Distributed SPH Fluid Simulator

for Hewlett-Packard Scalable Visualization Array

February, 2008

Budapest University of Technology and Economics

Department of Control Engineering and Information Technology

Contents

Chapter 1

Introduction

Simulating *natural phenomena* like smoke, sand or fluid by physics-based algorithms is a rapidly developing area of visualization. Animating fluid is time-consuming and interactive visualization on a single workstation is limited to a less detailed quality. Our goal is to develop an interactive scalable GPU-accelerated particle-based fluid simulation system which runs parallel on *Hewlett-Packard Scalable Visualization Array*.

This document describes the basic equations of fluid dynamics [2] and covers implementation details of a method called *Smoothed Particle Hydrodynamics (SPH)*. The particle-based approach reduces the complexity of the simulation because mass conservation equations can be disregarded. The particle system can be used to render the fluid surface using spheres, metaballs in the positions of the particles. Other visualization possibility would be using *ray casting* methods.

For faster single node simulation the traditional *SPH* implementations should be redesigned for efficient use of today's graphics processor architectures. Interactive visualization of accurate simulation needs a huge number of fluid particles. This complexity is handled using highly parallelized methods.

This report shows an approach how to implement an SPH-based fluid simulation engine using the graphics processor unit *(GPU)* and how to parallelize the solver for a visualization cluster distributing the computation for space domains. The architecture is peer-to-peer, there is no master node for the data set visualization. Displaying the result is also separated, the *ParaComp* compositing API is responsible for collecting the rendered image parts from the nodes and for showing the final result.

In the second chapter of this document the algorithmic background of *SPH-based fluid simulation* is introduced. The third chapter gives a short introduction to parallel rendering techniques, it describes the parallelization approach and details of the implementation for *HP-SVA graphics clusters*. Chapter 4 presents the installation and usage instructions for the mentioned platform. In Chapter 5 the structure of the code is summarized. Generated images and measured frame rates are reported in the last chapter.

Chapter 2

Algorithmic Background

The motion of a fluid system can be described with the *Navier-Stokes equations* [13]. These partial differential equations are usually written in the following form:

$$\frac{\partial \vec{u}}{\partial t} + \vec{u} \cdot \nabla \vec{u} + \frac{1}{\rho} \nabla p = \vec{g} + \nu \nabla \cdot \nabla \vec{u}$$

$$\nabla \cdot \vec{u} = 0$$

The symbol $\vec{u}$ is used for the *velocity* of the fluid, ρ stands for the *density*, p for *pressure*. The letter g is the acceleration due to *gravity*. The ν is called *kinematic viscosity*. It measures how much the fluid resists deforming while it flows.

The first equation is called *momentum equation* which describes how the fluid accelerates due to the forces. The second equation is the *incompressibility condition*.

2.1 Lagrangian and Eulerian Viewpoints

For tracking the motion of the fluid usually two different solutions are used that are called the *Lagrangian* viewpoint and the *Eulerian* viewpoint.

The *Lagrangian* approach represents the motion as a finite interpolated system like a given number of particles. Each point is labelled as a particle with current position, and velocity ($\vec{v}$). Solids are usually simulated in a *Lagrangian* way.

The *Eulerian* approach looks at fixed points in the volume and measures how the fluid quantities *(density, velocity, pressure etc.)* change in time.

The *Lagrangian* viewpoint corresponds to a particle system, the *Eulerian* viewpoint uses a fixed grid that does not change in space.

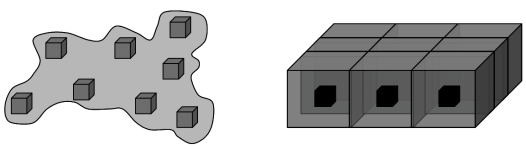

Figure 2.1: *Lagrangian and Eulerian viewpoints. Particles are represented by cubes*

2.2 The SPH Approach

For smoke, water splashes or spray effects the application of particles is the most comfortable solution. A particle system contains a number of particles moving in the space based on the effect of the surrounding forces. Usually they can collide with each other and with obstacles. Without a particle-particle interaction we call our system a *simple particle system*. Such a system can be implemented efficiently and with a low number of particles it runs real-time. These particles are created at the start of the simulation using a defined volume or generated by emitters. Particles are born at a given rate *(particles/sec)* and die after a certain time. When they get close to the end of their *lifetime*, they disappear.

Without a particle-particle collision model the dynamic properties can be described by a set of decoupled ordinary differential equations:

$$\dot{\vec{x}}_i = \vec{v}_i$$
$$\dot{\vec{v}}_i = \vec{f}_i / m_i$$

where x_i is the position, v_i is the velocity, m_i the mass of particle i and f_i is the force affecting the particle.

2.3 Modeling Fluids Using Particles

SPH [10] is an interpolation method for fluid motion simulation. SPH uses field quantities defined only at discrete particle locations and can be evaluated anywhere in space. SPH distributes quantities in a local neighborhood of the discrete locations using radial symmetrical smoothing kernels.

A scalar value A is interpolated at location ($\vec{r}$) by a weighted sum of contributions from the particles:

$$A_S(\vec{r}) = \sum_j m_j \frac{A_j}{\rho_j} W(\vec{r} - \vec{r}_j, h)$$

where j iterates over all particles in the scene, m_j is the mass of particle j. $\vec{r}_j$ is the position, ρ_j is the density, and A_j is the field quantity at $\vec{r}_j$. The $W(\vec{r},h)$ is called *smoothing kernel* with core radius h. The kernel is normalized if

$$\int W(\vec{r}) dr = 1.$$

Because $m_i = m$ constant in our case, we can evaluate the density at every step using a modified equation based on Mueller's work [10]:

$$\rho_S(r) = \sum_j m_j \frac{\rho_j}{\rho_j} W(\vec{r} - \vec{r}_j, h) = \sum_j m_j W(\vec{r} - \vec{r}_j, h).$$

With the SPH approach the derivatives only affect the smoothing kernel. The problem with the method is that these equations are not guaranteed to satisfy some physical rules including symmetry of forces and conservation of momentum. Mueller [10] also solves these SPH-related problems.

Particle-based simulation simplifies the solution of Navier-Stokes equations. Because the number and the mass of particles are constant, mass conservation is guaranteed. The *mass conservation* equation can be omitted. The expression $\frac{\partial \vec{u}}{\partial t} + \vec{u} \cdot \nabla \vec{u}$ on the left side can be simplified because the particles move with the fluid.

Based on simplified Navier-Stokes for the acceleration of particle i we get:

$$\vec{a}_i = \frac{d\vec{u}_i}{dt} = \frac{\vec{f}_i}{\rho_i}$$

where $\vec{u}_i$ is the velocity of particle i, $\vec{f}_i$ and ρ_i are the force field and the density field at the location of particle i respectivelz. The next sections describe how to model the force fields.

2.4 Pressure, Viscosity

Instead of an equation described by the SPH-rule a modified solution is used for pressure force because it guarantees the symmetry of forces:

$$f_i^{pr\vec{e}ssure} = -\sum_j m_j \frac{p_i + p_j}{2\rho_j} \nabla W(\vec{r}_i - \vec{r}_j, h)$$

The pressure at particle locations has to be calculated first, which can be computed via the ideal gas equation:

$$p = k\rho$$

where k is a gas constant that depends on the temperature. A modified version — which we used in our implementation — makes the simulation numerically more stable:

$$p = k(\rho - \rho_0).$$

where ρ_0 is the rest density. Applying the SPH-rule to the viscosity term also yields to asymmetric forces because the velocity field varies. The idea of symmetrizing the expression is using velocity differences:

$$f_i^{viscosity} = \mu \sum_j m_j \frac{\vec{v}_j - \vec{v}_i}{\rho_j} \nabla^2 W(\vec{r}_i - \vec{r}_j, h).$$

2.5 External Forces

Additional forces can be applied to the particles without using the SPH-method. Gravity or other external forces change the acceleration component of the particles. Particle-object collisions are solved by reflecting the velocity component that is perpendicular to the surface.

2.6 Smoothing Kernels

The accuracy of the algorithm highly depends on the smoothing kernels. For our implementation we used the following kernel:

$$W_{poly6}(r,h) = \begin{cases} \frac{315}{64\pi h^9}(h^2 - r^2)^3 & 0 \leq r \leq h \\ 0 & \text{otherwise} \end{cases}$$

The advantage of this kernel is that r only appears squared which means that it can be evaluated without computing square roots in distance calculations. Debrun's *spiky kernel* solves the problem of our basic kernel, clustering under high pressure. For the pressure computing we use the following expression:

$$W_{spiky}(r,h) = \begin{cases} \frac{15}{\pi h^6}(h - r)^3 & 0 \leq r \leq h \\ 0 & \text{otherwise} \end{cases}$$

Viscosity is a phenomenon that is caused by friction decreasing the fluid's kinetic energy by converting it into heat. For two particles that are close to each other, the Laplacian of the smoothed velocity field can cause negative result in forces that increase their relative velocity. For the computation of viscosity forces a third kernel was used because of stability problems:

$$W_{viscosity}(r,h) = \begin{cases} \frac{15}{2\pi h^3}(\frac{-r^3}{2h^3} + \frac{r^2}{h^2} + \frac{h}{2r} - 1) & 0 \leq r \leq h \\ 0 & \text{otherwise} \end{cases}$$

whose Laplacian is positive everywhere with the following properties:

$$\nabla W(r,h) = \frac{45}{\pi h^6}(h - r)$$
$$W(|r| = h, h) = 0$$
$$\nabla W(|r| = h, h) = \mathbf{0}$$

2.7 Single-Node Implementation

The main steps of a hardware accelerated SPH-based fluid simulation algorithm can be summarized as follows.

1. CPU generates a *neighbour texture map* for the GPU. The neighbour map's ith row contains the IDs of the neighbouring particles of particle ith. Particle attributes are *stored in attribute texture maps* (position and velocity maps).
2. A *distance cache texture* is calculated for later algorithm steps. The ith row of the texture contains the distance values between the particle and its neighbours.
3. A *density texture map* is used for particle *density computation* where the pixels contain per particle density and pressure data. The density values are inherited from a sum of the cache texture rows.
4. Per particle force data is cached in the *force texture map* which reads the density, distance, position, and velocity attributes from the above textures.
5. The final GPU step is the *acceleration, velocity and position computation* using Newton's laws implemented by pixel shaders.
6. *Particle positions are read back* to the CPU for refreshing the *neighbour map*

The physics engine was implemented for GPU using pixel shaders. Our solution is based on Amada's GPU-based fluid [1] research.

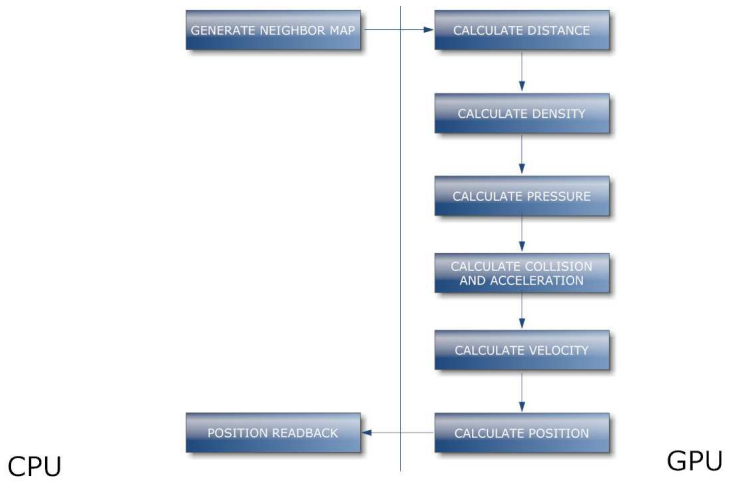

Figure 2.2: *Steps of our GPU-accelerated fluid simulator*

2.7.1 GPU Implementation

Shaders were developed to speed up physics calculation on the graphics hardware. Distances between particles, particle attributes like density, pressure, force, acceleration, velocity and position are computed by pixel shaders.

```
void main( float2 coord : TEXCOORD0,
           uniform samplerRECT attr_rect : TEXUNIT0,
           uniform samplerRECT neighbour_rect : TEXUNIT1,
           uniform float attrlen ,
           out float4 result : COLOR
          )
{
    float3 texel = f3texRECT(neighbour_rect, coord);

    float index = texel.x;        // index of particle
    float nindex = texel.y;       // neighbor index

    float2 index2 = convertInd(index, attrlen );
    float2 nindex2 = convertInd(nindex, attrlen );

    // valid indices
    if (nindex!=-1 && index!=-1) {
        // search in attribMap
        float3 pos_i = f3texRECT( attr_rect , index2);
        float3 pos_j = f3texRECT( attr_rect , nindex2);

        float3 distance_vec = pos_i - pos_j;

        // return distance vector and the vector length
        result = float4 ( distance_vec , length ( distance_vec ));
    } else result = float4 (0,0,0,0) ;
}
```

Listing 2.1: Pixel shader for distance computation between particles

Listing 2.1 shows the distance calculation function implemented in the fragment shader that returns the distance vector and the vector length as RGBA float values. The particle index and the neighbor index are read from the CPU generated neighbor texture map.

```
void main( float2 coord : TEXCOORD0,
           uniform samplerRECT dist_rect : TEXUNIT0,
           uniform samplerRECT dens_rect : TEXUNIT1,
           uniform float len,
           uniform float attrlen,
           uniform float itNum,
           uniform float h,
           uniform float mass,
           uniform float gas_const,
           uniform float maindens,
           uniform float poly6_coef,
           out float3 result : COLOR)
{
    float density = 0;
    float pressure = 0;

    float index = convertInd2(coord, attrlen);
    index = index - (itNum*4096);

    // iterate over neighbours
    for (float i = 0.5; i < len+0.5; i=i+1) {
        float r = texRECT(dist_rect, float2(i,index)).w;
            if(r) {
                float h2_r2 = h*h - r*r;

                if(h2_r2>0)
                density += h2_r2 * h2_r2 * h2_r2;
            }
    }
    // calculate final density
    density *= poly6_coef * mass;

    // calculate pressure
    pressure = gas_const*(density - maindens);

    // R = density, G = pressure, B = 0
    result = float3(density, pressure, 0);
}
```

Listing 2.2: Pixel shader for density and pressure computation

Listing 2.2 shows the density and pressure calculation function implemented in the fragment shader. The shader iterates over the neighbours of the actual particle and

calculates the density and pressure values returned as float values in red and green channels.

```
void main(float coord : TEXCOORD0,
          uniform samplerRECT neighbour_rect: TEXUNIT0,
          uniform samplerRECT dist_rect : TEXUNIT1,
          uniform samplerRECT dens_rect : TEXUNIT2,
          uniform samplerRECT vel_rect : TEXUNIT3,
          uniform float mass_rec,
          uniform float h,
          uniform float grad_spiky_coef,
          uniform float v_lap_visc_coef,
          uniform float len,
          uniform float attrlen,
          uniform float itNum,
          out float4 result : COLOR
          )
{
    // particle attributes
    float3 force = float3(0,0,0);
    float3 acc = float3(0,0,0);
    float3 texel;
    float h_r;

    // compute force
    for(float i=0.5; i < len+0.5; i=i+1) {

        texel = f3texRECT(neighbour_rect, float2(i,coord));

        float index = texel.x;
        float nindex = texel.y;

        float2 index2 = convertInd(index, attrlen).xy;
        float2 nindex2 = convertInd(nindex, attrlen).xy;

        if(index!=-1 && nindex!=-1) {

            float3 distvec = texRECT(dist_rect, float2(i,coord)).xyz;
            float r = texRECT(dist_rect, float2(i,coord)).w;
            float3 element_dist_vec = -distvec / r;

            float density = f3texRECT(dens_rect, index2).x;
            float ndensity = f3texRECT(dens_rect, nindex2).x;
            float pressure = f3texRECT(dens_rect, index2).y;
            float npressure = f3texRECT(dens_rect, nindex2).y;

            float3 vel = f3texRECT(vel_rect, index2);
            float3 nvel = f3texRECT(vel_rect, nindex2);

            h_r = h-r;

            force += distvec*(pressure+npressure)*h_r*h_r*grad_spiky_coef*-0.5*10;
            force += (nvel-vel)*(h_r* v_lap_visc_coef);
            force /= (density+ndensity)*200;

            // acceleration
            acc+= force * mass_rec;
        }
    }
```

```
    // external force - gravity
    acc += float3 (0,-9.81,0);

    // collision calculation
    acc += collForce ();

    // return acceleration
    result = float4 (acc,0);
}
}
```

Listing 2.3: Pixel shader for force computation

Listing 2.3 shows force and acceleration calculation function implemented in the fragment shader. The shader iterates over the neighbours of the actual particle, gets the distance values from the distance texture map and particle and particle neighbour density and pressure scalar values from density texture map. Acceleration computation is based on the sum of inside fluid force, external forces (like gravity) and collision force. Result vector is returned as RGB float color.

From acceleration texture velocity and position maps are generated using simple fragment programs.

Chapter 3

Distributed SPH Simulation

Previous distributed implementations usually use simplified 2D simulation for learning and testing purposes and CPU-based algorithms [15].

We show a scalable distributed SPH approach which has been implemented for the *HP SVA architecture*.

3.1 Parallelization Background

Computer graphics and simulation is one of the most resource demanding side of the information technology. Working with detailed geometry elements and heavy algorithms to imitate the real world need extra capacity and time. Our main goal is to utilize the advantages of distributed systems.

Before implementing a distributed solver the main question we want to answer was how and when we can get better performance with distributed simulation than using only a single computer. The success depends on two factors:

1. *how and which parts of a simulation step can we distribute?*
2. *how much data do we need to transfer among the nodes within a frame?*

In our simulation in the first step we divide the scene (where particles can move within) among the nodes. Each node or client is responsible for the particles in its calculation area. A client needs more data to calculate the actual acceleration of a particle, a client should also get the neighbors in the core. The inner area of the node is not overlapping but the edges of the neighbors belong to other clients.

3.2 Client Areas

Figure 3.2 shows a client area in 2D. We have named the significant units with own created names. There are three main areas and two zones on the map. The most important is the calculation area. This place contains those particles whose attributes are calculated by the client. The particles in the outgoing zone are in the calculation area but they are neighbors of an other node's own particles, so they are required to one (or more) other node(s). The incoming zone holds those particles which are in another node's calculation are, but are needed for this client to the edge calculations. These zones evolve the inner and outer areas.

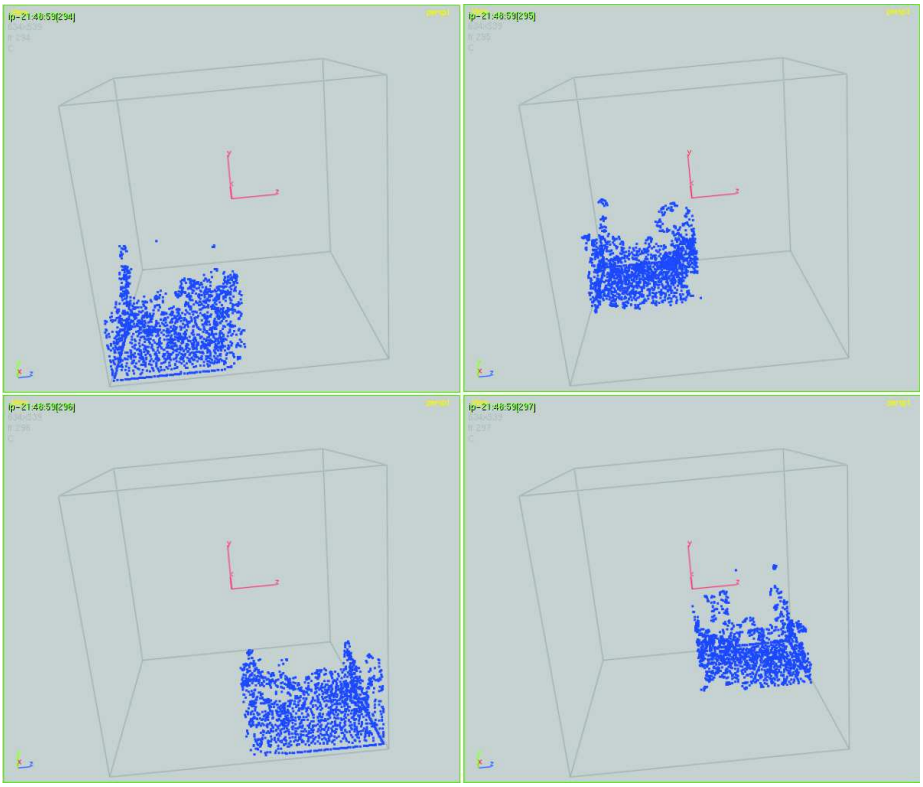

Figure 3.1: *Distributed fluid volume using 4 nodes*

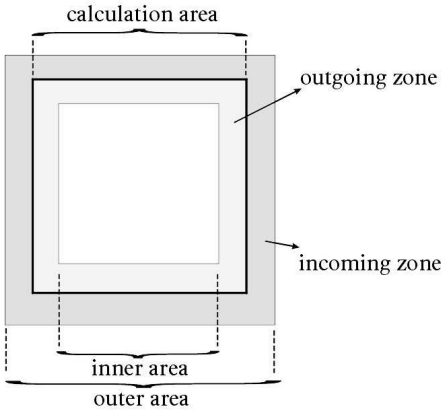

Figure 3.2: *Logical space partition of a client node*

3.3 Data Transfer

The outgoing zone of a client is the incoming zone of other clients and the incoming zone belongs to those clients in the outgoing zone. Particles in this volume must be transferred through the network. We get the best performance if the width of these

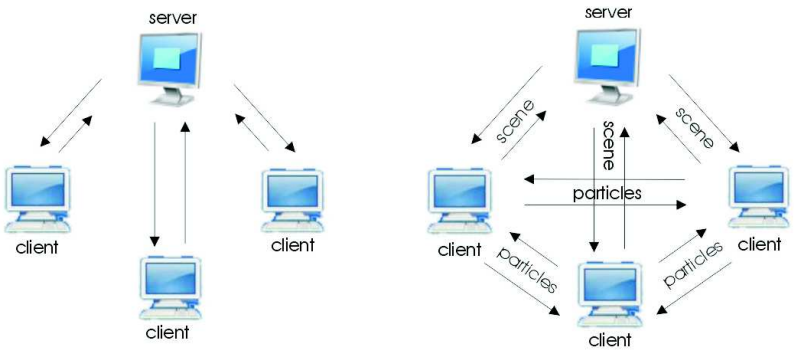

Figure 3.3: *Server — client and client — client communication models*

zones are the core value.

For accurate particle attribute values two communication steps are needed. At the beginning of the frame each client sends the location of the particles in their outgoing zone. Each client can set the neighbors and can start to calculate the local subvolume but local density and pressure values need global attributes from the neighboring node. We can see that the density and pressure values of the neighbors must be known to determine the inner forces. With a second data transfer these missing attributes will be sent and only these numbers should be transferred among nodes.

3.3.1 Peer-to-Peer Approach

Our *peer-to-peer* solution works without a head node that would store all the particles. At the initialization step each node generates its own particles in a specific volume and simulates the local particles on the GPU. Particles that should flow in or out of the volume are sent in Message Parsing Interface (MPI) [6] packets to the destination node.

1. Before data communication *each client is initialized* with the assigned scene slice. Subvolume calculation is based on MPI node identifier called MPI rank.

2. After the first data transfer *neighbor particle density and pressure values are calculated.*

3. The second communication step does not need synchronization because clients store the number of data they are waiting for. If the communication is completed, nodes can *calculate per particle forces, acceleration values, and the next position of each particle*.

4. The result is displayed without communication, *each node draws its own particles* and a *parallel compositing API merges* result images from the nodes.

3.4 HP Parallel Compositing Library

The *HP Parallel Compositing Library (ParaComp)* is a *sort-last parallel compositing* API suitable for *hybrid object-space screen-space decomposition*. The API was originally developed by Computational Engineering International (CEI) to make its products run efficiently in a distributed environments. The latest version is based on the abstract Parallel Image Compositing API (PICA) designed by Lawrence Livermore National Lab, HP, and Chromium team.

ParaComp is a *message passing* library for graphics clusters enabling users to take advantage of the performance scalability of clusters with network-based pixel compositing without understanding its inner structure and operation. The library makes it possible for multiple graphics nodes in a cluster to collectively produce images, thus significantly larger data sets can be processed and larger images can be created than on any individual graphics hardware by distributing the load over multiple nodes.

However, there is no explicit data distribution so no load balancing is done by the API. The philosophy of the designers is keeping the API as thin as possible. Therefore, only a *global frame* is defined and one or more nodes can contribute pixels to this frame and one or more nodes can receive a specified subset of the frame. ParaComp controls the operation of the nodes based on their request; it takes the results of their renderings and generates the needed composited images (see Figure 3.4). According to the nomenclature of the API a sub-image contribution is called *framelet* and the received image area is called the *output*. These framelets and the outputs can overlap each other without any restriction to their origin or destination nodes. The attributes of a framelet are the following:

- **horizontal and vertical position** in the global frame;
- **width and height** of the framelet in pixels;
- the **data source** which can be both the system memory and the frame buffer; and
- the **depth order** of the framelets which is needed by non-commutative compositing operators like alpha blending.

The size of the output does not necessarily equal the size of the global frame. For example, each tile can be connected to a separate node in a multi-tile display. The attributes of an output are:

- **horizontal and vertical position** in the global frame;

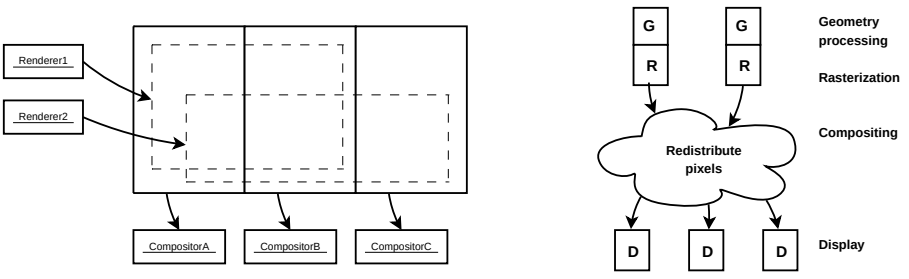

Figure 3.4: The operation of the *HP Parallel Compositing Library*

- **width and height** of the output in pixels; and
- the **pixel data** to be returned (RGB, RGBA, RGBA+depth).

For details see the official documentation of the HP Parallel Compositing Library [5].

Chapter 4

Installation and Usage of the Distributed SPH Fluid Simulator

The SPH fluid simulation application was implemented based on a very thin graphics library called Minimalist OpenGL Environment. This library was designed to handle the common issues of the development of a visualization application with the possibly maximal code reusability. This library has a parallel extension that eases the implementation of a parallel visualization application.

4.1 Installation

Both source and prebuilt versions of the application and the library can be found on the web site of the project[1].

4.1.1 Library Dependencies

The following libraries are required by the volume rendering application:

- **mingle**: Minimalist OpenGL Environment library (version 0.11)
- **mingle-parallel**: the Parallel Rendering extension of MinGLE (version 0.11)
- **paracomp**: Hewlett Packard implementation of the Parallel Compositing API (version 1.0-beta1 or later)
- **devil**: Developer's Image Library (version 1.6.7)
- **glew**: OpenGL Extension Wrangler library (version 1.3.4 or later)
- **Cg** and **CgGL** : NVIDIA Cg library
- **gl**: library implementing OpenGL API
- **glu**: OpenGL Utility Library

[1] `http://amon.ik.bme.hu/sphfluid/`

- **glut**: OpenGL Utility Toolkit

There are prebuilt packages for HP XC V3.2 RC1 platform for AMD64 architecture on the web site of the project for Developer's Image Library, OpenGL Extension Wrangler, Cg, CgGL, MinGLE, and MinGLE-parallel libraries. If one of them is missing from the target system, it can be installed in the usual way using the `rpm` package manager program:

```
# rpm -i devil-1.6.7-1.x86_64.rpm
# rpm -i devil-devel-1.6.7-1.x86_64.rpm
# rpm -i glew-1.3.4-1.x86_64.rpm
# rpm -i glew-devel-1.3.4-1.x86_64.rpm
# rpm -i Cg-1.5.x86_64.rpm
# rpm -i mingle-0.11-1.x86_64.rpm
# rpm -i mingle-devel-0.11-1.x86_64.rpm
# rpm -i mingle-parallel-0.11-1.x86_64.rpm
# rpm -i mingle-parallel-devel-0.11-1.x86_64.rpm
```

The `XXX-devel-YYY.rpm` packages are only needed when the SPH fluid simulation application is built from sources. Otherwise, only the shared libraries are to be installed.

The other libraries like the Parallel Compositing library, the standard C/C++ libraries, and the OpenGL libraries are platform specific and have to be installed based on the actual software stack.

4.1.2 RPM Package

The SPH fluid simulation (`sphfluid`) can be also installed from a prebuilt RPM[2] package in the same way:

```
# rpm -i sphfluid-0.1-1.x86_64.rpm
```

4.1.3 Building from Sources

The build system of the SPH fluid simulation application is based on GNU Autotools. So, it can be built with the usual procedure:

```
$ ./configure --with-inc-dir=<additional include directory> \
 --with-lib-dir=<additional library directory>
$ make
$ sudo make install
```

Since the only implemented parallel rendering support is the HP Parallel Compositing Library, it must be enabled. On a 64-bit HP XC platform the additional path values are the following:

- `<additional include directory>` = `/opt/paracomp/include`
- `<additional library directory>` = `/opt/paracomp/lib64`

[2] Red Hat Package Manager

MinGLE and MinGLE-parallel libraries can be also built from sources as follows.

Building MinGLE from Sources

The build system of Minimalist OpenGL Environment library is also based on GNU Autotools:

```
$ ./configure
$ make
$ sudo make install
```

Currently MinGLE supports only the GLUT windowing system. Hence, OpenGL headers and GLUT headers are needed. MinGLE is customizable, each feature can be disabled in the following way in the configuration step:

```
$ ./configure --disable-glew \
              --disable-devil \
              --disable-freetype
```

However, please note that the SPH fluid simulation application uses OpenGL extensions, therefore OpenGL Extension Wrangler support should not be disabled. Please also note that the application has a graphical user interface that requires font rendering, so Developer's Image Library is also needed. Nevertheless, FreeType support can be disabled if necessary, since the fonts are read from precalculated image files.

Building MinGLE-parallel from Sources

The parallel extension can be built and installed with the following configuration options:

```
$ ./configure \
 --with-inc-dir=<additional include directory> \
 --with-lib-dir=<additional library directory>
$ make
$ sudo make install
```

The meaning of the path options is the same as the volume rendering application.

4.2 Usage

The SPH fluid simulator can be executed in parallel mode using the SVA subsystem of the visualization XC clusters. The user interface of the program is simple; the navigation can be performed and force direction can be changed using the gui slider.

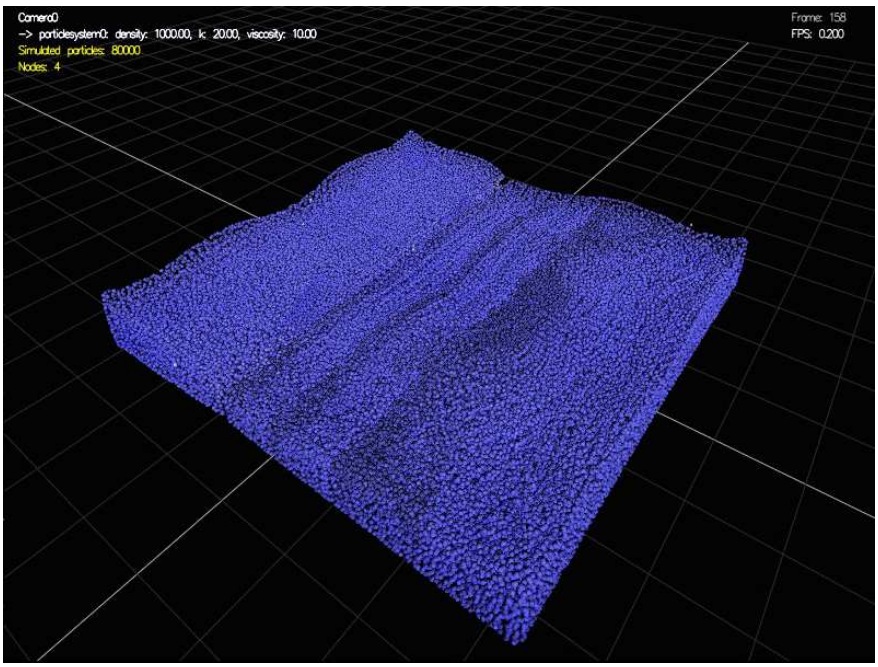

Figure 4.1: User interface of SPH fluid simulation application

4.2.1 SVA Startup Script

A SLURM[3] startup script is provided to use `sphfluid` for parallel rendering. It can be invoked with the following command:

```
$ sphfluid-hpxc.sh -r|--render <renderers>
```

The startup script has a parameter that should be set. The (`--render`) tells SLURM the number of *additional render nodes* to be allocated.

4.2.2 User Interface

The user interface of the SPH fluid simulator is rather simple (see Figure 4.1). There is a *frame* and *frame rate indicator* (a) on the right side which displays the current frame and the frame rate based on the last simulation steps. On the left side the *number of nodes* and the *physical parameters* of fluid volume *like density, viscosity* are displayed.

The navigation can be performed using the mouse in the following way:

- the *left button* can be used for *rotating* the scene,
- the *right button* is for *zooming*, and
- the *middle button* can be used for *translating* the scene.

The following hotkeys are defined:

[3]SLURM is an abbreviation for Simple Linux Utility for Resource Management. It is an open-source resource manager designed for Linux clusters of all sizes. This software solution is used for HP-XC clusters.

- Esc quits from the program,
- Tab toggles the user interface,
- C selects next camera,
- R resets camera,
- G toggles viewport grid

Chapter 5

Program Structure

The main parts of the distributed SPH application are the following:

- there is a specific **scene renderer** like presented in Listing A.1 (see Appendix A) in order to fit to the MinGLE system,
- there are specific classes responsible for **particle volume handling**,
- **neighbor** calculation is accelerated using a uniform grid structure,
- the **shader handling** is performed in designated classes,
- the **Cg shader sources** are in text files, and finally,
- a **GUI sheet** is responsible for tuning the shader parameters.

5.1 Container Object

Container object is a class referencing the fluid volume. A container initializes its contents, creates the uniform grid structure, sets cell sizes, simulates and moves particles and draws the fluid.

Files: `container.[h|cpp]`

5.2 FluidSim Application

`fluidsim` is the application class. The object initializes *OpenGL* including lights, cameras. It displays simulation results, screen texts, and FPS data.

Files: `fluidsim.[h|cpp]`

5.3 gpuSPH

This class is responsible for simulating particles with the *SPH* method. *gpuSPH* is the GPU-based fluid solver. The class initializes the *CG* context, generates textures from the particle data. The solver computes

- the distance between particles,

- density and pressure attributes for the particle points,
- forces, acceleration and collision,
- velocity values,
- position values.

The object can read back particles to the CPU for network communication.
Files: `gpuSPH.[h|cpp]`

5.4 Grid Cells

Grid cells are the basic components of the *uniform grid* computed on the CPU. Particles can be added or removed using *grid cell objects*.
Files: `gridcell.[h|cpp]`

5.5 Particle

SPH approach describes the fluid as a *particle system*. A *particle* with quantities represents the local fluid properties. A *particle* has mass, position, velocity, acceleration, density, pressure. For administration purposes particles have ids, neighbour number parameters.
Files: `particle.[h|cpp]`

5.5.1 Particle System

Particle System represents the fluid volume. It creates particles inside the specified container volume, adds particles to cells, sets particle neighbors, supervises the communication process among particle system parts on separate nodes, and draws the particles.
Files: `particlesystem.[h|cpp]`

5.5.2 Scene

The *Scene* contains containers of particle systems. The scene is loaded from a description file and built during application initialization. The object is responsible for handling multiple cameras, and switches among them.
Files: `particlesystem.[h|cpp]`

5.5.3 GPU Shaders

GPU Shaders are implemented using Nvidia's CG language. Pixel shaders compute physical quantities like distance, density, pressure, force, acceleration, velocity, and position.
Shaders are listed in 2.7.1
Files: `compute_density.cg, compute_distance.cg, compute_force.cg, compute_position.cg, compute_velocity.cg, sph.cg`

Chapter 6

Results

In this section execution results are presented for a five-node HP SVA cluster. Each node had a dual-core AMD Opteron 246 processor and NVIDIA Quadro FX3450 graphics cards. The software environment was HP XC V3.2 RC1.

Table 6.1 shows an exact view of the distributed working. Based on the computation time in Table 6.1 the distributed implementation seems to be useful for scenes containing large number of particles. The bottleneck of the parallel implementation is limited GPU memory on a single node and the network traffic.

particles	1 node	2 nodes	3 nodes	4 nodes
3000	8 fps	7 fps	12.6 fps	12.8 fps
10000	3 fps	2.2 fps	3.0 fps	4.2 fps
20000	2 fps	1 fps	1.4 fps	2.6 fps
30000	1.6 fps	0.5 fps	1 fps	2.3 fps
80000	-	-	0.3 fps	0.4 fps

Table 6.1: *Results of the distributed GPU-accelerated GRID-based application. Value "-" means that result is not interactive or GPU is out of memory*

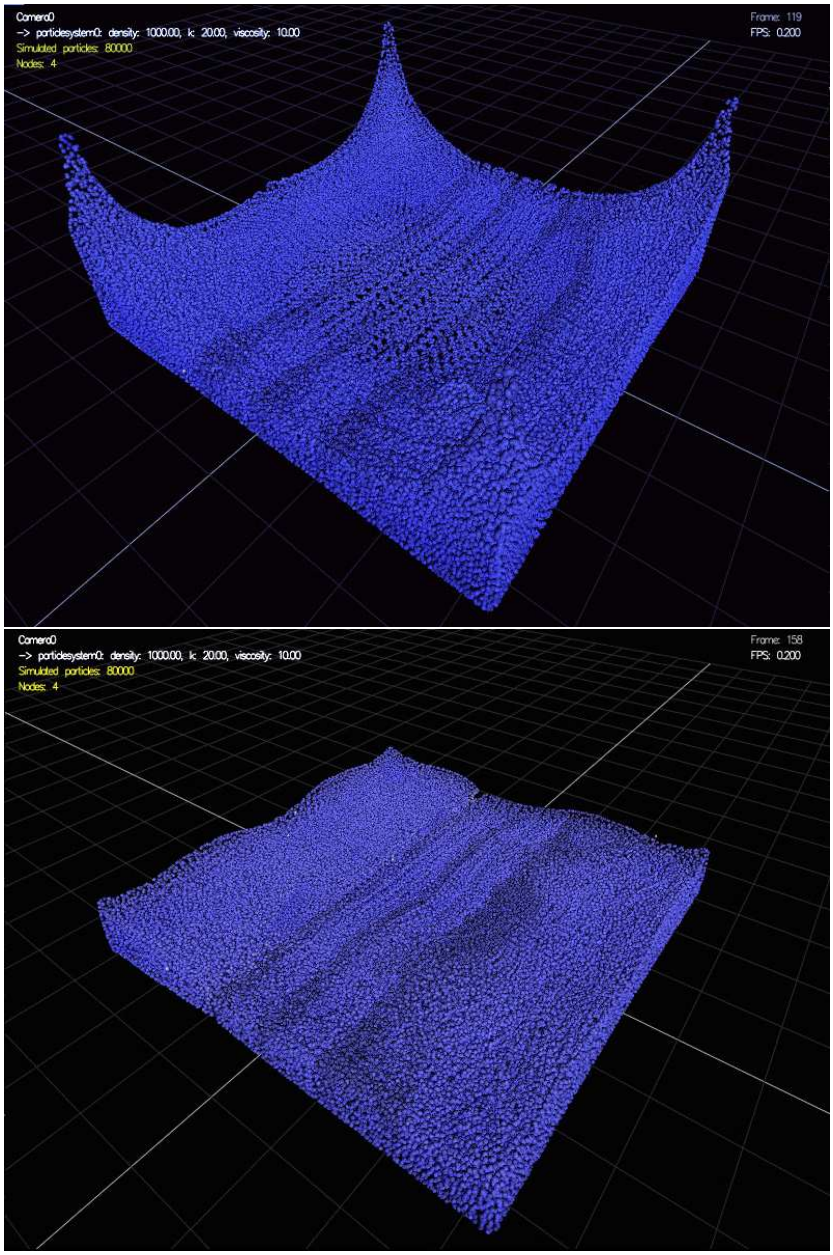

Figure 6.1: *Fluid simulation results computed by our distributed GPU-accelerated application. Subvolumes are rendered on 4 nodes running at 0.4 fps. Result is composited interactively using HP ParaComp Library*

Appendix A

MinGLE: Minimalist OpenGL Environment

The aim of this library is twofold. First, it forms a *thin object-oriented* window and event handling *wrapper layer* for GLUT, GLX, SDL and Windows systems. On the other hand, it contains basic helpers for some general common tasks involved in OpenGL based graphics applications, like camera handling and navigation, basic matrix operations, initializing OpenGL extensions, image handling, font rendering and simple user interface support, etc. Some of these features are implemented in-place and some of them use existing libraries. The overall goal is to provide platform independent aid for the very common tasks. A general API is defined, but at the moment only the GLUT platform is supported.

To get an impression of this library see the source code of a simple "Hello World!" application that renders a classic teapot object presented in Listing A.1. First, the singleton System object should be initialized. Next, a window is created to which several window listeners are added:

- our listener that overrides the `onRender()` method to render the teapot,
- a simple navigator that rotates, scales, and translates the scene based on the mouse interaction, and
- an application key handler that handles common keys for quitting, creating screen shots, and recording a frame sequence.

```
#include <mingle.h>
using namespace MinGLE;

#include <GL/glut.h>
#include <iostream>

// Custom window listener that does the rendering
class SceneRenderer : public WindowListener {
    protected:
        // This method is called when to render
        virtual bool onRender() {
            // Render a teapot using GLUT
            glutSolidTeapot (0.5) ;
```

```
            return true;
        }
};

int main(int argc, char **argv) {
    // Initializing the rendering system
    System:: initialize (&argc, argv);

    // Creating a window
    Window *win = System::createWindow();

    // Registering the window listener
    win->registerWindowListener(new SceneRenderer());

    // Adding a navigator
    ExaminerNavigator *navigator = new ExaminerNavigator();
    win->registerWindowListener(navigator);

    // Adding key handler
    ApplicationKeyHandler *keyHandler = new ApplicationKeyHandler();
    win->registerWindowListener(keyHandler);

    // Setting up OpenGL
    glEnable(GL_DEPTH_TEST);
    glShadeModel(GL_SMOOTH);
    glHint(GL_PERSPECTIVE_CORRECTION_HINT, GL_NICEST);

    GLfloat position [] = { 3.0f, 3.0f, 3.0f, 1.0f };
    GLfloat diffuse [] = { 0.8f, 0.8f, 0.8f, 1.0f };
    GLfloat specular [] = {1.0f, 1.0f, 1.0f , 1.0f};
    glEnable(GL_LIGHTING);
    glEnable(GL_LIGHT0);
    glLightfv(GL_LIGHT0, GL_POSITION, position);
    glLightfv(GL_LIGHT0, GL_DIFFUSE, diffuse);
    glLightfv(GL_LIGHT0, GL_SPECULAR, specular);

    // Entering the event handing loop
    System::enterMainLoop();

    return 0;
}
```

Listing A.1: Hello World! application using MinGLE

A.1 MinGLE Parallel: Parallel Extension for MinGLE

The parallel extension of MinGLE provides general support for sort-last parallel rendering in applications based on MinGLE. It *sets up compositing contexts*, *adds framelets*, *receives outputs*, and *transmits window events* to each application instance running in parallel.

The parallelized version of the previous program is presented in Listing A.2. For simplicity neither the teapot nor the screen is divided, but when more rendering nodes are added the programs renders more colored teapots appear along a circle line. All generated mouse and keyboard events are automatically transferred to the slave nodes that receive these events as if they would have been generated by real user interaction. The overall feeling of the user is that one application is running that renders several teapots.

```
#include <mingle.h>
#include <mingle-parallel.h>
using namespace MinGLE;

#include <GL/glut.h>
#include <iostream>

#include <math.h>

class SceneRenderer : public WindowListener {
    protected:
        int mThisRenderer, mRendererCount;
        double mPosition[3];

    public:
        SceneRenderer(int thisRenderer, int rendererCount) :
            mThisRenderer(thisRenderer), mRendererCount(rendererCount) {
            // Setting up position of the object
            mPosition[0] = 0.5 * ::cos(2.0*M_PI/mRendererCount*mThisRenderer);
            mPosition[1] = 0.0;
            mPosition[2] = 0.5 * ::sin(2.0*M_PI/mRendererCount*mThisRenderer);
        }

        virtual bool onRender() {
            glMatrixMode(GL_MODELVIEW_MATRIX);
            glPushMatrix();
            glTranslatef(mPosition[0], mPosition[1], mPosition[2]);

            // Drawing teapot with unique color
            int i = mThisRenderer+1;
            glColor4f( (i&1) ? 1.0 : 0.0,
                       (i&2) ? 1.0 : 0.0,
                       (i&4) ? 1.0 : 0.0,
                       1.0/mRendererCount * i);
            glutSolidTeapot(0.1);
            glPopMatrix();

            return true;
        }
};

int main(int argc, char **argv) {
    // Initializing the rendering system with parallel support
    System::initialize(&argc, argv);
    ParallelRenderingSupport::initialize(&argc, argv);

    // Creating a window
    Window *win = System::createWindow();
    win->setRenderMode(Window::RENDER_WHEN_IDLE);

    // Adding parallel rendering support to the window
    // Master/slave mode is auto-detected using the command line arguments
    ParallelRenderingSupport::addParallelSupport(win);

    // Adding key handler
    win->registerWindowListener(new ApplicationKeyHandler());

    // Registering scene renderer
    SceneRenderer *sceneRenderer = new SceneRenderer(
        ParallelRenderingSupport::getThisRenderer(win),
        ParallelRenderingSupport::getRendererCount(win)
```

```
    );
    win->registerWindowListener(sceneRenderer);

    // Add navigator
    win->registerWindowListener(new ExaminerNavigator());

    // Setting up OpenGL
    glEnable(GL_DEPTH_TEST);
    glShadeModel(GL_SMOOTH);
    glHint(GL_PERSPECTIVE_CORRECTION_HINT, GL_NICEST);

    GLfloat lightPosition [] = { 3.0f, 3.0f, 3.0f, 1.0f };
    GLfloat diffuse [] = { 0.8f, 0.8f, 0.8f, 1.0f };
    GLfloat specular [] = {1.0f, 1.0f, 1.0f , 1.0f};
    glEnable(GL_LIGHTING);
    glEnable(GL_LIGHT0);
    glEnable(GL_COLOR_MATERIAL);
    glLightfv(GL_LIGHT0, GL_POSITION, lightPosition);
    glLightfv(GL_LIGHT0, GL_DIFFUSE, diffuse);
    glLightfv(GL_LIGHT0, GL_SPECULAR, specular);

    // Entering the event handing loop
    System::enterMainLoop();

    return 0;
}
```

Listing A.2: Hello World! application using MinGLE-parallel

For implementing distributed applications a general API is defined, but at currently only the ParaComp library is supported. The applications based on the parallel extension library contains both the master and the slave parts of the visualization program. When writing the code this master-slave differentiation is hidden by the underneath MinGLE-parallel library. However, the application has to be executed in two different modes in order to exploit the benefits of parallel rendering power:

```
application <sessionid> <master> <slave1> <slave2> ... <slaveN>
```

for master mode, and

```
application <sessionid> <slave_i>
```

for slave mode. Note that using an application startup script the overall distributed startup can be done in one step, too. See Section 4.2.1 for the startup script of TextureVR designed for SVA and ParaComp.

Bibliography

[1] Y. Yoshihiro M. Yoshitsugu A. Takashi, I. Masataka and C. Kunihiro. Particle-Based Fluid Simulation on GPU. *Research Paper*, 2003.

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

[3] Tom Duff. Compositing 3-D rendered images. In *SIGGRAPH '85: Proceedings of the 12th annual conference on Computer graphics and interactive techniques*, pages 41–44, New York, NY, USA, 1985. ACM Press.

[4] J. Foley, A. van Dam, S. Feiner, and J. Hughes. *Computer Graphics, Principle and Practice*. Addison-Weseley, 1993.

[5] Hewlett Packard. *HP Scalable Visualization Array Parallel Compositing Library Reference Guide*, 2007.

[6] http://www-unix.mcs.anl.gov/mpi/. *The Message Passing Interface (MPI) standard*.

[7] Greg Humphreys, Matthew Eldridge, Ian Buck, Gordan Stoll, Matthew Everett, and Pat Hanrahan. WireGL: a scalable graphics system for clusters. In *SIGGRAPH '01: Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pages 129–140, New York, NY, USA, 2001. ACM Press.

[8] Williams J. L. and Hiromoto R. E. A proposal for a sort-middle cluster rendering system. In *Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications, 2003. Proceedings of the Second IEEE International Workshop*, pages 36– 38, 2003.

[9] 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.

[10] M. Gross. M. Mller, D. Charypar. Particle-Bsed Fluid Simulation for Interactive Applications. *Eurographics/SIGGRAPH Symposium on Computer Animation*, 2003.

[11] Steven Molnar, Michael Cox, David Ellsworth, and Henry Fuchs. A Sorting Classification of Parallel Rendering. *IEEE Comput. Graph. Appl.*, 14(4):23–32, 1994.

[12] Carl Mueller. The sort-first rendering architecture for high-performance graphics. In *Symposium on Interactive 3D Graphics: Proceedings of the 1995 symposium on Interactive 3D graphics*, pages 75 – ff, New York, NY, USA, 1995. ACM Press.

[13] D. Pnueli and C. Gutfinger. Fluid Mechanics. *Cambridge University Press*, 1992.

[14] Thomas Porter and Tom 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.

[15] K. D. Vertanen. A Parallel Implementation of a Fluid Flow Simulation using Smoothed Particle Hydrodynamics. *Research Paper*, 1999.