COLLABORATIVE VISUALIZATION ENVIRONMENT USING P2P TECHNOLOGY AND ELLIPSOIDAL MESH PARTITIONING

GANESAN SUBRAMANIAM

(B. Eng. (Hons.), NUS)

A THESIS SUBMITTED

FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

NATIONAL UNIVERSITY OF SINGAPORE

2011

ACKNOWLEDGMENTS

To begin at the beginning, I am deeply indebted to my mother, Kalaymai. Where I am today is in no small part due to her encouragement and support.

I also owe a lot to my supervisor, Professor Kenneth Ong. His assistance throughout my years of study here has been invaluable. He was willingly subjected to several early, very rough, versions of this dissertation, and his many comments and suggestions have undoubtedly improved things.

I would also like to thank all those who have contributed to this work by providing both data and results. Saravanan Silvarajoo provided the test 3D models and took time out of his busy schedule to help me vet through this thesis. Both Bernard Yeo and Kevin Parama Veragoo of Institute of High Performance Computing too helped me out in ironing out the rough edges of this dissertation.

And finally, I would like to thank my wife, Megala who has helped me see this thesis through to a completion with her encouragement and support.

TABLE OF CONTENTS

ABSTRACT

A common technique to perform distributed (or parallel) rendering of a single frame is to break up a 3D scene and share the rendering load across multiple machines (called the rendering agents). The rendered sub-images from each machine are then composited on a single machine (called the compositor) and displayed on the screen (or saved to a file). The end result is an overall improvement in per frame render times for large and complex 3D models. However, this technique suffers from 2 major performance bottlenecks. Firstly, communication between the rendering agents and the compositor is heavy since depth information is also transferred with the rendered sub-images for every frame. Secondly, composition of all the sub-images from every rendering agent is an expensive process as every pixel has to be subjected to depth comparison.

In this thesis, we propose a mesh partitioning algorithm (called Ellipsoidal Mesh Partition) and a mesh distribution algorithm (called Context Aware Mesh Partition) that eliminate the need for depth information for the compositing of the rendered sub-images. This reduces the compositing complexity. The key to both algorithms, is to break up a 3D mesh based on its unique features into smaller sub-meshes. If each sub-mesh is rendered by exactly one unique rendering agent, the composition of the rendered results will be equivalent to "piecing together a jigsaw puzzle". In other words, the compositing cost using our distributed rendering algorithm is reduced tremendously. Despite a minor (negligible) degradation in the final composited image, our results show an overall 40% performance improvement. Thus, we recommend this solution for use in distributed polygonal rendering applications and systems.

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

1 | INTRODUCTION

Physically based Rendering is the process of generating a 2D image from the abstract description of a 3D scene. The process of constructing a 2D image requires several phases such as modeling, setting materials and textures, placing the virtual light sources and rendering. Rendering algorithms take a definition of geometry, materials, textures, light sources and virtual camera as input and produce an image (or a sequence of images in the case of animations) as output. High-quality photorealistic rendering of complex scenes is one of the key goals of computer graphics. Unfortunately, this process is computationally intensive and requires a lot of time to be done when the rendering process requires graphics shaders such as Global Illumination[33, 70, 71]. The problem further escalates when multiple of these shaders are used in a single rendering. Thus, depending on the rendering method and the scene characteristics, the generation of a single high quality image may take several hours (or even days). Therefore, the rendering phase is often considered to be a bottleneck in photorealistic projects in which one image may need some hours of rendering in a modern workstation.

This thesis introduces our distributed rendering solution, the Ellipsoidal Collaborative Visualization Environment (ECOVE), to address the problems stated in following sections. ECOVE provides an implementation of distributed storage and computing to the problem of large-scale visualization. It shows how to distribute 3D mesh model for effective visualization throughout the devices in the network and form a communication framework to assemble the model at the required devices. In an effort to reduce the reliance on the client-server communication model, ECOVE employs P2P to discover computing resources for rendering of portions of a 3D mesh model or scene, and manage the rendering process. The following sections provide some background material for the thesis contributions described in Chapters 2 and 3.

1.1 | DISTRIBUTED RENDERING

A common method to reduce rendering time is to reduce the scene complexity but this might compromise the quality of the final animation scene. Therefore, animation studios often have to find a balance between the quality of the scene and the production time. In recent years, animation studios were able to render highly complex 3D scenes using a cluster of high performance networked computers, also known as a render farm. The rendering task is distributed across all the computers in the network and thus, this method of rendering is called Distributed Rendering.

The basic task in polygon rendering is to calculate the effect of each primitive on each pixel and can be viewed as a problem of sorting primitives from their world coordinates to the screen [28]. To date, most distributed renderers (e.g., [66]) have been software-only (with the exception of specialized systems such as the PixelFlow Machine [58]) and can be categorized into three classes, sort-first, sort-middle, and sort-last, depending on whether the sorting process takes place during the geometric transformation phase, between the geometric transformation phase and the rasterization phase, or after the rasterization phase [59].

In sort-first [16] (see Figure 1.1) and sort-middle [66], the geometric transformation and rasterization of a polygon may be performed by different nodes, depending on the specific work assignment of each frame, possibly requiring the redistribution of a significant number of primitives. This redistribution is a fundamental problem in our targeted environment. First, this requires either recomputing the geometric transformation of primitives that must be distributed or accessing information that may be hidden inside a hardware graphics pipeline. Second, the bandwidth requirement for these two classes directly depends on the complexity of the scene that is being rendered, violating one of our basic goals. Finally, because the mapping of primitives to screen coordinates change depending on the viewpoint, the amount of data that must be transmitted can change very unpredictably from frame to frame. For example, in an assessment of the practicality of sort-first, Mueller gives communication measurements for three different scenes where the system must redistribute anywhere from 5% to 100% of the scene primitives [16]. It seems extremely difficult to achieve predictable real-time response in the face of such large variability in bandwidth requirement.

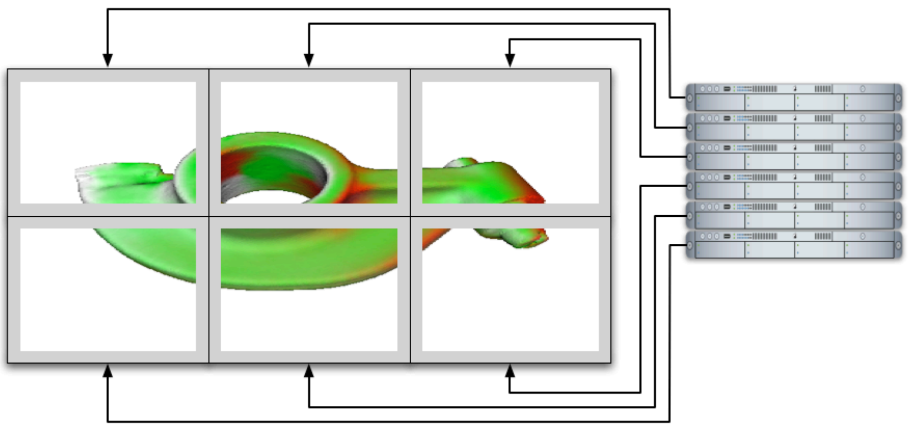


Figure 1.1: Shows the concept of sort-first distributed rendering of a scene by splitting the frame into tiles and rendering each tile in parallel on separate rendering nodes.
Image courtesy of http://www.equalizergraphics.com/.

Sort-last [59] (see Figure 1.2) corresponds to a data partitioning, where each node is assigned a subset of the polygons in the scene, without any restrictions on the position of the polygons. For each frame (once every node has rendered the image corresponding to its assigned polygons), the pixels must be sorted, typically using Z-buffering [36]. While compatible with our environment, sort-last is less than ideal for two reasons. First, because renderers generate pixels without regard to visibility ordering across different nodes, they must send the depth information (Z-buffer) along with the pixel values for compositing of the final image. This approximately doubles the required bandwidth. Second, primitives are typically assigned to renderers without regard to where they map to in screen space. In fact, sort-last renderers often distribute primitives randomly to load balance. This means that each renderer must typically send the entire image for each frame. This limits the scalability of sort-last as the required bandwidth is directly proportional to the number of nodes ($P \times$ *image-size*) instead of to the image size.

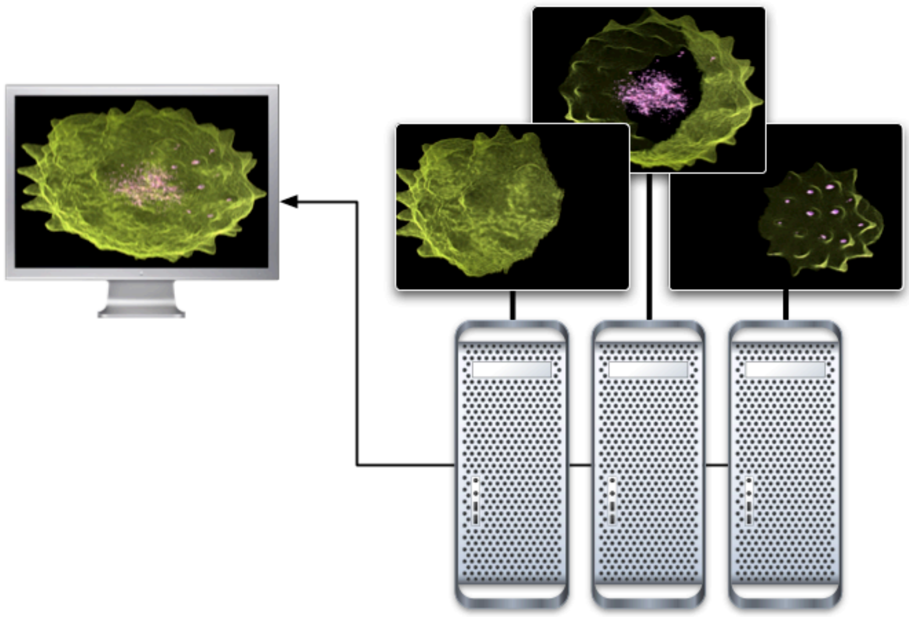


Figure 1.2: Shows the concept of sort-last distributed rendering of a scene by partitioning the scene into groups of polygons and rendering each group in parallel on separate rendering nodes.
Image courtesy of http://www.equalizergraphics.com/.

On one hand, a large-scale distributed 3D rendering environment needs to make sure that data replication is at a minimum across the network (as data transmission and storage can be expensive). On the other hand, every rendering node needs to have knowledge of the structural information of the 3D mesh model. This is especially important to graphics shaders (such as Global Illumination) that are needed in photorealistic rendering. Most often, these shaders only need an approximate description of the structure of the geometry. While sort-last ensures the former criterion, it does not however, solve the latter problem. By distributing polygons randomly, structural information of the geometry is not known to the rendering nodes. Thus, it is not possible to implement graphic shaders effectively to perform distributed rendering with current implementation of sort-last algorithm.

1.2 | MESH SIMPLIFICATION

In [29], we presented a method for simplification of arbitrary 3D meshes. *This technique has been adopted for mesh partitioning as discussed in Chapter 2 of this thesis.* The work presented in this paper was motivated in two ways. We first developed a prototype system to perform distributed rendering (see Chapter 3) using Sort-Last technique [59]. The most difficult part of the task was to reduce the transmission times of 3D mesh models to the remote rendering agents. This eventually led us to exploring techniques to simplify the mesh before transmitting it to the rendering agents while preserving key geometric features as much as possible. Based on the lessons learnt in this exercise, we built a novel technique to quantize the orientations of the polygons and identify regions of connected polygons that are similarly oriented to each other. The curvature of the resulting regions is considerably lesser than the entire mesh. As such, *flattening* each region produces a simplified mesh while preserving key features of the mesh.

The technique is described to contain three stages. Firstly, a code-book that contains the unique directional vectors is generated using our Ellipsoidal Schema. Secondly, the polygons of the mesh are grouped into patches: based on the code-vectors and the locality information of the polygons. And the resulting patch is approximately a flat plane with its corresponding code-vector as its normal. In the last stage, our mesh simplification technique re-triangulates all patches, in which the algorithm only considers the vertices on the boundaries of the corresponding patches.

As discussed in Section 2.2.1.3, this technique will help to produce pseudo polygons (or fragment masks) that will be used by every rendering agent to mask out the regions of overlap. Thus producing sub-images that do not overlap with other sub-images rendered by other rendering agents.

1.3 | Communication Framework

A pure P2P system does not require the existence of any centralized servers or resources to operate. In reality however, the data would usually reside on a data server. With these considerations in mind, ECOVE is designed to be a system where peers will only contact the data server to download a portion of the model (and the simplified mesh fragments). Once the peers, each has a portion of the scene, they can redistribute the dispersed model amongst themselves. In other words, peers of ECOVE need not rely on any centralized servers to distribute and manage a 3D model during the rendering phase. This is achieved by different peers offering to manage different portions of the model (i.e. sub-meshes) and thus leading to the concept of a distributed workload management. This means that the current day practices of using a centralized job queue to which all the clients would subscribe to, does not fit the ECOVE model too well. Instead, the workload (i.e. the sub-meshes) is dynamically distributed across the peers of the system and each peer will advertise the sub-meshes that they are managing for other peers to access.

ECOVE includes the Ellipsoidal Mesh Partitioning (EMP) technique to partition a 3D mesh and the Context-Aware Mesh Partitioning (CAMP) technique to manage the workload distribution. Unlike traditional mesh partitioning techniques where the scene is chopped up into equal number of polygons, EMP subdivides the mesh based on orientation and locality properties of the polygons of a scene. CAMP, then, distributes the partitioned mesh to the available rendering nodes in the system. As discussed in Chapter 3, this latter technique is coupled with P2P technology to collaboratively manage the rendering process.

1.4 | CONTRIBUTIONS

The contributions of this thesis are organized into Chapters 2 and 3, and they are namely:

Ellipsoidal Mesh Partitioning

1. It introduces the geometric model called Gaea-Sphere whose face normals are deterministic. The resolution of a Gaea-Sphere can be controlled parametrically. See Chapter 2 for more details.

2. It is a technique to partition a 3D mesh model based on orientation and locality of the polygons. It uses two schemas that are based on the polygon normals of a Gaea-Sphere, to sample the orientations of a 3D mesh model. The polygons are then grouped based on their connectivity to their neighbors and their orientation. The resulting groups of polygons are a set of connected polygons with similar orientations.

3. It provides both simplified and extended techniques for controlling the distribution of polygons during the fragmentation of the mesh. The end result is for the polygons to be distributed as evenly as possible for distributed rendering purposes.

4. It presents a method to automatically refine the fragmentation results to remove fragments that have low concentrations of polygons. Based on a set of user-specified criteria, the method reclassifies the polygons of identified fragments to their neighboring fragments.

Collaborative Rendering

5. It provides a technique to eliminate Z-buffer during transmission of the rendered sub-images from each rendering agent and during the composition of the sub-images. It has been demonstrated that this technique provides substantial performance gain.

6. Using our Context Aware Mesh Partitioning algorithm, it shows how a 3D mesh model can be partitioned and distributed amongst a network of peers. The partitioning of the mesh can be controlled to even out the number of polygons rendered on each peer.

7. It demonstrates how P2P technologies can be used to discover and setup a network of devices to render a 3D mesh model collaboratively. The task of monitoring for peers that drop out of the network is distributed amongst the peers themselves.

1.5 | Organization of Thesis

The rest of this thesis is organized into three main chapters. Our Ellipsoidal Mesh Partitioning technique is discussed in Chapter 2. Here some of our key contributions to the field of Mesh Partitioning are highlighted and how they can be used for Collaborative Visualization. Chapter 3 extends these contributions by describing in detail the CAMP algorithm of the ECOVE system. The chapter also shows the parametrization of the rendering pipeline and calculates the theoretical performance gains for ECOVE over classical methods. Results of the experiments conducted on an implementation of the ECOVE system are discussed in Chapter 4, "Implementation and Results". Finally in Chapter 5, we conclude the thesis by suggesting possible future work to be done based on this thesis.

CHAPTER 2

2 | ELLIPSOIDAL MESH PARTITIONING

Mesh partitioning is motivated by the fact that domain decomposition provides a natural route to parallelism. An automatic mesh decomposer should distribute the mesh across the individual processors so that the computational load is evenly balanced and the amount of inter-processor communication is minimized. However, numerical experience [74] has shown that several other issues, such as the sub-domain shape and connectivity, in addition to load balancing and communication costs, need to be addressed. A considerable attention [6, 7, 13, 15, 26, 32, 48, 74] has been focused on developing suitable techniques to solve the mesh partitioning problem and several powerful methods have been devised. The greedy algorithm [15, 52] is based on a successive expansion of a sub-domain, initially formed by one appropriately chosen element, until it comprises a sufficiently large number of elements. The expansion is usually driven by neighborhood search schemes using the depth-first or breadth-first search. The basic disadvantage of this very fast technique resides in the fact that the final partitioning is often very far from the "optimal" one. However, the speed makes this technique very suitable for an initial decomposition subjected to further optimization, based on, for example the relative gain concept [14] or simulated annealing [62]. The recursive bisection methods [74, 62] utilize the spatial distribution of a mesh. While the coordinate recursive bisection (Cartesian, polar, or spherical) exploits only the dimensional properties of the mesh with respect to a given coordinate system, the inertial recursive bisection accounts for principal inertial properties of the mesh which are invariant with respect to the coordinate system. The spectral recursive bisection [32, 62] is based on the finding that the second largest eigenvalue of the Laplacian matrix of an undirected graph associated with a mesh provides a good measure of the connectivity of the mesh and that the components of the corresponding eigenvector can be conveniently used for the mesh bisection. Although this approach provides decomposition of a high quality, computational complexity makes its use problematic when large meshes are under consideration. This deficiency was partially eliminated by a multilevel implementation of this technique [62].

However, we realized that the partitioned meshes of these techniques will not be ideal for the distributed rendering environment discussed in this thesis. A rendering cycle can consist of several graphical shaders such as Ambient Occlusion [47] and Mesh Deformation [43, 45]. These shaders require some or partial knowledge of the polygons in the other partitions. If each partition is distributed to a different peer, the rendering cycle will be burdened with the amount of communication that needs to take place between peers for each render of a frame. Thus, it is necessary for each peer to have all the necessary information about a scene to render a specific partition. Once the rendering cycle begins, there should be minimal communication between the peers to achieve a final rendered image. However, the partitioned meshes from the techniques discussed in the previous paragraphs, would not satisfy this criterion as the mesh partitions do not provide any information about the rest of the mesh. Thus, to address this problem for distributed rendering, we have identified the following considerations that our mesh partitioning technique should exhibit:

1. **View dependency**

   A view dependent partition consists of polygons orientated towards a general direction. This thus allows a partition to be approximated as a simple, flat plane. When a partition is distributed to a peer it will also receive the approximated representations of the other partitions. This partial information, albeit an approximated one, is useful for several graphical shaders.

2. **Non-overlapping partitions**

   Taking into consideration the cost of storage and distribution of the polygons across the network, each partition should observe distinct separation of regions. That is, the partitions should not overlap with each other. By eliminating polygon redundancy, each polygon is ensured to be rendered only once during a rendering cycle.

3. **Geometric feature preservation**

   For pre-processing tasks such as Mesh Simplification [31, 38], the features of a mesh need to be preserved as much as possible while the polygon count of the mesh is reduced. To adhere to this criterion, the mesh should be partitioned along major geometric features. This way, the final re-constructed 3D mesh will not lose its general geometric structure.

Adhering to these considerations, we present in this thesis our Ellipsoidal Mesh Partitioning technique. As depicted in Figure 2.1, the technique partitions a mesh into sub-meshes via a series of processes called Code-Book Generation (see Section 2.2 on Code-Book Generation) and Polygon Grouping (see Section 2.1 on Polygon Grouping). The former process generates a code-book of orientations that will be used to quantize the 3D mesh's polygon orientations in the Polygon Grouping process. The latter process further breaks down partitions by grouping polygons that are in the same partition and are adjacent to each other.

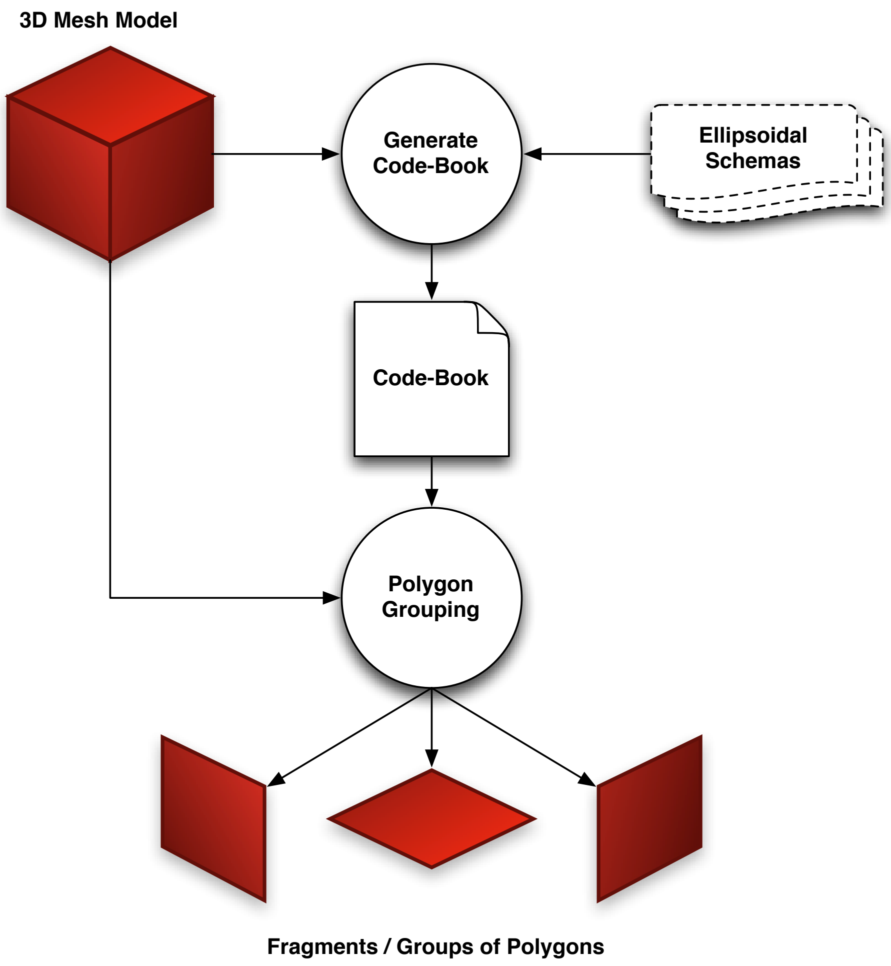


Figure 2.1: Shows the overall process of the Ellipsoidal Mesh Partitioning technique. The process starts from the top with a 3D mesh as input to the Code-book Generation process. Based on the intrinsic properties of the 3D mesh and the Ellipsoidal Schema, a code-book is generated and passed to the Polygon Grouping process. This process then breaks up the input 3D data into sub-meshes.

Note that, although the Code-Book Generation process is performed before the Polygon Grouping process, the next section is dedicated to discussing the latter process as it will build the base for understanding what is required of the former process.

2.1 | Polygon Grouping

The main aim of the Polygon Grouping process is to partition a mesh into groups of polygons with similar attributes. Adhering to the considerations outlined in the previous section, this thesis will focus on the Orientation and Locality attributes of a polygon. By grouping polygons with similar orientations (i.e. polygons facing in the same general direction), we can satisfy the View-Dependency consideration. By considering the locality of each polygon, these groups can be broken down further by clustering polygons that have connected edges or vertices. The resulting groups of polygons will be non-overlapping sub-meshes[1] that are partitioned along critical[2] geometric features and each will lie on a single flat plane (see Figure 2.2).

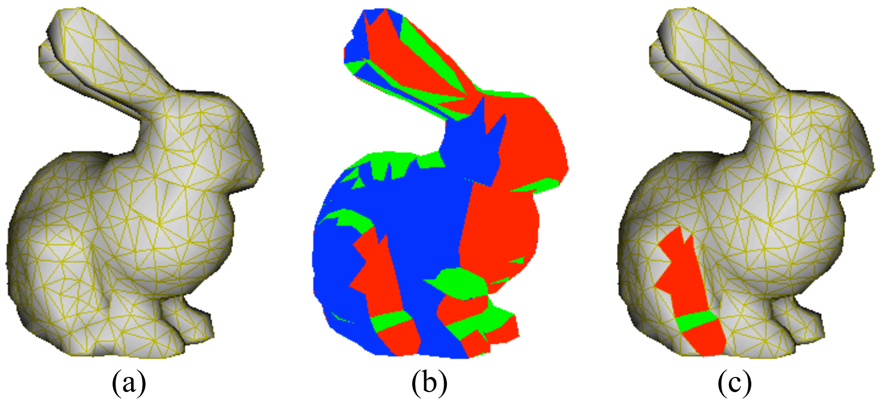


Figure 2.2: Shows the 3D mesh model of a bunny as it goes through the mesh partitioning process, Polygon Grouping. In (b), the mesh has been partitioned and each partition is represented in different colors. (c) highlights 3 sub-meshes (2 red regions and 1 green region), each of which lie on a single flat plane.

[1] The difference between a sub-mesh and a group of polygons is that a sub-mesh must contain polygons that share at least one edge with another polygon from the same group. Whereas a group of polygons need not adhere to this criteria.

[2] The criticalness of a geometric feature depends on the curvature of the geometry.

The following two sub-sections will cover the techniques of partitioning a mesh using the Orientation (see Section 2.1.1) and Locality (see Section 2.1.2) attributes respectively. In the sub-section 2.1.1, we will employ the concept of Voronoi Diagrams to explain how the Orientation attribute will work as perceived.

2.1.1 | GROUPING USING THE ORIENTATION ATTRIBUTE

Without loss of generality, suppose that a set of sites is given in the Euclidean plane (see Figure 2.3). The number of sites is assumed to be two or more. Given this site set, the problem is to assign every point in the plane to the closest member in the site set. As a result, the set of points assigned to each member in the site set forms its region. These regions are mutually exclusive (i.e. non-overlapping) and adjacent regions have borders that are equidistant from the two sites corresponding to the two regions. This collection of regions forms a tessellation. This tessellation is known as a Voronoi Diagram, and the regions constituting the Voronoi Diagram are called Voronoi Regions.

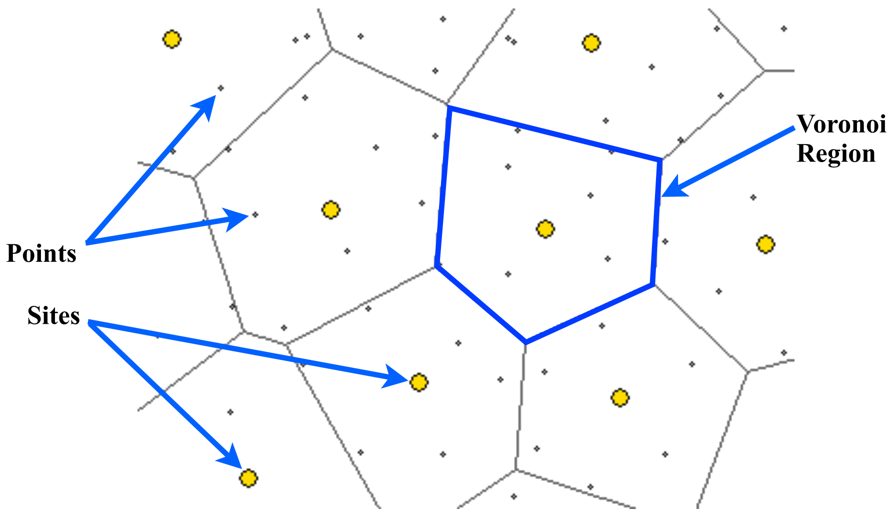


Figure 2.3: A Voronoi diagram divides a set of points (black dots) into

a region around each site (yellow dots) such that the borders of adjacent regions are equidistance from their corresponding sites.

Suppose each site is a unit vector (in 3-dimensional space) that radiates from the center of a unit sphere. Then the problem statement can be rewritten as assigning every point on the sphere's surface to the closest member in the site set. Points on the sphere have unit normal vectors that radiate from the center of the sphere (just like each unit vectors of the site set). Since these normals are orientated about the center of sphere, they represent the orientation of their respective points. Hence, the problem statement can be re-interpreted as assigning every point on the sphere's surface to a member in the site set that has the closest orientation. The metric to represent the distance between a point's orientation (or the unit normal vector) and a site vector can be stated as the dot product between the two vectors (see Equation 2.1).

$$d(P_i, S_j) = \left| \, cos^{-1}\left(\frac{N_i \cdot S_j}{\|Ni\| \cdot \|S_j\|}\right) : S_i \in S \, \right|, \qquad (eq\ 2.1)$$

where N_i is the unit normal vector of a point P_i on a unit sphere P_{sphere} and S_j is a unit vector in the site set S.

Equation 2.1 calculates the angle of separation between a point's normal vector and site vector. Hence, a point can be assigned to a site vector that yields the smallest angle of separation using Equation 2.1.

Consequently, we can define a Voronoi Region for the site vector as the region containing a set of points that yields the smallest angle of separation to that site vector.

$$VR_j = \cup P_j : \textit{for all } P_i \in P_{sphere} \quad \textit{and},$$

$$d_{min}(P_i, S_j) \textit{ for all } S_j \in S \qquad (eq\ 2.2)$$

where VR_j represents the Voronoi region (of the site vector S_j) containing all points (lying on the unit sphere P_{sphere}) that form the smallest angle of separation (d_{min}) with the site vector S_j.

A Voronoi Region, based on Equation 2.2, is a group of points that are orientated in the same direction as the site vector for the corresponding region (see Figure 2.4).

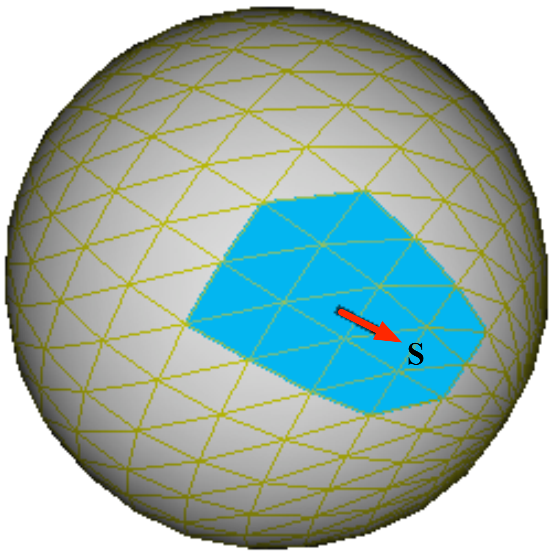


Figure 2.4: Shows a group of points on a unit sphere that are grouped to a Voronoi Region corresponding to the site vector S_j. That is, there exists a site vector S that can represent the general direction of the highlighted points in the figure.

For the purposes of this thesis, we have extended the problem space from a unit sphere to a complex 3D mesh (see Figure 2.5). If the 3D mesh is positioned at the same origin as the unit sphere (of the previous problem space), then we can represent each polygon as a point (corresponding to the center of the polygon) whose unit normal vector radiates from the origin. Thus, Equation 2.1 and 2.2 will still hold true for the set of polygons in a 3D mesh.

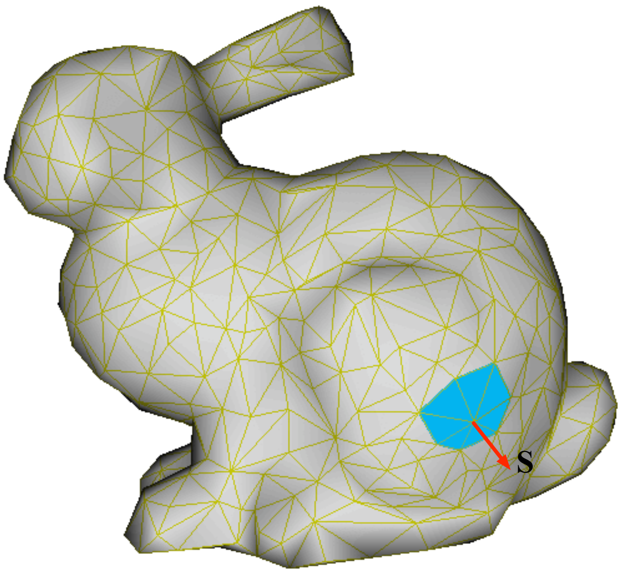


Figure 2.5: Shows a 3D mesh model positioned at the same origin as the unit sphere in Figure 2.4. And just like the unit sphere, the highlighted points are grouped to a Voronoi region that corresponds to the vector *S*.

Each polygon of the 3D mesh is assigned exactly to only one Voronoi Region. The resulting Voronoi Diagram is a set of Voronoi Regions, each containing a set of polygons that are orientated in the direction of their corresponding site vectors. However, at this stage, the Voronoi Regions sought to cluster the polygons based on their orientations only. Thus, in Euclidean space, the polygons of the same Voronoi Region can be spatially disjoint as shown in Figure 2.5. The sub-section 2.1.2 will discuss on why this discontinuity amongst the polygons in a group is not desirable and how it can be overcome.

Note that the selection of the site vectors is crucial for this stage of the Polygon Grouping process as it will determine the distribution of the polygons across the

Voronoi Diagram and the number of Voronoi Regions created. This will be covered in the Section 2.2: Code-Book Generation.

2.1.2 | Grouping using the Locality Attribute

In the previous sub-section, we have seen how polygons of a 3D mesh can be grouped based on the orientation attribute of a 3D polygon. The resulting groups will contain polygons that have similar orientations. However, these polygons can be spatially disjoint and may not lie on a single flat plane (see Figure 2.6). Thus, these groups will be further broken down by separating the disjoint groups of polygons. *The resulting sub-groups will be called "Fragments" for discussion purposes of this thesis.*

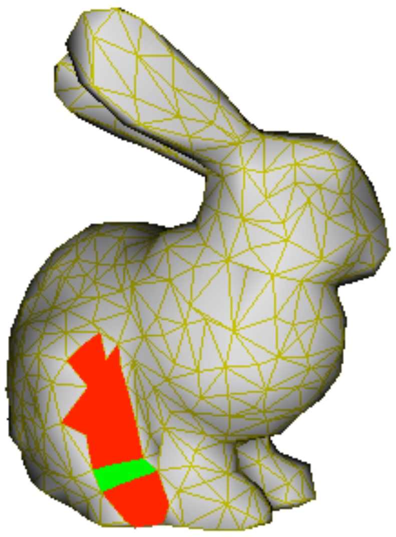


Figure 2.6: Shows 3 fragments (2 red regions and 1 green region) of a 3D mesh model. The red regions have the similar orientation but are disjoint. Thus, a fragment will be formed to represent each of the fragments.

By observation, the group of polygons presented in Figure 2.6 can be broken down into 3 smaller groups (or fragments). Mathematically it is possible for us to employ

K-Means Clustering [3, 5, 12] to form these groups. However, an initial number of groups (i.e. the K in the K-Means Clustering) needs to be provided for the clustering to begin. Since there is no way to know the number of groups initially, we abandoned the idea of using this technique, and looked at the connectivity information of the polygons and their vertices instead. Thus, we begin by defining a Fragment.

> *A fragment is a set of polygons, made up of vertices such that every vertex is connected to another directly or indirectly.*

In other words, every vertex in a fragment connects to another vertex in the same fragment by tracing along the edges of the polygons of the fragment. Given the following set of vertices, the path of connectivity between the vertices can be expressed as $T_{i,k} = \{ V_i, V_j, V_k \}$ where $T_{i,k}$ denotes the path from vertex V_i to V_k through V_j. In this arrangement, V_i does not have a direct link to V_k. However, V_j is linked to both V_i and V_k. Thus, V_i can reach V_k only via V_j. If the cost of connectivity of V_i to V_j is denoted as 1 unit, the cost of the path $T_{i,k}$ can be expressed as:

$$cost(T_{i,k}) = \frac{cost(T_{i,j}) + cost(T_{j,k})}{2} \qquad (eq\, 2.3a)$$

where $T_{i,j}$ denotes the path between V_i and V_j, and $T_{j,k}$ denotes the path between V_j and V_k. A generalization of equation 2.3a is given in equation 2.3b.

$$cost(T_{n,n+m}) = \frac{\sum_{i=n}^{m} cost(T_{i,i+1})}{m} \qquad (eq\, 2.3b)$$

where m is the number of paths needed to reach vertex V_{n+m} from vertex V_n. To normalize the cost information, the sum of the costs of all the paths taken is divided by the number of paths taken. In other words, the cost of connectivity does not depend on the number of paths taken and evaluates to 1 for vertices within a fragment. Since vertices from different fragments are never connected, m is 0 and thus, the cost of connectivity across fragments will always be ∞.

Based on this property, we can form fragments by iteratively looping through all the polygons (and their vertices) of a group (the result of the previous sub-section). In each iteration, a polygon is added to a fragment if any one of its vertices has a cost of 1 when connecting to the other vertices in the fragment.

At the end of this stage of the Polygon Grouping process, we will have fragments with polygons that are orientated towards a general direction. Thus, a fragment can be represented as a single flat plane. For some sub-processes in the rendering pipeline (where accuracy of the geometry is not crucial), a fragment can be approximated to be an n-sided polygon, orientated towards the normal of the fragment.

2.1.3 | POLYGON GROUPING ALGORITHM

Polygon grouping is a process that groups connected polygons with similar orientations together. While this process is presented in two distinct sections (Section 2.1.1 and 2.1.2) and thus suggesting two separate sub-processes, they can however, be implemented as a single process. The following shows the code listing for the polygon grouping algorithm for a given set of M polygons p and a set of S voronoi site vectors μ.

Algorithm 2.1: Polygon Grouping

```
1    begin initialize $M, p_1, p_2, ..., p_M, S, \mu_1, \mu_2, ..., \mu_S$
2        for every $p_i$ that is not assigned to a fragment
3            create a new fragment $f_y$
4            add $p_i$ to fragment $f_y$
5            classify $p_i$ to site vector $\mu_j$
6            push neighbors of $p_i$ onto queue $Q$
7            pop next $np_k$ polygon from $Q$
8                classify $np_k$ to site vector $\mu_w$
9                if $\mu_w$ is the same as $\mu_j$ then
10                   add $np_k$ to fragment $f_y$
11                   add neighbors of $np_k$ to queue $Q$
12               end if
13           until $Q$ is empty
14       next $p_i$
15   end
```

Algorithm 2.1 starts by looping through all the polygons in set *M*. However not all polygons will be selected to proceed. Only polygons that are not already assigned to a fragment, will be allowed to proceed. The objective of each successive entry into the loop in line 2 of the code listing above, is to identify all the polygons that are similarly oriented and connected together directly or indirectly (see Equation 2.3). So when a polygon p_i is allowed to proceed beyond line 2 of the code-listing, it will seed the creation of a new fragment f_y (line 3).

The *classify* operation in line 5 (and in line 8) refers to the classification of a polygon's orientation to a site vector. This requires a dot product (see Equation 2.1) between the polygon's direction vector and all the site vectors of the set S to find for a closest match. As such, for a larger set of S site vectors, the complexity of this process should increase linearly and thus, the complexity involved for this part of the algorithm is $O(S)$. Due to this cost, the classification result of a polygon is stored once it is subjected to the *classify* operation, thus ensuring that this cost is incurred only once per polygon.

To speed up the search for the closest site vector in the *classify* operation, the hierarchical nature of the site vectors can be utilized. As will be discussed in Section 2.2.1, every set of site vectors used will have 6 distinct key site-vectors. The rest of the vectors will be uniquely grouped to one of these key vectors. Therefore, the *classify* operation can be implemented in two passes. The first pass will be to determine with key site-vector that the polygon belongs to. Then in the second pass, a more refined search can be done within the list of site vectors corresponding to the key site vector found in the first pass. This implementation would speed up the *classify* operation. However for discussion purposes in this thesis, we will assume that the *classify* operation is implemented as a single pass.

To find all the other similarly-oriented and connected polygons for the fragment f_y, Algorithm 2.1 employs a queue to hold a list of polygons np_k identified to be the neighbors of the polygon p_i. As each polygon np_k is popped off the queue, it will be classified to a site vector (in line 8), if not already classified. Then the algorithm determines if this polygon np_k has a similar orientation classification as the polygon

p_i. If so polygon np_k is assigned to the fragment f_y (that also contains p_i). Subsequently, neighbors of polygon np_k are also added to the queue. At the end of each loop (in line 2) for polygon p_i, all the polygons for a fragment f_y would have been identified.

The loops in line 2 for p_i and in line 7 for np_k, will always skip to the polygon that has not been already assigned to a fragment. As such, each polygon is subjected to the *classify* operation only once during the run of the algorithm. In other words, for each run of the loop for polygon p_i, all the polygons for the fragment f_y, are identified and these polygons will not be subjected to the *classify* operation again. Since the complexity for rest of the operations in comparison to the *classify* operation is negligible, the complexity of Algorithm 2.1 is dependent on the *classify* operation for every polygon in the set M. Also since the two loops in line 2 and line 7 are mutually exclusive (i.e. once a polygon is processed in one of the loops, it is excluded from processing in the other loop), the algorithm is $O(MS)$. That is the cost of the algorithm is directly dependent on the number of polygons and number of site vectors.

2.2 | CODE-BOOK GENERATION

The Polygon Grouping process, as discussed in the previous section, consists of two stages. Based on a set of site vectors, the first stage groups polygons that have similar orientations to the sites vectors, into Voronoi Regions around each site vector. The second stage breaks down these groups into fragments that contain polygons that are connected to at least one other polygon in the fragment by at least one vertex. These two stages have been covered in detail in the previous section, except for the selection of the unit vectors for the sites set. *From here on, a set of site vectors will be referred to as a **Code-Book** and a site vector will be called a **Code-Vector**.*

A code-book will determine the number of fragments produced and the distribution of the polygons throughout these fragments. If the number of fragments is too *large*, the performance of the rendering pipeline might be adversely affected. For example, the performance of Ambient Occlusion [47, 49] operation decreases with increasing number of polygons. Since we can use a fragment to approximate a group of polygons as one large polygon, the performance of the Ambient Occlusion calculation will only improve by decreasing the number of fragments. However, if the number of fragments is too small, approximation of a fragment to single flat plane might not be desirable (see Figure 2.7).

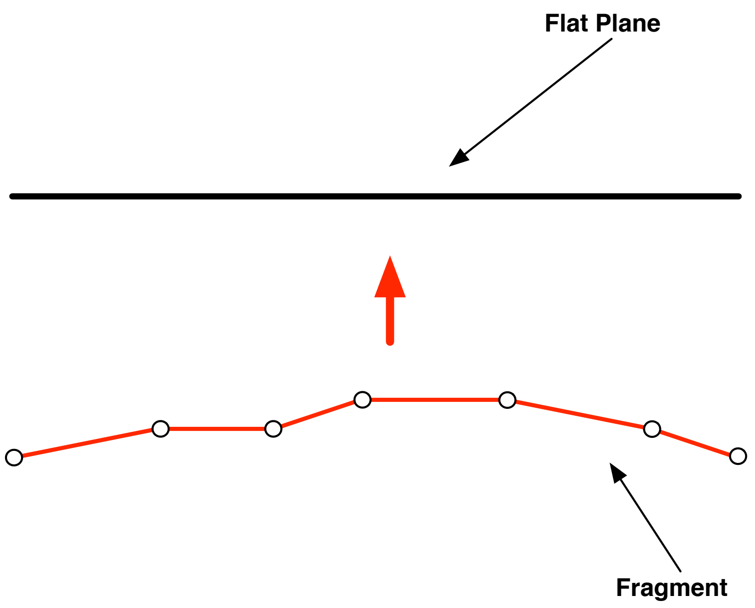


Figure 2.7: Shows the cross section view of a set of polygons that are grouped together to form a fragment. This fragment, however, cannot be approximated into a single flat plane without a lot of loss in accuracy due to the curvature of the fragment.

Approximating the fragment, shown in Figure 2.7, as a single flat plane will affect the accuracy of the results of operations such as Mesh Deformation (that depends on the accuracy of geometric features of a 3D mesh). Thus, finding the balance between the number of fragments and the distribution of polygons for a given 3D mesh is essential to the rendering pipeline's performance and quality of render.

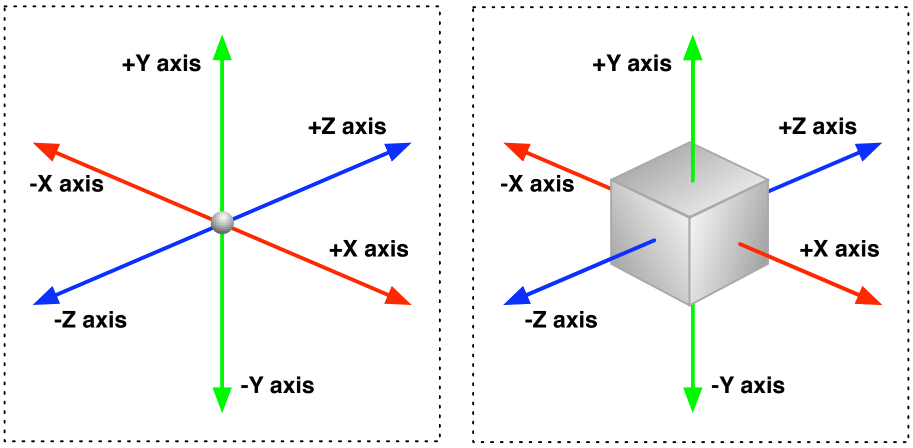
To achieve this balance, the Code-Book Generation process needs to provide the flexibility to adjust the number of code-vectors and their distribution about the center of a unit sphere (for explanation of using a unit sphere, see Section 2.1.1: Grouping using the Orientation Attribute). The following three sub-sections are devoted to discussing the various schemas to generate a code-book that will conform to the orientations of the polygons of a 3D mesh model. The first two sub-sections will introduce schemas that can create a generic code-book and map the orientations of the polygons in a set to the code-vectors. The first schema is a quick-and-dirty method to creating a code-book and does not take the dimensions of a 3D mesh model into account. The second schema, expands on the first one by selectively increasing code-vectors along certain axes. In the third sub-section, further refinements this process will be introduced.

2.2.1 | Schema 1: Uniform Ellipsoidal Schema

The aim of the first schema is to create a generic code-book whose code-vectors are distributed evenly throughout the code-book. In other words, each code-vector should point in a distinct direction and should have the same angle of separation between its neighboring code-vectors. In this sub-section, we will look at how to generate a code-book based on these two considerations.

In Section 2.1.1, Grouping using the Orientation Attribute, a site vector from a sites set is described as a unit vector that radiates from the center of a unit sphere positioned at the origin of a 3D scene. Also, each site vector is orientated about the center of the sphere. Thus, site vectors, distributed about the unit sphere, will point in distinct directions.

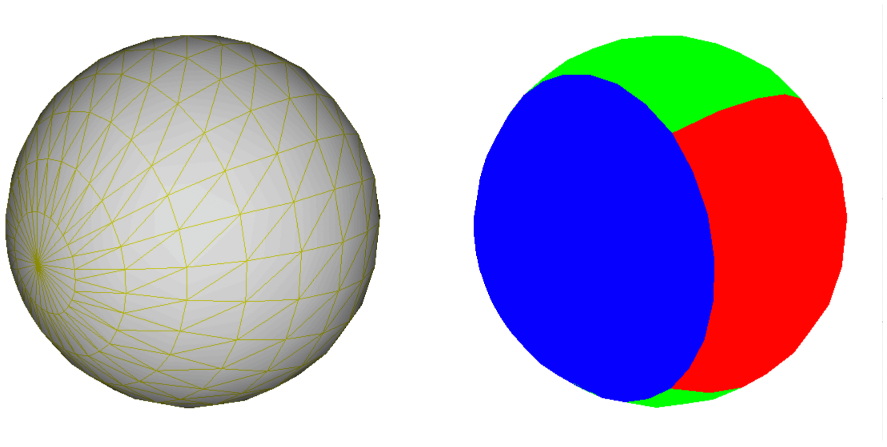
Figure 2.8(a) shows vectors radiating from a point in 3D space, along each of the six major axes (X, Y, Z, -X, -Y, and -Z). These six vectors point in six distinct directions and have a separation angle of 90 degrees from their neighbors. Another way to visualize these vectors would be as depicted in Figure 2.8(b).



(a) (b)

Figure 2.8: Shows (a) a set of 6 vectors radiating from the origin along the 6 major axes; (b) the polygon normal vectors of a unit cube are another way to visualize the vectors in (a).

Each polygon of the unit cube, in Figure 2.8(b), has a normal (expressed as a vector) that radiates from the center of the unit cube. These normal vectors corresponds to the vectors shown in Figure 2.8(a). Thus, these normal vectors are used to form the code-vectors that are used in the Polygon Grouping process. The following figure is an example of using these six vectors to partition a 3D mesh model.



(a) (b)

Figure 2.9: Shows the end result (b) after applying the Polygon Grouping process on the 3D model in (a). Polygons are grouped based on their orientations with respect to the 6 vectors introduced in Figure 2.8.

In Figure 2.9, the polygons of the sphere are grouped into 6 fragments. Each of these fragments has a normal that corresponds to the normal vectors of the unit cube in Figure 2.8(b).

In the beginning of this section (Code-Book Generation), the flexibility to control the number of fragments and the distribution of the polygons throughout these fragments was stated as an essential consideration. And in this sub-section, one of the goals for the first schema is to create a code-book of vectors that have equal separation angles from their neighbors. So, for example, in order to increase the number of fragments, the number of distinct code-vectors can be increased as shown in Figure 2.10.

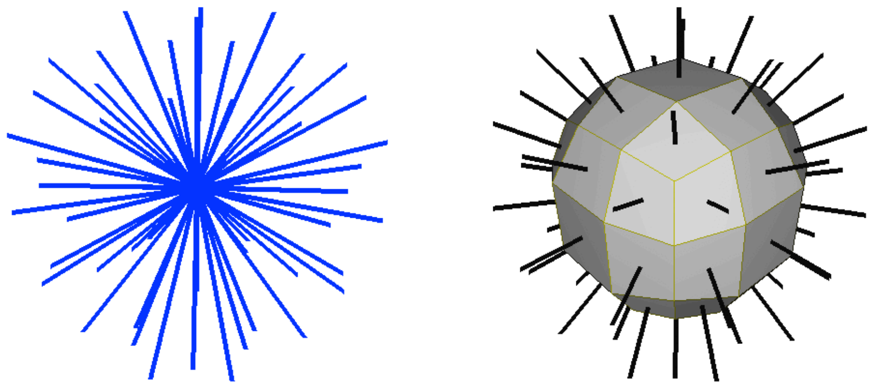


Figure 2.10: Shows (a) a set of 54 distinct code-vectors radiating from the origin; (b) an alternative way to visualize the set of 54 code-vectors as a sphere.

There are 54 distinct code-vectors shown in Figure 2.10(a). The angle separation between any two neighboring vectors is 30 degrees. The alternate method to visualize these vectors is shown in Figure 2.10(b) where the normals of the polygons of the unit sphere correspond to the vectors of the Figure 2.10(a). Incidentally if the vertices of the unit sphere in Figure 2.10(b) is expanded to fit the unit cube's profile, it will look like as follows:

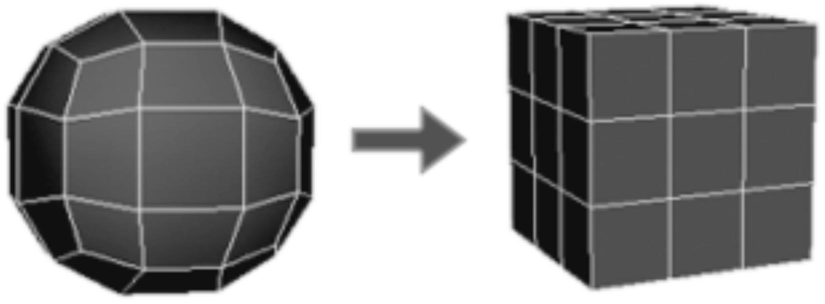


Figure 2.11: Shows that as the vertices of the unit sphere is expanded outwards to fit a cube's profile, a sub-divided unit cube results.

The resulting geometry is also a unit cube. In this case, the unit sphere is transformed into a unit cube with 3 sub-divisions along each of its sides. Reversing the transformation process will recover the unit sphere whose polygon normals will provide the code-vectors as shown in Figure 2.10. Thus, it is possible to use a unit cube with arbitrary number of sub-divisions to create a code-book with distinct unit vectors that are uniformly distributed about the center of the cube (or sphere). For discussion purposes the intermediate unit sphere of this process will be referred to as a Gaea-Sphere throughout this thesis.

2.2.1.1 | GAEA SPHERE PROPERTIES

This sub-section will look at some of the notable properties of a Gaea-Sphere.

◉ **Naming Convention:** A Gaea-Sphere is a 3D geometric model whose polygon normals are uniformly distributed about the model's center. As shown in Figures 2.8 through 2.11, a Gaea-Sphere can be derived from a unit cube with equal number of sub-divisions along each side of the cube. The number of sub-divisions will determine the number of polygon normals that the Gaea-Sphere will yield. For discussion purposes, a Gaea-Sphere whose corresponding unit cube has *N* sub-divisions, will be referred to as *Gaea-N* where *N* is greater than or equal to *1*. That is, the number of sub-divisions will be reflected in the name of the sphere. Figure 2.12 shows examples of using this naming convention to refer to Gaea-Spheres.

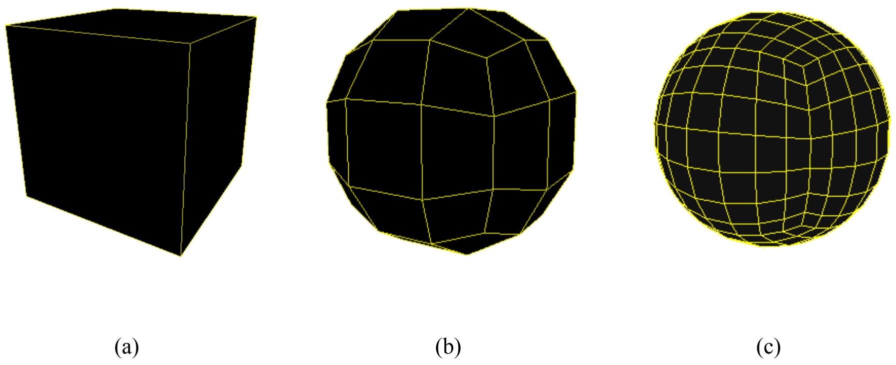


Figure 2.12: Shows geometric models of (a) Gaea-1; (b) Gaea-3 (c) Gaea-7.

◉ **One Subdivision:** The model shown in Figure 2.12(a) is referred to as *Gaea-1*. This model is essentially a unit cube with 1 sub-division along each of its axes. Despite its appearance, it will be referred to as a Gaea-Sphere in this thesis. This sphere will yield six distinct code-vectors, one along each axis (i.e. X, Y, Z, -X, -Y, and -Z). Since the minimum number of sub-divisions allowed on a unit cube is one, Gaea-1 creates the smallest code-book possible for this schema. In other words, a Gaea-Sphere will create a code-book of at least six code-vectors.

◉ **Number of Polygons:** The model shown in Figure 2.12(b) is a Gaea-3. This sphere has 54 polygons (i.e. it creates a code-book of 54 distinct code-vectors).

The model shown in Figure 2.12(c) is a Gaea-7 and it has 294 polygons. Thus, the relationship between the number of sub-divisions and the number of polygons of a Gaea-Sphere can be expressed as:

$$V(n) = V(1) \cdot n^2 , \qquad (eq\ 2.4)$$

where *V(n) is* the number of polygons of *Gaea-n* and *V(1)* is the number of polygons in Gaea-1, which is six polygons.

Using Equation 2.4, the number of polygons (i.e. the number of code-vectors yielded) for a Gaea-Sphere is calculated as the product between the number of polygons in Gaea-1 and the squared number of sub-divisions of the Gaea-Sphere. Thus, it can be deduced that all Gaea-Spheres will create a minimum of six distinct code-vectors.

◉ **Hierarchy of Code Vectors:** Another deduction from Equation 2.4 is that as *n*, the number of sub-divisions, decreases, the Gaea-Sphere converges to a Gaea-1. In other words, as *n* is decreased to *(n-1)*, the code-vectors of *Gaea-n* are collapsed into *Gaea-(n-1)*. Once a code-vector of the *Gaea-n* is collapsed into the *Gaea-(n-1)*, that code-vector will be replaced with the corresponding code-vector of *Gaea-(n-1)*. This will continue till all the code-vectors are collapsed into one of the 6 code-vectors of Gaea-1 (see Figure 2.13).

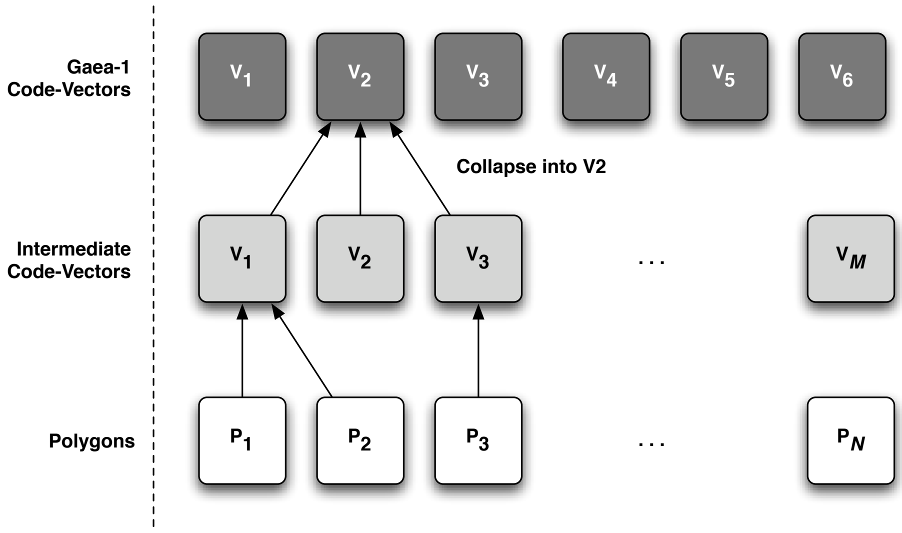


Figure 2.13: Shows an example of a hierarchy of code-vectors. Polygons P1 and P2 are mapped to the intermediate code-vector V1 while Polygon P3 is mapped to the intermediate code-vector V3. These code-vectors (V1 and V3) are in-turn mapped to the code-vector V2 of the Gaea-1. This is a simple demonstration of the multiple levels of Polygon Grouping for a polygon can be achieved by mapping intermediate code-vectors to the base code-vectors.

This hierarchical nature the Gaea-Spheres leads to dynamic reduction in the number of fragments created from the Polygon Grouping process, as each code-vector has a hierarchical path. As *n* is decreased, the Polygon Grouping process can follow the hierarchy of the code-vector upwards and determine which fragments will be merged.

2.2.1.2 | Uniform Ellipsoidal Schema

The schema discussed in this section requires the code-vectors to be uniformly distributed such that the angle of separation between neighboring vectors are equal. Using Gaea-Spheres, a code-book (whose size can be determined using Equation 2.4), that satisfies this condition, can be created. Since the resulting vectors are uniformly distributed about the center of the sphere, this schema is called Uniform Ellipsoidal

Schema. Figure 2.14 shows the result of applying this schema to the Polygon Grouping process on a 3D mesh model.

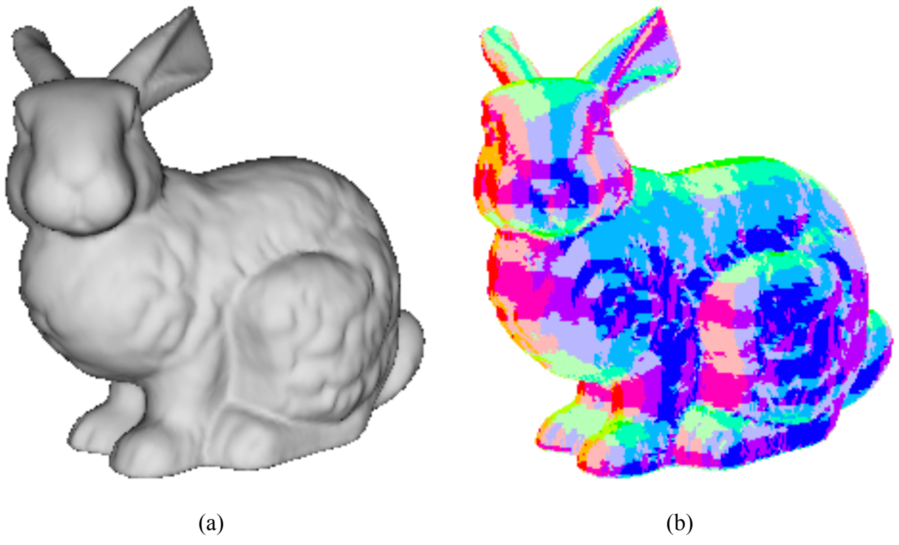


Figure 2.14: Shows a 3D mesh model (commonly known as the Stanford Bunny) when subjected to Polygon Grouping using Gaea-3. Colors for the fragments are based on the code-vectors.

2.2.1.3 | Fragment Masks

In Figure 2.14, a Gaea-3 is applied to the 3D mesh model that has a polygon count of 69,451. This resulted in creating 5,076 number of fragments. These fragments (as discussed in Chapter 3) will be distributed to various machines to render and the resulting sub-images will be composited to form the complete rendered image of the scene. An important goal as stated in Chapter 3 is to eliminate the need for depth buffer during the composition of the sub-images. In other words, each sub-image should not have any overlapping regions with other sub-images.

To achieve non-overlapping sub-images, we would need to mask out the regions occupied by fragments rendered in other machines. Masking out these regions would require information about all the fragments distributed to every rendering machine.

Replicating a 3D model or scene to every machine can be an expensive process, and thus we propose transferring the simplified fragments information only.

This problem was tackled in [29] as described in Section 1.2. Since the set of connected polygons in these fragments are similarly orientated, the curvatures of these fragments become negligible as the resolution of the code-book is increased. Thus it is possible to flatten a fragment by removing the finer details of the fragment. When a fragment is flattened, only the boundary vertices of the fragment are retained and re-triangulated.

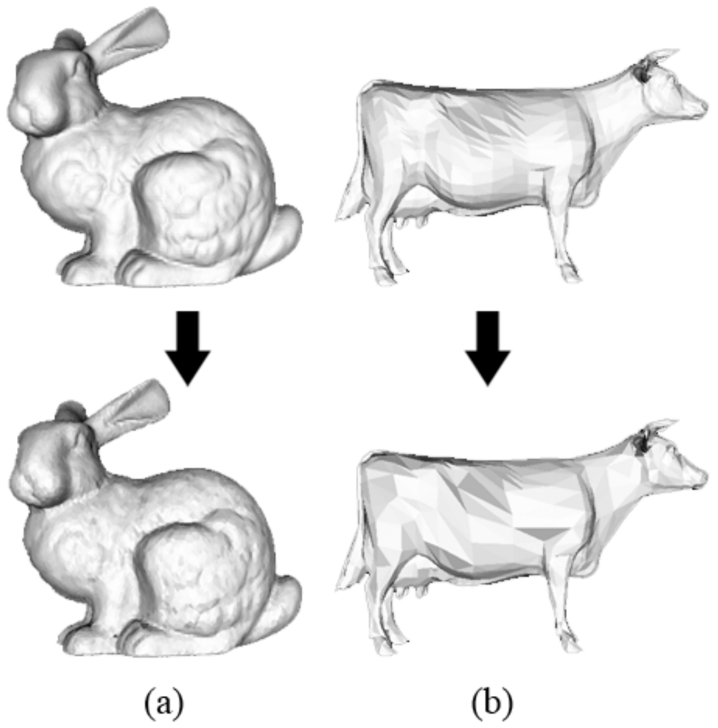


Figure 2.15: Shows the effects of flattening fragments. The resultant mesh in (a) has 64% polygon reduction; and (b) has 40% polygon reduction.

These flattened fragments shall serve as masks. When a machine is assigned a fragment to render, it is also provided with the flattened information of all other fragments. The fragment masks will be rendered together with the assigned detailed fragment (see Figure 2.16).

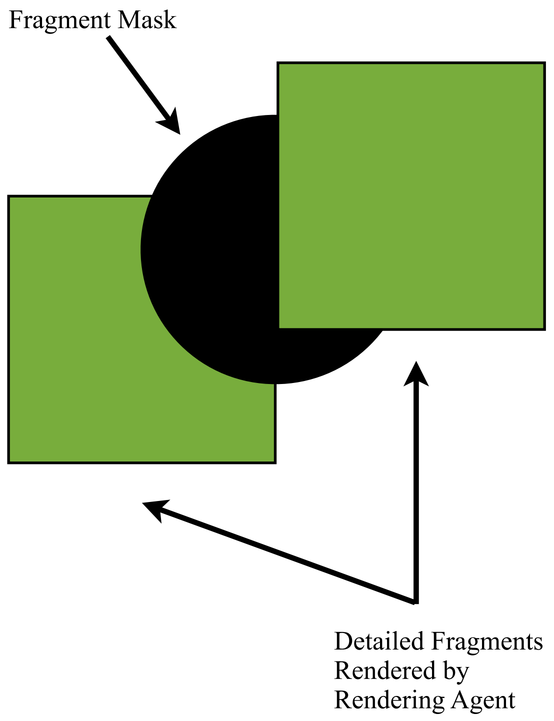


Figure 2.16: Shows an illustration of how simplified fragments are used to mask out overlapping regions in sub-images.

Unlike the detailed fragments, the flattened fragments will be rendered without texture and lighting effects. Non-overlapping regions occupied by the flattened fragments are set as transparent in the final sub-image before it is transmitted for composition with other sub-images.

2.2.2 | Schema 2: Non-uniform Ellipsoidal Schema

The role of the code-vectors can be viewed as the sampling of a 3D mesh model's polygon orientations. Thus, it is important for the code-vectors to represent key geometric features of the model. That is prominent polygon orientation information need to be captured in the code-book.

The previous sub-section dealt with how to design a code-book with code-vectors distributed uniformly. This code-book provides a generic solution for various 3D mesh models, regardless of their dimensions. By disregarding the dimensions of a 3D mesh model, some of the geometric information is lost[3] in the process of Polygon Grouping. This problem is illustrated in Figure 2.17.

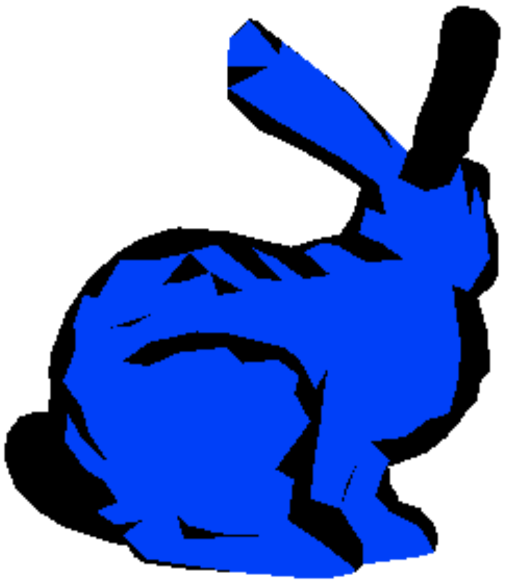


Figure 2.17: Shows a 3D mesh model (a low-resolution version of the Stanford Bunny model from Figure 2.14) partitioned using Gaea-1. Only the fragment with highest concentration is highlighted.

In Figure 2.17, partitioning the 3D mesh model (contains 1443 polygons) with Gaea-1 produces 116 fragments. On average, there are 12 polygons packed into a fragment. However, largest concentration of polygons found in a fragment is 254. That is about 18% of the polygons of the 3D mesh model has been grouped into that one fragment. The possibility that some these polygons will have critical geometric features is very high. The same 3D mesh model when subjected to mesh partitioning using Gaea-2

[3] The geometric information is not technically "lost" because the polygon orientations are still preserved within each fragment. Instead the distribution of the polygons and the orientation of fragments are not representative of the geometric features of the 3D mesh model.

yields 462 fragments. On average, there are 3 polygons packed into a fragment and the largest concentration of polygons found in a fragment is 34. That is only 2.4% of the polygons of the 3D mesh model has been grouped into that one fragment. Thus, the chances of losing a critical feature in that fragment is much lesser than the one from the previous mesh partitioning process. However, the number of fragments has increased four fold with the increase in the number of sub-divisions of the Gaea-Sphere for mesh partitioning.

The schema, proposed in this sub-section, solves this problem by creating a code-book that has varying number of code-vectors along the axes. In other words, the number of sub-divisions of the corresponding Gaea-Sphere will be different along the each axis. Thus, unlike the Uniform Ellipsoidal Schema, the current schema will not have a uniformly distributed set of code-vectors and hence, this schema is called Non-uniform Ellipsoidal Schema.

To realize such a schema, we employ an iterative process where the number of sub-division of the Gaea-Sphere is increased till the largest concentration of polygons in a fragment is below a certain threshold value. The process starts off by using Gaea-1 to partition a 3D mesh model. At the end of an iteration, the number of sub-divisions along the axis of the fragment that has the highest concentration of polygons, is increased. This, however, results in a Gaea-Sphere that does not use the same naming convention as that used in the Uniform Ellipsoidal Schema. To reflect the varying number of sub-division in each axis, the new naming convention for the Gaea-Spheres is *Gaea-*N_x,N_y,N_z. For example a Gaea-Sphere with 1 sub-division along the X axis, 4 sub-divisions along the Y axis, and 3 sub-divisions along the Z axis, is referred to as

Gaea-1,4,3. Since the number of divisions are different for each axis, Equation 2.4 is replaced with Equation 2.5 for finding the number of polygons (or code-vectors) for a *Gaea-N_x,N_y,N_z*.

$$V(nx, ny, nz) = (nx \cdot ny + ny \cdot nz + nz \cdot nx) \cdot 2\,, \qquad (eq\,2.5)$$

where *V(nx, ny, nz)* represents the number of polygons of a *Gaea-N_x,N_y,N_z*, *nx* represents the number of sub-divisions along the X axis, *ny* represents the number of sub-divisions along the Y axis, and *nz* represents the number of sub-divisions along the Z axis. When the *nx*, *ny*, *nz* are set to be the same, Equation 2.5 will reduce to Equation 2.4 as follows:

$$\begin{aligned} V(n,n,n) &= (n \cdot n + n \cdot n + n \cdot n) \cdot 2 \\ &= (3n^2) \cdot 2 \\ &= 6n^2 \end{aligned}$$

As mentioned in Section 2.2.1.1, *V(1)* is the smallest Gaea-Sphere that can be created and can yield 6 polygons. Thus replacing the value 6 with *V(1)* without loss of generality in the above derivation, shows that Equation 2.5 reduces to Equation 2.4 when the sub-divisions of a *Gaea-N_x,N_y,N_z* are the same.

Subjecting the 3D mesh model, shown in Figure 2.17, to the Non-uniform Ellipsoidal Schema, yields a fragment count of 9181 and the highest concentration of polygon count in a fragment is 255. Starting with Gaea-1, the iteration process was stopped at an upper limit of 256 code-vectors, resulting in Gaea-4,7,3. Thus, this schema provides a better distribution of polygons across the fragments while keeping the code-book size to a minimum as compared to the Uniform Ellipsoidal Schema.

2.2.3 | Weeding Orphans

In the previous section, a non-uniform distribution of code-vectors proofed to represent the geometric structure of a 3D mesh model better than a uniformly distributed set of code-vectors. This was achieved by sub-dividing a Gaea-Sphere along the axis that has the highest concentration of polygons in a fragment. However, this schema is not without its problems. As shown in Figure 2.18, applying the Non-uniform Ellipsoidal Schema to a 3D mesh model can sometimes create fragments with only one polygon in them. These fragments are called *orphans*.

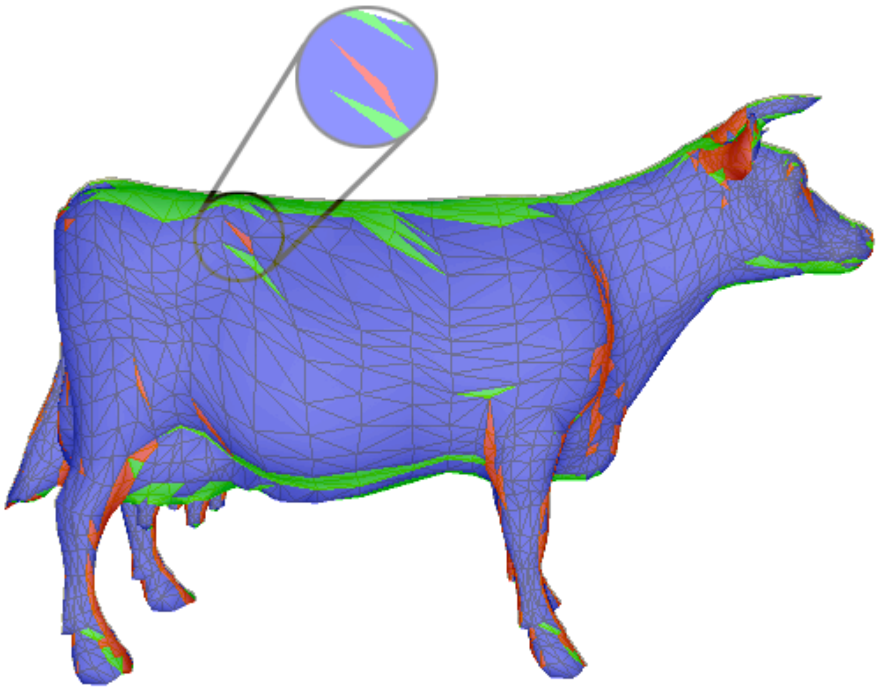


Figure 2.18: Shows an "orphan" fragment (highlighted in red color) of a 3D mesh, partitioned using Gaea-1.

Storing only one polygon in a fragment is not cost efficient in terms of both storage and computation. An orphan requires space for the polygon information as well as the fragment information. Thus, essentially the cost of storing too many orphans will exceed the cost of storing just the polygons, without the fragments information. Also processing too many orphans does not provide computation cost savings for operations such as Ambient Occlusion as the processing time for an orphan will be the same as that of its corresponding polygon. The 3D mesh model in Figure 2.18 yielded 120 number of orphans. That is 46% of the fragments created are orphans. In this subsection, a method called Orphan Weeding is introduced to reduce the number of orphans after subjecting a 3D mesh to polygon grouping.

The idea for this method is adopted from the Popularity Algorithm commonly used for color quantization in the reduction of 2D image resolution or for 2D image compression. The color quantization algorithm finds the more frequently occurring colors and includes them in its code-book. Then based on this code-book, the image is quantized (i.e. the original color pixels are replaced with the closest color values from the code-book). Likewise, the Orphan Weeding method seeks to retain the "popular" fragments (or the fragments with high polygon concentration) for a given 3D mesh. The "popularity" of a fragment is weighed against that of the other fragments and the less "popular" ones are weeded. These weeded fragments will be subjected to Polygon Grouping again but this time based on a code-book consisting of only the code-vectors of the neighboring fragments.

The "popularity" of a fragment is the weighted sum of two of its attributes and they are namely,

1. **Strength:** In the color quantization, popular colors are the ones that are recurring most often. Likewise the strength attribute of a fragment is defined as the number

of polygons that are associated with it verses the total number of polygons in a 3D mesh model. The following equation represents this attribute mathematically.

$$S(F_j) = \left\{ PF_j \in M : \frac{\Sigma_{PF_j} 1}{\Sigma_M 1} \right\}, \qquad (eq\ 2.6)$$

where F_j refers to a fragment j of a mesh model M and PF_j is the set of polygons of the fragment j. Attributing a cost of 1 to each polygon in the mesh M, the strength attribute of a fragment is expressed as the percentage of polygons of mesh M found in fragment j. This equation will hold true if and only if the polygon grouping process guarantees that every polygon is uniquely grouped to only one fragment.

2. **Influence:** An orphan can sometimes contain a polygon whose area of influence in the 3D mesh model is quite significant. Weeding the orphan can lead to the loss of geometric structure of the fragments. Thus, a influence attribute is considered and is defined as the amount of accumulated area covered by the polygons that are associated with a particular fragment. The following equation represents this attribute mathematically.

$$I(F_j) = \left\{ PF_j \in M : \frac{A_j}{A_M} \right\}, \qquad (eq\ 2.7)$$

where A denotes the surface area of a group of polygons in a fragment or mesh

$$A_k = \sum_{p=1}^{N_k} \left[\sum_{i=0}^{2} \| (v_{2,i} - v_{1,i}) \otimes (v_{3,i} - v_{1,i}) \| \right], \ for\ N \in R^k$$

Equation 2.7 calculates the influence of the fragment F_j, as the area of all the polygons associated with that fragment against the total surface area of the 3D mesh model, M.

The popularity of a fragment is determined as the weighted sum of the strength and influence attributes of the fragment. As a rule of thumb, the weights are set to 0.5 for both the attributes. This thus emphasizes both the attributes equally in identifying orphans. The weeding process removes the fragments from the bottom of the list (i.e. the least popular ones first) and stops when a terminating condition is reached (e.g. number of fragments). At the end of the process, a minimal set of fragments that represents the critical features of the 3D mesh model is created.

2.3 | Summary

Our development of the Ellipsoidal Mesh Partitioning technique was motivated by three main considerations: algorithmic approach, geometric feature preservation, and balanced distribution of polygons across all fragments. By using both the Uniform and Non-uniform Ellipsoidal Schemas that are modeled after the geometric shape called Gaea-Sphere, a code-book for the mesh partitioning technique can be created algorithmically. Since a Gaea-Sphere guarantees that each code-vector is distinct and covers all 6 orthogonal axes, polygons with similar orientations to the code-vectors can be grouped together, thus preserving geometric features. The resulting fragments that consist of connect polygons that are similarly oriented can be approximated into flat or simplified polygons. By controlling the resolution of the code-book, we have shown how to balance the distribution of the polygons amongst the various fragments.

CHAPTER 3

3 | COLLABORATIVE RENDERING

Collaborative Rendering is a form of Distributed Rendering whereby several computers come together to render a 3D scene. Unlike distributed rendering, collaborative rendering, has minimal reliance on a dedicated server to manage part of (or even the entire) process. That is, the computers involved in the rendering process, are aware of their peers on the network and communicate with each other to render a scene or a frame. Thus, issues such as server bottlenecks and scalability limitations of a distributed rendering environment are not present in a collaborative rendering environment that uses P2P technology.

This chapter introduces our collaborative rendering system called the Ellipsoidal Collaborative Visualization Environment (ECOVE). This system adopts the conventional graphics pipeline for distributed rendering, partitions a 3D mesh using Ellipsoidal Mesh Partitioning technique for distribution, and uses P2P to collaboratively manage the rendering process (see Figure 3.1).

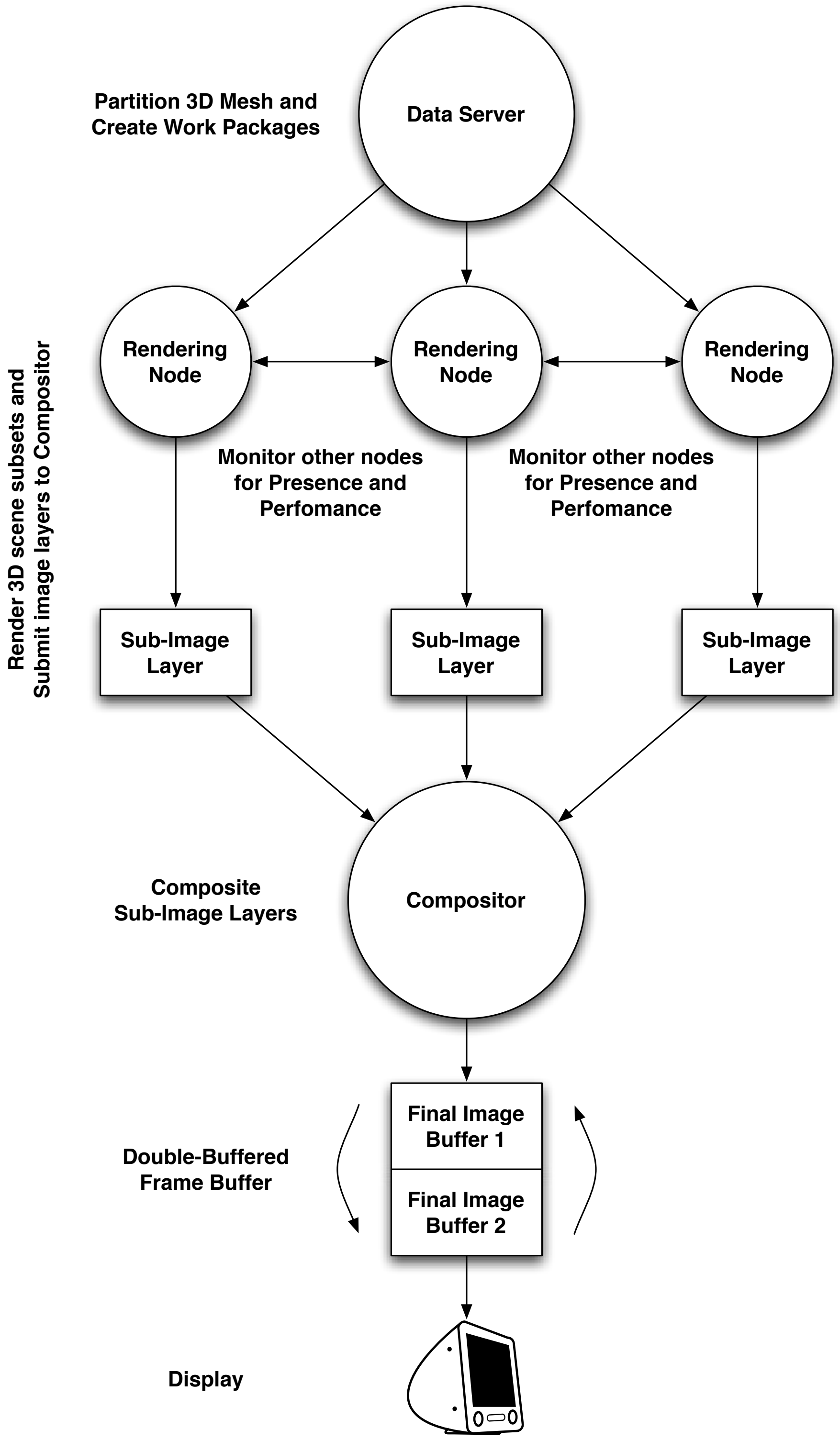

Partition 3D Mesh and Create Work Packages
Data Server
Render 3D scene subsets and Submit image layers to Compositor
Rendering Node
Rendering Node
Rendering Node
Monitor other nodes for Presence and Perfomance
Monitor other nodes for Presence and Perfomance
Sub-Image Layer
Sub-Image Layer
Sub-Image Layer
Composite Sub-Image Layers
Compositor
Double-Buffered Frame Buffer
Final Image Buffer 1
Final Image Buffer 2
Display

Figure 3.1: Shows the overview of the ECOVE system. The Data Server serves out fragments of a 3D mesh to the Rendering Nodes. The rendered sub-images from each Rendering Node are sent for compositing at the client machine and displayed on screen.

In particular, ECOVE is structured to:

1. Use the multiple hardware graphics accelerators available on the network to increase rendering performance over what is achievable by a sequential renderer that uses a single accelerator.

2. Decouple communication bandwidth requirements from the complexity of the scene and the number of rendering nodes.

3. Avoid and/or minimize load imbalances with minimal reliance on a centralized server.

Chapter 2 looked at how to partition a mesh into groups of polygons called fragments (see Section 2.1.2 for definition of a fragment). The problem posed was to break up the polygons of a mesh into fragments based on their orientation and locality. Another constraint introduced was to ensure the distribution of the polygons throughout the fragments is even. However this problem was approached without regard to work partitioning and assignment. Thus, this chapter considers the questions of how to partition the overall rendering work in each frame into individual tasks and how to perform initial assignment of these tasks in a load-balanced manner. While work partitioning and assignment is a fundamental problem for all distributed/parallel applications, the rendering domain poses two additional challenges when coupled with the above structural requirements:

4. To make use of hardware-assisted rendering, the rendering work must be partitioned in a way that does not require accessing information generated and maintained internally to the hardware accelerators.

5. Because of the need for visibility culling and sorting, the partitioning strategy used can have a considerable effect on the total amount of rendering computation and the size of the partial images that must be communicated [67].

ECOVE provides a novel approach to the partitioning problem called Context Aware Mesh Partitioning (CAMP) to help meet these challenges. CAMP extends on the Ellipsoidal Mesh Partitioning (EMP) technique to consider workload partitioning and assignment. Given a set of meshes in a 3D scene to be rendered, the basic idea behind CAMP is to assign two subsets of fragments to every rendering node in such a way that:

◉ the first subset, called the owner set, will have fragments with similar locality, and

◉ the second subset, called the buddy set, will be a copy of the another node’s owner set.

All nodes will focus on rendering just the owner set. In the event that a rendering node drops out of the network or quits the rendering environment, the node holding the abandoned workload as its buddy set will take on the additional responsibility until the Data Server re-partitions the workload. At the end of the frame, the partial images generated by different rendering nodes are composited together to form the final image.

More specifically, ECOVE as a system proposes to:

1. partition a 3D mesh using EMP technique into fragments,

2. for a system of *P* nodes, create *P* work partitions (or owner sets) such that:

- ◉ the fragments in the each owner set will have similar locality,
- ◉ the expected rendering plus transmission time of each owner set is load balanced, and
- ◉ total amount of data pixels that must be communicated is minimal.

3. assign an owner set to every node, and
4. assign each node to monitor another node (called buddy node) for presence.

Each node will additionally receive a pseudo mesh (see Section 2.3). Such an approximated mesh representation is particularly useful in advanced hardware rendering techniques such as Shaders. For example, an Ambient Occlusion Shader need not know the detailed structure of the entire mesh but only the general size and position of the polygons with reference to the point at which the shadow is calculated.

The remainder of this chapter will look into ECOVE in more detail, describing the CAMP algorithm's ability to perform initial work partitioning in a load-balanced manner while observing the necessary 3D mesh partitioning constraints. The final section of this chapter will look at how P2P is used to manage the rendering process collaboratively.

3.1 | Why ECOVE?

Recall from Section 1.1 that distributed rendering systems can typically be classified as one of sort-first, sort-middle, or sort-last. In sort-first, work is distributed based on an image partitioning, where each node in the system is assigned responsibility for calculating the effect of all primitives on the pixels in a portion of the final image. In sort-middle, the geometric transformation and rasterization phases are distributed independently across the system. In sort-last, each node is assigned a subset of the polygons in the scene, without restrictions on the position of the polygons. At the end of each frame, once each node has rendered the image corresponding to its assigned polygons, the pixels must be sorted, typically using Z-buffering.

In sort-first and sort-middle, the required redistribution of primitives is a fundamental problem in our targeted environment. This requires either recomputing the geometric transformation of primitives that must be redistributed or accessing information that may be hidden inside a hardware graphics pipeline. Second, the bandwidth requirement for these two classes directly depends on the complexity of the scene that is being rendered, violating one of our basic goals (see previous section). Finally, because the mapping of primitives to image space changes depending on the viewpoint, the amount of data that must be transmitted can change very unpredictably from frame to frame. For example, in an assessment of the practicality of sort-first, Mueller [16] gives communication measurements for three different scenes. For each of these scenes (and the particular path taken through the scene), the number of primitives that a sort-first system would need to redistribute can vary widely from frame to frame, ranging from 5% to 100% of the scene. It seems extremely difficult to achieve predictable real-time response in the face of such large variability in bandwidth requirement.

Sort-last, on the other hand, is compatible with hardware-assisted rendering because in each frame, the rendering nodes render their assigned work independently. Thus, the problem of generating each partial image looks exactly as if it were an independent rendering problem, allowing the rendering nodes to employ hardware-assisted rendering.

ECOVE is similar to sort-last in that rendering nodes generate images independently, and so is compatible with hardware-assisted rendering. However unlike sort-last, primitives in ECOVE, are typically assigned to renderers with regard to their orientation and locality. This means that each renderer does not need to send the entire image for each frame. Also, for certain hardware-assisted rendering techniques such as Graphics Shaders, the complete geometric structure needs to be provided to each renderer. Thus many sort-last implementations resort to replicating the entire 3D scene on all the nodes. ECOVE, however, only sends the portion of the mesh assigned to the corresponding renderer and a simplified representation of all the fragments (see Section 2.2.1.3). Shaders can also significantly benefit in terms of speed due to the reduced number polygons that they have to include in their calculations by using the simplified fragments.

3.2 | ECOVE ARCHITECTURE

Figure 3.1 shows a possible architecture for ECOVE. In this diagram, there are four types of nodes and they are namely, *Data Server*, *Rendering Nodes*, *Compositor*, and *Display*. For ease of implementation, the Data Server, the Compositor and the Display can be implemented as the same node (see Figure 3.2). That is, functionally this node is responsible for:

◉ partitioning the 3D scene meshes,

◉ assigning work partitions to the rendering nodes,

◉ computing the current viewpoint at the beginning of every frame,

◉ composite sub-image layers from the rendering nodes, and

◉ display final image at the end of every frame.

It is also responsible for re-partitioning and re-assigning the workload in the event a rendering node drops out the network.

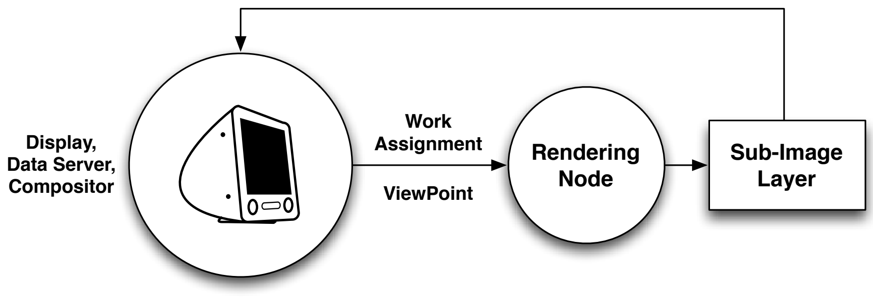


Figure 3.2: Shows the interaction between a Rendering node and the Display node. In implementation of ECOVE, the Display node that interacts with the user, also doubles up as the Data Server, and the Compositor for the Rendering node.

In each frame, every rendering node receives a new viewpoint, renders the fragments it is assigned, and sends the generated image back to the display node. The rendered (partial) image generated at each rendering node is referred to as a sub-image layer (since the final image is a composite of the generated images).

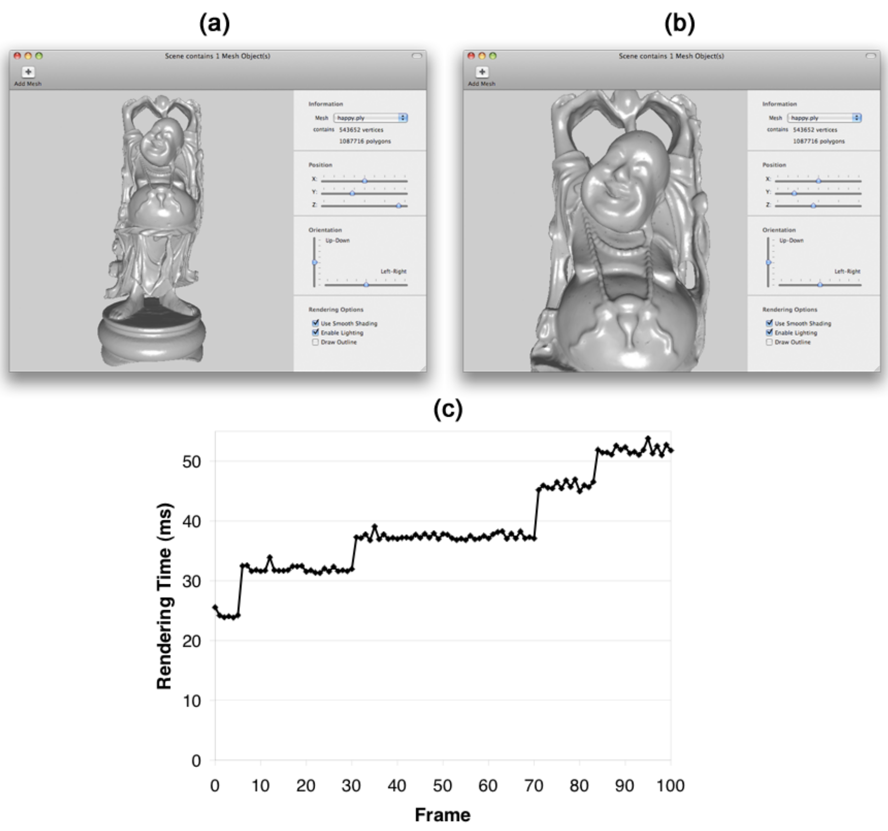
In this section, three essential components of ECOVE, with respect to the architecture shown in Figure 3.2, will be described:

1. A method for estimating the rendering time of each fragment.
2. A method for estimating the footprint of each fragment.
3. The CAMP algorithm with regards to rendering and communication costs.

3.2.1 | Rendering Time of a Fragment

In order to partition the overall rendering work in a load-balanced manner, we must be able to estimate the rendering loads of scene objects (since the rendering of each fragment corresponds to a task that must be assigned to some node). One possible basis for such an estimation is the number of primitives in each fragment. Estimation methods based on primitive count can be very inaccurate, however, because the time required to render a set of polygons can vary widely depending on the viewpoint. Thus, we take a different approach that leverages the fact that our targeted application domain involves interactive rendering, such as that performed by an OpenGL 3D mesh viewer.

In an interactive application such as an OpenGL viewer, the viewpoint typically does not change significantly from frame to frame because the user is navigating through the scene in real-time. This implies that each 3D mesh object's rendering time in one frame will be about equal to its rendering time during the previous frame. (Exceptions to this include abrupt jumps to predefined viewpoints, mesh objects coming into or going out of visibility, and crossings of level-of-detail thresholds.) For example, Figure 3.3(c) plots the rendering time of a 3D mesh object as the viewer "walks" from the viewpoint shown in Figure 3.3(a) to that shown in Figure 3.3(b). Note that while the rendering times of the object at (a) and (b) are quite different[4], they are (almost always) very similar in adjacent frames.



Information
Mesh happy.ply
contains 543652 vertices
1087716 polygons
Position
X:
Y:
Z:
Orientation
Up-Down
Left-Right
Rendering Options
☑ Use Smooth Shading
☑ Enable Lighting
☐ Draw Outline

Figure 3.3: Changes in the rendering time of a scene object with multiple levels of detail: (a) the initial view, where the object is far away from the viewpoint; (b) the final view, where the object is close to the viewpoint; (c) plot of rendering time vs. frame number as the

[4] This is consistent with and further supports the fact that estimation methods based on primitive count is not very accurate.

viewer moves from (a) to (b). This measurement was taken using an OpenGL viewer running on a 2.16 GHz Intel machine with Intel GMA 950 Graphics Chipset.

Based on this observation, the rendering time of a fragment in the last frame can be measured as the predictor of its rendering time in the current frame. On most current processors, the rendering time of a fragment is measured with very little overhead by reading a free running counter in the processor.

3.2.2 | Footprint of a Fragment

Estimating fragments' footprints in the final image of a frame is important for two reasons:

1. The transmission time of an image layer may comprise a substantial portion of the load on a rendering node and so must be taken into account by the CAMP algorithm. This required transmission time can be estimated if the (approximate) aggregated footprint of the fragments assigned to each node and the achievable bandwidth is known.

2. Figure 3.1 shows that, at the end of each frame, all rendering nodes send their image layers to the display node. Typically, this many-to-one communication must be performed sequentially because it is assumed that the display node has only one network connection and receiving is typically more expensive than sending (hence multiplexing sends from multiple senders would only degrade performance). This serialization implies that it is important to minimize the total amount of per-frame communication for ECOVE to scale. This in turn implies that CAMP should strive to assign fragments that are clustered together and have

similar orientations. This critical optimization is only possible if we can estimate fragments' footprints.

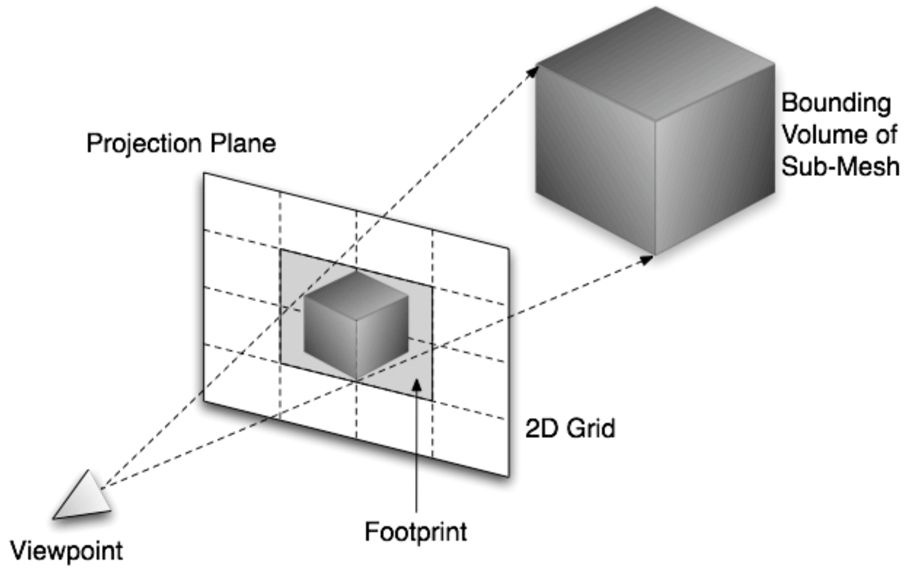


Figure 3.4: Shows the method for calculating the footprint of a fragment.

The most accurate way to compute the footprint of a fragment is to determine exactly the set of pixels it paints when it is facing the camera directly. This approach implies distribution overhead that is proportional to the scene complexity though, and so is too expensive for our purposes. Instead, a coarser, scene-independent approach is used, as follows (see Figure 3.4). The viewport is divided into a grid of cells $W \times H$, where each cell corresponds to a block of pixels[5]. Each frame, the tight rectangular bounding volume of the fragments, is projected onto the 2D grid. The footprint is then estimated as the set of grid cells that the projection overlaps.

Note that calculating the footprint of a fragment is itself a simple rendering problem. The hardware graphics accelerator on any Data Server node or Rendering node can be used to compute these projections efficiently.

[5] In our test cases, we use a 10 × 8 grid to represent a 640 × 512 pixel viewport.

3.2.3 | Context Aware Mesh Partitioning Algorithm

Context Aware Mesh Partitioning (CAMP) is an algorithm, employed by ECOVE to aggregate a scene's fragments into groups called work partitions. *Note that CAMP has not been been fully implemented for the experiments conducted in Chapter 4. As such this thesis does not provide a proof of the algorithm discussed in this section and will be left for future work instead.* At the beginning of the first frame, CAMP can be used by the Data Server to do an initial assignment of fragments to the rendering nodes. When a rendering node becomes unavailable for rendering, its corresponding buddy node takes over the rendering workload (see Section 3.4) causing an imbalance in the workload. CAMP, once again, is used by the Data Server to rectify the load imbalance.

Before describing CAMP more precisely and devising a solution, we first consider an essential characteristic of the expected system architecture. As already discussed in Section 3.2.2, the many-to-one communication required at the end of each frame must typically be serialized. Figure 3.5(a) shows that this serialization can cause significant idle time if the transmission time of each image layer is non-trivial compared to the distributed rendering time. To avoid this costly idleness, the communication phase of a frame need to be overlapped with the rendering of the next frame. Figure 3.5(b) shows this overlapping and the resulting performance increase. This overlapping is supported by most hardware accelerators via double buffering (and so is compatible with ECOVE's goal of using hardware graphics accelerators).

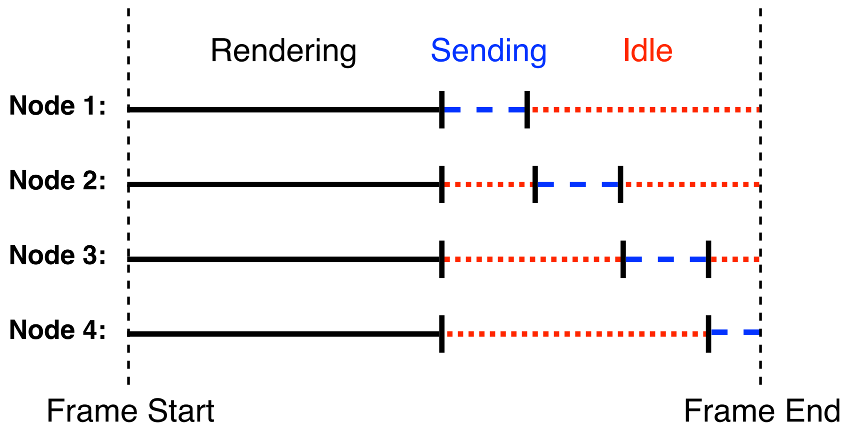


Figure 3.5(a): The distributed rendering of each frame consists of 2 operations, a rendering phase that can be performed in parallel and a communication phase that (typically) must be performed sequentially (each horizontal line represents the timeline of a rendering node).

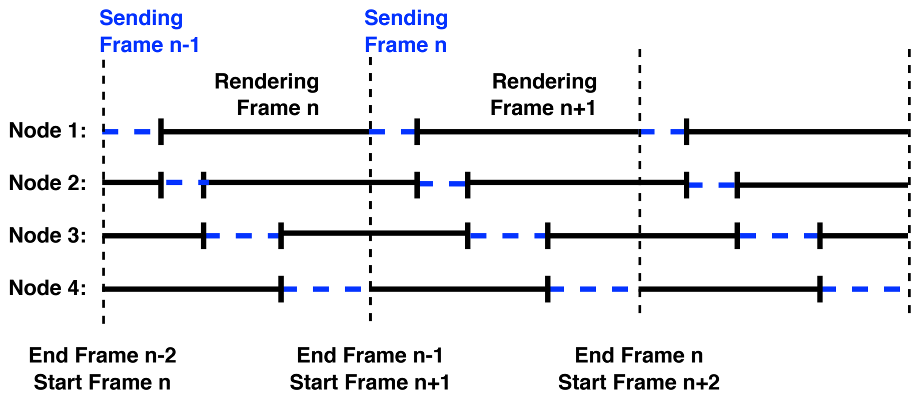


Figure 3.5(b): We can increase the efficiency of distributed rendering by overlapping the rendering operation of a frame with the communication operation of the last frame.

While critical to performance, overlapping communication and computation introduces an additional complexity to the partitioning and assignment problem. At the beginning of each frame, each rendering node is already loaded with the time required to transmit the image layer it generated for the last frame. The total (expected) load on each node for each frame is this transmission time plus the rendering times of its assigned fragments[6]. In addition, the minimum frame time is determined by the larger of the maximum load and the sum of the transmission times for the last frame, where the frame time can now be defined as the time from when the rendering of a frame is initiated until the time when the rendering nodes are ready to transmit the corresponding sub-image layers to the display node. Thus, a Context Aware Mesh Partitioning corresponds to a dual optimization problem: minimize both the load imbalance and the total transmission time.

3.2.3.1 | Problem Definition

The CAMP problem for a frame is stated more precisely as follows. Given:

- ◉ an ECOVE system *ES* with *P* rendering nodes, $p_0, p_1, p_2, ..., p_{P-1}$,
- ◉ a scene *M* with *N* fragments, $o_0, o_1, o_2, ..., o_{N-1}$,
- ◉ $\zeta(o_i)$, the locality of fragment *i* (i.e. centroid of the fragment),
- ◉ parameters *W* and *H* used to logically partition the viewport into a coarse $W \times H$ 2D grid,
- ◉ C_T, the cost to transmit the pixels in one grid cell,
- ◉ *RT(o,v)*, the expected rendering time of each fragment *o* in *M* when viewed from viewpoint *v*,
- ◉ a fragment to grid cells mapping

[6] This assumes that both rendering and networking use the CPU and so cannot be performed concurrently. If either task does not make use of the CPU (e.g., use of a Graphics Processing Unit, GPU, to off-load processing from the CPU), then the load on each node would be the larger of the rendering and the transmission times.

$$FP(o,x,y,v)=\begin{cases}1 & \textit{if cell}(x,y)\textit{ is in o's} \\ & \textit{footpr}\text{int}\textit{ when viewed} \\ & \textit{from viewpo}\text{int}\ v \\ & \\ 0 & \textit{otherwise}\end{cases},o\in M,\ 0\le x<W,0\le y<H,$$

Let S denote a subset of M with Q fragments and define:

◉ the locality of S as:

$$\zeta(S)=\frac{\sum_{o\in S}\zeta(o)}{Q}\ ,$$

◉ the expected rendering time of S as:

$$RT(\mathrm{S},v)=\sum_{o\in \mathrm{S}}RT(o,v)\ ,$$

◉ the mapping of S to grid cells as:

$$FP(\mathrm{S},x,y,v)=\cup\ FP(o,x,y,v)\quad \textit{where}\ o\in \mathrm{S},\ 0\le x<W,\ 0\le y<H,$$

and

◉ the size of S's footprint:

$$FPS(S,v) = \sum_{0 \le x < W} \sum_{0 \le y < H} FP(S,x,y,v)$$

Finally, let $Part_P = \{S_0, S_1, ..., S_{P-1}\}$ be a P-way partition of M. Define:

◉ the load corresponding to S_i as

$$L(S_i) = RT(S_i,v) + C_T FPS(S_i,v)\,,$$

and

◉ a cost function for $Part_P$ as

$$Cost(Part_P) = max\begin{cases} max_{0 \le i < P}\, L(S_i) \\ C_T \sum_{0 \le i < P} FPS(S_i\,,v) \end{cases} \qquad (eq\ 3.1)$$

The problem then is to find a partition, $Part_{P,best} = \{S_{0,best}, S_{1,best}, ..., S_{P-1,best}\}$, of M such that

◉ **Locality Criterion:** there does not exist $o_x \in S_{i,best}$ where

$$\|\zeta(o_x) - \zeta(S_{j,best})\| < \|\zeta(o_x) - \zeta(S_{i,best})\| \quad for\ i \neq j\,,$$

and

◉ **Optimization Criterion:** $Cost(Part_{P,best}) \leq Cost(Part_P)$ for all partitions that $Part_P$ satisfy the locality criterion.

The locality criteria ensure that the fragments of the chosen partition are clustered together. This is required for estimating the footprint of the partition (see Section 3.2.2). The optimization criterion attempts to minimize the larger of the maximum load placed on any node (the first component of $Cost(Part_P)$) and the total transmission time (the second component of $Cost(Part_P)$), which affects the completion time of the next frame. The intuition for including the latter component is that, while CAMP attempts to minimize the completion time of the first frame after it has distributed the work partitions, it should not do it at the expense of the completion time of the subsequent frames (by pre-loading the nodes with overly large transmission times).

Thus based on Equation 3.1, the cost of a chosen partitioning for frame f is not necessarily equal to the frame time. Rather, the frame time T_f for frame f is given by

$$T_f = max \begin{cases} max_{0 \leq i < P} \; L(S_{i,f}) \\ C_T \sum\limits_{0 \leq i < P} FPS(S_{i,f\text{-}1}, v) \end{cases} \qquad (eq\ 3.2)$$

where the second component of T_f is the sum of the transmission time for the previous frame (frame f - 1). This component is not included in the cost function because there is nothing that CAMP can do in frame f to lessen the transmission cost arising from frame f - 1.

3.2.3.2 | Work Partitioning Solution

CAMP is implemented as a two-part algorithm. The first part finds a set of vectors (i.e. cluster centers) using the *k*-means clustering technique [3, 5, 12] and locality of all the fragments. This is to allow each rendering node to be assigned to some initial set of fragments. *k*-means clustering also ensures that each fragment will be classified to only one rendering node, and thus will be rendered only once. The second part makes corrections in the workload distribution where *k*-means algorithm either overloaded or under-loaded individual sets.

For a system of *P* rendering nodes, *P* number of cluster centers, μ_1, μ_2, ..., μ_P, are randomly selected from the set of fragments in the scene as the initial set of cluster centers (or vectors). The Locality Criterion, presented in the previous section, determines how to refine these cluster centers. The algorithm for this phase is given as follows:

Algorithm 3.1: *k*-means Clustering for first part of Work Partitioning

```
begin initialize n, P, $\mu_1, \mu_2, ..., \mu_P$
        do classify n fragments according to nearest $\mu_i$
                recompute $\mu_i$
        until no change in $\mu_i$
        return $\mu_1, \mu_2, ..., \mu_P$
end
```

Given the partition, $Part_{P,LC} = \{S_{0,LC}, S_{1,LC}, ..., S_{P,LC}\}$, produced by using the Locality Criterion, CAMP will make a second pass if the cost is determined by the load component - that is, if $Cost(Part_{P,LC}) = \max_{0 \leq i < P} L(S_{i,LC})$. To correct this imbalance, CAMP employs an iterative approach. As outlined in the listing for Algorithm 3.2 below, each iteration attempts to re-classify a fragment to the next nearest cluster center. This re-classification is an exhaustive search process whereby the distances of every fragment in the partition with respect to the rest of the cluster centers, is calculated. The fragment with the shortest distance with a cluster center (or partition) is moved to new corresponding partition. This iteration process is aborted if the Optimization Criterion is violated.

Algorithm 3.2: Re-classifying a Fragment.

```
begin $\{o_1, o_2, ..., o_N\}, \{\mu_1, \mu_2, ..., \mu_P\}$
    do find shortest distance from every o to every $\mu$
        set shortest distance pair $o_i - \mu_j$
    do find shortest distance pair in all $o_i - \mu_j$
    do classify $o_i$ to $\mu_j$
end
```

3.3 | Distributed Rendering and Frame Composition

The rendering pipeline in ECOVE (modeled after the Sort-Last distributed rendering technique) begins with the decomposition of a 3D scene or mesh into sub-meshes. These sub-meshes are then distributed across the various Rendering Agents in the network (see Section 3.2.3). For every frame, each rendering agent renders a complete image of the data it has been assigned to, using its local GPU. Then it reads back the contents of the frame buffer from the GPU to main memory as a sub-image. This sub-image is sent to a compositor node where a parallel image compositing step is performed to blend all the full resolution sub-images into the final frame image; this step intensively uses the interconnection network to transfer pixel data from the rendering agents to the compositor. Finally, the composite image is written to the frame buffer of the GPU on the intended machine. We propose the rendering pipeline performance, *PF*, to be expressed as follows:

$$PF = DS + RT + CLT + CMP \qquad (eq\ 3.3)$$

where *DS* is the time required to dispatch scene settings (or animation information) to all the rendering agents; *RT* is the time required to render a portion of the scene by a rendering agent, and read back the color and depth information of the rendered sub-image; *CLT* is the time required to collect *n* sub-images to compose the final image; and *CMP* is the time required to merge all the sub-images together to form the final image for display or storage to disk.

3.3.1 | Dispatch Scene Settings

To render an animation sequence, the data server (see Figure 3.2) needs to broadcast the animation information about the 3D mesh to all the rendering agents. This information can be in the form of the angle of rotation or the entire transformation matrix. If it is assumed that multicast is not used to broadcast the animation information, then the time required to dispatch this information to *n* rendering agents can be expressed as:

$$DS = \sum_{i}^{n}\left[ls_i + \frac{\delta}{bnet_i}\right] \qquad (eq\ 3.4)$$

where ls_i is the network latency between the data server and the rendering agent *i*; δ is the size of the transformation matrix in terms of bytes; and $bnet_i$ is the network bandwidth for transferring information over the network from the data server to the rendering agent *i*.

For discussion purposes, if we assume that the latency *ls* is negligible and the *bnet* is the same for all rendering agents, then equation 3.4 can be simplified as follows:

$$DS = \frac{\delta \times n}{bnet} \qquad (eq\ 3.5)$$

In other words, the time required to dispatch the animation information for each frame is directly dependent on the number of rendering agents and the amount of information transferred to each of the agents.

3.3.2 | RENDER

We define the term *RT* as the time required for both rendering a scene and reading it from the frame buffer to main memory. If rendering of a scene is measured as the number of frames per second, *fps*, at a given resolution of *x* by *y* pixels, then the rendering performance can be expressed as:

$$RT = \frac{1}{fps_{xy}} + \left(lr_c + \frac{xy \times bpp_C}{br_C}\right) + \left(lr_z + \frac{xy \times bpp_Z}{br_Z}\right) \qquad (eq\ 3.6)$$

where *lr* refers to the latency in terms of reading either the color information or the depth information (i.e. the z-buffer) from the frame buffer; bpp_c and bpp_z refers to the bits per pixel of color and depth information respectively; and *br* is the bandwidth in bits per second of the GPU operation. The first component of the equation refers to the time taken to render each frame on a rendering agent. The second and third components formulates the time required to read the color and depth information from the graphics card, respectively.

For most parts, equation 3.6 holds true for our proposed solution, except that our solution does not need to read back the z-buffer information. As such the third component of equation 3.6 can be eliminated (see Equation 3.7). This shows that the performance of the rendering agents in our solution is dependent on the time required to read only the color information from the frame buffer.

$$RT = \frac{1}{fps_{xy}} + \left(xy \times \frac{bpp_C}{br_C}\right) \qquad (eq\ 3.7)$$

Equation 3.7 assumes that the latency *lr* is negligible when reading large buffers. Suppose for a target frame rate of 25 *fps*, the sub-image resolution *xy* is set at 1024 by

768 pixels with 32-bit color information and 32-bit z-buffer; the bandwidth br_c for the reading of color information is about 6.9G/s; and the bandwidth br_z for the reading of depth information is about 2.4G/s [72]. With this setup, a rendering agent in a classical sort-last system would yield a *RT* of 0.054 secs while a rendering agent in our solution would yield a *RT* of 0.044 secs. As such it is possible to achieve about 20% performance improvement at this stage.

3.3.3 | COLLECT

We define the term *CLT* (in equation 3.3) as the time required for all sub-images to be sent across the network to the compositor. This also takes into account the time required to compress the sub-image at the rendering agent and uncompress it at compositor. Thus, *CLT* of a classical sort-last system can be expressed as follows:

$$CLT = \left(lcpr + \frac{xy \times bpp_C}{bcpr}\right) + \sum_{i}^{n} \left\{ \left[ls_i + \frac{\zeta_i + \left(xy \times bpp_z\right)}{bnet_i} \right] + \left[ldcpr + \frac{\zeta_i}{bdcpr} \right] \right\} \qquad (eq\ 3.8)$$

where *lcpr* and *ldcpr* refer to the latency in the compression and decompression of a sub-image respectively; *bcpr* and *bdcpr* refer to the bandwidth in bits per second to compress and decompress the sub-image respectively; *bnet* is the network bandwidth; ls_i refers to the latency incurred by a rendering agent *i* when sending (or receiving) the sub-image across the network to a compositor; and ζ_i is the size of the compressed sub-image from a rendering agent *i*. Here we are assuming that only the color information of sub-image can be compressed using convectional techniques like RLE compression. Thus, the first component of the equation is the time required for compression of the color information of a sub-image. Unlike the compression process that can be done in parallel on all the rendering agents, the transferring of the sub-images (second component of Equation 3.8) and uncompressing them (third component of Equation 3.8) can only be done serially on the compositor node.

For discussion purposes, we would like to keep the network load a constant by not implementing compression and assuming that on a high speed network where the network latency, ls_i is assumed to be negligible, we can simplify Equation 3.8 as follows:

$$CLT = \frac{n \times xy \times (bpp_C + bpp_Z)}{bnet_i} \qquad (eq\ 3.9)$$

For a 32-bit bpp_c and bpp_z, Equation 3.9 shows that our solution can achieve up to 50% performance improvement over the classical sort-last algorithm since there will no z-buffer information transferred between the rendering agents and the compositor node. That is, it only takes half the time required to collect n sub-images from n rendering agents.

3.3.4 | Composition

The *CMP* term in equation 3.3 refers to image composition operation. Once all the sub-images are collected as specified in the previous section, the composite image is determined by sorting the pixel depths. A common technique is to fill a pixel on the composite image with the pixel on a sub-image that is closest to the screen (i.e. least in depth). The following equation shows cost of sorting the depth information for each pixel.

$$CMP = lcmp + \sum_{i}^{n} \frac{xy \times (bpp_C + bpp_Z)}{bcmp} \qquad (eq\ 3.10)$$

where *lcmp* is the latency incurred during composition operation; *bcmp* is the bandwidth required for blending of a source pixel over a target pixel. Typically the image blending option "over" is used to blend a new sub-image on top of the final composite image.

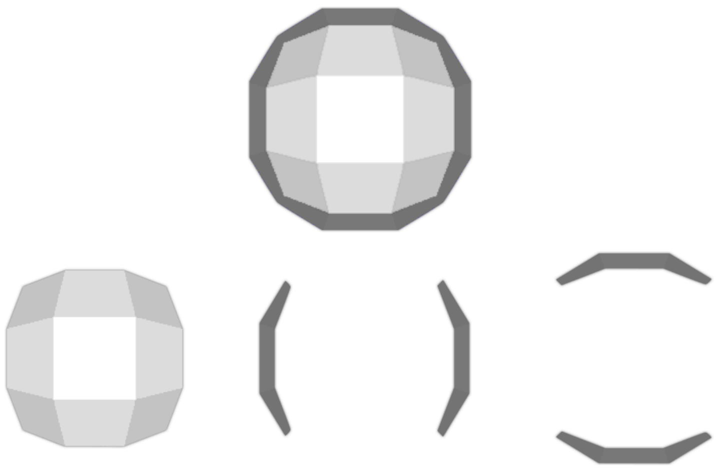


Figure 3.6: Shows that the distributed rendering of a 3D mesh across 3 different rendering agents. The final composite image is simply pieced together like a jigsaw puzzle, Thus, eliminating the need for z-buffer for image composition

Unlike the classical sort-last algorithm, our proposed solution would consider bpp_z to be 0 since there is no need for z-buffer comparison. This is because each sub-image is rendered by a rendering agent whose polygon set has similar orientations. Also the fragment masks (see Section 2.2.1.3) removes regions from the sub-image that would overlap with fragments from other sub-images. Therefore, there is no need to compare depth information between sub-images. Instead, the sub-images fit onto the final composite image like a "jigsaw puzzle" (see Figure 3.6).

$$CMP = \frac{n \times xy \times (bpp_C + bpp_Z)}{bcmp} \qquad (eq\ 3.11)$$

If *lcmp* is assumed to be negligible, then Equation 3.10 can be simplified as above. Equation 3.11 shows that cost of composition for our solution depends only on the cost of writing a pixel from a sub-image to the composite image. This gives us up to 50% performance gain over the classical sort-last algorithm.

3.3.5 | Overall Performance

To illustrate a theoretical performance comparison between our solution and the classical sort-last algorithm, we will assume the running of a rendering application on an ideal Commodity Off-The-Shelf (COTS) cluster with no latencies. Then the time to render an image of $x \times y$ pixels from Equation 3.6 can be re-expressed as follows based on Equations 3.4, 3.6, 3.8 and 3.10:

$$PF = \frac{\delta \times n}{bnet} + \frac{1}{fps_{xy}} + xy \times \left(\frac{bpp_C}{br_C} + \frac{bpp_Z}{br_Z}\right) + \left[n \times xy \times (bpp_c + bpp_z) \times \left(\frac{1}{bnet} + \frac{1}{bcmp}\right)\right]$$

(*eq* 3.12)

Likewise by eliminating the z-buffer components of Equation 3.12, we obtain a simplified equation for our solution as shown below, which is consistent with Equations 3.5, 3.7, 3.9 and 3.11:

$$PF = \frac{\delta \times n}{bnet} + \frac{1}{fps_{xy}} + xy \times bpp_c \times \left[\frac{1}{br_C} + n \times \left(\frac{1}{bnet} + \frac{1}{bcmp}\right)\right] \qquad (eq\ 3.13)$$

An ideal COTS cluster will be equipped with 3 GHz processors (provides a *bcmp* of 4 G/s [72]), full duplex Gigabit Ethernet with infinite aggregate bandwidth, and AGP x8 graphics. Let us assume only the angle of rotation is broadcast for each frame (8 bytes). Then for a target frame rate of 25 *fps* on a 6 node cluster with a 32-bit RGBA pixel format and 32-bit z-buffer, the our modified sort-last algorithm should yield a theoretical 20% rendering performance gain over the classical sort-last algorithm.

In typical parallel rendering systems, the rendering times overlap with the dispatching of the animation information and the collection of rendered partial images (see Figure 3.5). Assuming that rendering times for each frame on each rendering agent is removed from Equations 3.12 and 3.13, we obtain a theoretical, maximum performance gain of 50%.

3.4 | Discovering, Rendering and Monitoring

As mentioned in the beginning of this chapter, ECOVE adopts the conventional distributed rendering pipeline: load a 3D scene; create work partitions, distribute the workloads to various rendering nodes; and finally compositing the sub-image layers from these nodes for display. One of the goals of ECOVE, however, is to reduce the reliance on a centralized server and thus, P2P technology is used to take some of load of the Data Server node.

P2P is particularly used for discovering and monitoring of rendering nodes, and collaboratively rendering a 3D scene.

1. **Discovery**

   Using the P2P technology called Zero Configuration Networking (see Appendix A), discovery of the services of individual nodes can be dynamic. For example, rather than keeping a static list of the IP addresses of all the rendering nodes, ECOVE can generate a dynamic list of all the available rendering nodes. The Data Server advertises itself for the available Rendering Nodes to contact it. This thus, promotes dynamism and event based processing.

   Upon initial contact, the Data Server sends the node a Peer ID. This is a unique number for the node throughout the system and is in running order. That is, the first node will be tagged as *1* and the second node as *2* and so on.

   At the same time, the Data Server issues the node with its owner set (i.e. the set of fragments that the node will be responsible for rendering). The node will also receive a copy of the simplified fragments of the 3D scene’s mesh objects.

2. Assignment of Buddy

Every node in ECOVE will have a buddy node that it needs to monitor for presence. This assignment is issued by the Data Server after a node has received its owner set and pseudo meshes. The Data Server adopts a *cyclic assignment* method. For example, a node with Peer ID 2 will be assigned a buddy node with Peer ID 3. As such the last node in the list will be assigned to monitor the node with Peer ID 1.

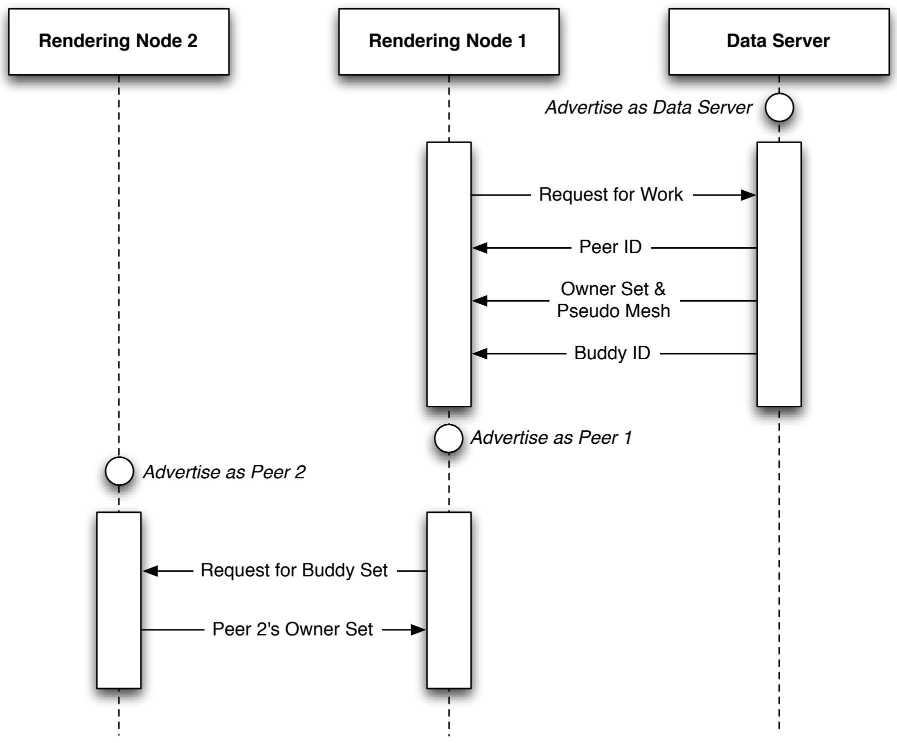


Figure 3.7: Shows the process of how a rendering node interacts with the Data Server and its peers to get the all the required fragment sets.

Once a node has been assigned its buddy node, it advertises it Peer ID and waits for its buddy node to advertise itself. Upon noticing the advertisement of the buddy node, it requests the buddy node to send its owner set and store the set as the buddy set. Rendering of the first frame can begin once the first node (i.e. the node with Peer ID 1) has updated the last node of its owner set.

3. **Collaborative Rendering**

At the start of each frame, the first node will receive the viewpoint settings from the Data Server. Once the node has finished rendering the frame, it will inform its buddy node to start rendering while the first node sends its rendered sub-image layer to the Compositor. This continues as a chain-reaction till the last node has rendered the frame and prompts the first node to start rendering the next frame. This works in accordance to the solution depicted in Figure 3.5(b) where the rendering process of one node is overlapped with transmission of the sub-image layer of another node.

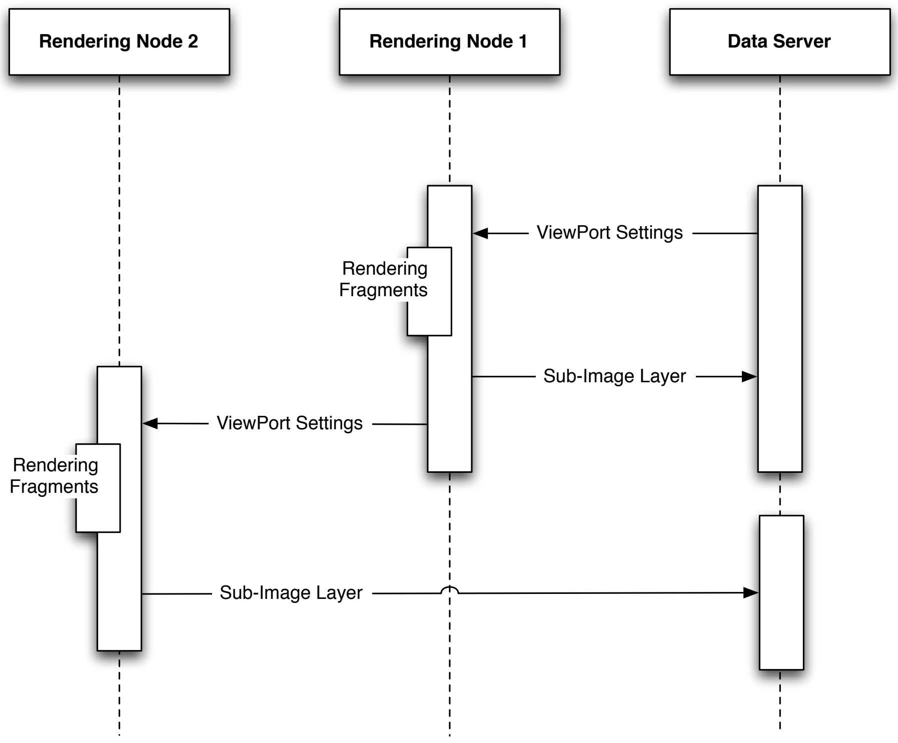


Figure 3.8: Shows the process of how rendering nodes collaboratively render a scene.

4. **Monitoring**

In the event a rendering node drops out of the network, there will not be anymore advertisements with that node’s Peer ID. As a result, the node monitoring for these advertisements will realize that its buddy node is down. It will first inform the Data Server of the change in the number of the rendering nodes and then include the buddy set of fragments into its workload. This results in an imbalance in workload and thus, the Data Server uses the CAMP algorithm once again to redistribute the workload to the rendering nodes.

3.5 | SUMMARY

The three key focuses of ECOVE's architecture were presented in this chapter and they are namely: CAMP, Rendering Performance, and P2P based discovery and runtime monitoring. CAMP treats each fragment (obtained from Chapter 2) as a unit of work and looks at how to optimally group them based on the available number of rendering nodes (or agents).

Once the groups of fragments are distributed, we analyze the rendering performance of ECOVE against a classical sort-last rendering system. The analysis showed that by eliminating the need for depth information throughout the rendering pipeline, we should be able to obtain about 20 to 50% performance gain. These results will be compared with the experimental results obtained in Chapter 4.

Finally this chapter looks at how P2P could be used to discover rendering nodes for distribution of mesh and for monitoring the availability of each rendering node. The latter is essential for continuity of a rendering cycle in the event a rendering node drops out of the network.

CHAPTER 4

4 | IMPLEMENTATION AND RESULTS

ParaView [53] is an open-source, multi-platform parallel visualization application. This application is designed to visualize data sets of size varying from small to very large. ParaView runs on distributed and shared memory parallel as well as single processor systems. For distributed rendering, ParaView employs a sort-last approach.

While both ECOVE and ParaView are both implementing the sort-last method, the difference, however, is in the way ECOVE partitions its 3D scene's mesh objects. Using Ellipsoidal Mesh Partitioning technique (see Chapter 2), ECOVE breaks up a 3D mesh model into fragments that have similarly orientated and connected polygons. Thus, each fragment is generally featureless and can be simplified without too much loss of detail. Context Aware Mesh Partitioning (see Chapter 3) is employed to group these fragments into work partitions taking into account the number of available rendering nodes and the locality of each fragment. ParaView, on the other hand, partitions the mesh model on a First-Come-First-Serve basis. That is, if there are *n* rendering nodes, ParaView divides the list of polygons into *n* equal partitions. Each partition might contain polygons from various parts of the mesh. These partitions are not broken along the geometric features of the mesh model. Also just as in a typical sort-last implementation, ParaView sends the entire rendered image with its depth buffer. ECOVE on the other hand, attempts to reduce the communication overheads by only transmitting the foot-print of the rendered polygons (see Section 3.2.2).

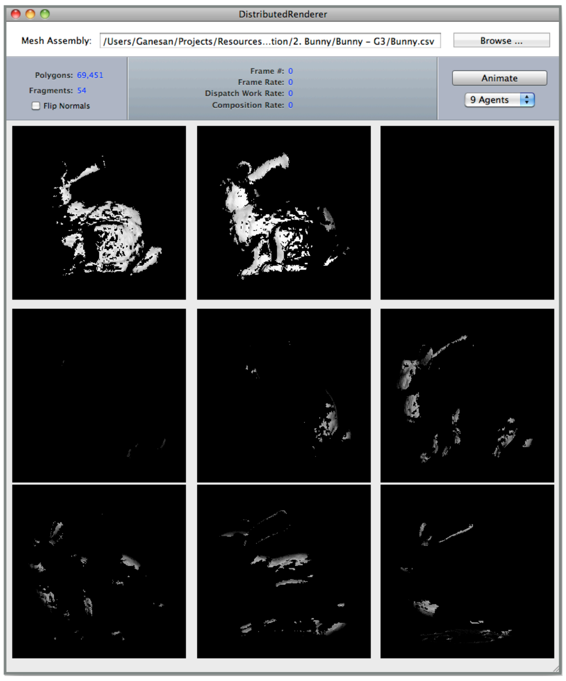
The next section looks at how both the ECOVE and a sort-last implementation (similar to ParaView's mesh partitioning technique), are implemented to render several 3D mesh models. The section following that will compare the results from the implementation to evaluate ECOVE's performance.

4.1 | Implementation

To assess ECOVE's performance, we measure its ability to:

1. achieve faster overall rendering pipeline performance over classical sort-last algorithm, and
2. limit the growth in bandwidth as the number of nodes in the system grows.

Thus to evaluate ECOVE, we have implemented both the classical and our modified sort-last algorithm as the software application shown in Figure 4.1.



Mesh Assembly: /Users/Ganesan/Projects/Resources...tion/2. Bunny/Bunny – G3/Bunny.csv

Polygons: 69,451
Fragments: 54
Flip Normals

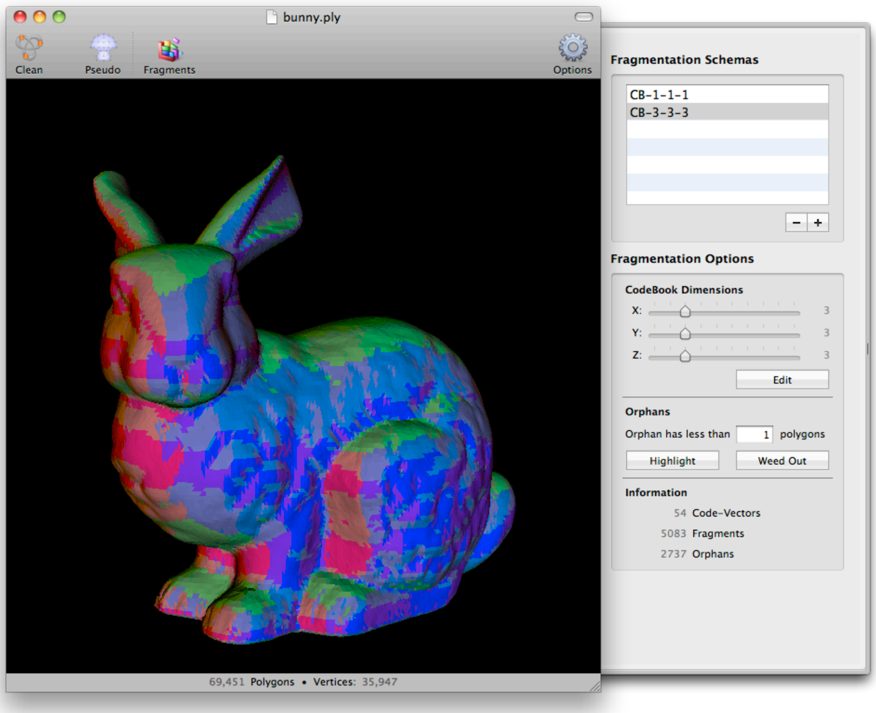
Frame #: 0
Frame Rate: 0
Dispatch Work Rate: 0
Composition Rate: 0

9 Agents

Figure 4.1: Shows the custom created application called Distributed Rendering Server.

This application called *Distributed Rendering Server* is implemented using the Visualization Toolkit [20] framework and provides information such as the achieved frame times for a number of traces. We ran our tests on a 9-node (2.26GHz Intel Core 2 Duo, NVIDIA GeForce 9400M) cluster with Gigabit Ethernet interconnect.

The 3D mesh models used in this experiment are fragmented with various Ellipsoidal schemas using an offline application as shown in Figure 4.2. This application called *Dihedral* accepts 3D mesh models either in PLY or OBJ file formats. It provides options to manipulate the resolution of a schema and apply the schema to a 3D model for mesh partitioning. The results are displayed with distinct colors for each code-vector. Other statistics such as the code-book size used, number of fragments generated and the performance of a mesh partitioning operation are also captured by Dihedral. The application also provides an option to export the fragmented mesh together with the simplified one to file for use by the Distributed Renderer.



Fragmentation Schemas

CB-1-1-1
CB-3-3-3

Fragmentation Options

CodeBook Dimensions

X: 3
Y: 3
Z: 3

Orphans

Orphan has less than 1 polygons

Information

54 Code-Vectors
5083 Fragments
2737 Orphans

69,451 Polygons • Vertices: 35,947

Figure 4.2: Shows the Dihedral application that is used for the fragmentation of a mesh.

When the Distributed Rendering Server application (see Figure 4.1) is launched, it assumes the role of a data server. A user can then begin by selecting a file containing all the fragments of a 3D mesh. The application then begins to search for a list of rendering agents. These rendering agents will run the Distributed Rendering Client application, which can communicate with a data server and amongst other rendering agents. A user can artificially control the number of rendering agents to focus on. If not, the application will default the maximum number of available rendering agents.

As each rendering agent renders and sends the sub-images to the Distributed Rendering Server, the server application assumes the role of an image compositor. The application displays all the sub-images (maximum of 9) and composited images. An animate button is provided to add additional complexity. The parameters for the animation are sent by the server application to the clients.

4.2 | RESULTS

4.2.1 | FRAGMENTATION

We have performed several runs of distributed polygonal rendering using both the classical sort-last algorithm and our modified sort-last algorithm. For these runs, we have used five unique 3D meshes (see Appendix B) that were subjected to our mesh partition algorithm. Table 4.1 shows the polygon count for each of these meshes.

Table 4.1: Polygon counts of the 3D meshes used in the experiment

Gaea-5	Bunny	Extinguisher	Dragon	Blade
300	69,451	300,572	871,414	1,765,388

4.2.1.1 | DENSITY DISTRIBUTION

As shown in Tables 4.2 through 4.5, these meshes were fragmented using three different Uniform Ellipsoidal Schemas and one Non-uniform Ellipsoidal Schema. For each fragmentation of the meshes, the highest, the lowest and the average densities (i.e. the concentration of polygons in one fragment) are shown in these tables.

Table 4.2: Shows the fragment densities using Gaea-1 based Uniform Ellipsoidal Schema

	Gaea-5	Bunny	Exting.	Dragon	Blade
Code-Book Size	6	6	6	6	6
Highest Density (%)	16.67	21.05	21.73	21.21	20.57
Lowest Density (%)	16.67	13.84	14.78	13.83	11.13
Average Density (%)	16.67	16.67	16.67	16.67	16.67

Table 4.3: Shows the fragment densities using Gaea-3 based Uniform Ellipsoidal Schema

	Gaea-5	Bunny	Exting.	Dragon	Blade
Code-Book Size	54	54	54	54	54
Highest Density (%)	2.67	6.89	11.49	3.25	10.10
Lowest Density (%)	0.67	0.58	0.55	0.76	0.22
Average Density (%)	1.85	1.85	1.85	1.85	1.85

Table 4.4: Shows the fragment densities using Gaea-4 based Uniform Ellipsoidal Schema

	Gaea-5	Bunny	Exting.	Dragon	Blade
Code-Book Size	96	96	96	96	96
Highest Density (%)	2.67	3.14	4.14	2.58	6.75
Lowest Density (%)	0.67	0.20	0.26	0.42	0.09
Average Density (%)	1.04	1.04	1.04	1.04	1.04

Table 4.5: Shows the fragment densities using Gaea-x,y,z based Uniform Ellipsoidal Schema

	Gaea-5	Bunny	Exting.	Dragon	Blade
GAEA-X,Y,Z	G-3,3,3	G-3,2,4	G-2,3,4	G-4,3,2	G-2,2,5
Code-Book Size	54	52	52	52	48
Highest Density (%)	2.67	4.66	5.03	3.42	8.07
Lowest Density (%)	0.67	0.51	0.59	1.10	0.48
Average Density (%)	1.85	1.92	1.92	1.92	2.08

Based on the results of the fragmentation provided in Tables 4.2 through 4.5, Figures 4.3 and 4.4 show the density distribution graph for the different models used. The graph in Figure 4.3 shows the deviation of the highest density from the average density for each fragmentation schema while the one in Figure 4.4 shows the deviation of the lowest density from the average density.

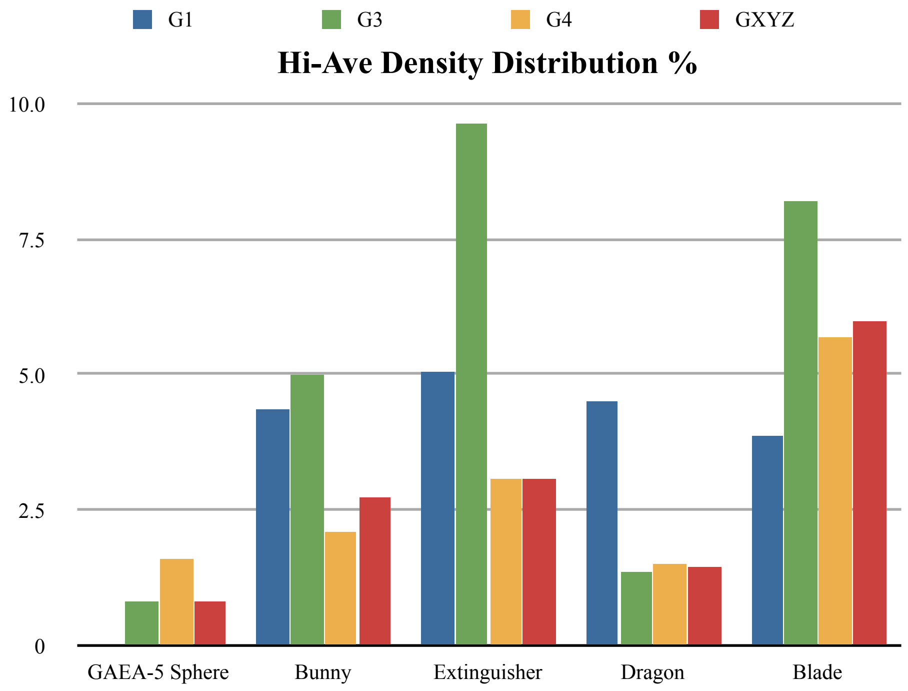


Figure 4.3: Shows deviation of the highest density from the average density of all the 3D meshes for the schemas using Gaea 1. The lower the value

For Gaea-5 Sphere, the deviation of the highest (and even the lowest - see Figure 4.4) density from the average density for Gaea-1 based schema (G1) is actually 0. This means that all the polygons are evenly spread across all the fragments. Thus this is the best partitioning strategy for this model.

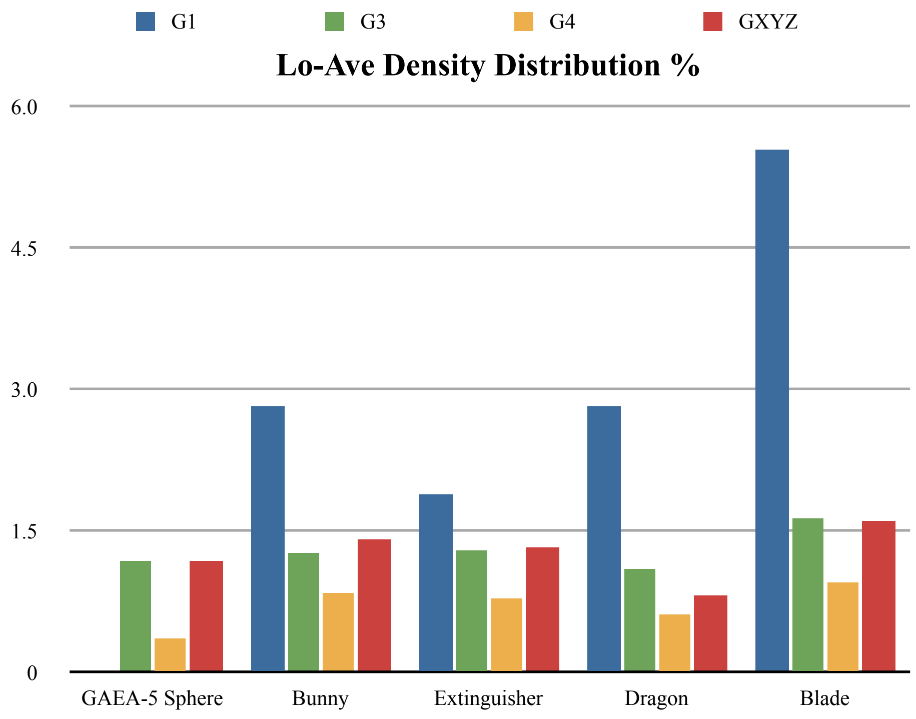


Figure 4.4: Shows deviation of the lowest density from the average density of all the 3D meshes under each schema.

Though G1 uses a smaller code-book than the other schemas, it does not necessarily produce the best polygon distribution. In fact, G4 and GXYZ (the Non-uniform Ellipsoidal Schema) generally produces lower density deviations as compared to G1 for models Bunny, Extinguisher and Dragon. Of the 4 schemas, G3 seems to produce relatively poor polygon distribution, especially for models Bunny, Extinguisher and Blade. While generally the lowest densities are not far off from the average densities for G3, large deviations of the highest densities from the average densities are observed. This shows that more code-vectors need to be introduced to distribute the concentration of polygons.

One option as discussed in Chapter 2, is to increase the resolution of the schema. The results produced by G4 are relatively better. However this is done at the expense of a larger code-book than a G3. The other option is to use a Non-uniform Ellipsoidal Schema like GXYZ. As shown in Figures 4.3 and 4.4, this latter schema has produces results that are comparable to G4’s polygon distribution while maintaining a code-book similar to that of G3’s. This thus shows that a Non-uniform Ellipsoidal Schema can general produce better polygon distribution for mesh partitioning than an equivalent Uniform Ellipsoidal Schema.

4.2.1.2 | Mesh Partitioning Performance

Section 2.1.3 discussed about the time complexity to perform a mesh partition using EMP. The conclusion of the analysis was that the algorithm is *O(MS)*. That is the algorithm’s performance is dependent on the number of polygons and the size of the code-book. Thus in order to ascertain this analysis, Bunny, Extinguisher and Dragon models were subjected to mesh partitioning using Ellipsoidal Schemas based on Gaea-3, Gaea-4, Gaea-5, Gaea-6, Gaea-7 and Gaea-8. This would let us analyze the effects of increasing the code-book size on a 3D model.

Table 4.6: Shows the number of milliseconds taken for mesh partitioning using Uniform Ellipsoidal Schema with increasing resolution.

	G3	**G4**	**G5**	**G6**	**G7**	**G8**
Bunny	1,087	1,382	1,837	2,319	2,870	3,486
Extinguisher	8,757	9,999	11,411	13,093	15,277	17,286
Dragon	13,972	16,528	20,950	26,611	32,943	40,260

When the values in Table 4.6 are plotted, Figure 4.5 reveals a linear increase in the time taken to partition a mesh with increasing schema resolution. However linearity is not observed when it was subjected to Gaea-3 and Gaea-4. This trend is more apparent for the Dragon model that contains the largest number of polygons (about 871,000 polygons) than that of the other 2 models.

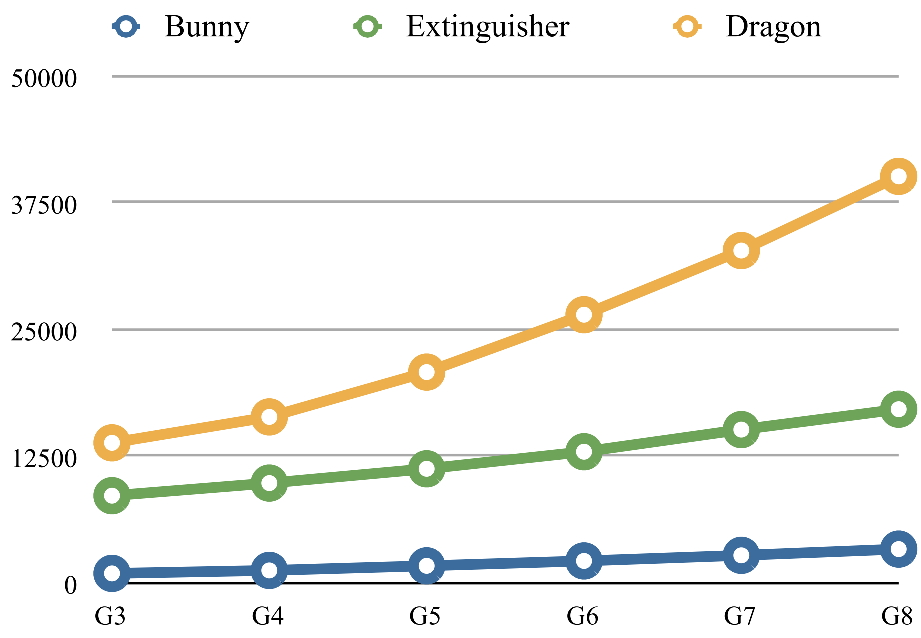


Figure 4.5: Plot of Table 4.6 shows a linear increase in the time taken for mesh partitioning with increasing schema resolution.

Based on observations in Figure 4.5, it would seem that there is a co-relation between the code-book size and the number of polygons in a mesh. To study this co-relation, we look at the situation where all the polygons are equally divided amongst the number of code-vectors in the code-book. As such, we define *R* as the resolution factor that is defined as the ratio of the number of polygons in the mesh to the size of the code-book. Table 4.7 shows *R* for all the models when subjected to the increasing code-book size.

Table 4.7: Shows the resolution factors for the 3 models when subjected to increasing code-book size.

	G3	**G4**	**G5**	**G6**	**G7**	**G8**
Bunny	1E+03	7E+02	5E+02	3E+02	2E+02	2E+02
Extinguisher	6E+03	3E+03	2E+03	1E+03	1E+03	8E+02
Dragon	2E+04	9E+03	6E+03	4E+03	3E+03	2E+03

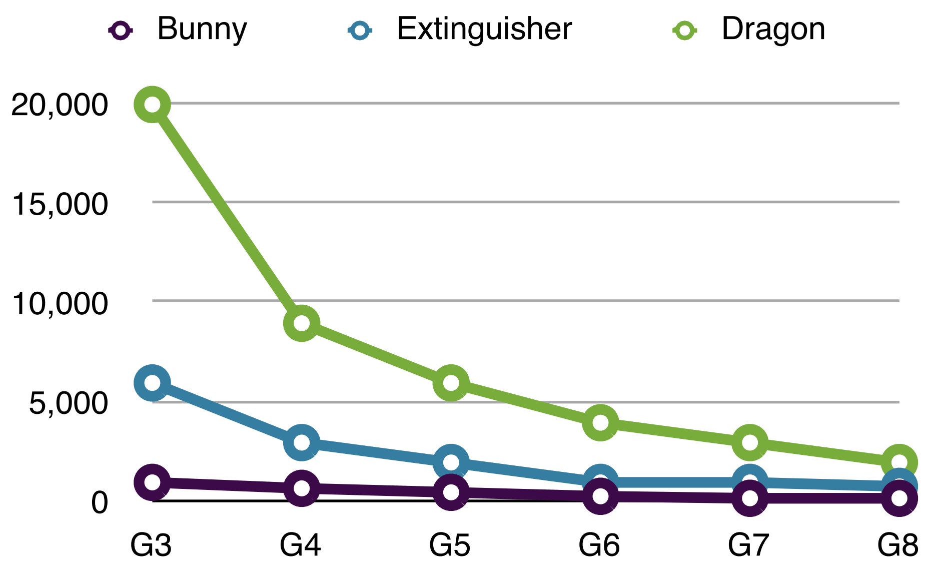


Figure 4.6: Graphical representation of the figures in Table 4.7.

Figure 4.6 shows that the smaller the code-book size, the larger the resolution factor *R* for a model. This is more prominent for the Dragon model when subjected to the Gaea-3 based code-book. Thus when studied with reference to the results in Figure 4.5, it is evident that the performance of Algorithm 2.1 is dependent on the resolution factor, *R*. That is, as *R* for a model tends to 0, the performance of Algorithm 2.1 exhibits linearity.

4.2.2 | Performance Gain over Classical Sort-Last

All the 3D models were rendered using the classical sort-last algorithm. For classical sort-last, the meshes was partitioned by randomly assigning polygons to the required number of work units (i.e. the number of rendering agents). The time taken to render 100 frames using 3 Rendering Agents was recorded. These results were compared against the time taken to render 100 frames using our modified method as shown below in Figure 4.7.

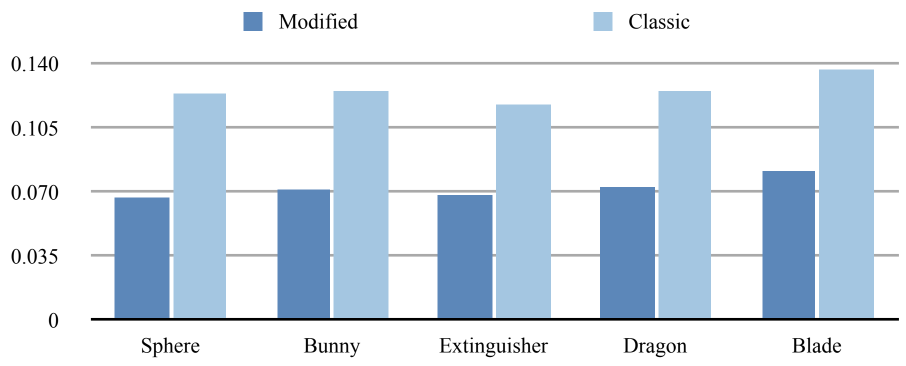


Figure 4.7: Shows the frame times for rendering on a 3-node cluster using our modified and the classical sort-last algorithms

The results are clearly better for our solution, because much less data has to be transferred over the network. The elimination of Z-buffer during the transmission and composition, has improved per frame rendering times. On average we are able to obtain a performance gain of about 40% performance increase by using our modified sort-last algorithm. This result is in accordance to our theoretical performance gain of between 20% to 50% as calculated in Chapter 3. Due to latencies in network transmission of the sub-images, the performance of the system might be capped at 40% gain.

4.2.3 | Rendering Quality

Besides the assessing the performance gain our solution, we have also subjected the quality assessment of the final render image. To perform this analysis, the rendered image produced from the compositor of our solution was compared with the rendered image produced from a standalone renderer. The difference in the analysis is shown in Figure 4.8 below.

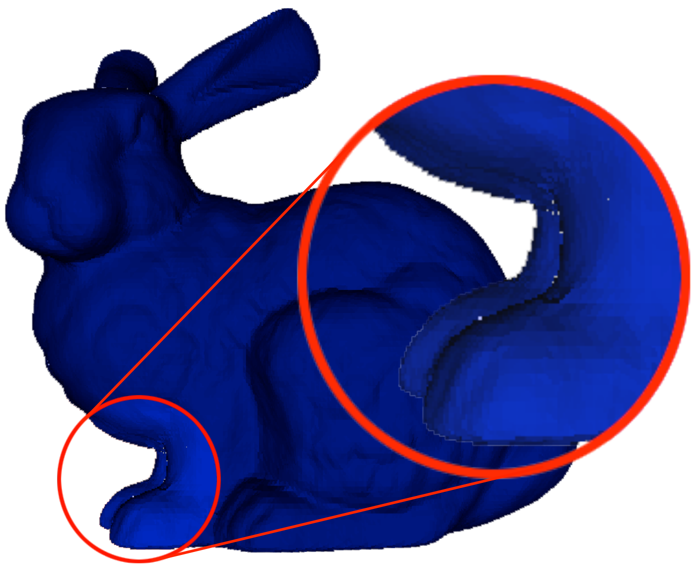


Figure 4.8: Shows the pixel to noise comparison between the bunny model rendered on a single machine vs a the same model rendered using our solution. The noise our results are highlighted as white spots.

The model in Figure 4.8 was subjected to fragmentation using Gaea-3,2,4 and was rendered on a 9 node cluster. The resulting rendered image was compared against a standalone renderer's output and the results of our solution shows high accuracy. The noise as seen in the figure is introduced by fragment masks that are over-flattened. As a result, the final image is masked out at unintended regions. These errors are but far and few. An the areas of unintended masking was is very small. As such the overall of the quality of the image is not impacted.

5 | CONCLUSION AND FUTURE WORK

In this thesis, we have presented techniques to build a collaborative visualization environment called ECOVE that partitions a 3D mesh model to balance the rendering load across all the peers of a P2P network. The basis for ECOVE is our mesh partitioning technique called Ellipsoidal Mesh Partitioning (EMP). This technique partitions a mesh in such a way that the geometric features are preserved. We have shown that this technique can parallelized (i.e. run using a cluster of computers in a P2P network). It has also been demonstrated that the system can be expected to outperform another that uses sort-last, a commonly used partitioning technique, for our targeted environment. Using Context Aware Mesh Partitioning (CAMP), we have shown how to efficiently distribute the rendering load across all the peers of a P2P network.

Implementation of pseudo polygons (or fragment masks) for use in graphics shaders has been left out for this thesis although it has been used for discussion purposes of this thesis. However, ECOVE is being considered as a viable solution for distributed application of graphics shaders such Ambient Occlusion, in the industry at the time of this writing. A full distributed rendering system is being left for future work as there has yet to be system that tackles the issues of applying graphics shaders on distributed 3D mesh models for comparison.

While ECOVE as it is in its present state, has fulfilled its current objectives. However, there is still room for improvement. Currently only static models are supported by ECOVE. That is once the models are loaded and partitioned, they are not modified or morphed to another model. These forms of transformations are common in interactive applications such as games, where a 3D avatar goes through many different transformations. These changes to the model cause the code-book to become invalid. Thus, we need to develop a method to allow code-books to dynamically adapt to changing orientations of the polygons of a 3D mesh.

Also another possibility for future work would be to consider supporting volumetric data models in ECOVE. This domain is becoming increasingly challenging as the amount of data generated by scientific simulations is increasing exponentially. While volumetric data models are significantly different from polygonal models, we believe the concept of Ellipsoidal Mesh Partitioning can help to reduce the complexity of the data models by decomposing them into simpler structures.

APPENDIX A

A | 3D Mesh Models

The following figures show the results of subjecting several 3D mesh models (obtained from http://www.cc.gatech.edu/projects/large_models/) to Ellipsoidal Mesh Partitioning. Each fragment is highlighted using distinct colors.

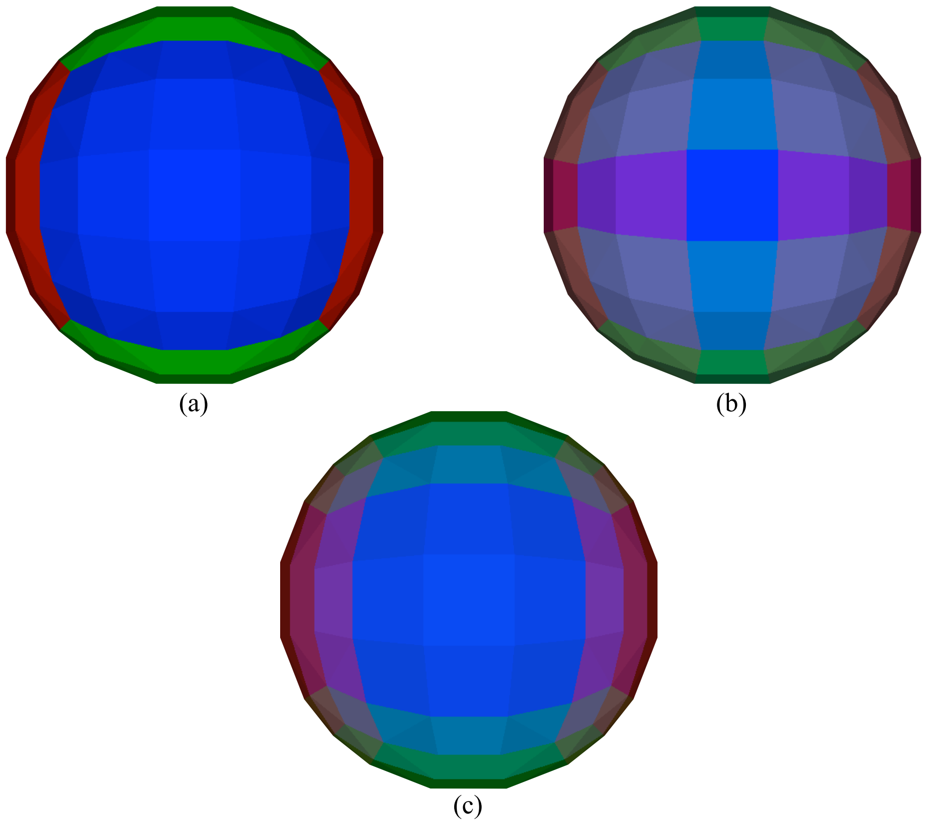


Figure A.1: Gaea-5 Sphere mesh partitioned using (a) Gaea-1 schema (b) Gaea-3 schema (c) Gaea-4 schema.

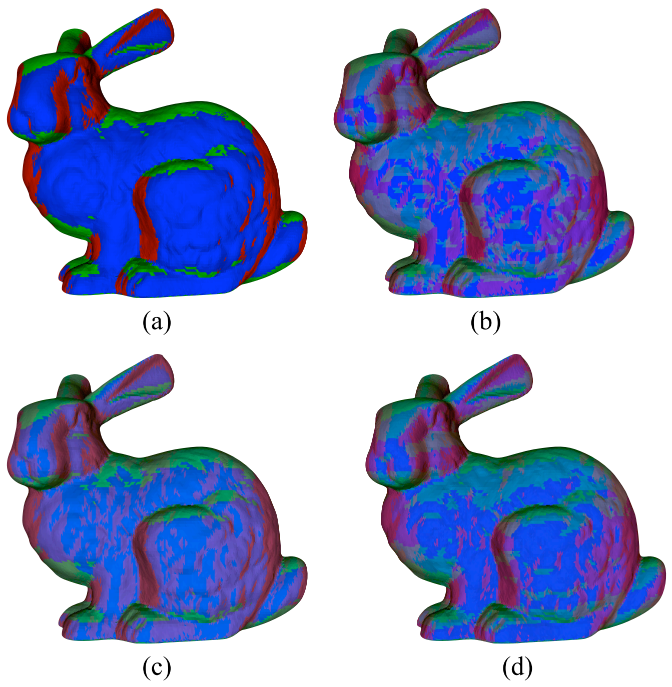


Figure A.2: Stanford Bunny mesh partitioned using (a) Gaea-1 schema (b) Gaea-3 schema (c) Gaea-3,2,4 schema (d) Gaea-4 schema.

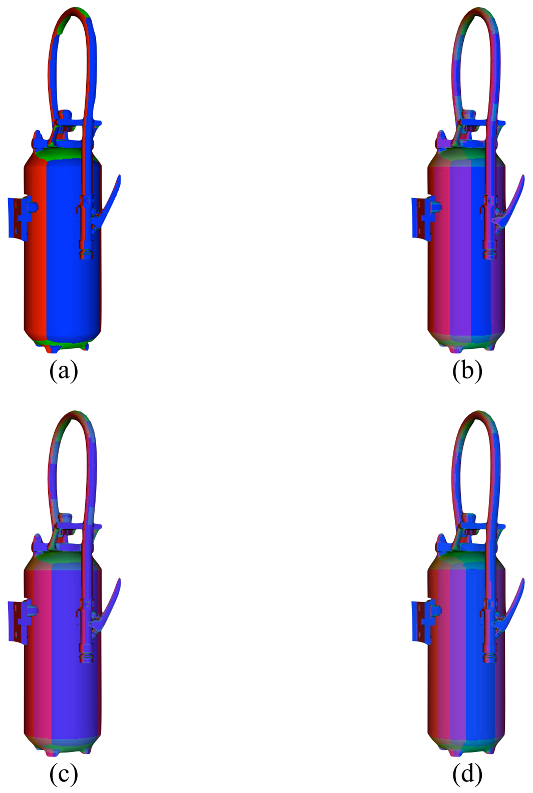


Figure A.3: Fire Extinguisher mesh partitioned using (a) Gaea-1 schema (b) Gaea-3 schema (c) Gaea-2,3,4 schema (d) Gaea-4 schema.

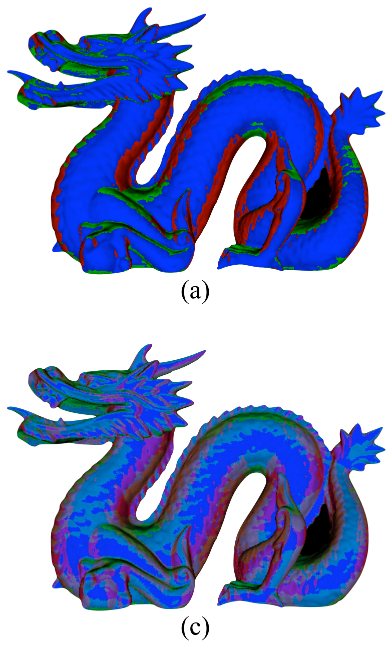


Figure A.4: Dragon mesh partitioned using (a) Gaea-1 schema (b) Gaea-3 schema (c) Gaea-4,3,2 schema (d) Gaea-4 schema.

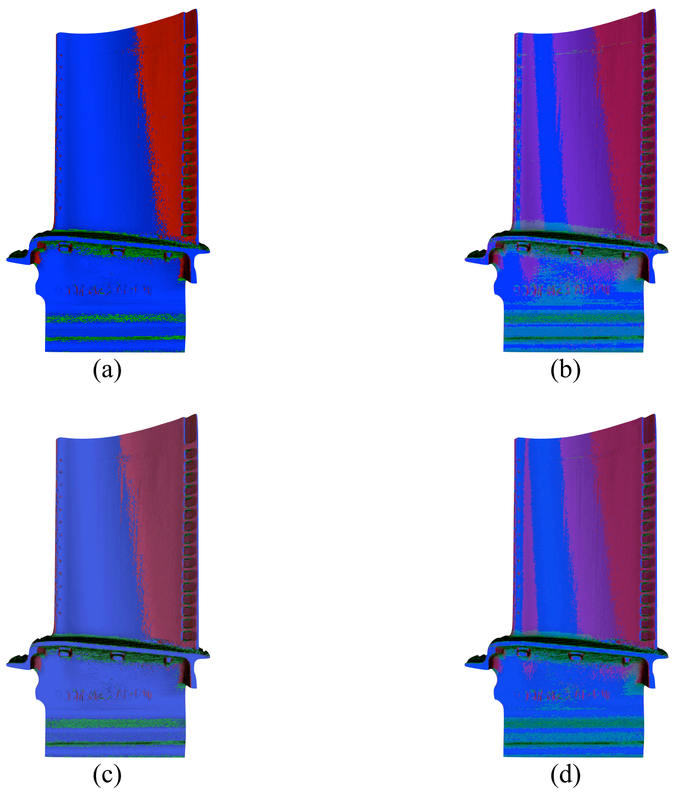


Figure A.5: Blade mesh partitioned using (a) Gaea-1 schema (b) Gaea-3 schema (c) Gaea-4,3,2 schema (d) Gaea-4 schema.

REFERENCES

1. A. Frederico, et al. A load-balancing strategy for sort-first distributed rendering. Proceedings of the Computer Graphics and Image Processing, XVII Brazilian Symposium, 2003.

2. A. Heirich, L. Moll. Scalable Distributed Visualization Using Off-the-Shelf Components. IEEE Parallel Visualization and Graphics Symposium, pp. 55–59, San Francisco, CA., 1999.

3. A. K. Jain, R. C. Dubes. Algorithms for clustering data, Prentice-Hall, Inc., Upper Saddle River, NJ, 1998.

4. A. Leonardis, et al. Superquadrics for segmentation and modeling range data. IEEE Trans Pattern Anal Mach Intell 19:1289–1295, 1997.

5. A. Likas, N. Vlassis, J J. Verbeek. The global k-means clustering algorithm. Pattern Recognition, Volume 36, pp. 451-461, 2003.

6. A. Mangan, R. Whitaker. Partitioning 3D surface meshes using watershed segmentation. IEEE Trans Vis Comput Graph 5(4):308–321, 1999.

7. A. Razdan, M. Bae. A hybrid approach to feature segmentation of 3-dimensional meshes. Computer-Aided Design, 2002.

8. A. Sanna, C. Zunino, L. Ciminiera. A distributed JXTA-based architecture for searching and retrieving solar data. Future Generation Computer Systems, Volume/Issue 21/3, pp. 349-359, ISSN: 0167-739X, 2005.

9. A.J. Cuadros-Vargas, et al, Generating Segmented Quality Meshes from Images. Math Imaging Vis (2009) 33: 11-23, 2009.

10. AH. Barr. Superquadric and angle-preserving transformation. IEEE Trans Comput Graph Appl 1:11–23, 1981.

11. Apple Inc. XGrid. http://www.apple.com/server/macosx/features/xgrid.html

12. C. Ding, X. He. K-means clustering via principal component analysis. ACM International Conference Proceeding Series, 2004.

13. C. Farhat and M. Lesoinne. Automatic partitioning of unstructured meshes for the parallel solution of problems in computational mechanics. International Journal for Numerical Methods in Engineering, 36:745-764, 1993.

14. C. Farhat. A simple and efficient automatic FEM domain decomposer. Computers and Structures, 28:579-602, 1988.

15. C. Farhat. On the mapping of massively parallel processors onto finite element graphs. Computers and Structures, 32:347-354, 1989.

16. C. Mueller. The Sort-First Rendering Architecture for High-Performance Graphics. Interactive 3D Graphics, pages 75–82, Apr. 1995.

17. C. Rossl, et al. Extraction of feature lines on triangulated surfaces using morphological operators. Smart Graphics, AAAI Spring Symposium, Stanford University, pp 71–75, 2002.

18. C. Walshaw, M. Cross, and M. Everett. Mesh partitioning and load-balancing for distributed memory parallel systems. In Proceedings of International Conference on Parallel and Distributed Computing for Computational Mechanics, 1997.

19. C. Xavier, M. Christophe. Pipelined Sort-last Rendering: Scalability, Performance and Beyond. Eurographics Symposium on Parallel Graphics and Visualization, 2006.

20. D. Bartz, D. Staneker, W. Strasser, B. Cripe, T. Gaskins, K. Orton, M. Carter, A. Johannsen, J. Trom. Jupiter: A Toolkit for Interactive Large Model Visualization. Parallel and Large-Data Visualization and Graphics, pp. 129–134, San Diego, CA, 2001.

21. D. N. Thu, Z. John. Image Layer Decomposition for Distributed Real-Time Rendering on Clusters. Parallel and Distributed Processing. pp 421, 2000.

22. D. Salomon. Data Compression, the complete reference 2nd Edition. Springer-Verlag, 2000.

23. D. Taubman, M. Marcellin. JPEG2000: Image Compression Fundamentals, Standards and Practice. Springer-Verlag, 2001.

24. D. Vanderstraeten and R. Keunings. "Optimized partitioning of unstructured computational grids". International Journal for Numerical Methods in Engineering, 38:433-450, 1995.

25. Discreet Inc. http://www.discreet.com/

26. F. Chen, B. Jüttler. 3D Mesh Segmentation Using Mean-Shifted Curvature. Springer-Verlag Berlin Heidelberg, GMP 2008, LNCS 4975, pp. 465–474, 2008.

27. F. Vivodtzev, et al. Hierarchical isosurface segmentation based on discrete curvature. Proc. of VisSym '03, Eurographics—IEEE TVCG symposium on visualization, 2003.

28. G. Humphreys, M. Houston, R. Ng, R. Frank, S. Ahern, P. D. Kirchner, J. T. Klosowski. Chromium: A Stream Processing Framework for Interactive Rendering on Clusters. ACM SIGGRAPH, pp. 693–702, San Antonio, TX, 2002.

29. G. Subramaniam, K. Ong. Mesh Simplification using Ellipsoidal Schema for Isotropic Quantization of Face-Normal Vectors. In Proceedings of International Conference on Asia Pacific Symposium on Information Visualization, 2006.

30. G.S. Oliver, et al. A Survey and Performance Analysis of Software Platforms for Interactive Cluster-Based Multi-Screen Rendering. In Proceedings of the workshop on Virtual environments, 2003.

31. H. Hoppe. View-Dependent Refinement of Progressive Meshes. Computer Graphics, SIGGRAPH '96, 99–108, 1996.

32. H.D. Simon. Partitioning of unstructured meshes for parallel processing. Computing Systems in Engineering, 2:135-148, 1991.

33. H.W. Jensen. A practical guide to global illumination using ray tracing and photon mapping, Proceedings of the conference on SIGGRAPH 2004 course notes, 2004.

34. I. E. Sutherland, R. F. Sproull, and R. A. Schumacker. A Characterization of Ten Hidden Surface Algorithms. ACM Computing Survey, 6(1):1–55, Mar. 1974.

35. I.J. Grimstead, N.J. Avis, DW Walker. RAVE: Resource-Aware Visualization Environment. All-Hands Meeting, 2004

36. J. D. Foley, A. van Dam, S. K. Feiner, J. F. Hughes. Computer Graphics: Principles and Practice. Addison-Wesley, Reading, MA, 1990.

37. J. Pineda. A parallel algorithm for polygon rasterization. Proceedings of the 15th annual conference on Computer Graphics and Interactive Techniques, 1998.

38. J. Van, P. Shi, D. Zhang. Mesh simplification with hierarchical shape analysis and iterative edge contraction. IEEE Transactions on Visualization and Computer Graphics, 2004.

39. J. Ziv, A. Lempel. A universal algorithm for sequential data compression. In Proceedings of IEEE transactions on information theory, 23, 3, 337-343, 1977.

40. J.S. William, et al.. Visualizing with VTK: A Tutorial. IEEE Computer Graphics and Applications, vol. 20, no. 5, pp. 20-27, 2000.

41. K. Moreland, B. Wylie, C. Pavlakos. Sort-Last Parallel Rendering for Viewing Extremely Large Data Sets on Tile Displays. Parallel and Large-Data Visualization and Graphics, pp. 85–92, San Diego, CA, 2001.

42. K. Wu, M.D. Levine. 3D part segmentation using simulated electrical charge distributions. IEEE Trans Pattern Anal Mach Intell 19:1223–1235, 1997.

43. K. Zhou, J. Huang, J. Snyder, X. Liu, H. Bao, B. Guo. Large mesh deformation using the volumetric graph Laplacian. In Proceedings of ACM SIGGRAPH, 2005.

44. L. Chevalier, et al. Segmentation and superquadric modeling of 3D objects. Journal of WSCG 11(1), 2003.

45. L. Shi, Y. Yu, N. Bell, W.W. Feng. A fast multigrid algorithm for mesh deformation. ACM Transactions on Graphics (TOG), 2006.

46. M. Al-Nasra, D.T. Nguyen. An algorithm for domain decomposition in finite element analysis. Computers and Structures, 39:277-289, 1991.

47. M. Bunnell. Dynamic ambient occlusion and indirect lighting. In GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation. Addison-Weseley Professional, 223--233, 2004.

48. M. Garland, et al. Hierarchical face clustering on polygon surfaces. ACM Symposium on Interactive 3D Graphics, 2001.

49. M. Sattler, R. Sarlette, G. Zachmann, R. Klein. Hardware-accelerated ambient occlusion computation. In Vision, Modeling, and Visualization 2004, 331--338, 2004.

50. M.E. Rettmann, et al. Automated sulcal segmentation using watersheds on the cortical surface. NeuroImage (15):329–344, 2002.

51. P. Hodgson, et al. Cluster visualization rises to the seismic processing challenge. SIS Global Forum First Break Volume 23, 2005.

52. P. Wu, E.N. Houstis. Parallel adaptive mesh generation and decomposition. Engineering with Computers, 12:155-167, 1996.

53. ParaView. http://www.paraview.org/

54. R. Samanta, J. Zheng, T. Funkhouser, K. Li, J. P. Singh. Load Balancing for Multi-Projector Rendering Systems. Eurographics/ SIGGRAPH workshop on Graphics hardware, 107-116, 1999.

55. R. Samanta, T. Funkhouser, K. Li, J. P. Singh. Hybrid Sort-First and Sort-Last Parallel Rendering with a Cluster of PCs. In Proc. ACM SIGGRAPH/ Eurographics Workshop on Graphics Hardware, pp. 97–108, Interlaken, Switzerland, 2000.

56. Render-IT. http://www.render-it.co.uk/

57. S. Molnar, et al. A Sorting Classification of Parallel Rendering. IEEE Computer Graphics and Applications, 14(4):23–32, 1994.

58. S. Molnar, J. Eyles, J. Poulton. PixelFlow: High-Speed Rendering Using Image Composition. SIGGRAPH '92, pages 231–240, July 1992.

59. S. Molnar, M. Cox, D. Ellsworth, H. Fuchs. A Sorting Classification of Parallel Rendering. IEEE Computer Graphics and Applications, 14(4):23–32, July 1994.

60. S. Pulla, et al. Improved curvature estimation for watershed segmentation of 3-dimensional Meshes. IEEE Trans Vis Comput Graph, 2002.

61. S. Rudrajit, et al. Hybrid Sort-First and Sort-Last Parallel Rendering with a Cluster of PCs. In Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware, 2001.

62. S.T. Barnard, H.D. Simon. A fast multilevel implementation of recursive spectral bisection for partitioning unstructured problems. In Proceedings of the Sixth SIAM Conference on Parallel Processing for Scientific Computing, pages 711-718, 1993.

63. T Duff. Compositing 3-D rendered images. ACM SIGGRAPH Computer Graphics, 1985

64. T. Apodaca, L. Gritz, M. Pharr, H. Hery, K. Bjorke, L. Treweek. Advanced RenderMan 3: Render Harder. ACM SIGGRAPH, 2001.

65. T. W. Crockett, T. Orloff. A MIMD Rendering Algorithm for Distributed Memory Architectures. ACM SIGGRAPH Symposium on Parallel Rendering, pages 35–42. ACM, Nov. 1993.

66. T. W. Crockett. 1997. An Introduction to Parallel Rendering. Parallel Computing, 23(7):819–843, July 1997.

67. T. W. CROCKETT. An Introduction to Parallel Rendering. Parallel Computing 23, 7, 819-843, 1997.

68. W. Correa, J. T. Klosowski, C. Silva. Out-of-Core Sort-First Parallel Rendering for Cluster-Based Tiled Displays. In Proc. Eurographics Workshop on Parallel Graphics and Visualization, pp. 89–96, Blaubeuren, Germany, 2002.

69. W. Lages, et al. A Parallel Multi-View Rendering Architecture. XXI Brazilian Symposium on Computer Graphics and Image Processing, 2008.

70. W. Leeson, C. O'Sullivan, S. Collins. An Efficient Framework for Implementing Global Illumination. Eighth International Conference in Central Europe on Computer Graphics, Visualization and Interactive Digital Media, 2000.

71. W. Leeson, S. Collins. EFFIGI An Efficient Framework For Implementing Global Illumination. Eighth International Conference in Central Europe onComputer Graphics, Visualization and Interactive Digital Media, 2000.

72. X. Cavin, et al. COTS cluster-based sort-last rendering: Performance evaluation and pipelined implementation. In Proceedings of IEEE Visualization, 2005.

73. Y. Pan, F.T. Marchese. A peer-to-peer collaborative 3D virtual environment for visualization. Visualization and Data Analysis, 2004.

74. Y.F. Hu, R.J. Blake. Numerical experiences with partitioning of unstructured meshes. Parallel Computing, 20:815-829, 1994.