UNIVERSITÀ DI PISA

Facoltà di Scienze Matematiche, Fisiche e Naturali

Corso di Laurea Specialistica in Informatica

Implementation of a sort-last volume rendering using 3D textures

Master Thesis

Author:

Matteo Salardi .....

Advisors:

Prof. Thilo Kielmann .....

Dr. Paolo Cignoni .....

Academic Year 2005/2006

Vrije Universiteit Amsterdam

Acknowledgements

The main part of the implementation described in this work was accomplished at the Vrije Unversiteit of Amsterdam. I am extremely grateful to Prof. Thilo Kielmann, for the great opportunity he gave to me. I am grateful to Tom van der Schaaf, for his endless patience and skillful support. I thank the Vrije Universiteit, since for several months i used its facilities. I wish to thank Dr. Paolo Cignoni and the Department of Computer Science of Pisa, which allowed me to prepare my master thesis abroad.

The University of Pisa supported this thesis with a scholarship. The rest was provided by my parents and my gung-ho grandmothers. I am deeply grateful to all of them.

to Valentina

Contents

Chapter 1

Introduction

The present work describes an implementation of a parallel volume rendering, accomplished by extending an existing graphics library, Aura, and employing the OpenGL API. The volume rendering is a field belonging to the more general area of computer visualization, that is the efficient rendering onto a display of a certain data set, in order to better understand and investigate properties and structures of the data set. The branch of visualization techniques concerning volumetric data set is the volume rendering. This work is implemented using the 3D texture method, that is a volume rendering technique which takes advantage of the new hardware facilities of modern graphics boards. This approach is coherent with state of the art works, since a truly interactivity can be achieved only exploiting the underlying hardware. The reason to adopt a parallel solution is the need for processing data set that are larger than the ones supported by a standard graphics board. A cluster of PCs was used to test the volume rendering, according with recent research trends. Actually clusters are preferred to customized high-end workstations,

basically because clusters of commodity components are cheaper and inherently easier to upgrade. Sort-last is the parallel model adopted, that is the original 3D texture is distributed among the nodes of the cluster, and it is splitted in a slicing phase. Each node is in charge of rendering its part of the volume, then the framebuffer of each node is read and sent to a node which performs the final compositing phase. Special cares are necessary if the final compositing also requires alpha blending. A binary swap algorithm among the nodes is implemented too. The binary swap allows to reduce the compositing bottleneck that is generated when all the entire framebuffers are sent to the same final node.

Chapter 2 exposes the main techniques in volume rendering. In section 2.1 a short overview is provided, where the main application fields are pointed out. It is introduced the first and simplest technique, the volume slicing. Section 2.1.3 is about the indirect volume rendering, where the data set is first fitted with geometric primitives and then is rendered. The aim is to approximate with the geometry a certain iso-surface of interest, then the rendering is achieved feeding the graphics board with the geometric primitives. The basic algorithm for performing an indirect volume rendering is the marching cubes. Section 2.2 presents the other main class of algorithms for volume rendering, that is the direct volume rendering. The main difference with the indirect techniques is that the data set rendering is achieved without any intermediate geometry. The main techniques are ray casting, splatting, cell projection, shear-warp and finally the hardware based texture mapping. Section 2.3 describes the state of the art. Recent research trends tend to exploit the new GPUs features and to achieve volume rendering for large data

set. Thus an algorithm is presented which intensively use the programmable shaders to perform an hybrid method made up of texture mapping and ray casting. Furthermore, the main issues related to irregular data set, out-of-core algorithm and dynamic data set are presented. Section 2.4 shortly describes the OpenGL API and its functions to create a 3D texture. Finally the Aura graphics library is introduced.

Chapter 3 deals with parallel volume rendering, since several techniques have been proposed to exploit hardware made of multiple computational nodes. Section 3.1 presents an overall analysis of parallel rendering, first comparing the hardware support given by high-end graphics workstations with clusters of commodity components. The Molnar's taxonomy, a landmark for classification of parallel rendering algorithms, is introduced, followed by a deeper description of its three main classes: sort-first , sort-middle and sort-last. Section 3.2 points out state of the art techniques and research trends in parallel rendering. Hierarchical data structures are a preferred way to manage large data set, especially when it has to be distributed among several nodes. The multi-threaded approach is also widely employed. An hybrid algorithm which fall into both sort-first and sort-last classes is described. Its performance overcomes the traditional sort-first or sort-last algorithms in many cases. Section 3.3 presents software related to the parallel rendering. The Chromium library is widely accepted as the outstanding software for parallel rendering. Aura, used for our work, also supports the parallel rendering.

Chapter 4 concerns our implementation of a sort-last volume rendering using the Aura library. Section 4.1 is about the design choices taken to

add the new components to Aura. First an overview of the Aura internal structure is given. The OpenGL buffers system and the read and write operations on buffers are described. The project design is motivated and the new components are introduced. Section 4.2 is about the slicing strategy adopted in our work and the compositing phase. The alpha compositing theory is introduced, according to the Porter and Duff results. OpenGL implementation of the alpha compositing is given. Alpha compositing in the 3D domain is generally achieved using the stencil buffer. However our implementation can avoid the use of the stencil buffer. Section 4.3 deals with the parallel compositing issue. The need for parallelism at the compositing phase is pointed out. Three algorithms are proposed: direct send, parallel pipeline and binary swap. The latter is the one adopted in our work.

Chapter 5 gives validation results. Chapter 6 presents the final considerations and gives suggestions for future works.

Chapter 2

Volume rendering

2.1 Introduction and indirect techniques

2.1.1 Overview

Scientific visualization of volumetric data is required in many different fields of interest. Medical applications are an obvious example. Several tools are nowadays available to obtain image data from different angles around the human body and to show cross-sections of body tissues and organs. The purpose is to support the physician diagnosing internal disorders and diseases. The computed tomography (CT) scanner consists of a large machinery with a hole in the center, where a table can slide into and out. Inside the machine an x-ray tube on a moving gantry fires an x-ray beam through the patient body, while rotating around it. Each tissue absorbs the x-ray radiation at a different rate. On the opposite side of the gantry a detector records this 1D projection, after a 360-degrees rotation around the body a computer can

compose all the projections to reconstruct a 2D cross-sectional view of the body, or slice. The table advances and the same procedure is repeated for the new section, until the interesting area is completely scanned and a set of 2D slices is ready. Besides CT, magnetic resonance (MR) and ultrasound imaging are widespread methods to produce a digital representation of the human internals.

There are also other scientific areas which require to handle huge amounts of image data. The goal is still to improve the knowledge of the data set achieving a high-quality rendering and allowing a flexible and efficient manipulation. Numeric simulations in physics as computational fluid dynamics are a good example. In sophisticated molecular modeling, oceanographic experiments or meteorological simulations the parameters needed to be investigated are associated with shades of gray or, more commonly, with RGBA colors. Generally speaking, all the areas concerning the study of a scalar field can find an important support representing the 3D domain as a volumetric data set:

$$f : D_1 \subset \Re^3 \rightarrow D_2 \subset \Re$$

Using a different and more convenient terminology, the data set is represented as a 3D array of scalar values, or voxel grid. Each of the elements of the array is called voxel. The function which describes how the voxels are spaced determines the grid type. The simplest case is of course the constant function, where each voxel is regularly spaced ($i * d, j * d, k * d$ or $i * dx, j * dy, k * dz$) and thus the grid is called regular. Otherwise the grid is curvilinear ($x[i], y[j], z[k]$ or also $x[i, j, k], y[i, j, k], z[i, j, k]$). The grid is referred to as unstructured if no function can be given to describe the voxels

position.

Since real world data sets are usually generated and stored using the scale of greys, it is necessary to map each voxel to a RGBA color, in order to highlight the wished elements of the data set. This task is accomplished using one or more transfer functions, which associate a RGBA value to a grey one. For instance, the skin of a human hand has a certain constant value on the scale of greys. The transfer function can be used to highlight the skin, associating to its value a bright RGBA color. If we prefer to investigate the hand internals the skin can be mapped to a transparent color (i.e, the alpha value is set to zero). In the next paragraphs a short taxonomy of the main techniques adopted in the volume rendering field is provided.

2.1.2 Volume slicing

The main task of a visualization application is to reduce the volumetric data set to a 2D image, since the output is usually displayed on a 2D surface. The simplest way to achieve such result is to extract a planar 2D slice from the volume and to directly show it onto the screen, $I(x, y)$. Of course with this technique the data loss is enormous, all the advantages to exploit the spatial organization of the data is lost. A method to partially overcomes this problem is to show several slices simultaneously, or in a temporal sequence

$$I(x, y, t) = f(\alpha_1 x, \alpha_2 y, t)$$

In this case time replaces the missing spatial coordinate. Otherwise it's possible to display several slices at the same time, using also orthogonal cut-

ting to allow the user to mentally reconstruct the three dimensional relations of the volume. Orthogonal slices can also be embedded together to improve the general comprehension of the volume.

2.1.3 Indirect volume rendering

A second approach is represented by the indirect volume rendering (IVR), also referred to as surface rendering. Data are converted in a set of polygons, and then rendered using the standard graphics hardware, typically able of efficient rendering for geometric primitives. Here the main task is to detect the surfaces of interest, and then to fit the volume with the appropriate primitives. The surface is usually identified with a constant value, or iso-value, so only the voxels of our volumetric data set with this value are displayed.

$$f(x, y, z) = k,\ k\ constant$$

Cuberilles was one of the first widely used methods for this purpose [12], the basic idea is to use a binary segmentation to determine which voxels belong to the surface and which do not. All the voxels of the object are then joined together to approximate the final surface. If the number of voxels is not large enough the final image can look like it's made up of squared bricks. To improve the image quality the original data set can be re-sampled increasing the number of voxels, but this implies at the same time to increase the computation time.

The well-known marching cubes algorithm is an important evolution of this technique, proposed by [21]. In this case the volume is organized in

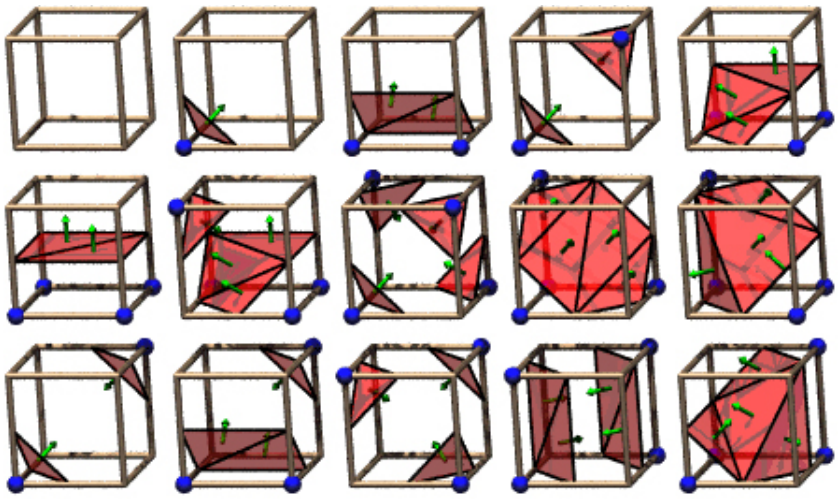

Figure 2.1: Marching cubes basic combinations. Source: http://groups.csail.mit.edu/graphics/classes/6.838/F01/lectures/Smooth Surfaces/0the_s047.html

cubes made up of 8 voxels. The goal is to detect which ones of the 8 voxels of each cube belongs to the iso-surface, and then to create the appropriate triangular patches which divide the cube between regions within the iso-surface and regions outside. At each step a cube is classified with an index for the values of its corners, potentially the combinations are 256 but a strong simplification exploiting symmetries gets only 15 combinations. The index is used to look up a precalculated table and to obtain the correct patches, then vertexes are interpolated and the normal is calculated. Finally, the surface representation is obtained by connecting the patches from all cubes.

Although marching cubes is a standard and wide used method for IVR, many improvements were proposed to overcome unexpected behaviors due to ambiguous cases. One way is to add 6 new combinations to the original 15 to remove ambiguities. Another way is the marching tetrahedra algorithm,

where tetrahedra are used instead of cubes to detect surfaces. A finer detail is provided and the arising of ambiguous cases is avoided.

2.2 Direct volume rendering

2.2.1 Ray casting

Another approach is the direct volume rendering, also simply known as volume rendering. The data set is directly projected onto the 2D viewing plane without fitting visible geometric primitives to the samples. The data set is represented as a kind of 3D cloud. The basic styles are ray casting, splatting, cell projection, shear-warp projection and texture-based mapping, but it's common to combine these ground techniques to obtain hybrid methods.

Ray casting [19] is a special case of the more complex ray tracing method. A ray is fired from every pixel in the image plane into the data volume, and for this reason ray casting is classified as an image-order approach. Along the intersection of the ray with the volume shading algorithms are performed to obtain the final color of the pixel, using a back-to-front order for the final composition. The contribution of each data sample to the pixel depends on a set of optical properties, like color, opacity, light sources, emission and reflection model. A first advantage compared to the IVR is that the volume shading is free of any classification of the voxels, while only the voxels detected as belonging the surface are rendered in the IVR, resulting in frequent false positives (spurious surfaces) or false negatives (unwished holes in surfaces).

Ray casting performs a surface classification by mapping the voxels to an

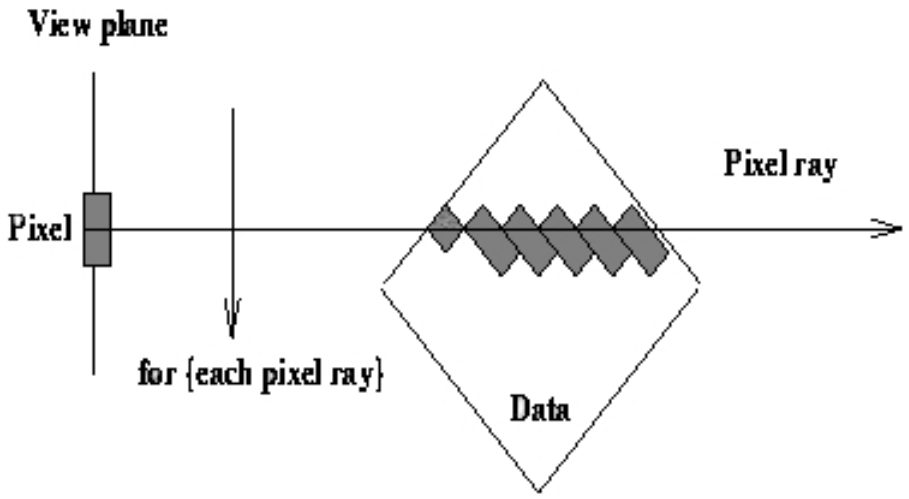

Figure 2.2: A ray is fired through the volume. Source: http://smohith.tripod.com/proj/vhp/thesis/node92.html

opacity value. But ray casting is an extremely computation intensive process, that's why many improvements have been proposed to increase the performance. The main remark is there are usually many transparent regions (i.e., opacity is zero) inside the volume, they do not give any contribute to the final image but they are processed as significant ones. Transparent regions can be skipped modifying the ray casting algorithm with the hierarchical spacial enumeration [20]. An additional pyramidal data structure is required. All the voxels are grouped together into cells of different sizes, the minimum with the same size of a single voxel and the maximum with the size of the entire volume. Each cell also contains links to parents and children cells and a value set to one if there are children of the cell with an opacity greater then zero, zero otherwise. So when a ray is cast the pyramidal structure is traversed starting from the biggest cell and at each step the cell value is tested. The

voxels on the base are reached and shaded only if all the cell parents get a one as result, if a zero is encountered it means there is a transparent region. Another optimization can be accomplished avoiding inner regions with an high opacity coefficient when the ray has already accumulated an opacity value close to one processing the data. A threshold is set and if it is reached the ray can be terminated, further voxels would not be visible.

Ray casting is also employed to implement a different technique which does not belong to the DVR algorithms, the maximum intensity selection (MIP). The basic concept is to select the maximum value along with each ray, in such a way only a little subset of voxels contribution to the final image. This can be useful to depict a specific structure from the volume ignoring the rest, as blood vessels in a data set produced by a MR scanner. MIP exists in many variants as local MIP, where the first value above a given threshold is selected. Furthermore, MIP can be joined with a DVR technique to obtain a hybrid method.

2.2.2 Splatting and cell projection

Splatting, first proposed by [38], is an object-order approach, either back-to-front or front-to-back; pixel values are accumulated by projecting footprints of each data sample onto the drawing plane. This method processes only objects existing in the volume, avoiding transparent regions. It's particularly efficient for sparse data sets. To skip opaque inner regions is more difficult, because a voxel which doesn't contribute to a certain footprint can contribute to another one.

A similar approach is represented by the cell projection algorithms, such

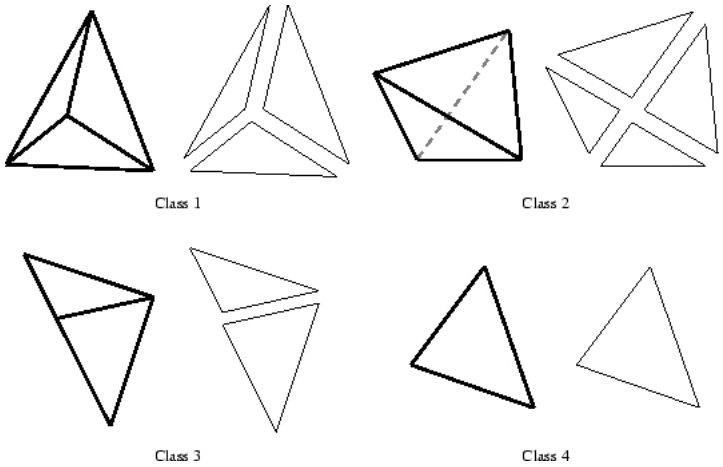

Figure 2.3: The four tetrahedra configurations. Source: http://www.cs.unm.edu/~kmorel/documents/dissertation/thesis_full/node28.html

as the projected tetrahedra of [31]. The volume is first decomposed in tetrahedrical cells which are then classified according with the viewpoint vector. Tetrahedrons were chosen as the cell-shape because they are the simplest possible polyhedron (4 vertices, 6 edges and 4 faces) and because they are simple (non self-intersecting) and convex. Any kind of simple polyhedron can be decomposed in a set of tetrahedrons. Furthermore, a tetrahedron projected onto a plane can be broken into triangles, only four configurations are possible. Thus once a tetrahedron is classified it can be decomposed with triangles, values are linearly interpolated and then the triangles are rendered.

All the above methods are high quality methods with similar performance, with a better behavior of splatting for sparse volumes and with an easier parallel implementation for ray casting. The next methods have a lower quality but they are faster.

2.2.3 Shear-warp

A third method is shear-warp [16]. It is a hybrid technique which traverses both image and object space at the same time. Unlike ray casting, the projection of the data set onto the surface is splitted in two different steps, a shearing along the axis of the volume and a final 2D warp operation. The sheared volume is projected on a base plane, normal to an axis of the volume coordinate; the goal of the shearing phase is to simplify the projection operations, that usually represent the computational bottleneck in many algorithms, like ray casting. Finally the warp step corrects the image distortion, due to the angle between the base plane and the viewing plane. To accelerate the shear-warp rendering usually run-length encoding is used for sequences of voxels with similar properties, for example for transparent regions. All the pixels covered by the projection of a run are treated equally, increasing the performance.

2.2.4 Texture-based rendering

Texture-based methods are a last approach, which takes advantage of the texture mapping hardware. Although all the above basic methods can reach important results and many efforts have been done to further improve their performances, a truly real time DVR cannot be achieved only via a software solution. If only 2D textures are available, we consider three axis aligned sets of slices of the volume, each applied as 2D texture; the set that is most perpendicular to the viewing direction is selected and alpha-blended in a back-to-front order. Otherwise, if 3D textures are supported, only one set

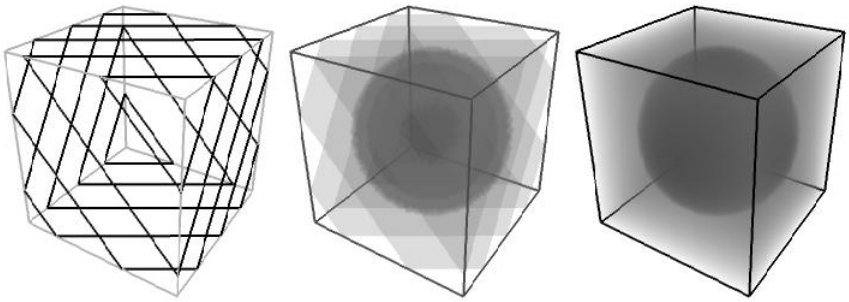

Figure 2.4: 3D texture. Source: [14]

of slices is necessary. They are constructed aligned in the object space or, more likely, in the image space (i.e., perpendicular to the viewing direction) and placed within the volume. Slices are then rendered using a trilinear interpolation for the slices coordinates, using color and opacity values from the volume loaded into the texture memory. To achieve the final rendering the slices are blended together, again in a back-to-front order.

Since its first introduction the hardware based approach could overcome long term performance issues related to volume rendering. In order to fully understand its success it is worth noting that nowadays the power of GPUs is increasing faster then the power of CPUs, thus there is a huge availability of cheap and powerful graphics boards. However, data set size is also increasing fast, and the texture memory available on a graphics board represents a strong constraint to the hardware based approach. Moreover, this method produces several fragments which may not contribute to the final image, as seen for other techniques such as ray casting.

2.3 State of the art

2.3.1 Hybrid methods

Recent works usually propose hybrid methods, where the good performances of hardware based texture mapping are combined with the advantages of other methods. [14] intensively exploit the new features available on the new generation of graphics boards, that is programmable shader units, high bandwidth to access the texture memory and high parallelism inside the rendering pipeline, in particular with the presence of several fragment units. The basic idea is to generate slices textured with color and opacity from the data set as in the standard hardware based texture mapping, and then to adopt ray casting to skip inner regions and empty voxels or to detect iso-values. In order to implement such method the hardware support has to allow the redirection of the rendering process to a 2D texture (used to store the intermediate results, the texture is aligned with the viewport and it has the same resolution), to allow the generation of texture coordinates (to access slices and temporary textures while a ray is traversing the volume) and to calculate arithmetic operations (but also more complex ones as dot product) in the shader programs, and finally to allow a fragment to change its depth value.

The method requires four passes:

1. Entry point determination. The front faces of the volume are rendered and stored in a temporary 2D RGB texture (TMP). The color values represent the 3D texture coordinates, thus the first intersection of the rays of sight with the volume.

2. Ray direction determination. The same procedure is executed for the back faces of the volume, the rays direction and length are computed and stored in a 2D RGBA texture (DIR), where the RGB components are used for the normalized direction and the A component for length.

3. Ray traversal and early ray termination. Rays are casted through the volume and the accumulated value is determined at each pass. The textures generated at the previous steps TMP and DIR are used as lookup tables, while another 2D texture (RES) is employed to store intermediate results. The early ray termination is easily performed testing the current ray length with the length stored in the A component of DIR. If the ray exits the volume the corresponding A component in RES is set to 1. The iso-surface detection can be performed traversing the volume from back-to-front at this pass. When the current ray value is equal or greater to a certain iso-value it is stored on a temporary variable.

4. Stopping criterion. The A component in RES related to the current ray is compared with a threshold value, if it is greater the respective depth value is set to a maximum value, otherwise to 0. The idea is to avoid useless computations using the z-buffer test. The same idea can be fruitful employed to perform an empty space skipping. An additional 3D RGB texture is needed to store octree hierarchy information. It corresponds to a particular octree level with nodes of constant size. The minimum and maximum value of the child nodes (i.e., the bounds of the region covered by the octree node) is stored in the R and G components. This extra 3D texture is used to address a 2D texture, where for each

pair R/G it is shown if there are non-zero color component inside, that is for each region it indicates if it is empty or not. The empty space skipping test can be performed together with the threshold test, if an empty region is detected the fragment depth value is set to the maximum value (but whenever a non empty region is found is reset to 0).

2.3.2 Unstructured grids

Besides algorithms exploiting new GPUs features as seen above, many algorithms have been developed to apply DVR techniques tailored for regular grids to unstructured ones. The irregular data set is usually represented by a tetrahedral mesh, that is a set of tetrahedra where for each pair of tetrahedra is possible to determine if they are disjoint, incident or adjacent. However meshes made of other types of cells are also possible, such as hybrid meshes made of tetrahedra and irregular hexahedra. A first issue is to perform a visibility ordering on the cells, that is a total order such as if cell A obstructs the cell B for a given viewpoint, then B precedes A in the ordering. The visibility ordering is essential to achieve a correct final result and it is useful to efficiently exploit the graphics hardware.

A first object oriented approach is the Meshed Polyhedra Visibility Ordering (MPVO), where an adjacency graph is computed from the mesh. Basically, each node of the graph correspond to a cell, and the edge between two nodes represents a shared face. The shared face defines a plane which divides the space into two half spaces, each containing one of the two nodes. The edge is oriented towards the node lying on the same half space of the

viewpoint, thus the entire graph can be oriented and the visibility ordering achieved.

An opposite, image oriented approach is the A-buffer, where the ordering is performed after the rasterization. Each primitive is rasterized without any concern about the ordering, then the generated fragments are stored in a per-pixel linked list containing the depth values. The depth values are then used to sort the list. The main disadvantage of the A-buffer method is it requires extra memory to store the linked lists.

An hybrid approach is represented by the k-buffer, which combines the previous methods. A first approximate ordering is performed by the CPU on the cells, then the primitives are rasterized and a further ordering is performed by the GPU using the k-buffer. The main difference between the A-buffer and the k-buffer is that while the A-buffer memory requirements are unbounded since linked lists are stored on the A-buffer, the k-buffer requires a constant amount of memory, that is k entries are stored for each pixel. Each entry contains the distance of a fragment from the viewpoint, it is used to sort the fragments.

After the visibility ordering is accomplished the standard DVR methods can be adopted as for regular grids. It is worth noting DVR of an unstructured grid is much easier if all the cells are convex, otherwise problems can arise sorting the cells and applying the conventional algorithms (e.g., a ray traversing a non-convex mesh with ray casting enters and exits a cell more then once). Convexication of a mesh can be performed, nowadays it is still a hard task though.

2.3.3 Large and dynamic data set

The need for processing large data set represents another outstanding area of interest, since data set size increases faster then hardware support. A first method is to simplify the data set in order to obtain a data set that is easier to handle. This is usually achieved with a pre-processing step, however it produces a static approximation, thus it can introduce artifacts. If a texture based method is being used, the pre-processing can split the original texture into several bricks, detect and discard the empty regions and finally rearrange the 3D texture. A more refined method consists in using hierarchical data structure to provide a multi-resolution or level-of-detail (LOD) representation of the data. [36] adopts this approach combined with the 3D texture based technique. The texture is divided into several bricks, then for each brick a set of coarser approximations is computed, thus a multi-resolution hierarchy structure is generated. Only the more interesting regions of the data set are rendered at the higher resolution, the less significative areas are loaded and rendered at a coarser resolution, reducing the impact of the trilinear interpolation and thus improving the interactivity.

Although all the above techniques allow to treat large data set, modern applications and scientific experiments may need to handle even larger data set, overwhelming the standard PC resources. This happens when the data set have been generated by a supercomputer but the visualization process has to be accomplished on a PC or a mid-range workstation. A solution to this issue is addressed by the out-of-core approach, that is to use the disks in order to overcome the main memory limits. The out-of-core approach was not first employed by computer visualization, since it was already adopted for

computational problems in science and engineering, which do not fit inside the main memory, and for the database field, which is inherently out-of-core. The main overheads introduced are due to the communication bottle-neck between the disk and the memory and to the inefficient random access to the disk, because each seek operation implies the movement of mechanical parts. Thus the out-of-core approach is required to overcome such overheads, usually providing an ad-hoc data structure to organize and efficiently handle the data set on disk. For instance, [6] restructure the original data set using an octree during a pre-processing step. The octree is recursively generated inserting at each phase the cells into the octree node. When a node needs to be splitted because it contains an exceeding amount of cells it creates its children and distributes the cells among them. If a cell could intersect more than one node it is replicated. After the pre-processing, the octree is used to load only the required portion of the data set on demand. [8] propose two out-of-core algorithms for DVR of large unstructured grids. The first is a memory-insensitive rendering that requires only a small amount of main memory. The basic idea is to pre-process the files containing the vertexes and cells lists in order to avoid random access to the disk. Then each cell is transformed and traversed by a ray, the intersection is computed and stored when the ray enters and exits the cell. The sampled values are then sorted, so all the values concerning the same pixel are grouped together, in a sequential order, ready to be employed for the compositing phase. The second algorithm is an extension of the in-core ZSWEEP, which basically sweeps the data with a plane parallel to the view plane increasing the z coordinate. All the cells incident to vertices are projected onto the plane, the contribute they give

to each pixel is stored in a sorted list, the data structure adopted is similar to the A-buffer. When the sweeping plane reaches a target z coordinate the compositing is performed. The out-of-core ZSWEEP divides the data set in chunks, so only a subset is loaded into the main memory at each time. To further limit the amount of memory required, also the screen is subdivided into tiles, for each tile the chunks that are projected onto it are rendered.

Another open topic of research is represented by the time-varying and dynamic data set, since many scientific applications require to analyze the evolution of the data set over time. In this case the field values can change, but topological changes of the structure are also possible. While many solutions already exist for the regular grids, the visualization of time-varying unstructured grid is still an hard task, basically due to the enormous amount of data needed to be handled. Compression techniques have been developed to overcome this issue.

Finally, nowadays DVR is becoming practical also for the video-games industry. Exploiting the new GPUs features, such as the increased texture memory and the programmable shaders, it is possible to adopt DVR techniques to enhance volumetric effects that usually are approximated. For instance, fire explosions or lava spurts are achieved using particle systems with point sprites. Since it is possible to procedurally generate 3D texture and dynamically change the values, DVR can perform animated and non-repeating effects.

2.4 Software

2.4.1 OpenGL and 3D textures

OpenGL is a widely used application programming interface (API) for developing applications concerning 2D and 3D computer graphics. It is supported on several platforms such as Windows, Linux and MacOS and there are various language bindings, for example to C, C++, Python and Java. The API interface consists of over 250 functions, a program running OpenGL can call such functions to exploit the underlying graphic hardware. However if no hardware acceleration is available the functions can be emulated through software, of course decreasing the performance. OpenGL core API is rendering-only since it is operating system independent, thus it provides functions for the graphic output and does not directly support audio, printing, input devices and windowing system. Additional APIs are required to integrate OpenGL with such extra functionalities, as GLX for X11 and WGL for Windows or the portable libraries GLUT and SDL. The OpenGL rendering pipeline operates on both geometric and image primitives and includes features as lighting, geometric and view transformations, texture mapping and blending. Nowadays OpenGL allows also to create and run customized parts of the rendering pipeline using the OpenGL shading language (GLSL). GLSL is a high level language based on C and extended with types that are useful for graphics as vectors and matrixes. It allows to implement programs (vertex and fragment shaders) loadable on the new generation GPU, giving to the programmers a low level and flexible control of the rendering process. The Architecture Review Board (ARB) is a consortium made of software and

hardware companies leading the graphics market, it governs the OpenGL development since its introductions in 1992 when the version 1.0 was released. A single vendor can decide to add a specific function to the core set to better exploit its graphic board or to achieve specific goals, thus the extension mechanism is provided. The vendor extension can be promoted to become a generic extension if it is adopted by a group of vendor, furthermore it can become an officially ARB extension or even part of the core API.

Naturally OpenGL can be employed to implement volume rendering applications, further functions to achieve DVR via hardware acceleration are provided as part of the core API since the version 1.2 released in 1998. The basic idea is to extend the traditional 2D texture concepts to a volumetric data set or 3D texture. The main function to generate a 3D texture is:

```
void glTexImage3D  (GLenum  target,
                    GLint  level,
                    GLint  internalformat,
                    GLsizei  width,
                    GLsizei  height,
                    GLsizei  depth,
                    GLint  border,
                    GLenum  format,
                    GLenum  type,
                    const GLvoid * pixels)
```

The main differences with the 2D case are the target (here GL_TEXTURE_3D or GL_PROXY_TEXTURE_3D) and the presence of a third spacial coordinate, while the other parameters have the same interpretation. If 3D textures

are enabled, image data are read from pixels and a 3D texture with size width * height * depth * bytes_per_texel is created. The 3D texture can also be thought as series or a stack of 2D textures, where the extra spacial parameter (i.e., depth or r in texture coordinate) is used to select the proper 2D texture. In order to create a 3D texture inside a program the basic steps are:

1. Load or procedurally generate the image array
2. Get a name for the texture (`glGenTextures`) and bind to it
3. Set specific parameters (`glTexParameteri`)
4. Create the texture (`glTexImage3D`)

The steps to render the texture are:

1. Bind to the texture (`glBindTexture`)
2. Specify geometric primitives and texture coordinates (`glTexCoord3d` and `glVertex3d`)

The 3D texture will be mapped inside the geometry.

2.4.2 Aura and 3D textures

Aura is a high-level graphics library developed by the Free University of Amsterdam and released under the GPL license, it is implemented for both Windows and Linux. It is designed for non-expert scientists who intend to use modern 3D graphics facilities both for teaching and research purposes. Aura uses the widely known and employed APIs OpenGL and Direct3D to exploit modern hardware accelerations, and it provides an easy to use C++

interface. Aura is scene graph oriented, that is it furnishes to the programmer a set of functions to easily build and manage a scene graph. Thus a typical Aura application creates a scene graph where all the objects to be rendered are inserted, then the main loop provides the rendering steps and updates the scene. Of course the scene graph does not only contain the 3D geometry, but also the light sources and the camera positions. In order to move the objects and to achieve the animation of the scene, the affine transformations (i.e., rotation, translation and scaling) can also be inserted into the scene graph. Moreover, interactivity is obtained with a further component, the reactor. Whenever an event handler detects an incoming event (e.g., input from the mouse or the keyboard but also collisions among objects of the scene graph), a signal is sent to the appropriate types of reactors, which are in charge of deciding the actions to perform. A reactor joined together with a camera is called an avatar, it allows to interactively control the camera movements.

Each object has a state, such as material properties and textures applied on it, which can be controlled and changed. Object geometry can be directly specified using basic shape functions provided by Aura, or it can be loaded from an external file if it is complex . Aura supports many formats such as 3DS, LWO, OBJ, PDB and PK3.

Many Aura settings can be changed quickly, avoiding a new full compilation, writing the configuration file aura.cfg. This XML file contains tags that control the general behavior and many initializing values (e.g., the tag `Display` controls the resolution, the tag `Server` specifies if the rendering has to be performed locally or on a remote host, etc..).

Finally, Aura supports and allows programming on virtual reality envi-

ronments such as the CAVE and rendering on tiled displays.

Since Aura encapsulates the OpenGL functions, it supports the usage of 3D textures. The class `Texture` allows to apply a texture with one, two or three dimensions to an object inside the scene graph. The main functions provided to manipulate a texture are:

- The constructor to create the texture (`Texture(uint aformat, uint axsize, uint aysize = 0, uint azsize = 0)`). Besides the internal format, it is requested to specify the sizes. If only a dimension is stated, it is assumed it is a 1D texture. Likewise if only two dimensions are stated the texture is 2D, thus to get a 3D texture it is necessary to specify all the three dimensions.

- Load functions to generate the texture from an external file. The supported formats are: JPG, PNG, TGA and RGB.

- Query functions to retrieve information about the texture state (e.g., the size, the internal format, the byte-per-pixel). It is also possible to get the color corresponding to a specified pixel (`math::Color Pixel(uint x, uint y = 0, uint z = 0) const`).

- Write functions to create a copy power of two of the texture, to add an alpha value to the texture, etc.. The function `void Pixel(math::Color c, uint x, uint y = 0, uint z = 0)` allows to change the color to the specified pixel, it is very useful to procedurally create a texture.

Thus a volumetric geometry can be easily filled with a 3D texture, whether loaded from a file or generated from functions. For an instance:

```
Geometry *volume = new Geometry("test");
```

A geometry node is created.

```
VolumeCube *v = new VolumeCube(NUM_SLICES,Box(-1,-1,-1,1,1,1));
volume->AddShape(v);
```

The generic geometry node is filled with a cube.

```
volume->SetAppearance(new Appearance());
Texture *tex = LoadTexture("NoiseVolume.dds");
volume->GetAppearance()->AddTexture(fog);
```

The geometry node is enhanced with an Appearance object, to handle the state. A 3D texture is loaded from a file .dds and it inserted into the Appearance object.

```
AddGraph(volume);
```

Finally the geometry node containing the cube and the texture is added to the scene graph.

Chapter 3

Parallel volume rendering

3.1 Parallel rendering

3.1.1 Clusters of commodity components

Since the size of data set produced by scientific experiments and simulations is constantly increasing, standard volume rendering techniques have been adapted to run also on parallel hardware. The aim is still to achieve a high-quality and truly interactive rendering. The first solutions to be proposed were to exploit expensive specialised parallel machines, such as supercomputers or high-end graphics workstations. However, nowadays the high availability and the decreasing costs of commodity PCs and network devices make clusters of PCs a cheap and competitive alternative to parallel machines. The main advantages to adopt a cluster are:

- Low cost: commodity PCs produced for a mass market are cheap, especially when compared to the prices of custom-designed machines.

Furthermore, the price-to-performance ratio is in favour of PCs, also because they can be equipped with powerful and cheap graphics accelerator. It is worth noting GPUs for PCs have a development cycle that is short compared to customised hardware (six months against one year or more), and their performance improving exceeds the Moore's law. Moreover, standard APIs as OpenGL allow to update and replace the hardware components easy.

- Modularity: the off-the-shelf components are combined together using the network devices and they only communicate by using the network protocols. Thus it is possible to easily add or remove PCs, or to combine heterogeneous machines together.

- Flexibility: the cluster is not constrained to be employed only for rendering purposes, but it can also accomplish other tasks. It can act also as a server and provide rendering services to a remote host.

- Scalability: increasing the number of PCs linearly rises the aggregate computation, storage and bandwidth capacity. Since each component does not share its CPU, memory, bus and I/O system the scalability is better compared to a customised, shared memory architecture.

3.1.2 Molnar's taxonomy

In the past several classifications were proposed for parallel volume rendering algorithms, however comparisons were hard to perform because each analysis tends to focus on unique features of the algorithms. In the next sections the taxonomy proposed by [24] is presented, it is widely used and it overcomes

comparison difficulties, all the parallel algorithms are evenly classified into 3 main categories: sort-first, sort-middle and sort-last.

Basically the standard rendering pipeline can be thought of as made of two main parts, the first concerning geometric operations (transformations, lighting, clipping, etc.) and the second concerning rasterization (scan-conversion, shading, texture mapping, etc.). In the parallel case the geometric and raster primitives are spread among different processors. The main issue is to understand how each primitive contributes to each pixel, thus it is the problem of sorting primitives onto the screen.

Each algorithm for parallel volume rendering is classified according to this scheme: if the sorting takes place at the geometric stage it is sort-first, if it occurs between the geometric and the rasterization stage it is sort-middle, otherwise if it occurs at the final rasterization stage it is sort-last.

3.1.3 Sort-first

Sort-first algorithms have the main feature to distribute geometric primitives at the beginning among the the processors. The final display is splitted into disjointed tiles and each processor is responsible only for a tile (for this reason sort-first is also referred to as an image-space approach), thus the main task is to determine if the primitives which the processors have received belong to its tile or they do not. In order to determine the primitive belonging to tile pre-transformations are performed, usually calculating screen coordinates of the primitives bounding boxes. If a primitive does not belong to the processor a redistribution step has to be accomplished. The primitive is sent through a communication channel to the proper processor. Of course

the redistribution step introduces an overhead, proportional to the number of primitives initially sent to the wrong processor. If only a frame has to be rendered, most primitives will be assigned to the wrong processor, with a related expensive redistribution step. To the contrary if multiple frames have to be rendered and if there is an high frame-to-frame coherence (that is, if the 3D model tends to locally change slightly between one frame and the next), the redistribution overhead can be strongly reduced. There is another overhead of sort-first that does not appear on the uniprocessor case. It is when the same primitive overlaps several tiles and it has to be assigned to multiple processors. If the primitive bounding box center falls into the tile border it will overlap 2 tiles, if the center is on a corner the primitive will overlap 4 tiles. Since each primitive has the same probability to fall anywhere within the tile, the overlapping cases can be reduced if the tile size is big compared to the bounding box size.

Sort-first can be prone to load-imbalance. A first case occurs when primitives are well spread among the nodes but they require a different amount of work to be rendered. A second case occurs whereas primitives are not equally distributed on the screen, thus a tile contains much more primitives then another one. A way to balance the amount of work is to reduce the tile size and to assign several tiles to the same processor. This can be done whether statically or dynamically. Static load-balance is simpler to implement but cannot be optimal for every data set, on the other hand the dynamic one increases the algorithm complexity but allows to achieve better results. Another disadvantage of sort-first approach is that in order to exploit the frame-to-frame coherence a processor has to retain primitives, to avoid a new initial random

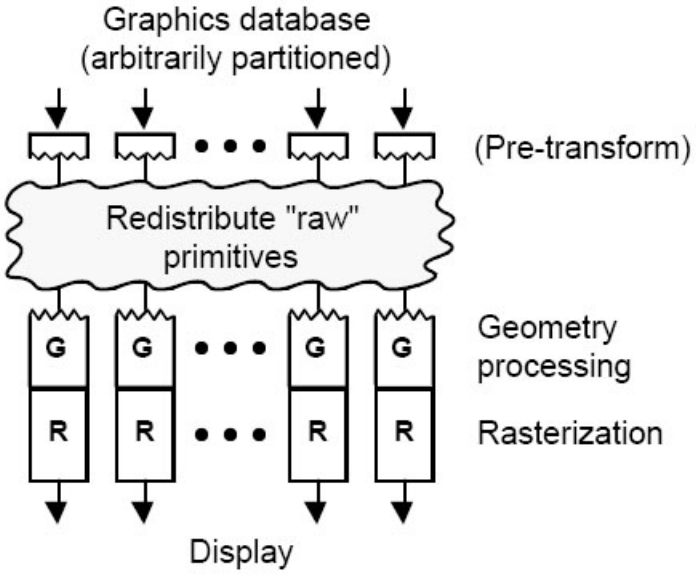

Figure 3.1: Sort-first. Source: [24]

distribution. Thus the algorithm has to provide appropriate mechanisms to handle the retained primitives.

3.1.4 Sort-middle

Sort-middle performs the sorting between the geometry processing and the rasterization. Since many systems compute these steps on different processors, this is a natural place where to accomplish the primitive sorting, however a sort-middle approach can be performed also if both the geometry processing and the rasterization take place on the same physical processor. At the beginning the primitives are randomly distributed among the nodes. Each node does the full geometry processing until primitives are transformed in screen coordinates. Transformed primitives are then sent to the appro-

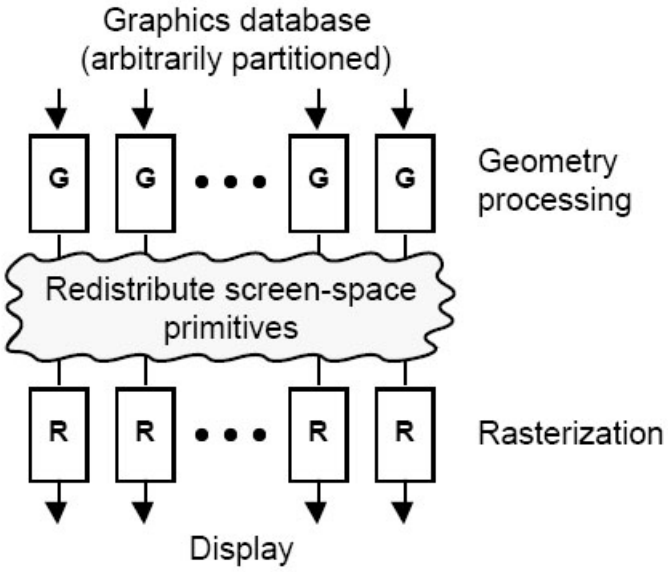

Figure 3.2: Sort-middle. Source: [24]

priate processors, that is, as in sort-first, the processors responsible of their display tile.

The main overheads occur during the redistribution phase, because primitives have to be sent through an interconnect network. Moreover, some primitives can fall into more then one display tile, rasterization efforts have to be at least doubled for such primitives.

Load-balance issues concern the object assignment and the unevenly distribution of primitives onto the screen. To guarantee a fair object assignment, geometric primitives can be handled with hierarchical data structures. To reduce load-imbalance during the rasterization the main technique is to dynamically assign display tiles to each processor and to use smaller tile (however, this can increase the number of overlaps).

3.1.5 Sort-last

Sort-last distributes primitives to each processor without any concern about the display tile the primitives fall in. Each processor performs a full rendering, both the geometry processing and the rasterization. Finally, once fragments are generated, pixels are sent to compositing processors, which are responsible for resolving the visibility of pixels and to display the result. Since every processor sends its pixels in the final phase, the network must provide a high bandwidth in order to handle a high data rate. The sort-last approach can be splitted into two main categories, the SL-full and the SL-sparse. SL-full transmits the full image rendered by a processor to the compositing processor, whereas SL-sparse sends only the pixels produced by the rasterization, the aim is to minimise the communication work-load.

Sort-last does not introduce overheads during the geometry processing and the rasterization, as in the uniprocessor rendering case. There are overheads during the pixel redistribution phase, where each processor has to send its pixels to the proper compositing processor. Here the two approaches SL-sparse and SL-full show their differences. SL-sparse sends only the generated pixels, thus the data sent over the network depends on the part of the scene assigned to the processor, but it is independent of the number of processors. SL-full is simpler to implement since the pixels messages sent are very regular, its complexity does not depend on the contents of the scene, but it depends on the number of processors and on the display resolution.

Load-imbalance occurs when the distributed primitives require a different amount of work to be rendered, as for sort-first and sort-middle. Each processor renders the entire scene, thus there are no overlapping problems.

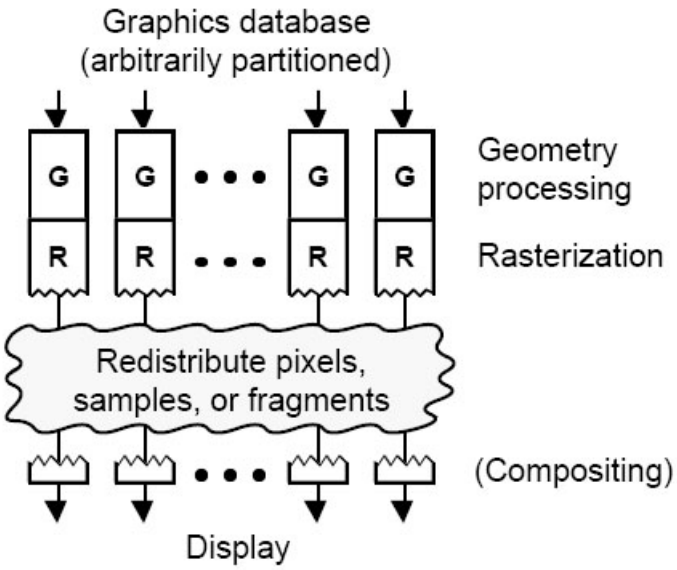

Figure 3.3: Sort-last. Source: [24]

SL-sparse presents load-imbalance if a processor sends more pixels to a compositor then another one. This issue can be overcome assigning to each compositor interleaved arrays of pixels. Of course SL-full does not present this type of load-imbalance, a full scene is always sent to the compositors.

3.2 State of the art

3.2.1 Complex hierarchical structures

Although for several years the most used approach was the sort-middle, recently both sort-first and sort-last are commonly employed. The reason is closely connected to the popularity of clusters of PCs. Customised hardware supports provide high communication bandwidth among nodes and fast access to the results of the geometry processing, thus a sort-middle approach is

perfectly suited. By contrast, clusters do not usually provide such features, thus sort-first or sort-last are generally preferred.

Efforts have been done to improve the sort-last, specially concerning the load-balance issue. Trivial approaches divide the data set into sub-volumes of the same size, then the sub-volumes are sent to the rendering nodes. The load-balance is guaranteed at a task level, because each node has to perform the same amount of work. However a deeper analysis shows that the contribution of each node to the final image can vary according to the viewpoint and the inner structure of the sub-volumes. Other approaches tend to employ parallel octrees to allow each rendering node to skip empty regions. The global octree is created and sub-branches are assigned to different nodes. A first processor reads data from an input file and sends them to the appropriate processors, which will provide the split-down phase (the octree recursive generation from the root to the leaves) and the final push-up phase (information about the octree nodes are sent backwards from children to parents). Links among octans (i.e., octree nodes) can be either local or off-processors. Despite this technique can speedup the rendering time on each processor, load-balancing remains static.

[18] propose to use composed hierarchical data structure to handle the data set in order to achieve not only a task parallelism but a data parallelism as well. The first step consists creating an octree, as already seen for other standard approaches, and then to re-organise the data set using an orthogonal BSP tree. A BSP tree (the acronym stands for Binary Space Partitioning) is a tree where each node has no more than two children, and where each sub-tree is obtained by splitting the space with an hyperplane (i.e., a plane

in the tridimensional case). The BSP tree is generated starting from the root node, which contains the whole data set, and then adopting a splitting plane to divide the data into two disjointed sets. The procedure is recursively applied to each sub-tree until a certain threshold is reached. At the end every siblings pair is sorted with respect of the splitting plane. This is very useful for rendering purposes because it has the same effect of a precalculated painter's algorithm.

A BSP tree satisfies the three main criteria to support a dynamic data set: capacity to move the camera, to add and to remove an object. At least the first criterion is easy to implement, it is only necessary to change the traversal order. A BSP tree used together with an octree is useful to achieve a true data parallelism, the octree allows to skip empty regions and the BSP tree to fairly partition the relevant sub-volumes. A crucial phase is represented by the choosing of the splitting planes, in order to get an easy partitioning scheme. Adopting orthogonal, axis-aligned planes may be a good strategy: first an axis is chosen, then all parallel planes are tested since a fair partitioning of octree sub-volumes is reached, finally the procedure is repeated for a new axis, until the number of BSP tree leaves is the same as rendering nodes. Despite so far the above ideas are employed only for static data set, they may be extended to the dynamic case as well, allowing re-partitioning mechanisms.

3.2.2 Out-of-core and multi-threaded approach

Of course out-of-core techniques seen for the uniprocessor case also apply to the parallel one. For instance, [5] extend the iWalk system originally thought

for a single PC to a cluster of PCs. The basic iWalk concepts are to build an on-disk octree representation of the data set performing a pre-processing phase. At each step only a portion of the data fitting into the main memory is loaded and processed. The resulting octree node is then stored in a different file, while an additional file is also created, the hierarchical structure file, which contains the spacial relationships among nodes and other information required for visibility tests. At run time, iWalk uses a multi-threaded approach. A rendering thread is in charge of detecting which octree nodes are visible, and then to forward the request to the fetch threads, responsible for loading the nodes from disk to the memory. A look-ahead thread is used to try to predict the next nodes required to be loaded. If no fetch requests are pending prefetch threads can load into the memory the nodes requested by the look-ahead thread. The parallel version of the iWalk system adopts a sort-first approach. A client node handles the user inputs and notifies the viewpoint changes to the rendering nodes. Besides, the full data set can be stored directly onto the local hard disks of the rendering nodes or can be stored onto a shared hard disk accessible through the network. Nevertheless, each node loads on demand the required sub-volumes into the local memory and performs the standard iWalk techniques. The only differences with the uniprocessor case are: occlusion culling (each node performs the test only according to the local frustum, since each display tile is independent), user input (it is sent through sockets by the client node), synchronisation among nodes (the MPI library is employed).

3.2.3 Hybrid sort-first/sort-last approach

Hybrid approaches combining the three main techniques have also been developed, in order to reduce overheads and speedup the performance. [30] propose an hybrid sort-first/sort-last algorithm and give several comparison test results with the standard sort-first and sort-last. The algorithm core ideas can be summarised in three phases:

- Phase 1: it consists basically in the pre-processing steps. Here the more relevant aspect is that the data set is partitioned both in the 3D model space and in the 2D screen space. The aim is to assign a disjoint group of geometric primitives and a display tile simultaneously to each rendering processor, attempting to overcome the overhead due to overlapping primitives for sort-first and the heavy pixel traffic for sort-last. The partitioning is accomplished with a recursive binary partition and a two-line sweep. At first, the longest screen axis is selected and the two sweep lines are chosen perpendicular to the axis, both placed at the opposite sides of the area to be partitioned (it is noteworthy that this partitioning scheme is view dependent). The line at the left side can be moved to the right direction and vice versa, each line has associated a group of objects, at the beginning empty. At each step the line with the smallest group is moved. When an unassigned object is encountered it is inserted into the line group. This process halts when all the objects have been assigned, then a little swath between the two lines usually remains. The swath represents an overlapping areas between the objects of the two groups, it will require a compositing phase after the rendering phase. The partitioning

procedure goes on using a new set of sweep lines perpendicular to the previous ones, it stops when the number of tiles and group is the same as the rendering processors.

- Phase 2: each group of objects and the corresponding tile is assigned to a processor. The rendering is performed and then the color and the depth buffer are read from the graphics board and loaded into the main memory. Each processor sends its overlapping area to the appropriate processors, and at the same time receives pixels from the other processors. Compositing is achieved using depth information. Pixels redistribution is done with a peer-to-peer communication model among the nodes of the cluster, the communication bandwidth required is lower compared to the pure sort-last approach.

- Phase 3: finally each processor sends its tile to the display node. Now all the tiles are complete, since the phase 2 has provided the compositing step in order to get the exact result also in the overlapping areas. The processors only have to send the color buffer because no more compositing is needed (the tiles are not overlapped). Moreover, every pixel is sent exactly once to the display node.

This hybrid approach gives good results (i.e., interactive frame rate) for medium sized clusters, that is clusters with a number of nodes ranged between 8 and 64. Increasing the number of nodes shows that the amount of work of the rendering nodes tends to decrease, the display node work remains almost constant, while the client node work grows. The latter is due mainly to the pre-processing algorithm, since the partitioning complexity depends

on the number of rendering nodes. So a large cluster with more then 64 nodes has its performance constrained by the client node. However cluster of such dimension are not common nowadays. Remarkably, the hybrid approach performance is better than the ones of the traditional sort-first and sort-last approaches, because its design is done to reduce their main overheads. Sort-first overlapping overhead is small for the hybrid approach, as the intense pixel traffic of sort-last. Moreover, the hybrid approach overheads for the rendering nodes tend to decrease augmenting the number of nodes. The study of the hybrid approach with an increasing screen resolution shows its performance is still better than the other two. However the impact of pixel write and read operations is significantly big, as for sort-last. Sort-first is probably the best solution for very high-resolutions. Finally, it is interesting to investigate how the number of objects (keeping constant the number of polygons) affects the overall performance. On the client side, increasing the number of objects (that is, the granularity) means a major work of partitioning. On the rendering processors side, the overheads tend to reduce. This fact points out the importance of improving the client partitioning scheme.

3.3 Software for parallel rendering

3.3.1 Chromium

Chromium is an open source library which extends the standard OpenGL API to specifically allow parallel rendering on clusters. It was first developed by the Stanford University as a further development of the WireGL project. It was first released on 2001 and it runs on several platforms such

as Windows and Linux. Chromium replaces the OpenGL library and provides almost every OpenGL function, so it is transparent to an application using OpenGL. Moreover, it provides extra functions for the parallel rendering. The key idea is to give basic and highly customisable building blocks to the programmer without constraining choices to a specific architecture. The three basic employees of Chromium are:

- rendering a standard application on a high resolution display, multi-screen display or tiled mural display (power wall).
- sort-first and sort-last parallel rendering.
- filtering and manipulation of OpenGL commands sent from the application to the rendering nodes.

An OpenGL application performing a local rendering can be thought as a graphics stream generator. It sends a stream of commands to the graphics system. Chromium basically works intercepting the application OpenGL commands with the application faker library (crappfaker), which then send the commands to a stream processing unit (SPU). Several SPUs can be combined together, creating a SPU chain. A SPU can be used to filter or change a specific function call and leave the rest unchanged. For example, a SPU can be used only to call `glPolygonMode` and set a wireframe rendering. It does not change other function calls, but it eventually discards `glPolygonMode` calls to preserve the wireframe setting. A SPU chain initialisation occurs in a back-to-front order, beginning with the final SPU. The idea is that every SPU declares a list of functions it implements. If a SPU does not change a certain function call, it uses the function pointer of the downstream

SPU, thus it avoids to introduce overheads. Specific SPUs are provided to pack and send the commands to the network nodes (crserver) of the cluster. Typically a SPU chain ends with a Render SPU, which performs the final rendering. There are about 20 standard SPUs: the Render SPU (it pass the OpenGL commands to the hardware support), the Pack SPU (it packs commands into network messages), the Tilesort SPU (used for sort-first), the Readback (used for sort-last), FPS SPU (it measures the frame rate), etc. SPU implementation is object oriented, new SPUs are generated using the inheritance mechanism. For example, the Readback SPU is derived from the Render SPU, all the functions but the `SwapBuffers` do not change. Instead `SwapBuffers` reads the current frame buffer with a `glReadPixels` and then send it to the next SPU in the chain with `glDrawPixels`, in order to perform a sort-last compositing.

The mothership is a python program responsible of the system configuration. It describes the global settings, the SPU chains loaded on each node and the nodes arrangement. The latter is achieved giving the directed acyclic graph (DAG) of the nodes, where each node in the graph represents a cluster node and each edge represents the network traffic among the cluster nodes. All the Chromium components query the mothership to get information about other components and global settings.

3.3.2 Parallel rendering with Chromium

The main SPU concerning the sort-first parallel rendering is the Tilesort SPU. It is usually employed to render an unmodified application on a mural display, to replicate a rendering to several screens and to divide a high res-

olution display into tiles. The application node contains the Tilesort SPU, which sends the OpenGL commands to the network nodes. The Tilesort SPU is responsible to send the geometric primitives to the appropriate nodes and to keep track of the OpenGL state. If something of the state changes, Tilesort SPU sends the updates to the nodes. Several options are available, for instance the size of the tiles can be either uniform or different on each node, furthermore the same commands can be sent in broadcast to every node. Since the Tilesort SPU consumes many computational resources and it is ran on the node which already hosts the application, a powerful hardware support is required. The network is also required to give enough bandwidth, strategies can be adopted to overcome limitations, such as display lists and vertex buffers.

Sort-last rendering may be achieved with three basic SPUs: Readback SPU, Binary Swap SPU (it distributes the compositing process among the nodes) and the Zpix SPU (it performs an image compression). Basically, the color buffer and the Z buffer are read and sent to the compositor node (or to a peer in the binary swap case). Buffers are then composited together using the OpenGL alpha-blending and the Z-compositing via stencil buffer (further details will be provided in chapter 4). Since Chromium is thread-safe, multiple threads can run on the same host in order to speed up the sort-last algorithm and take advantage of the host underlying hardware, such as multiple graphics pipelines.

The sort-last scheme is also important to highlight the need of synchronisation primitives. When several application nodes send their buffers to the compositor crserver, it may be necessary to blend the buffers in a certain

order, or to clean and swap the compositor buffer only once. These issues can be accomplished adopting well-known primitives, such as semaphores and barriers. The latter is used to guarantee all the application buffers are already blended before swapping the current buffer. Semaphores can be used to constrain to a specific order. For instance, the N application buffer waits to be blended the N -1. When N -1 is done, a signal is send and the N buffer can be processed. All these primitives are implemented on the compositor crserver. That is, if an incoming SPU is waiting, the crserver does not advance its execution until the SPU is unblocked. Another issue is represented by the input handling on the crserver. An additional component, CRUT, is required to track the input and send it back to the application node, which can update and render the new scene.

3.3.3 Parallel rendering with Aura

The Aura library supports the parallel rendering on clusters of PCs. The Aura design differs significantly from the Chromium design, at least for two reasons:

1. Aura adopts an ad hoc interface, so it is not transparent to an application. An application must be rewritten to take advantage of the Aura features, while the intercepting mechanism allows Chromium to be used without modifying the off-the-shelf application. However, the Aura approach permits to optimise the application according to the rendering environment.

2. Aura adopts a retained mode strategy, in contrast to the Chromium

immediate mode. On the one hand, Aura replicates the application on each node, allowing local changes (e.g., the frustum area) in order to achieve the appropriate rendering. Input events are broadcast from the client node to the rendering nodes, to keep their state consistent. Synchronisation among the nodes must be also provided to ensure frames alignment. The main reason behind the retained mode approach is related to the performance, despite the major drawback is represented by the redundancy. Problems can arise specially if the application reads data from a database, since every node has to accomplish the same operations.

On the other hand, Chromium avoids the replication issue, each stream of commands is sent only to the node that will actually performs the rendering. However, Chromium approach consumes a lot of network bandwidth.

To achieve the parallel rendering Aura uses three kind of components: master, slave and the Aura Rendering Service (ARS). The master is an Aura application which demands the rendering to at least another application, called slave, usually on a remote machine. Thus the scene graph generated and controlled by the master is sent throw the network to the slaves, which perform the rendering. The ARS is required in order to manage the connections among the master and the slaves. The ARS process should be the first to be run followed by the slaves, which try to establish a connection with the ARS to notify their presence. The ARS assigns to each slave a ranking number, so it is possible to identify each slave. Whenever a master decides to run an application as remote, it asks for the ARS the address of a slave,

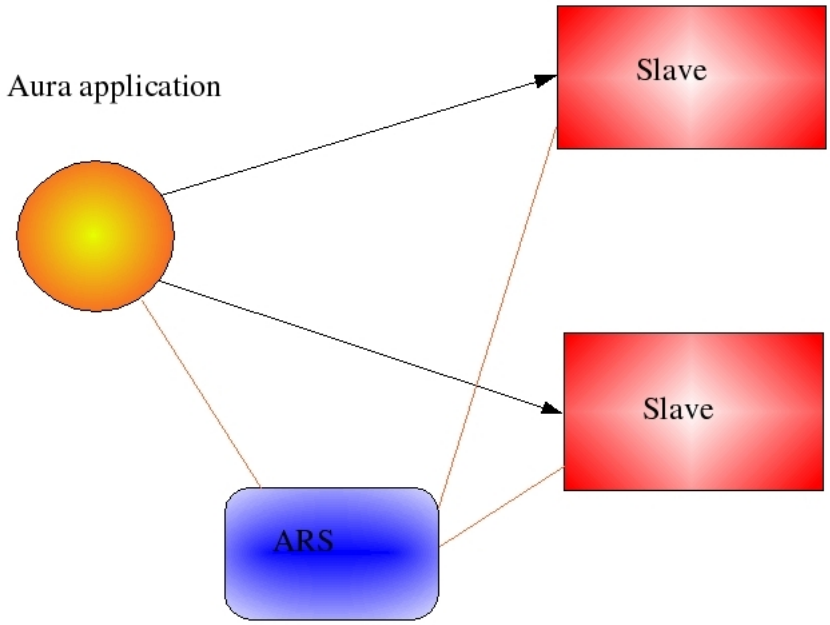

Figure 3.4: Logical structure of the parallel rendering in Aura

which acts as access point. Once the master obtains the address, it establishes a direct connection to the slave and starts to send to it the required messages. The access point slave will propagate the incoming messages to the other slaves.

Chapter 4

Implementation of a sort-last rendering in Aura

4.1 Code design

4.1.1 Aura internals

Implementing a sort-last rendering in Aura means adding additional components to the basic Aura library. A first step consists exploring the library itself, since a wide knowledge of the internal structure is necessary. First, it allows to reuse existing portions of code to create the extra components. Furthermore, it gives insights on how to shape the component design, that is which components to implement and where to place them.

Aura is strongly object-oriented, coherently with its high level and scene graph oriented approach. The best way to understand Aura is possibly to understand its classes and what is their purpose. In order to avoid an endless

list, an overall view is given. The classes are grouped together following a logical scheme (i.e., classes concerning the same aspect are grouped together). The main classes adopted in the present work are highlighted and deeper described.

- *Core classes*. This set of classes is required to create a basic Aura application. The *Environment* class reads the Aura configuration file and performs the initial set up. It creates the rendering context (*RenderingContext*) according to the operating system. On a Linux platform (as the cluster used to run the sort-last rendering), the rendering context basically encapsulates the glx functions, which are in charge of setting up the frame buffer and accomplishing operations such as the swap buffer and the clear buffer. The *Environment* class is also responsible for setting the viewport (*Viewport*). The viewport can be of several and customisable types. The three main choices are the default viewport, for stand-alone applications, and the remote and slave viewports, for the master and slave sides of a parallel framework respectively. The *viewport* class is also very important because it is here where the Aura processors of the application are specified. The event handler and the scene graph classes, even used when a new environment is created, will be treated below. Finally a special mention for the *Application* class, which uses the above classes to instantiate a new application. Although a programmer is not constrained to use this class to create its own application, nevertheless it represents a standard and time saving way to use the Aura features. It initialises the environment and it builds up the scene creating the scene graph and setting the application behav-

ior. The virtual function `Step` is usually adopted to provide the scene animation.

- *Interaction classes.* These classes are related to the managing of input events. Basic input handling as the keyboard and the mouse are provided. Besides, more complex interactions are managed, as the object selection and the collision detection. Is is also possible to attach a camera to a certain kind of input, in order to move the camera along the scene. For our purpose, the default Aura events were used, that is, to translate and rotate the scene using keyboard inputs.

- *Scene graph classes.* Here the classes concerning the scene graph creation are grouped. *Scene* is the root node, and it performs the scene global settings. *Graph* is the generic scene graph node, where all the other classes are inserted. These are: *Shape*, which models the geometry of an object, *Appearance*, which models the state of the geometry (*Shape* and *Appearance* are grouped together in a class *Geometry*), *Light*, which sets the light parameters, etc. For the present work a minimal scene graph was adopted, containing a *Geometry* object made of a parallelepiped (provided by *VolumeCube* class) filled with a 3D texture (*Texture*) defined in its *Appearance*.

- *Processors classes.* The processors allow to customise the rendering pipeline of an Aura application. This is a powerful mechanism, since several processors can be combined together to achieve more complex results (despite there are not guarantees that two different processors put together have a consistent behavior). This mechanism resembles

the Chromium SPUs, despite the granularity is different. Chromium SPUs act on OpenGL function calls, while Aura processors act on scene graphs. Examples of Aura processors are the *CullProcessor*, to perform the object culling, the *OpenGLProcessor* to perform the scene graph rendering, a *TransmitProcessor* and a *ReceiveProcessor* to send and receive a scene graph to a remote node in a parallel rendering. The feature of the Aura processors was the central one adopted to implement the current work and will be described further below (4.1.4).

- *Utils classes*. This set of classes gives additional tools to the programmer, extending the basic set. For example, an xml parser is provided, because it is required to read the configuration file. An image loader is also provided, and a log tool useful for debug purposes. Here the *Socket* class was heavily exploited, since it is required to exchange the messages among the nodes. Using the socket mechanism a connection between two nodes is established. The connection is then employed to send and receive messages (based on the Aura *Message* class). In this work the messages contain frame buffers or portion of frame buffers to be composed together. Utils classes were also used to implement the compositor node (see 4.1.7).

4.1.2 OpenGL buffers

Since Aura exploits the OpenGL API, it is also necessary to understand the main features of OpenGL concerning the current work. These are mainly two: the OpenGL rendering and the OpenGL buffers. In our work the former

concerns mostly how OpenGL manages and renders 3D textures. This is accomplished locally on each node, so it has not a direct impact on the overall design issue. The latter is more decisive, since the sort-last approach requires the nodes to read, exchange and compose their buffers.

An OpenGL buffer is a rectangular array of pixels, usually stored on the GPU memory. Each buffer gives some information on how to display the rendering result onto the screen, or how to manipulate it. All the buffers enabled for a certain OpenGL context are referred to as frame buffer. However in the rest of the chapter the notion of frame buffer will be slightly overloaded and it will refer only to the color and z buffer together, which are the buffers mainly used it this work, unless differently specified.

A basic OpenGL implementation usually provides:

- *Color buffer*. It is the only one directly visible. It provides color information for each pixel, so it contains what is actually displayed onto the screen. If the stereoscopic view is enabled, there are two color buffers, left and right. Moreover, if the double buffer is enabled, there are also a front and back color buffer. Each pixel can contain color-index or RGBA values.

- *Depth buffer (also called z buffer)*. It stores depth information for each pixel. Its main aim is to perform a hidden-surface removal on a per pixel basis. When the depth test is enabled, a pixel passes the test only if its depth value according to the current viewpoint is lesser than the one written onto the depth buffer. However, the depth buffer is not constrained to only perform this task, since the depth test function can be changed.

- *Stencil buffer*. The main idea behind the stencil buffer is to allow the user to restrict the drawing process onto the color buffer only to certain areas. This filtering is based on the current stencil values. The stencil test function and operation allow complex types of filtering.

- *Accumulation buffer*. It is used to accumulate RGBA images, so it is similar to a color buffer, although the per pixel resolution is usually higher. For instance, it is used to perform anti-aliasing by supersampling the image, or motion blur of images accumulated at different times.

Besides these standard buffers, OpenGL allows the user to create optional buffer, in order to achieve special rendering or ad hoc operations. The per pixel buffer resolution can vary according with the underlying hardware.

4.1.3 Read and write operations on buffers

The most common way to modify the buffers content is to render a scene. However, OpenGL gives ad hoc functions to directly modify the buffers. This is especially important for a sort-last rendering, because after the rendering each node has to read its own frame buffer and to send it to the compositor node (or to a peer with the binary swap algorithm). Furthermore, the compositor node works only by blending together different frame buffers, that is it does not perform any primitives rendering. The two main functions are glReadPixels (read operation) and glDrawPixels (write operation).

```
void glReadPixels(
    GLint  x,
```

```
    GLint  y,
    GLsizei  width,
    GLsizei  height,
    GLenum  format,
    GLenum  type,
    GLvoid  *pixels)
```

Its aim is to read pixels from a specified buffer. The `x` and `y` parameters specify the first pixel to be read, that is the lower left corner of the wished pixel block. `width` and `height` indicate the pixel block size, in pixel. `format` and `type` indicate the pixel format (e.g., RGBA) and type (e.g., byte), respectively. Finally, `pixels` is a pointer to memory location where the pixels block is stored.

```
void glDrawPixels(
    GLsizei  width,
    GLsizei  height,
    GLenum  format,
    GLenum  type,
    const GLvoid *pixels)
```

The write function parameters are almost the same as for the read function. The only remarkable difference is that the parameters required to specify the first pixel are missing. The main reason is to allow a more complex mechanism to specify the write starting point, in order to allow 3D write operations. Thus the `glRasterPos` function is used, it sets the current raster position (i.e., the write operation starting point) using 3D coordinates. Since

the use of `glRasterPos` can be too complex if the aim is only to perform a 2D write onto a buffer, recent OpenGL versions also allow the `glWindowPos` function. It allows to specify the write starting point using only 2D coordinates.

4.1.4 Project structure

The rationale for the project structure is to optimise the rendering performance and to exploit the already existing Aura structure. The latter is almost obvious, since the present sort-last rendering does not represent a stand alone application but it is a library extension. Summarising, a standard Aura parallel application is represented by a master node and several slaves (at least one slave), coordinated by the ARS (see 3.3.3). The master sends its whole scene graph to the slaves, which are in charge of performing the rendering. In our case, the master is an application having a minimal scene graph, mainly made of the 3D texture to be rendered. A first approach could be ask for the master to partition the data set (i.e., the 3D texture) and to send each chunk to a slave. However, the Aura parallel framework always sends the entire scene graph, that is each slave receives the whole 3D texture. Keeping the Aura behavior, as in our intent, means to take advantage of this feature. So the master side is not changed, only an application is implemented which has a scene graph basically made of a 3D texture and which sets up the scene (e.g., setting the blending coefficient). The application can be run locally or can act as the master, thus sending the texture to the slaves. At this point, each node has to perform the actions concerning the parallel rendering. The natural place where to add the new components

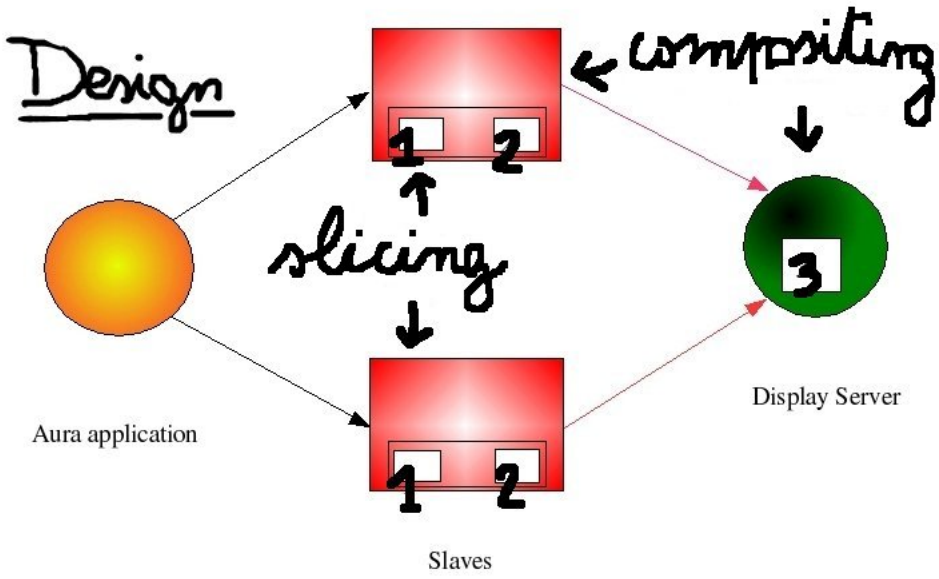

Figure 4.1: Design: the SortLastProcessor (1), the SendBufferProcessor (2) and the DisplayServer (3). Black arrows indicate the scene graph transmission, while red arrows the pixels transmission to achieve the compositing

is among the Aura processors on the slave side.

Each default slave has at least three Aura processors: one to receive the scene graph, one to forward the messages to the other slaves, one to render the scene. What is needed for a sort-last rendering is: each slave renders a different sub-volume (first additional Aura processor) and sends the frame buffer to a compositor node or a peer (second additional Aura processor). The rationale for using extra Aura processors is that this is the favourite Aura mechanism to provide customised components. Furthermore, extra Aura processors are easily inserted into the slave program. The default slave remains unaltered, with the exception of two extra Aura processors added to it, one before the rendering Aura processor and one after. The former is

called *SortLastProcessor*, it determines the sub-volume of the 3D texture to be rendered. The latter, *SendBufferProcessor*, reads and sends the image on the local buffer to a peer (if the binary swap algorithm is on), or directly to the compositor node. The compositor node (called *DisplayServer*) represents the last additional component added to Aura. It is a light-weight process which composes and shows the final image.

4.1.5 Aura SortLastProcessor

The SortLastProcessor has the task to re-size the slave 3D texture. Each slave has to render a different chunk, so the aim is to assign a different portion of the 3D texture to each slave (slicing phase). The constrains are: to ensure load-balance among the slaves, all the 3D texture has to be rendered (i.e., the chunks have to cover the whole texture). The former implies we want to split the texture into chunks of the same size. The latter means the chunks give back the original texture when together. Both these constrains depend on the number of the slaves. A trivial solution could be to allow the slaves to cooperate exchanging messages. However, a standard Aura slave gets information about its identity number (rank) and the total number of slaves from the ARS. The master sends to the slaves the viewport position. The user input on the master is also propagated to the slaves. Thus, each slave already knows enough about the global environment to be able to split locally its own copy of the 3D texture. Load-balance can be achieved using the information concerning the total number of slaves. The consistency is guaranteed using the rank number. Let n be the number of slave, the texture is splitted into n chunks. Each chunk has the same size and has a rank. Each

slave is in charge of rendering the chunk corresponding to its rank.

The SortLastProcessor basically detects the 3D texture into the received scene graph and re-sizes it according to the above considerations. The smaller texture (i.e., the chunk assigned to the slave) remains inside the scene graph, which will be processed by the next Aura processor, typically the rendering one.

4.1.6 Aura SendBufferProcessor

After the Aura rendering processor has accomplished the rendering, each slave has its own partial image onto the frame buffer. Each slave has to send its frame buffer to the compositor node (normal mode) or to a peer (binary swap mode). Both modes require to read the frame buffer with `glReadPixels`. If the normal mode is on, all the frame buffers are sent to the compositor node. It means a connection via Aura socket is established, and a message containing the frame buffer is sent. Afterward the SendBuffer-Processor duty is finished. Otherwise, if the binary swap algorithm is used (for full details see 4.3.4), the task of the SendBufferProcessor is more complex. At each binary swap step, each node has to determine its peer. Then a socket connection is established, and the two peers can exchange their frame buffers and perform locally the compositing. As for the SortLastProcessor, the best way to accomplish this task is avoiding extra slave communications. The only communications allowed are those to exchange the buffers. Once again, each slave exploits locally its global information: the number of slaves and the slave's own rank. Using this information each slave can determine: which is its peer at the current step, which frame buffer portion it has to

send, where it has to compose the incoming frame buffer.

4.1.7 DisplayServer

The compositor node is designed to display the final image onto a PC screen. Since its task is very specific, the idea is to use a light application instead of a typical Aura one. This means we do not use the scene graph or any other high level Aura feature. The compositor node simply waits for the incoming frame buffers. If the binary swap mode is active, its task is even simpler, since the compositing already happened on each node. It receives tiles of the final image, and it has only to write them onto the local frame buffer using the `glWritePixels`. The relative position of each tile is determined according to the rank of the slave sending the tile. If the more inefficient normal mode is active, the compositor node has also to perform the compositing. In this case every node has to send the whole frame buffer, so there is not any tile position to determine. In both cases, the compositor node has to keep a state about the number of frame buffers received. If it is the first one, a clear operation on the local buffer has to be accomplished. If it is the last, a swap buffer operation is necessary to display the final image.

4.2 Slicing and compositing

4.2.1 Slicing algorithm

Several approaches are suitable to split the original 3D texture. The one adopted was chosen because it is quick and it allows to optimise the following

compositing phase. The idea is that each slave is always in charge of the same chunk. So after the 3D texture is partitioned the first time, it is not required to rearrange the chunk division any more. The 3D texture is partitioned along the three main axis, without regarding to the current viewpoint. Both the volume geometry and the texture are reduced and rearranged. Despite the former is trivial to implement, the latter is more difficult since only the voxels falling into the selected sub-volume have to be kept. During the slicing phase all the chunks are also sorted using their own centroid. A centroid is calculated for each chunk. The distance between each centroid and the current viewpoint is determined. All the distances are compared and each chunk is classified according to the distance comparison result. That is, the chunk with the farther centroid gets the smallest coefficient, the one with the second outer centroid gets the same coefficient increased by one, and so on. The coefficients are a way to sort the chunks and, after the rendering, to sort their respective frame buffers. Since all the slaves share the viewpoint information, each slave classifies its chunk consistently with the other slaves. It means despite the algorithm is run locally by each slave, they all get the same result. This method is not generalisable to every 3D context, because it takes advantage of the simple and convex structure of the volumetric data set. The rationale for this sorting classification will be better pointed out in 4.2.5.

4.2.2 Alpha compositing

The compositing phase consists of merging together the rendering nodes framebuffers, in order to yield the correct final image. Here correct means

the final outcome has virtually to be the same as if the rendering was accomplished only on one node (despite the rendering on a single node could be impossible due to hardware or software constrains). A widely used technique (and the one adopted in our work) is referred to as alpha compositing, first formally introduced by [26]. An additional alpha component is added to the three standard color components RGB, originating the well known RGBA format. The alpha states how much of a pixel is covered by the RGB color. The alpha component is necessary since, generally, it is impossible to express the coverage information using only the RGB channels. An alpha set to 0 means the related color is completely transparent, while an alpha set to 1 means the color is full opaque. All the values between 0 and 1 can be used, arising partially transparent effects.

When two images A and B are composed together, it is required to determine how much of a pixel color contribution is given by A or B. Chosen a generic pixel and the corresponding α_A and α_B for A and B, $\alpha_A\alpha_B$ is the portion of the pixel occupied by both images, $\alpha_A(1-\alpha_B)$ is the portion occupied by a and not by B, $\alpha_B(1-\alpha_A)$ by B and not by A, and finally $(1-\alpha_B)(1-\alpha_A)$ represents the background visible through both A and B. So each pixel of the composed image can be conceptually divided into four subpixel areas: neither A or B contribute to the subpixel(0), only A (A), only B (B) or both (AB). It is then possible to describe compositing operators, according to the final effect on the above subpixel areas. Each operator may be described as a quadruple, (a, b, c, d), where each element refers to the 0, A, B or AB subpixel area respectively. For instance, (0, A, B, B), indicates the area 0 receives a null contribution (trivial, since neither A nor B do not

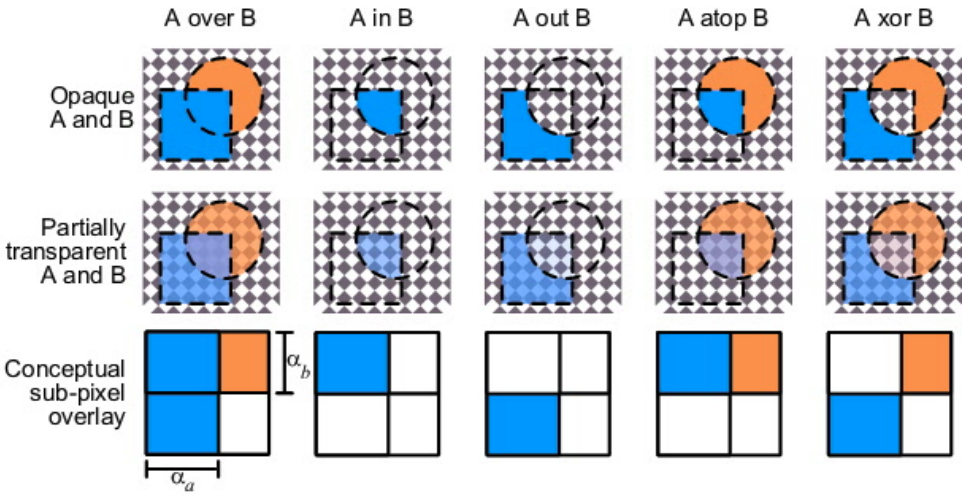

Figure 4.2: Porter and Duff operators. Source: Wikipedia

contribute to 0), the area A retains the color of A, the area of B the one of B, and finally the area of overlapping AB is covered by B. Exploiting all the possible configurations, the resulting compositing operators are:

- *over*. A over B, or (0, A, B, A), means that in the overlapping area the A color is the foreground covering the B background.
- *in*. A in B, or (0, 0, 0, A), where only the overlapping area is taken into account, using A as foreground.
- *out*. A out B, or (0, A, 0, 0), only the area of A not overlapped to B is displayed.
- *atop*. A atop B, or (0, 0, B, A), where the not overlapped A is missing.
- *xor*. A xor B, or (0, A, B, 0), where the overlapping area is missing.

The operators definition is useful because each operator may also be determined using the alpha values of A and B. So a generic operator, first

logically thought as working on a subpixel base, is then defined on a pixel base. For instance, the over operator definition can be formulated according to the fraction of A (F_A) and B (F_B) actually employed for the composition. If A over B is the case, then it means the full portion of A is used as foreground. This is expressed saying the fraction of A is 1. B is not covered by A only outside the overlapping area, so the fraction of B is $(1 - \alpha_A)$. Now it is possible to give a truly operative description of the over operator. The pixel color resulting from the composition of A over B is :

$$c_O = c_A + c_B(1 - \alpha_A) \tag{4.1}$$

(4.1) comes out from the generic operation equation:

$$c_O = c_A F_A + c_B F_B \tag{4.2}$$

It is noteworthy that (4.2) uses the pre-multiplied alpha. In order to understand the need for premultiplied alpha, let consider the RGBA quadruple (0, 1, 0, 0.5). It is generally thought as a full green color covering half of a pixel. It implies a multiplication has to be performed in order to compose the color, that is (0, 0.5, 0, 0.5) in the above example. The color can be pre-muliplied for the alpha, so a multiplication step is avoided. Given (r, g, b, α), it is intended to be pre-multiplied, the color is $(r/\alpha, g/\alpha, b/\alpha)$ and it covers α portion of the pixel. Otherwise, if no pre-multiplication is adopted, the equation becomes harder:

$$c_O = \alpha_A c_A F_A + \alpha_B c_B F_B \tag{4.3}$$

If (4.3) is used with the over fraction coefficients, the outcome is:

$$\begin{aligned} c_O &= \alpha_A c_A + \alpha_B c_B (1 - \alpha_A) \\ &= \alpha_A c_A + \alpha_B c_B - \alpha_A \alpha_B c_B \end{aligned} \tag{4.4}$$

while (4.1) can be written as:

$$c_O = c_A + c_B - \alpha_A c_B \tag{4.5}$$

Thus using (4.5) instead of (4.4) saves three multiplications. Since compositing operations usually concern a big amount of pixels, it may give a consistent performance advantage. However, (4.5) becomes even more useful if the composition steps are more then one. The c_O resulting from both (4.5) and (4.4) is a pre-multiplied value. Before (4.4) is applied again, c_O has to be converted to a non pre-multiplied value (i.e., divided by the alpha value). On the other hand, the pre-multiplied formula does not suffer this problem.

4.2.3 OpenGL alpha blending

The OpenGL API fully supports the above compositing operators. OpenGL refers to this feature as alpha blending. The main function is:

```
void glBlendFunc(GLenum sfactor,
                 GLenum dfactor)
```

It basically provides a mechanism to set the F_A and F_B of the above equation. `sfactor` and `dfactor` can be set to 0, 1, α_A, α_B, $(1-\alpha_A)$ and $(1-\alpha_B)$. All the Porter and Duff operations can be accomplished. However, OpenGL

extends this set of operations allowing also other values (e.g., c_A or c_B). OpenGL gives further compositing operations using the `glBlendEquation` extension, which allows to change the arithmetic operator of the equation (e.g., subtraction instead of addition).

4.2.4 3D compositing via stencil buffer

A standard technique to perform a true 3D compositing is to use the depth values from the z buffer. Each image is read from the color buffer together with the z buffer, in order to keep the spacial position for each pixel. When two images A and B are composed, first the A color and z buffers are copied on the local frame buffer. Then, a two-pass algorithm is used in order to perform the compositing. The basic idea is to compare the depth values of B with the depth values of A, and to store the result on the stencil buffer. The stencil buffer is afterward employed to enable the write operation only for the pixel of B closer to the viewpoint. The algorithm is shortly implemented as follows:

1. The stencil is cleared with 0 and the current color buffer is masked, that is the write operation is disabled.

   ```
   glClearStencil(0);
   glClear(GL_STENCIL_BUFFER_BIT);
   glColorMask(GL_FALSE, GL_FALSE,GL_FALSE,GL_FALSE);
   ```

2. The stencil test is enabled and the test function is set to always, that is the stencil test is always passed. The key point here is the test does not

depend on the stencil values, but only on the depth values comparison. Finally, `glStencilOp` sets the action to be performed after the test. Remarkably, if both the stencil test (trivial) and the depth test are passed, a 1 is written onto the stencil buffer.

```
glEnable(GL_STENCIL_TEST);
glStencilFunc(GL_ALWAYS, 1,1);
glStencilOp(GL_KEEP, GL_KEEP, GL_REPLACE);
```

3. The depth test is enabled, the test is passed only if the incoming pixel (image B) is closer than the current one (image A) to the viewpoint. Finally, the `glDrawPixels` writes the depth values of B, so the depth test is performed between the B and A depth values. When a pixel of B is closer, a 1 is written onto the stencil buffer.

```
glEnable(GL_DEPTH_TEST);
glDepthFunc(GL_LESS);
glDrawPixels(xs, ys, GL_DEPTH_COMPONENT, GL_UNSIGNED_BYTE,
             data_z);
```

4. Now the depth test is disabled and the write operation on the color buffer is enabled. The stencil test is passed only if the current stencil pixel is set to 1.

```
glStencilFunc(GL_EQUAL, 1,1);
glStencilOp(GL_KEEP, GL_KEEP, GL_KEEP);
glDisable(GL_DEPTH_TEST);
glColorMask(GL_TRUE, GL_TRUE,GL_TRUE,GL_TRUE);
```

5. The color buffer of B is written onto the local color buffer. The stencil buffer ensures only the pixels closer to the viewpoint are written. The final image A+B will be blended according to the spacial relationship.

```
glDrawPixels(xsize, ysize, GL_RGBA, GL_UNSIGNED_BYTE, data);
```

4.2.5 3D compositing without depth information

Despite early versions of this work used the 3D compositing via the stencil method explained above, further developments allowed to improve the compositing algorithm. This is basically due to two reasons: the 3D texture is a parallelepiped, all the slaves have the viewport position information. Both features can be exploited during the slicing phase in order to sort the chunks and the related rendering results stored on the local buffers. Thus in our work it allows to avoid to send the z buffer values, since the chunk coefficient is enough to compose all the color buffers in the proper order. The correctness is guaranteed by the chunk geometry. If the buffer A has a smaller coefficient than the buffer B, it means the A centroid is farther than B centroid from the viewpoint. Every pixel of A is behind B, or not covered by B. So it is required to compose A before B, and the depth information is not necessary. The compositing algorithm can be reduced to:

```
glDrawPixels(xsize, ysize, GL_RGBA, GL_UNSIGNED_BYTE, data);
```

where `data` represents the color buffer to be composed.

4.3 Parallel compositing

4.3.1 Parallelism at the compositing phase

The trivial approach to achieve a sort-last compositing consists of using only one compositor node. After the rendering nodes have accomplished their duty, they read and send their local buffers to the compositor node. In the present work this is often referred to as normal mode.

In order to speed up the performance and increase the scalability, the binary swap algorithm, first proposed by [22], has been implemented. The main issue here is that if the compositing phase is performed only by a single compositor, the number of buffers it receives increases linearly with the number of processors. On the one hand, increasing the number of processor implies each rendering node has to render a smaller sub-volume, so the rendering time decreases. On the other hand, the amount of work of the single compositor increases, thus it becomes the main overhead of the overall sort-last rendering.

The key idea is to use the availability of several nodes not only for the local rendering of sub-volumes, but for the compositing phase as well. A first approach is represented by the binary compositing. Basically, each node reads and sends its color and depth buffers to another node, in charge of performing an intermediate compositing with its own color and depth buffers. Let the number of nodes be n, $n/2$ nodes perform the first compositing step. Then, $n/2$ nodes seek a pair and $n/4$ perform the second compositing step. After $\log n$ steps the final image is achieved. A shortcome of binary compositing is that many nodes are idle during the compositing steps. Furthermore,

active nodes decrease at each step.

4.3.2 Direct send

A suitable approach to achieve balanced compositing is represented by the direct send method. Here a tile of the final image is assigned to each node. The tile can be a rectangular portion of the screen, or can be a set of pixels chosen randomly or interleaved. The latter is adopted in order to balance the compositing work, since a node could compose images for a tile covering an empty sector of the screen. Each node has to retrieve all the information inherent to its tile. So each node has to communicate with all the other nodes and get the pixels falling into its tile. At the same time, each node has to send all its own pixels, with the exception of the ones belonging to the tile it is responsible for. If the number of nodes is n, then each node has to contact $n-1$ nodes to compose its tile of the final image. Thus the overall number of communications among nodes is $n(n-1)$. In contrast with binary swap, direct send uses messages of constant size. If xy is number of pixels of the final image, each tile (and consequently each message) is made of xy/n pixels.

4.3.3 Parallel pipeline

The parallel pipeline algorithm resembles the direct send, since the final image is obtained after $n(n-1)$ message passing among the nodes. However they differ because each node sends and receives the tiles from the same nodes, while the tile it is responsible for changes. The nodes are organised as a circular ring, each node always sends its buffers to the next node in

the ring and receives the buffers of the previous ones. At each step, each node changes the area where the incoming sub-image has to be composed. This is done partitioning the z buffer into disjoint tiles, and using at each step a different z buffer tile (and the corresponding color buffer tile) for the compositing. On a node i a parallel pipeline step can be described as follows:

1. set to i the current composed area (only for the initialisation)
2. send the current area to the next node in the ring
3. receive the area of the previous node in the ring
4. calculate k, that is the local area where to compose the incoming area; the area is addressed using the z buffer tiles
5. compose
6. set to k the current area

All the nodes perform at the same time the algorithm. Each step is executed $n-1$ times. The k coefficient is determined using the current step index j, the node number i (i.e., the position of the node in the ring) and the total number of nodes n:

$$k = (i - j) \ mod \ n$$

4.3.4 Binary swap

The binary swap algorithm exploits the idea to use multiple nodes not only for the rendering but also for the compositing, as binary compositing. The main

difference is that binary swap employs all the nodes at each step, avoiding the lack of performance due to idle nodes. At each step, each node is coupled with a peer. Each node sends half of its own image to the peer, and receives a half image from the peer. Then, each node composes the received half image with its own half image (i.e., the one it did not send). A different way to see a binary swap step is that each image is partitioned into two disjointed images, let say left and right or top and bottom. A node is responsible only for one half, while the other is sent to the peer.

For example, the node A and B are coupled together at the step n. First, both nodes partition their own image and get two half images, left and right. Then, A is responsible only for the left images and sends its right to B. B is responsible for the right images, so it receives its left from A and receives A right. Finally, A composes together the left images and B the right ones. Now, A and B use their composed left and right images, respectively, as current image for the next step. It means a node is responsible of an image size decreasing from step to step. At the first step, the total image size is the one of the node color buffer, where the node already performed its local rendering. It sends an image half the size of the color buffer. At the next step, it sends an image that is a quarter of the original color buffer size, and so on.

After each step, each node keeps lesser information about the total image, since it becomes responsible of a smaller portion of the total image. However, its information is more accurate, since its portion is achieved using the image received from its peer. At the last step, each node has a tile of the final image. The final image is obtained tiling together all the nodes sub-images.

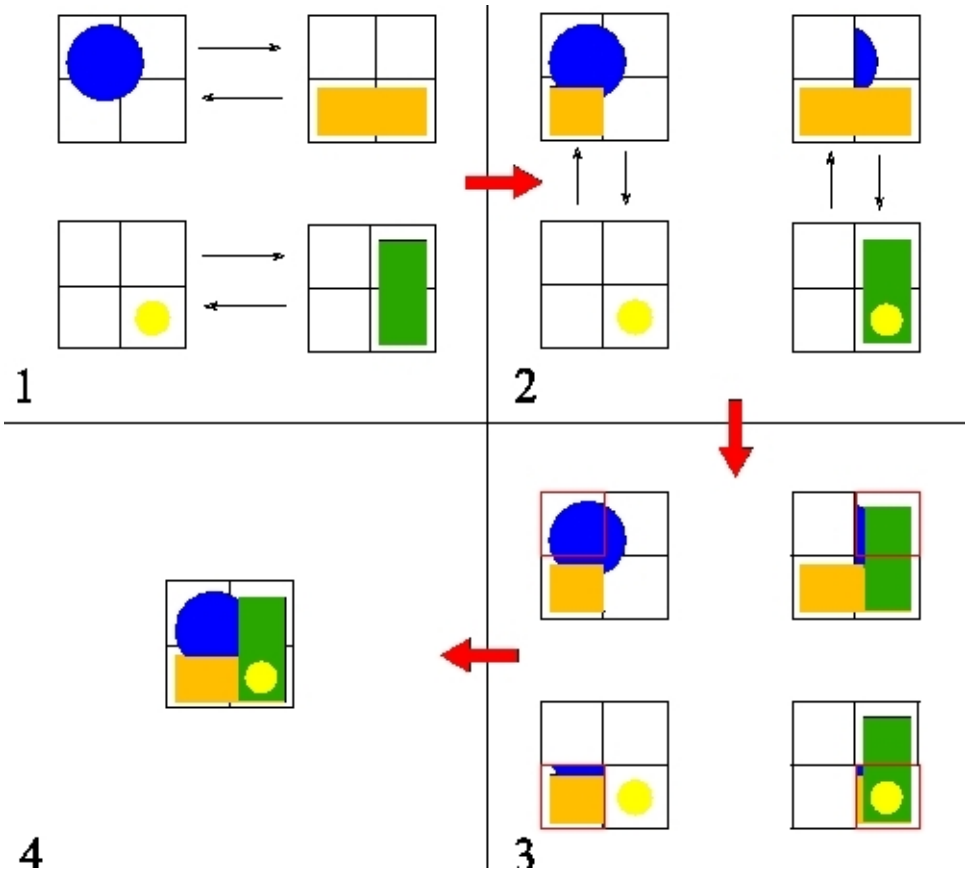

Figure 4.3: Binary swap example. Each couple of peers exchanges half frame buffer at first (1), then a quarter (2). Each node has a tile of the final image (3), which it is sent to a compositor node (4)

In order to keep all the nodes always active, a power of two number of nodes is required. This constrain can be relaxed allowing some nodes to stay idle at certain steps. Let n be the number of nodes, then each node had to establish $logn$ communications with other nodes in order to send and receive the sub-images. So the overall number of communications is $n \log n$.

Chapter 5

Validation

5.1 Analysis

5.1.1 Overall analysis

We define the parameters used in the following sections in order to compare the algorithms and evaluate the tests. The number of slaves (i.e, the rendering nodes) is n, while the master node is unique, and it hosts both the application and the display node. The image resolution is of xy pixels. The number of bits per pixel is given by *bpp*. The time to display a single frame can be summarized as follows:

$$time = setup + render + compose + collectanddraw$$

setup has a heavy impact for the first frame, since the global Aura setup is performed and the scene graph is sent from the application to the slaves. The `SortLastProcessor`, running on each slave, splits the 3D texture. It

requires to re-arrange the texture data, keeping the pixels falling into the sub-volume assigned to the slave. However, these operations are performed only once. Hereafter, *setup* is mainly represented by the transmission of the viewpoint from the application to the slaves and the `SortLastProcessor`. The latter classifies each sub-volume according to the distance from the current viewpoint. Besides, it changes the number of slices fitting the volume. *render* is the time required for the 3D texture rendering. Modern GPUs facilities allow fast rendering, despite the graphics card memory represents a constrain to the size of the texture. The *compose* term changes according to the compositing strategy adopted. When no parallel compositing is active, each slave simply reads its color buffer and sends it to the display node. Otherwise, each step of the parallel algorithm requires to send and receive a color buffer tile and to compose it locally. The `SendBufferProcessor` is in charge of this duty. At each step i:

$$compose_i = read_i + send_i + receive_i + write_i$$

$read_i$ and $send_i$ create and send the tile for the peer, while $receive_i$ and $write_i$ receive the tile and compose it onto the local buffer. All these terms introduce overheads related to the underlying hardware, that is the internal bus connecting the CPU to the graphics card for $read_i$ and $write_i$ (e.g., AGP 8x, PCI-X) and the network channel for $send_i$ and $write_i$ (e.g., Ethernet, Myrinet).

collectanddraw is the time needed by the display node for harvesting all the buffers, compose them together and swap its local buffer. When no parallel compositing algorithm is used, this is the main bottleneck. It does

not scale, since increasing the number of nodes increases the pixels received and the communication overheads. When a parallel compositing algorithm is used, the task becomes easier. The most expensive operation becomes to handle the incoming connections, while the compositing is reduced to n calls to `glDrawPixels` with xy/n pixels, in order to write each tile onto the local buffer.

5.1.2 Parallel compositing analysis

It is noteworthy that the number of pixels sent through the network is constant for all the algorithms described in 4.3. Furthermore, also the trivial approach where no parallel compositing is adopted sends the same amount of pixels. The latter is easily shown since each node sends its entire color buffer to the display node (here we do not consider the z buffer), thus it sends xy pixels. If we are running the direct send algorithm, each node sends $1/n$ of its color buffer to all the other $n-1$ nodes. It means it sends $(n-1)(xy/n)$ pixels. We have to add still a xy/n to this result, because it sends also its own tile to the display node. A similar logic applies to the parallel pipeline algorithm.

The standard binary swap requires a power of two number of nodes, $n = 2^k$. At each step, each node sends $xy_i = xy/2^i$ pixels to a peer through the network. The overall number of step is $k = \log n$. Thus each node sends:

$$\sum_{i=1}^{k} xy_i = \sum_{i=1}^{\log n} \frac{xy}{2^i} = xy(1 - 1/n)$$

Adding the final message to the display node we get xy pixels, as for the

above algorithms. This result can be easily explained, since each pixel of the final image is obtained compositing all the pixels in the same position generated on the slave side. It is the more general case, and the worst case if compression techniques are used. The rationale to use binary swap instead of the other parallel compositing algorithms is it sends less messages through the network. If the overheads related to the network usage are high, or if the number of nodes is high, the binary swap has the best performance. Its main disadvantage is it requires a power of two number of nodes.

5.2 Tests

5.2.1 Cluster overview

The present work has been tested on a cluster made up of 9 nodes. Each node is a PC equipped with the same basic hardware:

- *processor*: dual-CPU, each CPU is an AMD Athlon MP 1.2Ghz
- *main memory*: 512MB
- *graphics board*: NVIDIA GeForce 4 (128MB)
- *network interface*: 100Mbps Ethernet

One node acts as a master and controls the other 8 nodes, used as rendering nodes. The master provides standard input/output devices such as keyboard, mouse and monitor, in order to allow the user to manage the whole cluster. Each rendering node has a video projector. All the projection areas

are aligned, the resulting tiled display is of around 5 meters by 2 meters. Each node is connected to an Ethernet switch. Linux is employed as operating system, and each node has the hardware acceleration provided by the NVIDIA driver. The OpenGL API and the Aura library are also installed.

5.2.2 Tests

We ran our tests using several types of 3D textures, both procedurally generated or acquired from real world data sets. We were able to render textures of up to 256^3 voxels using windows of 512^2 pixels, obtaining around 7 fps (i.e, frame per second). [4] reports Kirihata obtained around 14 fps for the same parameters, but using a Gigabit Ethernet connection. Since communication overheads have a strong impact on our rendering, we would expect to obtain a similar outcome with a faster connection. The comparison of several compositing strategies shows the benefits related to the binary swap algorithm. The usage of the depth information together with the stencil buffer technique (Stencil on the diagrams) to achieve the compositing is highly inefficient. Remarkably, on the Display Server side the time required by the compositing is greater than the time for the communication, whereas all the other compositing strategies have the opposite behaviors. This can be easily explained, since the two-pass stencil algorithm is more computational expensive than a simple call to glDrawPixels. Using the sequential compositing without depth information (Normal mode on the diagrams) improves the performance, but it does not scale. The Display Server becomes the main bottleneck. Adding additional rendering nodes imply to decrease the overall performance. The binary swap shows the best performance, with an almost

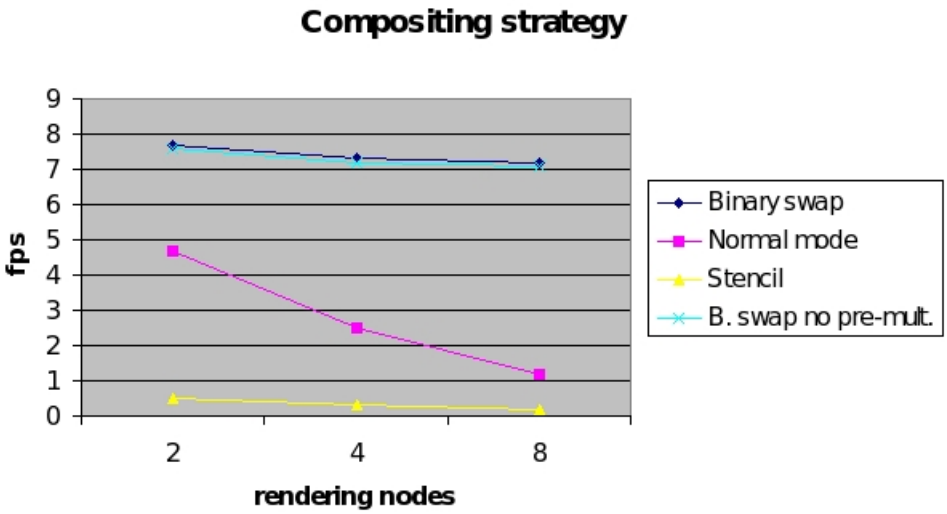

Figure 5.1: Binary swap has the best performance and scales better. Texture dimension: 128^3. Window size, in pixels: 512^2

double fps when compared to the normal mode in its best case (i.e., two rendering nodes). The binary swap performance slightly decreases adding extra rendering nodes. Moreover, the slave side analysis shows the time spent waiting for barriers is little, especially when compared to the stencil and normal mode cases. It means the global work-load among the nodes is well balanced. We also ran a test using binary swap but without pre-multiplied alpha, in order to investigate the advantage of using pre-multiplied colors. The impact on the performance is very small. This is due to the little portion of time spent on the rendering (Rendering processor), even smaller than the time required by the inexpensive slicing phase (SortLast processor).

In contrast with the above Aura processors, the slave side analysis shows most of the time is spent for network and compositing operations. On each slave the network operations are represented by the Receive and Forward pro-

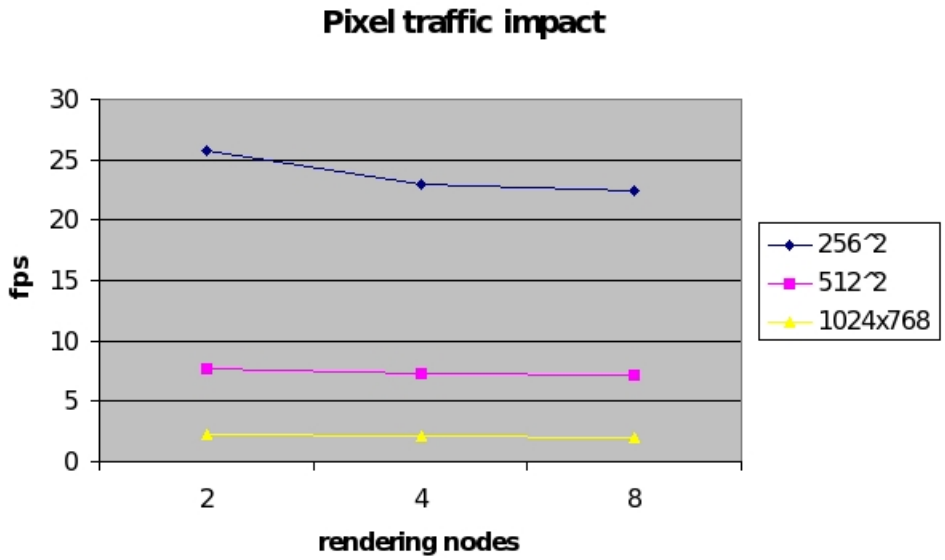

Figure 5.2: The window size has a strong impact on the performance. Texture dimension: 128^3. Compositing algorithm: binary swap

cessors, which handle and propagate the incoming messages from the master. Besides, the SendBuffer processor highly uses the network channel when the binary swap is active. It is noteworthy that much time (more than half with binary swap) is registered as not measured. This is related on how the performance is measured. The Aura performance monitor class is used to start a global timer, which at even intervals prints performance measurements on a log file. Important pieces of code, such as Aura processors, usually deserve a local timer, in order to understand the component usage. All the time of the global timer not belonging to a specific local timer is recorded as not measured. Hence the not measured time is related to code of secondary importance, at least on a logical point of view. Since Aura is a general purpose graphics library able to accomplish several tasks, it can be easily explained.

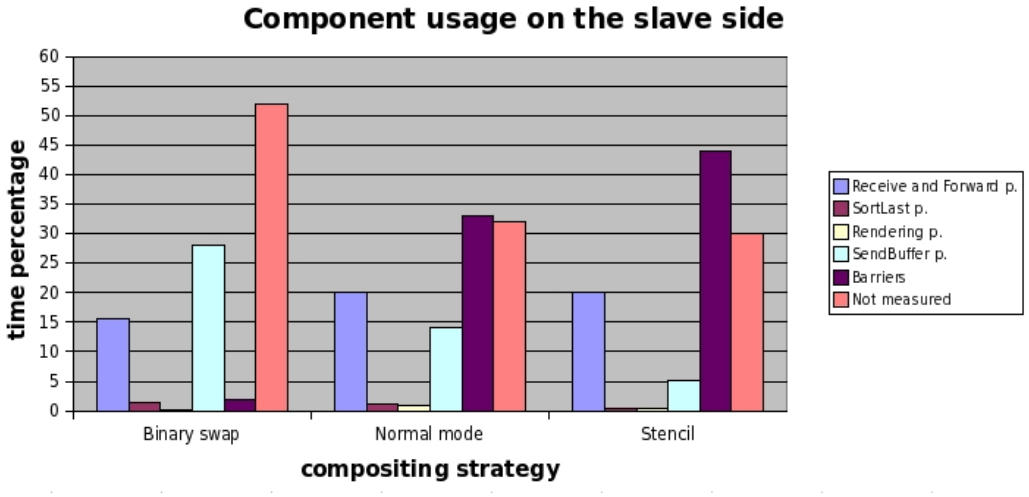

Figure 5.3: Components related to the communication and the compositing have the biggest impact on the performance (Receive-, Forward-, SendBuffer processor), while the slicing and the rendering components are negligible (SortLast-, Rendering processor). Barrier overheads can be curtailed using the binary swap. For an explanation of the not measured parameter see the text. Texture dimension: 128^3. Window size, in pixels: 512^2. Rendering nodes: 8

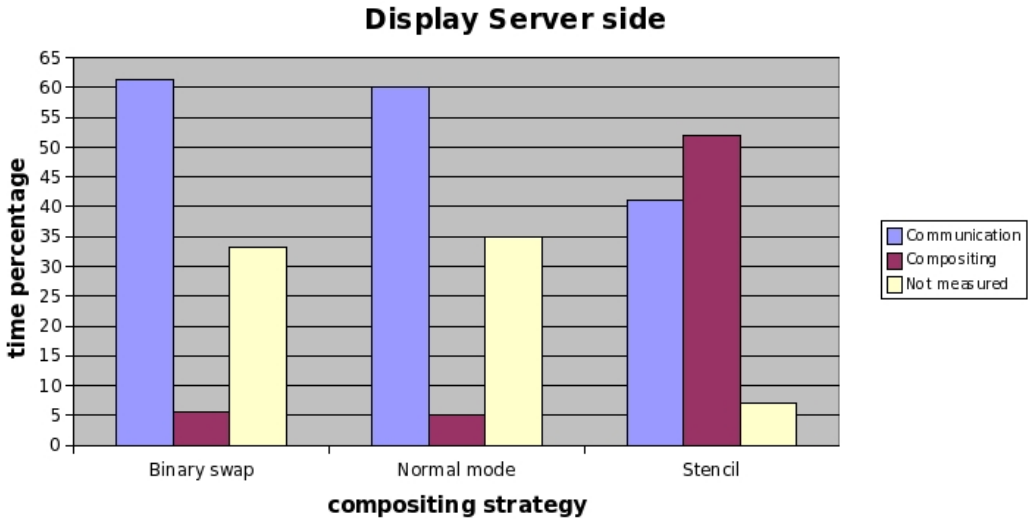

Figure 5.4: Except for the stencil algorithm, the communication with the slaves is the most time consuming operation on the Display Server. Texture dimension: 128^3. Window size, in pixels: 512^2. Rendering nodes: 8

Reshaping the components to achieve an ad hoc application focused on the parallel volume rendering would overcome this problem. Finally, tests on the window size shows the strong impact of the pixel traffic on the performance. This was fully expected, because it is the typical overhead of a sort-last approach. Compression techniques would be necessary to overcome such issue.

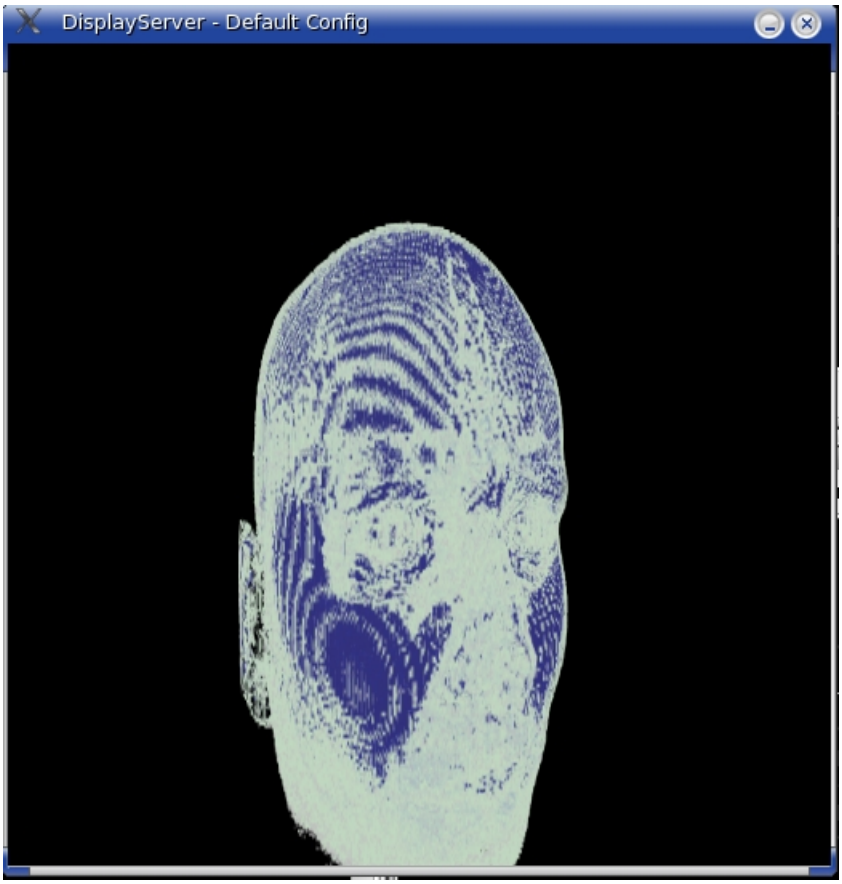

Figure 5.5: MRI scan of a human head. The data set is of 256^3 voxels. Each voxel is originally stored with 8 bits. The rendering is accomplished using 4 rendering nodes. Data set source: http://www9.cs.fau.de/Persons/Roettger/library/

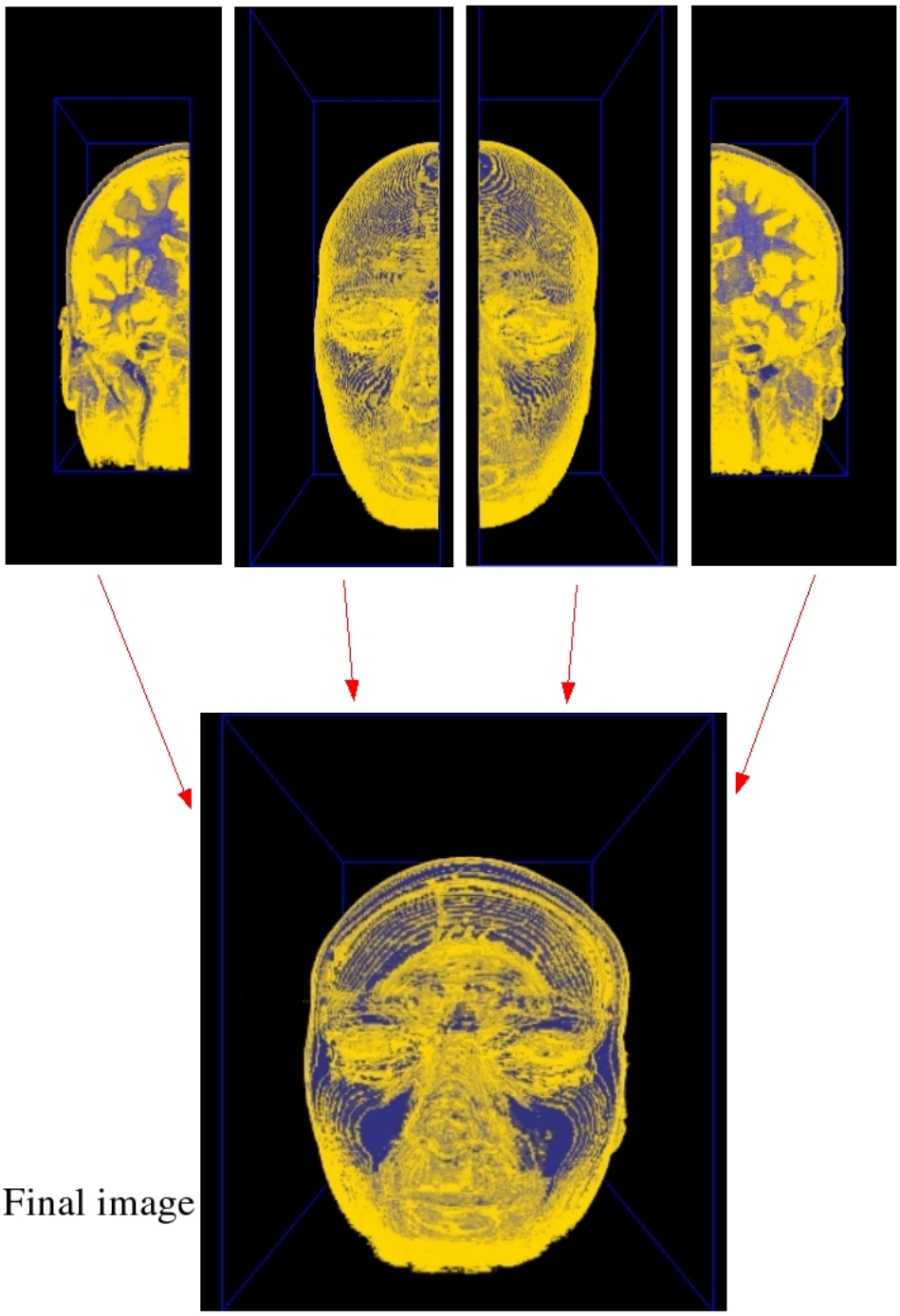

Figure 5.6: The thesis concept. 4 color buffers containing partial images are composed together, achieving the final image

Chapter 6

Conclusion

Implementing a parallel volume rendering implies to deal with several topics, since the graphics operations are achieved using several components and it is requested to fully exploit the hardware support. It is required to deeply understand how the rendering takes place on a uniprocessor architecture, and how it is possible to parallelize it. Several strategies are suitable, each one with its own advantages and drawbacks. In our case, the sort-last choice stressed the need for a full knowledge of the alpha compositing technique. Besides, the huge amount of pixels sent by the sort-last approach required to employ an efficient way to exchange data, that is the binary swap algorithm. The usage of the 3D textures allowed to locally exploit the hardware accelerators, and to use the Aura features concerning the textures manipulation. Updated works [4] adopt a similar approach for parallel volume rendering, that is texture-based and sort-last. This confirms the reasonability of our choices.

Future works could address some unresolved issues and provide optimisa-

tion techniques. Despite the parallel rendering is achieved equally splitting the volume data set among all the nodes, the load-balance is not actually guaranteed. The issue is well-known, as we pointed out in 3.2.1. Each node has to work on the same amount of data, thus the parallelism is balanced at a task level. However, a node can receive a complete transparent sub-volume, whereas a different node can render a full opaque sub-volume. Both nodes accomplish the same work, but only the latter gives a relevant contribution to the final image. This means the work of one node is completely useless. The problem can be addressed involving a pre-processing step. The whole volume is analysed and each voxel is classified according to its relevancy. The volume distribution can be done not merely assigning the same number of voxels to each node, rather assigning the same number of relevant voxels. The pre-processing would possibly require the employment of hierarchical data structures to handle the data set.

The compositing phase could be boosted using a multi-threaded approach, that is employing threads with specific tasks. For instance, a thread could be in charge of the communication, while another thread deals with the rendering. The threads introduction could imply drastically architectural changes. Our implementation performed the compositing exploiting the local GPU. The incoming buffer is composed with the one already stored on the local graphics board memory. A drawback is the rendering is blocked while the compositing is performed. A solution could be to use the CPU to yield the compositing with an appropriate thread, allowing a rendering thread to carry on the computation.

The sort-last could also be improved. In our work we used a sort-last full

approach, as explained in 3.1.5. Each node reads and sends its entire local buffer, even the pixels which do not contribute to the final image. A sort-last sparse approach would require to read and send only those pixels which actually contribute to the final compositing. It needs to filter the buffer in order to detect only the relevant pixels. Otherwise it can be approximated selecting the smallest rectangle containing the relevant pixels. The main advantage is the pixels sent over the network and composed are generally less if compared with the sort-last full approach.

Finally, the Aura lack of an appropriate transfer function tool had to be filled, in order to easily handle real world data sets. First, the tool should be able to load files containing the raw data. Then, it should provide an interactive framework to set and arrange the transfer values. Only professional transfer function tools allow to truly exploit the impressive potential of the volume rendering techniques.

Bibliography

[1] D. Bartz, R. Großo, T. Ertl, and W. Straßer. Parallel construction and isosurface extraction of recursive tree structures. In *Proc. of WSCG'98*, volume III, pages 479–486, 1998.

[2] P. Benölken and H. Graf. Direct volume rendering unstructured grids in a pc based vr environment. In *WSCG (Journal Papers)*, pages 25–32, 2005.

[3] E. W. Bethel, G. Humphreys, B. Paul, and J. D. Brederson. Sort-first, distributed memory parallel visualization and rendering. In *PVG '03: Proceedings of the 2003 IEEE Symposium on Parallel and Large-Data Visualization and Graphics*, page 7, Washington, DC, USA, 2003. IEEE Computer Society.

[4] X. Cavin, C. Mion, and A. Filbois. Cots cluster-based sort-last rendering: Performance evaluation and pipelined implementation. In *IEEE Visualization*, page 15, 2005.

[5] W. T. Corrêa, J. T. Klosowski, and C. T. Silva. Out-of-core sort-first parallel rendering for cluster-based tiled displays. In *EGPGV '02: Pro-*

ceedings of the Fourth Eurographics Workshop on Parallel Graphics and Visualization, pages 89–96. Eurographics Association, 2002.

[6] M. B. Cox and D. Ellsworth. Application-controlled demand paging for out-of-core visualization. *IEEE Visualization 97*, pages 235–244, 1997.

[7] T. J. Cullip and U. Neumann. Accelerating volume reconstruction with 3d texture hardware. Technical report, University of North Carolina, Chapel Hill, May 1 1994.

[8] R. Farias, J. S. B. Mitchell, and C. T. Silva. Zsweep: an efficient and exact projection algorithm for unstructured volume rendering. In *VVS '00: Proceedings of the 2000 IEEE symposium on Volume visualization*, pages 91–99, New York, NY, USA, 2000. ACM Press.

[9] R. Farias and C. T. Silva. Out-of-core rendering of large, unstructured grids. *IEEE Comput. Graph. Appl.*, 21(4):42–50, 2001.

[10] A. Garcia and Han-Wei Shen. An interleaved parallel volume renderer with pc-clusters. In *EGPGV '02: Proceedings of the Fourth Eurographics Workshop on Parallel Graphics and Visualization*, pages 51–59, Aire-la-Ville, Switzerland, 2002. Eurographics Association.

[11] H. Hauser, L. Mroz, G.-I. Bischi, and E. Grller. Two-level volume rendering - fusing mip and dvr. In *Proceedings IEEE Visualization 2000*, 2000.

[12] G. T. Herman and H. K. Lui. Three dimensional display of human organs from computer tomograms. *Computer Graphics and Image Processing*, 1979.

[13] G. Humphreys, M. Houston, Y. Ng, R. Frank, S. Ahern, P. Kirchner, and J. Klosowski. Chromium: A stream processing framework for interactive graphics on clusters. In *In Proceedings of ACM SIGGRAPH 2002*, volume 21, pages 693–702, 2002.

[14] J. Krueger and R. Westermann. Acceleration techniques for gpu-based volume rendering. In *Proceedings IEEE Visualization 2003*, 2003.

[15] P. Lacroute. *Fast volume rendering using a shear-warp factorization of the viewing transformation.* PhD thesis, Stanford University, 1995.

[16] P. Lacroute and M. Levoy. Fast volume rendering using a shear-warp factorization of the viewing transformation. In *Proceedings of ACM SigGraph Conference 1994*, pages pp. 451–458, Orlando, FL, July 24-29 1994.

[17] Tong-Yee Lee, C. S. Raghavendra, and John B. Nicholas. Image composition schemes for sort-last polygon rendering on 2d mesh multicomputers. *IEEE Transactions on Visualization and Computer Graphics*, 2(3):202–217, 1996.

[18] W. Lee, V. P. Srini, and T. Han. Efficient volume visualization on GPU clusters using a combination of hierarchical data structures. In *IEEE Visualization 2005, Minneapolis, MN, United States*, 2005.

[19] M. Levoy. Display of surfaces from volume data. *IEEE Computer Graphics and Applications*, 1988.

[20] M. Levoy. Efficient ray tracing of volume data. *ACM Transactions on Graphics*, 1990.

[21] W. E. Lorensen and H. E. Cline. Marching cubes: A high resolution 3d surface construction algorithm. *Computer Graphics*, 1987.

[22] K. Ma, J. S. Painter, C. D. Hansen, and M. F. Krogh. Parallel volume rendering using binary-swap compositing. *IEEE Comput. Graph. Appl.*, 14(4):59–68, 1994.

[23] K. Ma and S. Parker. Massively parallel software rendering for visualizing large-scale data sets. *IEEE Comput. Graph. Appl.*, 21(4):72–83, 2001.

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

[25] L. Mroz. *Real-Time Volume Visualization on Low-End Hardware*. PhD thesis, Vienna University of Technology, 2001.

[26] T. Porter and T. Duff. Compositing digital images. In *SIGGRAPH '84: Proceedings of the 11th annual conference on Computer graphics and interactive techniques*, pages 253–259, New York, NY, USA, 1984. ACM Press.

[27] L. Renambot, H. E. Bal, D. Germans, and H. J. W. Spoelder. CAVEStudy: An infrastructure for computational steering and measuring in virtual reality environments. *Cluster Computing*, 4(1):79–87, 2001.

[28] L. Renambot and T. van der Schaaf. Enabling tiled displays for education. Workshop on Commodity-Based Visualization Clusters, in conjunction with IEEE Visualization 2002.

[29] L. Renambot, T. van der Schaaf, H. Bal, D. Germans, H. Spoelder, and T. Kielmann. Tiled displays to enable collaboration on the grid. journal paper, submitted to publication.

[30] R. Samanta, T. Funkhouser, K. Li, and J. Singh. Hybrid sort-first and sort-last parallel rendering with a cluster of pcs. *Eurographics/SIGGRAPH workshop on Graphics hardware*, pages 99–108, 2000.

[31] P. Shirley and A. Tuchman. A polygonal approximation to direct scalar volume rendering. In *Proceedings of the 1990 Workshop on Volume Visualization*, pages 63–70, San Diego, CA, Dec 10-11 1990.

[32] A. Smith. Image compositing fundamentals. Technical Report 4, Microsoft, 1995.

[33] A. Smith. A pixel is not a little square, a pixel is not a little square, a pixel is not a little square! (and a voxel is not a little cube). Technical report, Microsoft, 1995.

[34] M. Strengert, M. Magalln, D. Weiskopf, S. Guthe, and T. Ertl. Hierarchical Visualization and Compression of Large Volume Datasets Using GPU Clusters. In *Eurographics Symposium on Parallel Graphics and Visualization (EGPGV04)*, pages 41–48. Eurographics Association, 2004.

[35] T. van der Schaaf, L. Renambot, D. Germans, H. Spoelder, and H. Bal. Retained mode parallel rendering for scalable tiled displays, 2002.

[36] M. Weiler, R. Westermann, C. Hansen, K. Zimmerman, and T. Ertl. Level-of-detail volume rendering via 3d textures. In *IEEE Symposium on Volume Visualization 2000*, 2000.

[37] M. Westerhoff. *Efficient Visualization and Reconstruction of 3D Geometric Models from Neuro-Biological Confocal Microscope Scans*. PhD thesis, Free University of Berlin, 2003.

[38] L. Westover. Footprint evaluation for volume rendering. *Computer Graphics*, 24(4):367–376, 1990.

[39] T. Whitted. An improved illumination model for shaded display. *Communications of the ACM*, 1980.