# NOTE TO USERS

# FUNCTIONALITY DISTRIBUTION IN GRAPHICS

**RAMGOPAL RAJAGOPALAN**

A THESIS

IN

THE DEPARTMENT

OF

COMPUTER SCIENCE

AND

SOFTWARE ENGINEERING

PRESENTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF MASTER OF COMPUTER SCIENCE

CONCORDIA UNIVERSITY

MONTRÉAL, QUÉBEC, CANADA

APRIL 2005

*Your file Votre référence*
*ISBN: 0-494-04448-9*
*Our file Notre référence*
*ISBN: 0-494-04448-9*

# Abstract

## Functionality Distribution in Graphics

Ramgopal Rajagopalan

Traditionally graphics clusters have been employed in real-time visualization of large geometric models (many millions of 3D points). Data parallel approaches have been the obvious choice when it comes to breaking up the computations over multiple processors. In the recent years, programmable graphics processing hardware units enabling a myriad of specialized algorithms to be programmed in the graphics unit have gained widespread acceptance. Today, every processing node in a graphics cluster has two powerful and fully programmable processors; a CPU (Central Processing Unit) and a GPU (Graphics processing unit), providing us with the opportunity of organizing the distribution of graphics computations over a cluster in more specialized ways to suit an application's needs. As part of our research we have studied and explored in detail functionality distribution in large graphics applications. Our thesis is that existing data parallel distribution approaches can be further enhanced by the addition of application specific functionality distribution achieved through suitable programming of the graphics hardware. To demonstrate this, we have chosen the upcoming application domain of point based rendering and implemented a functionality distributed point based rendering pipeline. The performance improvements are impressive. To the best of our knowledge, ours is the first attempt anywhere to devise a functionality distribution scheme for point based rendering and demonstrate the significant performance improvement achieved. We discuss the merits and limitations of such a distribution scheme by comparing it against traditional data parallel and single node schemes.

# Acknowledgments

I would like to thank my supervisors Dr. Sudhir P. Mudur and Dr. D. Goswami for their guidance, support and encouragement. To Dr. Mudur I am particularly indebted for all the insightful discussions which have been a great source of inspiration. I couldn't have hoped for a better mentor. Dr. Goswami's cheerful nature and friendly attitude helped provide a very pleasant and fruitful research environment.

I would also like to thank Sushil Bhakar for some of the in depth discussions on computer graphics and especially point based rendering.

Finally, I am forever indebted to my parents and sister for their endless love, and support.

# Table of Contents

# List of Figures

# List of Tables

# Chapter 1: Introduction

With the advent of 3D scanners and other data capture devices, it has become relatively easy to capture large geometric models. The size of these models, typically many millions of points, poses a serious challenge for real-time rendering. A single node is normally not capable of delivering real-time frame rates when rendering such large models.

Computer graphics applications for visualizing large data-sets have employed many data parallel solutions in the past to achieve divide and conquer. But not much attention has been given to handling data and distribution with regards to functionality of the application on the whole. Traditional data parallel approaches perform data distribution without taking into account any knowledge of the functional complexity of the graphics processing. For example a highly specialized point based graphics application and a simple polygon renderer of similar geometric complexity would both be distributed exactly the same way over a sort-first configuration provided by a system like Chromium [11], a stream processing framework that has popularly been used to implement graphics applications using the data parallel approach.

Yet another major development in computer graphics has been the advent of the specialized processors that accelerate 3D graphics computations through hardware implementation of a number of operations in the 3D graphics rendering pipeline. The more recent trend is a programmable graphics processing unit (GPU) which as the name indicates provides programmability and increases the flexibility to control the operations in the 3D graphics pipeline. Today, even the simplest of desktop machines will have two fully programmable processors, a CPU and a GPU. To better exploit the processing

power of these modern day processors we strongly feel that data sets have to be arranged for best overall system performance. Given that the programmable GPU has made its mark even in the mainstream processing and different non-graphics applications can now take advantage of its raw processing power (about 7 times a CPU in FLOPS) [9], it becomes even more critical to delve into distribution strategies which employ a mix of functionality and data partitioning schemes.

Also most systems which employ data parallel schemes in graphics suffer from a scalability problem on modern day clusters. This is discussed further in Chapter 2. Such systems could scale better if they break up their functional complexity and suitably distribute functionality as well.

The graphics applications we are targeting are data intensive, demanding real-time rendering performance. So it is important to exploit the application characteristics to overcome the memory limitations and also internal and external bandwidth limitations.

Our thesis is that modern day systems with CPU and GPU processors can be programmed to provide efficient functionality driven distribution strategies that result in superior performance when compared to traditional *sort-only*[1] based approaches which pay very little or no attention to application functionality. Understanding the application domain is crucial to achieve a better distribution. We have chosen to demonstrate this with a point based rendering application. And the performance improvements are impressive when compared to implementations with only data parallel distribution. We

---

[1] In distributed graphics, any strategy involving distribution of graphics primitives over a cluster of nodes has to eventually address sorting them in depth order on the screen. Hence the distribution scheme is often categorized with respect to when and where it does the sorting. More details can be found in chapter 2.

have implemented an efficient point rendering application with some significant enhancements of our own to the existing data organization techniques therein. We have also proposed a set of application and system parameters that play a key role in determining the efficiency of distribution over different schemes. Though not restricted to it, our work primarily targets cluster based graphics.

## 1.1 Contributions

The contribution of this thesis is threefold. The primary contribution is formulating functionality distribution in graphics clusters through the programming of GPUs to improve graphics rendering performance when dealing with large and complex graphics applications. Data parallel solutions exist and are needed but we show that when they are augmented with functionality distribution they can benefit significantly on performance.

Second, the thesis proposes a set of parameters for modeling performance at each node of a graphics cluster.

Finally the implementation of a distributed point rendering pipeline with efficient data organization techniques is the first of its kind and is a valuable contribution in itself.

## 1.2 Organization of Thesis

Chapter 2 gives a background of related research work in distributed graphics. The focus is on the different distribution schemes employed till date. We discuss various efforts that attempt functionality distribution. Next, we give a brief introduction to

modern day GPU and its capabilities. Lastly, a brief survey of our application domain - point based rendering, is provided.

Chapter 3 discusses a set of application and system parameters to determine the performance of a node in a graphics cluster. Subsequently we model the performance of a pipelined distribution scheme and discuss other useful distribution schemes.

Chapter 4 elaborately discusses our application domain of point based rendering. We start with the various ways of representing a point and the related tradeoffs. Later organization of points into a hierarchical structure is discussed. We discuss some optimization strategies herein. Subsequently various point selection and rendering techniques are discussed. We end the chapter with a summary of our rendering algorithm.

Chapter 5 focuses on distributing the point rendering application described earlier in chapter 4. We discuss on a distribution scheme for a three node pipeline. This scheme is further extended to incorporate data parallelism.

Chapter 6 elaborates on the implementation and provides performance figures which compare results from our distribution against a similar sort-first implementation. It models the performance of each node of our cluster employing the ideas from Chapter 3. We also summarize the implementation challenges faced.

Chapter 7 includes some concluding remarks and discusses future directions.

Appendix A is a list of publications that have resulted from our research investigations in distributed computing for graphics. Appendix B is a short glossary of graphics terms and phrases for those readers who are not familiar with the field of computer graphics. Appendix C lists the vertex and fragment shader programs which have been used in performance modeling.

# Chapter 2: Background and Related work

Figure 1 shows a standard 3D graphics rendering[2] pipeline. Details of this 3D graphics rendering pipeline are part of every graphics text book [37]. The 3D graphics pipeline consists of two principal parts – Object Space operations (transformations and lighting) and Image space operations (e.g. per fragment[3] operations like texture mapping, visibility computations, blending, etc.). Geometric primitives (polygons, lines and points) are sent down this pipeline. The essence of the rendering pipeline is to calculate the effect of each primitive on each display pixel. Due to the arbitrary nature of the modeling and viewing transformations, a primitive can fall anywhere on (or off) the screen. Thus rendering can be also viewed as a problem of sorting primitives to the screen, as noted by Sutherland et al in [18].

## 2.1 Distributed Graphics

Traditionally, in distributed graphics architectures, the classifications aim at categorizing the cluster based on the *sort* scheme employed. Molnar et al. [19] classified the various possible schemes into three categories: *sort-first*, *sort-middle* and *sort-last*, depending on whether the sorting process takes place before the object space operations,

---

[2] In the broader perspective 'rendering' refers to 'all' the algorithms run on the data-set per frame. Sometimes, it is also used to denote only the last stage of graphics processing which corresponds to the computations that occur from feeding the data to the GPU to getting an image frame on display. This does not account for the processing carried out on the CPU prior to sending the data to the GPU. We too would use the term for both denotations; depending on the context it would be aptly clear which one is intended.

[3] The term fragment as used in computer graphics denotes a fractional contribution to a pixel's color.

between the object space operations and image space operations (just before rasterization), or after the image space operations respectively. Most of the traditional graphics cluster applications have been built around one of the previous categories.

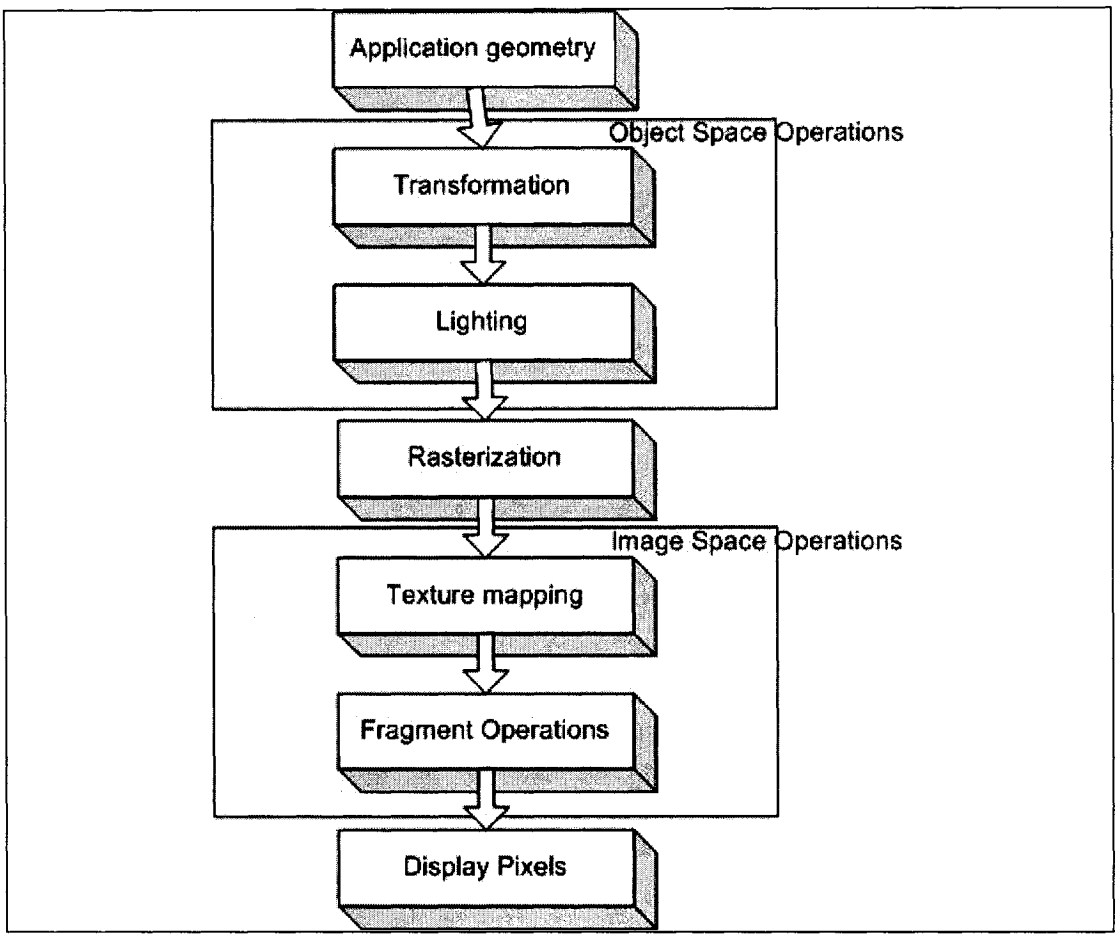


FIGURE 2-1 GRAPHICS PROCESSING PIPELINE

In *sort-first* algorithms, the display is partitioned into tiles. Each rendering node of the cluster is then assigned one or more of these tiles and is responsible for the complete rendering of only those primitives that lie within one of its tiles. The polygons making up the scene are pre-transformed to determine their screen space extents and hence the tiles they lie in. They are subsequently sent to only those nodes of the cluster that are responsible for the given tiles. The network bandwidth requirement could be brought down by utilizing the frame-to-frame coherence of the polygons.

Large display walls [20] could be easily implemented using a sort-first approach. The screen is already partitioned into tiles, with each tile being driven by a single node of the cluster.

Sort-first algorithms suffer from load imbalance due to *primitive clustering*. Primitive clustering refers to the increase in number of primitives which overlap the assigned screen regions of two or more processors. Both the amount of communication required as well as the number of primitive fragments that must be processed increase in direct relation to the number of regions into which the screen is divided. The increase is actually directly proportional to the total length of cuts made across the screen. Doubling the cut length (by increasing the number of regions per processor, say, from 4 to 16) approximately doubles the amount of communication as well as the number of additional primitive fragments in the system [67].

Samanta et al. [21] suggested dynamically changing the tiling. This scheme also does not scale well when the number of nodes in the cluster increases. As the number of tiles increases, the number of boundary primitives also increases. This problem is solved by using a hybrid of sort-first and sort-last approaches [22]. The above approach required the full replication of the geometry data on each node in the cluster. Furthering their work, Samanta et al. [23] propose a k-way replication scheme in which each 3D primitive is replicated on $k$ out of $n$ nodes of the cluster requiring only a partial replication, trading off memory usage for efficiency.

A different approach to data partitioning and scalability was taken in Chromium [24] by Humphreys et al. It packs OpenGL[4] geometry drawing commands into data packets and streams them through the nodes of the cluster. Its stream based architecture allows manipulating the contents of the stream on the fly over the cluster nodes (using "stream filters"). In addition to its existing set of "stream filters", Chromium allows the user to write custom filters in each node. This allows the nodes of the cluster to be arranged in sort-first, sort-last and other hybrid configurations.

But as observed earlier there is no robust solution to the load imbalance caused due to primitive clustering.

*Sort-middle* algorithms begin by arbitrarily distributing primitives to the cluster nodes for object space operations. After the primitive has been transformed into screen space, it is forwarded to another node for rendering. Similar to the sort-first approach, the screen space is divided into tiles, and each node is responsible for rasterization of primitives within its tile(s). SGI's RealityEngine [26] is a sort-middle tiled architecture which uses a shared high-speed bus to broadcast state commands and primitives. The granularity of task partition is very fine-grained since RealityEngine broadcasts one triangle each time and dispatches every 2 scan lines to one rasterization processor. UNC's Pixel-plane [27] is another example of sort-middle tiled hardware architecture. It distributes primitives from a retained-mode scene description and composes framebuffers by high-speed ring network. The rasterization and fragment stages are executed as SIMD.

---

[4] OpenGL is the current industry 3D graphics programming standard that implements the 3D graphics rendering pipeline and enable rapid development of 3D graphics applications.

Sort-middle systems require an intervention at the rasterization phase of the rendering pipeline, so that screen space primitives can be redistributed. In order to implement this approach efficiently, specialized hardware is needed as noted above. Hence, the above approach is not suitable on cluster based graphics.

*Sort-last* algorithms arbitrarily distribute primitives to the cluster nodes for object space operations as well as image space operations. After all primitives have been rendered, the frame buffer (color buffer and z-buffer) of every rendering node is transmitted over an interconnect network to a compositing node which resolve the visibility of pixels from each renderer to form the final image. This usually requires a large amount of bandwidth, because each node must send the entire image to a compositing node. PixelFlow [28] is an example of sort-last architecture. It distributes the rendering task over an array of identical renderers, each of which computes a full-screen image of a fraction of the primitives. A high-performance image-composition network composites these frame buffers into a single image for final display.

Nguyen et al. [29, 30] propose a variant of the sort-last scheme ILD (Image Layer Decomposition). At every frame, it creates a total order of the primitives in the scene based on the depth order using an Octree[5] and the view point. The geometric primitives are then distributed in a sort-order which ensures that visibility issues are taken care of at the compositing node. This circumvents the need for transmitting the z-buffer for every frame. But this scheme is unsuitable for large sized-models as it would require huge

---

[5] Octree is hierarchic spatial data structure much like the binary tree, but one that recursively divides a 3D cube into smaller sub-cubes. It is a popular data organization technique in computer graphics. We too use it in our point based application and hence describe it much greater detail in later chapters.

amounts of memory to create a total order of the primitives using an Octree. Moreover, this scheme does not deal with intersecting polygons.

As each node is required to send a screen resolution sized color and depth buffer (minimally) in every frame, sort-last approaches are not suitable for graphics clusters unless special hardware is used for compositing the pixel data.

### 2.1.1 Functionality distribution and related work

As already noted, much of the earlier distributed graphics work concentrated primarily on data partitioning schemes based on the classification by Molnar et al. [19]. These approaches can easily exploit hardware parallelism when the computation can be split into smaller equal weight sub-computations, with good data locality. Such a distribution is also called *regular*. Simple mapping and scheduling algorithms can be used to satisfactorily exploit the hardware parallelism. The same does not happen if the sub-computations do not have similar complexities or do not have good data locality properties. Parallel computations in this category are called *irregular* (e.g. numerical treatment of large sparse matrices). 3D graphics rendering pipeline computations could be viewed as a collection varied complexity algorithms operating on different data structures. Functionality distribution could be used to create smaller regular sets of computations. The computations in these sets would best exploit the hardware and data locality.

For example, Govindaraju et al [31] demonstrate occlusion based partitioning on a cluster of GPUs. They try to generate an occlusion representation on one node, cull away occluded objects on another and render the geometry on a third different node. They

propose an occlusion switch consisting of two GPUs where each GPU is used to either compute an occlusion representation or cull away primitives not visible from the current viewpoint. They switch the roles of each GPU between successive frames. The visible primitives are finally rendered in parallel on a third GPU. They utilize frame-to-frame coherence to lower the communication overhead between different GPUs and thus improve the overall performance. Their algorithm also employs levels-of-detail and is implemented on three networked PCs, each consisting of a single GPU. They further extend their distributed visibility culling technique to perform interactive shadow generation over a cluster [40]. The significant point to note is that in every frame, each of the three nodes above implement different algorithms (e.g. level of detail selection and occluder rendering, frustum culling and hierarchical Z-buffering, and rendering of visible nodes) which operate on specialized data structures (Z-buffer, scene graph and queue of visible nodes).

Isard et al [41] perform distributed soft shadow rendering by programming every GPU on a cluster to calculate the contributions of a disjoint set of light sources for every object and finally compose the result over a Sepia 2a compositing network [42] for display.

Heirich et al [43] demonstrate the need for parallelizing the iterative multi-grid solver routines of a CFD in order to visualize the pressure field of a developing steady-state solution.

Zara et al [44] simulate cloth animation over a cluster of 100 nodes by exploiting data as well as task parallelism. Their simulation models cloth as a collection of 3D points. At every time step it computes position and velocities of sample points and

displays the associated surface. Using an implicit integration method they solve a sparse linear system to compute particle velocities. Each node of their cluster stores properties of a small set of points which interact with the neighboring nodes (to find the force exerted by neighboring particles) at runtime to determine velocities and accelerations of the particle at the node in consideration.

Zhe et al [25] use a cluster of 30 GPU nodes to perform parallel flow simulation using the lattice Boltzmann model (LBM). Their application virtualizes the cluster as a 2D grid which facilitates communication sideways and diagonally.

Kipfer et al [45] demonstrate a distributed lighting network by distributing radiosity, ray tracing and photon mapping.

All the above mentioned schemes exploit the application characteristics like data locality and functional complexity to distribute their computations. They cannot be categorized purely on the basis of sorting classification.

While the approach in each of the above cases is largely that of providing a specific distributed architecture suited to the needs of the computations in a single application, in our research we have investigated distribution of computations and data in the more general setting of a graphics cluster to take into account the differential capabilities of CPUs and GPUs. It is accepted that optimal functionality distribution cannot take place without adequate knowledge of the graphics application under consideration. To that extent, we have been able to identify various architectures that can augment data parallel approaches with functionality distribution to provide distribution scalability. We have also specified a set of system and application parameters to model

the performance of a system which can help in making the appropriate choice for distribution in a given application.

## 2.2 Graphics Hardware and Applications

Over the last decade while the CPU's performance has consistently followed Moore's Law, the GPU has outperformed it [1]. Many of the standard graphics processing algorithms have been successfully incorporated in the hardware. This has consistently fuelled the interests of the end users and set higher expectations from the graphics and animation industry. As a result the sheer complexity of a modern day graphics processing pipeline has increased enormously. For example, the OpenGL (open graphics API) extension set which allows the expansion of the hardware graphics pipeline with a host of specialized algorithms ranging from occlusion to specialized imaging filters, has grown from a few dozens to over three hundred since the first release of OpenGL (version 1.1).

### 2.2.1 Graphics hardware programmability

If we replace the Object Space and Image Space Operations block of Figure 2-1 with user programmable units then we get the programmable GPU model. Of the two, the former is called the vertex processing unit and the latter is the fragment processing unit. The vertex processing unit accepts geometry data in the form of vertices and related attributes like color, normal, texture coordinates, etc. Each primitive is scan-converted in the rasterization block (refer to Figure 2-1), generating a set of fragments in screen space. A fragment is essentially a candidate pixel. It may combine with other candidates for the same pixel position on the screen. The fragment processing unit is used for performing

useful computations on fragments. For example, the texture coordinates of a fragment can be used to fetch colors from one or more textures. Finally, various tests (e.g., depth and alpha) are conducted to determine whether the fragment should be used to update a pixel in the frame buffer.

An application has the flexibility to embed any computation inside the vertex processing and fragment processing units. Figure 2-2 illustrates the implementation per pixel Phong shading [32] on the GPU.

FIGURE 2-2 ILLUSTRATION OF PER PIXEL PHONG SHADING.

**Using fragment shaders of a user programmable GPU per pixel lighting calculation is performed.**

Modern GPUs are highly parallel in their internal architecture. They can process up to 6 vertices in the vertex processing unit, and up to 16 fragments in the fragment processing unit in parallel. The GPU hardware supports vector data types and allows matrix operations, dot and cross products, and other high-level operations on vector types at instruction level. However, there is no provision of storing intermediate state

information in the above units and also there cannot be any communication between the multiple units which operate in parallel. This is required to keep the graphics hardware design simple and its operation efficient.

With the onset of programmable vertex and fragment shader units more specialized application specific algorithms could be incorporated into the hardware graphics pipeline. In fact there is no upper limit to the functional complexity (apart from desired frame rate). Other emerging areas like point-based graphics demand significant software processing of the datasets, both offline and online. For example, a GPU based point-graphics pipeline with high quality splatting (discussed in chapter 4) and filtering algorithms can barely render a few million points per second in real-time. To sum up, the functional complexity of the real-time graphics processing pipeline has become too large to incorporate all the desired algorithms in a single rendering pass. Due to the specialized nature of these hardware algorithms at one time only a small subset of these can be incorporated to keep real-time frame rates for rendering.

### 2.2.2 Application - Point based rendering

We have chosen to focus our study in the application domain of point based rendering. 3D scanners are gaining popularity very fast. Such a device helps to capture a precise 3D representation of a real world object. The data produced by a 3D scanner is a huge collection of 3D points which represent a sampling of the surface of the scanned object. With the advent of low cost scanners, the costs of geometric modeling, particularly in entertainment applications are rapidly decreasing. Other applications include engineering, medicine (orthopedics), security (3D face matching) etc.

The absence of topology and decreased setup and rasterization costs as compared to polygons makes 3D points an interesting alternative as a rendering primitive. With the current graphics hardware being largely rooted on polygon based rendering we are faced with a twofold problem. First, we must invent novel techniques to represent the point-cloud surfaces, and second, efficiently adapt these techniques over the current generation graphics hardware. Over the past few years this has fuelled the interest of the computer graphics community towards point based rendering techniques. Our focus in this research is on solutions to the second problem. We present some efficient improvements to the rendering process by distributing the selection and rendering algorithms on different nodes of a cluster. As a result the renderer works on the point-set and the selection nodes on the spatial-subdivision hierarchy exclusively. This helps us exploit data locality better and to achieve an efficient functionality distribution.

The use of points as a display primitive for continuous surfaces was introduced by Levoy and Whitted [47]. In 1989, Westover [54], introduced splatting for interactive volume rendering. In splatting, each projected voxel (a unit of volume data) is represented as a radially symmetric interpolation kernel (e.g. Gaussian) giving the appearance of a fuzzy ball. Grossman [53] investigated the use of point sampled representations as an alternative to triangles for rendering. One of the first point based rendering systems was QSplat [35]. In QSplat, a multi-resolution hierarchy, based on bounding spheres, is employed for the representation and progressive visualization of large models. The system was written for use in a large-scale 3D digitization project. In the same year, Pfister and Zwicker introduced *surfels* [60], a zero dimensional n-tuple that captures shape and color attributes that locally approximate the surface of a given

object at any given point. Surfels are a useful paradigm for efficiently rendering complex geometric objects at interactive frame rates. It allows the representation of point sampled surfaces augmented with additional attributes. Depending upon desired visual quality of the renderer, a surfel's basic structure could be decided.

Zwicker et al. [61] introduce surface splatting wherein they render opaque and transparent surfaces from sampled point representations. They employ an elliptical Gaussian kernel in the image space to achieve high quality anisotropic texture filtering and anti-aliasing. Their approach is based on a screen space formulation of the Elliptical Weighted Average (EWA) filter [62]; Disc-shaped splats in object-space project to elliptical splats (stored as textures) with Gaussian intensity distribution in image-space. It results in high-quality anti-aliased rendering but the number of point samples rendered is less.

Due to the inherent lack of topological information, point data-sets could be efficiently represented in a hierarchical data structure. The QSplat [57] system uses an efficient quantized representation for each hierarchical node using a mere 48 bits. They serialize the entire hierarchy into a file and use this compressed representation for remote rendering. Botsch et al. [63] proved that a pure software implementation could render up to 14 million Phong shaded samples per second by using a quantization of splat shapes. However the models used to achieve these rendering times are not complex in terms of memory requirement. Their quantized hierarchical data representation is very compact with a memory consumption of less than 2 bits per point position. High quality results can be achieved but their rendering speed is limited.

Of late, there has been a growing interest in using programmable graphics hardware to accelerate the rendering process. In [55] the authors provide high quality, as well as efficient, rendering based on a two-pass splatting technique with Gaussian filtering. Finally, in their most recent publication the authors propose to base the lighting of a splat on a linearly varying normal field associated with it, resulting in a visually high quality image [64]. Dachsbacher et al. [48] present a hierarchical LOD structure that is suitable for GPU implementation. They can process 50M low quality[6] points per second. Although extremely fast, a GPU's on-board memory is currently rather limited in terms of data storage. It is inevitable to employ a PC cluster for larger data models since a PC cluster provides a scalable memory model. Also for larger complex scenes a single GPU can not handle all the processing so distribution is inevitable.

### 2.2.3 Distributed point rendering

In [65] Hubo and Bekaer have described a sort-first point rendering configuration and noted that in the absence of topological information, point-rendering is ideally suited for sort-first rendering. They demonstrate a peak performance of splatting 1.5 million points per second per node. In comparison, our functionality distributed pipeline splats over 3.5 million points per second per node. To the best of our knowledge, there has been no other attempt at functionality distributed point-rendering.

---

[6] Low quality splats (points) are rendered as flat squares on the screen. Effects like filtering, blending and orientation are avoided to achieve efficiency. For large and dense models this can give reasonable rendering quality.

# Chapter 3: Functionality Distribution

## 3.1 System Considerations for Functionality Distribution

The last decade has seen tremendous advances in VLSI technology. The average performance of graphics processors (GPU) has increased by 2.8 times, CPUs by 1.7 times, and DRAM memory by 1.1 times per year averaged over the same time period [57, 58]. The significant point to note is that the *gap in memory and core processor bandwidth* has also increased nearly *exponentially*. The traditional DRAM memory model of unit access time to all memory locations, no longer holds true. Cache memories were invented with the sole aim to alleviate this problem. But their mere existence does not guarantee optimal performance. As an extreme example, several current high-end machines run simple arithmetic kernels for out-of-cache operands at 4-5% of their rated peak speeds which implies that they are spending 95-96% of their time idle and waiting for cache misses to be satisfied [59]. Hence, it will not be inappropriate to conclude that cache conscious programming [46] has to be employed in memory intensive applications. Needless to say large data-set graphics applications are good candidates.

The *serial bus interface* between the CPU and GPU is a well known bottleneck. In addition, contemporary bus architectures like AGP 8x have an asymmetric bandwidth (2.1GB/sec peak for downstream and 133MB/sec peak for upstream). The asymmetric bandwidth reflects the need for the CPU to push vast quantities of graphics data to the GPU and to read back only a small portion of data. Even with 2.1 GB/sec downstream bandwidth a modern day GPU (typically 200GFLOPs [2, 3]) can barely be sustained at its fullest potential.

Coming to cluster based application, the *external network bandwidth* is yet another well known bottleneck. For data intensive applications like large model visualization the situation only worsens if clever strategies for compression, data locality, etc., are not employed.

Any functionality distribution can capitalize well by taking into consideration the above three factors as it is generally not possible to fully avoid these bottlenecks in a *pure* data parallel distribution.

## 3.2 Application and System Parameters

Table 3-1 presents a list of application and system parameters that play a vital role in determining the following two issues governing real-time performance:

- Choosing a functionality distribution architecture for a given application, and
- Determining the set of computations which need to be offloaded to a node of a cluster and deciding on what could be done on the GPU of that node.

We assume that texture data and other application constants are loaded once in the beginning of the application; so these do not play a significant role in deciding on the real-time performance of a system. On the other hand, the GPU-CPU communication parameters play a key role. The computations are assumed to be floating point. Given that all the geometric data are floating point scalars or vectors, this is a reasonable assumption.

| *Sr. No.* | *A* | **System Parameters** |
|---|---|---|
| *1* | $S_{GPU}$ , $S_{CPU}$ | Processing speeds of the GPU and CPU in FLOPS. |
| *2* | $Mem_{mm}$ | Size of the main memory. |
| *3* | $Mem_{ca}$ | Size of the system cache memory. |
| *4* | $Mem_{vdo}$ | Size of the on board video/graphics card memory. |
| *5* | $l_d$ , $B_d$ | Downstream (CPU to GPU) communication latency and bandwidth. |
| *6* | $l_u$ , $B_u$ | Upstream (GPU to CPU) communication latency and bandwidth. |
| *7* | $l_m$ , $B_m$ | CPU to Main memory communication latency and bandwidth. |
| *8* | $l_e$ , $B_e$ | Node to Node communication latency and bandwidth of the cluster. |
| *9* | $P_n$ | Number of processors in the cluster (participating in the computation). |
| | ***B*** | **Application Parameters** |
| *1* | $D_{FB}$ | Size of the framebuffer of the GPU (proportional to the screen resolution). |
| *2* | $\eta$ | % of the framebuffer read back by the application. |
| *3* | $D_{CP}$ | Maximum data size processed on the CPU per frame. |
| *4* | $D_{VP}$ , $D_{FP}$ | Maximum vertex and fragment data sizes processed on the vertex and fragment processing units of the GPU respectively per frame. |
| *5* | $O_{CPU}$ | Computational complexity of the algorithm running on the CPU. Number of operations (FLOPS included) per $D_{CP}$ data unit. |
| *6* | $O_{VP}$ , $O_{FP}$ | Computational complexity of the algorithms running on the vertex and fragment processing units of the GPU respectively. Number of times executed per second. |
| *7* | $F$ | Number of frames of computation desired per second on the GPU. |
| *8* | $N_{FB}$ | Number of times the framebuffer is read back per frame by the application. |
| *9* | $D_e$ | Maximum data size communicated between two nodes per frame. |
| *10* | $l_{es}$ | Software latency incurred in data transmission per frame. |
| *11* | $N_D$ | Number of display primitives. |

TABLE 3-1: ESSENTIAL PARAMETERS FOR PERFORMANCE MODELING A GRAPHICS CLUSTER NODE.

**Functionality distribution over a graphics cluster can be modeled with the help of above listed parameters. The system parameters give an idea of the flexibility the developer has over deciding the data structures and functionality distribution. The application parameters help the developer model the performance of a distribution and find out the bottlenecks.**

It is important to note that the parameters listed in Table 3-1 Part A are affected by changes in hardware features of the system and the ones in Part B depend exclusively on the application behaviour. The bottlenecks in each of these categories are the significant aspects to catch while planning on a distribution scheme.

For example the intra node communication parameters (A-5 and A-6) depend on the bus type (PCI, AGP 4x/8x, PCI Express, etc). As already mentioned, it is well known that contemporary bus architectures like AGP 8x have an asymmetric bandwidth (2.1GB/sec peak for downstream and 133MB/sec peak for upstream). The asymmetric bandwidth reflects the need for the GPU to push vast quantities of graphics data at high speed and to read back only a small portion of data. Clearly, applications that need to read back the framebuffer per frame would suffer from this inherent hardware bottleneck.

Simple ad-hoc performance evaluations could be formulated to check on the load of each node of a cluster. For example, one obvious condition to hold true for all the cases to guarantee real-time performance is that the total computation time on the GPU and CPU of a node and their respective internal and external communication latencies per frame should be less than *1/F*. There are other conditions that can be similarly formulated for different distribution schemes. A consistent model for evaluating the performance at each node is highly desirable, but is beyond the scope of this research.

## 3.3 Distribution Schemes

We now briefly present three distinct distribution schemes and analyze in detail the one which we use for our application in point based rendering.

### 3.3.1 Pipelined

Sequential computations of the form F4 (F3 (F2 (F1))) can be split across multiple nodes to create a pipeline of nodes (Figure 3-1). The last node in the pipeline drives the display. As the output of each node drives the immediate next node's input this architecture could effectively hide communication latency.

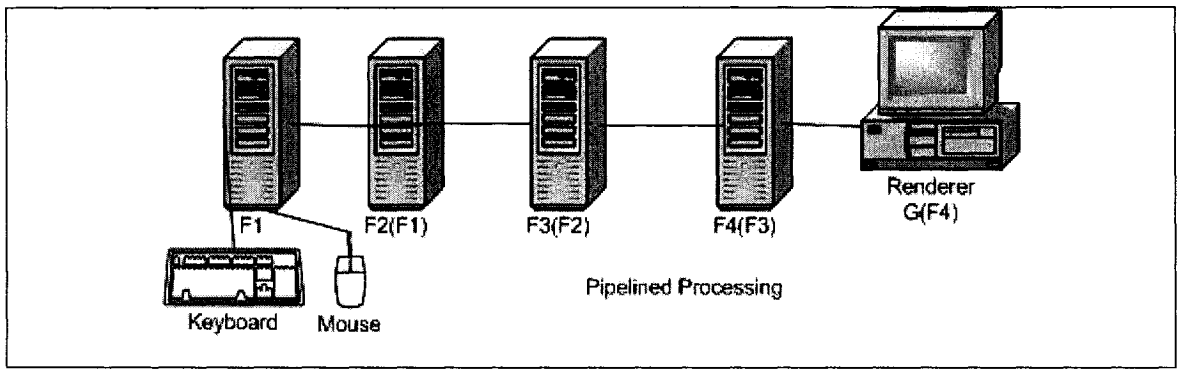


FIGURE 3-1 PIPELINED PROCESSING.

We analyse the distribution herein with respect to a user desired frame rate $F$. Please note that the discussion pertains to worst case analysis. Also each of the cases is analyzed in *isolation*, i.e. for each frame the node performs just the listed activity only.

- Feeding data into the GPU and reading back the results incur latency. Assuming that an application reads framebuffer data multiple times in a frame (as in the case of a hardware occlusion query) we have then the following criteria on read back:
  - $((1 - N_{FB} \times F \times l_u) \times B_u) / F) < \eta \times D_{FB}$ ... (1)
- The transmission of data between two nodes should not exceed per frame the following:
  - $((1 - F \times l_e) \times B_e) / F)$. ... (2)
- The vertex shader consists of instructions whose overall throughput might vary depending upon the way the driver chooses to optimize them for execution on the

GPU. For example, a MUL ($x \times y$) and a POW ($x^y$) instructions take about the same time (0.6783 Ginstr[7]/sec) on a Raedon 9800 graphics card when issued once in a vertex shader but if they are issued for eight times in the same shader program the MUL instruction gives about three times the throughput as compared to POW (3.0221 Ginstr/sec for MUL and 1.099 Ginstr/sec for POW). Hence we measure the computational effort of the shader code ($O_{VP}$) by considering it as a whole rather than in parts. We employ it to render a large model. For models larger than half a million points we find that the communication overhead between the CPU and the GPU is offset by the rendering time of the model. For example three different models above half a million points gave similar $O_{VP}$ value for the same shader code. The computational complexity of the vertex shader is governed by multiple factors. We analyze three cases here –

- Employing Vertex Buffer Objects: The data required per frame of the computation is stored conveniently in the video memory once at load time. No data transfer takes place between the CPU and GPU subsequently apart from the transformation and rendering commands. This is the ideal way to measure the $O_{VP}$ for a vertex shader code. The governing condition for the vertex shader complexity then would be :

  $$D_{VP} \times F > O_{VP} \quad \ldots (3)$$

- Employing Vertex Arrays: Vertex arrays are a mechanism to collect all the data required to send to a GPU per frame and send it in one chunk from the system memory to the video memory. Since the downstream communication

[7] Giga instructions per second.

from CPU to GPU occurs once per frame, the deciding factor is the input data size. Hence in addition to (3) we have:

$((1 - F \times l_d) \times B_d) > D_{VP} \times F$ ... (4)

- Employing Immediate Mode: Using immediate mode, each primitive (e.g. points, polygons, lines, etc.) is sent to the GPU separately in every frame. Now we perform multiple communications per frame to the GPU. One trip for each display primitive. Hence (4) gets modified to –

$((1 - F \times l_d \times D_{VP}) \times B_d) > D_{VP} \times F$ ... (5)

- The computational complexity of the fragment shader is governed by the following criterion –
  - $D_{FP} \times F > O_{FP}$ ... (6)

It should be noted that we perform the worst case analysis by treating each aspect of computation in isolation. As the fragment shader receives direct input from the rasterizer, we need not consider the three cases discussed before. Only the worst case value of $D_{FP}$ has to be assessed. Once this is found out we could carry the above test for $O_{FP}$ by drawing screen sized quadrilaterals to generate the required number of fragments. The cost of transmitting the vertex data in such a case would be minimal.

- The computational complexity on the CPU is more difficult to measure. Many parameters play a role in determining the optimal usage of CPU including the type of operating system. Hence we try for an approximation. Criterion (7) provides a way to measure the upper bound on the computations. Criterion (8) governs the upper bound on the data size. We assume the worst case is where there is 0% cache hit hence a memory access occurs for every data unit.

- $D_{CPU} \times F > O_{CPU}$ ... (7)
- $((1 - F \times l_m \times D_{VP}) \times B_m) > D_{VP} \times F$ ... (8)

Each of the above conditions is derived by measuring a system or application parameter in isolation. This provides us with a measure to check whether a distribution is violating the worst case scenario for any of the measured parameters. For optimizing the application to achieve better performance on each node, we need to view individually the time taken for the respective latencies as well as the computations as a whole.

As stated earlier a consistent model which could be used to evaluate the performance of each node is highly desired. Such a model would combine the equations stated above and present a holistic view of the node's functionality.

It should be noted that the data that flows through each stage of the pipeline is application dependent. In the above we assume that the pipeline flow is unidirectional. Later in chapter 5, we shall be analyzing the functionality distribution in our application in the light of above parameters.

### 3.3.2 Master-Slave

Here the distribution of computations (F1, F2, F3 and F4) is such that each node outputs an intermediate result, which is combined using a function (G (F1, F2, F3, F4)) at the renderer. Much of the analysis per node would be similar to section 3.3.1. The effective data output per node per frame is reduced by an order of $P_n$ as compared to the nodes in a pipelined architecture.

The renderer (or master) node should be capable of feeding the data to the slave nodes at a suitable rate to avoid starvation of the slave nodes. .

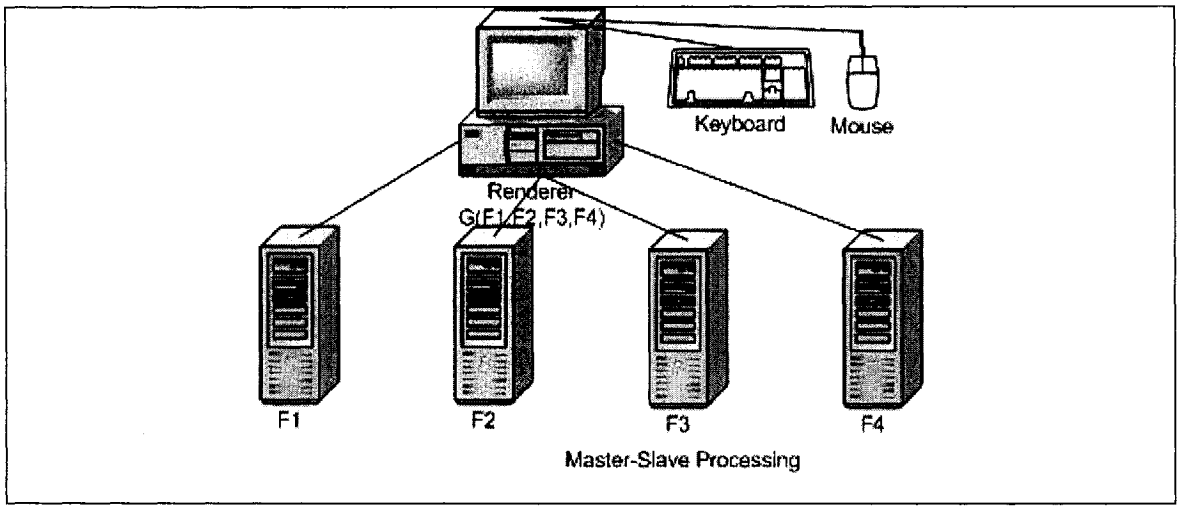


FIGURE 3-2 MASTER-SLAVE PROCESSING.

### 3.3.3 Hybrid Schemes

For large datasets, first stage functionality distribution and second stage data parallel processing can be carried out (Figure3-3). We demonstrate an alternative example of a similar distribution scheme in our distributed point based rendering application (section 5.2.1). Once again, the distribution analysis could be carried out along similar lines as stated in 3.3.1. Also a first stage data distribution and second stage functionality distribution could be carried out as shown in figure 3-4. It shows a second stage functionality distribution using pipelined processors. The results of multiple pipelines feed into a display wall. This gives an easy way of coupling pipelined functionality distribution with a sort-first data distribution. Each pipeline is configured to carry out the same set of computations in compliance with a sort-first scheme, although the processors in them distribute the task. The data and functionality distribution analysis can be carried out as in sections 3.3.1.

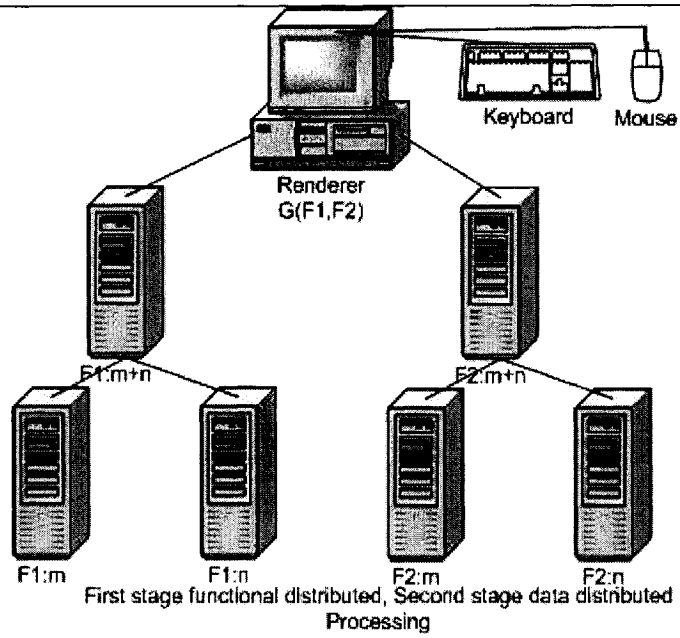


FIGURE 3-3 USING FUNCTIONALITY FIRST, DATA NEXT SCHEME.

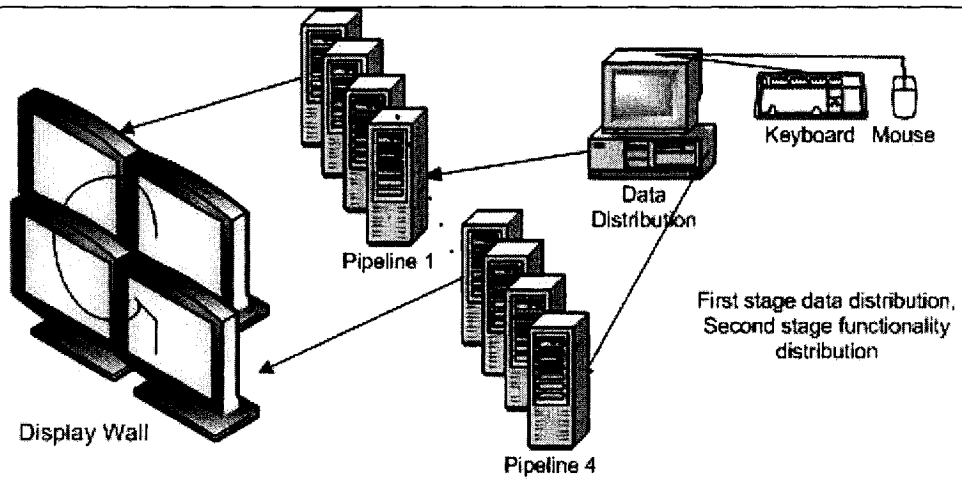


FIGURE 3-4 USING PIPELINES OF FUNCTIONALITY DISTRIBUTION IN A SORT-FIRST ARCHITECTURE.

# Chapter 4: Application - Point Based Rendering

In this chapter we discuss the stages involved in point data processing with the help of our own point rendering pipeline. Just as a good understanding of the computations involved in a graphics application is essential for achieving optimal functionality distribution, the choices for representation, access mechanism, organization, and storage of graphics application data are important factors that can considerably affect the overall performance of a distribution scheme. A substantial part of our experimental investigation efforts have been devoted to the analysis of these factors in our point data processing application. We discuss these in detail below. The performance enhancements achieved herein are themselves some of the significant contributions of this research.

We broadly categorize the stages (Figure 4-1) involved in point based rendering into two major phases – selection and rendering. The selection phase consists of a set of view dependent algorithms which decide on the candidate points to be rendered to obtain a hole-free image (Figure 4-2). The rendering phase feeds the candidate points through the GPU for generating an image on the display.

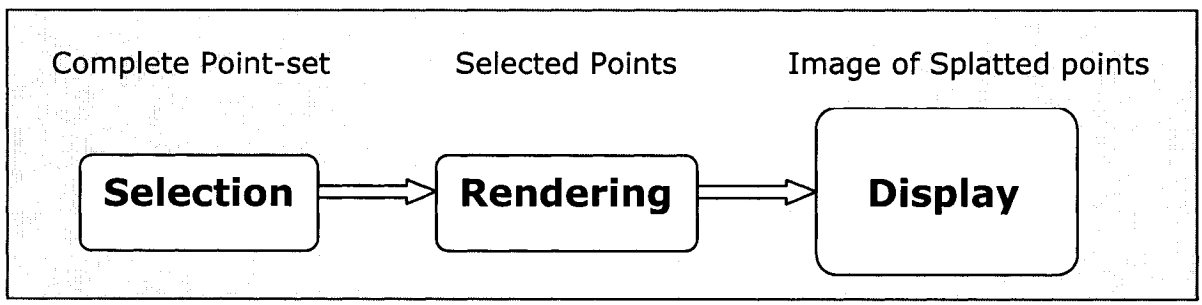


FIGURE 4-1 TWO MAJOR PHASES OF THE POINT PROCESSING PIPELINE

Each of the above two phases have multiple sub-stages that are discussed in detail below.

## 4.1 Point Representation

A point may possess several attributes depending upon the application. For a simple watertight rendering we need coordinates, surface normal, neighbor points, color, texture and splat size. Some of these properties could be represented as a dedicated structure sometimes also referred to as a *surfel* [60], a short form of surface element and others could be calculated at runtime: the decision as to which attributes are put in the surfel structure and which ones are to be calculated at runtime is an application dependent issue.

If an attribute is expensive to calculate and occupies relatively less space it might be better to pre-compute and store it in the structure during a preprocessing phase. However, with the growing relative[8] memory latency, computing some of the desired point attributes at runtime may be beneficial as compared to storing them in memory and incurring an access cost at runtime [46]. For example Table 4-1 defines three point structure representations and Table 4-3 the costs associated with rendering them. For the sake of measuring just the latency incurred, all the points are accessed and sent through the graphics rendering pipeline without any selection and the splatting algorithms applied (discussed later). The readings indicate that the costs incurred in accessing the memory is more than 3 times for an array of points based on the structure `LPoint` as compared to

[8] It's relative with respect to the computational bandwidth offered by a modern day processor.

the `TPoint` structure for the same geometric model (i.e. same number of points). For our current application we have chosen the `TPoint` structure.

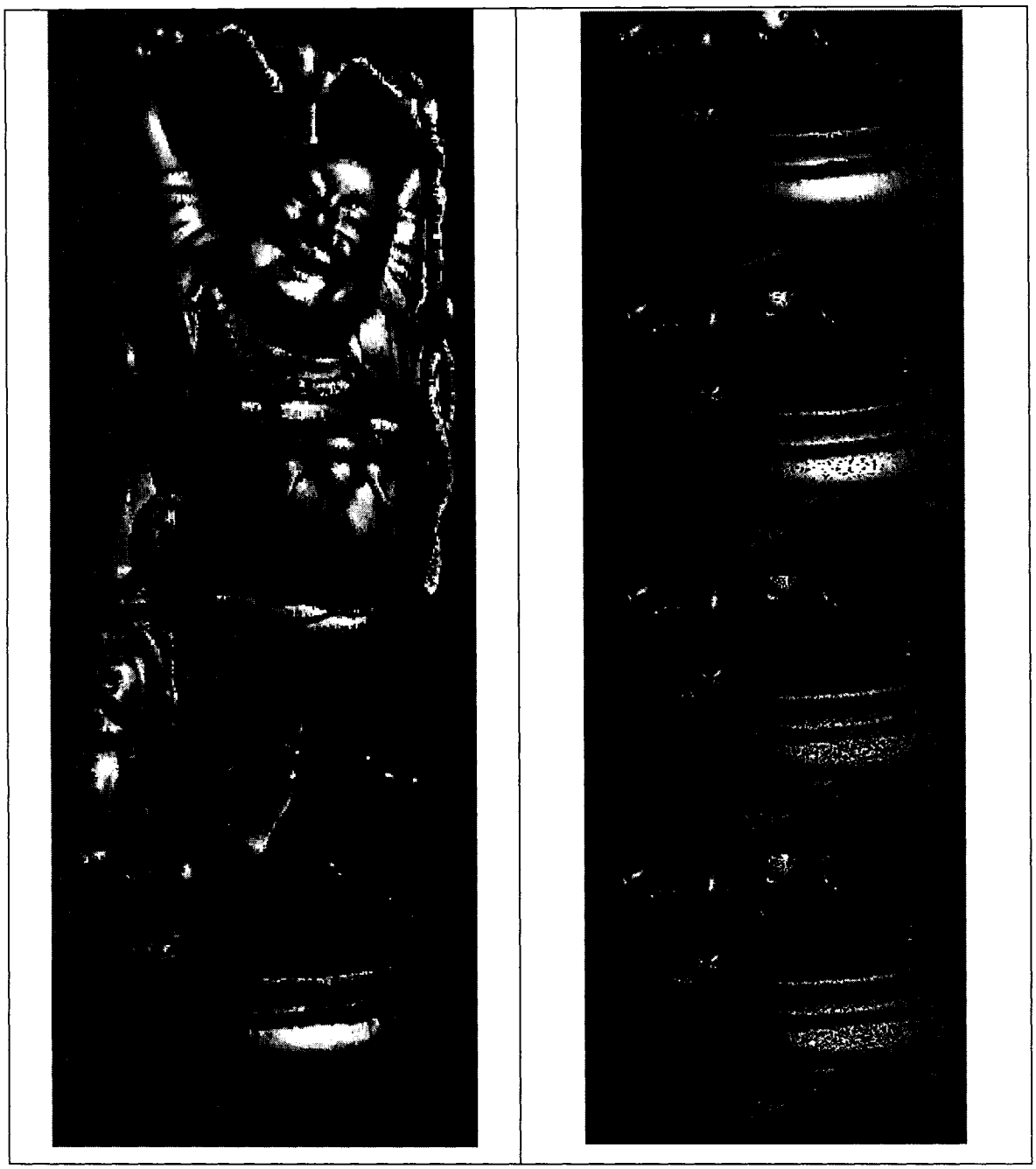

FIGURE 4-2 ILLUSTRATION OF HOLES IN STANFORD BUDDHA.

**This figure shows example of holes in the rendering of a point sampled model. When the number of points in a screen region becomes less than the number of pixels they project on, holes are created. Splatting is employed in such cases to achieve a hole-free rendering.**

```
struct TPoint{
float pt[3];                // point coordinates.
unsigned char color[3];     // color at the point.
float fRad;                 //splat radius.
};
```

```
struct MPoint{
float pt[3];                // point coordinates.
unsigned char color[3];     // color at the point.
float nr[3];                // the normals.
MPoint** pNibors;           // pointers to neighbors.
unsigned nNibors;           // # Neighbors.
PointProperties *pProp;     // additional properties
};
```

```
struct LPoint {
float pt[3];                // point coordinates.
unsigned char color[3];     // color at the point.
float nr[3];                // the normals.
LPoint** pNibors;           // pointers to neighbors.
unsigned nNibors;           // # Neighbors.
 int nfaces;                // # Faces to which this vertex contributes to
                            // when triangulated.
Face** faces;               // Faces to which this vertex contributes to
                            // when triangulated.
 PointProperties *pProp;    // additional properties
};
struct Face {
int * points;               // point index in the LPoint structure array
 unsigned char nPoints;     // number of point indices in point array
 Face** faces;              // Faces sharing an edge with this face.
};
```

TABLE 4-1 EXAMPLE OF DIFFERENT SURFEL STRUCTURES.

**Choice of three different types of surfel structures (point attributed data structures), listed in decreasing order of space - time tradeoff where the first TPoint structure is most space efficient.**

We would henceforth, be referring to the set of *surfels* (point-structures representing point attributes,) as "*point-set*" or simply "*points*".

### 4.1.1 Design tradeoffs

If we try and make the members of the `TPoint` struct (implemented as a C++ class) `private`, it would be ideal from a software design perspective on data encapsulation. But the penalty incurred on rendering performance is enormous. For example we tried out encapsulating the access to the `TPoint` attributes through a *setData()/getData()* mechanism. The results are shown in Table 4-2. We notice that the overhead associated with a function call is almost 3 times for our application. Hence we opt to tradeoff encapsulation for better performance by making all the attributes accessed per frame, `public`.

| **Stanford Bunny model rendered from -** | **FPS** |
|---|---|
| TPoint-Struct with public access to attributes | 720 |
| TPoint-Struct with private access to attributes | 242 |

TABLE 4-2 ACCESS CONTROL AND PERFORMANCE TRADEOFFS.

## 4.2 Point Organization

Hierarchical data structures have proven to be the best when it comes to capturing spatial data. They help in organizing the data (point-set, in this case) into smaller sub-spaces and in many cases encapsulate them in bounded volumes recursively, giving rise to a hierarchical representation. Such an organization is immensely helpful in accelerating typical geometric queries related to operations like – culling, intersections,

screen space occupancy, etc. In fact one of the reasons for employing a hierarchy is the gain in query and traversal performance; typically yields an improvement from *O(N)* to *O(log N)* in comparison to a linear organization.

We employ octrees to organize our point data-set. It's a data structure based on regular 3D space subdivision, constructed by first enclosing the entire point-set into a minimal axis-aligned bounding box. The box is split recursively along the three primary axes, centered about the center of the box. Every box, on splitting creates eight equally sized axis aligned boxes; and hence the name Octree. The simplicity and uniformity of the octree naturally lends itself to efficient queries and traversal. Also the construction time is minimal as compared to other space partitioning schemes. Figure 4-3 illustrates visually the recursive construction of an octree for a point-set.

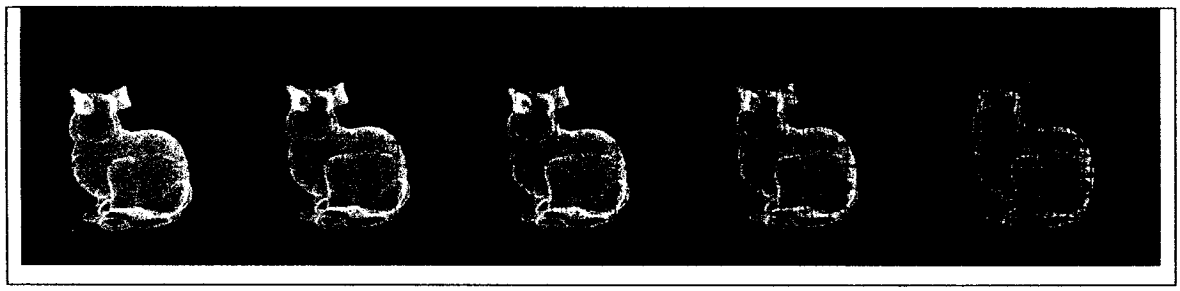

FIGURE 4-3 OCTREE CONSTRUCTION FOR THE STANFORD BUNNY MODEL

The subdivision of an octree cell is governed by the following criteria:

- Maximum number of points in the octree cell defined by the application constant MAX_POINTS_PER_OCTREE_CELL.
- The maximum height of the octree defined by the application defined constant MAX_DEPTH_OF_OCTREE.
- Flatness measure (discussed in the next section,) of the region enclosed in the octree cell.

### 4.2.1 Decoupling the point-set from the octree

*Data Clustering* attempts to pack data structure elements likely to be accessed contemporaneously into a cache block. This increases cache block utilization and reduced the cache block working set. Storing the points inside the octree cells results in poor data clustering. This results in cache thrashing [46] when the hierarchy is traversed at runtime for rendering. Since the data gets accessed multiple times in a second, data clustering is a non-trivial issue for most real-time computer graphics applications. Also traversing the hierarchy for rendering adds the overheads of recursive function calls and pointer dereferencing. Additionally, with current graphics hardware, there is no method for delegating this task over to the GPU. Hence, we decouple the spatial organization (*octree*) from the data-set collection (*point-set*) itself; the spatial organization and the data-set collection are stored separately. The point subset of each octree cell is collected into a single contiguous array (of `TPoint` structures, Table 4-1). The points are stored in depth first traversal order of the octree in the point-array. Each octree cell now contains an offset to the point-array and the number of points contained in it. Figure 4-4 illustrates the decoupled organization. The performance benefits[9] can be noticed by comparing the 3rd and 4th row of Table 4-3.

### 4.2.2 Sequential Octree

As mentioned in the previous sub-section, traversing the octree hierarchically adds overheads of the recursive function calls and pointer dereferencing. Further, to achieve better cache hit, we would like to cluster the octree cells in their traversal order.

---

[9] The performance reported is worse than expected because of storing pointers to individual points of the point-array in the octree cell which in turn contributes to additional cache thrashing.

This is achieved by serializing the octree cells to a flat array. We are presented with two choices of serialization here, either to follow a depth order or breadth order traversal of the octree.

| **Stanford Bunny Model Rendered from -** | **TPoint (FPS)** | **MPoint (FPS)** | **LPoint (FPS)** |
|---|---|---|---|
| Flat Point-array | 724.3 | 631.7 | 240.7 |
| Sequential Octree with point data clustering. | 246.5 | 211.5 | 83.8 |
| Hierarchical traversal of Octree with point data clustering. | 192.4 | 174.6 | 65.6 |
| Hierarchical traversal of Octree without point data clustering. | 64.3 | 59.7 | 20.4 |

TABLE 4-3 RENDERING PERFORMANCE WITH VARIOUS POINT ORGANIZATION SCHEMES.

We chose the breadth order traversal as it helps in achieving an implicit LoD (Level of Detail) in the traversal order [35, 48]. This is extremely efficient at runtime when we restrict ourselves to a certain level of the octree depending upon the results of the selection algorithms. Also as the different octree cells at the same level of detail are located contiguously it helps in data clustering. We call the sequential structure a ***Sequential Octree***. Figure 4-4 illustrates the construction of a Sequential Octree elaborately. The performance gains achieved could be inferred from comparing rows 2 and 3 of Table 4-3. We can now see that we achieve a significant performance enhancement (about 30%) over the hierarchical structure. However, we still end up with only one-third of the performance by rendering out of a flat point-array. (Compare rows 1 and 2 of Table 4-3.) This is largely due to the unavoidable cache thrashing that occurs when we toggle between the octree (for selection) and the point-set (for rendering) at runtime. We will present an innovative scheme to alleviate this problem when we discuss functionality distribution in subsequent chapters.

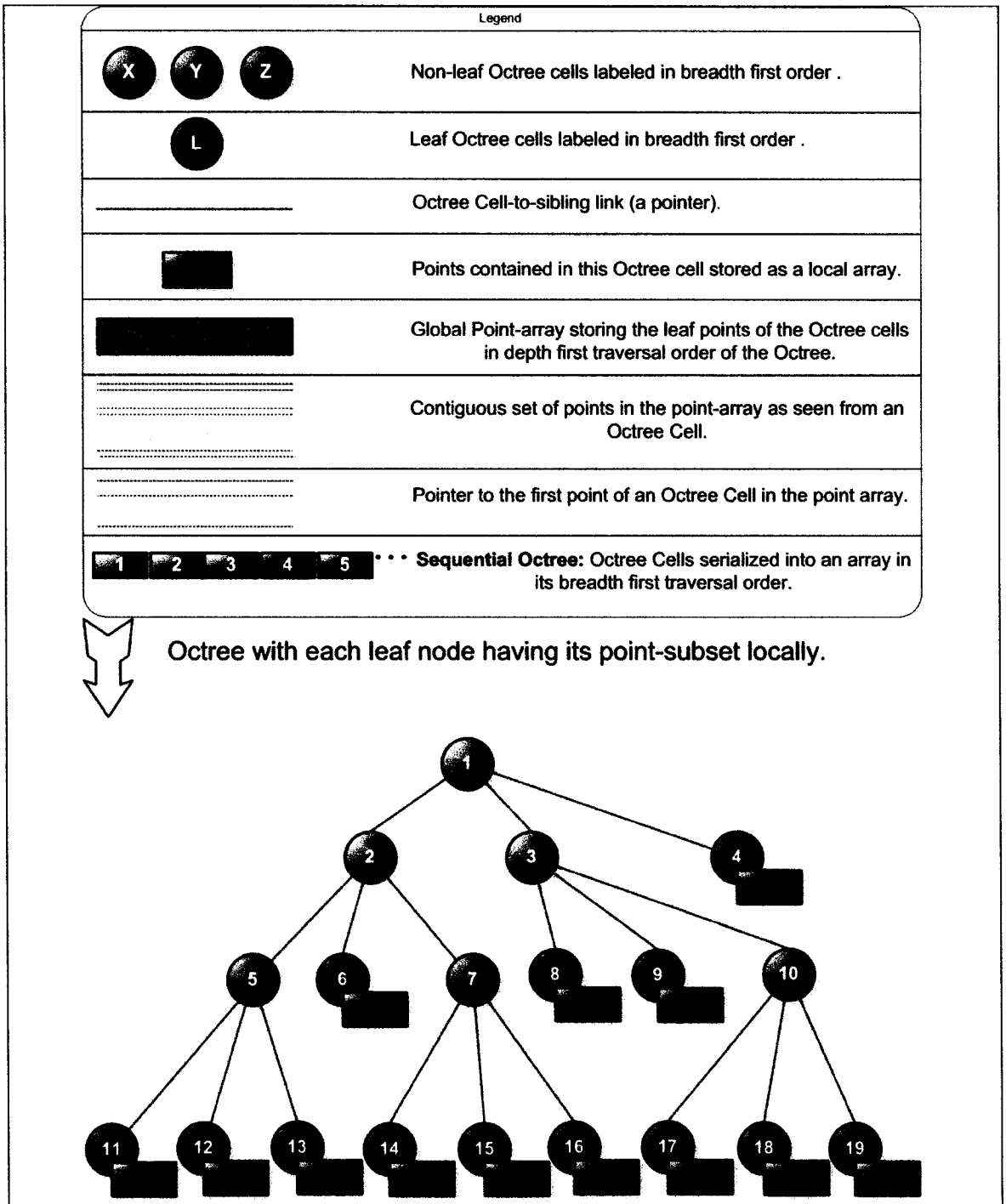


FIGURE 4-4A ILLUSTRATION OF DATA DECOUPLING - 1

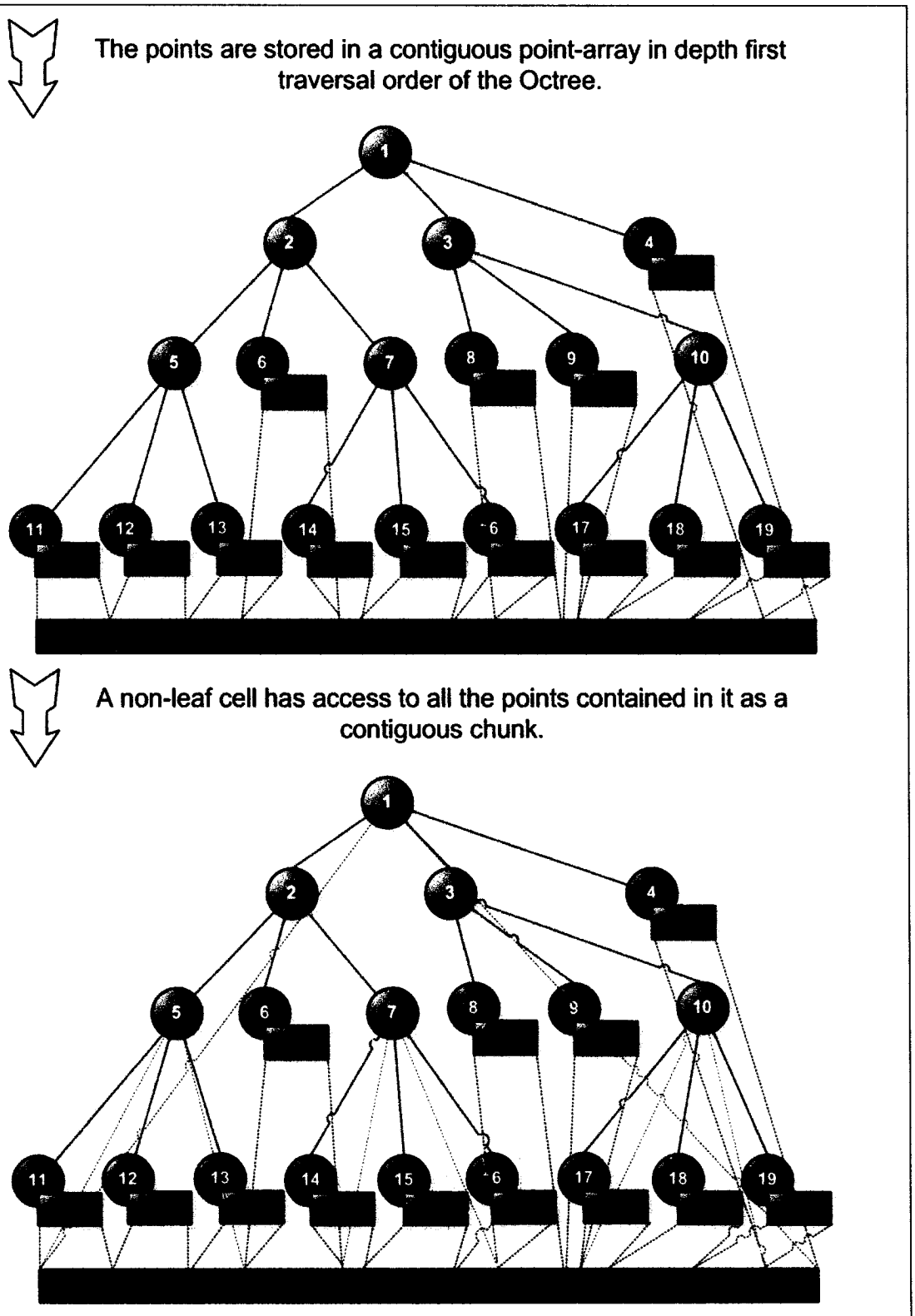


FIGURE 4-4B ILLUSTRATION OF DATA DECOUPLING - 2

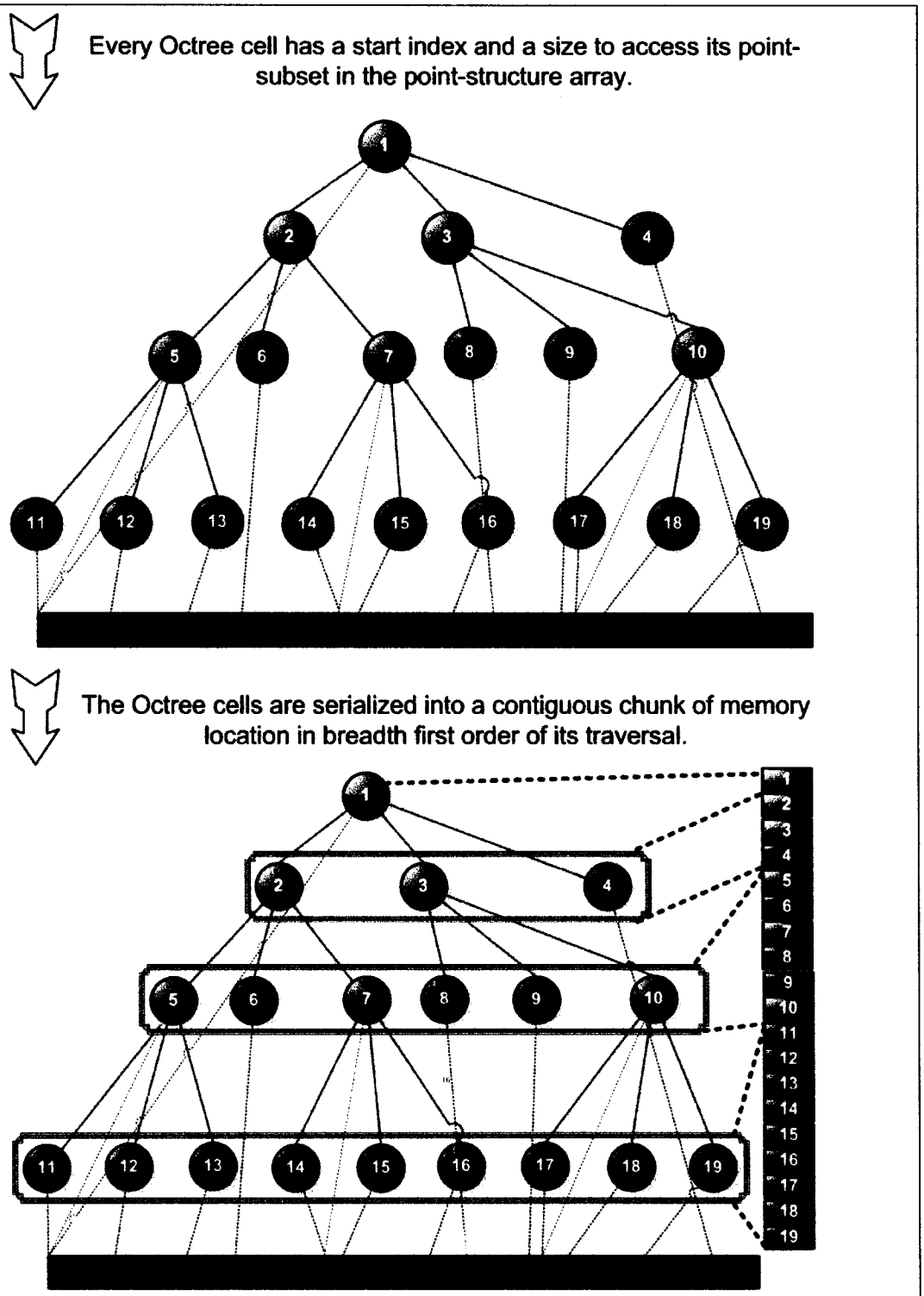


FIGURE 4-4C ILLUSTRATION OF DATA DECOUPLING - 3

**We decouple the spatial organization (*octree*) from the data-set collection (*point-set*). This helps us create a Sequential Octree for efficient runtime traversal.**

## 4.3 Point Selection

Selection algorithms (Figure 4-5) are used to decide on the number of points which need to be splatted in a given rendering pass. This decision may vary depending upon a number of factors like features present in a region, screen space projection of an octree cell and visibility.

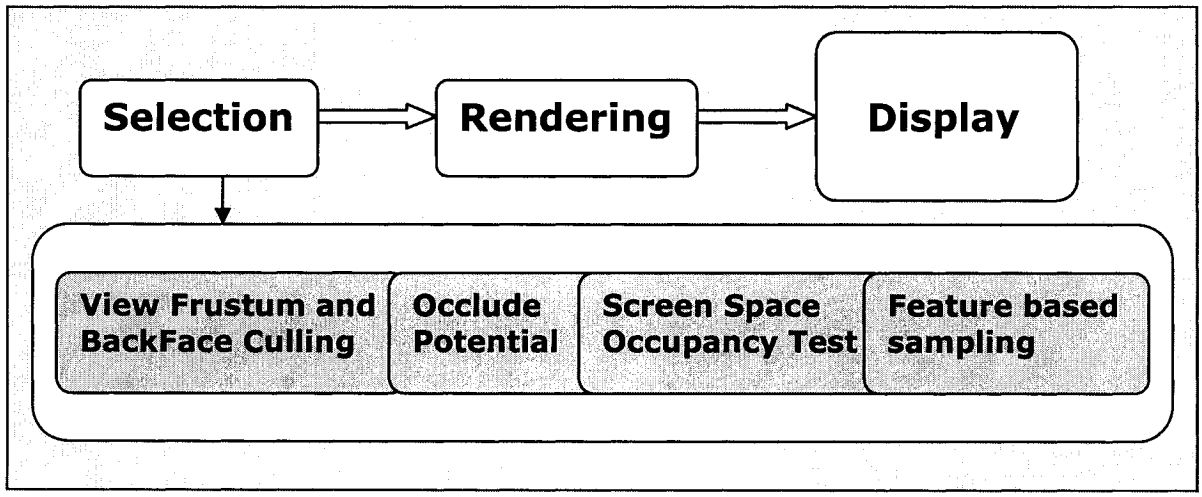


FIGURE 4-5 SELECTION PHASE ALGORITHMS VIEWED AS A PIPELINE OF TASKS

### 4.3.1 Feature based sampling - EVA analysis

The use of Eigenvalue Analysis (EVA) in feature detection is well known and has also found wide use in image processing and pattern recognition. In [52], their use for detecting features in point cloud models was proposed and demonstrated. Since the main concern in this thesis is distributed rendering of large point sampled surface models, we shall restrict our discussion only to those point model features concerned with rendering.

1. Region Flatness: Flatness in any region of a surface is a significant cue that can be used to optimize rendering. For e.g., if a model has no texture or color and only lighting effects (with the exception of specular) are needed, then clearly flat

regions can be rendered with fewer samples. Hence, octree traversal could stop at a higher level.

2. Edges and Corners: Another important aspect for high quality rendering is special treatment of edges and corners. Edges and corners are discontinuities which are important visual features and their presence/absence is easily noticed by the human visual system. As explained later, splatting unduly blurs edges and corners.

We can think of any given point-set in 3D-space as a sampling of a random vector variable P with scalar components $P_i$ corresponding to $x$, $y$, $z$ coordinates. Covariance matrix of P is defined as $V_{ij} = cov\ (P_i, P_j)$ where $cov\ (i; j)$ gives covariance between 2 given scalar variables. Since this $3 \times 3$ covariance matrix is symmetric, we can always decompose it to find 3 normalized eigenvalues $\lambda 0, \lambda 1, \lambda 2$ such that $\lambda 0 \leq \lambda 1 \leq \lambda 2$ and $\lambda 0 + \lambda 1 + \lambda 2 = 1$. Corresponding eigenvectors $e_0$, $e_1$, $e_2$ form a new basis for the given point set such that there is no correlation.

Normalized eigenvalues can be used to detect features in point sets. We would discuss three cases of our interest. A more detailed analysis is out of the scope of the thesis; we refer the reader to [34, 50].

1. $\lambda 0 \approx 0$. This means that all the data is nearly in a plane. In this case $e_0$ gives the normal direction to this plane. It should be noted that correct orientation of the surface normal is a global property [51].

2. $\lambda 0 \approx \lambda 1$ and $\lambda 2 \approx \lambda 0 + \lambda 1$. This means that the data is grouped in two parts separated by an edge. This gives us a test to determine edges which can be used in rendering. The edge direction is given by $e_2$.

3. $\lambda 0 \approx \lambda 1 \approx \lambda 2$ This condition indicates the presence of a corner.

Given a point and its neighborhood of point samples, the above conditions enable us to classify the point into one of three categories - ***flat, edge or corner***. Figure 4-6 shows the result of such classification on all the sampled points of three objects, a two-cube object, a sculptured surface and a mechanical component. The flat points (the largest number) are colored green, the edge points are blue and the corner points are red.

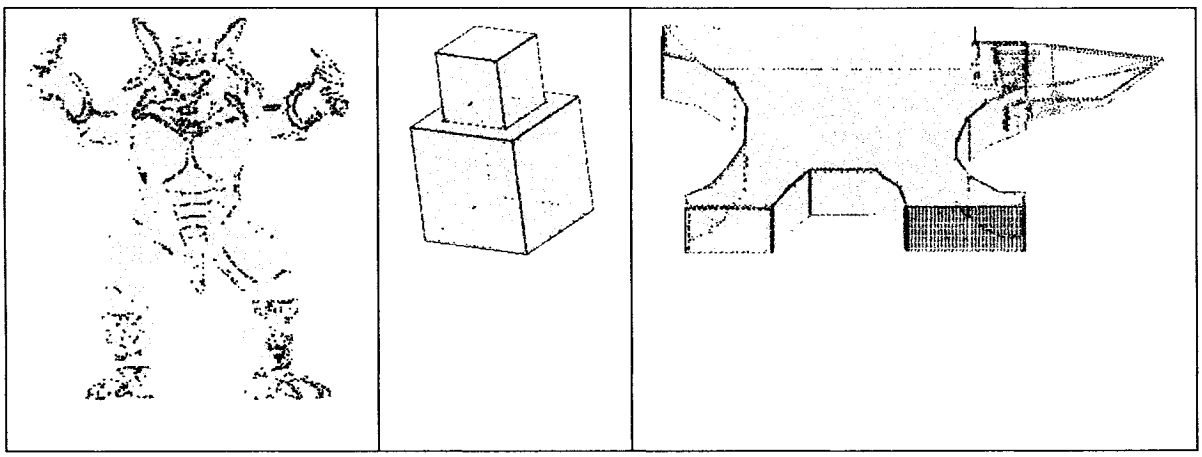

FIGURE 4-6 ILLUSTRATION OF EIGEN VALUE ANALYSIS ON A POINT-SET.

**Points in the flat, edge and corner region are colored red, green and blue in color respectively.**

### 4.3.2 Screen space occupancy test

The Sequential Octree is traversed once per frame, and for each cell we compute its image size projected onto the screen space. If its size is less than a *user defined pixel threshold N*, (e.g. 4 pixels), then the child nodes for this octree cell are not traversed. We

pick up $\lfloor N \times \ln N \rfloor$ random points [49] from the current cell for rendering. Calculating the accurate screen space projection involves expensive readback operations (e.g. employing an ARB_Occlusion_Query) on the framebuffer so an approximate value is generally calculated. Many different ways exist to calculate an approximate value for the screen space projection. We use the following measure inspired from [35, 48]:

*Cell's projected image size*[10] $\cong$ *(Cell Diagonal) / (View distance of the Cell's Center)*

### 4.3.3 Visibility culling

Points that do not contribute to the final image are generally 'culled' (or removed) from the rest before rendering. This is generally done for a collection of points rather a single point. It saves us the significant overhead of redundant calculations performed by the GPU on these points if they are sent down the rendering pipeline.

View Frustum Culling:

It uses the bounds of the view frustum to cull away points not lying within it. The view frustum is defined by a 'near' plane, a 'far' plane, and the four planes which pass through the camera position and the edges of the screen (Figure 4-7). We use an octree cell's center and size to determine whether or not the cell lies entirely outside one of these planes. If it does, then we do not need to render the points in it.

---

[10] Value in Normalized Device Coordinates.

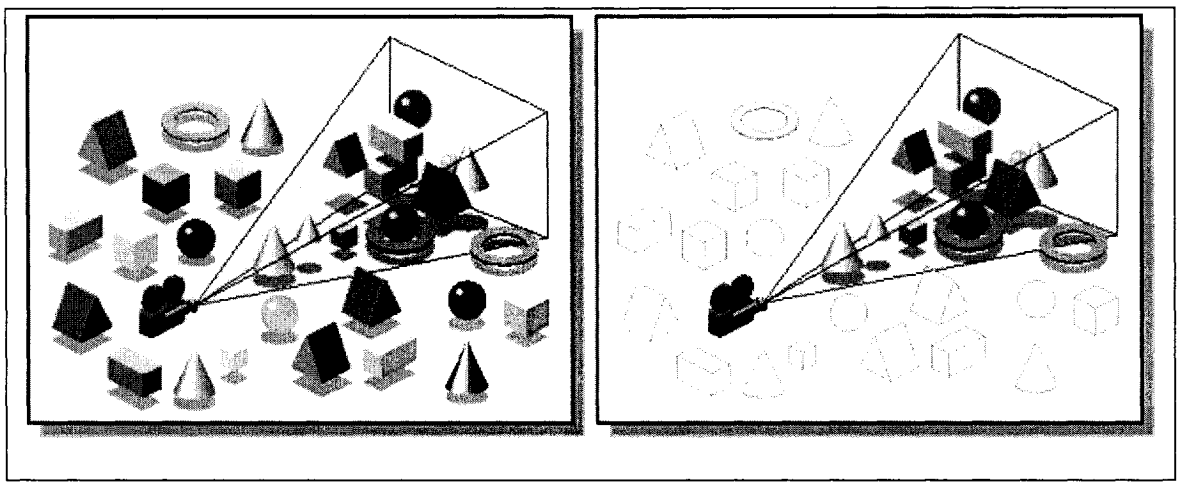

FIGURE 4-7 ILLUSTRATION OF VIEW FRUSTUM CULLING.

**[Image courtesy, Dr. A Kolb].**

Backface culling:

As with polygon rendering, backface culling holds for points too. There is no need to render points whose normals are pointing away from the viewer (Figure 4-8). In other words, the tangent plane of each point defines a Euclidian half space from which the point is not visible. Given a set of points, the intersection of their respective half spaces defines a region from which none of the points is visible. This would be particularly true at the lower level octree cells including the leaves. The visibility of the points in such cells from a given viewpoint could be decided by testing the half spaces of all points individually. A better and efficient technique as suggested by [53], is to approximate the region defined by the intersection of the half spaces by a cone inside it called the visibility cone. It allows us to efficiently perform a conservative visibility test for all points of a given octree cell.

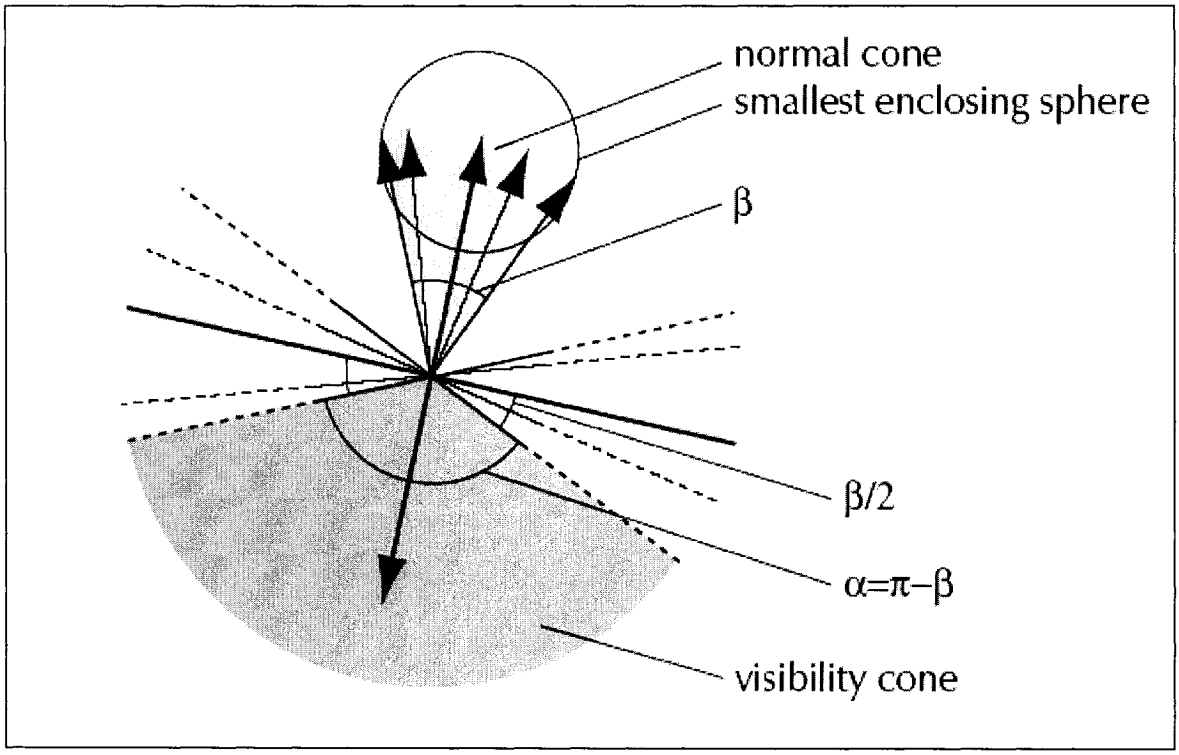


FIGURE 4-8 ILLUSTRATION OF A VISIBILITY CONE OF NORMALS.

**[Image Courtesy: Zwicker M]**

Occlude Potential:

Given any octree cell we compute it's occlude potential as follows: Let C1, C2,.. Cn be the octree cells that are in front of this cell along the view direction. The view direction is chosen as the line joining the cell center and the eye point. The total number of points in the cells C1, C2,.. Cn is directly used as a measure of the occlude potential of C.

We find that view frustum and backface culling cull away much of the unwanted points. This fact coupled with the time needed for implementing the added computational complexity involved in computing occlude potential made us decide not to implement this step in our current implementation.

## 4.4 Point Rendering

The final step in any graphics data processing is rendering. In our case it corresponds to pushing the selected points through the GPU to obtain an image rendered onto the display. The selection algorithms' results feed into this process and help us to pick a smaller number of points from the point-set. A contiguously stored set of points are picked from particular locations (as pointed from selected octree cells) of the point-array and rendered. Figure 4-9 captures the rendering tasks involved.

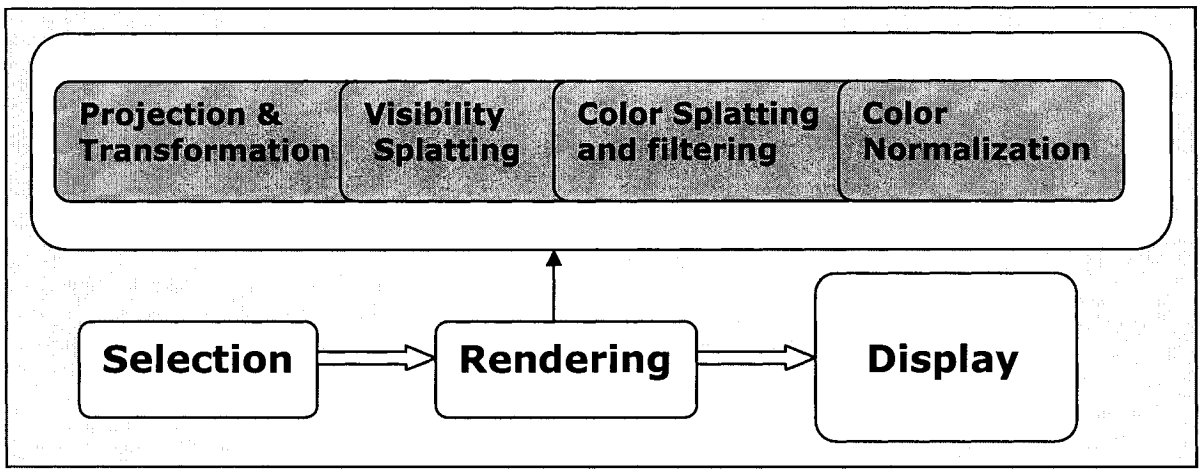


FIGURE 4-9 RENDERING PHASE ALGORITHMS VIEWED AS A PIPELINE OF TASKS

All the rendering tasks described herein are implemented over the GPU as vertex and fragment programs. This gives a much better performance as compared to a pure software implementation.

### 4.4.1 Splatting

In 1989, Westover [54], introduced *splatting* for interactive volume rendering. In splatting, each projected voxel (a unit volume measure used for rendering) is represented as a radially symmetric interpolation kernel, equivalent to a fuzzy ball. Projecting such a

basis function leaves a fuzzy impression on the screen, called a *splat*. Over the years splatting has emerged as a widely accepted technique for rendering point sampled surface. In point rendering, a splat is an extended region on the screen, centered about the rendered point. The extent of the region depends upon the point density in the neighborhood. The shape of the splat (circular, square or elliptical) depends on the kernel used to represent the point sample. Based upon the visual quality desired, different Gaussian filters can be applied as texture to the splats.

The requirement of watertight rendering considerably influences the spatial extent of the splat at any point being rendered. A conservative strategy adopted by all is to choose the splat shape dimension so as to cover the worst case. This is done by using the distance to the farthest neighbor. The extent of the neighborhood is pre-decided. It could be any one of the following:

1. fix a value k, and use k-nearest neighbors,

2. fix a radius depending on the surface sampling frequency (local or global) and choose all the points within a sphere of that radius centered at the point,

3. fix a radius depending on differential geometric properties such as curvature at that point.

We have chosen option 1 and calculated the splat radius based on the farthest of the k-nearest neighbors.

### 4.4.3 Three pass approach for rendering

In the absence of adequate hardware support for direct point rendering algorithms, a three pass rendering approach is commonly following in point based rendering [55]. In each pass a distinct set of tasks outlined in figure 4-9 is accomplished.

Visibility Splatting: In the first pass we perform visibility splatting. We render a depth image of the object into the z-buffer such that it holds only the splats representing the visible surfaces and does not contain any holes. This depth image will be used to control the alpha blending of the filtered splats in the second rendering pass. To render a point-based object without artifacts, we must accumulate all the splats of the visible surface and discard all other splats. During the second pass of the rendering, we decide for each pixel whether to discard or accumulate a splat's contribution by comparing the depth value of the splat with this depth image. We add a z-offset to make sure that splat fragments covering the same surface portion are not accidentally discarded. To ensure correct occlusion, each point in the scene is moved along the viewing ray [62], as illustrated in the figure 4-6. The offset value $\varepsilon$ could vary for different objects depending upon their surface curvatures. Then the object is rendered in the z-buffer after a traversal of the Sequential Octree. For each flat node (satisfying the criteria sub-section 4.3.1), an opaque quad whose orientation is defined as eigenvector $e_0$ is used as the splat rendering primitive. It may be recalled that eigenvector $e_0$ is close to the normal direction of the surface in that part. For non-flat nodes, we splat all the points delivered from the selection stage. It may be noted that our flat octree node represents a subset of sample points of the original model. By using a single quad for flat nodes, we achieve the same

effect as surface splatting, which would have to use as many splats as the number of points in the node.

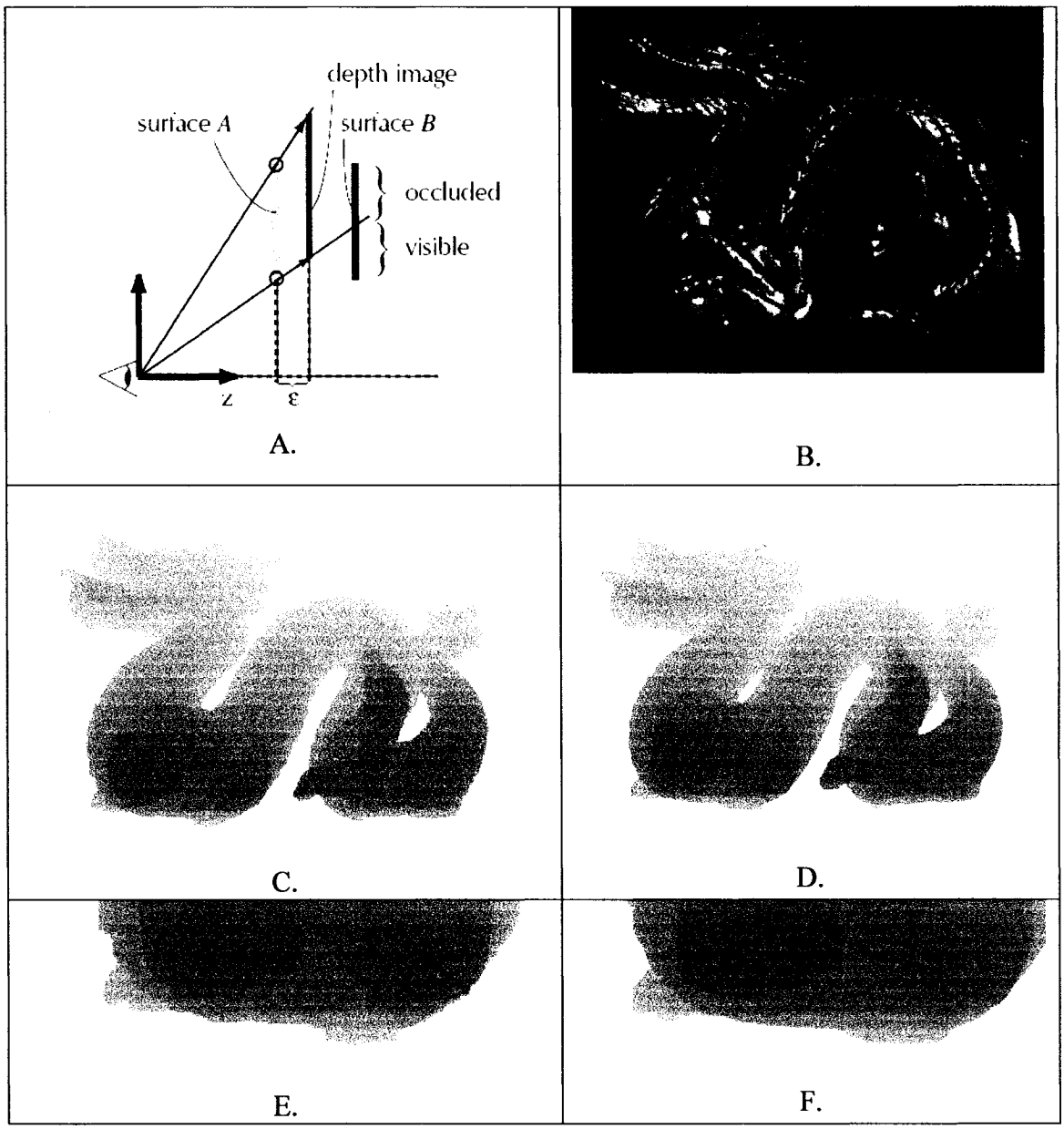


FIGURE 4-10 ILLUSTRATION VISIBILITY CULLING.

**The figure demonstrates the z-buffer i.e. the depth values as grey-coded values.**
**A - applying the depth offset; B- The rendered model: Stanford Dragon;**
**C, D – Grey-coded z-buffer values after the visibility splatting pass with feature driven visibility splatting and normal visibility splatting respectively.**
**E, F – Zoomed in view of the z-buffer values for C and D respectively.**
**Please note the jagged edges which represent the approximate visible splats are caused due to the flat quads that are rendered for the z-offsetting. They don't cause any visible artifact in the rendered model B.**

Figure 4-10 shows the z-buffer as a pseudo-image for comparing our *feature driven visibility splatting* with the conventional approach that uses the same splats in both passes.

The left image shows the contents of the z-buffer for the Stanford dragon model after the first pass using *feature driven visibility splatting* whereas the right image shows the z-buffer for the complete traversal for the Sequential Octree without any consideration for flatness. It may be noted that except at the silhouettes, the z-buffers are the same. This is as expected. From the statistics shown in Table 4-4, we can see that our algorithm renders about 70 percent less splats in first pass than second pass. This results in rendering speed improvements of over 60 percent. For higher density models, this gain will be even more significant, as each planar octree cell will have a larger number of sample points associated with it.

| **Model rendered at 1024 X 1024 res.** | **No. of Flat Octree Cells** | **No. of points in Flat Cells** | **Points used in visibility splatting** | **% reduction w.r.t 2nd pass.** |
|---|---|---|---|---|
| Stanford Dragon (437645 points) | 54418 | 380467 | 111596 | 74.50 |
| Stanford Buddha (543652 points) | 127199 | 508405 | 162446 | 70.12 |

TABLE 4-4 PERFORMANCE GAIN WITH FEATURE DRIVEN VISIBILITY SPLATTING.

**The above readings are with the models renderded at a screen resolution of 1024 X 1024. At this resolution the second pass (color splatting and filtering pass) of the rendering phase would typically use all the points of the dataset. Then the number of points used in the visibility splatting pass is over 70% less in this case as compared to a naïve visibility splatting approach.**

Color splatting and filtering: In the second pass, actual blending of fragments takes place. During this pass, lighting and color buffers are enabled. z-Buffer is made read only to ensure that only fragments belonging to same surface portion are blended together. The flat octree nodes are sampled a little more stringently than in the first pass so that we are able to capture details such as color variation and texture. Next, the contributions from each point splat are blended to get the rendered image. In case the node being rendered has an edge or corner present, we increase the sampling rate by choosing more points from the octree cell. For anti aliasing the rectangular screen space splat, a filter based on a radially decreasing Gaussian function is pre-computed and loaded into the texture memory. The splat filter is finally alpha blended.

Normalization: In the third pass, normalization (weighted additive blending of colors) is performed to ensure correct color ranges for the pixel, independent of the number of fragments contributing to each pixel.

### 4.4.4 Rendering Algorithm

After examining the stages of our point rendering application the rendering algorithm could be stated as in Table 4-5. It should be noted that the octree used in the algorithm itself is a Sequential Octree. The hierarchy is needed for visibility culling. Also the points referred to are picked up from the point-array.

```
void traverseOctree(Octree*  Cell){
   if(Cell is NULL)
      return;
   else if(Cell is not visible)
      return;
   else if(Cell is a Leaf)
      splat all points in the Cell.
   else if(screen space test and features present don't need recursion)
      splat candidate points from the Cell to ensure hole free image.
   else if(Cell is a parent)
      for each Child in Children(Cell)
            traverseOctree(Child)
      }
```

TABLE 4-5 RENDERING ALGORITHM.

# Chapter 5: Distributed Point Rendering

In the previous chapter we note that both the selection and rendering stage work with the spatial subdivision data structure. The rendering stage in addition needs the point-array. As noted by Levoy et al [35], the majority of time is spent in the selection stage of the pipeline. For large models the computational effort is enormous. We also note the performance degradation resulting from cache thrashing (rows 1 and 2 of Table 4-3) when we toggle between the Sequential Octree and the point-array during rendering. We propose to alleviate these bottlenecks by distributing the computations into multiple nodes.

## 5.1 Separating Selection from Rendering

The strongest reason for separating selection and rendering into separate nodes is the data structures they operate on. Comparing rows 1 and 2 of Table 4-3 immediately tells us that the hierarchical organization of data costs us nearly 3 times in performance over traversal of a flat array of points. This is in spite of the fact that we have decoupled the octree from the point-set and serialized the hierarchical octree into a Sequential Octree. Further performance optimizations are not quite fruitful as the problem lies in the inherent runtime coupling between the usage of the octree and the point-set. Unavoidably, there is going to be cache thrashing again when we toggle back and forth between the point-array and the Sequential Octree array (as outlined by the rendering algorithm from Table 4-5). We notice that for rendering we just need the following pair:

***<An offset into the Point-array, Number of points to render from the offset>***

We utilize this knowledge to decouple the rendering process completely from the octree. Hence we perform rendering from the point–array on a separate node. In fact the renderer node works just with the point-array. After constructing the point-array during the preprocessing phase, it destroys the octree hierarchy as it no longer needs it. For every frame, the selection nodes send across the aforementioned pair in a network optimized packet. We notice a performance improvement of more than 2 times employing this clever strategy of separating the selection and rendering phases on separate nodes. Further, the selection nodes destroy the point-array after the construction of the octree.

## 5.2 Distributing the Computations

Figure 5-1 illustrates how the computations are split among different nodes of a cluster. The nodes are connected in a pipelined fashion. Each node operates concurrently on an incremental frame. This means when node 3 is rendering $f_i$, node 2 is computing $f_{i+1}$ and node 3 is working on $f_{i+2}$. Although this causes a 2-frame delay in the rendering node, we observe that the benefit we achieve in overall frame rate offsets this delay by a large margin. The point-set data is replicated on each node to avoid expensive geometry data distribution at runtime.

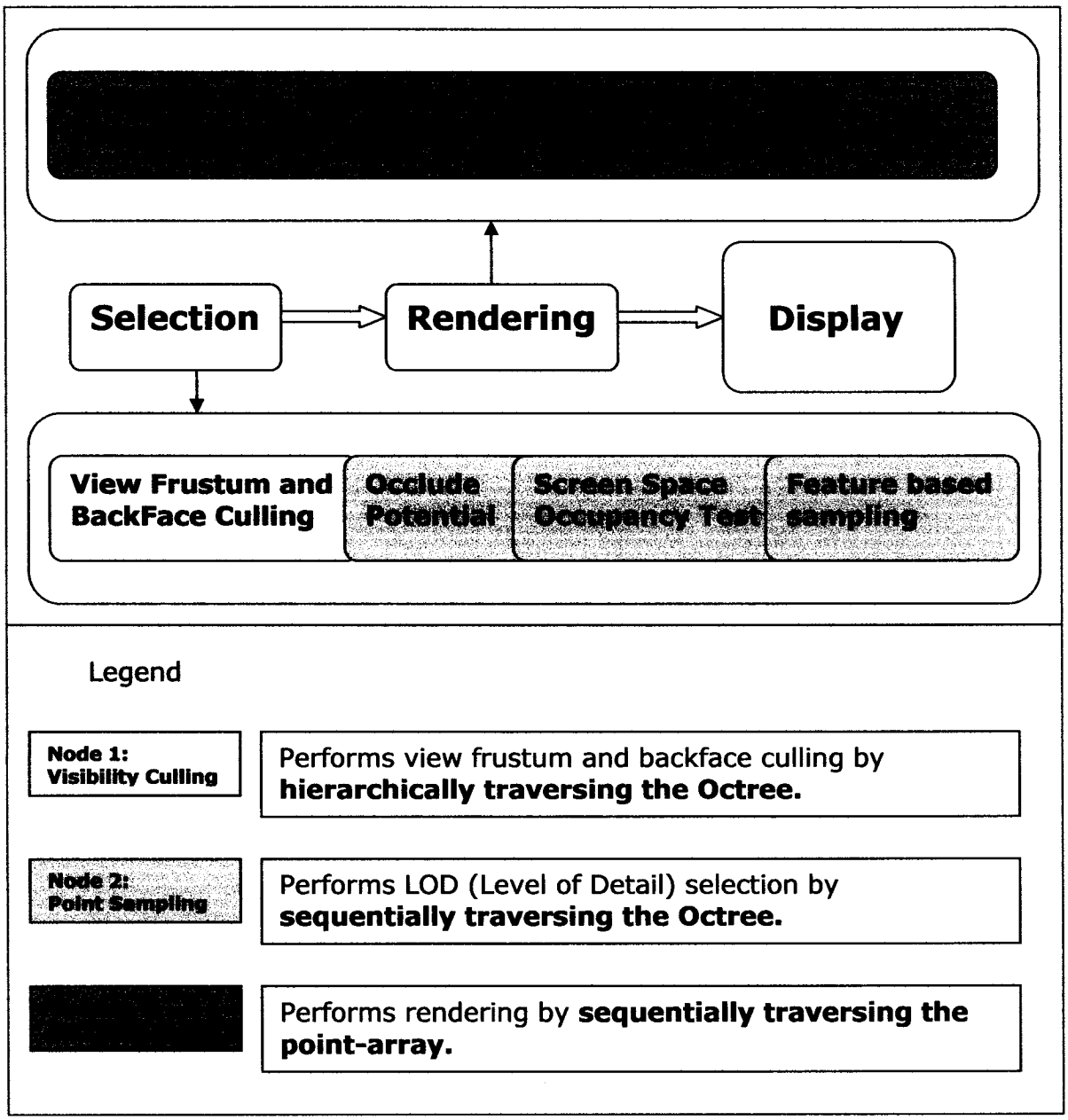


FIGURE 5-1 DISTRIBUTING THE POINT RENDERING PIPELINE.

**The legend outlines the assignment of the respective computations and data structures to the different nodes of the cluster.**

### 5.2.1 Responsibilities of each node

Node 1: Visibility Culling: The first node performs view frustum and back face culling. It does a hierarchical traversal of the octree. For better locality we create a Sequential Octree (during application load time), although it is traversed hierarchically. After construction of the octree we unload the point-set from the memory. This gives a much smaller memory footprint to the culling process.

A *bit-based array* representing the Sequential Octree is used as the network data structure. The array has as many bits as the number of octree cells. The complete octree is traversed irrespective of the visibility of each octree cell. When the octree cell lies completely inside (outside) the view frustum then its children are all marked as visible (invisible) by setting(resetting) the respective bit in the bit-array without further visibility testing on them. Similar handling is done for backface culling with the children of the octree cell when the visibility cone of the candidate cell is pointing entirely towards (away from) the viewer.

The bit-array created as a result of visibility culling is communicated over the network to node 2.

Node 2: Point Sampling: This node takes in the bit-array sent in from node 1 and performs point sampling for octree cells that are indicated as visible through their respective bit positions in the input visibility bit-array.

Again we create a Sequential Octree at this node and subsequently unload the point-set data from the memory (during application load time). For every frame we do a sequential traversal of the octree and perform screen space occupancy test for each

visible octree cell until we arrive at the level of detail beyond which it is no more beneficial to traverse further down the array. As noted by Dachsbacher et al [56] the candidate octree cells from which the points needs to be picked for rendering lie as a contiguous segment in the Sequential Octree which we call the *LOD window* as shown in Figure 5-2. The LOD window is dependent on the zoom level employed for viewing the model. In our case we may also have some leaf octree cells up the sequential array before this LOD window as indicated in Figure 5-2. Also the LOD window could contain some culled cells as indicated by the visibility bit-array.

```
struct  RenderPointsStruct{
unsigned offset;
char range;
unsigned stride;
};
```

TABLE 5-1 DATA STRUCTURE FOR A NODE OF RENDER-POINTS ARRAY.

We represent the candidate points sampled from the LOD process as an offset and range into the point-array. This information is represented as an array of structures of type shown in Table 5-1. We call it the *render-points array*. Please note that this array captures all the points that need to be rendered for the given frame. The offset is relative to the start position in the point-array. Also as we do not ever go beyond a few dozen points per octree cell, we have chosen to represent the range as a single byte. Finally we have also stored a stride for rendering out of non-leaf nodes when the octree cells fall below a threshold size on the screen. The stride is used to hop around the *render-points array* from the offset to the range to pick up points to render. Altogether it amounts to 9 bytes per unit element in the *render-points array* (assuming 4 bytes for an unsigned int

and 1 for the char). This network data structure could be further quantized by encoding each offset relative to the previous cell's offset. This would give a more compact representation for *render-points array*. But we leave the `RenderPointsStruct` structure as it is, since we achieved good results without any compression.

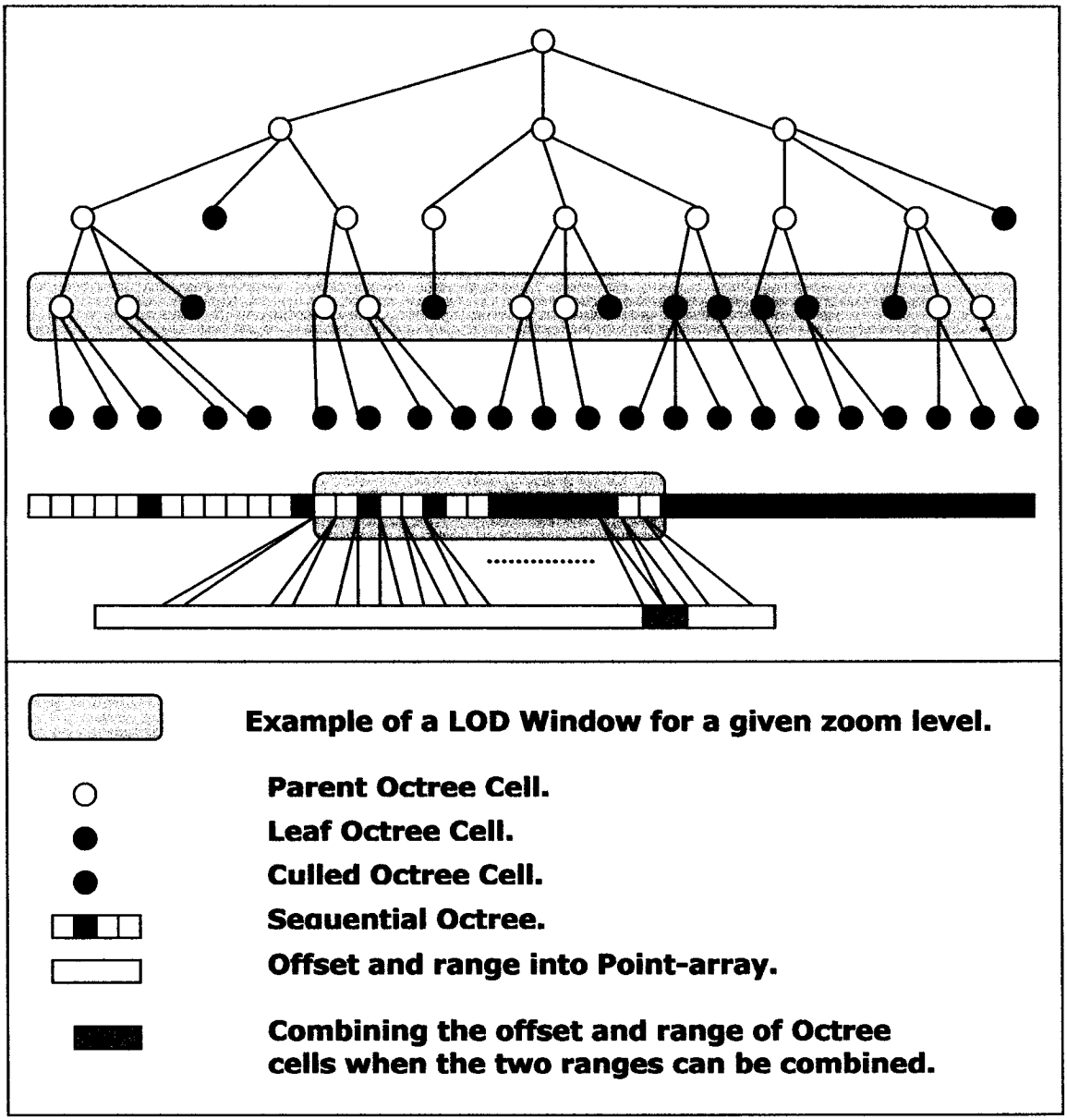


FIGURE 5-2 ILLUSTRATION OF THE LOD WINDOW AND CULLED OCTREE CELLS.

**For octree cells in the LOD window where the range and offset point to set of locations immediately adjacent to each other in the *point-array*, they could be combined into a single offset and range.**

Some octree cells in the LOD window will have their range and offset point to sets of locations immediately adjacent to each other in the point- array. These could be combined into a single offset and range. This condition is quite frequent when the user is viewing the model closely. In such cases we may require to render all the points from the leaf octree cells. By combining various contiguous ranges of point locations into a single offset and range we send lesser data over the network. Also the renderer node benefits by accessing a larger chunk of points from its memory.

On the downside, currently we have not worked out the techniques needed to incorporate EVA analysis results into the existing implementation. The use of EVA results in the rendering process required the Eigen values and vectors to be stored per octree cell. As our renderer has no knowledge of a spatial subdivision, we are unable to use EVA results. Hence feature driven visibility splatting and rendering is not performed.

Node 3: Rendering: For each frame the render node receives a *render-points array* from node 2. For each entry of the *render-points array*, the candidate points are indexed by the `RenderPointsStruct.offset` into the point-array. The corresponding `RenderPointsStruct.range` tells us the number of points to be picked up contiguously from the indexed location. The rendering task at node 3 is reduced to packing for each entry of the *render-points array*, a set of points into the vertex buffer. The vertex buffer is sent across to the GPU for rendering. Toggling between the compact render-points array and point-array gives better cache hit.

### 5.2.2 Distributed rendering algorithm

```
void traverseOctree(Octree*  Cell){
   set the visibility bit-array element for this octree cell to 1;
   if(Cell is NULL)
      return;
   else if(Cell is not visible)
      set the visibility bit-array element for this octree cell to 0;
   else if(Cell is a parent)
      for each Child in Children(Cell)
            traverseOctree(Child)
      }
```

**A. Hierarchial Traversal of Sequential Octree. (At Node 1.)**

```
for(int i = 0, j = 0; i <= number of Octree Cells; i++){
   if(visibility bit-array[i])
      if(screen space test and features present don't need recursion){
         render-points-array[j].offset = cell[index].offset;
         render-points-array[j].range = cell[index].range;
         render-points-array[j++].iStride = Octree.ThresholdNumPoints;
      }
     if(cell[i] is a leaf node){
         render-points-array[j].offset = cell[index].offset;
         render-points-array[j].range = cell[index].range;
         render-points-array[j++].iStride = 1;
      }
  }
```

**B. Sequential Traversal of Sequential Octree using visibility-bit array. (At Node 2.)**

```
for(i = 0; i <= number of render-points-array elements; i++){
      int index = render-points-array[i].offset;
      int count = render-points-array[i].range;
      pack into vertex buffer count points from point-array[index];
}
Send the vertex buffer to the GPU for rendering.
```

**C. Sequential Traversal of Render-points array. (At Node 3.)**

TABLE 4-5 DISTRIBUTED RENDERING ALGORITHM.

**A,B and C represent the distributed functionality on nodes 1, 2, and 3 of our cluster respectively.**

After examining the distribution of computations of our point rendering application, the rendering algorithm at each node could be stated as in Table 4-5.

## 5.3 Scalable Architectures

Distributing the point rendering pipeline on the basis of functionality helps us achieve better frame rates for large geometric models (of sizes close to a million points; discussed in Chapter 6). But we observe that the octree grows with larger data models. With the flatness criteria from EVA analysis, the octree expansion can be considerably contained for models with large flat surfaces. This is achieved by restricting further subdivision of flat octree cells (refer to section 4.3.1) irrespective of the number of points enclosed in them. But with models of considerable geometric complexity (larger number of edges and corners), we can not avoid the octree expansion. Also the flatness criteria based octree construction does not work for our current distributed implementation as mentioned earlier in section 5.2.1. As the models grow in size, incorporation of data parallel approaches is inevitable. Below we discuss some data parallel strategies that could be implemented over a visualization cluster.

### 5.3.1 Incorporating data parallelism

Each of the selection stages which operate on the octree could be split into multiple nodes as shown in Figure 5-3. The first node of our distribution performs *visibility culling*. It traverses the octree hierarchically and generates the *visibility-bit-array*. To distribute the task of this node different sub-trees of the octree are assigned to multiple nodes. Depending upon the data size we can distribute the first level of the

octree cells or the second (the latter case would be rare). In Figure 5-3 the visibility culling calculations on the first level sub-trees are distributed.

To achieve a fair distribution, on application instancing the first level octree subdivision is created such that each octree cell receives equal number of points. As visibility calculations are view dependent runtime load balancing will give better results.

The results from each visibility node are fed into a *point sampling* node. The point sampling node works with a sequential version of the same sub-tree as the previous *visibility culling* node. It calculates the *render-points-array* for the given sub-tree. The *render-points-array* is fed to the next stage for rendering.

The rendering stage can be arranged in a sort-first configuration (refer to section 2.1). The screen space occupancy test of the previous *point sampling* node can be used to decide the render nodes to which the contents of the *render-points-array* needs to be distributed. The images generated by the render nodes are fed to a display wall or composted into a single node.

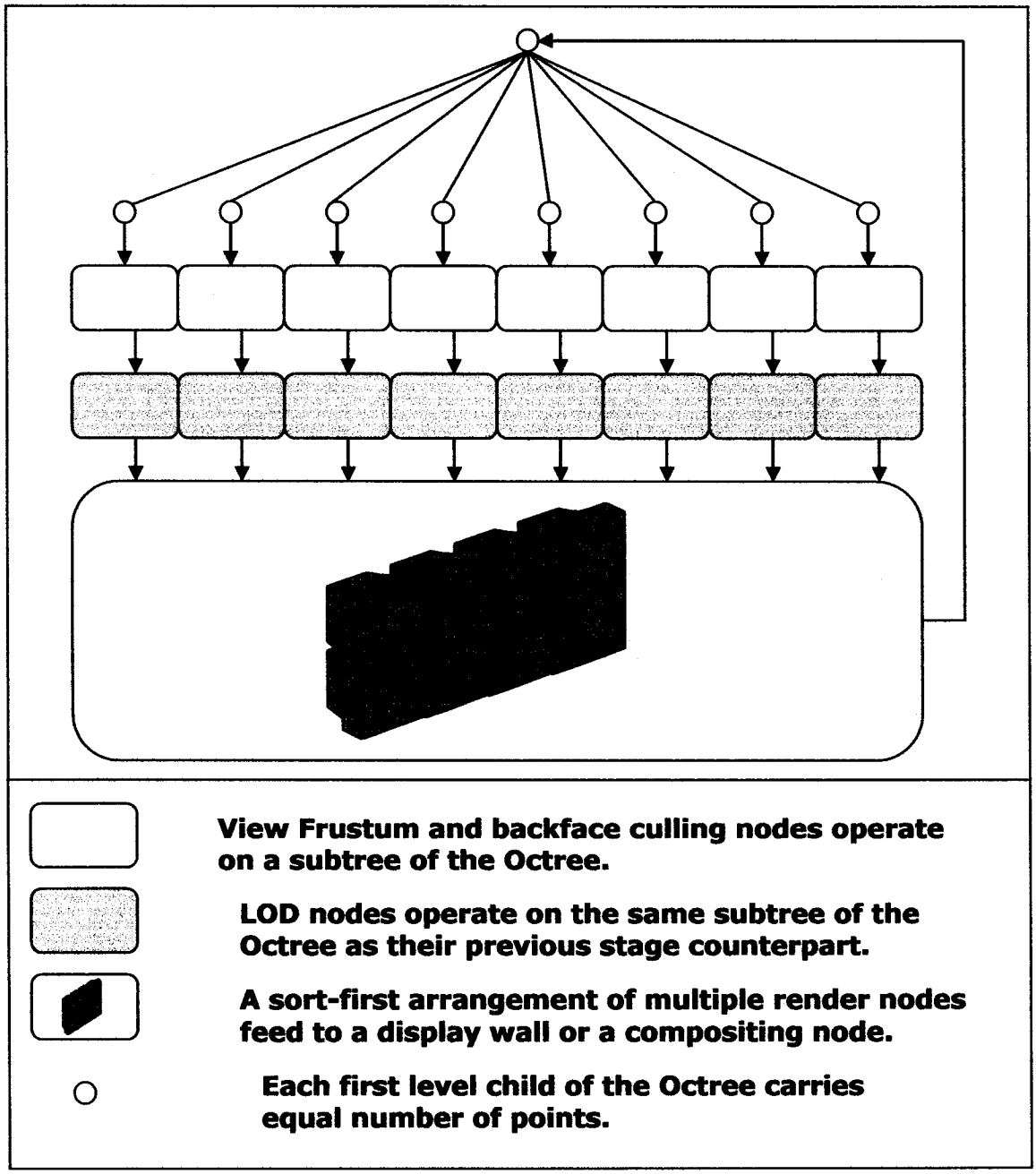


FIGURE 5-3 SCALING THE FUNCTIONALITY DISTRIBUTED POINT RENDERING.

**We employ data parallelism in the individual stages of our pipelined model to scale the distribution for larger models.**

### 5.3.2 Out-of-core strategies

Our current distribution scheme allows us to exploit excellent out-of-core strategies on the render node. After constructing the point-array in depth order traversal of the octree, we can serialize the *point-array* to the disk and unload it from the memory. In every frame, we can selectively load only those points from the disk which are needed. Due to frame-to-frame coherence we obtain large benefits, as the amount of data loaded from the disk in a frame is usually very low for incremental view changes. It should be noted that at application load time a fixed contiguous array is allocated in the memory to hold the points to be rendered. Points from the disk are loaded into this array.

### 5.3.3 Experimentation Plan

While we have not implemented the ideas in sub-sections outlined in 5.3.1 and 5.3.2 above, it is mainly due to the fact that the hardware setup available in our lab was limited to a cluster of just 4 nodes connected over a LAN. However, we do not see any major obstacle and are confident of implementing such architectures on a larger cluster.

# Chapter 6: Implementation, Results and Analysis

In this chapter we discuss the performance results achieved with the functionality distribution outlined in Chapter 5. We compare the implementation with a sort-first only implementation on a similar configuration. We also outline the minimal model sizes needed to achieve performance benefits from the distribution by comparing the results with a single node implementation. We subsequently model the performance at each of the nodes using the parameters stated in Chapter 3.

## 6.1 System Configuration

We use a cluster of 4 nodes. Each node is a Pentium-4 2.8 GHz, 512 K L2 Cache, 1 GB RAM with ATI Raedon 9800 128 MB graphics card, Catalyst 4.2 driver running Windows 2000. The backbone is supported by 100Mbps Ethernet connectivity. We have employed MPI over the cluster for communication among the nodes.

## 6.2 Results

### 6.2.1 Performance of functionality distribution

Table 6-1 shows the frame rates achieved for rendering models of different complexity. It compares the results on a single node implementation of a point rendering pipeline, functionality distributed rendering and a sort-first rendering as reported in [65]. For the latter two we compare the results from recently published results from [65]. We note (row 2 of Table 6-1 A and B) that our performance optimizations and functionality aware distribution gives us over three times the performance benefit. We compare the results for Turbine Blade model that the authors published in [65]. This is the largest

model they used for their cluster which they have organized in traditional sort-first configuration. For our other models we don't have the results to compare.

| Model Name | Model Size in # of Points | # of Octree Cells | Single Node (FPS) | Functionality Distribution (FPS) | Sort-First Distribution (FPS) |
|---|---|---|---|---|---|
| Stanford Buddha | 543,652 | 38,351 | 9.8 | 20.0 | Not Available |
| GeorgiaTech Blade | 882,954 | 65,209 | 8.0 | 12.5 | 3.5 |
| Stanford Bunny | 35,947 | 3,106 | 185 | 72 | Not Available |

TABLE 6-1 A. FRAMES PER SECOND FOR OCTREE WITH THRESHOLD PIXEL SIZE OF 4 X 4.

| Model Name | Model Size in # of Points | #of Octree Cells | Single Node (FPS) | Functionality Distribution (FPS) | Sort-First Distribution (FPS) |
|---|---|---|---|---|---|
| Stanford Buddha | 543,652 | 176,387 | 5.5 | 9.15 | Not Available |
| GeorgiaTech Blade | 882,954 | 267,793 | 3.78 | 6.8 | 3.5 |
| Stanford Bunny | 35,947 | 12,626 | 83 | 70 | Not Available |

TABLE 6-1 B. FRAMES PER SECOND FOR OCTREE WITH THRESHOLD PIXEL SIZE OF 1 X 1.

**Rendering performance in frames per second for different data models over a single node renderer, functionality distributed renderer and sort-first renderer. The Readings are for two different octree sizes.**

From the first row of Table 6-1A we note that we obtain twice the performance over a single node when we use functionality distribution. We also note that as the model size reduces the benefit of a performance distribution is offset by the overhead in communication (row 3 of Table 6-1A). Another interesting point to note is the MAX_POINTS_PER_OCTREE_CELL (please refer to section 4.2) used to create the octree. This number is set a little higher than the number of points needed to splat for the screen space threshold number of pixels (please refer to section 4.3.2). For smaller pixel thresholds we end up creating larger octrees which might be needed to get a better

sampling of a region on the surface. But it results in larger data packets getting transmitted over the network from the point sampling node to the renderer node per frame (compare $D_e$ column of table 6-2 A and B for any given model.). This explains the drop in performance between Table 6-1 A and B. The single node performance drops too as the recursion level of the hierarchical octree increases with a decreased pixel threshold value. It must be recalled that we have to perform a hierarchical traversal on the single node because the visibility calculation makes them mandatory. So all the performance gains of using a Sequential Octree as stated in chapter 4, are not achieved.

| **Screen Resolution** | **Stanford Buddha** | | **GeorgiaTech Blade** | | **Stanford Bunny** | |
|---|---|---|---|---|---|---|
| | **FPS** | $D_e$ | **FPS** | $D_e$ | **FPS** | $D_e$ |
| 1024 | 20.0 | 5.7 | 12.5 | 5.8 | 72 | 0.4 |
| 512 | 19.9-21.6 | 5.8-5.9 | 12.5-12.8 | 5.8 | 64-78 | 0.4 |
| 256 | 29.1-36.2 | 5.8-6.0 | 38 | 10.0 | 70-88 | 0.4 |
| 128 | 65-75 | 1.7-2.5 | 51 | 0.6 | 86-110 | 0.4 |

TABLE 6-2 A. DATA SENT PER FRAME: OCTREE WITH THRESHOLD PIXEL SIZE OF 4 X 4.

| **Screen Resolution** | **Stanford Buddha** | | **GeorgiaTech Blade** | | **Stanford Bunny** | |
|---|---|---|---|---|---|---|
| | **FPS** | $D_e$ | **FPS** | $D_e$ | **FPS** | $D_e$ |
| 1024 | 9.0–9.2 | 30.9-31.1 | 6.8 | 29.8 | 70 | 1.3-1.62 |
| 512 | 8.8-8.9 | 31.6-32.2 | 6.6-6.7 | 30.2 | 65-81 | 1.62 |
| 256 | 14.4-15.6 | 24.5 | 6.5-8.2 | 30.8-53.5 | 78-100 | 1.62 |
| 128 | 17.4-17.8 | 2.9-3.9 | 9.7-11.6 | 2.84-30.1 | 140-167 | 1.3-2.2 |

TABLE 6-2 B. DATA SENT PER FRAME: OCTREE WITH THRESHOLD PIXEL SIZE OF 1 X 1.

**Maximum Data transferred between two nodes (in our case between the point sampling and renderer nodes), every frame for different resolutions of the model. $D_e$ above is measured in terms of number of *renderArray* structs transmitted per frame.**

### 6.2.3 Measuring performance parameters

Table 6-3 lists the system parameters of the nodes of our distribution cluster.

| Sr. No. | Parameters | System Parameters |
|---|---|---|
| 1 | $S_{GPU}$, $S_{CPU}$ | 3122.75 MFLOPS, 265.4 MFLOPS |
| 2 | $Mem_{mm}$ | 1 GB |
| 3 | $Mem_{ca}$ | 512 K |
| 4 | $Mem_{vdo}$ | 128 MB |
| 5 | $l_d$, $B_d$ | 5.126 ns, 736.794 MB/sec |
| 6 | $B_u$ | 113.87 MB/sec (with RGBA as floating point values) |
| 7 | $B_m$ | 1265.2 MB/sec (uncached), 1436.4 MB/sec (cached) |
| 8 | $l_e$, $B_e$ | 0.375 ms/byte, 10.75 MB/sec |
| 9 | $P_n$ | 3 |

TABLE 6-3: SYSTEM PARAMETERS OF A NODE OF THE GRAPHICS CLUSTER.

We analyze the pipelined model with the help of the Georgia Tech Blade model which has 882,954 points and set our desired frame rate at 15 FPS. The threshold pixel size is set to 4 X 4. Table 6-4 lists the application parameters of our interest. Each octree cell occupied 112 bytes.

| Sr. No. | Parameters | Visibility Culling Node (1) | Point Sampling Node (2) | Rendrer Node (3) |
|---|---|---|---|---|
| 1 | $D_{FB}$ | 16 MB | 16 MB | 16 MB |
| 2 | $\eta$ | 0 | 0 | 0 |
| 3 | $D_{CP}$ | 7.027 MB/frame | 7.773 MB/frame | 24.323 MB/frame |
| 4 | $D_{VP}$, $D_{FP}$ | 0 | 0 | 23.577 MB, variable |
| 5 | $O_{CPU}$ | log(n)/frame | (log(n) + n)/ frame | log(n)/frame |
| 6 | $O_{VP}$, $O_{FP}$ | 0 | 0 | 0.4371giga execs/s, O(1) |
| 7 | $F$ | 15 | 15 | 15 |
| 8 | $N_{FB}$ | 0 | 0 | 0 |
| 9 | $D_e$ | 63.68 KB | 63.68 KB, 117.19 KB | 117.19 KB |

TABLE 6-4: APPLICATION PARAMETERS FOR EACH NODE OF THE GRAPHICS CLUSTER.

**The *n* in $O_{CPU}$ refers to the number of octree cells.**

We now refer to the equations from section 3.3.1 and model the performance of the nodes of our cluster.

- In our distribution the frame buffer is not read at any node; hence equation (1) is by default met.
- For equation (2) we measure the available external bandwidth and latency as reported in Table 6-3. We observe that for maintaining 15 FPS we need the transmitted data to not exceed the following: $((1 - 15 \times 0.375 \times 10^{-3}) \times 10.75 \times 1024 \times 1024) / 15 =$ 730.417 KB/frame. Row 9 of table 6-4 shows that we are reasonably within this limit for all nodes.
- For equation (3), we measure the complexity of our vertex shader and find out that by storing our data in video memory we achieve an $O_{VP}$ value of 0.4371 G executions/sec. From equation (3) we find out the upper bound on $D_{VP}$ to be 29.14 million points (0.4371 / 15). Row 4 of table 6-4 shows that we are reasonably within this limit from the data perspective on node 3. As for nodes 1 and 2 we barely use the GPUs of those nodes. For rendering we pack indices into pointArray stored in the video memory and use vertex arrays and send them over to the GPU for rendering. Using equation (4) we get the upper bound on data transmitted to the GPU per frame as $((1 - 15 \times 5.126 \times 10^{-9}) \times 736.794 \times 10^{6}) / 15 = 49.12$ MB per frame. We are transferring at the maximum 23.577 MB/frame from the CPU to the GPU which is well within the limit.
- As our fragment shader just outputs the final color so its inherent complexity is negligible. Hence equation (6) is not violated.
- With a memory bandwidth (uncached) of 1265.2 MB/sec accessing a maximum of 23.577 MB/frame for 15 frames should not be a trouble on node 3. As for node 1 and 2 they access the Sequential Octree where the maximum data accessed could be 6.965

MB/sec. This again is within tolerable limits. Hence equation (8) is not getting violated anywhere.

- We now turn our attention to equation (7) and try to evaluate the performance of the CPU computations. When comparing the three computational complexities at nodes 1, 2 and 3 (row 5 of Table 6-4), we find that node 2 has the highest computational complexity. As the computations involve many function calls, we don't go for modeling the accurate FLOP value, instead we isolate the node and test the time taken for its computations explicitly. We find that it takes the maximum computation time for the worst case FPS (12.5 in our case). We also find that at medium resolutions like 512 X 512, node 3 has to hop about the *renderPointArray* as many parent octree cells fail the pixel threshold test. This results in a reduction of performance too. We can't do much to alleviate this problem.
- Conclusion: The performance of node 2 is the bottleneck. The CPU code, executed per frame on it, needs optimization. Compression schemes for sending *renderPointsArray* will be useful at higher volumes of data transfer as seen in Table 6-2 B.

We employed GPUBench [68] for measuring some of the GPU instruction speeds, Performance Test [69] to measure memory and CPU performance. Rest of the system and network resources were measured using our own performance measurement programs. We note that performance modeled is not a complete picture of the entire computation. It models the performance of individual application components in isolation. Though we are not able to accurately model the performance of each node, we are able to capture the system bottlenecks instantly with the help of our performance modeling.

## 6.3 Implementation Challenges

Implementing an efficient point based rendering pipeline was a sizable challenge. It involved experimenting with many new algorithms on upcoming graphics hardware. In the absence of a standard framework to assist in programming and debugging graphics hardware programs, ad-hoc experimentation was unavoidable, but also resulted in very good learning experience.

The existing implementation went through three rounds of versioning and stands at 25 KLOC. In addition many numerous proofs of concepts have been carried out to test whether a particular feature is supported by the graphics hardware or not.

# Chapter 7: Conclusion

The graphics performance of GPUs, which are now part of virtually every computer workstation, has been growing even faster than Moore's law for CPUs. In terms of sheer floating point computational speed (FLOPS), today's GPU can outperform the CPU in most workstation by a factor of seven or more. The programmability that has been introduced in GPUs, a recent trend in graphics hardware, now makes it possible to offload application specific computational functionality to the GPU, thus enabling functionality distribution among the CPUs and GPUs available in a graphics cluster. The focus of our research has been to study this type of functionality distribution for large graphics applications. While there have been quite a few attempts to program GPUs with special algorithms, ours is the first research investigation that has tried to address the problem of functionality distribution in a more general setting of a graphics cluster. It is this investigation that has led us to formulate our thesis that functionality distribution achieved by programming multiple GPUs combined in effective ways with traditional sort based data parallel approaches provide scalability to an existing data parallel scheme.

In chapter 3 we have proposed different architectures for configuring the nodes in a graphics cluster, enabling us to combine functionality distribution with data parallel approaches. Of course, understanding the application is important to make the optimal choice of architecture and the allocation of computations to the different processing units. We have defined a comprehensive set of system and application parameters that can be used to model the performance of a distributed application. Later in chapter 6, we show the application of this performance modeling for the pipelined architecture implemented

by us for our point based rendering application. The benefits of a good performance evaluation model are two-fold. One, it would help in designing better load balanced applications and two, it would significantly decrease the programming efforts involved in post deployment optimization.

Functionality distribution becomes advantageous primarily due to the flexibility provided by programming the GPUs of a cluster and by organizing data for better cache hit. In chapter 4, we analyze these issues in detail for our point based rendering application and present a number of innovative solutions which demonstrate very well the advantages of cache-conscious organization. Our ideas on feature driven visibility splatting and rendering are yet another original contribution of this thesis. We have demonstrated the performance gains of such a strategy in this chapter. Presently, the cache conscious data distribution scheme does not allow us to use Eigen value analysis results in the renderer node. However, we plan to investigate this in our future research and extend the EVA results over our clustered implementation.

Chapters 5 and 6 amply demonstrate the effectiveness of such a distribution, wherein we show that with just 3 nodes we clearly outperform a sort-first configuration by a factor greater than 3.

In future we would also incorporate data parallelism in our distribution as noted in Chapter 5 and develop core strategies for very large models (> 100 million points). We also strongly feel that a central monitoring system like DAMon [66] would be crucial to monitoring the runtime performance of each node of our cluster as it grows. It will be a valuable addition for monitoring an application's performance over a graphics cluster.

# Bibliography


[1] D. Kirk. Innovation in graphics technology. Talk in Canadian Undergraduate Technology Conference, 2004.

[2] ATI X800 (http://www.ati.com/products/radeonx800)

[3] nVIDIA GeForce 6800 technical specification (http://www.nvidia.com/page/geforce_6800.html)

[4] ACM Workshop on General Purpose Computing on Graphics Processors, Sponsored & Co-located with ACM SIGGRAPH, (editors. A. Lastra, M. Lin and D. Manocha) August 7-8, 2004.

[5] General-Purpose Computation Using Graphics Hardware (http://www.gpgpu.org)

[6] OpenGL Shading Language version 1.10 (http://www.opengl.org/documentation/oglsl.html)

[7] "Microsoft, 2003. High-level processing unit language." (http://msdn.microsoft.com/library/default.asp?url=/library/enus/directx9c/directx/graphics/reference/Processing units/HighLevelProcessing unitLanguage.asp)

[8] W. R. Mark, R. S. Glanville, K. Akeley, and M. J. Kilgard. Cg: a system for programming graphics hardware in a C-like language. ACM Trans. Graphics. (SIGGRAPH), 22(3):896–907,2003

[9] "Brook for GPUs: Stream Computing on Graphics Hardware" -Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan in SIGGRAPH 2004 Proceedings.

[10] G. Humphreys, M. Eldridge, I. Buck, G. Stoll, M. Everett, and P. Hanrahan. Wiregl: a scalable graphics system for clusters. In Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques(SIGGRAPH), pp. 129–140, 2001.

[11] G. Humphreys, M. Houston, R. Ng, R. Frank, S. Ahern, P. D. Kirchner, and J. T. Klosowski. Chromium: a stream processing framework for interactive rendering on clusters. In Proceedings of the 29th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH), pp. 693–702, 2002.

[12] N. K. Govindaraju, A. Sud, S.-E. Yoon, and D. Manocha. Interactive visibility culling in complex environments using occlusion-switches. In Proceedings Symposium on Interactive 3D Graphics, pp. 103–112, 2003.

[13] GPU Cluster for High Performance Computing, Zhe Fan, Feng Qiu, Arie Kaufman, Suzanne Yoakum-Stover, To appear in Proceedings of ACM / IEEE Supercomputing Conference 2004, November 6-12, Pittsburgh PA.

[14] Concordia University Graphics Cluster (http://www.cs.concordia.ca/~mudur/GraphicsCluster)

[15] Stanford's Scalable and Parallel Interactive Rendering Engine (http://spire.stanford.edu/)

[16] Stony Brook's Visual Computing Cluster (http://www.cs.sunysb.edu/%7Evislab/projects/cluster/)

[17] "Using the GPU for point data-set processing." Technical Report, Ramgopal R, July 20, 2004.

[18] I.E. Sutherland, R.F. Sproull, R.A. Schumacker, "A Characterization of Ten Hidden Surface Algorithms," ACM Computing Surveys, Vol. 6, No. 1, March 1974, pp. 1–55.

[19] S. Molnar, M. Cox, D. Ellsworth, and H. Fuchs. A Sorting Classification of Parallel Rendering. IEEE Computer Graphics and Algorithms, pp. 23-32, July 1994.

[20] Alliance Display Wall-in-a-Box project at NCSA. (http://www.ncsa.uiuc.edu/TechFocus/Deployment/DBox/overview.html)

[21] R. Samanta, T. Funkhouser, K. Li, and J. Singh. "Hybrid Sort-First and Sort-Last Parallel Rendering with a Cluster of PCs." Eurographics/SIGGRAPH workshop on Graphics hardware, pp. 99-108, 2000.

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

[23] Rudrajit Samanta, Thomas Funkhouser, and Kai Li "Parallel Rendering with K-Way replication," IEEE 2001 Symposium on Parallel and Large-Data Visualization and Graphics, San Diego, California - October 2001.

[24] G. Humphreys, M. Houston, R. Ng, R. Frank, S. Ahern, P. D. Kirchner, and J. T. Klosowski. "Chromium: A Stream-Processing Framework for Interactive Rendering on Clusters." SIGGRAPH, pp. 693-702, 2002. (http://chromium.sourceforge.net/)

[25] Zhe Fan, Feng Qiu, Arie Kaufman, Suzanne Yoakum-Stover, "GPU Cluster for High Performance Computing," To appear in Proceedings of ACM / IEEE Supercomputing Conference 2004, November 6-12, Pittsburgh PA, USA.

[26] J. Montrym, D. Baum, D. Dignam, and C. Migdal. "InfiniteReality: A Real-Time Graphics System." Proceedings of SIGGRAPH 97, pp. 293-302, August 1997.

[27] H. Fuchs, J. Poulton, J. Eyles, T. Greer, H. Goldfeather, D.Ellsworth, S. Molnar, G. Turk, B. Tebbs, and L. Israel. "Pixel-Planes 5: A Heterogeneous Multiprocessor Graphics System using Processor-Enhanced Memories." Proceeding of SIGGRAPH 89, pp. 79-88, July 1989.

[28] S. Molnar, J. Eyles, and J. Poulton. "PixelFlow: High Speed Rendering Using Image Composition." Proceedings of SIG-GRAPH 92, pp. 231-240, August 1992.

[29] Image Layer Decomposition for Distributed Rendering on NOWs, Thu D. Nguyen and John Zahorjan, In Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS), May 2000.

[30] DRRRRaW: A Prototype Distributed 3D Real-Time Rendering Toolkit for Commodity Clusters. T. D. Nguyen, C. Peery, and J. Zahorjan. In Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS), May 2001.

[31] N. K. Govindaraju, A. Sud, S.-E. Yoon, and D. Manocha. Interactive visibility culling in complex environments using occlusion-switches. In Proceedings Symposium on Interactive 3D Graphics, pages 103–112, 2003.

[32] PHONG, BUI TUONG. 1975. "Illumination for Computer Generated Pictures," Communications of the ACM, vol. 18, pp. 311-317, 1975.

[33] G. Humphreys, M. Eldridge, I. Buck, G. Stoll, M. Everett, and P. Hanrahan. Wiregl: a scalable graphics system for clusters. In Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques(SIGGRAPH), pp. 129–140, 2001.

[34] Sushil Bhakar, Liang Luo, and S.P. Mudur "View Dependent Stochastic Sampling for Efficient Rendering of Point Sampled Surface", WSCG 2003.

[35] Szymon Rusinkiewicz and Marc Levoy "QSplat: A Multiresolution Point Rendering System for Large Meshes" Computer Graphics (SIGGRAPH 2000 Proceedings)

[36] Chapter 2 of "Parallel Computing: Theory and Practice" by M J. Quinn. 2nd Ed, 1994. pp. 25 – 50.
[37] J. D. Foley, A. van Dam, S. K. Feiner, and J. F. Hughes, Computer Graphics: Principles and Practice: Addison-Wesley, 1997, pp. 866--871.
[38] Ramgopal R, D Goswami, S P Mudur, "Functional Parallelism using Programmable GPUs". ACM Workshop on General Purpose Computing on Graphics Processors, Sponsored by and Co-located with ACM SIGGRAPH, August 7-8, 2004
[39] Ramgopal Rajagopalan, Dhrubajyoti Goswami, Sudhir Mudur, "Functionality Distribution for Parallel Rendering", IEEE IPDPS 2005, sponsored by ACM, and IEEE Computer Society.
[40] Naga K Govindaraju, Brandon Lloyd, Sungeui Yoon, Avneesh Sud, Dinesh Manocha, "Interactive Shadow Generation in Complex Environments", in ACM SIGGRAPH 2003.
[41] Michael Isard, Mark Shand, Alan Heirich, "Distributed rendering of interactive soft shadows.", in Proceedings of the Fourth Eurographics Workshop on Parallel Graphics and Visualization (EGPGV 2002), pages 71-76, September 2002
[42] A. Heirich and L. Moll, "Scalable Distributed Visualization Using Off-the-Shelf Components.", IEEE Parallel Visualization and Graphics Symposium (1999).
[43] L. Moll, A. Heirich, and M. Shand. "Sepia: Scalable 3D Compositing Using PCI Pamette." IEEE Symposium on Field Programmable Custom Computing Machines (1999).
[44] Florence Zara, François Faure, Jean-Marc Vincent, "Physical cloth simulation on a PC cluster.", Parallel Graphics and Visualisation, 2002
[45] P. Kipfer, Ph. Slusallek, "Transparent Distributed Processing for Rendering", Parallel Visualization and Graphics Symposium (PVG), San Francisco, 1999
[46] Trishul M. Chilimbi, Mark D. Hill, and James R. Larus, "Making Pointer-Based Data Structures Cache Conscious", IEEE Computer , December 2000.
[47] Levoy, M., Whitted, T., "The use of points as display primitives." Tech. rep., CS Departement, University of North Carolina at Chapel Hill, January 1985.
[48] Carsten D, Christian V, Marc S, "Sequential Point Trees", SIGGRAPH 2003.
[49] Michael Wand, Matthias Fischer, Ingmar Peter, Friedhelm Meyer auf der Heide, Wolfgang Straßer, "The randomized z-buffer algorithm: Interactive rendering of highly complex scenes", Siggraph 2001.
[50] Hubeli A., Gross M., "Multiresolution feature extraction from unstructured meshes.", in Proceedings of the conference on IEEE Visualization '01 (2001), IEEE Computer Society.
[51] Hoppe H., Derose T., DuChamp McDonald J., Stuetzle W., "Surface reconstruction from unorganized points." Computer Graphics 26, 2 (1992), 71.78. 3
[52] Gumhold S., Wang X., McLeod R.: Feature extraction from point clouds, Proc. 10th Int. Meshing Roundtable, pages 293-305, 2001.
[53] J.P. Grossman , "Point Sample Rendering" Master's Thesis, Dept. of Electrical Engineering and Computer Science, MIT.
[54] L. Westover, "Interactive Volume Rendering", Proc. Chapel Hill Workshop Volume Visualization, C. Upson, ed., pp. 9-16, May 1989.

[55] Botsch M., Kobbelt L.: "High quality point-based rendering on modern GPUs.", Proceedings of the 11th Pacific Conference of Computer Graphics and Applications, 2003.

[56] C. Dachsbacher, C. Vogelgsang, and M. Stamminger, "Sequential Point Trees", SIGGRAPH 2003.

[57] Dinesh Manocha, Ming C. Lin, Maria Bauer, and Michael Macedonia "The Edge: Intelligent Computing Using Graphics Processors (GPUs)", Modeling & Simulation, Volume 3, Number 1, January-March 2004

[58] Sally A. McKee, "Reflections on the memory wall", ACM Conference On Computing Frontiers , 2004, Ischia, Italy April 14 - 16, 2004.

[59] STREAM Benchmark (http://www.cs.virginia.edu/stream/ref.html)

[60] Pfister, H., Zwicker, M., van Baar, J., Gross, M., 2000. "Surfels: Surface elements as rendering primitives." SIGGRAPH 2000. pp. 335–342.

[61] Zwicker, M., Pfister, H., van Baar, J., Gross, M., "Surface splatting." SIGGRAPH 2001, pp. 371–378.

[62] L. Ren, H. Pfister, and M. Zwicker. "Object space EWA surface splatting: A hardware accelerated approach to high quality point rendering." In Eurographics 2002 Conference Proceedings, pages 461–470, 2002.

[63] M. Botsch, A. Wiratanaya L. Kobbelt, "Efficient high quality rendering of point sampled geometry", 13th Eurographics workshop on Rendering, pp 53-64, 2002.

[64] M.Botsch ,M.Spernat, L.Kobbelt, "Phong Splatting", pp 25-32, Symp.on Point-Based Graphics,2004.

[65] Erik Hubo, Philippe Bekaer , "A Data Distribution Strategy for Parallel Point-Based Rendering.", WSCG 2005.

[66] M M Akon, Ramgopal R, D Goswami, and H. F. Li, "DAMon: A Distributed Monitoring System for User Defined Application Parameters", PDPTA 2004.

[67] Carl Muller, "The Sort-First Rendering Architecture for High-Performance Graphics", Proceedings of the 1995 Symposium on Interactive 3D Graphics.

[68] Ian Buck, Kayvon Fatahalian, Pat Hanrahan, "GPUBench: Evaluating GPU Performance for Numerical and Scientific Applications", ACM Workshop on General Purpose Computing on Graphics Processors, Sponsored by and Co-located with ACM SIGGRAPH, August 7-8, 2004

[69] Performance Test V5.0 (http://www.passmark.com)

# Appendix A: ePublications Resulting from this Research

[1] Ramgopal R, D Goswami, S P Mudur, "Functional Parallelism using Programmable GPUs". ACM Workshop on General Purpose Computing on Graphics Processors, Sponsored by and Co-located with ACM SIGGRAPH, August 7-8, 2004

[2] Ramgopal Rajagopalan, Dhrubajyoti Goswami, Sudhir Mudur, "Functionality Distribution for Parallel Rendering", IEEE IPDPS 2005, sponsored by ACM, and IEEE Computer Society.

[3] M M Akon, Ramgopal R, D Goswami, and H. F. Li, "DAMon: A Distributed Monitoring System for User Defined Application Parameters", PDPTA 2004.

# Appendix B: Glossary of Graphics Terminologies

- ***AGP*** - The Accelerated Graphics Port is a graphics subsystem architecture developed by Intel. AGP is the combination of two features; first, a shared-memory design (i.e., some operations related to graphics will be performed inside the system's main memory), second, a faster bus than the current PCI bus.
- ***Frame Buffer*** - An area of video memory used to store the pixel data for a single screen image, or frame.
- ***Fragment Shader/Program*** - A low level program written to program the Fragment shader unit of the graphics hardware to achieve user customized operations on the input fragments.
- ***GPU*** - Graphics Processing Unit, a synonym for the graphics card inside a PC.
- ***Graphics Pipeline:*** - In almost all computer graphics applications, geometry data with associated information such as color and texture, is processed through a number of stages until it is finally transformed into a picture. These processing stages are arranged in the form of a pipeline.
- ***Open GL*** - A set of specifications for a cross-platform 3D graphics API developed initially by Silicon Graphics Inc. There are several implementations of Open GL, provided by different vendors.
- ***Rasterization*** - The process of transforming a 3D image into a set of colored pixels, i.e. the process of giving a color to each pixel, depending on light sources, the position of the object that the pixel represents textures, etc.

- ***Rendering*** - A term which is often used as a synonym for rasterization, but which can also refer to the whole process of creating a 3D image.
- ***Splatting***- Splatting is the process of associating a surface with a point and rendering the surface with appropriate screen space extent when the point density in a region of screen falls below the pixel density.
- ***z-buffer*** - The z-buffer is a portion of memory used to store the coordinate on the z-axis of the closest opaque point for each value of x and y (i.e., if the resolution is 640x480, the z-buffer is a 640x480 array). All other points with the same coordinate on the x and y axes, but with higher coordinates on the z axis will be invisible to the player and, therefore, will not be drawn. The z-buffer is used for hidden surface removal.
- ***Vertex Shader/Program*** - A low level program written to program the vertex shader unit of the graphics hardware to achieve user customized operations on the input vertices.

# Appendix C: Shaders Used for Performance Modeling

```
!!ARBvp1.0
#Declarations
ATTRIB in = vertex.position;
ATTRIB nrm = vertex.normal;
ATTRIB tex = vertex.texcoord[0];

PARAM mvp[4] = { state.matrix.mvp };
PARAM mv[4] = { state.matrix.modelview };

PARAM light = { 0.57735, 0.57735, 0.57735, 0.0};
PARAM color = {0.2, 0.3, 1.0, 1.0};
PARAM const = {0.2, 0.2, 0.2, 32.0};

# size_factor allows the user to modify splat size
# size_fac = size_factor * 2.0 * n * h_vp / (t-b)
PARAM c = program.env[0];

OUTPUT out = result.position;
OUTPUT oPointsize   = result.pointsize;
OUTPUT oColor       = result.color;
OUTPUT oTex = result.texcoord[0];

TEMP spec, dif, tmp, tmp1;
TEMP vDir;  # For storing view direction.
TEMP nNrm;  # For storing the normal.

#Code
#transform the positions
DP4 out.x, in, mvp[0];
DP4 out.y, in, mvp[1];
DP4 out.z, in, mvp[2];
DP4 out.w, in, mvp[3];

#transform the position to eye-space
DP4 tmp.x, in, mv[0];
DP4 tmp.y, in, mv[1];
DP4 tmp.z, in, mv[2];
DP4 tmp.w, in, mv[3];

#view direction
MOV vDir, -tmp;

#normalize the view direction
DP3 vDir.w, vDir, vDir;
RSQ vDir.w, vDir.w;
MUL vDir.xyz, vDir, vDir.w;
MOV vDir.w, c.w;              # We pass 0 in c.w

#transfrom the normal to eye-space
DP3 nNrm.x, mv[0], nrm;
DP3 nNrm.y, mv[1], nrm;
DP3 nNrm.z, mv[2], nrm;
```

```
#Normalize the normal
DP3 nNrm.w, nNrm, nNrm;
RSQ nNrm.w, nNrm.w;
MUL nNrm.xyz, nNrm, nNrm.w;
MOV nNrm.w, c.w;                # We pass 0 in c.w

#compute half angle vector
ADD spec.xyz, vDir, light;
DP3 spec.w, spec, spec;
RSQ spec.w, spec.w;
MUL spec.xyz, spec, spec.w;

#compute specular intensisty
DP3 spec.w, spec, nNrm;    # H dot N
LG2 spec.w, spec.w;             # log (H dot N)
MUL spec.w, spec.w, const.w; # specularExponent * log(H dot N)
EX2 spec.w, spec.w; # 2^(specExp * log(H dot N)) = (H dot N) ^ specExp

#compute diffuse illum
DP3 dif, nNrm, light;       # N dot L
ADD dif.xyz, dif, const;

#sum
MAD dif.xyz, color, dif, spec.w;
MOV dif.w, color.w;

MOV oColor, dif;

#output the texcoords
MOV oTex, tex;

# Compute point size (integer!), Eq. 5 from Phong Splatting paper.
RCP  tmp1.z, tmp.z;
MUL  tmp1.x, tex.x, tmp1.z;
ABS tmp1, tmp1;          # This step is required to get positive values.
MAD  tmp1.y, c.x, tmp1.x, c.y;
FLR  oPointsize, tmp1.y;

END
```

**A. Vertex Shader Program**

```
!!ARBfp1.0
#simple phong lighting shader
#Declarations
ATTRIB col = fragment.color;
ATTRIB tex = fragment.texcoord[0];

OUTPUT out = result.color;

#code
MOV out, col;


END
```

**B.Fragment Shader Program**