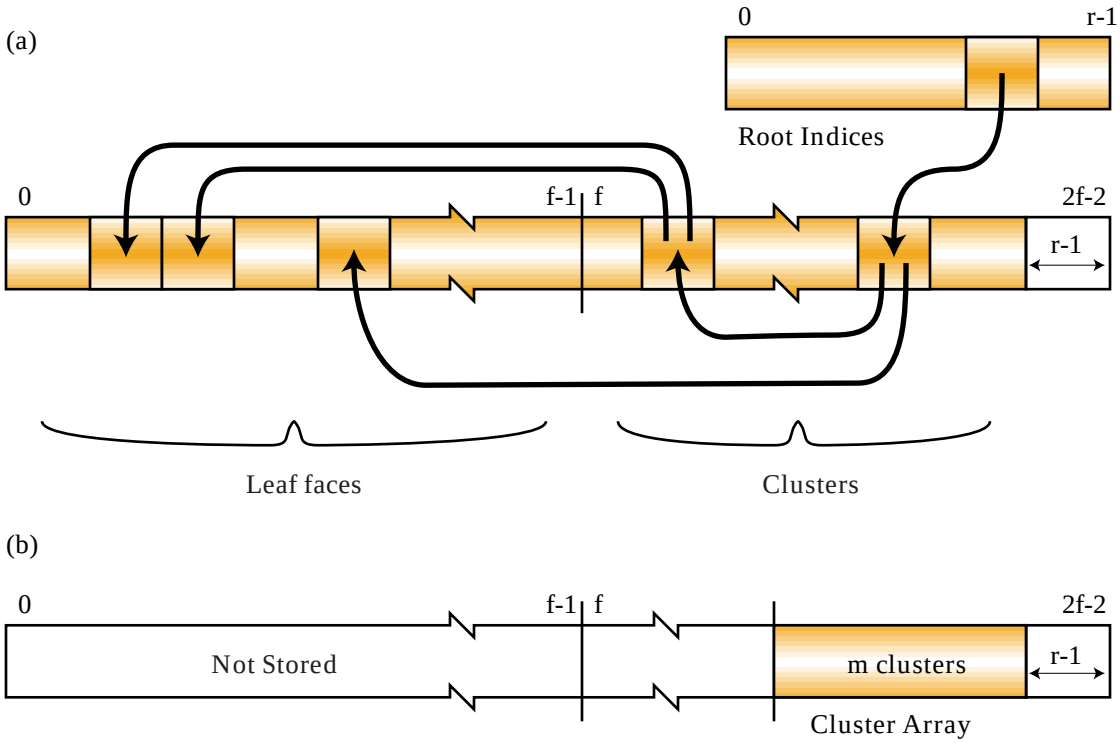


**Figure 82:** The cluster hierarchy data structure. In (a) we see an array of $r$ root indices, one for each connected submesh. Each root index pointing into the cluster array, which contains the clusters written in log format, from the original model faces to the last cluster. There are $f$ leaf faces, and $2f - r - 2$ total clusters. As shown in (b), we don't store clusters for the leaf faces, and if we want to truncate the hierarchy, we can optionally store only the top $m$ clusters.

sal, and marking any cluster whose children are both currently active[1]. An illustration of the latter is shown in **Listing 4**.

Generally, this field is unused (and unallocated) by the radiosity simulation.

1. This is similar to the approach taken by Hoppe in his method for view-dependent refinement using progressive meshes [Hopp97].

```
// a preorder traversal: from the leaves to the root nodes
for (i = 0; i < maxID; i++)
     VisitNode(i);

// a postorder traversal, checking for cluster type
for (i = maxID - 1; i >= 0; i--)
     if (i < faces.NumItems())
          VisitFaceNode(faces[i]);
     else
          VisitClusterNode(Cluster(i));

// adapting the active hierarchy cut upwards
for (i = 0; (activeClusters > targetClusters) && i < maxID; i++)
{
     if (clustersActive[Cluster(i).child[0]]
          && clustersActive[Cluster(i).child[1]])
     {
          // if both children are active, deactivate them and
          // active the parent

          clustersActive[Cluster(i).child[0]] = 0;
          clustersActive[Cluster(i).child[1]] = 0;
          clustersActive[i] = 1;
          activeClusters--;
     }
}
```

**Listing 4:** Traversing the face cluster hierarchy. The **`Cluster()`** function maps an ID to the appropriate cluster in the clusters array.

## 6.2.2. The Face Cluster Data Structure

To represent the oriented bounding box of the cluster we need to store both an orientation, and minimum and maximum bounds of the cluster in each of the three directions of that orientation. The orientation can be represented by an $\Re^3$ rotation, i.e., a $3 \times 3$ matrix. We can save space over the nine floats required to store a full rotation matrix (or more correctly, the six floats required to store the symmetric matrix) by storing it as a quaternion, which requires only four floats. (As a comparison, RAPID [Gott96] also uses quaternions, though they store a centroid and the three dimensions of the OBB instead of three min / max pairs). The algo-

rithm for converting the three principle axes to a quaternion and back [Shoe85] has proved to be very stable. Thus it requires a total of 10 floating point numbers to store the oriented bounding box.

Storing the side areas of the cluster, which we need to construct our upper bounds on projected area, requires a further six floats. It is tempting to use instead the side-areas of the bounding box, which are easily available from the OBB min and max fields, and as discussed previously are in turn an upper bound on the cluster side-areas. A compromise, which I use, is to store the more accurate sampled side-areas as 8-bit fixed-point fractions of the side-areas of the bounding boxes. Of course, when converting the fractional area to the 8-bit representation, the fraction must be rounded up so as to maintain our upper bound.

The face cluster data structure is shown in **Listing 5**. Further space savings could be possible by using Deering's normal encoding technique in conjunction with a projected-area scale factor to represent the sum area normal. Such compression must be approached with caution given how heavily we rely on the sum area normal [Deer95].

```
struct FaceCluster                                              bytes
{
     Int           child[2]; // two child cluster IDs          (8)
     Quaternion    axesQuat; // orientation of the OBB         (16)
     Point         min; // min point in OBB coord system       (12)
     Point         max; // max point in OBB coord system       (12)
     Vector        areaNormal;   // sum area normal            (12)
     Float         totArea;      // total surface area         (4)
     Byte          sideArea[6];  // side area fractions        (6)
};
```

**Listing 5:** The face cluster data structure. The storage in bytes of each field is shown in brackets. The total structure requires 66 bytes.

We can write the total memory use of the face cluster hierarchy data structures as:

```
totalMem = (66 * clusters + 12 * faces + 12 * vertices) bytes
```

If we assume there is only one root cluster, and thus $f - 1$ clusters, and that, as in the limit case for a manifold, there are twice as many faces as vertices, we have:

```
totalMem = (84f - 66) bytes
```

Thus on average we incur an overhead of 84 bytes per face, assuming we store the full hierarchy.

### 6.2.3. Reordering the Hierarchy

We can reorder the faces in any face cluster model to have the property that, for any cluster in the hierarchy, all its corresponding faces occur in sequential order. This is easily accomplished by a pre-order traversal of the hierarchy in which faces are added to a new face list in the order in which they are encountered. This has the following benefits:

- It improves the locality of face storage, because faces belonging to the same cluster are always guaranteed to be stored contiguously.
- It makes it easier to locate the faces corresponding to a particular cluster. Rather than recursively descending the hierarchy to find these faces, we descend only the left-most and right-most child paths, as in **Listing 6**, and can then perform any operation over the faces via a simple for loop. Indeed, if we are willing to add an overhead of two indices to each cluster, we can get the corresponding range of IDs directly.
- It makes it possible to handle truncation of the hierarchy cleanly. If we wish to modify the hierarchy to remove all clusters below ID *ic*, we simply iterate through the remaining clusters, and replace all indices smaller than *ic* with their corresponding left ID or right ID. Again, see **Listing 6**.

To test the benefits of hierarchy reordering and truncation, a simple radiosity simulation was run using different versions of the clustered 1,000,000 triangle “buddha” model from **Figure 79**. All file caches were cleared before each run[1], and the results of five runs averaged together for each version. The results are presented in **Table 9**. Shown are the total size of the face-clustered model, the time taken for radiosity solution, the time for the final post-processing pass (which touches all faces of the model), and the total wall-clock time for the radiosity algorithm. For the original clustered model, the wall-clock time is much longer than the total measured solution and post-processing time (47s compared to 6.5s), indicative of the heavy disk activity taking place. Both the reordered cluster hierarchies do not suffer from this problem; the final post-processing pass becomes a simple ordered

1. Under Linux this is most easily achieved by unmounting and then remounting the partition the cluster file resides on.

```
VisitClusterFaces(Int clusID)
{
     leftFaceID = rightFaceID = clusID;

     while (!IsFace(leftFaceID))
          leftFaceID = Cluster(leftFaceID).child[0];

     while (!IsFace(rightFaceID))
          rightFaceID = Cluster(rightFaceID).child[1];

     for (i = leftFaceID; i <= rightFaceID; i++)
          PerformFaceAction(i);
}

PostOrderAction(Int i, Int truncID)
{
     if (Cluster(i).child[0] < truncID)
          Cluster(i).child[0] =
               Cluster(clus.child[0]).child[0];

     if (clus.child[1] < truncID)
          Cluster(i).child[1] =
               Cluster(clus.child[1]).child[1];
}
```

**Listing 6:** Visiting the faces in a cluster with a reordered hierarchy, and truncating the hierarchy.

| Type of Scene | Size | Soln. | Post. | W/C |
|---|---|---|---|---|
| Base model | 18.5 MB | | | |
| Original clustered model | 87 MB | 4s | 2.5s | 47s |
| Reordered | 87 MB | 4s | 1.8s | 8.7s |
| Reordered and truncated | 21.4 MB | 4s | 0.9s | 8.0s |

**Table 9:** The effect of reordering face clusters.

step through the faces array. The truncated hierarchy not only takes only 3 MB of extra storage over the 18.5 MB base model, but saves a little on post processing

time, because the distance to the end of the cluster hierarchy from the radiosity solution leaf nodes is smaller.

## 6.3. Radiosity Implementation

In this section we will look at some of the specifics of the face cluster radiosity system built for this thesis.

### 6.3.1. Volume Clustering

To build volume clusters, I followed the methods described in [Gibs96, Sill95a]. An octree that encloses the scene is created, and scene polygons are placed within that octree according to their size and position. This is the “tightest-fit octree” method referred to by Hasenfratz et al. [Hase99]; see that paper for alternative volume clustering algorithms. In my experience this method provides acceptable quality, and is very fast. Because I don’t also use the volume cluster data structure for visibility testing, the advantages of more sophisticated schemes summarized by Hasenfratz et al. are not so important.

### 6.3.2. Visibility Testing

Visibility is sampled using ray-tracing; the spatial data structure used for acceleration is a nested grid data structure [Fuji86, Jeva89]. Traditionally, of the ray-tracing optimising schemes, grid-traversal is amongst the fastest, if not the fastest algorithms for ray-casting against a scene with many similar-sized primitives. When the scene contains primitives with widely disparate sizes, however, this approach suffers from the so-called “teapot in a stadium” problem. A single uniform-density grid cannot efficiently cover all primitive sizes, leading to many primitives in some cells.

To solve this problem, we can instead create a number of nested grids, each adapted to a particular primitive size. To build such nested grids, within each grid primitives are sorted into cells, and, depending on their local density and size, become candidates for insertion into a subgrid. The grids rely on lazy evaluation—this sorting process only takes place when a ray hits the grid in question. Thus if no rays hit a subgrid, the primitives within it are never processed, saving considerable pre-processing time. However, this optimization is not so important for most radiosity solutions, where the scene tends to be completely and evenly sampled.

Fractional visibility is used during simulation, but the visibility can be resampled at higher resolution during the final push-to-leaves step. This is discussed further in **Section 6.3.7**. The visibility data-structure is usually built from the input polygons only; it ignores subdivided input polygons and face clusters.

### 6.3.3. Approximate Visibility

While the nested grid scheme is highly efficient, even for large models, for very complex scenes the ray-casting part of the algorithm ends up being carried out at a much higher resolution than the finite-element part of the algorithm. This can result in the occlusion-testing data-structures dominating the time and memory cost of the algorithm.

Ideally, we would like to incorporate some kind of multiresolution visibility algorithm, taking advantage of the face-cluster hierarchies to test visibility against coarser resolutions of the model when our accuracy requirements are not high. This is an interesting area of research, but outside the scope of this thesis.

A simpler approach I currently employ is to use the face cluster hierarchies to construct an approximate occlusion data-structure for the scene. This can be seen as a combination of using the cluster hierarchy to test visibility, as per Sillion [Sill95a], and the voxel-based opacity grids of Christensen and Gibson [Chri95, Gibs95]. It can be summarized as follows:

- adapt the complexity of each cluster hierarchy to a user-specified fraction $f_v$ of the total number of clusters. This is achieved by starting at the root clusters, and expanding the active cluster list until it covers the required number of clusters, as in **Listing 4**;
- add the bounding boxes of these clusters to the nested grid data structure;
- ray-trace against the bounding boxes for the purposes of occlusion testing, optionally using the projected area estimates to determine fractional opacity[1].

There is one minor adjustment needed to make this approach work well. To avoid self-occlusion by clusters, we ignore a cluster for the purpose of occlusion testing when either the start or end point of a shadow test ray falls within it. (Recall that intra-cluster occlusion is subsumed into the visible projected area estimates of **Chapter 4**.)

1. The probability of a ray hitting a surface is proportional to its visible projected area in the direction of the ray [Gold87]. Thus a fractional visibility or albedo estimate for a cluster's bounding box can be obtained by taking the ratio of the bounding box's projected area to that of the surface it contains.

The major disadvantage of this approach is that another parameter must be set for the rendering process. Generally, the smaller epsilon is chosen to be, the larger $f_v$ should be. It is desirable that the clusters used for visibility testing be smaller than those used in the radiosity solution.

This approach can lead to considerable speed and memory savings for large models. See **Section 7.4** for an example.

### 6.3.4. Transport Calculation and Refinement

In face cluster radiosity we must handle a number of different types of light transfer; radiosity exchange between face clusters, standard planar elements, and volume clusters. This can lead to problems in choosing an error metric that is consistent for all transfers. To handle these in a common framework we use sampling across the receiver to estimate the error in the transfer at the same time as we estimate the transfer itself. We use the $L_1$ norm to measure error, i.e., the *BFA*-weighted refinement discussed in [Smit94].

The basic "link" data structure representing radiosity transport between two elements is shown in **Listing 7**. The pseudo code shown in **Listing 8** illustrates how the error and the transport vector **m** can be calculated for a link, given a number of sample points over each element. For face clusters, the minimum and maximum projected areas are calculated as described in **Chapter 4**. For an input polygon or refinement thereof, the upper and lower bounds are both just the projected area of the polygon. For a volume cluster, the upper bound is established by summing the upper bounds of all the root face clusters or polygons it contains, and the lower bound is taken to be zero.

Given the transport error, we must also decide whether the corresponding transport link should be refined or not by comparing it to a refinement epsilon. For links that are partly occluded, the refinement epsilon is reduced to encourage subdivision at shadow boundaries. (The refinement epsilon controls link subdivision; links with transport error greater than this are split.) Pseudo code for the resulting refinement oracle is given in **Listing 9**.

The algorithm presented chooses which end of the transport link to subdivide by comparing the total surface areas of the elements at either end. This has the advantage of simplicity and robustness, but we can do better with a bit more calculation. **Listing 10** gives pseudo code corresponding to the approach outlined in **Section 4.4.3**. Here, the fraction of the estimated error due to each element is calculated, and the element contributing the most error is subdivided.

```
struct VecLink
{
     Element   from, to;     // can be patches, face or volume
                             // clusters
     Vector    m;            // transport vector m
     Float     error;        // ⟨G_{D←S}⟩ from § 4.4.2
     Float     visibility;   // average visibility between
                             // the two elements
}
```

**Listing 7:** The vector radiosity transport data structure.

### 6.3.5. Face Cluster Solution Elements

The data structure for a generic, non radiosity-specific face cluster was presented in **Section 6.2.2**. For those clusters used in the radiosity solution, we must in addition store radiosities and other extra information. Because typically the number of clusters used as radiosity elements is much smaller than the total number of clusters, it makes sense to save time by storing some face cluster attributes transformed into world space coordinates, rather than performing these transformations on the fly. The radiosity element data structure is shown in **Listing 11**.

The use of vector instead of scalar irradiance means that, compared to the storage required for a face in standard hierarchical radiosity, we store 9 real numbers per hierarchical element instead of 3, and 3 reals per link instead of 1. Although the face cluster hierarchical elements are more expensive than standard Haar elements, they are in general more lightweight than volume cluster elements, which require 8 child pointers in our implementation, and much more lightweight than storing a general radiance distribution.

### 6.3.6. Irradiance Vector Interpolation

One of the great advantages that face clusters have over volume clusters is that they are intrinsically surface-based. This means that it is much easier to interpolate irradiance values across the cluster. In volume-based clusters, there is no

```
VecLink::CalcTransport()
{
     m = 0;
     factor.InitBounds(-infinity, infinity);

     // find samples over the upper face of each cluster's
     // bounding box
     p1 = from.CreateSamples(numSamples);
     p2 = to.CreateSamples(numSamples);

     for (i = 0; i < numSamples; i++)
     {
          r = p1[i] - p2[i];

          rLen = len(r);
          if (rLen > 0.0)
          {
               r /= rLen;
               m2Len = pi * sqr(rLen) + from.area;

               l1 = from.MinProjArea(-r);
               l2 = to.MinProjArea(r);
               factor.UpdateBounds(l1 * l2 / m2Len);

               u1 = from.MaxProjArea(-r);
               u2 = to.MaxProjArea(r);
               factor.UpdateBounds(u1 * u2 / m2Len);

               m += 0.5 * (l1 + u1) * r / m2Len;
          }
     }

     m *= visibility / numSamples;
     error = factor.Range();

     linkViable = (len(m) > 0.0 || error > 0.0);
}
```

**Listing 8:** Calculating link transport and error. We must calculate an estimate of the transport vector **m**, and the error in the transferred power. Only if both are zero should we ignore this link for the purposes of further refinement (**`linkViable`**).

```
RefChoice VecLink::RefineOracle()
{
     w = abs(1.0 - 2 * visibility) * (1.0 - visEps) + visEps;

     // compare error in power (radiosity * error) to epsilon
     if (error * from.B < w * ε)
          return(SubdivideNone);

     if (to == from)
          return(SubdivideBoth);

     if (to.area > from.area)
          if (to.area < minRefinementArea)
               return(SubdivideNone);
          else
               return(SubdivideTo);
     else
          if (from.area < minRefinementArea)
               return(SubdivideNone);
          else
               return(SubdivideFrom);
}
```

**Listing 9:** RefineOracle: deciding how to refine cluster links given the transport error and average visibility.

notion of a largely co-planar, contiguous surface, and interpolation leads to problems with visibility, and requires deciding which reference planes to use for sampling irradiance. (Greger and Shirley's "irradiance volume" approach is one solution to this, albeit an expensive one [Greg98].)

A useful strategy in dealing with inter-cluster irradiance discontinuities is as follows:

- On the final solution pass, resample the point irradiance at the corners of each face cluster, rather than once over the entire cluster.
- During the push-to-leaves phase, interpolate the irradiance vectors for the cluster a leaf polygon falls within in order to find the irradiance at its vertices.

```
RefChoice VecLink::FindClusterToSubdivide()
{
     pa1.InitBounds(-infinity, infinity);
     pa2.InitBounds(-infinity, infinity);
     p1 = from.CreateSamples(numSamples);
     p2 = to.CreateSamples(numSamples);

     r = normalised(from.centre - to.centre);
     s1 = from.MaxProjArea(-r);
     s2 = to.MaxProjArea(r);

     for (i = 0; i < numSamples; i++)
     {
          r = to.centre - p1[i];
          rLen = len(r);
          if (rLen > 0.0)
          {
               r *= s2 / (rLen * (pi * sqr(rLen) + to.area));

               pa1.UpdateBounds(from.MinProjArea(r));
               pa2.UpdateBounds(from.MaxProjArea(r));
          }

          r = from.centre - p2[i];
          rLen = len(r);
          if (rLen > 0.0)
          {
               r *= s1 / (rLen * (pi * sqr(rLen) + from.area));

               pa1.UpdateBounds(to.MinProjArea(r));
               pa2.UpdateBounds(to.MaxProjArea(r));
          }
     }

     if (pa1.Range() < pa2.Range())
          return(SubdivideTo);
     else
          return(SubdivideFrom);
}
```

**Listing 10:** Choosing which cluster to subdivide by estimating the error contributed by each to the transport.

```
struct RadFaceCluster : Element
{
     Int      fci;     // ID of corresponding face cluster
     ModelInfo* info;  // associated model's information

     // copies of face cluster items in world-space coordinates

     Vector   areaNormal;   // sum area normal
     Float    sideArea[6];  // for VPA upper bound
     Float    totArea;      // total surface area
     Float    maxVisArea;   // max externally visible area

     Element  *child[2];    // solution child elements
     Colour   B;            // cluster radiosity
     IrradVec R;            // vector irradiance: § 3.4.2
};
```

**Listing 11:** A radiosity face cluster solution element. The **`ModelInfo`** structure contains auxiliary information about the model containing this element, such as a pointer to its cluster data structures (see **`Listing 3`**), and its position and orientation in the scene. Thus a single model can be instantiated at multiple points in the scene. An **`IrradVec`** contains a vector for each of the colour channels represented.

This process is illustrated by **Figure 83**, and the dramatic results it can have in some situations are shown in **Figure 84**.

We can use the midpoint of each of the four edges of the cluster's bounding box that are parallel to the sum area normal to sample vector irradiance. Unfortunately, this is not necessarily the best place to sample visibility. If clusters lie on a concave surface, and their bounding boxes overlap, it is possible for a sample point to lie beneath the adjacent cluster. This problem can be largely overcome by insetting the sample points from the edge of the bounding box by a small amount. However, this leads to a further problem: interpolation overshoot. When we use our irradiance vector samples to reconstruct the vector irradiance at a point on the cluster, we may have to extrapolate if such a point lies outside the samples, which is a possibility now that we have inset them from the edges of the cluster. If the nearest sample is small in magnitude, or zero, this can lead to negative irradiances.

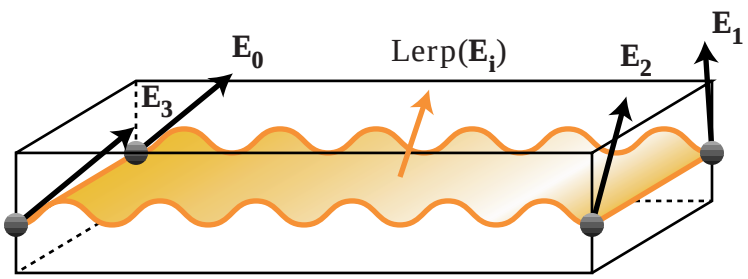


**Figure 83:** Irradiance vector interpolation. The irradiance vector at any surface point within the cluster is interpolated from samples at the corners of the cluster.

A simple way to overcome this is to only inset the visibility samples, and treat them as if they were collocated with the irradiance samples at the edge of the cluster. This leads to small errors in visibility interpolation, but avoids overshoot.

### 6.3.7. Post-solution Refinement

It is often the case that some post-solution refinement is carried out after the radiosity solver has finished. This can range from Gouraud-shading the mesh to an image space "final gather" (**Section 2.5.3**) to resampling the visibility more accurately [Gibs95]. The motivation in all cases is that the eye is more sensitive to shadow and shading discontinuities than we can account for with linear radiosity-centred metrics, and thus it makes sense to refine such features in the solution before display. While an image-space final gather has its place, it imposes an expensive cost for each viewpoint generated. Instead, I have adapted some of

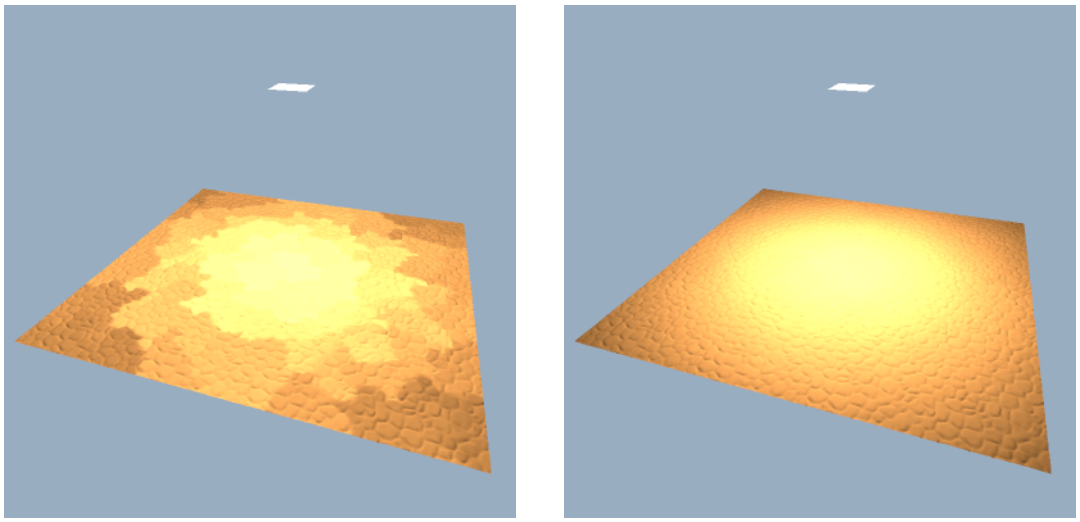

**Figure 84:** Vector interpolation. On the left, using a constant irradiance vector across each cluster leads to a blocky effect, reminiscent of the Haar basis. On the right, interpolating the irradiance vector across each cluster leads to a smoother result. The level of refinement for both pictures is the same.

Gibson's ideas on post-solution refinement for face cluster radiosity. Whereas Gibson re-evaluates visibility at the input polygon level, for the type of scenes considered in this thesis, this becomes prohibitively expensive.

There are a number of alternative post-processing solutions that can be applied.

- The minimal approach: we use the irradiance vector at each leaf cluster to colour all of its constituent faces. This can work well if visibility does not vary much, and surfaces are rough enough that vector irradiance discontinuities between clusters are masked.
- The irradiance-interpolation approach. We re-evaluate the vector irradiance at all corners of each leaf cluster, and use vector interpolation to colour its constituent faces, as above. This requires reevaluating all transport links at

the leaves, which can be expensive. A link pointing to a node relatively high up in the hierarchy must be re-evaluated at each leaf element it contains. Still, this is a reasonably robust approach.

- The irradiance-interpolation-everywhere approach. We perform vector irradiance interpolation at all face cluster nodes throughout the hierarchy. Irradiance vector samples at any given node are calculated by interpolating the parent's samples, and adding in resampled vector irradiance for the set of links pointing to the node, similar to the push phase of the familiar push/pull algorithm. This is less robust, but much faster.
- Finally, we can optionally subdivide all solution leaf elements a further $m$ levels before doing our final visibility-resampling pass. This achieves some of the benefits of Gibson's resampling-at-vertices approach, in that shadow resolution is markedly improved, without the cost of descending all the way to the input polygons.

Typically I use the first or third approach. In my experience the vector irradiance interpolation post-process typically takes 50% of the solution time, when it is performed throughout the hierarchy rather than just at the leaves. The extra solution quality makes this cost is almost always worth it.