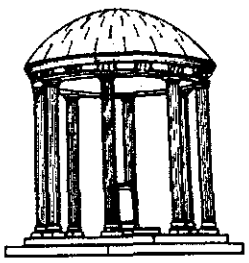
Taxonomy and Algorithms for Volume Rendering on Multicomputers

TR91-015

February, 1991

Ulrich Neumann



The University of North Carolina at Chapel Hill
Department of Computer Science
CB#3175, Sitterson Hall
Chapel Hill, NC 27599-3175

Taxonomy and Algorithms for Volume Rendering on Multicomputers

Ulrich Neumann

Department of Computer Science
University of North Carolina at Chapel Hill
Chapel Hill, NC 27599-3175, USA

Abstract:

The design space of parallel volume rendering algorithms is large and relatively uncharted. No adequate framework currently exists for classifying approaches so that distinctions between them are clearly understood based on the categories they fall into. We present an analytical decomposition of volume rendering tasks that yields a taxonomy of volume rendering algorithms for distributed memory multicomputers. In addition to the primary categories, secondary attributes are introduced to build a hierarchical classification system. Representative algorithms in each category are described and analyzed, giving insight into a whole range of approaches; some as yet untried.

1. Introduction:

Recently, in analyzing parallel volume rendering algorithms for a distributed memory machine, it became apparent that evaluation of proposed algorithms was hampered by the lack of descriptive names and categories for the various approaches. This experience motivated a search for a set of meaningful classification parameters which could serve to describe, as well as group, the algorithms.

A good classification scheme is useful in that it provides a framework for problem analysis, and makes clear the distinctions between different solutions. The major features of distinction should place an approach into a category with only similar approaches. Minor attributes should then serve to distinguish between items in a category, not move them to a new category. Thus, a hierarchy of attributes is developed with corresponding levels of importance in determining an algorithm's features. We present two major and two minor attributes useful for creating a hierarchy.

Volume rendering techniques can be roughly divided into three approaches: Surface Representation, Binary Classification, and Multi-valued Classification (MVC). MVC is useful, general, and potentially presents the fewest visually objectionable artifacts [Hoehne '90]; therefore we focus on this technique. Uniprocessor MVC volume rendering may be decompose into three distinct tasks which are: *shading*, *resampling*, and *compositing*. In multicomputer systems we must add a fourth task which is: *redistributing*. The main taxonomy categories are derived from the gamut of possibilities which arise from re-ordering these tasks. Sub-categories stem from attributes that are common to all algorithms, regardless of which main category they fall into.

Each main category contains algorithms which share major performance features. By describing the features of a category, the main features of all algorithms in that category are communicated. This simplifies tradeoff analysis since the number of major alternatives is limited. A representative algorithm from each category is presented and discussed to gain an appreciation for that categories features.

The taxonomy and algorithms are general. No specific architecture is targeted, but features required to achieve good performance for an algorithm are discussed. The following assumptions are made about the system:

1) It is a message passing distributed memory machine with identical nodes.

2) Any node may send or receive messages from any other node.

3) Routing cost may not be constant across all pair of nodes.

4) A host system is present that can initialize the system and pass user input to the nodes.

The algorithms scale well since there is no central or master node acting as synchronizer for the activities within a frame. This is frequently a bottleneck in systems with many nodes. Frame to frame synchronization is performed by the host.

2. Volume Rendering Tasks:

For our purposes, *shading* will refer to all the processing that is required to convert a data sample to a color and opacity. The output of this process is an R, G, B, alpha four-tuple. Examples of this process are amply described in the literature [Hoehne '89] [Levoy '88] [Drebin '88]. Levoy, for example, uses the difference between neighboring samples to compute a normal vector at each point. The intrinsic color and normal vector is used in a Phong lighting calculation [Phong '75] to yield the final shaded R, G, B values for each point.

In Levoy, shading is applied to the original input data array. Alternatively, one may reconstruct the continuous input function, resample it based on the current view direction, and shade the new samples [Upson '88]. Using either alternative, the shading of any sample point is a function of its neighbor's data. Global data is not required.

The *resampling* task refers to the reconstruction of a continuous-valued function from fixed samples, and the resampling of that function at new positions. Ray-casting with interpolation [Levoy '88] , and splatting [Westover '89] techniques are commonly used and appropriate for our algorithms. Reconstruction during ray casting requires multiple data values in the neighborhood of the new sample point. The size of the the reconstruction filter kernel determines how large a neighborhood is needed. For simplicity, we assume a maximum kernel size such that eight nearest neighbors to a new sample point are sufficient.

Given a three-dimensional array of shaded points aligned with the view plane, *compositing* [Porter, Duff '84] is done to compute their projection onto the view plane. The points must be ordered front-to-back, or back-to-front for this process to work correctly. Note that compositing is associative and therefore works correctly if arbitrary, contiguous sets of points are composited to a single value which is then composited with the remaining points. This fact is crucial to several algorithms.

The fourth task, *redistribution,* arises from the need to move data from node to node in a distributed memory multicomputer. Since computation and data is distributed, it is necessary at some point to merge or re-sort the data via a communications network. Any such non-local data access is part

of the redistribution task.

3. Major Attributes

Given the four tasks we have outlined, we may re-order them within certain constraints. Compositing must come after shading and resampling; of the 24 possible orderings, only 8 options remain and are shown below:

1. SHade -> ReSample -> COmposite -> ReDistribute => SH-RS-CO-RD
2. SHade -> ReSample -> ReDistribute -> COmposite => SH-RS-RD-CO
3. SHade -> ReDistribute -> ReSample -> COmposite => SH-RD-RS-CO
4. ReDistribute -> SHade -> ReSample -> COmposite => RD-SH-RS-CO

5. ReSample -> SHade -> COmposite -> ReDistribute => RS-SH-CO-RD
6. ReSample -> SHade -> ReDistribute -> COmposite => RS-SH-RD-CO
7. ReSample -> ReDistribute -> SHade -> COmposite => RS-RD-SH-CO
8. ReDistribute -> ReSample -> SHade -> COmposite => RD-RS-SH-CO

These eight re-orderings give rise to the taxonomy categories. There are two attributes that determine an algorithms major category:

i) Where redistribution occurs in the sequence of tasks.

ii) Whether we shade and then resample or vice-versa.

We group the algorithms by their sequence of shading and resampling. Due to their similarity and in the interest of brevity, we analyze only the first four approaches in detail.

4. Minor Attributes

There are minor attributes which an algorithms in any category may possess. Two such attributes are the initial data distribution scheme and whether tasks are performed in image or object-space. These attributes help to refine the specification of the algorithm.

4.1. Image and Object-Parallelism:

Image-space and *object-space* are often used to refer to parallelism schemes. While they serve to distinguish between two methods for parallelizing a single task, like ray casting for example, they do not adequately identify algorithms which are a set of tasks; nor do they account for communication functions. Parallelism in multicomputer systems is difficult to capture in a single term, since it exists on many levels. Traditional object-space parallelism at a high level may coexist with screen-space parallelism at a lower level. These terms are inadequate for describing parallelism methods, but are still useful for relating task organization for a single node.

MVC volume rendering tasks may be performed in object-space or image-space. Object-space tasks are performed per macro at each node, regardless of that data's projection onto the viewing plane. This is a *forward mapping* [Westover '90] of the data toward the screen projection. Image-space tasks are performed per screen element (pixel or ray); a *reverse mapping* of a pixel into the data at a node. The redistribution task generally does an image-space sort of the data, and usually forms a boundary between object-space and image-space tasks. It is important to note that the data is already partially sorted due to its organization into macros and distribution among nodes.

4.2. Data Distribution:

Since volume data sets may be very large, and we assume each node of the system has a fixed memory size which is insufficient to hold the entire data set; simple replication of the entire data set at each node is not practical. Data distribution is a parameter that classifies an algorithm. Data distribution may be static or dynamic, and of any *granularity*. The minimum redistribution quanta is the

granularity of the data distribution. *Coarse* granularity implies that a node may only send all or none of its data to another node. *Fine* granularity allows any number of arbitrarily connected samples to be transmitted. *Medium* granularity implies that data is transmitted in fixed-size chunks, each some fraction of the total data at a node. Note that fine granularity has very high addressing overhead; how does one efficiently address arbitrary regions of the 3-D sample grid? Coarse granularity is potentially wasteful in that if only one voxel is needed from a node, an entire node's data set must be transmitted. Medium granularity is a good compromise between addressing complexity and waste. We will refer to a data chunk of medium granularity as a *macro*. Typically, macros are spatially adjacent cubes of data addressed by a single coordinate (ie: the lowest x, y, z voxel coordinate in the macro). The optimal size and aspect ratio of a macro is a function of the communication network and the algorithm.

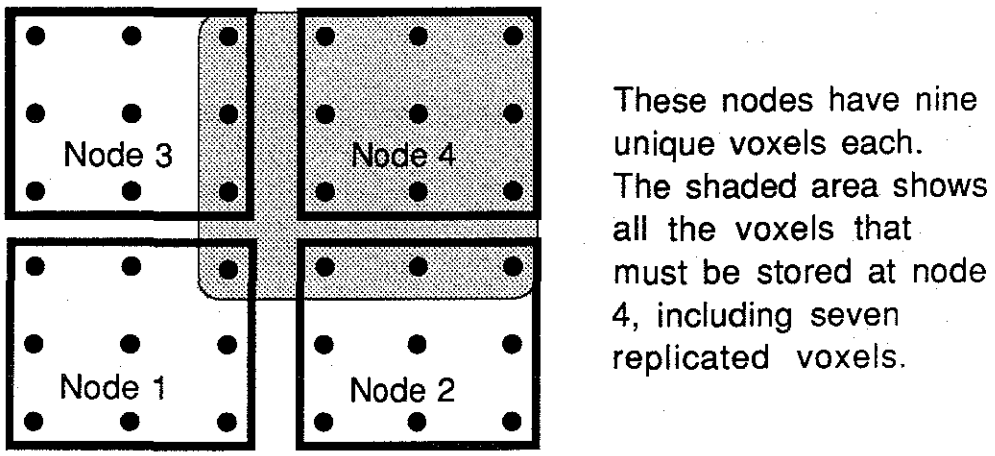
Peano addressing of the data set [Laurini '85] produces a high probability that spatially close points are also close in a linear address space. Peano addressed macros have contiguously addressed data points. This has good implications for cache coherence during ray casting, and simplifies addressing during macro redistribution.

Grouping of medium granularity macros is another issue. A node's macros may be a spatially adjacent subset of the data set, or some interleaved or random subset. If they are adjacent, we refer to the macro distribution as *grouped*, otherwise it is *interleaved* or *random*.

Resampling by ray casting requires data at neighboring points which gives rise to a redistribution problem at node data boundaries. Nodes must request neighboring voxels from other nodes by *fetching* the macros that contain these voxels. The cost of these fetches is a function of the macro size and grouping. An adjacent macro distribution may avoid these fetches by replicating the neighboring data at each node. Assume N nodes, V samples in the data set, and the macros at each node form a cube. Each node must store V/N unique samples. Each face of a node's local data has $(V/N)^{2/3}$ samples. To resample at an arbitrary position, there must be some node with the 8 nearest samples. By replication of 3 faces, 3 edges, and 1 corner of neighbor voxels at each node, arbitrary points can always be resampled at some node. This results in

$$D_{node} = V/N + 3(V/N)^{2/3} + 3(V/N)^{1/3} + 1 \qquad (1)$$

total voxels at each node (except at three faces of the the data set where no replication is needed). For a non-adjacent macro distribution, the replication must be done for each macro, resulting in a much higher total cost. Fig.1 illustrates the replication scheme in two dimensions; extension to three dimensions is straight forward.



(Fig. 1)

2D nodes with shading showing voxels included in upper right node after replication.

The table below shows the effect of changing the number of nodes for a 128^3 data set.

# of Nodes	Unique Voxels/Node	Replicated Voxels/Node	% Replication
8	262,144	12,481	4.76
64	32,768	3,169	9.67
512	4,096	817	19.9
4,096	512	217	42.4
32,768	64	61	95.3

The efficiency of memory usage clearly decreases as the number of nodes increases.

5.0. Algorithm analysis:

Analysis of the algorithms is done with respect to **memory usage, cpu task organization,** and **inter-node communication.** We take into account **data size, screen size, number of processing nodes, image quality,** and **data distribution options.** Dependence upon **data grouping** and **distribution of interesting voxels** will be discussed with respect to their impact upon load balancing. The ability to support **efficiencies** such as adaptive ray termination or **alpha cut-off, adaptive refinement,** and **octtrees** will also be considered [Levoy '90].

5.1. SHade -> ReSample -> COmposite -> ReDistribute:

Each node shades its local data and computes its projection onto the view plane in the first three steps. The re-sampling and local compositing may be done by either the ray-casting or splatting methods. Depending on the macro grouping, the projection at each node will yield one or more sub-images on the view plane. These sub-images are globally collected and composited in front-to-back or back-to-front order. The redistribution task is therefore moving 2D image data rather than 3D volume data.

The global compositing may be distributed or centralized to one (or more) screen servers. If centralized, the order of region arrival may be controlled by using tokens; or the server may store and sort the image regions as they arrive in arbitrary sequence, if it has sufficient memory. Centralized compositing is a serial operation and may thus be a poor match for machines with many nodes.

Distributed compositing would have each node partition its image region(s) into sub-regions which are then sent to the node(s) whose data is next in line for compositing. In effect, view rays are passed among nodes and travel through the data, wherever it may reside. If this sub-region propagation is also used to prioritize the scheduling of rays at each node, then efficiencies due to alpha cutoff may be realized. Nodes whose data has not yet been "reached" by any sub-region, should have a default ray schedule and hope that their work is not wasted. To minimize waste, adaptive refinement may be used

to compute only the coarsest level of refinement for each region; further refinement is performed or skipped depending on the alpha values of incoming sub-regions. Some alpha cutoff benefit will be realized regardless, since compositing local regions with opaque sub-regions need not be done.

Goldwasser, et al ['85] have implemented an algorithm of this category in custom hardware with a macro size of 64^3. It did not do any filtering when it resample data so no data was replicated. and used central compositing. Simple shading was done as a postprocess based on Z value only. Yazdy, et al ['90] used this approach, with central compositing, on an array of transputers.

Memory Usage: Equation (1) gives the average number of data samples at each node. These are shaded and resampled to produce again that many new shaded points. Additional memory is needed for the screen region, and with distributed compositing, a node must also provide memory for incoming regions which total at least the size of its own region.

CPU Tasks: Below are image-space and object-space examples. Image-space methods favor ray casting, while object-space methods favor splatting techniques.

Image-space

```
for each ray of the sub-image {
  for each ray step {
    if step neighbors not shaded
      shade step neighbors;
    resample;
    composite into local region;
  } }
global composite;
```

Object-space

```
for all voxels
  shade;
for all voxels in view order {
  resample;
  composite into local region;
}
global composite;
```

Communication: It is possible for each node to produce a full screen array of pixels as its sub-image. Slab shaped macros will cause this; and in this case each node must send and receive an entire screen

array of data. More cubic macro aspect ratios will produce smaller sub-images and lower bandwidth requirements.

Efficiencies: Alpha cut-off, adaptive refinement, and an octtree may be used within a node's local data. Distributed compositing offers additional efficiencies described previously in connection with ray scheduling.

Load Balance: Load balance is statically set by the data distribution since no 3D data samples move. It is dependent on the relative effects of the efficiency techniques at each node, and the view dependent compositing sequence. Levoy [89] has shown that there is high spatial coherence in the set of interesting voxels for some data sets. Therefore, a medium granularity random or interleaved macro grouping would improve the load balance, however this increases the memory requirement and the redistribution cost.

5.2. SHade -> ReSample -> ReDistribute -> COmposite:

This approach is essentially algorithm 1 with only global compositing. Each node is assigned one or more rectangular screen regions, called *tiles*, which it is responsible for. Tile assignment may be static or dynamic for each frame. The volume data is statically distributed evenly among nodes. Each node shades and resamples its local data in the first two steps. Resampling produces a view-aligned data array for each macro. The resampled macro arrays are redistributed to all nodes upon whose tiles they project. This task moves 3D volume data which may also be duplicated up to four times if a macro's projection overlays four tile corners. The redistributed macros are composited at each node, producing tile pixel arrays that are sent to a frame buffer.

A major complication of this approach is that resampling on the pixel coordinate grid does not yield a fixed number of sample points per macro. During compositing, each node could pre-compute the "shape" of a resampled macro array it requests, or "shape-parameters" could be sent with each macro. An alternative is to resample on the voxel coordinate grid so that each resampled macro has the same

size and shape. After compositing, the voxel coordinate pixels are interpolated across the actual screen pixels. This is only practical for orthographic projections, but we assume it as the method used for the remaining analysis.

Memory Usage: Equation (1) gives the average number of shaded voxels per node. Resampling adds approximately the same number of new shaded points. After redistribution, if resampled macros are clipped against tiles as they arrive, we have V/N additional shaded points that must be composited.

CPU Tasks: Shading and resampling each macro is done in object-space, while requesting and compositing macros is most naturally done in screen-space. Ideally they are independent processes.

Shading and resampling

```
for each macro {
  for each voxel of macro
    shade;
  for each new sample point
    resample RGBA;
}
```

Compositing

```
for each tile {
  for each ray in tile {
    for each sample along ray {
      if sample not local
        fetch macro of needed sample;
      composite sample into ray;
  } }
  interpolate rays across tile pixels;
}
```

Communication: A macro must be transmitted to all the tiles that need it. As the number of nodes is increased, tile size shrinks more rapidly then macro size; so macro replication will also increase.

Efficiencies: With redistribution between shading and compositing, alpha cut-off can not avert wasted shading and re-sampling. Adaptive refinement may be used for each tile, but saving only compositing.

By detecting completely transparent macros during shading, (a one-level octtree), we may save the effort of resampling, redistributing, and compositing. Levoy has shown that often a large fraction of the data set is transparent, so this savings is significant. Macros must be small to exploit this.

Load Balance: Without alpha cutoff, each node will shade all of its data; but due to transparent macros, there may be large variation between nodes in the effort of resampling, redistributing, and compositing. Dynamic tile assignments provide a solution at the expense of an assignment mechanism (ie: centralized tile-pool, or tile-tokens).

Multi-tasking is useful for keeping the nodes busy; shading and resampling should be interleaved with requesting macros and compositing received macros.

5.3. SHade -> ReDistribute -> ReSample -> COmposite:

As in algorithm 2, nodes are assigned tiles, and data is statically distributed, but by moving the redistribution task one step closer to the beginning, we gain several advantages over algorithm 2. Each node shades its data but retains it as a fixed size array. This allows perspective projections and arbitrary resampling points. With resampling and compositing done on a single node, splatting may be used in addition to ray-casting and multi-pass methods to resample the volume projected onto a tile. The redistribution task is moving 3D volume data which may be duplicated up to four times if a macro's projection overlays four tile corners.

Westover [89] has presented this approach in the form of multiple shading nodes and one centralized resampling and compositing node. Levoy [89] outlined this approach using the Pixel-Planes 5 heterogeneous architecture. The author and John Rhoades have implemented Levoy's approach on Pixel-Planes 5. Current performance is about 1.4 frame per second for 128^3 data sets.

Memory Usage: Macro fetches are performed by each node to acquire the data that projects onto its tile(s). During redistribution, if shaded macros are clipped against tiles as they arrive, we have approximately V/N shaded points that must be resampled and composited. The faces, edges, and

corner of each node's fetched data must be replicated, as previously described, so that rays cast along tile edges can be properly resampled. This will increase the memory requirement.

CPU Tasks: Shading each macro is done in object-space; resampling and compositing macros may be done in object or screen-space. Ideally they are independent processes. We illustrate the object-space splatting method below.

Shading

```
for each macro {
 for each voxel of macro {
  shade;
} }
```

Resampling and compositing

```
for each macro in view-order {
 if macro not local
  fetch macro;
 for each voxel of macro in view-order {
  resample;
  composite;
}}
```

Communication: A macro must be transmitted to all the tiles that need it. As the number of nodes is increased, tile size shrinks more rapidly then macro size; so macro replication will also increase.

Efficiencies: Alpha cut-off can not avert wasted shading. Adaptive refinement may be used for each tile, saving resampling and compositing. Detecting completely transparent macros during shading saves resampling, redistributing, and compositing effort.

Load Balance: Each node shades all of its data; but due to transparent macros, there may be a large variation in the effort of redistributing, resampling, and compositing. Dynamic tile assignments provide a solution at the expense of an assignment mechanism. Multi-tasking is useful for keeping the nodes busy; shading should be interleaved with requesting, resampling, and compositing macros.

5.4. ReDistribute -> SHade -> ReSample -> COmposite

Each node is assigned one or more screen tiles, then we redistribute the raw data based on macro projections onto the tiles. When macros arrive at a node, they are shaded, resampled, and composited to produce tile pixel arrays which are then sent to a frame buffer for display. This approach has the unique attribute that the raw data itself is moved during resampling. This permits a dynamic macro distribution. Macros are not bound to any single node over successive frames; they migrate to nodes as they are needed. For systems where adjacent tiles are assigned to adjacent nodes as far as routing is concerned, the average macro migration distance during redistribution is proportional to the change in view point. This allows some predictability of message traffic in the routing mechanism that may be exploited by some machines. If only shading parameters change from frame to frame, no redistribution is needed.

While tile assignments may be dynamic, efficiency of redistribution and machine routing distance dictates that consistent tile assignments be made from frame to frame whenever possible. Care must be taken when macros are clipped off screen; they must be kept at some node or they will be lost. We suggest two ways to handle this case: If no node requests a macro during a frame, it is retained by the current owner node even if it has no use for it itself. Original macros are tagged, copies may be deallocated and disappear if not used, but the tagged originals may only be passed on.

Since macros have no fixed home, how is a node to know where to sent a request for one? One solution, which depends on static tile assignments, is to use the previous frame's viewing transformation to determine where the macro was last frame. An alternative solution assigns tiles to preferred nodes, which are globally known, until a node has consumed all of its preferred tiles. Then as further tiles are assigned to the finished but non-preferred nodes, a message is sent to the preferred node informing it of the tile reassignment. When, in the next frame, a macro request arrives at the preferred node, and the macro is not found locally, a tile reassignment list may be used to forward the request. Load balancing and macro tracking require careful consideration, but are not mutually exclusive.

Memory Usage: There will be data replication after redistribution due to the need for replication imposed by resampling. Unlike in algorithms 1, 2, and 3, the memory requirement is not constant or equal at each node; it varies with view direction and scaling. The worst case occurs when the object is scaled down to appear entirely in one tile. The trivial solution is to limit the view transformation to some acceptable bounds.

CPU Tasks: As in algorithm 1, shading, resampling and compositing macros may be done in object or screen-space. Image-space ray casting and object-space splatting are shown below:

Image space

```
for each tile {
  for each ray through tile {
    for each ray step {
      if step neighbors not local
        fetch macro(s);
      if step neighbors not shaded
        shade step neighbors;
      resample;
      composite;
} } }
```

Object space

```
for each tile {
  fetch all macros that project onto tile;
  for each macro in view order {
    for each voxel in view order {
      shade;
      resample;
      composite;
} } }
```

Communication: As mentioned above, the redistribution cost is variable. If we limit the change in view point between successive frames, we may likewise limit the redistribution cost in terms of routing distance. The acceptability of this, from a user standpoint, is a function of frame rate. As we tighten the transformation limit, we increase the frame rate, thus the real-time rate of rotation is kept about the same, but rotation is smoother due to the higher frame rate.

Efficiencies: Alpha cut-off is effective at averting wasted shading, resampling, and compositing. Adaptive refinement may be performed for each tile saving shading, resampling and compositing. A local octtree may be constructed to save the effort of resampling and compositing, but the octtree construction requires shading every voxel which is what alpha cutoff and adaptive refinement could prevent. The architecture will likely influence the mix of efficiencies to use.

Load Balance: As mentioned above, dynamic tile assignments with preferred tile-node pairing, and constrained view positions, are necessary for effective load balancing.

5.5. Hybrid Extensions:

If we generalize the notion of tiles to allow each screen tile to be the composite of multiple stacked tiles, then we may add an additional composite phase to the end of algorithms 2, 3, and 4. This technique allows machines with large numbers of nodes to maintain larger tiles and therefore minimize the replication of volume data during redistribution (as given by equation (2)).

6. Results:

We have presented a hierarchical taxonomy of volume rendering algorithms for multicomputers based on a decomposition of tasks. Four algorithms, representative of the eight major categories in the taxonomy, were described. Two of the four were presented for the first time. Several actual implementations were classified into their appropriate categories. Secondary attributes were detailed that succeed in further strengthening the descriptive power of the taxonomy. This has created a frame work for further algorithm development and system exploration.

7. Acknowledgements:

The author wishes to thank Marc Levoy for many conversations that were instrumental in starting down this road; and also for his careful review and comments on an early draft of this work. Henry Fuchs's advice and encouragement were appreciated. Many thanks to Terry Yoo for his editorial inspirations;