University of Zurich[UZH]

Department of Informatics

Level-of-Detail and Parallel Solutions in Computer Graphics

A dissertation submitted to the Faculty of Economics, Business Administration and Information Technology of the University of Zürich

for the degree of
Doctor of Science (Ph.D.)

by
Prashant Goswami
from India

Supervisor: Prof. Dr. Renato Pajarola
Co-referee: Prof. Dr. Enrico Gobbetti

Zürich, October 5, 2011

ABSTRACT

Computer graphics has gained tremendous importance in modern industry, especially in last two decades. The applications of this field are manifold; whereas areas in entertainment like movies, games, art etc are obviously visible to all, of late several other important domains like medical and engineering, rely on graphics for myriad of their jobs. Given the advanced scanning devices to procure gigabytes of data, the challenge has been to be render and visualize high quality data in real time. The task of providing user with visually pleasing graphics in real time is not just limited to visualization but extends to other fields like fluid simulation. To reduce the computational burden while still maintaining the visual quality, level-of-detail (LOD) based algorithms have been a popular choice. The first contribution of this thesis is to present LOD based algorithms in three different fields : point-based rendering, terrain rendering and particle-based fluid simulation.

The design of rendering and visualization techniques has seen close proximity to the developing hardware. Several works have been focussed solely to creation of GPU or multi-machine cluster-friendly algorithms. Parallelization of task can yield a significant speed-up thereby making such techniques highly attractive for practical use. Hence, our next focus would be to present our novel parallelization schemes in the above mentioned fields. In the point-based and terrain domain, we combine our LOD techniques with parallelization together with out-of-core rendering.

ACKNOWLEDGEMENTS

I would first like to thank my Phd supervisor, Prof. Dr. Renato Pajarola for giving me the opportunity to pursue my PhD studies at Visualization and Multimedia Lab, University of Zürich. I am grateful for his support and would express my gratitude to be able to benefit from his knowledge and experience. I would also like to thank Prof. Dr. Enrico Gobbetti for accepting to be co-referee of my dissertation and being in the committee. Many thanks to Prof. Dr. Prem Kalra, my master thesis advisor for sowing the seeds of interest in me in Computer Graphics.

I would also like to thank Maxim Makhinya and all other group members for their unconditional support and help throughout my Phd studies. Last but not the least, I am greatly thankful to my mother and sister for all time support in whatever I did.

CONTENTS

LIST OF FIGURES

LIST OF ALGORITHMS

LIST OF TABLES

Part I

Out-of-core, LOD-based Parallel Point and Terrain Rendering

CHAPTER 1

INTRODUCTION

Computer graphics invariably deals with capturing and imitating natural phenomena directly or indirectly. There is an everlasting drive to reach perfect realism. The challenge arises from the limited computational resources and demand for high performing, near real-time solutions. The two major modes of rendering images in graphics can be broadly grouped as ray-tracing and rasterization. In ray-tracing, for each pixel of the screen a ray is casted from the eye and based on geometry and light sources, the color value of that pixel is computed. In rasterization, on the other hand, geometry is made available as primitives to a renderer together with camera and screen information. Rasterization has a rendering time that is typically linear in the number of triangles that are drawn, because each polygon must be processed. With the advent of modern graphics hardware, this time is further reduced and modern GPUs can draw nearly 600M polygons per second. A number of applications therefore, employ rasterization based rendering. For this, the dataset is expressed in basic primitives like triangles, points, volume scalar values etc. A significant amount of research in computer graphics thus deals with efficient and realistic visualization of these datasets in their various forms.

1.1 Large datasets

With the super linear increase in the processing power of graphics hardware, the acquisition precision of modern scanners also increased several times. Such model

complexity continues to outspace the explosive growth of CPU and GPU processing power. Brute force rendering cannot achieve interactive frame rates. The challenge, therefore, shifted from achieving barely interactive(3-5 fps) visualization for small to moderate size data sets to much higher frame rates even for larger datasets. At such a stage, the earlier laid out solutions to inspect data that fits into the main memory do not help anymore.

According to Moore's Law, CPU performance has increased by 60% per year for nearly two decades. However, main memory and disk access time only increased by 7-10% per year. Further, the bandwidth limitations while transferring data on and off the main and GPU memory continue to exist. Hence real time rendering of large models remains to be an active topic of research.

1.2 Solutions

Due to their limitless size and current hardware trends, interactive massive model will always require specialized rendering algorithms which are output-sensitive: performance is a function of the model that is visible to the user. All the approaches dealing with such massive models are invariably out-of-core. Further, any part of the geometry that is invisible to the user can be discarded by techniques like backface and occlusion culling. This can reduce the rendering burden to a sizable extent depending on the topology of scene and the viewing parameters. Improvements can also be obtained by proper memory management and caching methods. The two most important techniques, however, on which this thesis concentrates are as *level-of-detail* and *parallel* based solutions.

(a) (b)

Figure 1.1: *Point-based renderers, (a) Google Earth (courtesy : Google) (b) St. Matthews model (courtesy : Stanford University)*

1.2.1 Level-of-Detail (LOD)

Nature operates in seemingly continuous domain by humane perceptions. In order to be able to observe and imitate natural phenomena, humans need to discretize them. Further, the amount of captured information visible to the human eye varies with factors like the distance of viewer from the model. In such cases, to show the entire discretized data often does not add to the quality but degrades user-interactivity with the application. Thus, to be able to visualize colossal data models on the modern machines, level-of-detail based methods have been a hot topic of research. In computer graphics, level-of-detail involves decreasing the complexity of a 3D object representation as it moves away from the viewer or according to other metrics such as object importance or topography, speed etc. LOD techniques increase the efficiency of rendering by decreasing the workload on graphics pipeline stages, usually vertex transformations. The reduced visual quality of the model is often unnoticed because of the small effect on object appearance or its uniformity over the simplified region when distant.

LOD techniques are not limited to vertex or geometry processing but can be generalized even to image and texture mipmapping. The earliest foundations on LOD were laid out in [Clark, 1967] when in fact, machines and programming languages were pretty different. In current real life applications, LOD techniques are commonly used everywhere from mesh-based rendering to particle and cloth simulation. A common example can be illustrated with Figure 1.2 where Stanford bunny represented using mesh is repeatedly simplified. Each simplified model carries fewer triangles than its previous counterpart. Whereas for a closer camera position, the most detailed (leftmost) bunny could be displayed, in many situations of a far-off viewing showing the least detailed (rightmost) model might suffice. The same idea is reflected in Figure 1.3 where with distance from the viewer, level-of-detail of the tree changes. For far off distances, an object with several times less details can create an equally pleasing effect.

Out-of-core applications benefit the most from LOD solutions. This is because the original data cannot fit into the memory and unless one uses some LOD simplification, there is no other way to visualize it on limited hardware. Even with the most sophisticated hardware, object complexity will always catch up. In such cases, LOD is constructed offline and stored on the hard disk. Later at runtime, a selected portion of this stored simplified model is chosen and fetched on to the main memory. Granularity level of LOD selection varies on a number of factors which include CPU or GPU bound process, level of accuracy desired etc. Better granularity provides better fidelity wherein LOD is specified exactly but comes with the higher cost. With CPU-based solutions this used to be the general scenario. However, as an efficient alternative, this selection can be coarsely performed over a larger chunk of primitives. This might lead to slight overestima-

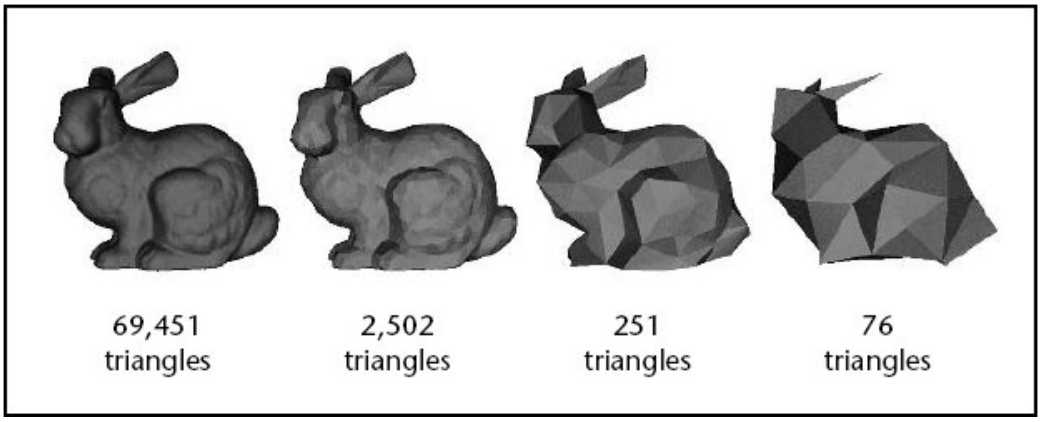


Figure 1.2: *Level-of-detail mesh representation of Stanford bunny, courtesy: polygon-reducer.pc-guru.cz*

tion or approximation on primitives which is worth the speed-up obtained when considering the batch rendering capabilities modern graphics hardware.

The simplicity of LOD creation procedures depends on the complexity of problem. Whereas in mesh, one has to take care of connectivity, topology preservation etc., with points these issues do not come into play. Simplification itself can be done using local or global simplification techniques. For choice of LOD at runtime, an error metric is often chose. Similar to preprocessing, here too granularity plays an important role. This error metric could be defined on per-primitive or per-batch. A very common practice is to include the screen-space error in this error metric. Another problem arises due to abrupt or non-uniform shift from one LOD to another. This is usually alleviated using operations like morphing which make a smooth transition by blending between two LODs to make switching seamless. However, one also has to take into account the computational cost of blending which might not hurt in many cases.

In some other fields like particle simulation, although data might fit into the main memory, it is still beneficial to employ LOD solutions. This is to obtain better interactivity or reduction in overall computational time. Similarly in computer games, where a higher frame rate could be obtained via using LOD approaches. In short, there is hardly any area in computer graphics, where one cannot apply or does not find motivation to apply LOD techniques. A major contribution of this thesis is in the area of LOD-based algorithms.

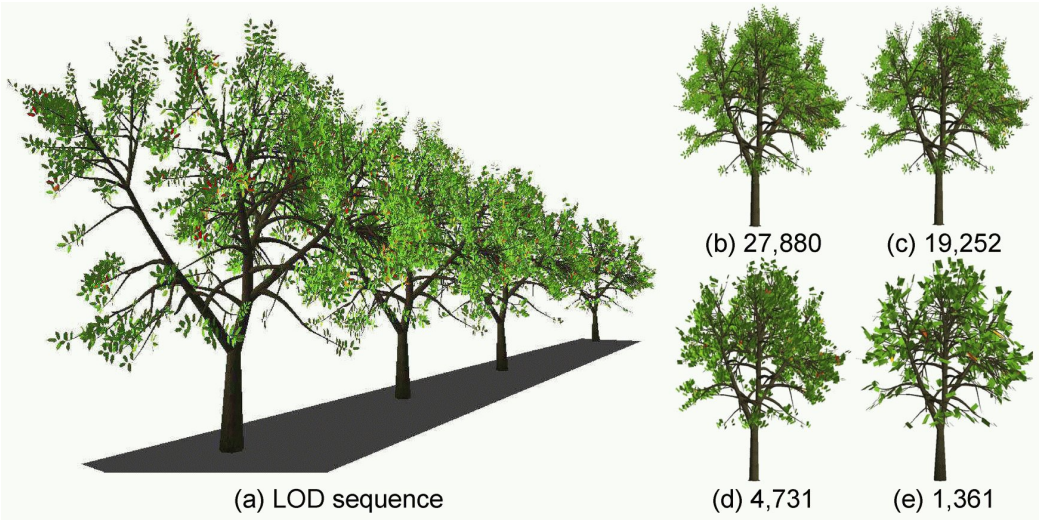


Figure 1.3: *Level-of-detail mesh representation of tree, courtesy: INRIA, France*

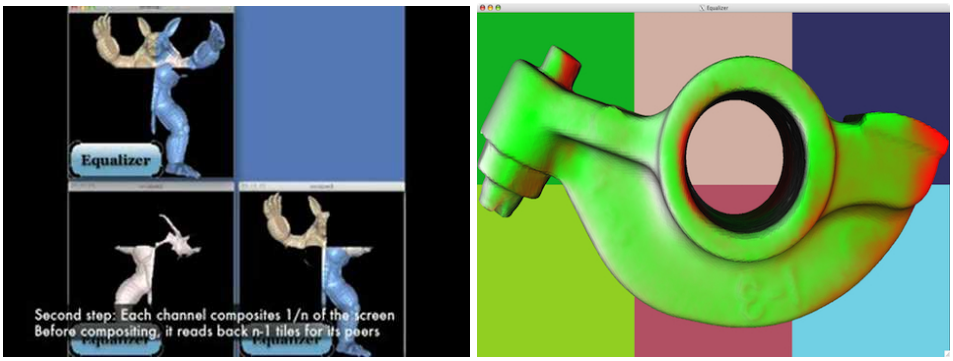
1.2.2 Parallelism

Parallel rendering is yet another way to improve the performance of computer graphics software. It requires massive (for ex. cluster) or specialized (for ex. gpu) computational resources to divide the work to be done in parallel. A simple example in this case is that of a raytracer where the pixels can be easily divided for casting rays in parallel. While parallel processing on a low number of processors is relatively straightforward, the challenge comes when confronting an implementation on a large system. This is because one has to device means to synchronize between various machines or processors which in itself might not be a trivial task. Further, it introduces an overhead of its own.

PC graphics card are getting faster at an exponential rate. With the reduction in costs, graphics card offers a viable alternative for parallel rendering in comparison to large clusters in several domains. While parallelizing the task on graphics card, it is distributed among its many threads as in the case of per-vertex operations with shaders. On the other hand, one achieves division of task on multi-machine clusters using one of the following two kinds approaches:

1. Sort-last: This refers to division of task in database or geometry domain, for ex. distributing polygons of a model among various machines for rendering. Often each machine has entire scene in this case with it, out of which it renders a selected portion only. Finally the individually rendered portions are combined on one destination machine by taking color and depth values for each pixel into account.

2. Sort-first: This refers to the division of task in screen domain itself, for ex. each of the machines selects a part of the frustum and renders data visible in it. Thereafter, similar to sort-last, one destination machine combines all the data. Since each machine renders on mutually exclusive section of screen, one does not need depth-based comparison and final image can be simply obtained by collating sub-images from individual computers.



(a) Sort-last (b) Sort-first

Figure 1.4: *Parallel task division using sort-last and sort-first approaches, courtesy: www.equalizergraphics.com.*

These two division modes are demonstrated in Figure 1.4. In addition, one could use both sort-last and sort-first in combination. In order to achieve optimal task distribution, load balancing is often employed. By doing so, load is shifted from a busy machine to a relatively unoccupied one.

Higher throughput can often be obtained by using more resources to accomplish a given task. A variety of algorithms can be parallelized efficiently exhibiting the obvious benefit of putting multiple instances of hardware to work. Especially in the last one decade, researchers have come up with several parallelizing approaches. The second major contribution of this thesis is in the area of parallel computing.

1.3 Application

Interactive visualization of out-of-core models finds applications in many fields including terrain and point-based rendering. For example, Google in 2006 (Figure 1.1(a)) had 70 terabytes of compressed satellite imagery stored in Bigtable [Fay et al., 2006] and accessed by Google Earth and Google Maps. In a typical real life application scenario like games, multimedia etc., terrain itself could be just one of

the components of rendering and even relinquished to the background. Additionally, the texture data itself is often several times the size of digital elevation model (DEM) data.

Triangles and points are often interchangeably used to represent and process models. A similar challenge therefore, exists in context of point-based rendering. For example, [Rusinkiewicz and Levoy, 2000] (Figure 1.1(b)) in 2000 introduced the concept of dealing with giga size models at interactive rates. The largest model, St. Matthews consisted of about 122 million vertices which cannot fit into the main memory of an ordinary working machine of that time. Similar to terrain, even point models might constitute only one component of rendering scene.

The aforementioned fields bear similarity in the problem in that both deal with out-of-core rendering of colossal models at interactive frame-rates. To cope with the expectations and available resources, several LOD based solutions have proposed to tackle terrain as well as point based rendering. Further, with the growth of efficient hardware the nature of algorithms proposed has also seen modifications. Also recently, the idea of using multiple hardware resources in parallel to accelerate the application has been adopted.

1.4 Dissertation Overview

In the last sections, we introduced the main challenges and commonly employed solutions in the field of computer graphics, specifically for out-of-core rendering. This section briefly introduces the contributions made in the field of out-of-core terrain and point based rendering.

1. Terrain rendering: The proposed work deals with efficient compression and management of large texture units built on top of existing terrain renderer. In order to maintain continuous display of LOD and hide out-of-core latency, asynchronous fetching is suggested. Then as a next step, to be able to visualize very large terrain models with reasonably low error, parallelizing algorithms have been developed for multi-machine clusters and compared. This includes task division both in screen and database domain and evaluating related pros and cons.

2. Point based rendering: As a second contribution, an efficient preprocessing and rendering has been proposed for large point datasets. This is made possible by a novel data structure, *multi-way kd-trees* which makes rendering more GPU oriented. In order to construct equally sized multi-way kd-tree nodes, several simplification algorithms have been proposed. Further to leverage equally sized multi-way kd-tree nodes, a budget based rendering approach is developed. We introduce sophisticated operations to bridge the

quality gap between triangles and points. Finally, point rendering is parallelized on multi-machine clusters and compared with triangles as OpenGL primitives in comparison to points.

The remainder of the thesis is organized as follows. This thesis is divided into parts. Whereas in Part I, out-of-core, LOD-based parallel point and terrain rendering contributions are discussed, Part II draws details on LOD and parallel solutions in context of particle simulation. Chapter 2 begins with briefly introducing the notion of point-based rendering followed by a detailed presentation of our contribution in level-of-detail (LOD) based point preprocessing and rendering. In Chapter 3, we extend our work by introducing advanced features and parallel rendering in context of point-based rendering. Chapter 4 and 5 deal with LOD-based and parallel terrain rendering. Finally, Chapter 7 and 8 in Part II detail parallel and LOD-based contributions respectively in particle simulation. Before this, Chapter 6 briefly outlines the basics of Smoothed Particle Hydrodynamics (SPH).

CHAPTER 2

POINT BASED RENDERING (PBR)

2.1 Point models

Point models are usually acquired by range scanning or image-based construction methods. Because real-world objects are captured with as much detail as possible, the resulting models are usually highly complex and have to be simplified in order to be of suitable complexity for visualization. Some examples for the application of laser scanning are in the field of cultural heritage to document places and buildings of historical interest, in the field of geodesy to measure the earth's topography or in the production industry to document the status of a large industrial plant in order to support change management. In all these areas the physical size of the scanned objects is ever increasing. Another reason for the tremendous growth of data produced is that the possible number of samples per scan is surpassing 1 billion for the latest generation of laser scanners. The point clouds resulting from such scans are often used to inspect objects. The biggest advantage of such point-based representation of models over polygonal meshes is that they can easily be restructured without the need to take care of manifold or connectivity conditions. Hence applications requiring frequent geometry resampling will benefit most from point-based methods. One prominent example is PointShop3D [Zwicker et al., 2002] which is a Photoshop-like tool for manipulating point-sampled models.

2.2 PBR Basic Principles

In this section, we will briefly touch upon the basic PBR principles as laid out in the survey [Kobbelt and Botsch, 2004].

2.2.1 Point Primitives

A point-based geometry representation can be considered a sampling of a continuous surface, resulting in 3D positions $\mathbf{p_i}$, optionally with associated normal vector $\mathbf{n_i}$ or other properties like color. In the absence of normal vectors, they can be estimated by considering a local neighborhood of each point. A commonly used approach is to consider *k*-nearest neighbors for this purpose. This method is more robust than defining all samples within an ϵ-ball around a query point to be its neighbors. This is because the neighborhood estimate is reliable even in case of irregularly sampled models. In contrast to triangle meshes where neighborhood information is represented explicitly, the local neighborhoods are usually computed either dynamically or as a pre-processing step in case of point-sampled geometries. Let $\mathbf{p_0}$ be a sample point and $\{\mathbf{p_1}, \mathbf{p_2}, ..., \mathbf{p_k}\}$ its *k*-nearest neighbors. The covariance matrix

$$C = \sum_{i=0}^{k} (\mathbf{p_i} - \tilde{\mathbf{p}}_\mathbf{i})(\mathbf{p_i} - \tilde{\mathbf{p}}_\mathbf{i})^T \in \mathbb{R}^{3\times 3} \tag{2.1}$$

where $\tilde{\mathbf{p}}_\mathbf{i} = \sum_{i=0}^{k} \mathbf{p_i}/(\mathbf{k}+\mathbf{1})$ is symmetric and positive semi-definite. The normal direction is estimated by taking the eigenvector corresponding to the smallest eigenvalue.

2.2.2 Surface Definition

Since we want to generate continuous images by rendering a discrete set of surface samples, methods for closing the holes and gaps in-between the samples have to be found. This cane be done by image-space reconstruction technique [Pfister et al., 2000] or by object-space resampling. Surface splats have been proposed by [Zwicker et al., 2001] for rendering. In order to give a continuous representation and generate surface, points $\mathbf{p_i}$ is additionally associated with a radius r_i and normal vector $\mathbf{n_i}$. This way we bridge the gap between neighboring point samples. Within points, there are several ways of generating surface with the use of different primitives. The simplest one is OpenGL points in which each point is rendered as a quadrilateral. Another way which also provides local optimal adaptation to the curvature of the underlying surface is elliptical splats. One could also use anti-aliased circular disks where the points get rounded with respect to

their radius. Simple surface elements (Surfels) as point rendering primitives were introduced in [Pfister et al., 2000]. The main idea there was to describe objects in a view-dependent, object centered rather than image-centered fashion. A high quality anisotropic anti-aliasing method is proposed in [Zwicker et al., 2001] resembling the EWA texture filtering of [Heckbert, 1989], in which each splat is assigned a radially symmetric Gaussian filter kernel. [Alexa et al., 2001] present a smooth manifold surface from a set of points using moving least squares (MLS) as illustrated in Figure 2.1.

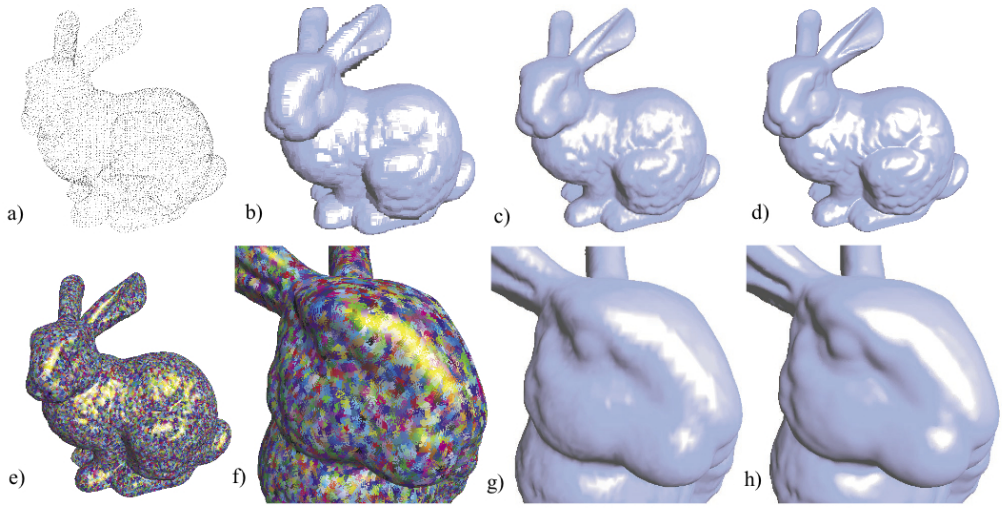


Figure 2.1: *Stanford Bunny: (a) Points (b) Points splatted to satisfy screen space resolution (c),(g) Piecewise Linear Mesh (d),(h) Non-conforming polynomial patches, courtesy: University of California, Davis.*

The basic idea in all the above representations is to obtain an implicit continuous surface. This can be also be explained as follows: starting with a triangular mesh, one can obtain point representation of a model by generating a point for every vertex and assigning it a radius large enough such that it touches all its neighbors. The normal vector corresponding to the point can be estimated either from the triangular mesh by considering adjacent vertices or using k-nearest neighbors in the point representation. Now each point can be rendered as a primitive (for example, circular disk or quadrilateral) such that the mesh representation and point representation of the model are close enough. In essence, each point intrinsically carries the local property of its area in the mesh representation.

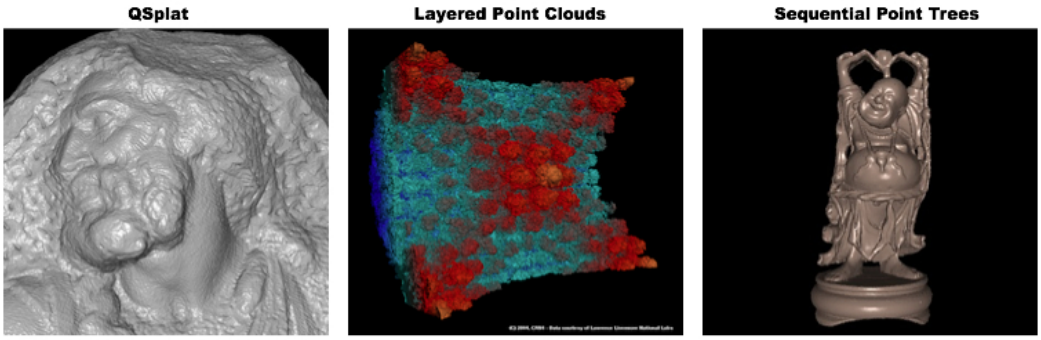


Figure 2.2: *Point-based renderers, (a) QSplat, (b) Layered Point Clouds and (c) Sequential Point Trees.*

2.2.3 Rendering

The final stage for interactive geometry processing applications is the efficient rendering of point-sampled geometries. Depending on the suitableness, one could choose any of the above representatives for rendering. To save per point culling, data structures like octree, kd-tree are employed to hierarchically group the points spatially as nodes. Thereafter, a simple test is made on visibility of these nodes and the elements within all visible nodes are sent for rendering. The culling test is usually performed on CPU and the selected point data is passed over for rendering. The earlier works like [Rusinkiewicz and Levoy, 2000], [Rusinkiewicz and Levoy, 2001] are CPU-dependent, whereas with the advent of modern graphics hardware coarse LOD selection was still done on the CPU and rendering moved to the GPU [Gobbetti and Marton, 2004a], [Gobbetti and Marton, 2004b]. [Dachsbacher et al., 2003] developed a completely GPU-based method for models that can fit into GPU memory which could perform LOD selection on GPU itself but not culling. Figure 2.2 shows snapshots using three different popular rendering approaches.

In order to improve the rendering quality further, smooth shading of points is carried out. For this, often 2 + 1 rendering passes need to be carried out: two passes over the geometry and one image processing pass. To achieve smooth interpolation and resolve correct visibility of overlapping point splats, a separate visibility-splatting rendering pass is needed. In a second point-blending rendering pass the smooth interpolation between visible overlapping points is performed. [Zhang and Pajarola, 2007] introduced *deferred blending* which was based on a separation of the point data into non self-overlapping minimal independent groups, a multi-target rendering pass and an image compositing post-processing stage.

2.3 LOD Principles

The idea of using points as primitives instead of triangles was introduced as early as [Levoy and Whitted, 1985] and later reintroduced by [Rusinkiewicz and Levoy, 2000] to visualize large polygonal meshes. With the advent of graphics hardware, more sophisticated approaches were introduced that could also utilize the on-board memory and computational power of GPUs [Dachsbacher et al., 2003] and [Gobbetti and Marton, 2004a]. Since modern scanning devices can generate massive datasets with millions of points, the field of out-of-core high quality efficient visualization has received continuous attention. Massive models are tackled with LOD and out-of-core and sometimes parallel techniques.

QSplat [Rusinkiewicz and Levoy, 2000] has for long been the reference for massive point rendering. It is based on bounding spheres hierarchy maintained out-of-core, that is traversed at runtime to generate points. This algorithm is CPU bound, because all computations are made per point, and CPU-GPU communication requires a direct rendering interface, thus the graphics board is never exploited at its maximum performance. Since modern GPUs can sustain very high primitive rendering rates, a slow CPU based adaptive data selection process can easily lead to starvation of the graphics pipeline. This consideration has lead to emergence of a number of coarse-grained techniques which perform high-level selection on CPU and hand over the batch process over to graphics card.

Sequential Point Trees [Dachsbacher et al., 2003] introduced a sequential adaptive high performance GPU oriented structure for point models that can fit on the graphics board. XSplat [Pajarola et al., 2005] and Instant points [Wimmer and Scheiblauer, 2006] extend this approach for out-of-core rendering. XSplat is limited in LOD adaptivity due to its sequential block building constraints, while Instant points mostly focuses on rapid but moderate quality rendering of raw point clouds. Layered point clouds (LPC) [Gobbetti and Marton, 2004a] and Wand et al.'s out-of-core renderer [Wand et al., 2008] are prominent examples of high performance GPU rendering systems based on a hierarchical decomposition into large sized blocks maintained out-of-core.

LPC is based on adaptive BSP subdivision and subsamples the point distribution at each level. In order to refine an LOD, it adds points from the next level at runtime. This composition model and pure subsampling approach limits the applicability to uniformly sampled models and produces moderate quality simplification at coarse LODs. These limitations are partially removed in [Bettio et al., 2009] by making all BSP nodes self-contained and using an iterative edge collapse simplification to produce node representations, similar to [Pauly et al., 2002]. The quality obtained however, can be improved further by better simplification methods. Similarly, [Wand et al., 2008] is based on out-of-core octree of grids, and deals primarily with grid based hierarchy generation and editing of the

point cloud. The method's limitation is in the quality of resolutions created by the grid.

LOD based simplification is a natural choice for all massive point rendering pipe-lines. While some methods [Gobbetti and Marton, 2004a], [Wand et al., 2008], [Wimmer and Scheiblauer, 2006] are inherently forced to use fast but low-quality methods, others can employ higher quality simplification methods [Rusinkiewicz and Levoy, 2000], [Pauly et al., 2002], [Bettio et al., 2009]. In this context, we focus here on an approach a fast high quality approach. The main features and contributions of our approach can be summarized as following:

1. *Multi-way kd-trees*, a data structure that is more suited to gpu-based rendering.

2. A fast yet high quality preprocessing framework which includes hierarchical multiresolution simplification method combined with delayed sorting.

3. A budget-based rendering approach based on asynchronous fetching targeted to maintain a constant frame rate and high quality at the same time.

2.4 Multi-way kd-trees

Most LOD point rendering approaches use either a kd-tree or a regular octree for basic hierarchical data organization. The main benefits of octrees are that they are simple and subdivide the 3D data uniformly in space. However, octrees have a number of drawbacks, especially when considering point simplification and GPU rendering constraints. Octrees are inflexible due to their fixed fanout factor and therefore, there is no direct control over the number of internal nodes. Octrees can thus be highly imbalanced and therefore, deeper than necessary. Due to strict spatial division policy, each node might obtain suboptimal or above optimal number of nodes leading to further subdivision. One therefore, has no direct control over number of nodes and the points per node i.e the target *Vertex Buffer Object* (VBO) size. Since most caching schemes make use of similar sized cache blocks, due to irregular node sizes this might easily lead to wastage of cache memory.

Although kd-trees can be constructed so as to be balanced and symmetric, one might encounter similar limitations there. Using a fixed split strategy at the middle of the bounding box produces wildly different VBO sizes, which can vary by as much as 50%. This can be alleviated by moving the split plane to always produce equal sized children. However, with the strict fanout of 2, one does not have direct control over the total and internal number of nodes in the tree. The number of internal nodes in the tree determine the LOD quality and also the rendering expense as the more the count of internal nodes, the more the VBO switches.

The challenge, thus, is to design a data structure that can not only handle the point distribution in the nodes but also allows to have better control over the total number of nodes and hence VBOs. In this thesis, we propose to reach this goal by exploiting the properties of *multi-way kd-tree*, i.e., kd-trees with a fanout factor of N at each level instead of 2, see also Figure 2.3. At each level a node is divided along its longest axis into N children containing equal number of points each. If the original model has n points, s is the target VBO size and m the number of leaf nodes in the tree then $m = n/s$. For a given N, $l = \lceil logm/logN \rceil$ is the maximum level or depth of the multi-way kd-tree.

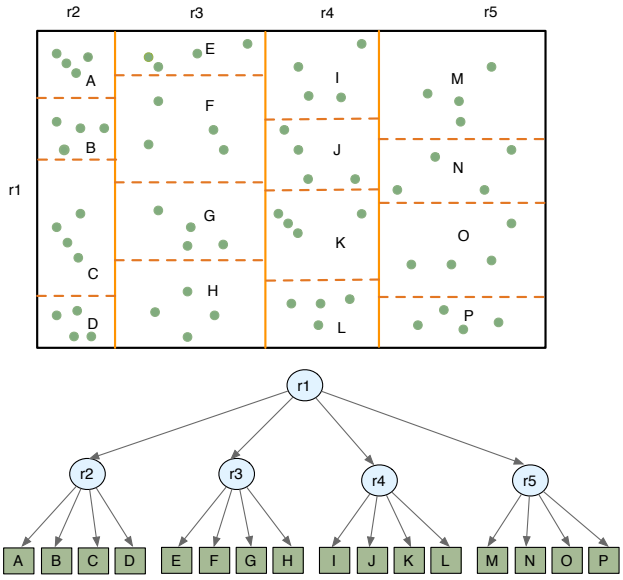


Figure 2.3: *Multi-way kd-tree example for N = 4. Each of the leaf regions contains almost equal number of points.*

With the proper choice of paramters s and N, one can construct an efficient multi-way kd-tree such that each leaf node contains approximately s points. The value of N decides the trade-off between LOD adaptivity and number of VBOs. For larger N, tree height will be lower with fewer internal nodes thereby reducing VBO switches at the expense of number of LODs. On the other hand, smaller values of N will create trees with more height and internal nodes thereby increasing the cost due of VBO switching. For a proper choice of N, a good trade-off between LOD quality and VBO expense can be obtained. A multi-way kd-tree thus has the advantages that it is flexible in its fanout factor, giving more control over LOD adaptivity, it is symmetric and balanced and thus can be maintained easily in an array and that it has an almost uniform point distribution close a desired target VBO size.

2.5 LOD construction

The next challenge after obtaining all leaf nodes close to target VBO size s is to construct internal nodes. The simplification procedure should generate approximately s points per internal node too. By doing so, we ensure that the nodes are uniformly sized and hence good for caching and contain high quality LOD. Our procedure to construct a multi-way kd-tree is based on a top-down recursive subdivision pattern, followed by a bottom-up simplification, see also Algorithm 1. Each node is subdivided into N children along the longest axis if the number of points in it exceeds the threshold s. In a fully balanced tree, $\frac{n}{s}$ is a power of N and thus N can be chosen such that $log\frac{n}{s}/logN$ is as close as possible to an integral value. However, in order to avoid large values of N, one can choose a smaller value that achieves closest possible construction.

Algorithm 1 MWKdTree(node)

```
1:  if (number of points in node ≤ s) then
2:      return

3:  determine the longest axis of node
4:  if (an ancestor sorted along same axis exists) then
5:      get sorted data from ancestor
6:  else
7:      sort data of node

8:  split the node data into N child nodes Nc_i
9:  for all (children i ≤ N) do
10:     MWKdTree(Nc_i)
11:     get reduced LOD points from child node by
12:     calling Representatives(Nc_i)
```

In the tree construction procedure, we exploit delaying the sorting wherein we first check if an ancestor was sorted along the desired axis. This can be implemented by retrieving sorted lists from ancestors as indicated in Figure 2.4. If an ancestor in the multi-way kd-tree has already sorted the points along the desired axis, the order is carried over to the new node, otherwise the points are sorted locally. For example, on level 4 points sorted along the x-axis and can be retrieved directly from level 1. Assuming that we use optimal sorting, in the worst case there would be just $O(n \log n) + O(n \log(n/N)) + O(n \log(n/N^2))$ sorting cost in delayed sorting compared to $3 \cdot O(n \log n)$ when the entire data is sorted along all 3 axes. However, if the size of desired sorted segment is much smaller than that of the sorted ancestor, the binary search within the latter might induce a

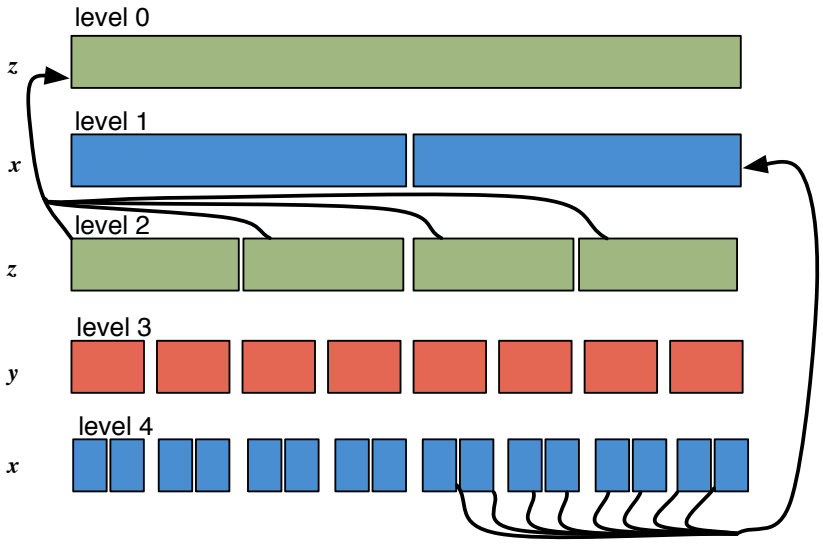


Figure 2.4: *Delayed sorting during node construction.*

higher overhead. In such cases, we resort to occasionally perform a new local sort on that segment in the current node.

During bottom-up simplification, fixed-point internal nodes are constructed from their children representation using a simplification procedure. This can be achieved by using methods like iterative edge collapse [Pauly et al., 2002] or with a modification of [Wand et al., 2008] wherein a K^3 grid is established in every node, where K is an integer. Thereafter, a representative point is obtained from every grid cell by averaging up all the points present in it. The drawback with the latter approach, however, is that it does not take the spatial point distribution into account. All the points falling in the same grid cell are used to generate one LOD representative including those which do not belong to the same surface component thereby generating poor LOD quality. Moreover, it cannot ensure that all the inner nodes have a uniform targeted VBO size. On the other hand, [Pauly et al., 2002] might not be the best choice for processing millions of points both quality- and efficiency-wise.

In our novel LOD construction approach, we remove the major limitations of both these approaches. To have s lower resolution LOD points in a parent node, we pass-up s/N representative points for each of the N children, see also Algorithm 3. The basic idea is to cluster points according to their normal deviation. To enable fast clustering, points are first quantized to grid cells and starting with each point being a cluster, clusters are iteratively merged within cells. Since points within a cell are already spatially localized, we can group points differing in their

Algorithm 2 Representatives(node)

initialize empty priority queue Q of list of clusters
set cluster count $c = 0$
setup an auxiliary K^3 grid within node
assign points to grid cells

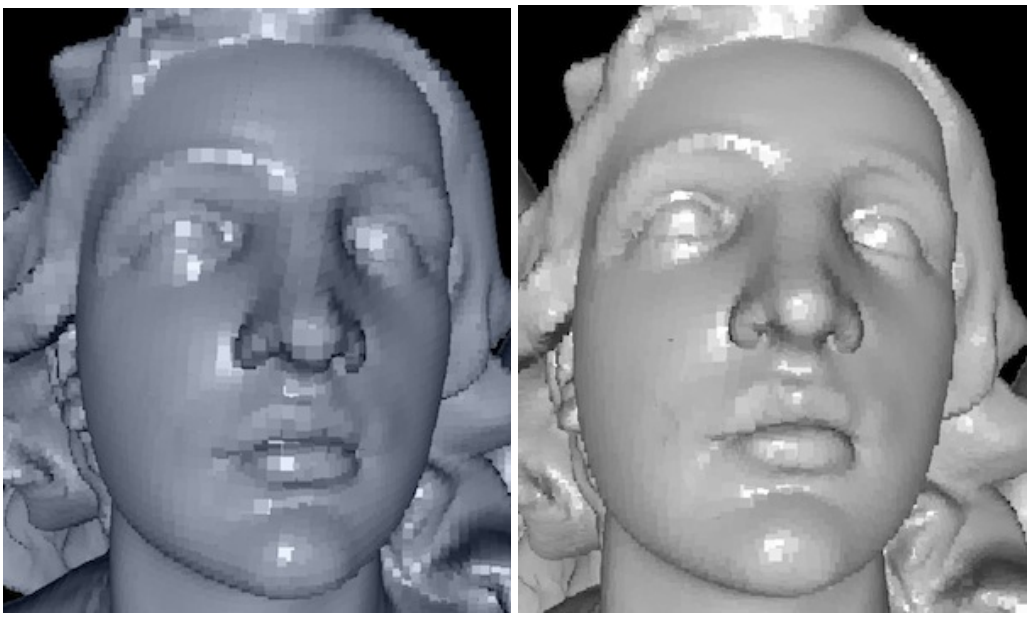
for all grid cells **do**
  initialize list l of clusters to be empty
  initialize point clusters C_i and add them to l
  increment c by $|l|$
  push l onto Q

while $(c > s/N)$ **do**
  pop list l from Q
  merge clusters in l
  push l back to Q
  if (necessary) **then**
    relax threshold θ
  update c
for all clusters in Q **do**
  calculate a representative LOD point

  return c representative LOD points

normal by less than a threshold θ into the same cluster. The list of clusters identified in a grid cell is pushed into a priority queue Q, which is ordered by an error metric based on normal or even color deviation within a cluster. Till a desired number of target clusters (s) are remaining, a list is popped out from Q and points within are merged respecting normal threshold. In the end, a representative point is obtained for each of the remaining clusters thereby giving us $c \approx s/N$ points.

θ itself could be relaxed depending on the number of clusters or level of the node during point simplification. As long as the grid size K is not extremely large or small, it does not have much impact on the LOD quality but might only delay convergence to s/N representatives. Our approach is simple, efficient and provides excellent rendering quality. In Figure 2.5, we compare the quality of multiresolution representation obtained by Wand et. al [Wand et al., 2008] versus our approach using Algorithm 3. It can be noticed that our approach can preserve features by taking the normal deviation into account. Further, it also removes the grid artifacts caused by brute-force combination of all points falling within the same grid cell.



(a) (b)

Figure 2.5: *(a) Multiresolution generated by using simple grid approach in [Wand et al., 2008] versus (b) our normal based clustering.*

2.6 Data Organization

After the multi-way kd-tree construction, all nodes in the tree are rearranged and kept on the disk. Vertex positions and splat radii of all the points are contiguously aligned, followed by normal and, if available, color data. Each of these attributes is quantized and compressed using LZO compression. The position values (x, y, z) of a point are quantized with respect to the minimum and maximum coordinates of the bounding box of the node using 16 bits. Normals are quantized using 16 bits with a look-up table that corresponds to points on a $104X104$ grid on each of the 6 faces of a cube. The radius of a point is quantized using the minimum and maximum splat sizes in its multi-way kd-tree node with 8 bits. Each node in the tree is referenced using its ID and also stores its level in the multi-way kd-tree, bounding box, splats with minimum and maximum radii and size of the node file. Therefore, the multi-way kd-tree itself is small and is loaded at runtime into main memory as an array. With the help of indexing, any node in the multi-way kd-tree can be accessed in constant time.

2.7 Rendering

In interactive rendering applications, it it often preferable to maintain a constant high frame rate, rather than adhering to a strict LOD requirement. To optimize between LOD quality and interactivity, rendering can be controlled using rendering budget B which indicates that no more than B LOD points are displayed every frame. In order to achieve this, we maintain a priority queue Q of LOD nodes at runtime which is ordered by the LOD metric for refinement or coarsening given by Equation 2.2 where l_{max} refers to the maximum level in the tree, l to a node's level, d to the distance from the viewer and c_i to the parameterization constraints. For rendering on a budget, this metric works similar to projected screen-space point size measure but is more efficient to incrementally adjust the rendering front.

$$\epsilon = \frac{c_0(l_{max} - l + 1)}{c_1 d + c_2 d^2} \tag{2.2}$$

Algorithm 3 outlines the basic steps to compute the current rendering front given a budet B, the fanout factor N and VBO size s. Q holds the currently selected nodes for display which are incrementally refined in the $while$ loop according to the budget availability. Our system, however, also supports pixel based rendering wherein no budget limit is imposed. Figure 2.6 compares the rendering quality for pixel based rendering with respect to budget based rendering.

Algorithm 3 LOD selection for rendering on a budget B.

```
initialize empty queue Q prioritized on ε_l
push the root node r onto Q
count = |r|
n = null

while (count − |n| + s · N ≲ B and Q is not empty) do
  pop node n from Q
  if (n is not a leaf) then
    count = count − |n|
    for all children c of n do
      push c onto Q, prioritized on ε_l
      count = count + |c|
  else
    add n to rendering front

  if Q is not empty then
    add nodes from Q to rendering front
```

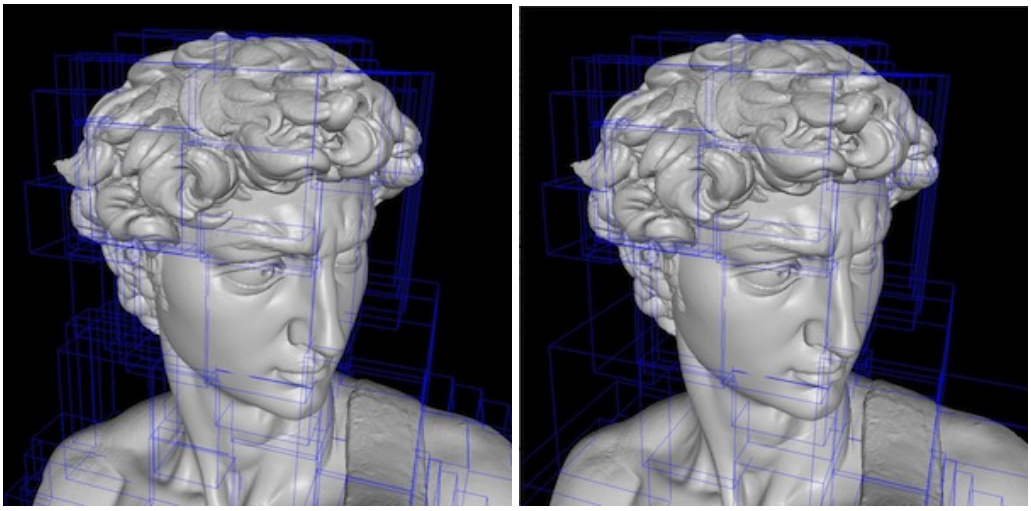
2.8 Asynchronous Fetching

One main feature of our system is its support for asynchronous fetching of data from hard disk to graphics card memory. In interactive applications, LOD changes often occur gradually and one can benefit by delaying the fetching of new data for a few frames which is accomplished by an asynchronous sever thread running concurrently to the main thread. This concept is described in detail in Chapter 4. For point-based rendering, this is simplified further as there is no connectivity limitation unlike meshes. Basically, out-of-core latency can be hidden by delaying child-to-parent and parent-to-child fetches for a few frames and handing them over to an asynchronous thread running concurrent to the main thread. These nodes once ready, are included for rendering in the main thread.

2.9 Results

The proposed method has been implemented in C++ using OpenGL, GLUT and GLSL shaders. We have used two different hardwares to test our implementation:

1. 2x2.66 GHz Dual-Core Intel Xeon processors, NVIDIA GeForce 8800 GT graphics card



(a) (b)

Figure 2.6: *(a) Rendering on pixel error* 3.0 *versus (b) rendering on budget* $3M$.

2. 2x2.8 GHz Quad-Core Intel Xeon processors, NVIDIA GeForce GTX 285 graphics card

The display window size used is 1024×1024 in both the cases.

2.9.1 Preprocessing

In Table 2.1, various preprocessing statistics are listed which include the time required for building the multi-resolution model together with compressing it, as well as the disk usage (uncompressed and compressed) for the given value of N. Throughout all experiments, the grid size used to support clustering was $K = 35$. The preprocessing time depends to some extent on the fanout factor N and hence number of internal nodes, but in general our approach can preprocess about 40K to 60K points per second.

2.9.2 Rendering Quality

Figure 2.7 demonstrates the rendering quality for various models with similar rendering budgets. Despite the models varying considerably in size and with similar rendering budget, the quality of LOD produced seems pretty similar.

Figure 2.8 shows the comparison of sampling and rendering quality depending on the choice of LOD tree data structure, i.e. using octree, kd-tree or multi-way

Model	#Samples	N	Time (s)	Disk Usage (MB)		
				In	Out	Comp.
David 2mm	4129614	3	71	158	229	37
Lucy	14027872	2	310	535	757	143
David 1mm	28184526	5	447	1127	1525	225
St. Matthew	186850683	3	4915	7473	10808	1652
Pisa Cathedral	368585469	4	9044	14743	20937	2973

Table 2.1: *Preprocess measurements on Platform 1.*

kd-tree. Our multi-way kd- tree clearly outperforms the octree due to better sampling adaptivity and is at least equally good as a (binary) kd-tree.

2.9.3 Rendering Efficiency

We conducted four kinds of experiments to evaluate our rendering efficiency :

1. *Varying VBO sizes on same model :* For this St. Matthews model was chosen. Table 2.2 shows the tree structure formation for various configurations indicating fanout factor N, number of points in VBO, as well as number of levels and nodes in the multi-way kd-tree. Corresponding rendering rates are given by Figure 2.9 for two different rendering budgets, $6M$ and $12M$ points. It is clear that the proper choice of N and VBO size can be important to obtain an optimal performance. With smaller VBO sizes there are more context switches and fetches occurring per frame during interactive rendering. On the other hand, large VBO sizes provide high frame and point throughput rates but limit the LOD adaptivity.

Fanout (N)	VBO size	Levels	Nodes
6	4009	7	55987
6	24029	6	9331
3	77822	8	3280
7	85437	5	2801
8	364943	4	585

Table 2.2: *Multi-way kd-tree structure for the St. Matthew model using different VBO sizes and fanout factor N.*

2. *Varying VBO sizes on various models :* As is clear from Table 2.3, rendering efficiency in terms of frames per second and points per second is quite similar for all models despite them varying significantly in size. We achieve rendering rates of nearly 290M points with peaks exceeding 330M even for

larger datasets. Our compression scheme is conservative in terms of disk space and could be improved by compressing refined point splats relative to their coarser parent points. However, our experiments have shown that the performance and display quality are hardly affected at all due to current compression, even though decompression is performed on the CPU.

Model	#Samples	N	VBO Size (K)	Fps	Pps (M)
David 2mm	4129614	3	51	87	264
Lucy	14027872	2	55	85	262
David 1mm	28184526	5	45	86	265
St. Matthew	186850683	3	85	87	265
Pisa Cathedral	368585469	4	90	88	282

Table 2.3: *Rendering performance statistics for various models and VBO sizes on Platform 1, given a rendering budget of $B = 3M$.*

3. *Comparison with kd-tree:* In Table 2.4, multi-way kd-trees are compared with binary kd-trees for similar VBO sizes. It is clear that one can benefit in terms of frame and point throughput with the proper construction.

Model	VBO Size		Fps	Pps	CS	VBOF	Time(s)	Space(MB)
David1mm	55048	Kd-tree	84	254	297170	526	620	234
	13762		44	132	1110910	2135	705	250
St. Matthew	24029		85	255	268723	811	7686	2186
	45614		73	219	406126	1013	6152	1750
David1mm	55048	MW Kd-tree	91	281	86685	162	382	224
	13762		58	173	831707	1399	573	250
St. Matthew	24029		96	289	92580	174	5208	1739
	45614		95	287	120341	202	4977	1674

Table 2.4: *Comparison of performance and preprocessing statistics between kd-tree and multi-way kd-tree using a budget of 3M points on Platform 2. CS refers to total context switches and $VBOF$ to total VBO fetches from disk to graphics memory.*

4. *Comparison with bindless graphics:*

   Table 2.5 compares the rendering performance of our multi-way kd-tree with and without bindless graphics for a budget of 5M. Our experiments suggest that use of bindless graphics does not necessarily imply a performance boost especially within the order of VBO switches we work with. On the other hand, with the proper construction of multi-way kd-tree one can clearly benefit both over binary kd-tree and bindless graphics.

Model	Normal		Bindless	
	fps	pps	fps	pps
Lucy	57	290.00	57	289.67
David1mm	55	284.23	55	282.94
St. Matthew	54	271.45	53	269.33

Table 2.5: *Performance comparison with bindless graphics for a budget of 5M on Linux.*

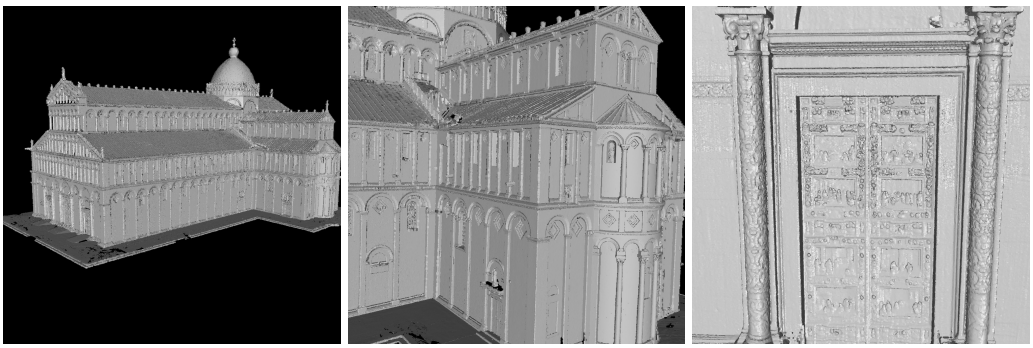
2.10 Discussion

Our simple and efficient framework can handle massive point datasets both for fast preprocessing and rendering purposes. The hierarchical multiresolution preprocessing is flexible due to the underlying data structure of multi-way kd-trees in that it can adapt to a desired LOD granularity by adjusting its fan-out factor N. The LOD quality can be further improved by considering advanced approaches like *k-means clustering* or *entropy based reduction* to obtain k output points.

Further, our algorithm supports adaptive out-of-core rendering on budget as well as pixel error, coupled with asynchronous fetching. As a future work, better compression schemes could be applied by encoding the children attributes with respect to the parent splat. With the optimized compression, the method can be extended for networked rendering.

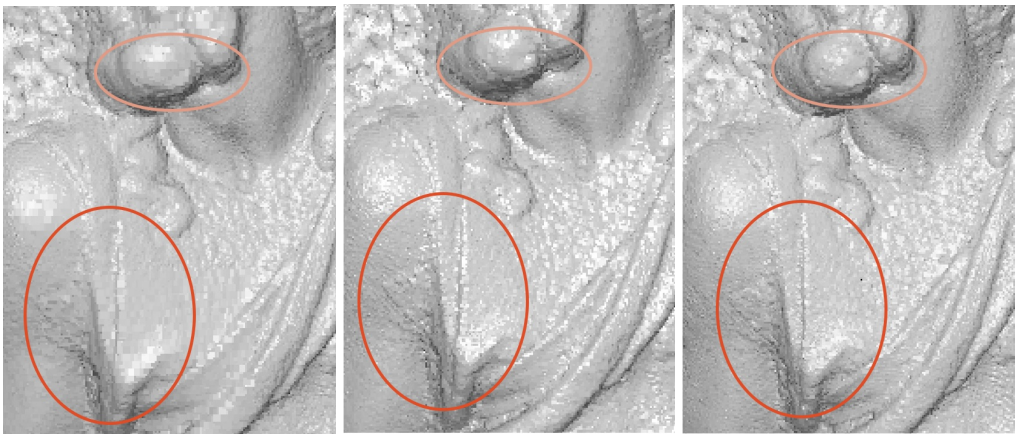
(a)

(b)



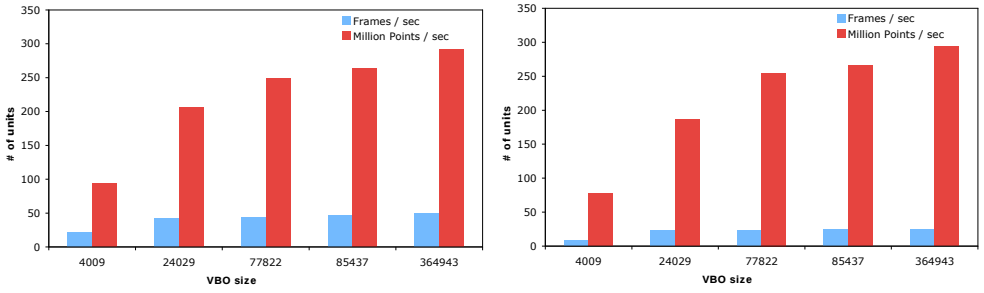
(c)

Figure 2.7: *Varying zoom views of the David (28M samples), St. Matthews (187M samples) and Pisa Cathedral (368M samples) models.*



(a) (b) (c)

Figure 2.8: *Comparison of LOD sampling quality depending on the choice of LOD tree data structure, i.e. octree, kd-tree or multi-way kd-tree respectively using maximal node sizes.*



(a) (b)

Figure 2.9: *(a) Rendering rates for a rendering budget of (a) 6M and (b) 12M, on Platform 1.*

CHAPTER 3

PBR - ADVANCED FEATURES AND PARALLEL RENDERING

3.1 Background

In Chapter 2, a framework to preprocess and render points was presented. The framework enables quick and high quality, simple preprocessing together with efficient rendering. Points as primitives are particularly attractive due to simplicity and high quality rendering. On the other hand, triangles score on quality over points. In order to present points as a viable alternative to triangles, we further explore options in this section of thesis. This includes in-depth analysis of preprocessing approaches and operations leading towards bridging the quality gap between triangles and points. To this end, we suggest more sophisticated operations like geomorphing and deferred blending on large point models. Further, we also introduce PBR of large models in the context of cluster-based parallel and multi-display rendering environments. These features are added on top of the existing point renderer employing multi-way kd-trees.

3.2 Advanced Preprocessing

In Chapter 2, we introduced a fast preprocessing method capable of generating quick LODs for out-of-core models. It is shown to furnish high quality in comparison to existing approaches like [Wand et al., 2008]. We present here two more

preprocessing methods, *entropy-based reduction* and *k-clustering*, to compare the statistical and visual performance of normal deviation based clustering method. All the proposed methods operate on a given point set to generate k output clusters. We keep this constraint so as to generate balanced LOD in our multi-way kd-tree data structure. Thereafter, we compare these proposed methods with the iterative simplification presented in [Pauly et al., 2002] which also targets to produce k clusters for a given point cloud.

The aim of all the presented approaches is to generate k best clusters through global simplification of the input dataset. In order to facilitate fast clustering and neighborhood search, all these methods employ a virtual K^3 grid.

3.2.1 Entropy-Based Reduction

This technique targets to create k points using entropy as an error metric, given by Equation 3.1. Here $Entropy_i$ refers to the entropy of the i-th point which is calculated using θ_{ij} and $level_i$. θ_{ij} is the normal angle deviation of i-th point to its j-th neighbor. $level_i$ indicates the number of times i-th point is merged. $Entropy_i$ is actually a reflective of the amount of entropy induced into the system by combining a point with its neighbors and hence the summation in Equation 3.1. In this approach, at each iteration we pick up a point, which when merged with its neighbors introduces minimum entropy to the system. The more often a point is merged, the higher is its level and hence its entropy. On the other hand, the lower the normal angle deviation is with its neighbors, the smaller the entropy.

$$Entropy_i = \sum_j \frac{1 + level_i}{1 + cos\ \theta_{ij}} \tag{3.1}$$

The basic steps of the entropy-based reduction are outlined in Algorithm 4. The algorithm begins by initializing all points by finding their neighbors and computing their entropy using Equation 3.1. All points are marked valid and their level initialized to 0. A global minimum priority queue $\boldsymbol{Q}$ is prioritized on the entropy values and all points are pushed into it. Thereafter, the top point from $\boldsymbol{Q}$ is popped out and merged with its neighbors to create a new splat. All combined neighbors are marked invalid and the entropy is recomputed for the newly created point before adding it back to $\boldsymbol{Q}$ (lines 8-19). This procedure is repeated until k points are obtained or $\boldsymbol{Q}$ is empty. The final set of points is obtained by popping $\boldsymbol{Q}$ in the end and collecting all valid points.

3.2.2 k-Clustering

Standard k-clustering is the natural choice that comes up in one's mind when k clusters are desired. The basic method is derived from k-means clustering [Mac-

Algorithm 4 Entropy-Based Reduction.

Require: ***S***, the input point set
Initialize min priority queue ***Q*** on entropy
Initialize $c = |S|$
Mark all points valid
Set level of all points to 0

Establish a K^3 grid in the node ***S***
Compute neighbors **for each** point **in each** grid cell
Compute entropy **for all** points using Equation 3.1 and push them into ***Q***

while ($c > k$ **and** ***Q*** is not empty) **do**
  p = Pop ***Q***
  if ($p.valid$) **then**
    – Increment $p.level$ by one
    – Compute new position, normal and radius using valid neighbors of p
    – Mark all neighbors of p as invalid
    – Update neighbors of p and recalculate their entropy
    – Recompute entropy of p
    – Push p to ***Q***
    – Decrement c for each point freshly marked invalid

Output: Valid points in ***Q***

Queen, 1967] that aims to partition n observations into k clusters such that each observation is grouped with the cluster having closest mean. For this, the observations are repeatedly moved among the clusters until an equilibrium is attained. The complexity of solving the original k-means clustering problem is NP-hard. For our purpose, we need a simpler formulation of k-means clustering to obtain k points from the given set ***S***, which are good enough representatives of their cluster such that the remaining $|\boldsymbol{S}|$ - k points can be grouped into one of the chosen k points.

The choice of k clusters (set ***M*** in Algorithm 5) is crucial to obtain high quality aggregate clusters. However, to obtain an initial crude guess for the k seed points, we use the hashing method proposed in [Zhang and Pajarola, 2007] which is based on a separation of the point data into non self-overlapping minimal independent groups. We divide the original set S of splats into 8 groups using such an online hash algorithm which is faster, since we need an initial estimate which is anyway refined further. However, one could employ either the offline or online algorithm as suggested in [Zhang and Pajarola, 2007]. Furthermore, one can choose to divide into more or less than 8 groups without loss of generality. Thereafter, we add or remove points in ***M*** such that the model is adequately sampled with minimal overlap and complex regions having higher sampling density.

The basic steps followed to obtain k clusters are given in Algorithm 5. Following division of input points into 8 groups using the hashing method proposed in [Zhang and Pajarola, 2007], the group with maximum number of splats is picked (line 3, 4) which constitutes the initial ***M***. Thereafter, splats are pushed into ***Q*** based on overlap priority and points are removed from ***Q*** until there is no more overlap in ***M*** (lines 5-11). Overlap can be determined using overlap length, area or even volume between two or more splats. In order to enforce more uniform sampling, points are selected from ***(S - M)*** which have none of their neighbors in ***M*** (lines 12-18). This gives us a good initial estimate of ***M***. ***M*** itself is further refined until it has only k points left. If ***M*** has more than k elements, the point with the least deviation with its surrounding set of points is removed (lines 16-18). It should be noted that at this point (following line 11), ***M*** has almost no overlap. Therefore, we determine the deviation by taking an extended neighborhood which is in fact all the points in ± 1 grid cells of the current cell that contains the point. To ensure that nowhere point densities are significantly reduced, we include the number of points in the extended neighborhood set in the error metric. On the other hand, if ***M*** has fewer than k elements, points are chosen from ***(S - M)*** which have the least overlap in ***M*** (lines 19-20). Following this, each of the remaining points in ***(S - M)*** are grouped with a neighbor point present in **M** that has least minimum distance with it(lines 21-22). The set of output points is simply obtained by merging all points in M with their neighbors.

Algorithm 5 k-Clustering.

Require: ***S***, the input point set

Establish a K^3 grid in the node ***S***
Compute neighbors **for each** point **in each** grid cell
Group points into 8 groups using online group hashing as given in [Zhang and Pajarola, 2007]
Pick group with most number of splats, ***M***
Initialize maximum priority queue ***Q*** on overlapping extent with neighboring splats
Calculate overlap **of each** point in ***M*** with its neighbors and push it into ***Q*** if there is an overlap

/*— Remove the points from over-sampled regions —*/
while (there is overlap in ***Q***) **do**
  – p = Pop ***Q***
  – Remove p from ***M***
  – Recalculate overlap of its neighbors

/*— Add points to the under-sampled regions —*/
for each point p in ***(S - M)*** **do**
  if p and none of its neighbors are in ***M*** **then**
    – Add p to ***M***

/*— Iterate for k points by removing or adding points as required —*/
while ($|\boldsymbol{M}| > k$) **do**
  – Remove a point from ***M*** that has least deviation with its (extended) surrounding points
while ($|\boldsymbol{M}| < k$) **do**
  – Add a point from ***(S - M)*** to ***M*** that has least overlap in group ***M*** with its neighbors

for each point p in ***(S - M)*** **do**
  Group p with its neighbor in ***M*** having minimum distance

Output: Points in ***M*** merged with their neighbors

3.3 Geo-Morphing

Multi-way kd-trees constructed with a high fanout factor may exhibit jumps in the LOD transition during rendering. In order to make the LOD shift continuous, we additionally apply geo-morphing. This can easily be implemented in our case for all the proposed preprocessing methods by storing correspondences between the children and parent splats during preprocessing. When constructing the multi-way kd-tree hierarchy and selecting the representative points, the number of finer LOD points that are aggregated in a new coarser LOD point is maintained with the representative point in the parent node.

Given the asynchronous front based fetching, during each *parent-to-child* and *child-to-parent* transition, the additional data required for geo-morphing is supplied as a per vertex texture to the vertex shader for interpolation. To achieve this, each splat in the currently rendered VBO is also given the target splat position, size and normal that will replace it. This is simple in our case as each parent splat maintains the count of its refined splats in the child VBO and hence can compute the index to these splats. The transition from a coarser parent LOD point to a set of refined child LOD points (or the other way around) is then smoothed over a few frames during which the positions, sizes and normals of source and target splats are slowly interpolated. Figure 3.1 shows three intermediate frames taken from a geo-morphing parent-to-child transition sequence on David1mm model.

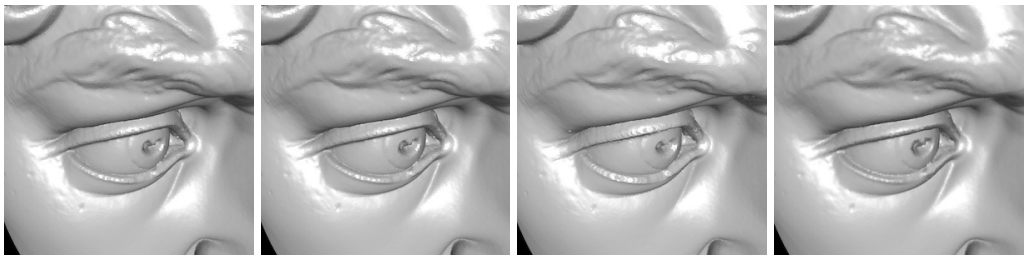
(a) (b) (c)

Figure 3.1: *Three intermediate frames in a parent-to-child transition sequence showing geo-morphing on David1mm model.*

3.4 Smooth Point Interpolation

Real-time smooth point interpolation for small models is easy to achieve with conventional blended splatting algorithms. However, most of these approaches are not well suited for very large scale point sets as ours since they are too resource intensive and slow down rendering speed significantly. Object-space smoothing approaches often use two passes over the point geometry and another pass over the image. To avoid multiple processing and rasterization of geometry, we adopt the deferred blending approach as introduced in [Zhang and Pajarola, 2007]. While rendering, point splats in a node are separated into eight groups such that the overlap within a group is minimal. This is done based on an online hashing scheme [Zhang and Pajarola, 2007] and can be combined with the asynchronous loading of LOD nodes from hard disk. Each group is then rendered into a separate frame buffer texture and finally the eight partial images obtained from these groups are blended using the algorithm given in [Zhang and Pajarola, 2007] in a final fragment pass.

In Figure 3.2, we compare the rendering quality between different kinds of rendering primitives together with deferred blending using simple OpenGL points.



(a) (b) (c) (d)

Figure 3.2: *Choice of splat primitive: (a) Square OpenGL points, (b) round antialised points, (c) elliptical depth-sprites and (d) blended splats.*

3.5 Parallel Rendering

The integration of sophisticated features like geo-morphing and smooth blending together with the capability of rendering several hundreds of millions of points per second, makes point-based rendering attractive for large display walls or multi-machine rendering. Not only rendering work load can be distributed over available resources, but also a wider range of applications can employ out techniques for efficient visualization of high quality data.

Among various parallel rendering framework options, like VR Juggler [Bierbaum et al., 2001] and Chromium [Humphreys et al., 2002], we have chosen Equalizer [Eilemann et al., 2009] for its configuration and task distribution flexibility and extensibility features to port out point renderer to run on a cluster driving multiple displays. Many works (e.g., [Goswami et al., 2010a; Correa et al., 2002]) leverage the parallel power of multiple machines to achieve speed-up or for large wall based displays using triangular primitives. Similar solutions (e.g., [Hubo and Bekaert, 2005; Corrêa et al., 2002]) have been introduced for parallel point-based rendering of moderately sized models. None of these works, however, compare points as parallel rendering primitives on large wall displays with triangles for very large models. This thesis further provides an initial evaluation of parallel rendering in the context of point-based graphics in comparison to triangles, both on performance and quality front.

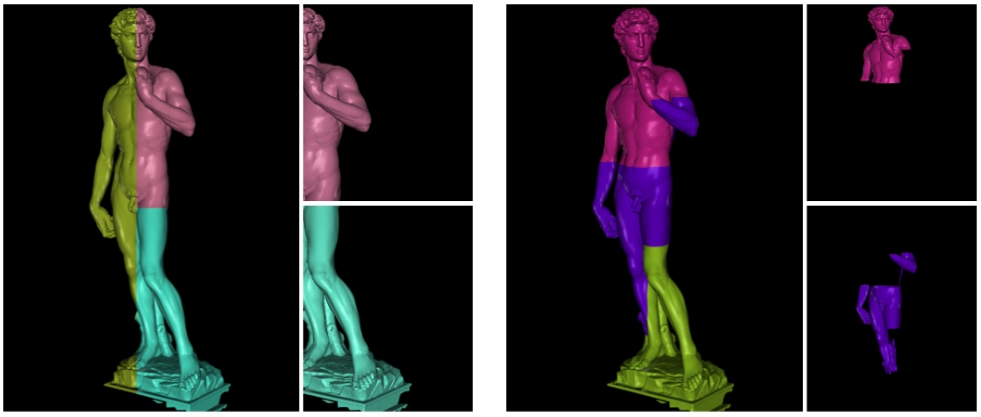
3.5.1 Decomposition Modes

Equalizer supports two basic task partitioning modes which are directly applicable in our case: screen domain or sort-first and database domain or sort-last [Eilemann et al., 2009].

- Sort-first or screen-space decomposition divides the task in image space. Therefore, all rendering machines receive the complete range $[0, 1]$ for database but a subset of overall frustum for rendering. Each of the machines performs culling with the supplied frustum and renders the selected multi-way kd-tree nodes. This configuration is particularly useful for wall displays, see also Figure 3.3(a).

- Sort-last or database decomposition refers to the division of the geometry data among the rendering machines. Each of the rendering clients obtains a range $[l, r]$ from the Equalizer server which is a subrange in the interval $[0, 1]$. Therefore, any given machine renders the geometric data lying in its supplied range based on some subdivision. Our division strategy is same as [Goswami et al., 2010a] wherein we divide the list of multi-way kd-tree nodes post tree traversal, among the machines based on this range, see also Figure 3.3(b). This achieves an implicit load balancing of rendering burden among machines.

The basic motivation to choose points over triangles on multi-machine large displays comes from the possibility of more efficient rendering with not much loss in quality. The rendering data can be more easily divided among the machines without worrying about the connectivity between meshes of different resolutions. The quality gap between triangles and points can be partly bridged by using more



(a) (b)

Figure 3.3: *Task division mode on our point renderer using (a) frustum or sort-first modes (b) database or sort-last*

sophisticated operations like geo-morphing. In Figure 3.4, St. Matthew model is running on a Linux cluster with 10 nodes with a maximal rendering budget of $3M$ per machine at 15 fps.

Figure 3.4: *St. Matthew model on multi-display cluster using glPoints and a rendering budget of 3M per machine at 15 fps.*

3.6 Results

The standalone version of the proposed method has been implemented in C++ using OpenGL, GLUT and GLSL shaders. The results have been evaluated on

a system with 2x2.8 GHz Quad-Core Intel Xeon processors, NVIDIA GeForce GTX 285 and a display window of 1024×1024 pixels.

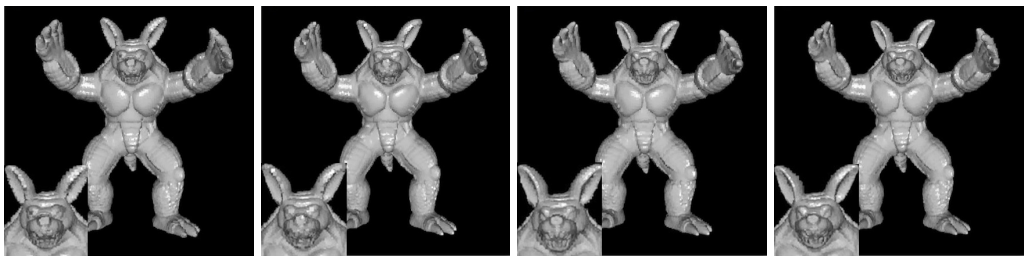
The parallelized version of this software ported to *Equalizer* is used to run experiments on a PC cluster of six Ubuntu Linux nodes with dual 64-bit AMD 2.2 GHz Opteron processors and 4GB of RAM each. Each computer connects to a 2560×1600 LCD panel through one NVIDIA GeForce 9800 GX2 graphics card, thus resulting in a 24Mpixel 2×3 tiled display wall. Each node has a 1Gb ethernet network interface, which is also utilized to access out-of-core data from a central network attached disk. For comparative analysis of quality and efficiency, we use a simple polygonal rendering application, *eqPly*, which renders triangle mesh data in parallel using optimized static display lists.

3.6.1 Preprocessing

In this section, we describe and compare the additional results obtained using various point simplification methods. Figure 3.5 compares the outputs obtained using: normal deviation clustering, entropy-based reduction, k-clustering and iterative simplification ([Pauly et al., 2002]). All these methods produce a desired number of output (representative) points k, for a given input point set. It shows that simplification through normal deviation and k-clustering produce the best results followed by entropy-based reduction and iterative simplification. Normal deviation and entropy-based reduction are simple to implement. The relative loss of quality in iterative simplification is attributed to the fact that each time a point pair is chosen to collapse, it replaces it with a new representative point with larger radius which results in accumulating conservative coverage attributes. It also needs a higher number of iterations to achieve k points which ultimately leads to a larger overlap as compared to other methods. In our methods, a group of splats are replaced by a single representative point, thereby reducing this overlap. Furthermore, as listed in Table 3.1 simplification through normal deviation runs much faster than all other methods producing high quality clusters. In fact, all the three proposed methods reduce preprocessing time while enhancing point quality as compared to [Pauly et al., 2002] while still yielding the desired k clusters. k-clustering can be chosen over normal deviation if strict quality control is preferred over time.

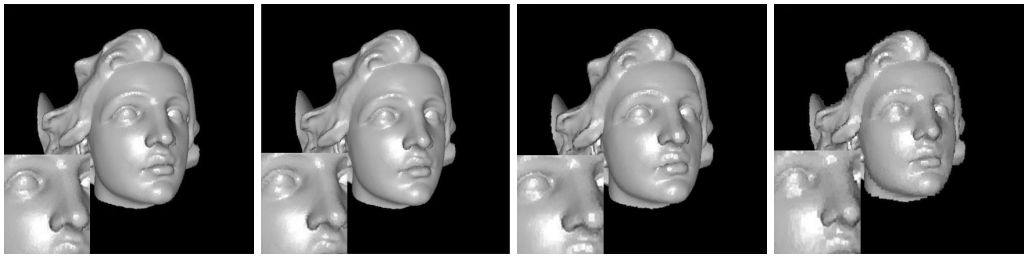
3.6.2 Geo-Morphing

Table 3.2 compares the performance between standard and geo-morphing enabled rendering. It can be noticed that the penalties are agreeably low when enabling geo-morphing.



(a) Armadillo, Input: 173K, Output: 39K

(b) David Head, Input: 417K, Output: 77K



(c) Lucy Head, Input: 513K, Output: 47K

Figure 3.5: *Point clustering created with (from left to right) normal deviation, k-clustering, entropy-based reduction and Pauly's iterative simplification methods respectively for three different models.*

3.6.3 Parallel Rendering

In Figure 3.7, we compare the quality between triangles and points on multi-machine large displays for different rendering budgets. Figure 3.6 summarizes the comparison on multiple machine cluster with varying rendering budgets for points vs. triangles. We make two observations here:

1. Rendering with points is about 3-4 times faster even when using the maximal rendering budget.

2. There is not significant quality difference between Figure 3.7(a) and 3.7(b) which compare the maximal budget rendering using both kinds of primi-

Model	In	Out	Nor. D.	k-Clust.	Entr. R.	Iter. S.
Armadillo	173K	39K	1.24	3.08	4.27	7.72
David Head	417K	77K	1.63	6.91	7.55	26.73
Lucy Head	513K	47K	1.69	7.47	9.44	38.43

Table 3.1: *Comparison of preprocessing time (in sec) on various models using normal deviation, entropy-based reduction, k-clustering and iterative simplification ([Pauly et al., 2002]) methods. Input and output point counts are as given in each case.*

Model	#Samples	N	VBO(K)	Fps	Pps(M)	Fps	Pps(M)
				Normal	Normal	Geo	Geo
David 2mm	4129614	3	51	95	288	80	244
Lucy	14027872	2	55	98	294	80	241
David 1mm	28184526	5	45	94	290	78	241
St. Matthew	186850683	3	85	97	290	81	240
Pisa Cathedral	368585469	4	90	93	285	77	237

Table 3.2: *Rendering performance statistics for various models and VBO sizes, given a rendering budget of $B = 3M$.*

tives. Even as the rendering budget is reduced, the frame rates obtained increase but quality is not affected noticeably.

This implies that one could obtain close to an order of magnitude of speed-up when rendering points in comparison to triangles without losing too much on the quality front.

In Figure 3.8, we compare the LOD quality generated by our method with respect to state of the art approaches. [Rusinkiewicz and Levoy, 2000] starts with a mesh and uses much finer granularity to produce lower resolution and tree traversal as compared to ours. On the other hand, in [Gobbetti and Marton, 2004a] hierarchy construction purely relies on point subsampling leading to a somewhat noisier LOD with lesser budget.

3.7 Discussion

We have demonstrated that points as primitives are more efficient than triangles, even on large displays with multiple machines. The quality gap between the two can be bridged by using more sophisticated operations like geo-morphing and smooth point interpolation. Further, simple techniques can be used for task division for parallel rendering both in screen space and database domain. One suggested extension could be to make the approach suitable for streaming over

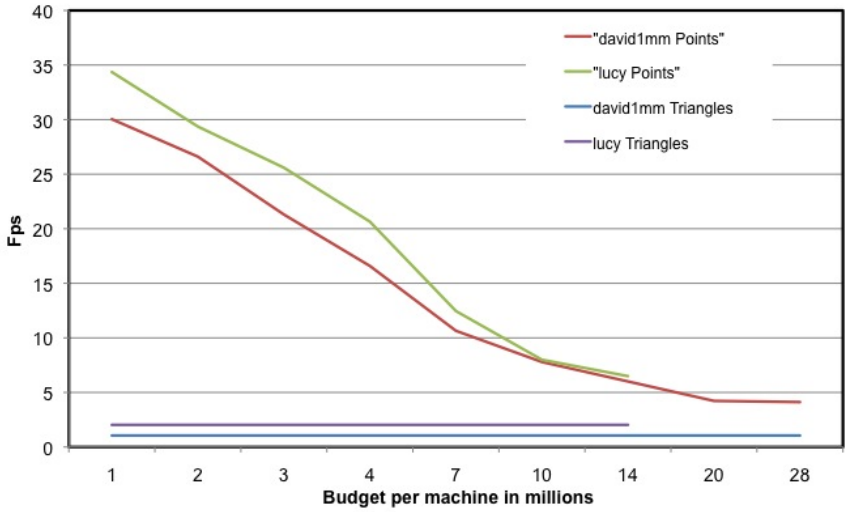
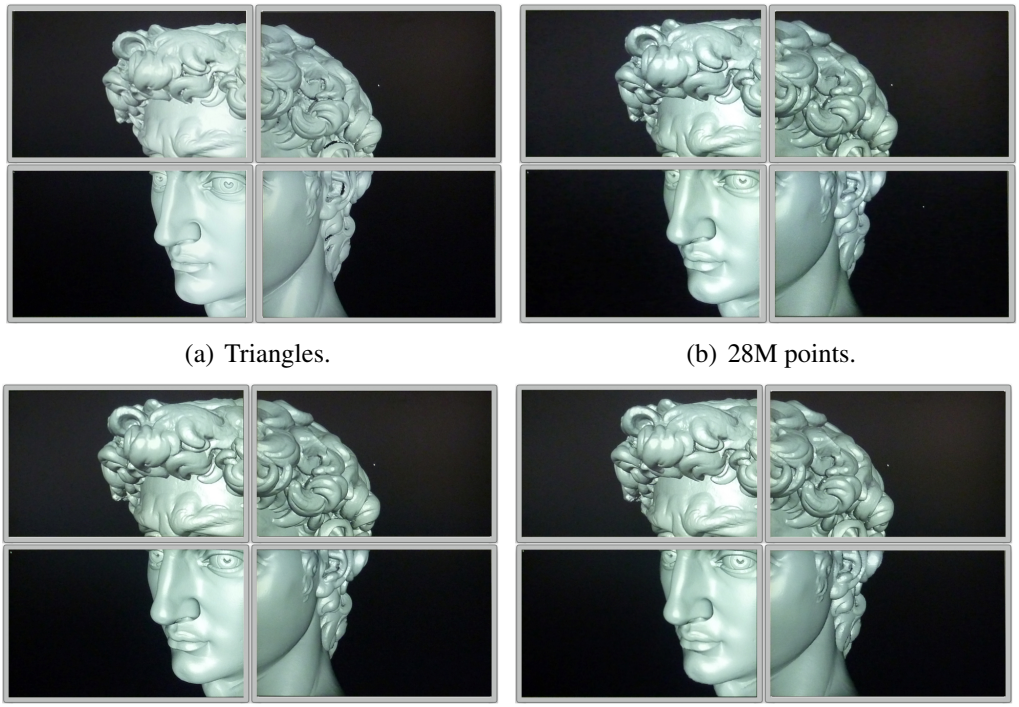


Figure 3.6: *Comparing triangles and points as rendering primitives on parallel multi-machine large displays.*

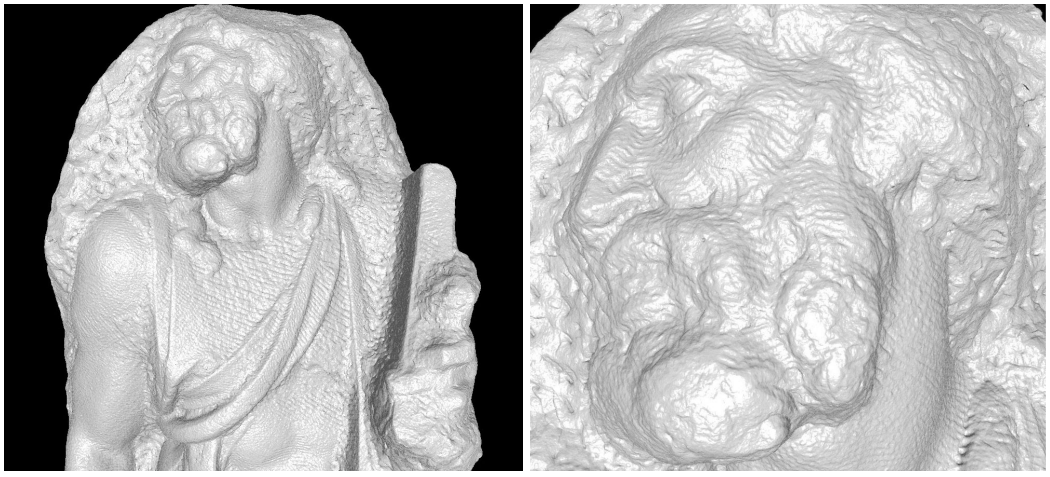
network and remote rendering. This can be achieved by applying better compression schemes to reduce the per node VBO data size while still allowing it to be used by the GPU with minimal runtime overhead on the CPU.



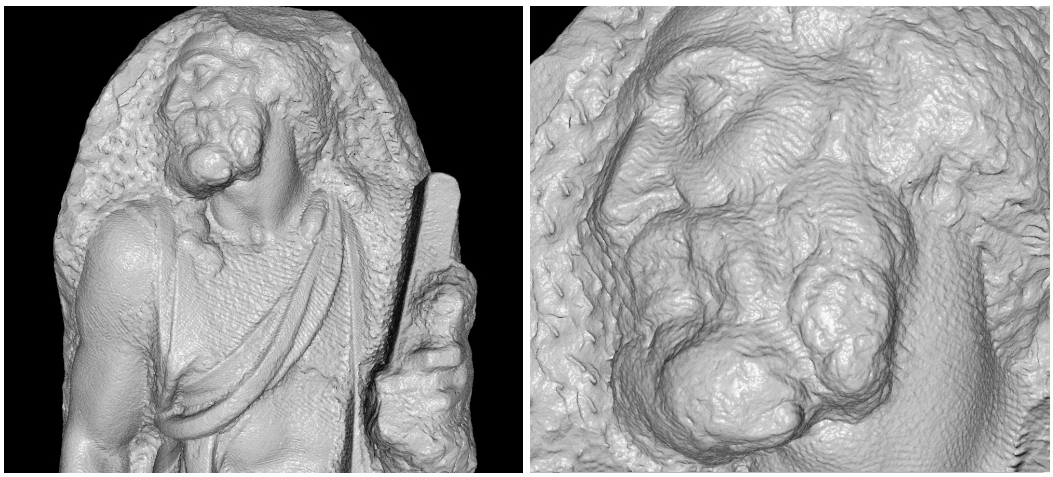
(a) Triangles. (b) 28M points.

(c) 2M points. (d) 1M points.

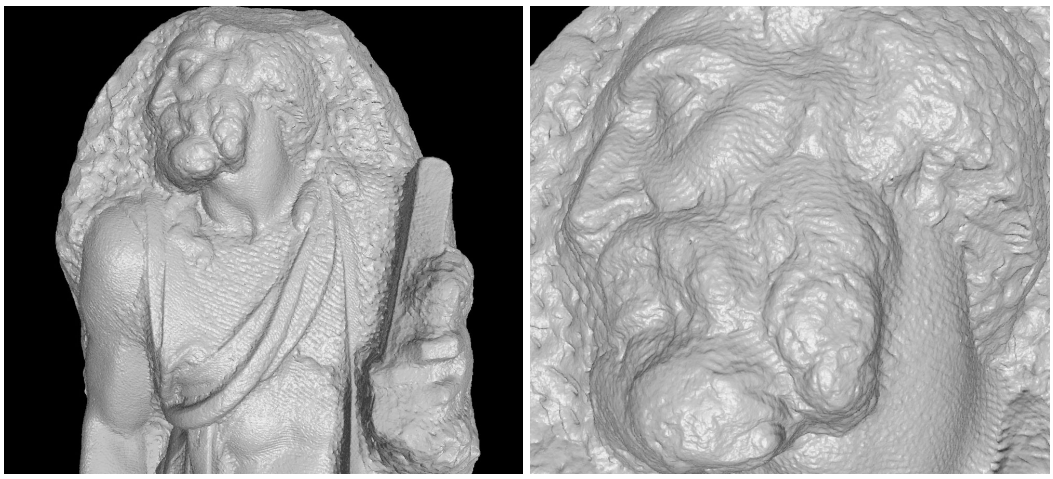
Figure 3.7: *Quality comparison between triangles and points as rendering primitives on various rendering budgets per machine for points (a) Triangles (b) Points (28 M) (c) Points (2 M) (d) Points (1 M).*



(a) Layered Point Clouds.



(b) QSplat.



(c) Ours.

Figure 3.8: *LOD quality comparison between (a) Layered point clouds, (b) QSplat and (c) our approach for a rendering budget of approximately 3 million points.*

CHAPTER 4

TERRAIN RENDERING

4.1 Terrain Datasets

Efficient real-time visualization of large terrain data has been an active field of research for past several years. Similar to point models, with the growth of rendering hardware the precision of digital elevation models (DEMs) has increased many times. Due to this ever increasing complexity of DEMs, real-time display imposes strict efficiency constraints on the visualization system, which is forced to dynamically trade rendering quality with usage of limited system resources. In a number of visualization application terrain rendering itself is only one task and might be even relegated to the background. Today, there are a whole bunch of applications hosted on the internet that allow users to interact with gigasize datasets, eg. Google maps. Therefore, terrain visualization has been an ongoing area of research with newer methods improving upon the existing ones or adding new features and functionalities.

4.2 Terrain Principles

This section presents a brief overview of the terrain preprocessing and rendering principles.

4.2.1 Adaptivity Techniques

Early works on terrain rendering used a static level-of-detail based on tiled blocks wherein the terrain was divided on tiled square blocks and a few representations are precomputed and stored off-line. The problem with this approach was its limited adaptivity and the occurrence of cracks at the boundaries between different resolution representations. Thereafter, [Losasso and Hoppe, 2004] introduced the geometry clipmap in which the terrain data is represented as a pre-filtered mipmap pyramid (usually at successive powers-of-two), along the lines of LOD treatment of images. As the viewpoints moves, the clipmap levels shift and are incrementally refilled with data. Cracks and T-junctions are avoided by stitching the level boundaries with zero-area triangles. This allows on-the-fly terrain synthesis and progressive compression based on height values of previous or next level.

However, from the point of view of rapid adaptive construction and continuity in display of terrain surfaces, quadtree or triangle bintree triangulation offers one of the most promising approach. The main idea behind these kind of approaches is to build a regular multiresolution hierarchy by refinement or by simplification. Whereas in refinement, starting from an isosceles triangle the longest edge is bisected creating two smaller right triangles, during simplification pairs of right triangles are selectively merged. Further, from the refined definition presented in [Sivan and Samet, 1992; Sivan, 1996], one can consider the restricted quadtree triangulation and triangle bin-tree to produce the same class of adaptive grid triangulations.

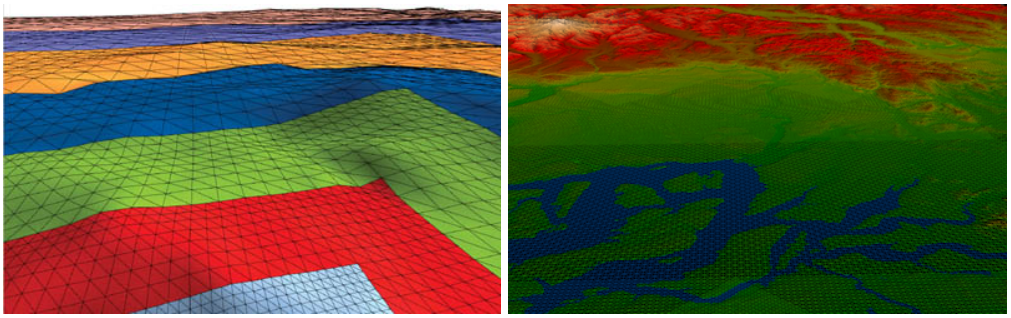
4.2.2 Batch Triangles

In order to be able to couple modern GPUs which are capable of processing several millions of triangles per second with CPU, the workload of the latter has to be reduced to few instruction cycles per rendered triangle. The natural choice therefore, is batched or clustered triangles which are chosen by the CPU on a coarse metric and then supplied to GPU for further processing. It is further expected that the geometry is made available on the graphics card memory and no editing is done on it. [Duchaineau et al., 1997] algorithm improved rendering performance through the addition of coarse-grained on-board caching using triangle bin-trees. BDAM [Cignoni et al., 2003], on the other hand, combined regular and irregular triangulations in the same GPU friendly framework.

4.2.3 LOD Error Metric

The hierarchy of height data is constructed on the clustered triangles as a bintree or quadtree. Instead of per primitive, an error value is associated with each of the units of hierarchical data structure. This error metric itself is a combination

of object-space and screen-space error. The object-space error is built recursively bottom-up using *error saturation*: each vertex or triangle stores the maximum value of all propagated errors (usually of the descendants) and its own computed error, and propagates this value further along the dependency graph. This has to be combined with a dynamic view-dependent parameter which controls refining or coarsening of the regions depending on their proximity from the viewpoint. Further, this error metric can be additionally adjusted to make sure that adjacent selected clustered-triangles do not create cracks or T-junctions as in case of RASTeR which employs octagon error-metric. Figure 4.1 shows two different schemes for creating LOD terrain meshes. While one creates adaptive triangulation simply based on the distance from the viewer, the other one is more sophisticated and also incorporates the object space features to generate triangles.



(a) (b)

Figure 4.1: *LOD view for terrain (a) Simplistic, triangle size grows with distance from the camera, courtesy: NVIDIA (b) Sophisticated, also takes into account terrain features together with distance for adaptivity.*

4.2.4 Related Work

The techniques developed for terrain rendering generally rely on a variety of methods specifically tailored to 2.5-dimensional surfaces, see also Figure 4.2. To improve rendering performance, an appropriate level-of-detail (LOD) of the terrain data is selected which is sensitive to surface features and viewing parameters. In earlier algorithms, optimization is performed on individual geometric primitives, such as vertices to choose LOD. However, with current generation CPU-GPU configurations, LOD rendering is optimized on batched graphics primitive. This prevents CPU from consuming too much computational time which might starved fast-paced graphics hardware pipeline. Apart from providing LOD support and fast retrieval from hard disk, a terrain renderer is expected to provide maintain a continuous display of LODs. However, due to expensive hard disk to memory

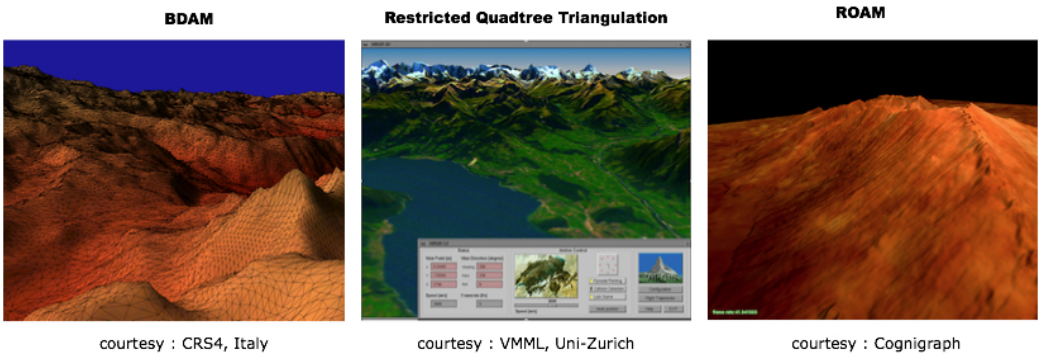


Figure 4.2: *Terrain preprocessors and renderers, (a) BDAM - Batched Dynamic Adaptive Meshes for High Performance Terrain Visualization, (b) Large Scale Terrain Visualization Using the Restricted Quadtree Triangulation and (c) ROAM - Realtime Optimally Adapting Meshes.*

fetching, this particular requirement might turn out to be hard to satisfy. Fortunately, asynchronous methods come to rescue here where immediate fetching is deferred by a few frames hiding out-of-core latency. In this part of the thesis, we describe our approach to allow continuous display using a novel front-based asynchronous fetching mechanism.

LOD based polygonal meshing and multiresolution rendering has received much attention over the last decade [Luebke et al., 2003]. Exploiting the regular grid structure of DEMs, multiresolution restricted quadtree or bintree approaches such as [Sivan and Samet, 1992], [Lindstrom et al., 1995], [Duchaineau et al., 1997], [Pajarola, 1998] have shown generally to be more performant than irregular triangle meshes (TIN) based methods as proposed in [Puppo, 1996], [Hoppe, 1998], [De Floriani et al., 1996].

Among CPU-based approaches, RUSTIC [Pomeranz, 2000] and CABTT [Levenberg, 2002] improve the ROAM approach [Duchaineau et al., 1997] by coarse-grained on-board caching of static and dynamic triangle clusters respectively. On the other hand, GPU-based approaches like [Cignoni et al., 2003], [Lario et al., 2003] and [Hwa et al., 2005] demonstrate the performance benefits of coarse LOD adaptation despite the increased number of per-frame rendered triangles. For further details and comparisons of the different related techniques we refer the reader to the survey [Pajarola and Gobbetti, 2007].

In the next section, we provide a brief recap of RASTeR principles. Thereafter, this thesis focuses on three main contributions of our terrain renderer, RASTeR: height data compression, texture selection and asynchronous data and texture fetching mechanism.

4.3 RASTeR Summary

Real-time adaptive simplification and terrain rendering (RASTeR) system is based on two conceptual units: *K-patches* for triangulation and *M-blocks* for data.

K-patches are regularly triangulated clusters with a constant number K of vertices along each triangle-patch edge, see also Figure 4.3. In our case, K itself is always a power-of-two plus one. K-patches can be interpreted as macro triangles of a batched *meta* bintree and thus could be arranged in a triangle strip sequence. The orientation of a K-patch is always an instance of one of eight basic isosceles triangle types. The hierarchy of K-patches is organized in a meta bintree where each K-patch represents a node. K-patch is only a triangulation unit which is initialized at the beginning of application and never explicitly stored. Cracks between adjacent K-patches of different LODs can be avoided by constraining their height difference in meta bintree by at most one.

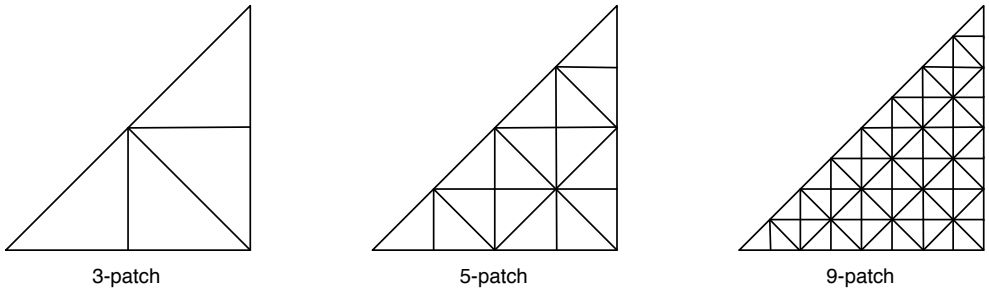


Figure 4.3: *Triangle K-patches for different sizes of K*

An M-block is a square block of a regular grid of height sample data - and possibly other attributes such as surface normal - stored in a file on the disk. All M-blocks are defined to be of equal size, that is, they have the same number of MXM of vertices with $M = 2^m + 1$. M-blocks are organized in a quadtree hierarchy, with each M-block representing a node. In this reduced resolution pyramid the resolution of terrain changes by a factor of two between levels. The relation between K-patch bintree and M-block quadtree is demonstrated in Figure 4.4. The basic RASTeR mechanism is shown in Figure 4.5. For more details on RASTeR, please see [Bösch et al., 2009].

4.4 Height Data Compression

M-block constitutes the data unit of RASTeR. These M-blocks are kept both in uncompressed and compressed formats on the hard disk. In our implementation, M-block data is reversibly compressed using JPEG2000 format which offers a compression factor or about 2. Decompression is done both synchronously and

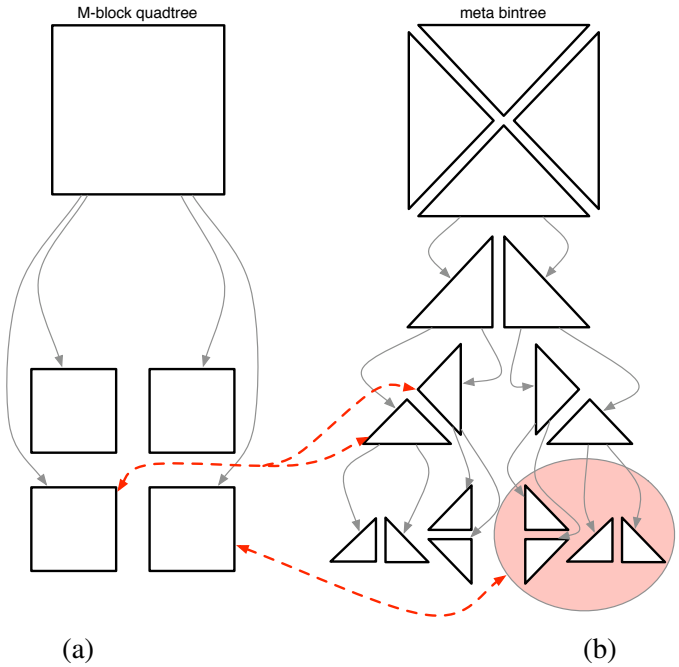


Figure 4.4: *M-block height-field quadtree nodes (a) can correspond to different K-patch meta bintree nodes (b) depending on the selected LOD triangulation. Elevation data in the M-block quadtree is separated from the triangle mesh connectivity in the K-patch meta bintree.*

asynchronously. Finally, M-blocks are passed on as Vertex Buffer Objects (VBO) to the graphics card.

4.5 Texture Selection

To match the K-patch and M-block structures, textures are managed in square units and organized in a texture mipmap pyramid [Williams, 1983]. The hierarchy of texture units is closely linked to that of M-blocks. Further, since texture units typically occupy more space than the M-blocks, corresponding to each M-block several resolutions of texture are kept. The texture resolution used for one or more related K-patches is chosen depending on the distance from the camera. But to limit excessive use of high-resolution textures, due to triangle patches spanning a wide depth, we additionally constrain the texture mipmap levels such that only a certain percentage is rendered at high(est) resolution near the viewpoint. Each texture unit can be compressed using GPU compression to almost 1/6 of its original size. Figure 4.7 demonstrates the texture selection with 3 different DEM

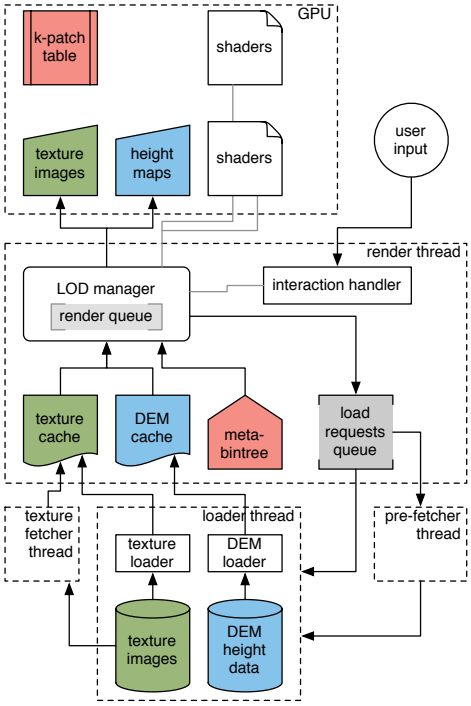


Figure 4.5: *RASTeR system and resource management*

datasets.

4.6 Asynchronous Fetching

At run-time, the K-patch bintree is traversed by the LOD manager in the rendering thread, and LOD selection is performed on per K-patch basis based on their saturated view-dependent octagon error metric. The selected K-patch nodes activate their corresponding height-field data M-blocks. The system's render queue then renders the associated K-patches of all active M-blocks. The only resources which have to be actively managed in the sense of loading, caching and prefetching are the data intensive grid-digital height-field M-blocks and textures. The height-map data of currently activated M-blocks as well as their relevant texture is loaded and stored in GPU memory, and other recently used M-blocks are cached in main memory, also see Figure 4.5.

In interactive rendering, the viewing parameters generally vary smoothly over time and thus LOD changes between frames happen gradually and predictively. In case of unavailability of a new LOD node, and to avoid synchronous loading of its M-block data (and corresponding texture) from out-of-core, the LOD change

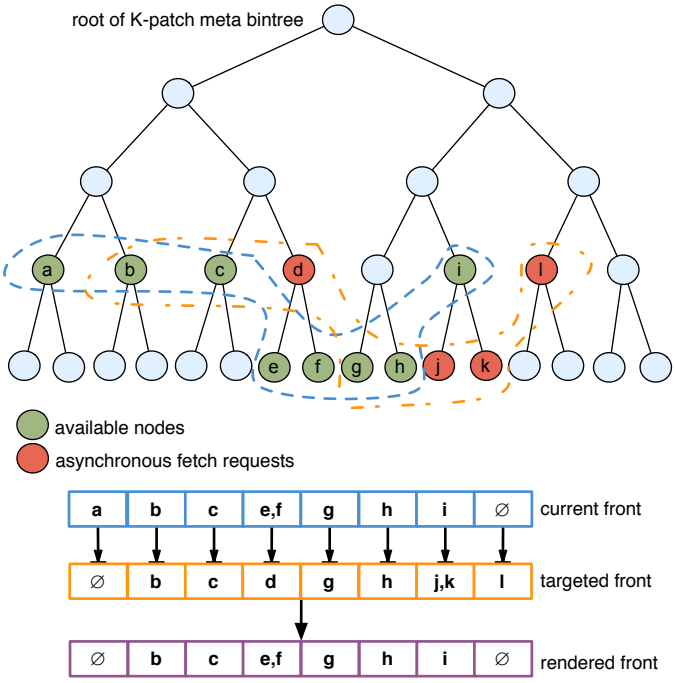


Figure 4.6: *Asynchronous fetching*

may be delayed to keep the frame rate constant until the new LOD data has been loaded from the disk. The requested higher or lower LOD details, which consist of (compressed) height-field M-blocks and their corresponding color textures, are processed asynchronously from a queue by separate threads. We refer to these LOD updates as asynchronous fetches.

To support the above strategy, we perform incremental frame-to-frame LOD updates as follows. The state of the currently selected and rendered LOD can be viewed as a *front* through the K-patch meta bintree, as indicated in Figure 4.6. A change in LOD then consists of an incremental update of that front, down- or upwards for LOD refinement or coarsening respectively. With respect to asserting the consistency of the LOD triangulation, incremental K-patch refinement or coarsening is only performed as long as the resolution of the neighboring patches can be matched without introducing cracks. Therefore, neighboring K-patch nodes are allowed to differ by at most one level.

Hence the actual new updated front constitutes of those nodes from the current and the targeted fronts which are already available in GPU or main memory as well as closest to the targeted front. If the M-block data for targeted nodes is not available, then an appropriate request is pushed onto the asynchronous load queue. Moreover, the corresponding nodes from the current front which can thus

not be replaced are retained in the render queue instead.

Whenever a K-patch causes an M-block to be pushed onto the load-requests queue, its texture and texture resolution are also considered and fetched if necessary. While uncompressed M-blocks can directly be loaded onto memory, compressed formats require on-the-fly decompression.

In case of a smooth user navigation, asynchronous fetches with delayed LOD display can satisfy almost any targeted updated with little latency. However, abrupt changes in navigation direction, e.g. through sharp rotations, cannot always be handled that way. Therefore, one might make use of the predictive nature of interactive navigation via spatial coherence and prefetch M-blocks that fall within an extended view frustum via the same asynchronous loading request queue.

Eventually, situations may still occur where no existing LOD data can directly be used without introducing cracks, holes and artifacts in the terrain display. In these rare cases a *synchronous fetch* must be executed which loads the required data directly for immediate rendering.

4.7 Results

Figure 4.7 shows the textured terrain for three different datasets. In Table 4.1, rendering statistics for various terrain datasets are listed. As one can observe, even though we perform on-the-fly decompression of height data on CPU, rendering performance is hardly affected when using compressed data.

(a) Zurich (DHM25 © swisstopo) (b) Ofenpass (© RSL Univ. of Zurich) (c) Puget Sound

Figure 4.7: *Example screenshots of interactive terrain rendering of different DEMs.*

4.8 Discussion

In this part of the thesis, we have demonstrated the added efficiency to the RASTeR engine by using asynchronous out-of-core fetching of both M-blocks and their textures with our *front-based* approach. The overhead caused by disk latency is further reduced by compressing the M-blocks using the reversible JPEG2000

Data Set	Resolution	Texture	Uncompressed		Compressed	
		GB	Fps	MTps	Fps	MTps
Ofenpass	4K x 3K	0.111	109	251	105	246
Zurich	2K x 2K	3.38	131	241	130	237
Puget Sound	4K x 4K	0.7328	113	249	109	238
Puget Sound	16K x 16k	0.6961	98	223	95	218

Table 4.1: *Rendering performance for pixel error* 2 *in frames (Fps) and million triangles per second (MTps). Texture size given after processing and compression, courtesy: [Bösch et al., 2009]*

compression. One big issue encountered while applying higher compression is the absence of exact reproducibility while decompressing the height data which creates cracks between the adjacent k-patches. As a future direction, one could investigate into better compression schemes that can additionally be efficient to decompress at runtime either on CPU or GPU.

CHAPTER 5

PARALLEL TERRAIN RENDERING

5.1 Motivation and Background

Parallel rendering of massive datasets has become popular in recent years. The demand of high quality visualization and continuing development of graphics hardware creates a natural inclination towards the use of parallel rendering approaches. The huge size of DEMs can create a challenge to visualize them on a single machine, even with the modern hardware. To given an example, Figure 5.1 shows SRTM dataset rendered with RASTeR with progressively reduced pixel error. Whereas rendering is pretty interactive for 6 pixel error, the machine can hardly render one frame per second for 1 pixel error. One can however, clearly notice the difference in quality between these images. In fact, at significantly higher pixel errors with larger terrains, some features are not even visibly represented. Parallel rendering, therefore, appears as a natural solution to render such huge datasets. This part of the thesis deals with efficient out-of-core parallel and scalable terrain rendering approach that allows interactive visualization of huge DEMs on multi-machine and multi-display platforms at high resolutions.

The presented parallel terrain renderer is built upon RASTeR [Bösch et al., 2009], which introduced the concept of a paired multiresolution tree structure where the multiresolution triangulation is independent of the DEM data. RASTeR offers benefits of continuous out-of-core LOD fetching through asynchronous server thread. Further, it keeps DEM height data in a format independent of the multiresolution triangulation thereby making it useful for use by other applica-

tions. The most important feature of RASTeR for our use is its ability to easily parallelize terrain rendering on multiple distributed machines.

The early fundamentals of parallel rendering have been laid down in [Crockett, 1997] and [Molnar et al., 1994]. Cluster-based parallel rendering has been commercialized for offline rendering for computer generated animated movies or special effects and for other special application domains. Rendering of realistic terrain images on massively parallel computer systems has initially been addressed in [Vezina and Robertson, 1991], [Agranov and Gotsman, 1995] and [Li et al., 1996]. However, these approaches are not capable of handling very large datasets at interactive frame rates exploiting current generation of GPU hardware. Recent works include [Johnson et al., 2006] which relies on shared resources from a community of users to view 3D data, [Yin et al., 2006] that focuses on rendering on a PC cluster and [Hu et al., 2007] that describes a remote visualization system for large-scale terrain rendering based on a parallel streaming pipeline architecture.

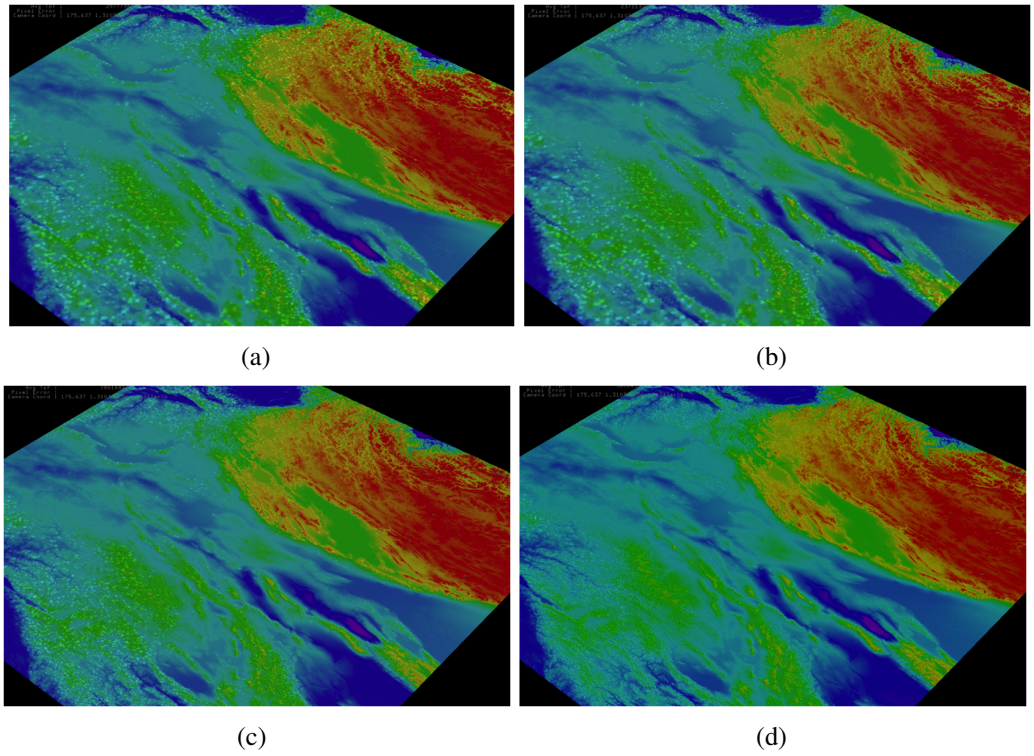
In this part of thesis, we specifically address the problem of rendering task decomposition in the screen (sort-first) or database (sort-last) domain. We also offer a comparative analysis of rendering performance for various rendering modes, configurations and data sizes.

5.2 Parallel Terrain Rendering

Our terrain rendering approach is parallelized and implemented using the Equalizer [Eilemann et al., 2009] framework for cluster-parallel rendering on top of existing terrain renderer RASTeR [Bösch et al., 2009]. Equalizer provides an API and library to facilitate the development of distributed as well as non-distributed parallel real-time rendering applications exploiting multiple GPUs. It is driven by a client-server approach in which the task decomposition and parallel rendering configurations to be executed are independent from the rendering client and entirely managed by the Equalizer server.

Task distribution is managed by the Equalizer server process according to the user specified configuration. In order to achieve distribution of the rendering task, the modified RASTeR terrain renderer has to take into account either of the two parameters supplied by Equalizer: a view frustum or database range for sort-first and sort-last rendering respectively. These parameters are passed to the application nodes by the Equalizer server. All other user parameters, such as pixel-error LOD threshold values and key or mouse controls are duplicated and broadcasted to all nodes.

Since Equalizer has its own OpenGL context handling mechanism and the original implementation of the terrain renderer uses multiple asynchronous threads,



(a) (b)

(c) (d)

Figure 5.1: *SRTM data set (32K × 32K vertices) rendered at progressively smaller pixel errors: (a) 6 (b) 4 (c) 2 (d) 1 pixel error respectively.*

for example to fetch textures and DEM data M-blocks into GPU memory, we have to provide each application thread the relevant OpenGL context using Equalizer's data and object distribution features. For more details on Equalizer and RASTeR, we refer the reader to [Eilemann et al., 2009] and [Bösch et al., 2009] respectively.

5.3 Sort-Last or Database Decomposition

Sort-last or database decomposition refers to the division of 3D geometry data among rendering machines, as indicated in Figure 5.2(a). In order to ensure optimal and scalable parallelization for best performance, it is important that any partitioning scheme ensures that:

1. the rendering primitives or rendering task is divided as equally as possible between the nodes,

2. the per-frame inter-communication traffic between the nodes is kept minimal.

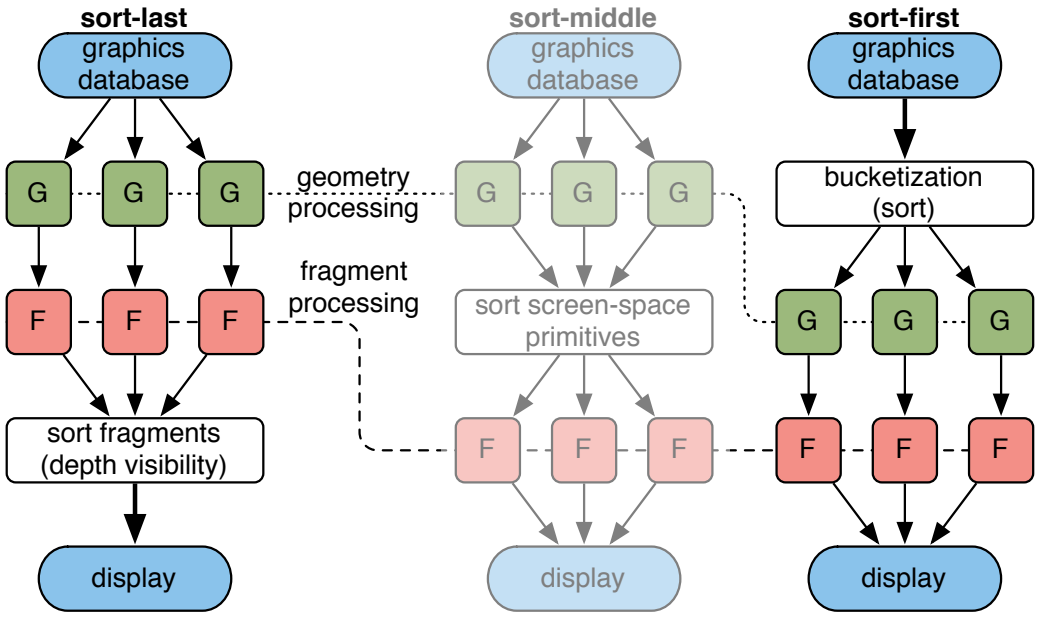


Figure 5.2: *Sort-last (a), sort-middle and sort-first (b) task decomposition and parallel rendering data flow.*

The first constraint is often non-trivial to achieve as in LOD based visualizations, a simple spatial range based decomposition of 3D data fails to distribute the rendering primitives equally among the different machines. Even more, this has to be done keeping in mind the overhead created by synchronization and communication traffic between the rendering machines.

For sort-last rendering, our terrain renderer decomposes the rendering task by dividing the terrain data into N parts, among the N rendering machines. For this, the equalizer divides the linear range of $[0, 1]$ into N equal parts. Therefore, the i^{th} machine receives a range $R_i = [\frac{i}{N}, \frac{i+1}{N}]$. The task before each of the terrain renderers is now to select and draw almost $\frac{1}{N}^{th}$ of the visible terrain data. Using a naive approach, this can be accomplished by enumerating all multiresolution triangles and then redistributing them over all the machines. Since this involves excessive per-primitive evaluations by the CPU, this would lead to starvation of the GPU.

As discussed in the last chapter, the bases of RASTeR are triangle K-Patches and terrain data M-Blocks. At rendering time, the LOD manager selects all the K-patches within the given LOD error value for rendering. The triangle K-patches in

turn activate their corresponding M-blocks which are the data units containing the height and normal values of the terrain. Using these units as a basis for sort-last rendering we discuss three decomposition approaches, each improving upon the other.

5.3.1 Linear Block Enumeration

A simple way to achieve database domain decomposition is to enumerate the terrain M-blocks and assign them equally to the participating rendering machines. All machines traverse the meta bintree in parallel and identify the LOD K-patches to be displayed. Thereafter, each machine checks if the selected K-patches correspond to an M-block within its own range R_i and only this filtered set of K-patches is then rendered. For example, if the range of the current machine as supplied by Equalizer is $R_i = [l, r]$, the origin of an M-block is given by $O_M(x, y)$ and x_{max} the maximum x-dimension value, then the check $l * x_{\mathrm{max}} \leq O_{\mathrm{M}}(x) \leq r * x_{\mathrm{max}}$ can be made to decide if a K-patch should be rendered on this node or not, see also Figure 5.3(a) In fact, as soon as a K-patch fails this test, the traversal of the meta bintree can be stopped as the child nodes do not fall in the given range either.

Though we can exploit the K-patches as triangle cluster units instead of per primitive LOD querying, this approach has several drawbacks. Firstly, the meta bintree LOD traversal has to be constrained to the branches corresponding to the M-blocks indicated by R_i and therefore, LOD selection is easily susceptible to view changes upon rotation or translation. Moreover, this approach cannot guarantee an even division of data across all machines (Figure 5.4(a)) and therefore is not the best for parallel rendering.

5.3.2 Quadtree Enumeration

An improved approach makes use of the quadtree structure of the M-block hierarchy, see also Figure 5.3(b). Starting from the root M-block, all quadtree nodes are recursively enumerated. Bottom-up, intervals to all internal nodes are assigned that cover the range of its descendants. At runtime, all bintree K-patches that correspond to M-blocks in the range $R_i = [l, r]$ supplied by the Equalizer are selected for rendering on a particular machine. The range test is simple and can be made as follows:

$$\begin{aligned} l * n_{max} &\leq L_{\mathrm{M}} \leq r * n_{max} \vee \\ l * n_{max} &\leq R_{\mathrm{M}} \leq r * n_{max} \end{aligned}$$

where n_{max} refers to the maximum number of a leaf node and $[L, R]_M$ is the interval of the M-block node itself covering its descendants. This decomposition

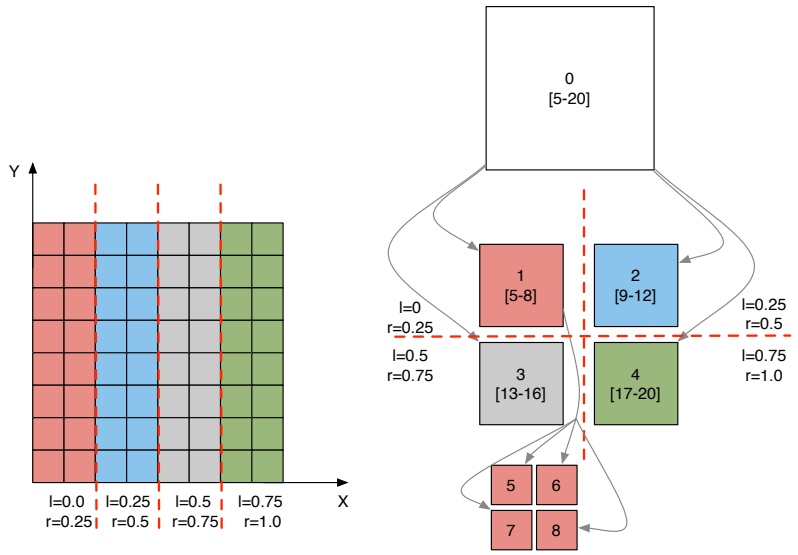
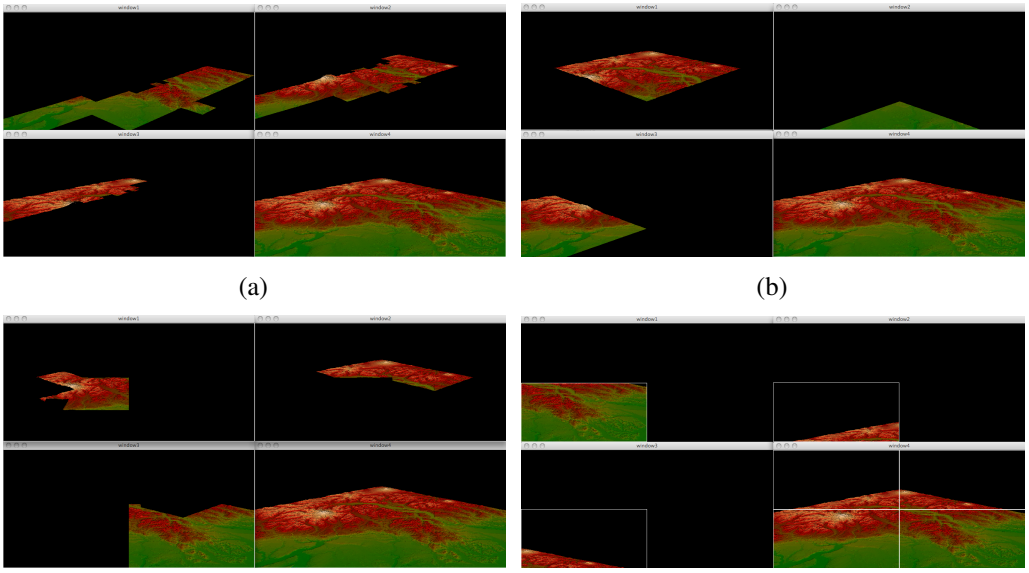


Figure 5.3: *Sort-last database decomposition of terrain for four nodes using (a) linear block and (b) quadtree enumeration.*

strategy allows us to select coherent terrain data per machine that is not as susceptible to rotational and translational changes as in the previous approach. Figure 5.4(b) clearly demonstrates the advantage of this improved decomposition mode over a simple linear enumeration.

5.3.3 Active K-Patch Enumeration

The problem with both of the above decomposition schemes is that they do not ensure that all rendering machines get a similar amount of rendering workload and hence optimal performance is not reached. Since our basic rendering units are triangle K-patches and M-blocks, any simple spatial division of these units cannot guarantee that the rendering load is evenly distributed across all machines in terms of the number of drawing primitives. This can however, be achieved by making the observation that each K-patch contains the same number of triangles. Therefore, an optimal task distribution can be achieved by dividing the list of visible K-patches equally among all rendering nodes as illustrated in Figure 5.5. After the meta bintree LOD traversal, the front of visible K-patches is the same across all machines. Each machine can choose from this list a particular subset of K-patches to render. Thus, the front of selected K-patches is enumerated and mapped to the ranges R_i provided by Equalizer. Using this view-adaptive assignment of visible K-patches, each node can select a similar number of geometric primitives



(a) (b)

(c) (d)

Figure 5.4: *Screenshots of sort-last database decomposition of terrain on four nodes using (a) linear block, (b) quadtree and (c) active K-Patch enumeration. (d) Sort-first view frustum decomposition.*

to display without the need for communication overhead, see also Figure 5.4(c) for a run-time view using this decomposition.

5.4 Sort-First or Screen Decomposition

The sort-first decomposition mode involves task division in screen space and is relatively simple (Figure 5.4(d)). For each frame, before the LOD meta bintree is traversed, each rendering machine updates its view frustum parameters to the ones indicated by the Equalizer server. The meta bintree traversal is then restricted to the particular view frustum, performing view-frustum culling of the LOD meta bintree on that node. Since in sort-first mode different machines render different parts of terrain that occupy mutually separate parts on screen, final image assembly is simple and fast as it does not involve any costly z-depth or α-compositing stage.

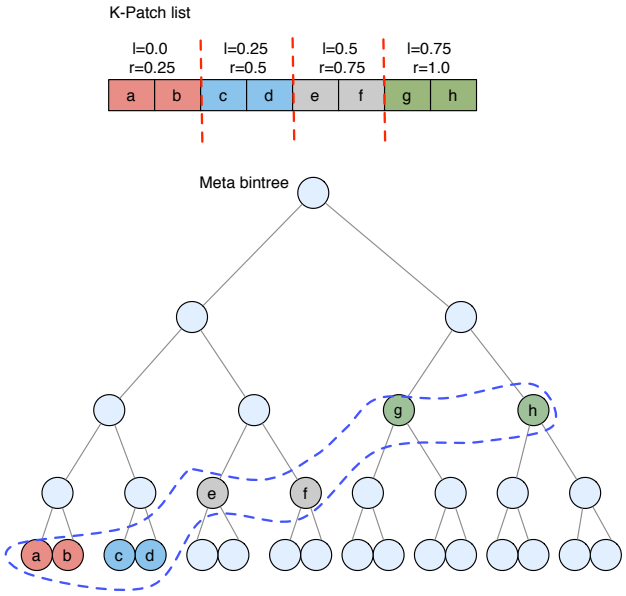


Figure 5.5: *Sort-last database decomposition of terrain for four nodes using active K-patch enumeration.*

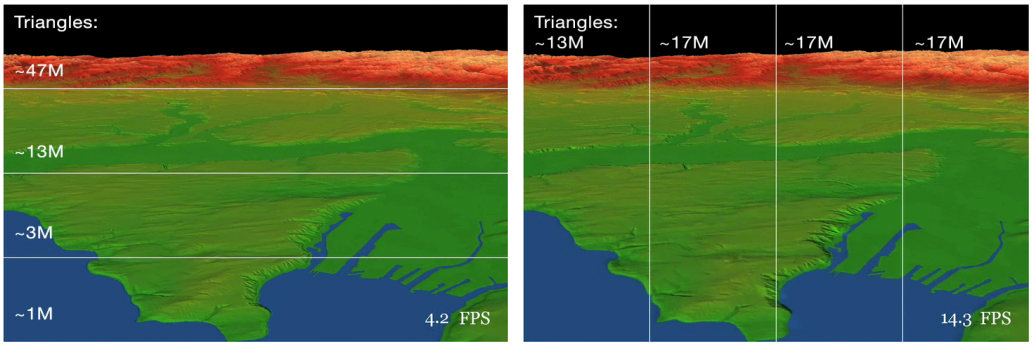
5.5 Results

Equalizer and RASTeR both are written in C++ using GLSL shaders and the implementation is tested on a 10-node Linux cluster with 2 Gbit/s Myrinet for image compositing and 1 Gbit/s network for out-of-core terrain data retrieval. Each node features a dual 2.2 GHz AMD Opteron CPU, 4GB of RAM and GeForce 9800 GX2 graphics. Two different datasets were used : Puget Sound (16k X 16k vertices) and SRTM (32k X 32k vertices) on a 1280 X 1024 pixel viewport.

Our tests have shown that linear block and quadtree enumeration do not provide scalable sort-last rendering. Only our third approach using active front K-patch enumeration showed performance improvements when adding more rendering machines. This method provides simple and automatic load balancing (since all machines render almost equal number of triangles) which is not based on past rendering times, while other approaches require more sophisticated load balancing computations.

For sort-first parallel rendering we have analyzed two simple screen decomposition modes, vertical (Figure 5.6(a)) and horizontal tiling (Figure 5.6(b)), that partition the view frustum equally. Horizontal partitioning leads to a much more even distribution of geometry per screen tile as compared to vertical partitioning as demonstrated.

Figures 5.7(a) and 5.7(c) present frame rate graphs for moving forward and



(a) (b)

Figure 5.6: *Sort-first screen partitioning causes a less even distribution of data to be rendered in the case of vertical (a) compared to horizontal (b) partitioning. (four nodes)*

turning camera trajectories, while Figures 5.7(b) and 5.7(d) present those for the camera zooming into the terrain. As we can see from the graphs, in both sort-first (2D) and sort-last (DB) parallel rendering modes pure drawing performance scales at least linearly. Pure sort-last rendering scales better than sort-first because each machine renders a similar number of triangles, thus data is well distributed among them.

Superlinear performance of rendering can be explained by reduced data fetching since each machine fetches only those terrain M-blocks that are not already cached locally in main memory. Further, the reduced size of rendering front of active K-patches on a single machine allows it to be cached more efficiently in GPU and main memory, hence avoiding repeated data fetching.

Overall rendering performance depends largely on the compositing stage of the parallel rendering framework, which includes reading of partial images back from GPUs, transmitting them to the destination node and assembling final frames for display. The decrease of the final performance with increasing number of nodes in Figure 5.7 happens due to the image throughput bottleneck. The amount of data that has to be sent over the network in case of sort-last rendering and compositing is roughly twice larger than for sort-first, thus network saturation happens earlier despite the rendering itself being faster. In our case, sort-last network saturation happens at around 15 fps, which is independent of the drawing speed. That means if the initial rendering on one node is already fast, overall performance will not scale well with more rendering nodes. For the smaller Puget Sound terrain model, when the camera is zooming in and the initial speed is about 3 fps, sort-last rendering does not scale anymore after 6 nodes. For sort-first rendering, network saturation should happen at around 30 fps. This speed is not

reached in our experiments, therefore, the final performance of sort-first rendering scales almost linearly up to the tested number of nodes.

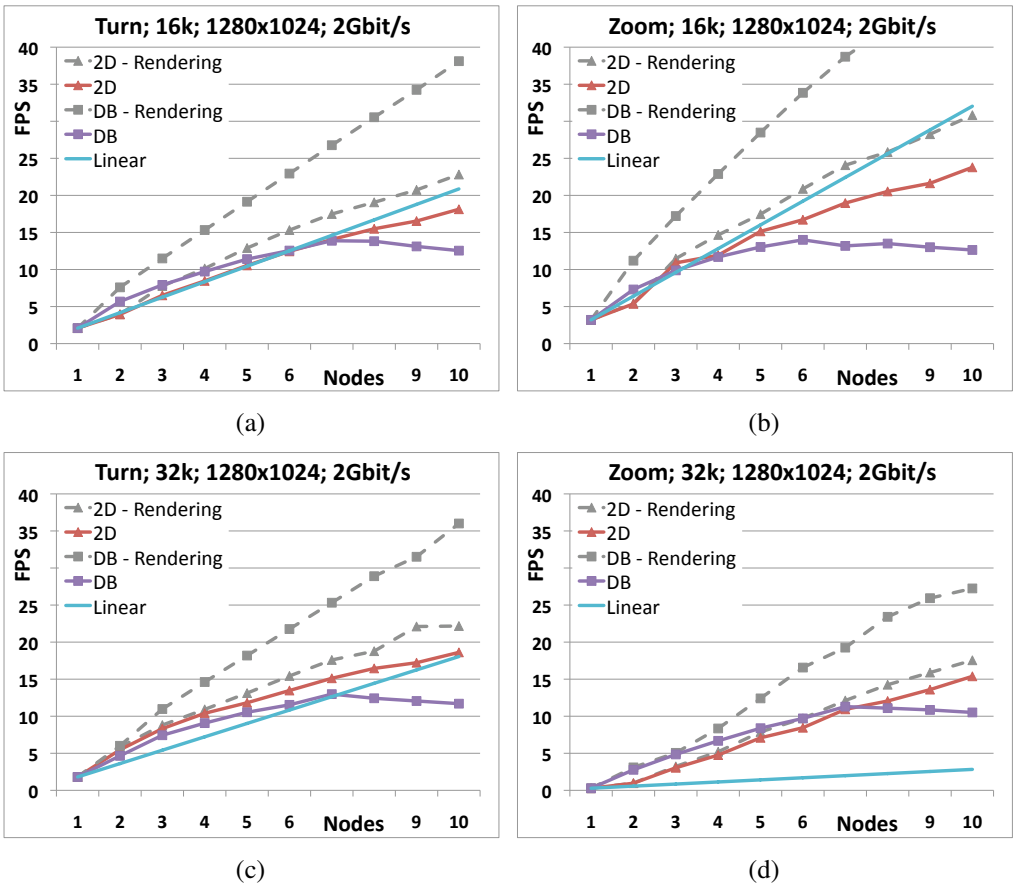


Figure 5.7: *Graphs showing rendering performance on 10 machines in parallel using DEM models of (a),(b) 16k×16k Puget Sound and (c),(d) 32k×32k SRTM grids with camera in turning and zooming trajectories respectively. 2D - Rendering refers to sort-first rendering, 2D to sort-first rendering with compositing, DB - Rendering to sort-last rendering, DB to sort-last rendering with compositing.*

5.6 Discussion

In this work, we have identified and addressed challenges encountered when dealing with scalable cluster-parallel out-of-core multiresolution terrain rendering. We have shown that scalable parallel rendering cannot be easily achieved by simple sort-last or sort-first task distribution, even if applied to adaptive LOD terrain rendering approaches. To achieve effective parallel rendering using distributed

graphics hardware resources over a network, more aspects on task assignment and out-of-core loading have to be taken into account.

Further, we have introduced a novel sort-last data decomposition technique that achieves per-frame automatic load balancing. While this method realizes highly scalable, multiresolution terrain rendering from out-of-core for very large grid-digital elevation models, the introduced technique and analysis only presents the first step towards efficient cluster-parallel terrain rendering.

Part II

Parallel and LOD-based Particle Simulation

CHAPTER 6

SMOOTHED PARTICLE HYDRODYNAMICS

6.1 Background

Physically-based fluid simulations have a wide range of applicability in several important domains such as medicine, science, engineering and entertainment. Fluid simulation methods can be broadly divided into two main categories: Eulerian and Lagrangian. Whereas Eulerian methods compute fluid attributes at fixed locations in the grid, in Lagrangian methods particles are the attribute carriers and information moves with the particles. Particle-based Lagrangian methods, such as the Smoothed Particle Hydrodynamics (SPH) approach, are particularly attractive for fluid simulation due to their capability to generate small scale detailed fluid motion effects and to handle complex simulation domain boundaries. The high quality of rendering obtained with SPH is demonstrated by Figure 6.1 using three different recent popular approaches.

6.2 SPH Basics

In this section, we briefly introduce the fundamentals of SPH as laid out in [Monaghan, 1992], [Monaghan, 2005] and [Müller et al., 2003]. In SPH, a scalar quantity A is interpolated at location $\mathbf{r}$ by a weighted sum of contributions from nearby particles as given by Equation 6.1. Here m_j refers to the mass of particle j, ρ_j is

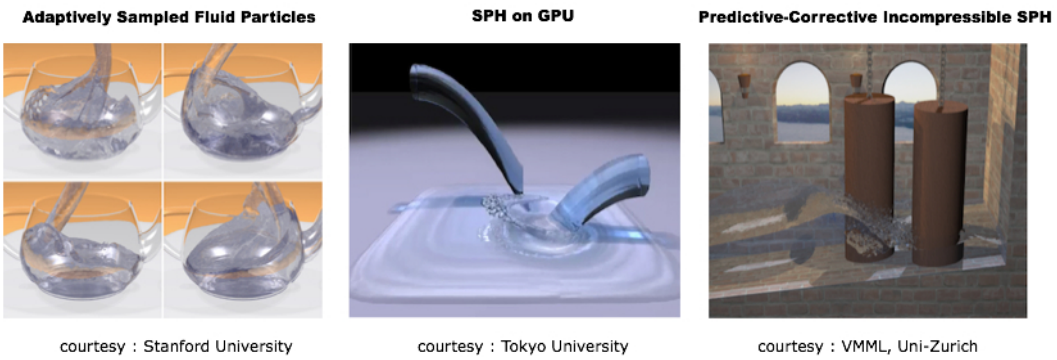


Figure 6.1: *Recent high performing SPH fluid solvers (a) Adaptively Sampled Fluid Particles, (b) SPH on GPU, (c) Predictive-Corrective Incompressible SPH.*

its density, $\mathbf{r}$ its position and $W(\mathbf{r}, h)$ is the smoothing kernel with global support radius h. The gradient of A is obtained by replacing W by its gradient (Equation 6.2) and a similar formulation exists for the Laplacian (Equation 6.3).

$$A_i = \sum_j A_j \frac{m_j}{\rho_j} W(\mathbf{r}_i - \mathbf{r}_j, h) \tag{6.1}$$

$$\nabla A_i = \sum_j A_j \frac{m_j}{\rho_j} \nabla W(\mathbf{r}_i - \mathbf{r}_j, h) \tag{6.2}$$

$$\nabla^2 A_i = \sum_j A_j \frac{m_j}{\rho_j} \nabla^2 W(\mathbf{r}_i - \mathbf{r}_j, h) \tag{6.3}$$

We can define the density by replacing A in Equation 6.1 by ρ, leading to Equation 6.4. This density is used to compute pressure which in turn is substituted in Equation 6.7 to give pressure force. Pressure can be derived from the equation of state (EOS) according to [Batchelor, 1967] by Equation 6.5 where k is the stiffness constant. When $\gamma = 1$, Equation 6.5 corresponds to the pressure formulation in [Desbrun and Cani, 1996], given by Equation 6.6.

$$\rho_i = \sum_j m_j W(\mathbf{r}_i - \mathbf{r}_j, h) \tag{6.4}$$

$$p_i = k \frac{\rho_0}{\gamma} ((\frac{\rho_i}{\rho_0})^\gamma - 1) \tag{6.5}$$

$$p_i = k(\rho_i - \rho_0) \tag{6.6}$$

$$\mathbf{f}_i^{pressure} = \sum_j m_i m_j (\frac{p_i}{\rho_i^2} + \frac{p_j}{\rho_j^2}) \nabla W(\mathbf{r}_i - \mathbf{r}_j, h) \tag{6.7}$$

In addition to pressure forces, viscosity (Equation 6.8) and gravity forces are also applied to particles. The total force on a particle is obtained by summing up individual contributions from all these forces (Equation 6.9). The basic steps in the SPH simulation loop are given in Algorithm 6.

The surface of the fluid can be defined by Equation 6.10, by specifying the threshold on the magnitude of $\mathbf{n}_i$, where $\mathbf{n}_i$ is the gradient of a color field, and extracting all particles above the threshold [Müller et al., 2003]. For more in-depth survey on SPH formulae and techniques, we refer the reader to [Müller et al., 2003], [Monaghan, 1992] and [Monaghan, 2005]. Unless otherwise specified, throughout our work we have used the smoothing kernels specified in [Müller et al., 2003].

$$\mathbf{f}_i^{viscosity} = \mu \sum_j m_j \frac{v_j - v_i}{\rho_j} \nabla^2 W(\mathbf{r}_i - \mathbf{r}_j, h) \tag{6.8}$$

$$\mathbf{f}_i^{total} = \mathbf{f}_i^{pressure} + \mathbf{f}_i^{viscosity} + \mathbf{f}_i^{gravity} \tag{6.9}$$

$$\mathbf{n}_i = \sum_j \frac{m_j}{\rho_j} \nabla W(\mathbf{p}_i - \mathbf{p}_j, h) \tag{6.10}$$

Algorithm 6 SPH Algorithm

```
while animating do
  for all particles i do
    find neighborhood N_i(t)
  for all particles i do
    compute density ρ_i(t)
    compute pressure p_i(t)
  for all particles i do
    compute forces F^{p,g,v}(t)
  for all particles i do
    compute new velocity v_i(t+1)
    compute new position p_i(t+1)
```

6.3 Challenges

For realistic fluid visualization, a high particle density is required, in particular to resolve fine-scale surface details. Although interactive frame rates can be achieved for a few thousands of particles, accelerating SPH solvers for larger particle counts remains a challenging task. After the introduction of SPH in computer graphics field by [Desbrun and Cani, 1996], [Müller et al., 2003], several methods have

been proposed to accelerate particle simulation both for real-time and offline purposes. Similar to terrain and point-based rendering, most of these approaches attempt to accelerate SPH using either LOD or parallel techniques. In contrast to aforementioned fields, the nature of LOD-based solutions in particle simulation comes with additional set of challenges. The most important of them concerns stability. This is because methods like SPH are highly sensitive to change in attributes which can even lead to instability.

Techniques parallelizing SPH target both GPUs and multi-processor environments. One of the most expensive operations in the SPH animation loop is the neighborhood computation where the neighbors to each particle are determined. An optimal parallel implementation demands the porting of entire SPH animation loop, including neighborhood search, on the parallel hardware.

6.4 Overview

Several efforts have been made to accelerate particle simulation. Some of these works leverage the parallel computational power of GPU. On the other hand, simulation with adaptive particle sizes has also been proposed as a means for LOD-based computation. In this thesis, we discuss our parallel and LOD contributions in the field of SPH. This part of the thesis focuses on our two-fold contribution in the field of particle simulation:

1. Efficient GPU-based SPH employing space- and time-efficient *Z-indexing*
2. SPH acceleration through approximation

The first chapter discusses another approach to speed-up SPH computation by parallelizing it on CUDA using the novel Morton-based Z-indexing for neighborhood search. In the second chapter, we introduce our heuristic to employ larger time-steps than deduced from CFL condition. Further, the SPH itself is approximated by leveraging the inactivity zones.

CHAPTER 7

PARALLEL PARTICLE SIMULATION

7.1 Fluid simulation

The particle-based fluid solver SPH, needs a large number of particles to achieve smooth surfaces and to resolve fine-scale surface details. Real-time constraints, however, required in the past low fluid resolutions resulting in poor physical and visual results. To accelerate the simulation, enabling real-time simulation with a higher particle resolution, our method implements the SPH fluid solver and particle rendering on the GPU. Although executing the SPH physics on the GPU accelerates the simulation compared to a CPU implementation, [Amada et al., 2004], [Harada et al., 2007b] and [Zhang et al., 2008], previous solutions come with a number of limitations. The main disadvantage is that the common grid based approach overestimates the memory consumption per grid cell a priori, thus excess use of GPU memory cannot be avoided. Furthermore, these approaches are highly constrained in their choice and usage of problem attributes with respect to functionalities allowed by shader languages.

In the last one decade, several approaches have been proposed towards performance improvement in SPH. While [Adams et al., 2007] use adaptive particle sizes, [Solenthaler and Pajarola, 2009] accelerate fluid simulation by enforcing incompressibility with prediction-correction. Both these approaches are completely CPU based. In [Amada et al., 2004], neighbor search is computed on the CPU while the standard SPH physics computation is done on the GPU. [Harada et al., 2007b], [Zhang et al., 2008] execute all steps of the computation on the GPU in

which a grid-based structure is used to simplify the shader based neighbor search.

[Zhou et al., 2008] present a method to construct a real-time kd-tree on the GPU. Not only the kd-tree construction can create bottleneck in the whole SPH loop, it is also well known that hierarchical data structures are not the best for SPH like computations [Harada et al., 2007a]. On the other hand, the sliced data structure proposed in [Harada et al., 2007a] might consume as much memory as the grid volume itself. Further, it might make direct mapping of SPH onto CUDA more complicated. Though the simulation of simple particle interactions has been done on CUDA [NVIDIA, 2009] using a uniform grid structure with an upper bound of 8 neighboring particles, there is no other work besides [Harada et al., 2007b] and [Zhang et al., 2008] leveraging its huge computational power for SPH simulation.

To cope with these issues, we present a novel CUDA based parallel SPH implementation in this thesis. The approach relies only on basic CUDA structures like textures and arrays and hence is very flexible and generic and can therefore accommodate any extra attributes. Spatial indexing and search is based on *Z-indexing* that eliminates the use of buckets and allows to determine the neighborhood set of a particle in near constant time without wasting space. All other computations, which include sorting particles, are done on the GPU avoiding any CPU-GPU transfer overhead. As a result, the approach produces more efficient results for a similar particle count than state-of-the-art real-time SPH simulation methods. Also our solution can be used for offline SPH simulation of larger particle counts than existing GPU based methods. Alternately, the available free graphics memory can be used for visualization purposes together with simulation, see also [Goswami et al., 2010b] for more details on this.

7.2 Z-Indexing

Our approach introduces an efficient *Z-indexing* [Morton, 1966] in the context of range queries which enables obtaining a neighborhood set for a particle without any space overhead. The simulation domain is divided into a virtual indexing grid in X, Y, Z along each of the dimensions and the grid location of a particle is used to determine its bit-interleaved Z-index, see also Figure 7.1. The Z-index can be computed very efficiently using a table lookup. All particles lying within any power-of-two aligned block have contiguous Z-indices.

For range queries given a radius R, the global support radius of the SPH simulation, we determine the nearest power-of-two block size S in the indexing grid domain. The starting Z-index s of any block of size S can easily be determined and particles falling into that block form a sequence between s and $s + S^3$. At the start of each time step, the Z-indices of all particles are calculated in paral-

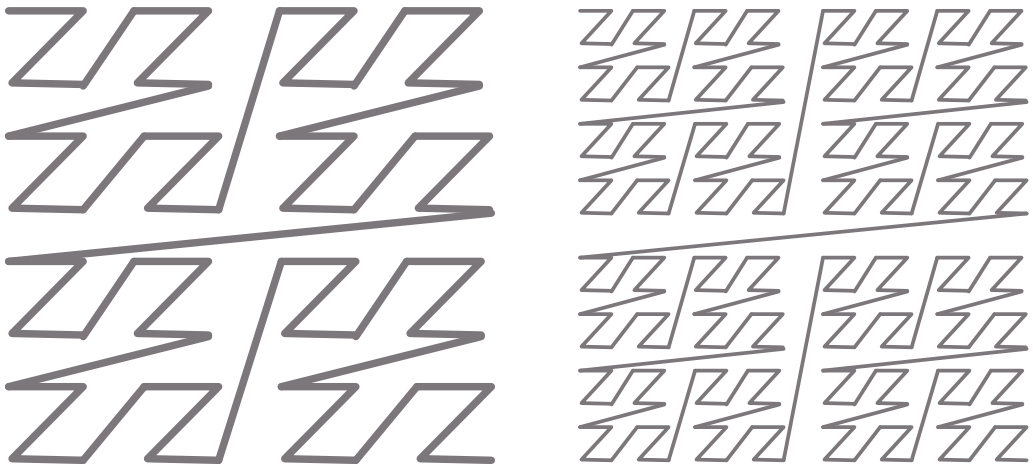


Figure 7.1: *Z-indices of particles falling within an aligned block of some power of 2 are contiguous and can be constructed using bit-interleaving from grid locations.*

lel following which the particles are sorted using parallel radix-sort in CUDA on their Z-index [Le Grand, 2007]. Hence for each block we just need to determine the index of its first particle and the number of particles it contains. This is accomplished by launching as many CUDA threads as there are number of particles wherein each particle determines its block. Whereas the first particle in a block can be determined using the $atomicMin$ operation in CUDA, the number of particles is found by incrementing particle count in it with $atomicInc$. Thus each particle updates both the starting index and particle count of its block in the list B, which is of size $|B| = (\frac{X_{max}}{S})^3$ (assuming a simulation domain grid dimension X_{max}).

7.3 Neighbor Search

For subsequent steps, we have the information of the starting index of every block in B and the number of particles in it. Populated blocks are split if holding more than some N particles and compacted to a set of non-empty CUDA blocks B' in parallel also using $atomicMin$ and $atomicInc$, see also Figure 7.2. Instead of directly launching a pre-decided number of threads, we launch only as many CUDA kernel blocks $|B'|$ as necessary. Each of these blocks has at most N CUDA threads which is the maximum number of particles per block. Each block is responsible to copy particles iteratively from one of its 27 neighboring blocks in B (which includes self block) into the shared memory of its own CUDA block in $|B'|$ as illustrated in Figure 7.3. Alternating with this copy process, each CUDA

thread computes the physical attributes for one particle in its block in B', see also the algorithm in Algorithm 7.

Many blocks may contain fewer than N particles but still run N threads. However, a thread with thread-id tid will not be completely idle as long as the particle count in the current block is equal to or greater than tid, or $tid < 27$ in which case the thread has to copy particles from neighboring blocks.

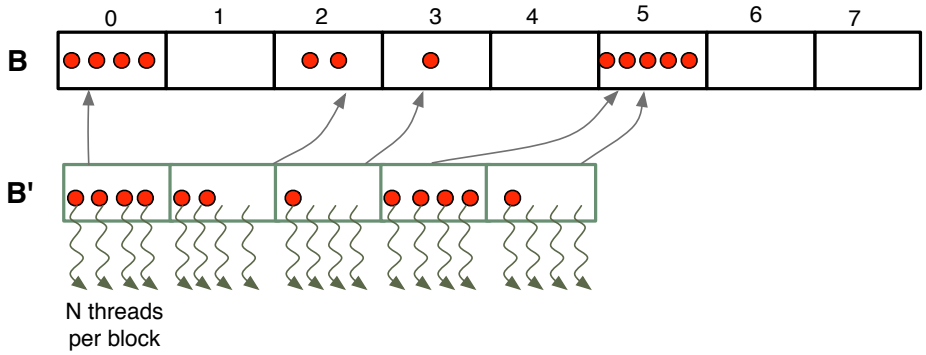


Figure 7.2: *For each non-empty block in B, a CUDA block is generated in B' and launched with N threads (N = 4 here).*

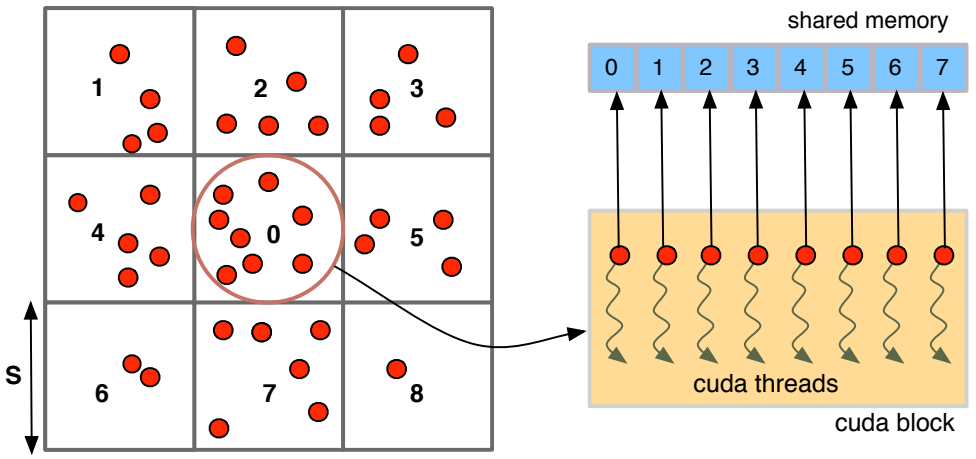


Figure 7.3: *Each CUDA thread in a block computes attributes for one particle and at the same time copies particles from a neighboring block into its shared memory.*

Since in order to keep track of a block, one just needs two attributes and therefore the space required to maintain all blocks is only $2 \cdot B$ for $|B|$ grid blocks in the simulation domain. Note that blocks refer to their particles only by index, using the starting particle and number of particles in a block and the particles themselves are maintained in a CUDA attributes array. Moreover, block computations can be

carried out within the memory allocated for particle attributes and we do not need any extra memory allocated for it.

7.4 Physics Computation

The next step after neighborhood determination is to compute densities of all particles. For an overview of basics on SPH, please see Chapter 6. Since our simulation domain is divided into blocks of size equal to or greater than h, the neighbors to any particle in a block for density or force computation, lie no farther than the particles in its immediate neighbor blocks, see also Figure 7.3. Each of the N threads in a CUDA kernel block copies one particle at a time from a neighboring block to its shared memory and at the same time computes physical attributes for one particle in the current block.

In the first step, every particle in the current block updates its density by determining and copying neighboring particles into its shared memory as discussed above. The computed densities are then made available as CUDA textures for force computation. This is done to allow efficient access even with random access pattern from global memory. Each particle then repeats the same procedure as above for accessing the neighbors and their new densities to compute the pressure forces using Equation 6.7 where p_i is the pressure and $\nabla W(\mathbf{r}_i - \mathbf{r}_j, h)$ is the gradient of the smoothing kernel. This is followed by time-integration wherein each particles updates its velocity and position and new Z-index calculation. In a CUDA kernel block, each thread also computes attributes like inverse density square once and stores them in the shared memory. This way we optimize by reducing expensive operations like division, since these values are used by multiple threads in the block. Finally, each thread writes the computed attributes for its particle to the global memory.

It is important to note that density and force computations cannot be clubbed together in a single CUDA procedure as updated densities are required to compute force values. This necessitates launching separate CUDA kernels for each of them thereby forcing fresh neighbor finding and copying into the shared memory separately in each procedure. However, in our case neighbor finding is inexpensive and does not hurt the overall performance.

7.5 CUDA Computation

Algorithm 7 outlines the complete structure of our CUDA SPH algorithm. All the global look-up tables for SPH kernel computation are generated beforehand and kept in constant memory. We avoid bit operations in Z-indexing by computing interleaved bitwise representations of all possible grid values along any x, y and

Algorithm 7 SPH Physics on CUDA

Copy all particles from CPU to GPU memory
for all frames **do**
  /*—— Z-index and Sorting ——*/
  Calculate Z-indices for particles
  Sort them using radix-sort
  Copy sorted particles in the ping pong array
  Make CUDA texture of sorted particle positions

  /*—— Block Generation ——*/
  Create blocks from sorted particles by determining
  for all blocks (using positions from texture) **do**
    – Starting index or index of first particle in array
    – Number of particles in it
  B_0 : Number of blocks identified
  Split all blocks containing more than N particles each
  B' : Number of compacted blocks after splitting

  /*—— Density Computation ——*/
  Launch B' CUDA kernels with N threads each
  for all cuda blocks **do**
    Determine M: max particles in neighbors
    N: particles in current block
    Copy its own N particles into its shared memory
    for $i = 1 \rightarrow M$ **do**
      – Copy a particle from neighboring blocks 0 to 26 (one per thread) to its own shared memory
      – syncthreads()
      for $j = 1 \rightarrow N$ **do**
        Compute new densities from new copied neighbors in shared memory
    for $j = 1 \rightarrow N$ **do**
      – Write updated densities to global memory
    Make CUDA texture of newly computed densities

  /*—— Force Computation ——*/
  Launch B' CUDA kernels with N threads each
  for all CUDA blocks **do**
    Determine M: max particles in neighbors
    N: particles in current block
    Copy its own N particles into its shared memory
    for $i = 1 \rightarrow M$ **do**
      – Copy a particle from neighboring blocks 0 to 26 (one per thread) to its own shared memory
      – syncthreads()
      for $j = 1 \rightarrow N$ **do**
        Compute new forces using texture densities and neighbors copied in shared memory
    for all $j = 1 \rightarrow N$ **do**
      – Handle collisions and boundary forces
      – Update particle positions
      – Write updated positions to global memory

z dimension and storing them as CUDA texture. The Z-index value for a given position ($\mathbf{r}_x$, $\mathbf{r}_y$, $\mathbf{r}_z$) on the grid can be obtained by bitwise OR of these texture look-ups.

Our implementation requires four ping-pong arrays: first for radix-sort, second for position and Z-index, third for velocity and pressure and fourth for density. Since it could be the case that global memory accesses are not coalesced, we obtain the old attribute values from CUDA texture arrays while we write updated values into the ping-pong arrays. These two sets of CUDA arrays reverse their role every frame as readable textures and writable global memory. Since force values are not required outside the force kernel, we do not need any global memory for them. Also, we avoid allocating separate memory for block computations by doing it in one of the same CUDA array as we use for radix-sort. Once updated particle positions are copied to the position array, the radix-sort array is free and can be used for block computations. Hence, no extra space is in fact needed for blocks maintenance.

7.6 Results

Our simulation was implemented and tested using OpenGL, GLSL and CUDA 2.3 on two different platforms:

1. MAC OS X 10.5.8, 2 X 2.66 GHz Dual-Core Intel Xeon and NVIDIA GeForce 8800 GT with 512 MB VRAM

2. Linux, 2.93 GHz Core i7 and NVIDIA GeForce FTX 280 with 1 GB VRAM

The virtual grid size for Z-indexing is kept at the same fine resolution of 1024^3 throughout the experiments. In order to represent a particle's grid position, we need just 10 bits. Therefore, the entire Z-index can be packed into a single 32-bit integer. Our experiments suggest that CUDA blocks with dimension 32 or 64 are more performant since there are lesser threads sitting idle as compared to higher dimensions. Also since the block size S is dependent on global support radius, it is important to choose parameters such that the difference between projection of global support radius on grid and its closest power of 2 is minimal.

For testing the SPH simulation and rendering we have used two different setups. The first scene consists of a water column collapsing due to gravity and the second of a water-jet filling a basin. Whereas the number of particles is chosen initially and remains constant with time in the former, it changes during the simulation (limited by an upper bound) in the latter.

As can be seen from Table 7.1, our CUDA based SPH implementation achieves excellent simulation rates. Platform 1 demonstrates improved simulation performance in comparison to [Harada et al., 2007b] on a slower GPU. While for

the simulation part [Harada et al., 2007b] report 17fps for 60K particles and 1fps for 1M particles on an NVIDIA 8800 GTX, we achieve 16fps for 75K and 1.25fps for 1M particles on the much slower NVIDIA 8800 GT. On the faster 8800 GTX, [Zhang et al., 2008] report faster times that [Harada et al., 2007b] although not only at the expense of extensive memory consumption but also at a loss of quality and accuracy since the density computation is combined with that of pressure force, which is then updated with a delay of one frame. This kind of combination might boost the performance but is known to cause stability problems [Solenthaler and Pajarola, 2008]. Further in comparison to NVIDIA 8800 GTX, 8800 GT has 30% less memory bandwidth and 12.5% less CUDA cores. In addition, considering the combined density and pressure force computation as in [Zhang et al., 2008], we estimate similar physics computation speed even on a slower hardware. Further, in comparison to a normal CPU based SPH solver on the same processor, our CUDA based simulation is about 9 times faster.

Particle	Platform 1	Platform 2
16'128	69fps	123
75'200	16fps	26
129'024	9fps	17
255'600	5fps	10

Table 7.1: *Simulation performance results for a collapsing water column on Platform 1 and 2 respectively*

An efficient real-time visualization framework has been introduced by Schlegel et. al in [Goswami et al., 2010b] which employs volume rendering with distance field generation. However, the images can be generated offline using POVRAY as demonstrated in Figure 7.4 by using the particle position from simulation.

7.7 Discussion

We have presented a novel GPU accelerated parallel SPH simulation that achieves interactive simulation for particle counts of nearly a quarter million on current consumer graphics hardware. Our Z-index based neighbor search approach on CUDA is computationally as well as memory efficient. The spatial indexing and search method is flexible and generic, and as well can be used for other purposes like surface particle extraction for visualization. Our parallel SPH simulation outperforms prior state-of-the-art solutions in terms of performance plus accuracy metric. As a future direction, one could integrate adaptive particle sizing together with morton-based Z-indexing to achieve even faster results. The challenge there would be maintaining multi-level grid for different global support radius.

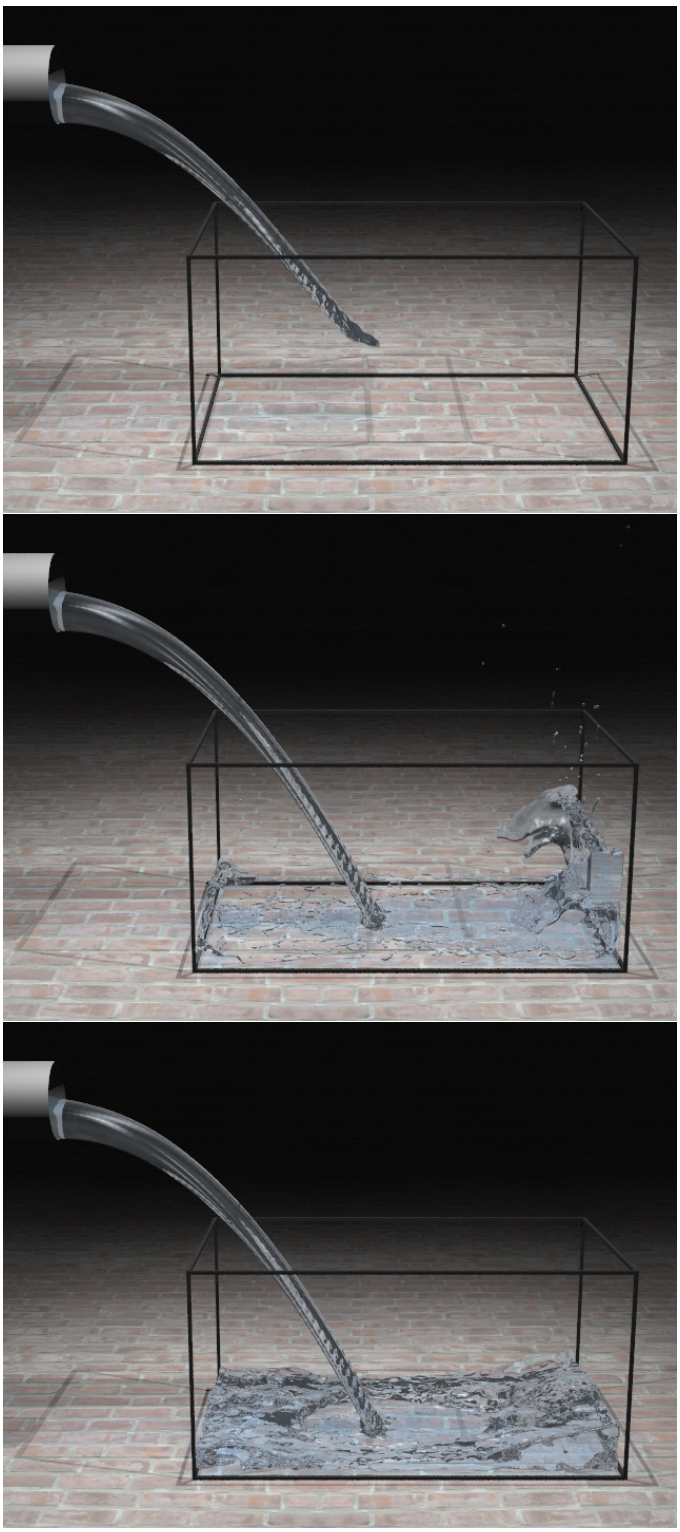


Figure 7.4: *Offline rendered images using POVRAY of a simulation with 250'000 particles using our CUDA based SPH solution. On average 5.6 physics time steps per second are reached on Platform 1.*

CHAPTER 8

TIME ADAPTIVE LOD SPH

8.1 Background and Contributions

In SPH, particle velocities and positions are integrated with a time step at the end of each iteration. A key constraint for SPH based simulations is the time step limitation. A well accepted time step limit for low viscosity fluids is defined by the Courant-Friedrichs-Lewy (CFL) condition. However, although the CFL condition guarantees convergence and stability in simulation, it is often a too conservative choice. Though techniques have been proposed to employ larger time steps in context of incompressible and weakly compressible fluid simulations, no general formulation exists that could be applied for fluids with lower stiffness values as well. We present an adaptive heuristic that employs time steps much larger than induced by CFL, thereby speeding up the computation while preserving stability.

Many fluid simulations used in virtual environments such as 3D games do not need to guarantee exact physical correctness as long as they can produce a visually convincing and physically plausible effect at a higher speed. To this end, a commonly considered solution to speed up SPH is to use adaptive particle sizes. This, however, comes with its own share of limitations and invariably requires dealing with particles of different sizes. As a second contribution, we address the problem of approximating the physical correctness from a different viewpoint, that eliminates the challenges encountered when using variable particle sizes and ensures stability together with acceleration. The presented method segregates particles into active and passive sets based on their location and activity within the fluid. It

is on the passive particles that a sizable computational burden can be saved.

The two main contributions of our approach can thus be summarized as follows:

1. Adaptive global time step optimization for low viscosity fluids.

2. Approximated advection based on particle location and activity.

8.2 Related Work

The commonly accepted criteria to set the time step for low viscosity fluids is the CFL condition. However, this is often a rather restrictive estimate. While [Desbrun and Cani, 1999] provides some insight into selecting adaptive global time steps based on velocity, force and divergence per iteration, it shows that these criteria might not always lead to an optimal choice. [Solenthaler and Pajarola, 2009] and [Ihmsen et al., 2010] can significantly increase the time steps, thereby reducing the overall computation time for incompressible and weakly compressible fluids.

Among other methods that work towards performance optimization are parallel GPU accelerated techniques [Goswami et al., 2010b; Harada et al., 2007b] and adaptive particle sizes [Solenthaler and Gross, 2011; Keiser et al., 2006; Adams et al., 2007; Hong et al., 2008]. In [Zhang et al., 2008; He et al., 2009; Jian et al., 2009] adaptive particle sizing is integrated with GPU based acceleration. Although GPU algorithms can achieve significant performance improvements in comparison to processor independent methods, the number of particles that can be used for the simulation can be limited by graphics memory.

However, several important issues remain unresolved when using particles with different sizes. Not only do the large particles close to smaller ones inflict on them a high pressure force causing stability problems ([Jian et al., 2009]), non-uniform neighborhood search and time steps must also be dealt with in the implementation. Furthermore, often these methods make use of non-trivial functions, like distance of particles to surface, to carry out merging or splitting operations. Moreover, the density profiles before and after splitting or merging particles are not equivalent anymore.

In our work, we introduce a more general and in fact simple way for choosing adaptive time steps per iteration that alone can bring about a significant performance boost. As a second contribution, we present a new approach to efficiently approximate the SPH fluid solver. Our approximation is based on the observation similar to adaptive techniques, that not all particles in a typical simulation play an equal physically relevant role in the global flow and surface reconstruction. The key of our method lies in identifying and separating particles according to

their activity levels thereby indirectly introducing a kind of level-of-detail notion. Thereafter, computational efforts can be geared towards the more active particles. This way we completely circumvent the difficult problems faced by simulations using adaptive particle sizes.

8.3 Global Time Step Optimization

Our first contribution is to introduce a heuristic to select larger time steps for stable simulation. The basic time step formulation in SPH derived from the CFL condition is,

$$\Delta t_{\mathrm{CFL}} = \alpha \cdot \frac{h}{c} \tag{8.1}$$

$$c \approx \sqrt{k} \tag{8.2}$$

where h is the global support radius, c is the velocity of sound propagation in the fluid medium, usually given by Equation 8.2 where k is the stiffness constant, and the constant α is set to 0.4 as per [Monaghan, 1992]. This formulation basically ensures that information propagating through the medium at velocity c cannot skip the discretization distance h in a single time step Δt.

As is obvious from the formulation of Equation 8.1, the CFL condition does not take into account the particle motion itself, their velocity or force to compute a tighter approximation for the time step. In order to consider particle dynamics to determine the time step, Equations 8.3 and 8.4 have been proposed [Desbrun and Cani, 1996], [Becker and Teschner, 2007]. Here $f_{\max}$ refers to maximum force per unit mass on a particle, $\nabla \cdot v$ to the divergence of velocity, and $\beta = 0.25$ and $\gamma = 0.005$ are user chosen constants. The final time step in Equation 8.5 is thus determined by the minimum of Equations 8.1, 8.3 and 8.4:

$$\Delta t_f = \beta \sqrt{\frac{h}{f_{\max}}} \tag{8.3}$$

$$\Delta t_v = \frac{\gamma}{\|\nabla \cdot v\|} \tag{8.4}$$

$$\Delta T = \min(\Delta t_{\mathrm{CFL}},\ \Delta t_f,\ \Delta t_v) \tag{8.5}$$

Figure 8.1(a) compares the time steps obtained for each iteration using the above three equations for a fluid with $k = 1000$. Whereas Δt_v mostly underestimates the time step, Δt_f overestimates with respect to the CFL condition which might lead to instability or shock waves. For low viscosity fluids, the term incorporating the speed of sound dominates the force factor and Equation 8.3 can be

ignored. Therefore, Equation 8.5 will not really provide a tight estimate of current time step based on velocity or force on particles.

[Bridson, 2009] offers a slightly more robust treatment of the CFL condition by suggesting the modification in Equation 8.6. Here v_{max} is the maximum velocity in the simulation, F is the total force acting on a particle and $V_{\max}$ is the largest particle velocity value in the simulation. This solution is slightly more robust because it takes into account the effective force during the current time step.

$$V_{max} = v_{max} + \sqrt{h \cdot F} \tag{8.6}$$

We refine this idea further and overestimate $V_{\max}$ (Equation 8.7) by also including the maximum particle force magnitude in the simulation. The resulting velocity $V_{\max}$ is then used to obtain a larger estimate on the global time step.

$$V_{\max} = v_{max} + \sqrt{h \cdot F_{max}} \tag{8.7}$$

Algorithm 8 outlines our heuristic to choose the time step for each iteration. Here c refers to the relevant speed which is usually taken to be $\sqrt{k}$ and $\alpha = 0.4$ ([Monaghan, 1992]). The basic motivation is to detect the simulation condition where a larger time step can be allowed (line 4 in Algorithm 8). Under such a circumstance, one can safely use a time step η times larger than dictated by the CFL condition without introducing instability or shock waves. η itself is a scalar factor dependent on the stiffness constant of the fluid and is experimentally approximated by the curve in Figure 8.1(b).

The proposed heuristic employs a binary choice for the time step: using the conservative CFL condition, Δt_{CFL} time step when the fluid wave is traveling fast, i.e $V_{\max} > \alpha \cdot c$, and $\eta \cdot \Delta t_{\mathrm{CFL}}$ otherwise. For fluids with larger stiffness constants, up to 2.5 times larger time steps can be used, also see Figure 8.1(b). In PCISPH [Solenthaler and Pajarola, 2009; Ihmsen et al., 2010], one could choose time steps several times larger than directly derived from CFL condition. However, the presented formulation is particularly beneficial to reduce the overall computational time for compressible fluids where the same does not hold true.

Algorithm 8 Time step selection

1: Obtain $v_{\max}$
2: Obtain $F_{\max}$
3: $Compute : V_{\max} = v_{\max} + \sqrt{h \cdot F_{\max}}$
4: **if** $(V_{\max} < \alpha \cdot c)$ **then**
5: $\quad \Delta T = \eta \cdot \Delta t_{\mathrm{CFL}}$
6: **else**
7: $\quad \Delta T = \Delta t_{\mathrm{CFL}}$

8.4 Acceleration by Approximation

In SPH, particles are the information carriers. The movement of particles changes density, which in turn induces pressure forces based on which the particles are moved. The basic steps in SPH computation are given in Algorithm 6, Chapter 6. After each iteration, all particles near the surface are used for detailed surface reconstruction and rendering. A large particle count is in particular required for the rendering of high quality detailed fluid surfaces. However, in order to maintain physical correctness and stability, normally all uniformly sized particles of the densely sampled fluid are processed by the computational simulation loop. This applies equal computational cost to all particles irrespective of their positions or activity levels within the fluid, hence unduly increasing the overall processing cost.

Our second contribution focuses on optimizing the computational burden based on particle location and movement. This comes from the observation that not all particles in any given iteration equally influence the global flow of the fluid. The movement of some particles might create significant changes in the visual and physical details within a few frames, whereas not much difference might be noticed for others. However, the latter kind still continues to claim computational resources since their updated densities and positions are required for the overall stability of the simulation. Seemingly obvious and plausible optimizations, like skipping neighborhood searches or reusing velocities or forces from the last iteration for particles with low activity are not really scalable, and we have experienced that these can quickly lead to unstable simulations. The challenge therefore, is to determine a way to utilize the inactivity of particles in certain regions without using variable particle sizes or other techniques that are critical to the physical simulation stability.

To cope with these problems while still leveraging the non-uniform activity of particles across the simulation, we adopt another approach. In order to save on the computational cost of these passive particles, we set them apart in each iteration from the still active ones. These inactive particles have the following two properties:

1. Their movement does not contribute to a noticeable difference in the visual details (especially at the surface) of the fluid during a few time steps.

2. Their movement does not affect significantly the global flow of neighboring particles, i.e they exhibit rather small local movements.

Our approach now is to temporarily restrict the movement of inactive particles and make such particles stationary until they become reactivated. This results in temporarily stationary, non-moving zones of particles within the fluid, which are

rather in the interior and not usually at the surface. The idea is somewhat similar to *freezing* or *sleeping* or *deactivating* in rigid body simulation [Schmidl, 2002] wherein bodies that have come to rest in simulation are fixed at their current place and their state is not simulated anymore. However, these zones do not stay static for long and can be reactivated upon new nearby fluid motion.

The activity status of a particle can be decided using Equation 8.8 where v_i refers to the magnitude of velocity of a particle and n_i to the magnitude of gradient of the color field (Equation 6.10). V_{cutOff} and N_{cutOff} are corresponding user chosen thresholds. That way no particle is ever treated as passive if it is either moving faster than a certain velocity or if it is near the boundary of the fluid. Here we have chosen $\mathbf{n}_i$ as a metric to select surface particles. One could however use it in combination with the number of neighbors threshold for a better selection. Alternatively, a particle could be defined to be at the surface if the distance to the center of mass of its neighborhood is above a given threshold as given in [Solenthaler et al., 2007].

$$\mathbf{p}_i.\text{active} = (v_i \geq V_{\text{cutOff}} \ \mathbf{OR}\ n_i \geq N_{\text{cutOff}}) \tag{8.8}$$

The proposed method, Algorithm 9, starts with finding all currently active particles first in each iteration. In the next step, each active particle with velocity $v_i \geq V_{\text{cutOff}}$ polls its neighborhood and sets the status of all neighboring particles to be active. By doing so, we prepare neighboring inactive particles to become active again by updating their velocity. This step makes sure that no significant fluid activity is lost. This completes $A(i)$, the set of all active particles.

In principle, all remaining particles are passive and will not be advected during the given iteration. However, some of these inactive particles are in close proximity of active ones and hence their densities and pressures are needed, see also Figure 8.2. We thus define the set $SA(t)$ of semi-active particles for which we also compute neighborhoods, as well as densities and pressures, but force computations are skipped on them. All particles not in $A(t) \cup SA(t)$ constitute the set $P(t)$ of passive particles for which neither neighborhood nor density or force calculations are performed, thereby saving on the computational cost.

Marking neighboring particles as active (line 7) or semi-active (line 9) can be combined with neighborhood computation of particles in $A(t)$. This way we can avoid running the same loop twice for particle selection. Furthermore, we skip neighborhood computation for passive particles. It should be noted that the neighborhood search is one of the most expensive parts in the entire SPH routine, see also [Adams et al., 2007]. Since the choice of active particles has to be made in the beginning of the Algorithm 9 (line 3), we make use of color values from previous frame to decide particle activity in Equation 8.8. By a similar logic, inactive particles skip color computation (and hence neighborhood computation) till they are reactivated by a neighboring active particle.

Algorithm 9 Accelerated SPH Algorithm

Initialize global time step $\Delta T = \Delta t_{\mathrm{CFL}}$
while animating **do**
  find all *active* particles $A(t)$ according to Equation 8.8
  for all particles i in $A(t)$ **do**
    find neighborhood $N_i(t)$
    if $v_i(t) \geq V_{\mathrm{cutOff}}$ **then**
      mark all particles in $N_i(t)$ *active*
      $A(t) \leftarrow A(t) \cup N_i(t)$

  make all non-active neighbors of $A(t)$ *semi-active*, $SA(t)$
  mark all remaining particles as *passive*, $P(t)$

  for all semi-active particles i in $SA(t)$ **do**
    find neighborhood $N_i(t)$
  for all particles i in $A(t)$ and $SA(t)$ **do**
    compute density $\rho_i(t)$
    compute pressure $p_i(t)$
  for all particles i in $A(t)$ **do**
    compute forces $\mathbf{F}^{p,g,v}(t)$
  for all particles i in $A(t)$ **do**
    compute new velocity $\mathbf{v}_i(t+1)$
    compute new position $\mathbf{p}_i(t+1)$

  if required **then**
    adjust V_{cutOff}

  update ΔT using Algorithm 8

In our accelerated approximate SPH method, both semi-active and passive particles do not move. Such particles are static until reactivated by a propagating nearby fluid activity. We achieve this by trimming the standard SPH routines to affect the different particle categories accordingly. Whenever a passive particle is reactivated, it restarts advection with its last active known velocity.

Since in each iteration every active particle adjusts its state with respect to all its neighbors, whether active or passive, simulation never gets unstable. We observe though that for the semi-active and passive particles Newtons' third law is not obeyed since we do not use symmetric and opposite forces to move them. However, this is similar to collision on boundaries, the semi-active particles temporarily form virtual boundaries between advected and static fluid regions. The computational speed-up is obtained at the expense of some momentum loss in each iteration. This is comparable to ignoring smaller higher-order coefficients in a polynomial evaluation and hence is a numerical approximation.

The complete outline of our accelerated SPH fluid solver is given in Algorithm 9. Note that at the end of each iteration, the maximal global time step ΔT is determined using Algorithm 8. Optionally V_{cutOff} can be adjusted if there are too few active particles (line 22).

8.5 Results

The proposed method is implemented in C++ on Mac OS X, with 2.8GHz Quad-Core Intel Xeon hardware and 4 GB 800MHz DDR2 RAM. All the images are generated offline with POVRAY using the particle positions from the simulation.

The graph in Figure 8.3 shows the per iteration time step ratio obtained using our heuristic (Algorithm 8) on the typical *falling block of water* simulation example with stiffness value $k = 1000$ and 100K particles. It can be noticed that a time step as large as η times Δt_{CFL} can be used for many iterations thereby speeding up the simulation by a factor approaching η overall, even for compressible fluids with pretty low stiffness values.

In Table 8.1 we demonstrate the performance gain of our approach over standard SPH. With adaptive time stepping alone, one can achieve speed-up of a factor close to η and sometimes even higher without changing the simulation at all. The performance is then further compared to using adaptive time stepping with approximation on particle movement. We achieve a maximal speed-up of a factor close to 7 for 1M particles and the performance gain improves with particle count and stiffness parameter.

For each demo scene, the initial V_{cutOff} value is specified. Starting with this higher value, V_{cutOff} is incrementally adjusted if the number of active particles are below some threshold. In our experiments, this higher value of V_{cutOff} is initially

set to an arbitrary value $\leq 0.025\%$ of $\sqrt{k}$. Each time the number of active particles reach below certain percentage, V_{cutOff} is reduced by 1% subjected to a lower threshold which is set to be around 0.003% of $\sqrt{k}$ in our experiments. However, these values of V_{cutOff} can be easily altered depending on how much damping is tolerable. On the other hand, N_{cutOff} is set once initially and remains constant throughout the simulation. A simple threshold can be set for n_i to select surface particles, also see [Müller et al., 2003].

Our approach compares well to prior (CPU based) adaptive particle results as reported in [Adams et al., 2007] which reaches a 4.3 times gain for their largest 630K particle original size armadillo model. Our method reaches a 5 times speedup for the 512K particle falling block setup and a nearly 7 times speedup for the large 1M particle simulation. This is also quite in agreement with the two-scale approach presented in [Solenthaler and Gross, 2011] where a 6.7 times speed-up is obtained for 2.8M particles. Since in our case acceleration increases with the particle count, we expect speed-ups higher than 7 for more than 1M particles. Note that our approach does not suffer from handling adaptive particles (i.e. problematic density profiles, merging/splitting particles, stability problems or boundary handling) as well as it does not depend on complex functionality (i.e. adaptive distance and search functions) and is thus comparably much easier to implement. Furthermore, in addition to speeding up standard SPH, our approximation approach could also be integrated into PCISPH. For this, one just needs to include density error additionally in Equation 8.8. This would reduce computational burden to fix the density error once the passive particles are reactivated.

Particles	Demo	k	η	V_{cutOff}	Stan. SPH	Adap ΔT		Adap ΔT + Approx.	
					Time	Time	Speed-up	Time	Speed-up
200,000	WPWC	500	1.5	0.1	24243	15498	1.56	13462	1.8
110,592	SWB	1000	1.9	0.09	21272	12086	1.76	6307	3.37
201,348	WBWC	2000	2.2	0.35	58438	26103	2.24	16263	3.59
512,000	SWB	1000	1.9	0.1	274630	155158	1.77	54305	5.06
1,000,000	SWB	1000	1.9	0.09	927740	509747	1.82	133772	6.94

Table 8.1: *Comparison of performance between standard, time-adaptive and time-adaptive plus approximated SPH for various particle counts and stiffness values k. η is the adaptive time step factor employed in Algorithm 8. All simulation times are given in seconds. WPWC refers to water pipe with collisions, SWB to simple water block and WBWC to water block with collisions.*

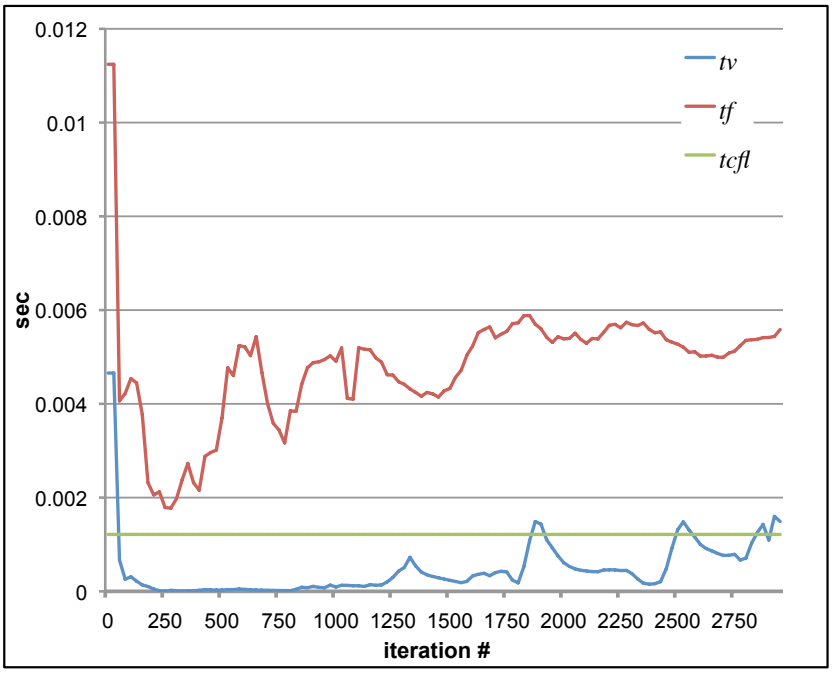
Figure 8.4 demonstrates the frames obtained from our particle simulation with four different demo setups and different stiffness constants and particle counts. Figure 8.6(b) depicts the configuration of semi-active (blue) and passive (red) particles for the displayed simulation in Figure 8.6(a). The zones of inactive particles

move depending on where the particle activity is less. As one can observe from the supplemental video, our optimized and approximated method visually behaves almost indistinguishably from the standard SPH simulation. In Figure 8.5 we compare the visual difference between standard, time adaptive and time adaptive plus approximated SPH simulations for corresponding frames. It can be noticed that all three appear to be virtually identical, with only very minor differences.

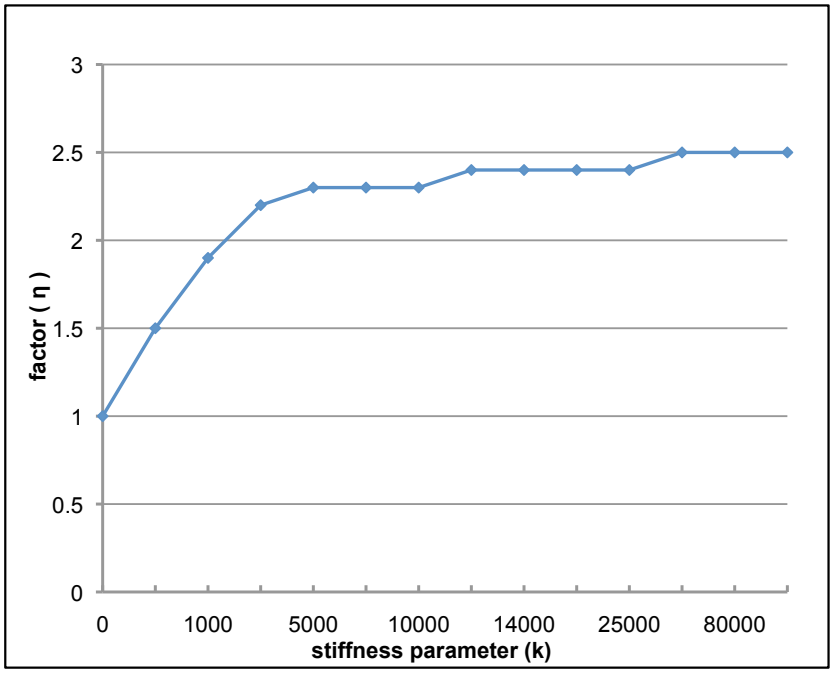
8.6 Discussion

We have presented two techniques to accelerate the standard SPH method. Our global time step selection produces results practically equivalent to the CFL conditioned time step but at a much higher speed, especially in the context of compressible fluids. Furthermore, our particle update optimization introduces an additional performance boost at the expense of advection approximation in the simulation, but still keeps the simulation stable and visually equivalent while circumventing the challenges imposed by adaptive particle size models.

The main limitation of our method is that it can achieve higher speed-ups only if the fluid motion has sizable still regions or eventually settles down. Setting much higher values of V_{cutOff} in such cases can lead to pronounced damping giving the simulation an artificial look. One potential future work along this line could be to assign macro movements to the semi-active and inactive particles in chunks such that all computations for particles within a chunk can be avoided. Further, a theoretical and tighter estimate for choosing η, the speed-up factor in global time step optimization can be investigated.



(a)



(b)

Figure 8.1: *(a) Time step ΔT selection over time using Equations 8.1, 8.3 and 8.4. (b) Approximate value of scale factor η in relation to stiffness constant k.*

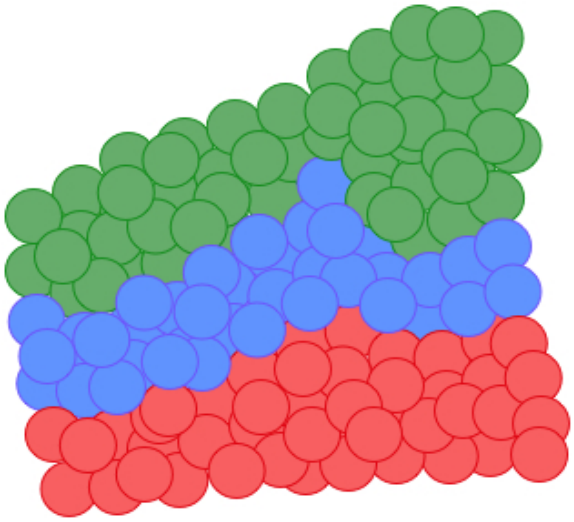


Figure 8.2: *Semi-active particles (blue) separate active (green) and inactive (red) particles forming an implicit virtual boundary.*

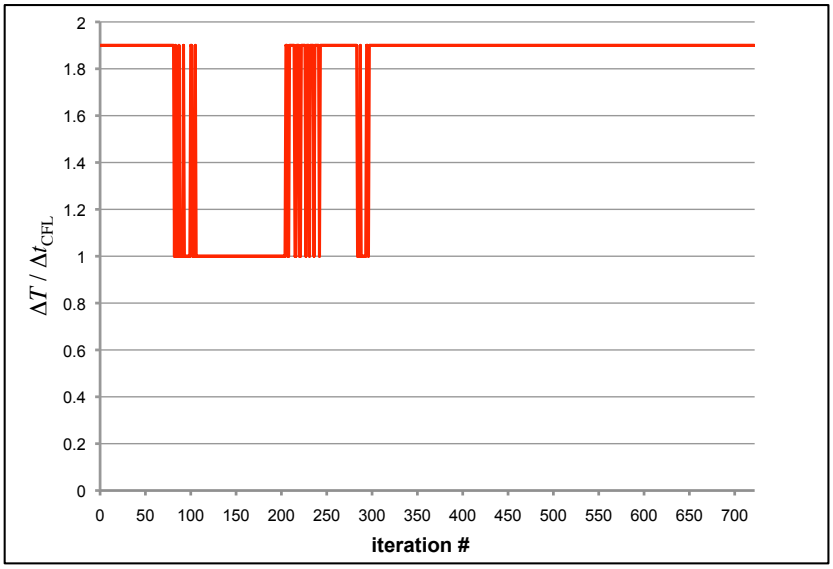
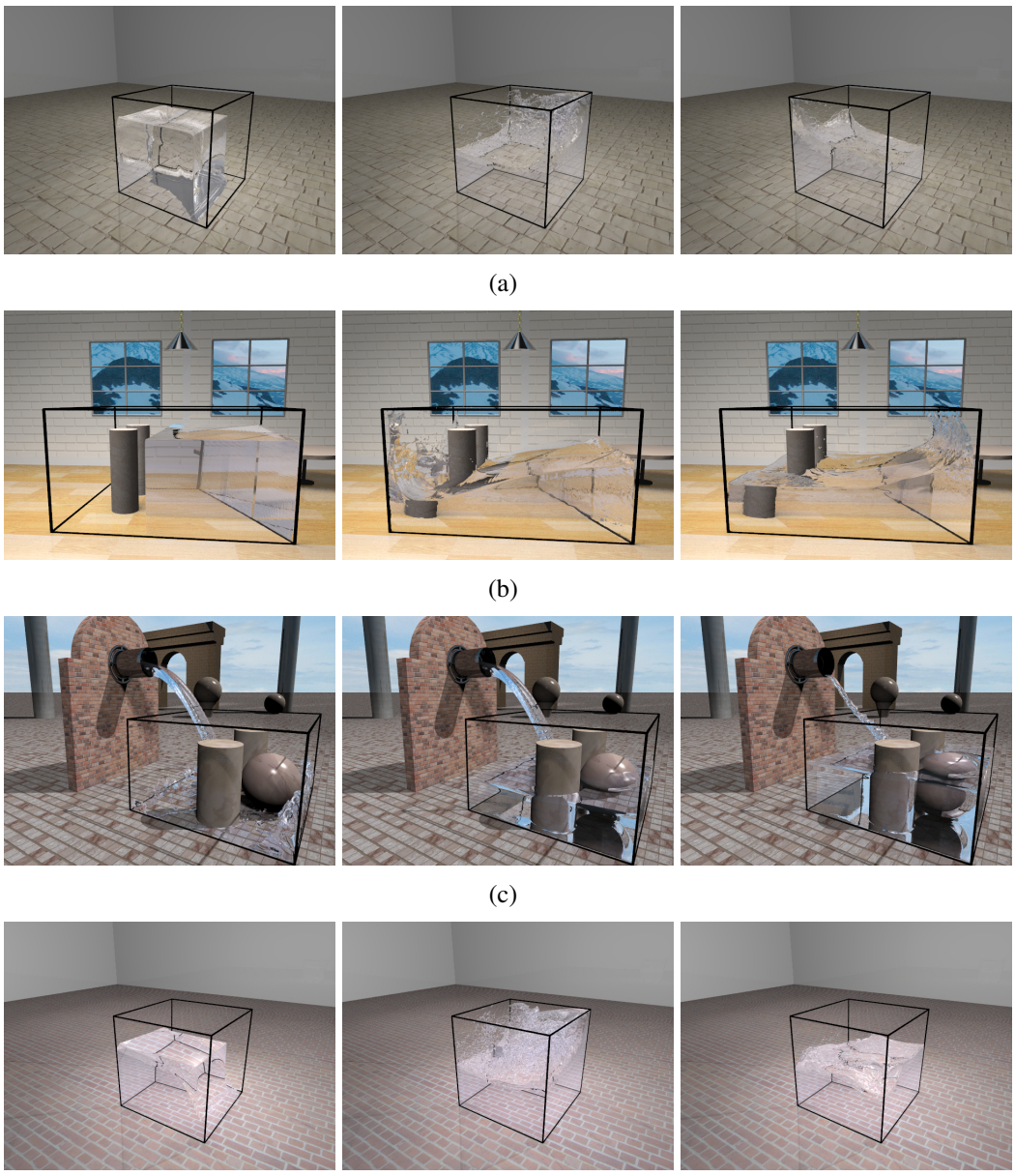


Figure 8.3: *Per iteration $\Delta T/\Delta t_{\mathrm{CFL}}$ time step ratio for 100K particles using stiffness value 1000.*



(a)

(b)

(c)

(d)

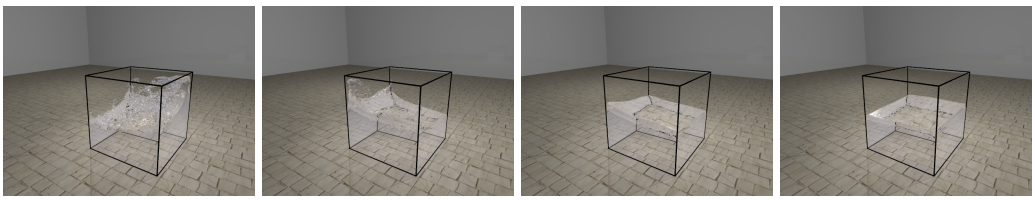
Figure 8.4: *Fluid simulation with the optimized SPH using (a) 100K particles simple water block, (b) 201K particles water block with collisions, (c) up to 200K particles water pipe with collisions, and (d) 1M particles simple water block. Figures (b) and (c) depict collision with cylindrical as well as spherical objects in addition to domain boundaries.*

(a) Standard SPH

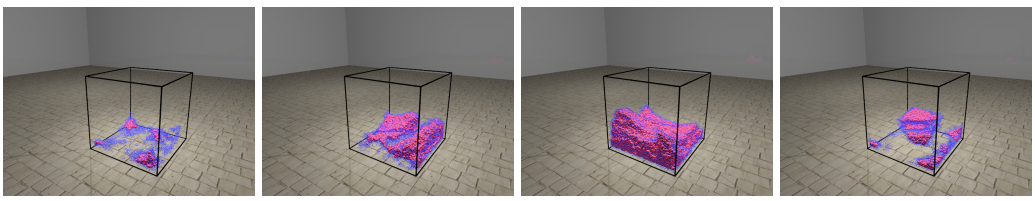
(b) Time Adaptive SPH

(c) Time Adaptive + Approximated SPH

Figure 8.5: *Visual comparison between (a) standard, (b) time adaptive and (c) time adaptive plus approximated SPH for the same time steps.*



(a)



(b)

Figure 8.6: *(a) Surface, and (b) corresponding semi-active (blue) and passive (red) particles for different time steps.*

CHAPTER 9

CONCLUSIONS

We have provided LOD-based and parallel solutions in three different areas: point-based rendering, terrain rendering and fluid simulation. We notice that both these techniques can add to the overall value of the application in terms of user interactivity and resource management. Further, one could achieve parallelization on consumer graphics hardware using GPGPU or CUDA based algorithms on GPU or through OMP on multi-core processor. The choice of parallelization algorithms is often crucial since parallelization itself does not necessary imply acceleration with a given approach. In this concluding chapter, we briefly summarize our work and conclusions for each of these fields individually.

9.1 Point-based rendering

With points as rendering primitives, one could achieve simple preprocessing and more efficient rendering for LOD-based solutions. Preprocessing is simpler due to non-requirement of connectivity information and rendering is faster because unlike triangle, a point is rendered using similar attributes throughout its drawing span and no normal or color interpolation is performed. The benefit also arises in the regions where the LOD representation is a pixel or less. With parallel rendering on multi-machine multi-display clusters, points can be a better choice for colossal models to achieve a proper balance between quality and efficiency, even as a single point might project to several larger pixels. The quality difference can be partially alleviated by employing more sophisticated operations like smooth

blending and geo-morphing or even using round anti-aliased splats. These additional operations do incur a separate cost but in the end points still turn out to be more efficient overall.

In order to achieve constant frame rate, budget-based rendering is efficacious, even more with front-based asynchronous fetching. This is how out-of-core latency can be hidden to a significant extent. For multi-machine parallel solutions, rendering on budget provides a straight edge over traditional pixel error cut-off while still intrinsically maintaining the view-dependent error. One could additionally utilize the categorization of splats into 8 groups within each multi-way kd-tree node for streaming over networks for remote visualization along the lines mentioned in [Gobbetti and Marton, 2004a]. Whereas one of these 8 groups needs to be sent synchronously to the client, others can be sent asynchronously and added to the existing VBO.

The preprocessing operation requiring $\approx k$ output splats for each multi-way kd-tree node can have other variations. This includes *entropy coding* of the simplification process and *k-means clustering*. Whereas latter would turn out to be a more expensive operation, former might provide better quality clusters together with decent efficiency. Our experiments suggest that with the current simplification procedure, even using number of splats in a cell as the error metric provides relatively high quality clusters as compared to some other approaches, for instance iterative edge collapse in [Pauly et al., 2002].

9.2 Terrain rendering

RASTeR improved upon the existing approaches on a few fronts, including separation of triangulation from height the field and GPU oriented handling of M-blocks or the data units. Further benefit was realized with asynchronous front-based delayed fetching of M-blocks and their corresponding textures to maintain continuous out-of-core supply. In order to avoid cracks between adjacent K-patches while still reducing the data size, height values are compressed using conservative reversible *openjpeg* compression. Decompression of the DEM data is done on the CPU. Both compression ratio and decompression time can be improved by employing a suitable GPU based compression scheme wherein the data could be decompressed on the fly on GPU in parallel. Again such a compression could be two-level compression, one level of which maintains regular compression-decompression while the other fixes the errors so as to avoid any cracks or T-junctions at the boundaries.

In our implementation, texture data was compressed up to 5-6 times using GPU compression formats. The quality of compressed texture units is not exactly same to those of the original ones, however, the obvious benefit arises from

compression ratio obtained.

9.3 Fluid simulation

The novel Z-indexing neighborhood search on GPU using CUDA together with other SPH routines accelerated the overall computation time in comparison to existing GPU-based implementations. Barring [Zhang et al., 2008], our method remains to be the fastest performing SPH method on GPU. However, [Zhang et al., 2008] obtain the additional benefit at the expense of dual cost:

1. Reusing density from past iteration which combines force and density computation in same routine.

2. Allocating fixed, pre-allocated neighbor space which overestimates the actual space required for neighbors.

As a next step, to speed-up the SPH loop on CPU, we suggested a heuristic to employ larger time steps than allowed by CFL condition. We further proposed an approximation method for LOD-based SPH without using variable particle sizes. This is achieved by identifying and halting zones within fluids with little global activity. These methods are easy to implement and can also be ported to GPU easily. In applications demanding higher speed with plausible looking accuracy, our method can be easily employed with a simple implementation.

BIBLIOGRAPHY

[Adams et al., 2007] Adams, B., Pauly, M., Keiser, R., and Guibas, L. J. (2007). Adaptively sampled particle fluids. *ACM Transactions on Graphics*, 26(3):48–54.

[Agranov and Gotsman, 1995] Agranov, G. and Gotsman, C. (1995). Algorithms for rendering realistic terrain image sequences and their parallel implementation. *The Visual Computer*, 11(9):455–464.

[Alexa et al., 2001] Alexa, M., Behr, J., Cohen-Or, D., Fleishman, S., Levin, D., and Silva, C. T. (2001). Point set surfaces. In *Proceedings IEEE Visualization*, pages 21–28. Computer Society Press.

[Amada et al., 2004] Amada, T., Imura, M., Yasumoro, Y., Manabe, Y., and Chihara, K. (2004). Particle-based fluid simulation on GPU. In *ACM Workshop on General-Purpose Computing on Graphics Processors*.

[Batchelor, 1967] Batchelor, G. (1967). *An Introduction to Fluid Dynamics*. Cambridge University Press.

[Becker and Teschner, 2007] Becker, M. and Teschner, M. (2007). Weakly compressible SPH for free surface flows. In *Proceedings ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, pages 209–217.

[Bettio et al., 2009] Bettio, F., Gobbetti, E., Marton, F., Tinti, A., Merella, E., and Combet, R. (2009). A point-based system for local and remote exploration of

dense 3D scanned models. In *Proceedings Eurographics Symposium on Virtual Reality, Archaeology and Cultural Heritage*, pages 25–32.

[Bierbaum et al., 2001] Bierbaum, A., Just, C., Hartling, P., Meinert, K., Baker, A., and Cruz-Neira, C. (2001). VR Juggler: A virtual platform for virtual reality application development. In *Proceedings IEEE Virtual Reality*, pages 89–96.

[Bösch et al., 2009] Bösch, J., Goswami, P., and Pajarola, R. (2009). RASTeR: Simple and efficient terrain rendering on the GPU. In *Proceedings EUROGRAPHICS Areas Papers, Scientific Visulization*, pages 35–42.

[Bridson, 2009] Bridson, R. (2009). *Fluid Simulation For Computer Graphics*. A.K. Peters.

[Cignoni et al., 2003] Cignoni, P., Ganovelli, F., Gobbetti, E., Marton, F., Ponchio, F., and Scopigno, R. (2003). BDAM - batched dynamic adaptive meshes for high performance terrain visualization. *Computer Graphics Forum*, 22(3):505–514.

[Clark, 1967] Clark, J. H. (1967). Hierarchical geometric models for visible surface algorithms. In *Communications of the ACM*, volume 19, pages 547–554. IEEE, ACM Press.

[Corrêa et al., 2002] Corrêa, W. T., Fleishman, S., and Silva, C. T. (2002). Towards point-based acquisition and rendering of large real-world environments. In *Proceedings of the 15th Brazilian Symposium on Computer Graphics and Image Processing*, page 59.

[Correa et al., 2002] Correa, W. T., Klosowski, J. T., and Silva, C. T. (2002). Out-of-core sort-first parallel rendering for cluster-based tiled displays. In *Proceedings Eurographics Workshop on Parallel Graphics and Visualization*, pages 89–96.

[Crockett, 1997] Crockett, T. W. (1997). An introduction to parallel rendering. *Parallel Computing*, 23:819–843.

[Dachsbacher et al., 2003] Dachsbacher, C., Vogelgsang, C., and Stamminger, M. (2003). Sequential point trees. *ACM Transactions on Graphics*, 22(3):657–662.

[De Floriani et al., 1996] De Floriani, L., Marzano, P., and Puppo, E. (1996). Multiresolution models for topographic surface description. *The Visual Computer*, 12(7):317–345.

[Desbrun and Cani, 1996] Desbrun, M. and Cani, M.-P. (1996). Smoothed particles: A new paradigm for animating highly deformable bodies. In *In Eurographics Workshop on Computer Animation and Simulation*, pages 61–76.

[Desbrun and Cani, 1999] Desbrun, M. and Cani, M. P. (1999). Space-time adaptive simulation of highly deformable substances. Technical report, Caltech, iMAGIS.

[Duchaineau et al., 1997] Duchaineau, M., Wolinsky, M., Sigeti, D. E., Miller, M. C., Aldrich, C., and Mineev-Weinstein, M. B. (1997). ROAMing terrain: Real-time optimally adapting meshes. In *Proceedings IEEE Visualization*, pages 81–88. Computer Society Press.

[Eilemann et al., 2009] Eilemann, S., Makhinya, M., and Pajarola, R. (2009). Equalizer: A scalable parallel rendering framework. *IEEE Transactions on Visualization and Computer Graphics*, 15(3):436–452.

[Fay et al., 2006] Fay, C., Jeffrey, D., Sanjay, G., Wilson, C., Deborah, W. A., Mike, B., Tushar, C., Andrew, F., and Robert, E. (2006). Bigtable: A distributed storage system for structured data. In *Proceedings OSDI*, pages 205–218.

[Gobbetti and Marton, 2004a] Gobbetti, E. and Marton, F. (2004a). Layered point clouds. In *Proceedings Eurographics/IEEE VGTC Symposium on Point-Based Graphics*, pages 113–120.

[Gobbetti and Marton, 2004b] Gobbetti, E. and Marton, F. (2004b). Layered point clouds: A simple and efficient multiresolution structure for distributing and rendering gigantic point-sampled models. *Computers & Graphics*, 28(1):815–826.

[Goswami et al., 2010a] Goswami, P., Makhinya, M., Bösch, J., and Pajarola, R. (2010a). Scalable parallel out-of-core terrain rendering. In *Proceedings Eurographics Symposium on Parallel Graphics and Visualization*, pages 63–71.

[Goswami et al., 2010b] Goswami, P., Schlegel, P., Solenthaler, B., and Pajarola, R. (2010b). Interactive SPH simulation and rendering on the GPU. In *Proceedings ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, pages 55–64.

[Harada et al., 2007a] Harada, T., Koshizuka, S., and Kawaguchi, Y. (2007a). Sliced data structure for particle-based simulations on GPUs. In *Proceedings 5th international conference on Computer graphics and interactive techniques in Australia and Southeast Asia*, pages 55–62.

[Harada et al., 2007b] Harada, T., Koshizuka, S., and Kawaguchi, Y. (2007b). Smoothed particle hydrodynamics on GPUs. In *Proceedings Computer Graphics International*, pages 63–70.

[He et al., 2009] He, Y., Zhangye, W., Jian, H., Xi, C., Changbo, W., and Qunsheng, P. (2009). Real-time fluid simulation with adaptive SPH. *Computer Animation and Virtual Worlds*, 20(2):417–426.

[Heckbert, 1989] Heckbert, P. S. (1989). Fundamentals of texture mapping and image warping. Master's thesis, Department of Electrical Engineering and Computer Science, University of California Berkeley.

[Hong et al., 2008] Hong, W., House, D. H., and Keyser, J. (2008). Adaptive particles for incompressible fluid simulation. *The Visual Computer*, 24(7):535–543.

[Hoppe, 1998] Hoppe, H. (1998). Smooth view-dependent level-of-detail control and its application to terrain rendering. In *Proceedings IEEE Visualization*, pages 35–42. Computer Society Press.

[Hu et al., 2007] Hu, C., Tian, J., Ming, D., and Shen, D. (2007). Multi-screen tiled displayed, parallel rendering system for a large terrain dataset. In *MIPPR 2007,Medical Imaging, Parallel Processing of Images, and Optimization Techniques, Vol:6789*, pages 47–54.

[Hubo and Bekaert, 2005] Hubo, E. and Bekaert, P. (2005). A data distribution strategy for parallel point-based rendering. In *Proceedings International Conference on Computer Graphics, Visualization and Computer Vision (WSCG)*.

[Humphreys et al., 2002] Humphreys, G., Houston, M., Ng, R., Frank, R., Ahern, S., Kirchner, P. D., and Klosowski, J. T. (2002). Chromium: A stream-processing framework for interactive rendering on clusters. *ACM Transactions on Graphics*, 21(3):693–702.

[Hwa et al., 2005] Hwa, L. M., Duchaineau, M. A., and Joy, K. I. (2005). Real-time optimal adaptation for planetary geometry and texture: 4-8 tile hierarchies. *IEEE Transactions on Visualization and Computer Graphics*, 11(4):355–368.

[Ihmsen et al., 2010] Ihmsen, M., Akinci, N., Gissler, M., and Teschner, M. (2010). Boundary handling and adaptive time-stepping for PCISPH. In *Proceedings Eurographics Workshop on Virtual Reality Interaction and Physical Simulation*.

[Jian et al., 2009] Jian, H., Xi, C., Zhangye, W., Chen, C., He, Y., and Qunsheng, P. (2009). Real-time fluid simulation with complex boundaries. *The Visual Computer*, 26(4):243–252.

[Johnson et al., 2006] Johnson, A., Leigh, J., Morin, P., and Van Keken, P. (2006). GeoWall: Stereoscopic visualization for geoscience research and education. *IEEE Computer Graphics and Applications*, 26(6):10–14.

[Keiser et al., 2006] Keiser, R., Adams, B., J. Guibas, L., Dutré, P., and Pauly, M. (2006). Multiresolution particle-based fluids. Technical Report 520, ETH Zurich, Switzerland and KU Leuven, Belgium and Stanford University.

[Kobbelt and Botsch, 2004] Kobbelt, L. and Botsch, M. (2004). A survey of point-based techniques in computer graphics. *Computers & Graphics*, 28(6):801–814.

[Lario et al., 2003] Lario, R., Pajarola, R., and Tirado, F. (2003). Hyperblock-QuadTIN: Hyper-block quadtree based triangulated irregular networks. In *Proceedings IASTED International Conference on Visualization, Imaging and Image Processing (VIIP)*, pages 733–738.

[Le Grand, 2007] Le Grand, S. (2007). *Broad Phase Collision Detection with CUDA. GPU Gems*. Addison-Wesley.

[Levenberg, 2002] Levenberg, J. (2002). Fast view-dependent level-of-detail rendering using cached geometry. In *Proceedings IEEE Visualization*, pages 259–266. Computer Society Press.

[Levoy and Whitted, 1985] Levoy, M. and Whitted, T. (1985). The use of points as display primitives. Technical Report TR 85-022, Department of Computer Science, University of North Carolina at Chapel Hill.

[Li et al., 1996] Li, P. P., Duquette, W. H., and Curkendall, D. W. (1996). RIVA: A versatile parallel rendering system for interactive scientific visualization. *IEEE Transactions on Visualization and Computer Graphics*, 2(3):186–201.

[Lindstrom et al., 1995] Lindstrom, P., Koller, D., Hodges, L. F., Ribarsky, W., Faust, N., and Turner, G. (1995). Level-of-detail management for real-time rendering of phototextured terrain. Technical report, Graphics, Visualization, and Usability Center, Georgia Tech. TR 95-06.

[Losasso and Hoppe, 2004] Losasso, F. and Hoppe, H. (2004). Geometry clipmaps: Tterrain rendering using nested regular grids. *ACM Transactions on Graphics*, 23(3):769–776.

[Luebke et al., 2003] Luebke, D., Reddy, M., Cohen, J. D., Varshney, A., Watson, B., and Huebner, R. (2003). *Level of Detail for 3D Graphics*. Morgan Kaufmann Publishers, San Francisco, California.

[MacQueen, 1967] MacQueen, J. B. (1967). Some methods for classification and analysis of multivariate observations. In *Proceedings of 5th Berkeley Symposium on Mathematical Statistics and Probability*, pages 281–297. University of California Press.

[Molnar et al., 1994] Molnar, S., Cox, M., Ellsworth, D., and Fuchs, H. (1994). A sorting classification of parallel rendering. *IEEE Computer Graphics and Applications*, 14(4):23–32.

[Monaghan, 1992] Monaghan, J. (1992). Smoothed particle hydrodynamics. *Annu. Rev. Astron. Astrophys.*, 30:543–574.

[Monaghan, 2005] Monaghan, J. (2005). Smoothed particle hydrodynamics. *Rep. Prog. Phys.*, 68:1703–1759.

[Morton, 1966] Morton, G. (1966). A computer oriented geodetic data base and a new technique in file sequencing. IBM, Ottawa, Canada.

[Müller et al., 2003] Müller, M., Charypar, D., and Gross, M. (2003). Particle-based fuid simulation for interactive applications. In *Proceedings Eurographics/ACM Symposium on Computer Animation*, pages 154–159.

[NVIDIA, 2009] NVIDIA (2009). CUDA SDK Samples. *http://developer.download.nvidia.com/compute/cuda/sdk/website/ samples.html*.

[Pajarola, 1998] Pajarola, R. (1998). Large scale terrain visualization using the restricted quadtree triangulation. In *Proceedings IEEE Visualization*, pages 19–26,515.

[Pajarola and Gobbetti, 2007] Pajarola, R. and Gobbetti, E. (2007). Survey on semi-regular multiresolution models for interactive terrain rendering. *The Visual Computer*, 23(8):583–605.

[Pajarola et al., 2005] Pajarola, R., Sainz, M., and Lario, R. (2005). XSplat: External memory multiresolution point visualization. In *Proceedings IASTED International Conference on Visualization, Imaging and Image Processing (VIIP)*, pages 628–633.

[Pauly et al., 2002] Pauly, M., Gross, M., and Kobbelt, L. P. (2002). Efficient simplification of point-sampled surfaces. In *Proceedings IEEE Visualization*, pages 163–170. Computer Society Press.

[Pfister et al., 2000] Pfister, H., Zwicker, M., van Baar, J., and Gross, M. (2000). Surfels: Surface elements as rendering primitives. In *Proceedings ACM SIGGRAPH*, pages 335–342. ACM SIGGRAPH.

[Pomeranz, 2000] Pomeranz, A. A. (2000). ROAM using surface triangle clusters (RUSTiC). Master's thesis, University of California at Davis.

[Puppo, 1996] Puppo, E. (1996). Variable resolution terrain surfaces. In *Proceedings of the 8th Canadian Conference on Computational Geometry*, pages 202–210.

[Rusinkiewicz and Levoy, 2000] Rusinkiewicz, S. and Levoy, M. (2000). QSplat: A multiresolution point rendering system for large meshes. In *Proceedings ACM SIGGRAPH*, pages 343–352.

[Rusinkiewicz and Levoy, 2001] Rusinkiewicz, S. and Levoy, M. (2001). Streaming QSplat: A viewer for networked visualization of large, dense models. In *Proceedings Symposium on Interactive 3D Graphics*, pages 63–68. ACM SIGGRAPH.

[Schmidl, 2002] Schmidl, H. (2002). *Optimization-Based Animation*. PhD thesis, The University of Miami.

[Sivan, 1996] Sivan, R. (1996). Surface modeling using quadtrees. Technical Report CS-TR-3609, University of Maryland, College Park, Computer Vision Laboratory, Center for Automation Research.

[Sivan and Samet, 1992] Sivan, R. and Samet, H. (1992). Algorithms for constructing quadtree surface maps. In *Proceedings 5th International Symposium on Spatial Data Handling*, pages 361–370.

[Solenthaler and Gross, 2011] Solenthaler, B. and Gross, M. (2011). Two-scale particle simulation. *ACM Transactions on Graphics*, 30(4):72:1–72:8.

[Solenthaler and Pajarola, 2008] Solenthaler, B. and Pajarola, R. (2008). Density contrast SPH interfaces. In *Proceedings ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, pages 211–218.

[Solenthaler and Pajarola, 2009] Solenthaler, B. and Pajarola, R. (2009). Predictive-corrective incompressible SPH. *ACM Transactions on Graphics*, 28(3):40:1–6.

[Solenthaler et al., 2007] Solenthaler, B., Zhang, Y., and Pajarola, R. (2007). Efficient refinement of dynamic point data. In *Proceedings Eurographics/IEEE VGTC Symposium on Point-Based Graphics*, pages 65–72.

[Vezina and Robertson, 1991] Vezina, G. and Robertson, P. K. (1991). Terrain perspectives on a massively parallel SIMD computer. In *Proceedings Computer Graphics International*, pages 163–188.

[Wand et al., 2008] Wand, M., Berner, A., Bokeloh, M., Jenke, P., Fleck, A., Hoffmann, M., Maier, B., Staneker, D., Schilling, A., and Seidel, H.-P. (2008). Processing and interactive editing of huge point clouds from 3D scanners. *Computers & Graphics*, 32(2):204–220.

[Williams, 1983] Williams, L. (1983). Pyramidal parametrics. In *Proceedings ACM SIGGRAPH*, pages 1–11. ACM SIGGRAPH.

[Wimmer and Scheiblauer, 2006] Wimmer, M. and Scheiblauer, C. (2006). Instant points: Fast rendering of unprocessed point clouds. In *Proceedings Eurographics/IEEE VGTC Symposium on Point-Based Graphics*, pages 129–136.

[Yin et al., 2006] Yin, P., Jiang, X., Shi, J., and Zhou, R. (2006). Multi-screen tiled displayed, parallel rendering system for a large terrain dataset. In *International Journal of Virtual Reality, 5(4)*, pages 47–54.

[Zhang and Pajarola, 2007] Zhang, Y. and Pajarola, R. (2007). Deferred blending: Image composition for single-pass point rendering. *Computers & Graphics*, 31(2):175—189.

[Zhang et al., 2008] Zhang, Y., Solenthaler, B., and Pajarola, R. (2008). Adaptive sampling and rendering of fluids on the GPU. In *Proceedings Eurographics/IEEE VGTC Symposium on Point-Based Graphics*, pages 137–146.

[Zhou et al., 2008] Zhou, K., Hou, Q., Wang, R., and Guo, B. (2008). Real-time kd-tree construction on graphics hardware. Technical report, Microsoft Research.

[Zwicker et al., 2002] Zwicker, M., Pauly, M., Knoll, O., and Gross, M. (2002). Pointshop 3D: An interactive system for point-based surface editing. *ACM Transactions on Graphics*, 21(3):322–329.

[Zwicker et al., 2001] Zwicker, M., Pfister, H., van Baar, J., and Gross, M. (2001). Surface splatting. In *Proceedings ACM SIGGRAPH*, pages 371–378. ACM SIGGRAPH.

CURRICULUM VITAE

Personal Information

Name	Prashant Goswami
Date of birth	May 17, 1980
Place of birth	Jhansi, India

Education

2007 - 2011	Doctoral student and teaching assistant at Visualization and Multimedia Lab, University of Zürich, Switzerland
2000 - 2005	Dual degree (M.Tech, B.Tech) in Computer Science and Engineering, Indian Institute of Technology, New Delhi, India
1984 - 1998	Primary and secondary education, Carmel Convent School, Gwalior, India

Publications

Conference

J. Bösch, P. Goswami, R. Pajarola. RASTeR : Simple and Efficient Terrain Rendering on the GPU. *Proceedings EUROGRAPHICS Areas Papers, Scientific Visulization*, pp. 35 - 42, 2009.

P. Goswami, M. Makhinya, J. Bösch, Renato Pajarola. Scalable Parallel Out-of-core Terrain Rendering. *Proceedings Eurographics Symposium on Parallel Graphics and Visualization*, pp. 63 - 71, 2010.

P. Goswami, P. Schlegel, B. Solenthaler, R. Pajarola. Interactive SPH Simulation and Rendering on the GPU. *Proceedings ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, pp. 55 - 64, 2010

P. Goswami, Y. Zhang, R. Pajarola, E. Gobbetti. High Quality Interactive Rendering of Massive Point Models using Multi-way kd-Trees. *Proceedings Pacific Graphics Poster Papers*, 2010.

P. Goswami, R. Pajarola. Time Adaptive Approximate SPH. *Proceedings of Virtual Reality Interaction and Physical Simulation*, pp. To Appear, 2011

Journal

Under submission

Student thesis

S. Holm, P. Goswami. Data Management for Terrain Rendering. *University of Zürich*, 2009.

R. Mukhi, P. Goswami. Point Based Simplification. *University of Zürich*, 2011.

Thesis/Projects

P. Goswami, P. Kalra Octree-based Rendering of Deformable and Dynamic Objects. *Indian Institute of Technology, Delhi*. Master's thesis, 2005.

P. Goswami, K. Biswas. Audio Rendering. *University of Central Florida*. Summer Internship, May - July, 2004.

P. Goswami, P. Sampath. Reducing the cost of data flow analysis by congruence partitioning. *TRDDC, Pune*. Summer Internship, May - June, 2003.

Industrial Experience

July 2005 - March 2006	Oracle, Bangalore Applications Engineer in iStore team.
March 2006 - June 2007	Webaroo, Mumbai Software Engineer in Client team.

Achievements

Masters GPA : 8.412 (Position 3^{rd})
Graduate Aptitude Test in Engineering, CS, 2004 : 99.04 percentile
IIT Joint Entrance Examination, 2000 : All India Rank 132 (/125,000)
State Engineering Test, 1999 : Ranked 11^{th} (/200,000)
High School, 1996 : 88.4%, CBSE (Position 1^{st})
TOEFL, 2005 : 273(/300)