UNIVERSITY OF ZAGREB

Faculty of Electrical Engineering and Computing

Master's Thesis no. 103

Comparative Rendering of Simulation Scenarios

Hrvoje Ribičić

Zagreb, September 2010.

Acknowledgments

First of all, I would like to thank my mentors, who provided leadership and inspiration when it was needed the most. Jürgen Waser and Raphael Fuchs helped out with both the theory and implementation, and were willing to put aside a large amount of time for discussion and assistance. Their help was invaluable, and I feel that I've learned a lot from the time spent with them. Meister Eduard Gröller accepted me as his student, and provided insightful comments about the thesis. His ability to spot weak spots in a thesis is both useful and frightening. My mentor at my home university, Željka Mihajlović, provided valuable comments and references. I'd like to thank her for always being supportive and helpful, and especially so during the writing of this thesis.

It is also necessary to mention the people I've worked with - thanks go out to the entire Visdom team, in particular to Benjamin Schindler, whose coding and software engineerings skills I hope to reach someday. I would also like to thank the VRVis for allowing me to do my internship there, preparing me for the thesis. The ERASMUS programme allowed me to visit Austria and led to this thesis, so I'd like to thank all the people involved in the exchange programme as well.

Finally, I would like to thank my family and friends, who shared the good and the bad times with me. Without them, I have no doubt that this thesis would not have been completed.

This thesis was supported in part by a grant from the Austrian Science Fund (FWF):P 22542-N23 (Semantic steering).

Contents

1 Introduction **1**

2 Basics and Related Work **4**

2.1 SPH . . . 4
2.2 PhysX . . . 7
2.3 Architecture and framework description . . . 7
2.3.1 Visdom . . . 7
2.3.2 World Lines . . . 9
2.4 Earlier work in comparative rendering . . . 11
2.4.1 Visualization of multi-variate data . . . 13
2.4.2 Visualization of time-varying data . . . 14
2.4.3 Visualization of ensemble data . . . 15
2.5 CUDA . . . 15

3 Simulation handling **17**

3.1 Simulation system . . . 17
3.2 Abstract simulation node . . . 19
3.3 PhysX simulation system . . . 23

4 Scene rendering **26**

4.1 Rendering subsystem . . . 26
4.2 Geometry rendering . . . 29
4.3 Particle rendering . . . 29

5 Aggregate renderings **35**

5.1 Aggregate rendering pipeline . . . 35
5.2 Flood exposure . . . 37
5.2.1 Measuring exposure . . . 37
5.2.2 Results aggregation and visualization . . . 40
5.3 Fluid property visualization . . . 41
5.3.1 Property texture extraction . . . 42
5.3.2 Texture application . . . 46
5.3.3 Comparative rendering . . . 47

6 Results and discussion **52**

6.1 Performance . . . 52
6.2 Usability . . . 53

7 Conclusion **55**

7.1 Future work . . . 55
7.1.1 Exploration in aggregate rendering . . . 56
7.1.2 Multiple gradual events use . . . 57

1 Introduction

One of the most disastrous floods in recent history occured in New Orleans as an aftermath of the hurricane Katrina. New Orleans is situated below the sea level, and its flood protection system consists of a series of flood walls and levees channeling the flow of the water. After the passing of hurricane Katrina, multiple breaches in the flood protection system occured and more than 80 percent of the city was submerged in water. One of these breaches, situated in the 17th Street Canal, was responsible for most of the flooding in the city and had to be closed for the restoration efforts to continue. The army attempted to close the breach by dropping large bags of sand, but soon found that the task was not straightforward [26]. Bags failed to make a stable barrier and were washed away, and multiple tries were needed to close the barrier.

This work was developed around a case study of breach closure procedures based on the events that took place in New Orleans [32]. In the case study, various procedures were tried on a scale model of the breach, and several possible solutions were found. The idea of the research that this thesis is a part of was to create software support that would allow for similar experimentation in a virtual enviroment. The behavior of the system is based on simulation and visualization techniques applied to the results to allow for better insight into the problem.

The main characteristic of the case study that our research was trying to improve upon was experimentation. In the study, many various approaches to sealing the breach were tried to see how they performed. Even though some of these approaches differed only in parts of the procedure, each one required the study to be started anew. The ability to explore how various choices taken at a certain time affect the state of the system is hard to achieve in real-world laboratory models. This is due to the difficulty of returning the system to a previous state and the work involved in doing so. However, a virtual enviroment allows for saving and restoring states of a system, which gives the user the ability to easily check how some changes would affect the system. In comparison to the laboratory conditions, the cost of experimentation is decreased. The user can afford to explore possibilities that would have been too expensive to pursue in a laboratory setting, which may lead to new insight about the system's behavior.

The type of exploration in which the user fine-tunes the parameters of the simulation until a desired result is reached is called computational steering. Previously developed solutions like SCIrun allowed the user to apply small modifications to the parameters of a simulation in progress, but major changes required the simulation to be stopped and started anew [25]. In our solution, the user manipulates simulation runs, which are executions of simulations with different parameters and starting states. The user can monitor simulation runs, control their execution and even change the settings at any point of the run. Setting changes do not require the simulation to be restarted, allowing the user to consider multiple alternatives at the same time. The introduction of computational steering requires the development of visualization tools that will allow the user to deal with the additional information and complexity that emerges. The first tool created was World Lines, which provides a representation of the simulation flow and of the choices the user has made, and will be talked about in more detail later on. The tools this thesis focuses on are related to the comparison of simulation scenarios. The ability to experiment

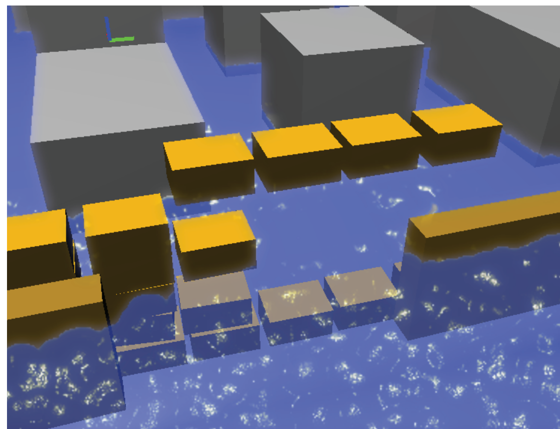
produces many different states of the system, and while the user may experiment blindly, to use the system efficiently there is a need to visualize the relationship of different states.

The prerequisite for the development of these new tools is the software support for the simulation and rendering of a flooding scenario. The simulation engine we use has been modified to work with our framework, and extended to allow for parameter changes and state saving and retrieval. In our extended engine, these operations work by using the least amount of resources possible, and provide a formal model that allows them to be understood and extended easily. While we could not exactly reproduce the laboratory conditions because of the complexity of the experiment, we developed a simulation scenario that manages to capture the essential parameters of the problem. The scenario is accompanied by views capable of rendering it. The views need to visualize the scenario accurately and realistically enough for it to resemble the laboratory conditions. The resulting simulation scenario allows users to perform experiments in an enviroment that behaves and looks similar to the actual procedure and the laboratory recreation (see Figure 1).

Flooded area

Real-world procedure

Laboratory recreation



Simulated procedure

Figure 1: The differences and similarities of the procedures

Some experimentation with the simulation scenario revealed what properties these views need to posess. A realistic view of the simulation scenario does not show how successful the user is in closing the breach, nor does it show the hidden properties of the fluid.

A measure of success is needed to help guide the user's experimentation towards more favorable solutions. However, the complexity of the problem is such that the exploration is not straightforward. One example is the behavior of barriers under stress. Barriers that are considered stable break when more pressure is put on them later on, and the user needs to be able to understand the fluid's behavior in order to predict such events. The prerequisite for gaining such an understanding is a tool that shows the parameters of the simulation which are invisible in a realistic rendering of the fluid.

An additional challenge stems from the fact that these views are to be used with multiple system states. To avoid having the user manually compare the results, the views need to be modified to allow handling of multiple states. To this end, we have developed aggregate renderings that serve two purposes. They extract the similarities of all selected states, and allow a single simulation state to be compared with the created combination. This allows the user to gain insight into both the similarities and the differences of states.

The final goal of the system is to create an enviroment in which the user can perform complex experiments to find plausible solutions to real-world problems. While our tools cannot help with the inherent difficulty of the scenario being explored, our hope is that they will help reduce the complexity of managing such an exploration and help the user gain new insights.

2 Basics and Related Work

To succesfully create a simulation scenario similar to the laboratory procedure, it is necessary to prepare for the task by studying various previous solutions that might be useful. The first problem is deciding on the method of fluid simulation that would be best for replicating the conditions. We chose the SPH method, but instead of writing our own simulation engine capable of simulating both fluids and rigid bodies, the PhysX engine was integrated into the Visdom visualization framework. The integration was done in a generic way, by creating an interface usable with different simulation engines. The design of this interface is such that it can work with tools for controlling simulations already present in the framework. Since the specifics of the framework and these tools affected the development of the simulation infrastructure, they will be described in this section in more detail. The choice of framework and simulation method also defined the format of the data that was output, making it possible to look into various methods used to comparatively visualize data.

2.1 SPH

The method of simulation used to model the behavior of water in the flooding scenario is called Smoothed Particle Hydrodynamics (SPH). It was originally developed in astrophysics by Monaghan[14] and Lucy[20] to study the dynamics of the behavior of gases. Later on it was adapted to allow for simulations of incompressible fluids as well. A description of the method and its development can be found in Monaghan's review of the SPH method and its uses [21].

The basic idea of any fluid simulation method is to create a discrete representation of the inherently continuous fluid that can be used to predict the future states of the fluid. In traditional computational fluid dynamics (CFD), the space that the fluid occupies is divided into smaller cells, creating a mesh or a grid. The interaction between neighbouring cells can be described using Euler or Naiver-Stokes based differential equations. Given the initial state of the cells, numerical methods can be used to solve the equations and predict the future states of the cells. In contrast to traditional methods, SPH discretizes the fluid by representing it as a number of particles. Each one of them has certain physical properties e.g. density, velocity, temperature. The behavior of the fluid can then be modeled by considering the pairwise interactions between particles. This problem is known as the n-body problem, and is often encountered in physical modeling, where many efficient solvers for it already exist [6] [1].

The difficulty of SPH modeling lies in converting the discrete particles to a continuous description of a fluid. We expect to be able to evaluate the value of some property of the fluid at any point that belongs to the fluid. For traditional CFD methods, finding out this value requires an interpolation between the cells of a grid near the point of interest. In the SPH method, a single point in the fluid can be affected by multiple particles. While some processes like rendering can work with the particle representation of the fluid, the model would be incomplete without the ability to determine the value of one of the properties of the fluid at any point. SPH resolves this by defining ranges of influence of

individual particles and combining them into one continuous field from which values can be extracted. The range of influence of a single particle is defined by its kernel function. One such kernel function is the cubic function, named after the degree of the polynoms used:

$$W(\vec{r}, h) = \frac{\sigma}{h^v} \begin{cases} 1 - \frac{3}{2}q^2 + \frac{3}{4}q^3 & if\ 0 \leq \frac{\vec{r}}{h} \leq 1; \\ \frac{1}{4}(2-q)^3 & if\ 1 \leq \frac{\vec{r}}{h} \leq 2; \\ 0 & otherwise \end{cases} \tag{1}$$

where q equals:

$$q = \frac{\|\vec{r}\|}{h} \tag{2}$$

The kernel function takes two values as input - the vector distance $\vec{r}$ and the smoothing length of the particle h. As can be seen from Equation 2, the smoothing length defines how the input vector $\vec{r}$ is scaled. The value of the kernel function monotonously drops as the norm of $\vec{r}$ increases, and reaches zero when the norm exceeds two smoothing lengths.

When the function argument $\vec{r}$ is taken to be the distance from the center of a particle, the kernel function describes how the influence of that particle changes in space. Since the value of the kernel function drops to zero after a certain distance, the influence of individual particles is localized to an area defined by their smoothing length. To ensure that the contribution of each particle is equal regardless of the smoothing length size, the integral of the kernel function over its area of definition is required to equal one. Depending on the number of dimensions v present in the simulation, the constant σ in Equation 2.1 is chosen to fulfill that requirement by normalizing the value of the integral to 1.

Using Equation 2.1 mentioned earlier, evaluating a property A of the fluid at some point $\mathbf{r}$ can be done by summing up the contributions of all the particles using the formula:

$$A(\vec{r}) = \sum_j m_j \frac{A_j}{\rho_j} W(|\vec{r} - \vec{r}_j|, h_j) \tag{3}$$

where m_j is the mass of the j-th particle, ρ_j is its density, h_j its smoothing length and $\mathbf{r}_j$ its position. Since this formula can be evaluated at any point in space, it allows the discrete particles to be interpreted as a scalar field. Figure 2 shows how kernels are combined to create to find particle properties. If it is necessary to find out which points belong to the fluid, its border can be found as the set of points $\vec{r}$ where the following equation holds:

$$\sum_j W(|\vec{r} - \vec{r}_j|, h_j) = 0.5 \tag{4}$$

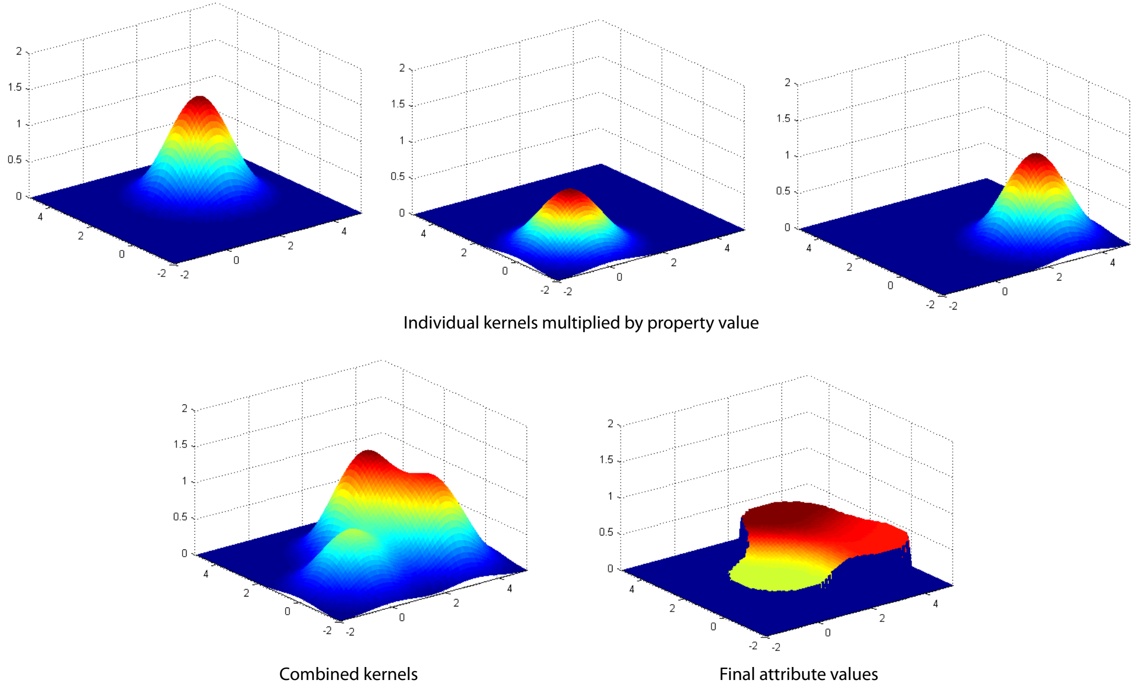


Figure 2: A property field of three particles extracted using SPH interpolation

Because kernel functions limit the effects of a particle to an area defined only by its smoothing length, it is possible to evaluate Equation 3 more efficiently. This is done by limiting the number of particles taken into consideration by using a space-partitioning structure such as a KD-tree or an octree. As can be seen in Equation , when evaluating the kernel function, only particles whose distance to the point in question is lesser than h_j return a value different from zero. The contribution of other particles to the sum in Equation 3 can be ignored. Space-partitioning structures can eliminate some distant enough particles whose contribution would be zero, speeding up the calculation. As the repulsive forces inherent to the SPH simulation work to spread the particles apart evenly, such a strategy can work very well and prevent many unnecessary calculations.

While the grid-based methods often allow for greater precision and can model more complex phenomena [4], the SPH method works well in situations where a free surface is involved. The advantages of using SPH in modeling the simulation scenario come from its meshlessness and the particle-based interaction model. The meshlessness allows for quick changes in the shape the water may take, which is vital to simulating levee breaches and adding barriers during the simulation. The particle-based interaction model allows for easy interaction with dynamic objects, making barriers breakable. Both of these advantages are vital for running a flood simulation scenario, and the SPH method has been successfully applied to such scenarios previously [13].

2.2 PhysX

The interaction between the geometry and the fluid that the scenario requests severely limits the number of engines that can be used to implement the simulation scenario. The simulation engine chosen to model the behavior of all the objects in the scenario is NVIDIA's PhysX [24]. Originally made to enhance game physics by using the power of modern GPUs, PhysX allows for the simulation of various physical phenomena - rigid bodies, soft and deformable bodies, fluids, and more importantly, provides collision detection and allows for interaction between objects in the scene. While the engine was not built with the accuracy of simulations as a primary goal, it supports and produces realistic enough models to be used to reproduce the scenario from the case study with sufficient complexity. An additional advantage is that in contrast to engines like SPHysics [1], the preprocessing phase required for efficient collision detection used in rigid body physics can be done at runtime. This allows changes to geometry such as levee breaches to happen without losing the benefits of fast collision detection.

However, the PhysX engine has some restrictions which make it more difficult to use in a steering enviroment. Most parameters that regulate the physical properties of objects can only be set at the time of the objects' creation. This applies to fluid properties, rigid body properties, and the parameters that govern interactions between the two. Since the very idea of steering while considering alternatives requires that the parameters be changed during a run, workarounds must be used to bypass this restriction. PhysX allows the user to extract all the relevant data from objects, making it possible to use the existing object data to create another set of objects with the same physical properties apart from the parameters that need to be changed. The same approach needs to be used when the static geometry is changed, as the collision information is updated only when objects are created. While this approach allows parameter changes to be applied, the recreation of objects is expensive due to the graphics card use, and should be avoided whenever possible.

Additional restrictions related to the simulation scenario stem from the different types of objects used. PhysX does not allow every type of interaction, making it impossible to use it to the fullest. As an example, while it would have been better to model the bags as deformable rather than rigid, interactions between particles and soft objects are disallowed. Because of this, bags need to be modelled as rigid bodies, lessening the realism of the scenario. Even though PhysX has its restrictions, it still offers a very large number of features that other SPH simulation engines do not have, making it the best choice for the simulation of the scenario.

2.3 Architecture and framework description

2.3.1 Visdom

The implementation part of this thesis was done as an extension of the Visdom framework. Visdom is a visualization framework that allows a user to create a visualization pipeline from premade and self-supplied components. It is developed jointly by the ETH Zurich

and VRVis, and has seen use in various research projects and the development of several scientific papers [36][12].

The architecture of the Visdom framework is based on a two-tiered server-client model with a thin client. The client controls the work-flow, displays the results, and handles user interactions while all the calculations, rendering, and data management are done on the server. The separation of the computationally intensive part from the interaction-heavy one is meant to allow users access to the framework from low-powered and mobile devices. The differences between the server and the client have influenced the choice of the technologies used for their implementation. The server is written in C++ for reasons of performance and to ease algorithm and library reuse, and the client is written in the Flash-based Flex framework which allows for quick implementation of rich user interfaces [3]. Due to the nature of libraries used, the server runs on x64 enviroments only, and is deployable on Windows and Linux platforms. The client works on all major platforms which support the Adobe AIR runtime [2].

The basic concept common to both server and client is the node. A node is the basic building block of the system. It can be seen as a data processing component with a certain number of input and output connectors. The client offers a design view in which the user constructs a dataflow by adding nodes and connecting the input and output connectors of nodes. The counterpart of the design view is the semantic view, which consists of semantic windows created by nodes. Each semantic window contains a view or an interactive component and is linked to other semantic windows, meaning that changes in one window may be reflected in others [30]. Additional changes to the way nodes work can be applied by changing the individual settings of the nodes, accessible by highlighting a node in the design view. A sample node layout can be seen in Figure 3.

The nodes can be divided into four major categories according to their function: producers, filter nodes, view nodes, and configuration nodes. Producers are nodes that have no input connectors and are considered the starting points of the data-flow pipeline. They either load data from external sources or produce it according to the provided settings. The majority of the nodes in the pipeline are filter nodes, which perform calculations on the inputs provided and create new data. The ending points of the pipeline are view nodes, which create semantic windows that serve as views. Configuration nodes extend the functionality of connected nodes by adding new settings or semantic windows to an existing node. They can be connected to multiple nodes at once by using the special top and bottom connectors. These connectors do not participate in the horizontal data-flow and can only be used for connecting the configuration nodes.

Once a user has finished arranging the nodes and wishes to execute the data-flow, a run request is issued from the client menu. A run request specifies what calculations the server must perform. It contains information about the nodes, settings and connections in an XML format. For every node described in the request, the server creates a corresponding node object capable of processing data, and executes the node when all of its connected inputs are available. Once the entire request is processed, the server sends back all the results the nodes have produced as a result of the request. The results are interpreted by the client and used to update the semantic windows. The created node objects persist as long as a request does not notify the server of their deletion. This feature allows the nodes to posess internal states and the server to preserve the state of the data-flow even

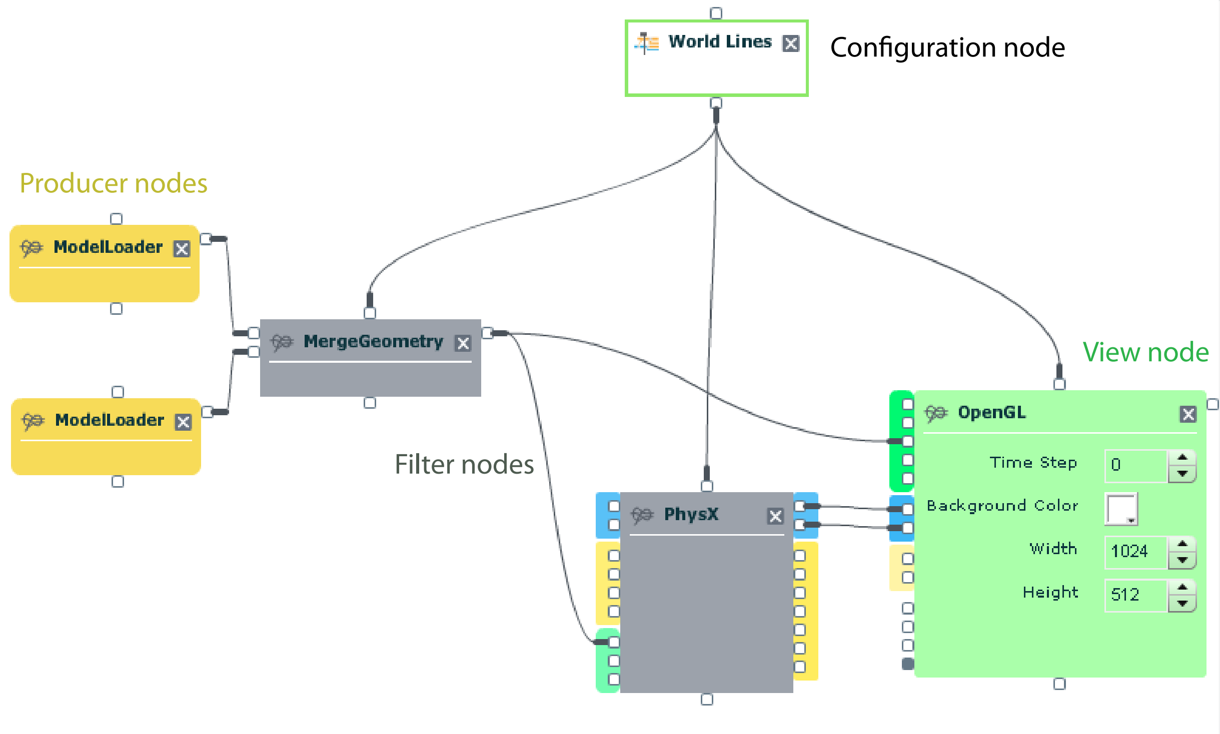


Figure 3: A screenshot of a data-flow setup

if the connection between the server and the client is lost.

Subsequent run requests that do not change the layout of the nodes cause the execution of only those nodes that need updating. The nodes that have to be updated are determined according to changes in settings and connected nodes. This feature is especially important for steering simulations, as it makes it possible to change the parameters of views without having to generate the data gained from the simulations again.

While this system encourages the reuse of nodes and provides many designed to perform common tasks, it is not always possible to decompose the desired functionality into nodes. More complex functionality such as rendering or simulation handling cannot be spread across multiple nodes. Rendering requires access to many common resources, and a decomposition would require that these resources are shared between multiple nodes. Since such a setup would be inefficient and hard to maintain, almost all the rendering code remains within a single node. Another special case involves nodes that have an internal state, such as simulation nodes. These nodes consider their inputs to be only a specification of the initial state of the system. As these special cases are relevant to this thesis, they will be explored in more detail later.

2.3.2 World Lines

World Lines is a visualization tool that allows the user to intuitively control the flow of simulations and explore alternative scenarios in one comprehensive view [36]. They are integrated into the client as a semantic window which appears when a World Lines

configuration node is created. This configuration node is connected to view nodes, and it controls the settings of the nodes.

World Lines show each simulation run within the window as a horizontal panel called a track. Different tracks take up different vertical positions, and the horizontal position and the length of a track are determined by the start and end time of the run it represents. Since different tracks can contain different settings and inputs, the time value alone cannot be used to uniquely identify the state of the system. The combination of track and time value called a frame is used instead. Clicking on a track sets the current active frame of the system according to the point clicked. The active frame defines what data values and settings are used to update all the semantic windows linked to World Lines.

In most cases, when a user begins to work with World Lines, no data will be present in the system. Each track in World Lines is colored to show which frames have been simulated. The user can specify which track he wants to execute and up to which point by setting the active frame and pressing the record button. The World Lines interface interprets the user's actions and breaks them down as a sequence of execution requests that are sent to the server. All the tracks on screen can also be simulated using a simulation schedule, which defines in which order tracks and frames are executed.

When World Lines is started, the only track present is the base track, which describes the settings of the simulation system. If the user has reached an interesting point in the simulation that he wishes to explore with different simulation parameters, he changes the settings of the system, creating a new track. This process is called branching, and involves changing exactly one parameter marked as steerable. While it could be possible to change multiple parameters while branching, the system is currently designed in such a way that only one setting can be changed at every branch. It is possible to continue simulation in both the original and the current track, and to do more branches in both tracks, allowing the exploration to continue. Figure 4 shows how the World Lines window looks like after a lengthy period of experimentation.

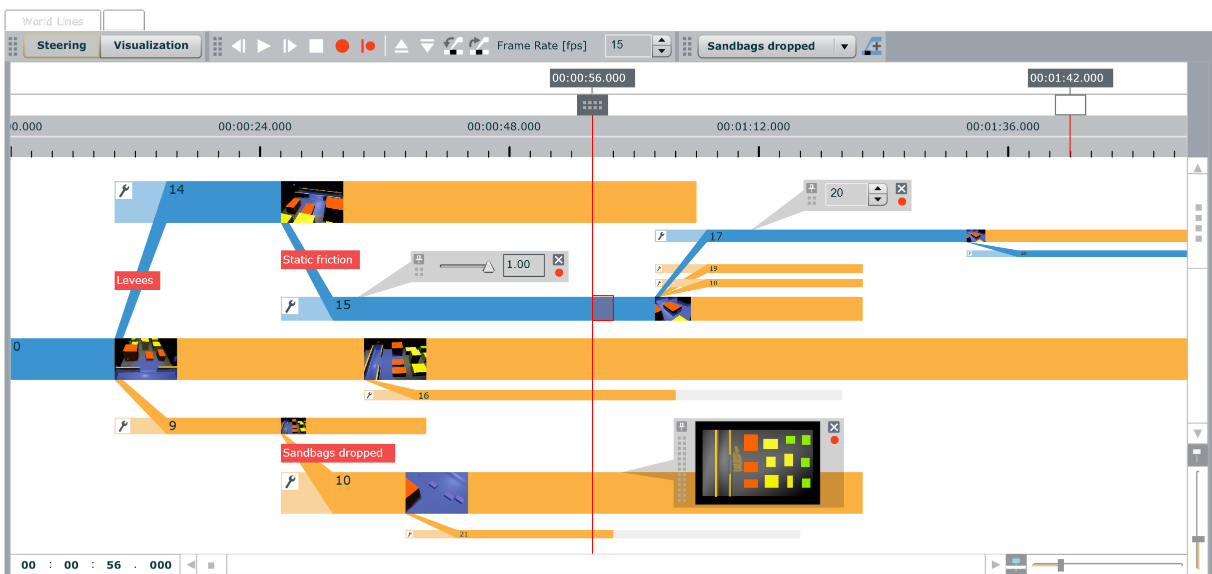


Figure 4: The World Lines window after a period of experimentation with the simulation scenario

It is also possible to change settings in a track without branching. If a user changes the setting value of an existing track, the entire track becomes invalid. The results already present for that track are no longer useful because they have been created with different settings. All children tracks also become invalid, and the effect propagates further, invalidating all descendants. This action causes the server to delete the data of the affected tracks, requiring a resimulation for the results to be visible again. The simulate schedule option is useful in such cases as it automatically resimulates all tracks.

Lengthy periods of experimentation can cause many tracks to be created, and World Lines provides mechanisms for dealing with the emerging complexity. One problem is screen clutter. With many created branches, it is possible that the user will not be satisfied with the automatic resizing or that the resizing will create a layout that does not emphasize the path of exploration the user is taking. To counter this problem, the World Lines interface allows the user to manipulate the tracks and rearrange them. The user can change the thickness and length of the tracks and their vertical positions. In addition, tracks can be collapsed into their parent tracks, hiding them to make the tracks important to the user more visible.

A remaining problem with using World Lines is how to learn from the experimentation. Continued experimentation requires constant comparisons of tracks to find valuable information. Examples include finding out in which tracks a certain building is not flooded or what causes a barrier breach to happen in certain tracks. Since this task is difficult and requires much manual labor, World Lines provides a mechanism that offers visually intuitive track ordering and coloring based on analysis criteria supplied by the user. This approach is shown in Figure 5.

The core of this approach are the analysis nodes. When given a state of the simulation system, the analysis node outputs a scalar value which describes the state according to some quality the analysis monitors. In a special visualization mode, the WorldLines can use this value to color the tracks using a transfer function and sort the tracks according to value, giving the user visual cues that show differences among tracks.

The analysis results can also be presented in linked views. In this work, special views have been developed which help the user understand the differences and similarities of the selected tracks. These comparative renderings have been designed with different approaches in mind, and some of them will be presented in the following section.

2.4 Earlier work in comparative rendering

The idea of using appropriate illustrations to compare sets of data is very old and has been applied successfully in information visualization for a very long time. The simplest approach towards visualizing the differences in data consists of using graphs or scatter plots to create an image that shows correlation or progression in time in a way the user can easily grasp. A comprehensive overview of these classical approaches to showing and comparing data can be found in Tufte's "The Visual Display of Quantitative Information" [34] . These visualizations are effective, but mostly concern data sets that are dependent upon only one variable.

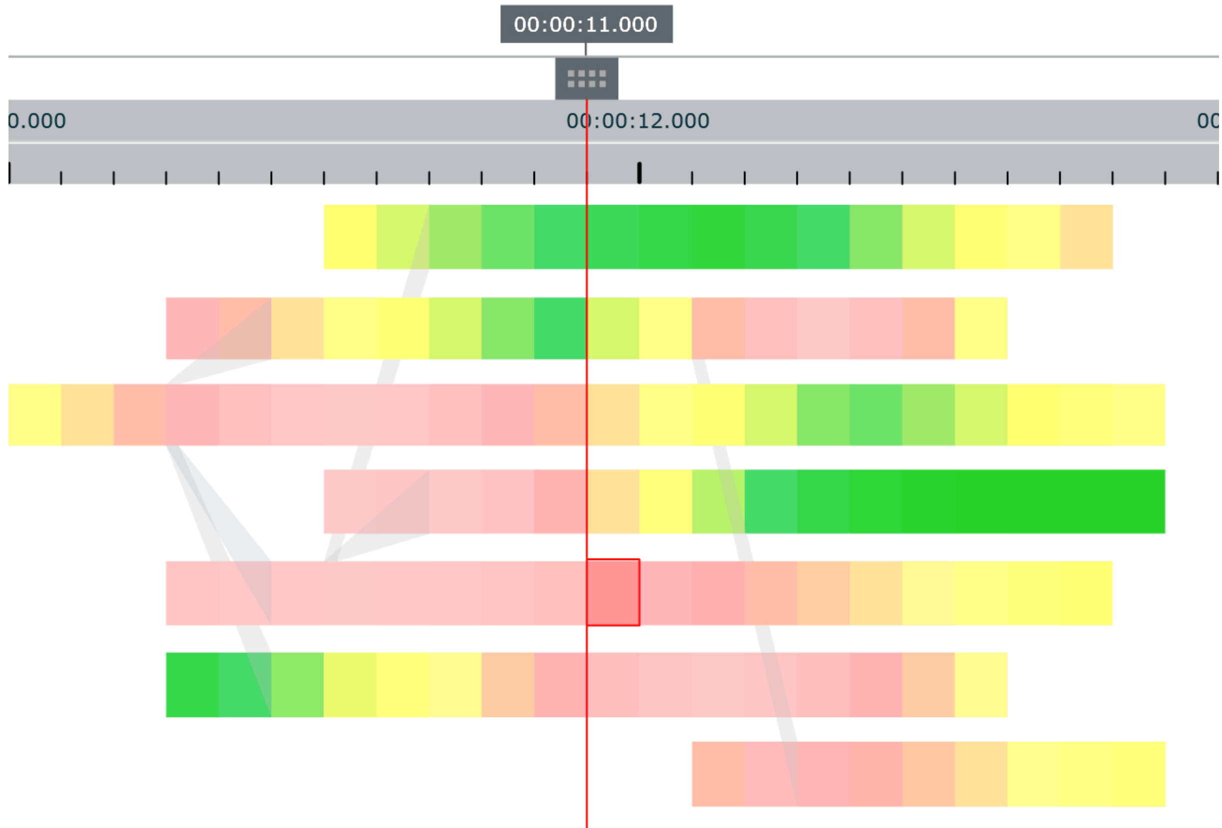


Figure 5: Tracks colored according to how much danger the buildings are in

Approaches for showing multiple dimensions in one picture exist as well, with parallel coordinates being one example that works particularly well in terms of comparing sets of data [17]. The parallel coordinates visualization consists of vertical lines, each corresponding to a dimension of the data. Each line represents a span of values the related variable can take, with the top position being the maximum and the bottom most the minimum. For each data sample, points corresponding to the values of the sample are found on the lines, and connected with a multiline. When a very large number of data samples is placed in the parallel coordinates, the user may gain additional insight into the data by observing the way the lines are grouped, allowing him to focus on desired attribute values or detect outliers of interest. One example of how the parallel coordinates visualization may look like is given in figure 6.

Parallel coordinates work especially well with linked views, an approach that uses multiple visualizations at once. The approach allows the possibility of combining visualizations such as scatterplots and graphs with more complex ones such as parallel coordinates and actual renderings of data. In such a setup, the user can select areas of interest in one visualization in a process called brushing [8]. The brushed values are highlighted in other visualizations, linking the views together and allowing the user to filter the values shown. A good example of how this technique can be applied to sets of real-world data can be found in Weaver's paper on visual analysis using cross-filtered views [37]. This approach allows the user to examine sets of data for outliers and other interesting values, and differentiate them in multiple views. The linked views can also be combined with scientific visualization approaches, as shown in solutions like SimVis [11] .

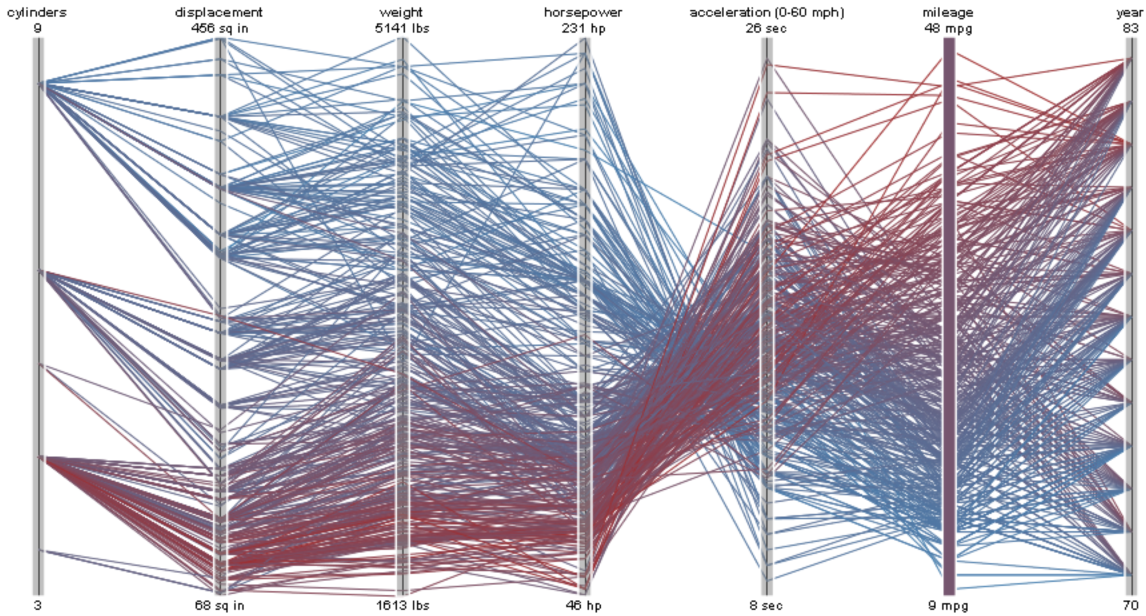


Figure 6: A parallel coordinates visualization of car fuel consumption data. Taken from the Protovis website [33]

While information visualization has many approaches that focus on comparison, scientific visualization lacks commonly accepted methods that could be used for the same purpose. Such a condition is far from unexpected though, as information visualization often deals with a large number of independent data items in a data set, making a method of comparison vital for the finding outliers and trends. In contrast, scientific visualization is primarily concerned with massive data sets which typically focus on a single phenomenon. Creating a suitable visualization for such a data set is a big enough problem by itself, and most approaches in scientific visualization are designed without even considering the idea of comparison. Despite that, several approaches to comparing data for certain scientific visualization tasks have been developed. While these approaches are not compatible with the simulation scenario or the method of exploration used, they are worth describing in more detail.

2.4.1 Visualization of multi-variate data

In scientific visualization, the data to be visualized can be described as a mapping that assigns a scalar or vector value to each point in a multi-dimensional field. When the mapping assigns more than one value to each point, the data is called multi-variate. Multi-variate data commonly occurs in scientific visualization when more sources of data are present or more than one quantity is measured. For example, in volume visualization the same object might be scanned with more than one type of machine, producing multiple values for each voxel. Since the connections between different values are important for gaining insight into the data, a good visualization of multi-variate data is expected to show more than one attribute at the same time in a clear and cohesive way. According to a report on the visualization of multi-variate data, it remains an important problem of the field, with many different approaches already existing, but each tailored to fit the

type of data set it is meant to visualize [10].

Visualizing multiple scenarios to compare them can be seen as a type of multi-variate data visualization, as it also attempts to visualize multiple spatially related values with the goal of giving insight into the connectedness of different attributes. One method that deals with both multi-variate data and comparative rendering uses contextual cues to explore data [38]. The idea behind the method is to combine all the layers of data from different datasets, attributes and time values by using operators that combine these values to create new ones. The user specifies an expression that uses the operators and the layers to be combined.

The combination is done on a per-element basis, with the end result rendered in the same way as the individual data layers. Based on the structure of the expression, a tree depicting the various intermediate results is also rendered, allowing the user to see the possibly complex process of recombination in more detail and to experiment with different operators. The flexible nature of the expressions leaves the user a lot of space to experiment, making it possible to create many different images. The flexibility also has its disadvantages, as the process requires a lot of user engagement. Comparing fields can require the specification of transfer functions for each field and the expression that links the fields, making quick comparisons impossible. While the authors consider the idea of the automatic generation of combination expressions as a part of their future work, no such follow-ups have been made at the time this thesis was written.

2.4.2 Visualization of time-varying data

Another type of data that can be considered to be related to comparative rendering is time-varying data. One of the goals in visualizing time-varying data is to show how it changes over time, which also allows the user to compare various points in time. An interesting approach to showing such changes is described in the earthquake visualization made by Hsieh et al. [16]. A two-dimensional earthquake data set with a certain number of time steps is treated as a volume to be rendered, with the third spatial dimension being substituted by time. Several isosurfaces of the amplitude of the earthquake are generated and shown together using transparency to make the isosurfaces distinguishable from one another. Color cues are also used to signify the intensity of the amplitude. In the resulting image, the user can see significant events as clearly distinguishable shapes. Because time is used as one of the dimensions, the way the shapes change shows how the earthquake unfolded. The idea of using a dimension of space to convey information about time is powerful, but an approach similar to this one could not be used with World Lines. The frames to be compared can come from different parallel worlds, and showing the way they are connected would require a tree-like structure that cannot be shown using only one dimension.

While the nature of the data the World Lines approach generates is different from that of time-varying data, some methods developed for time-varying data can still be used to provide additional information about the frames compared. Feature extraction is a technique that usually requires knowledge about the underlying phenomena, but the method described in the paper by Janicke et al. [18] attempts to create a general

approach to extracting features by using principles of information theory and automata theory. The method uses a measure known as local statistical complexity, which calculates how interesting a cell in a celluar automaton is based on its past and future states. The idea of the measure is to find out how much information about surrounding cells is needed to predict the future state of a cell. The authors of the paper use the measure to detect interesting areas of fields by treating time-varying two-dimensional fields as automata and applying a modified version of the formula to them. The approach manages to detect structures similar to ones classic visualization techniques reveal without making any assumptions about the type of the field analyzed. While the frames selected for comparison in World Lines need not be temporally related in any way, the information about how those frames were created still resides within the system. For each selected frame, the preceding and following frames may be retrieved, allowing the use of the described method to gain more information about the selected frames.

2.4.3 Visualization of ensemble data

In visualizations for the various types of data mentioned so far, the ability to make comparisons is beneficial, but not the primary goal of the visualization. Ensemble data contains heterogenous multi-variate and time-varying data sets produced with different numerical models and different parameters. Comparing these data sets and finding common features is vital in visualizing ensemble data. Potter et al. [29] approach the task by using multiple simple visualization techniques to show various statistical descriptors of data. This includes techniques that map the values to their actual spatial representation such as a heat map of the world, and simpler techniques such as graphs that show the value of a statistical parameter as it changes over time. The statistical descriptors used, such as the mean and the deviance, are simple yet effective for reducing ensemble data to multi-variate time-varying data which can be displayed using well-known methods.

This overview of various techniques for visualizing scientific data shows that the approaches attempted vary significantly according to the properties of the data and the aim of the visualization. Because of this, the thesis will focus on the ways the data is acquired through simulation first, then explain the classic rendering approaches used to show the simulated phenomena, and only after that explain what attemps at comparative rendering have been made.

2.5 CUDA

The NVIDIA CUDA parallel architecture is used in some compute-intensive parts of the implementation to boost performance [23]. CUDA allows NVIDIA graphics processing units (GPUs) to be used for general-purpose computation. GPUs typically have much greater computing power than CPUs, but that power comes from the large number of cores functioning in parallel. Programming parallel algorithms is a difficult task, and not every algorithm can be efficiently parallelized. The requirement that an efficient CUDA-using application must split the work into a large number of threads further limits the problems that can benefit from GPU computation. An additional issue is the memory

usage. Because of the limited bandwidth between the graphics card and the CPU, memory transfers between GPU and CPU memory must be carefully considered to avoid losses in performance, which requires good knowledge of how the GPU transfers and retrieves data.

While knowledge about how CUDA programs must be made to perform efficiently is important for anyone writing or analyzing CUDA programs, it is not necessary to know much about CUDA to understand this thesis. Details about the internals of CUDA have been omitted from this thesis because a quality explanation of the subtleties of CUDA would take up several pages. The CUDA reference manual serves as a good introduction for any interested reader [22].

Despite all its drawbacks, CUDA can yield great speedups for many problems. Users can take advantage of CUDA-accelerated applications if they have a modern NVIDIA graphics card, and for professional uses, specialized hardware dedicated to providing raw computing power exists. Since Visdom works as a distributed system, it is only necessary to make sure that the server is equipped with CUDA-capable hardware. Because of this, the choice of using CUDA in this thesis and in the framework is not likely to limit the number of users capable of running it.

3 Simulation handling

The first problem addressed in the thesis concerns the simulation handling. The task of introducing simulations into the framework was difficult by itself because of the way nodes exchange data in the system. A standard node creates its output on the basis of its input. One example of that is the FunctionNode, which performs an operation on two inputs and produces one. A more advanced example would include a node that uses more than one time value - e.g., a node that calculates a derivative of its inputs needs the current and previous value to approximate the derivative. Unlike the standard nodes, a simulation node must use input values to create its initial state, but after that, all the outputs are created based on this initial state. A special case happens when a branching causes one of the input values to change, e.g. by modifying the levees geometry to create a breach. In such a case, the node must decide whether the changes can be incorporated into the current state of the simulation or whether the simulation has to be restarted, causing the loss of all data. Using a simulation node alongside World Lines creates new problems. The ability to jump to any frame at will requires that the simulation system can revert its internal state to a previously used one, and recognize when it needs to generate new data. The process of branching requires the parameters of the simulation to be changed, and the internal state converted to match the new parameters.

From a design viewpoint, the problems that needed to be solved consist of:

- Finding an adequate representation of internal states within a simulation node
- Desigining mechanisms that allow for transitions between these internal states based on changes in simulation parameters
- Extracting what parts of such behavior are common to all simulation nodes

This resulted in a design consisting of two vital parts - the abstract simulation node and the simulation system. The abstract simulation node contains all the functionality common to various simulations, and controls a simulation system. The simulation system performs the actual simulation, processes the input, and creates the output. The abstract simulation node is represented by the AbstractSimulationNode class, and the ISimulationSystem class is an interface that describes the functionality of a simulation system. An UML diagram showing the classes and their relationship can be seen in Figure 7. The abstract simulation node and the simulation system deal with different aspects of the system's complexity, and will both be talked about in more detail to explain the division and how it affects the system's capabilities.

3.1 Simulation system

A simulation system is a wrapper built around a simulation engine, designed to adhere to the `ISimulationSystem` interface. The interface contains methods that allow the abstract simulation node to control the system and handle its inputs and outputs. The simulation system's constructor is expected to initialize all simulation-engine related resources

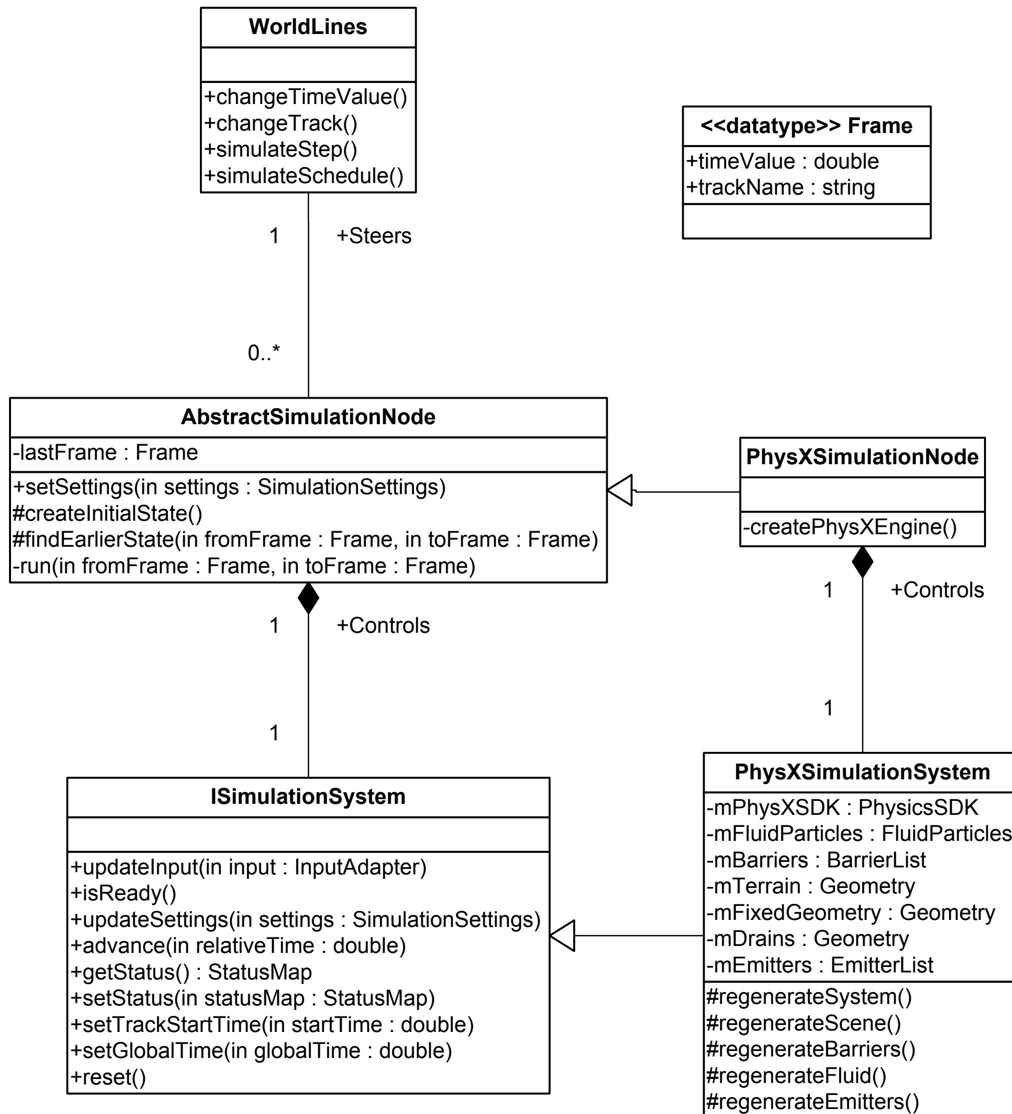


Figure 7: The UML class diagram of the simulation classes

to assure that all subsequent method calls can access the engine without problems. The first method of the interface to be called is the `reset` method, which resets the internal state of the system and prepares it for receiving data. Once the system is initialized, the `updateInput` and `updateSettings` methods can be called. These methods check for changes in the input or settings provided, comparing them to the ones the simulation system is currently using. Once the differences are known, the methods call the appropriate functions of the underlying simulation engine to apply the changes. While these methods may seem simple, practice has shown that each change may trigger an expensive call of the underlying engine's functions. That is why an efficiently designed simulation system must accumulate the effects of many individual settings or input changes. Since these changes can be split between methods as well, the accumulated changes are applied only

when a simulation needs to be performed. To achieve this, the design invokes parts of the simulation engine separately whenever a setting related to the part is changed.

Once the simulation system has been initialized and provided with the input and settings, the `run` method can be used to perform the simulation. The `run` method accepts only one parameter - the amount of time that needs to be simulated. While the method may internally perform smaller timesteps to enhance precision, these are invisible to the user of the interface. The `run` method is the elementary operation that can be invoked from the interface. Any complex simulation scenario that involves changes in the input or the settings must be broken down into multiple `run` calls with the settings or the input changed between the calls.

The `run` method can only advance the simulation, and by itself it would not allow the user to jump between frames. Because of this, the interface also contains status saving (`setStatus`) and loading (`getStatus`) methods that allow for more complex behavior. The status of a system consists of the values of all the variables in the simulation which are necessary reconstruct the state of the system, saved in a simple map which associates strings with various parts of the status. By saving and loading status information, the interface can be used to return the simulation to an earlier point in time, allowing branching to be implemented. The status also serves as the output of the system, as the variables that describe the system are often exactly the output the system is expected to produce. It should be noted that the status itself is not a complete description of the internal state of the system, as the correct settings and inputs are required to describe the state as well.

While these methods are enough to allow the simulation system to be used in conjunction with World Lines, a few other convenience methods have been added to the interface. The most important one is `setGlobalTime`, which allows the current time value to be given to the simulation system, providing information about the absolute time that has passed since the beginning of the simulation. This method was added because the run method in combination with the state loading allows for relative measurement of time only, which has proven not to be sufficient for implementing more advanced behavior such as gradual changes in the parameters. The global time value cannot be passed in the settings because a `run` method may fragment a run. In this case, the global time value needs to be updated after each part of a run so that gradual changes can be executed. The method `setTrackStartTime` is also used only for gradual changes only. It informs the ISimulationSystem about the beginning of the current track, allowing it to find out how long the change has lasted.

3.2 Abstract simulation node

The purpose of the abstract simulation node is to provide support for dealing with the complex handling issues inherent to the Visdom framework and the World Lines technique. A simple command issued in the WorldLines interface may have to be decomposed into setting changes, input changes and multiple simulation runs in order to be used with the simulation system interface. Depending on the data already present on the server, only a number of those runs might have to be executed. Because this decomposition requires detailed knowledge about the layout of the tracks, a simulation node cannot perform the

decomposition alone. Instead, World Lines reduces the command into smaller requests that describe transitions from track to track and forwards them to the simulation node, which decides whether and how these requests should be executed.

The run request that a simulation node receives is no more than a change in the settings, consisting of four important values. The first two are the time value and track which form a frame that describes data that needs to be produced after the node has completed its run. Additional information that describes the target track is also supplied: when it was started and what is its parent track. Based on these settings, the simulation node is tasked with finding a saved state of the simulation system from which the requested frame can be reached with the least amount of work. After the state is found, the simulation is run until the desired frame has been reached. Depending on the frame time step size, the run may be broken up into multiple calls in order to save the intermediate results.

Since World Lines is implemented on the client and thus cannot know what data is stored on the server, the duty of handling cases where previously created data can be reused falls to the simulation nodes. The pseudocode for this procedure can be seen in Figure 3.2. When a run request is issued, the first check a simulation node performs involves seeing if any data is present on the server. In the case that none is present, the simulation system has to create the first frame of the base track from the available input and settings. After the simulation node is certain that at least one frame is present in the system, it checks if the requested data is present on the server. If the data is present, no further calculations need to be done, and the search terminates.

In the case that the requested data is not present, the node has to generate it. The simulation node checks if any frame belonging to the requested group exists within the server, and if it does, it finds a frame that has an earlier time value. Should no such frame be found, the check is performed again, but in the parent track, seeking frames saved earlier than the current track creation time. Once a fitting frame has been found, its state is loaded into the simulation system. Since state loading is an expensive operation, the simulation node also checks whether the simulation system is in an appropriate state already. After the simulation system is properly prepared, the simulation runs up until the point specified.

The principal disadvantage of this approach is that two tracks cannot be simulated at the same time using only one run request. Each of these two tracks has different settings, and since the run request is transferred as part of the settings, the settings of only one track can be sent alongside the request. While the simulation node can transition from one track to another by using a parent track to create data in a new one, it can only do so if the appropriate start state has been prepared. To explain how World Lines must create requests in more detail, an example will be given.

In the track layout described in Figure 9 there are two tracks, track 0 and track 1. Track 1 is created by branching off track 0 at frame B, and the last known saved simulation state is present at frame A. When given frame C as the next target, the simulation node will find frame A as the frame it needs to continue the simulation from, and run the scenario as if the branch was made at frame A and not frame B. To counter this, World

```
function run(fromFrame, toFrame)

if the base frame does not exist then
   createInitialState()
end if

if frame toFrame exists then
   return
end if

previousFrame = findEarlierState(fromFrame, toFrame)
if previousFrame isn't already loaded then
   simSystem.loadStatus(previousFrame)
end if

for time = (previousFrame.time + frameTimeStep) to toFrame.time; step = frameTimeStep do
   simSystem.run(time - lastTime)
   simSystem.saveStatus(Frame(track, time))
   lastTime = time
end for
return


function findEarlierState(fromTime, toTime, parentTrack, track)

if toFrame.track exists then
   find an earlier frame on the starting track
   return  earlierFrame
end if

find an earlier frame on the parent track
return  earlierFrame
```

Figure 8: The pseudocode of the abstract simulation node

Lines must create two requests: one that states frame B as its target and has the settings of track 0, and another that states frame C as its target and has the settings of track 1. To ensure that multiple requests are not issued all the time, World Lines keeps track of which tracks have been simulated, limiting the number of requests sent. The system is designed in such a way that the client will never issue a request that the server cannot process.

The main disadvantage of this system is that the logic needed to process a user's request is split into two locations. The separation requires much information to be transmitted, resulting in many requests which can be expensive due to the distributed nature of the system. In the future, our intent is to refactor the system and make all the knowledge about the layout of the tracks available to the server. This will cut down on processing time and eliminate similar yet separate code that exists on the client and server.

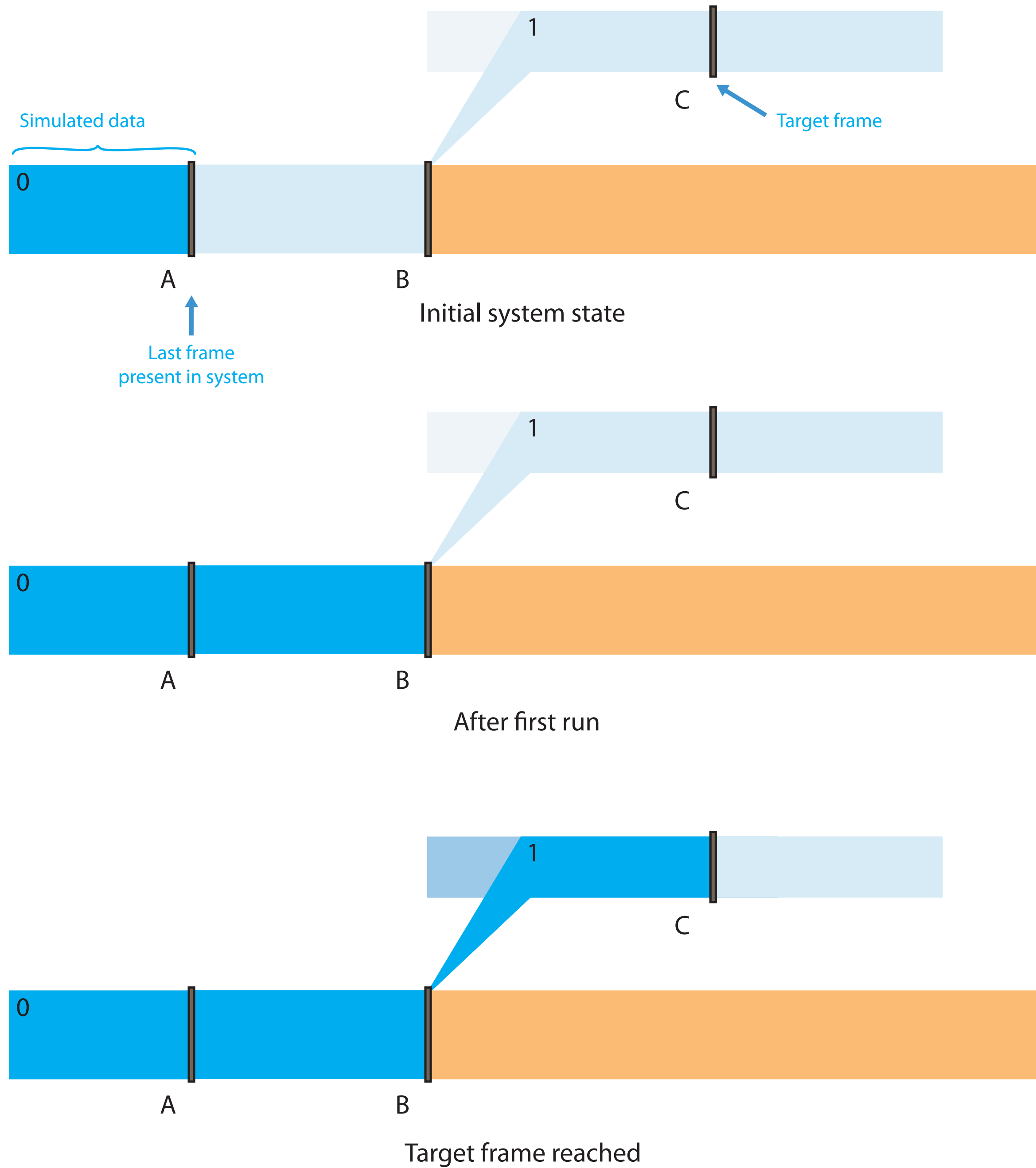


Figure 9: An example of a simulation run that has to be broken up into smaller runs

Another function the simulation node performs is intermediate frame recording. While the states of the frames that the user specifies must be recorded, it is practical to save the intermediate states as well because of the jumping and branching. Since World Lines allows the selection of only the frames whose time value is a multiple of the frame time step size, the intermediate states are saved at only those time values. To prevent the overhead resulting from too many states being saved, each simulation node has an additional saving frequency property, which determines how often the intermediate values are saved. While the status could be saved at every time step, the operation is both time and memory intensive, and the user can make a tradeoff between faster exploration and faster

simulation. The frequency can be changed at any time without any processing costs as it is not used during state retrieval, but only while saving states.

3.3 PhysX simulation system

To show how a simulation system has to be designed to conform to the common interface, this section will contain more information about the way the PhysX simulation system works and about its internal architecture. While most of the logic in the system is related to adapting data to and from the PhysX format, there are two interesting parts of the system that should be described in more detail. The first is the internal hierarchy, which allows the system to function efficiently in spite of settings and input changes. The hierarchy defines the effects of each change that can be done to the system in terms of how it affects the individual parts that compose it. The second are the gradual changes which occur when branching introduces a change that needs to be aplied incrementally.

In order to better understand why efficient handling of input and settings changes is important, it is necessary to understand the restictions placed on the user by the PhysX SDK. The SDK provides support for many different types of physical phenomena by means of classes that represent them. Objects of these classes are initialized with data, which serves as the initial state, and settings specific to the phenomenon they are modeling. For some objects, like those that represent particles, the settings cannot be changed after the initialization, prompting the recreation of the object in question with each modification. Since the PhysX engine uses the GPU as a source of computing power, the creation of an object may include transferring the initial data to the GPU, making the operation expensive. As many such examples of hidden expensive operations exist within the SDK, they cannot be ignored, especially when handling simulations that take a significant time to complete or handle great amounts of data.

One simple way of dealing with the problem would be to apply all settings at the same time, and only then recreate all the objects. However, due to the design of the framework, the passing of settings and input to a node do not happen at the same time, causing the existence of two such phases. Another phase would be needed for the act of loading a status, potentially tripling the amount of work that needs to be done. The objects also contain interdependencies that require additional recreations - e.g. the only way to change the geometry marked as fixed to support fast collision detection is to recreate the scene, destroying all other created objects and requiring their recreation.

To resolve these problems, the system has been modified to delay applying the settings and input changes up until the point the simulation starts. All the changes are saved within in a non-PhysX adjusted format, and the system is initialized on the basis of this copy of all inputs and settings. Every change also sets one or more flags, marking which part of the system the change influences. To account for the influence that some parts have on others, some flags activate others after being changed. The parts can be ordered into a tree-based hierarchy in which the setting of a part's flag means that the flags of all descendant parts get set as well. Figure 10 shows this hierarchy and how various changes in input and settings affect parts of the system. Based on the flags set, various parts of the system are recreated prior to using the simulation with the stored settings and

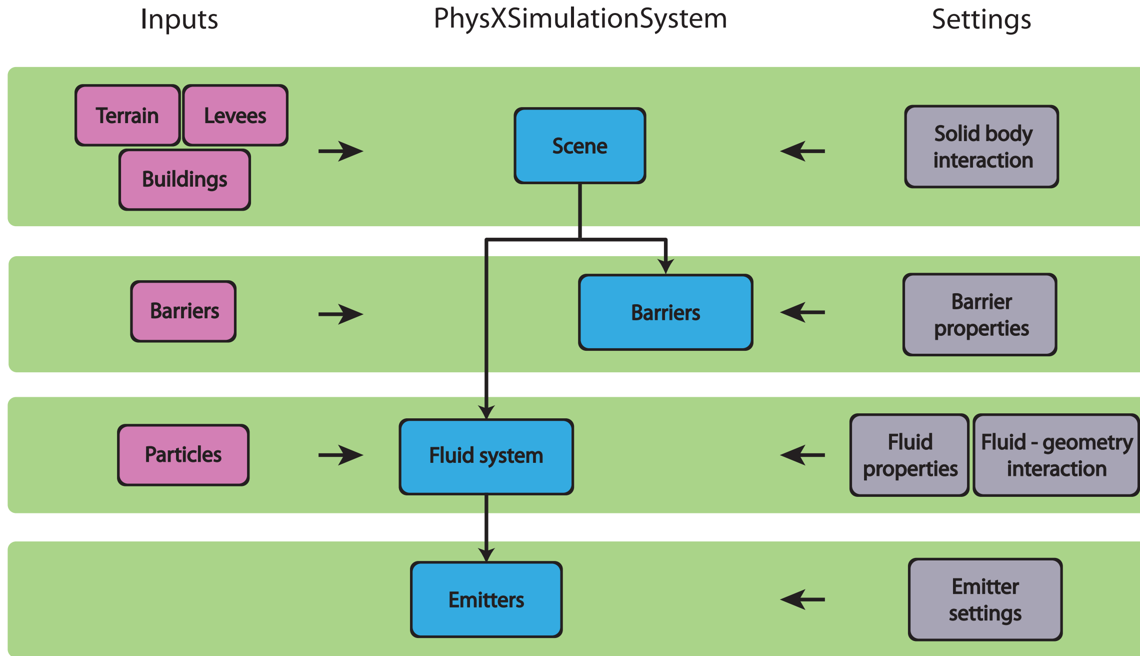


Figure 10: An illustration of how various parts of the system are connected. Arrows indicate how changes propagate through the system.

inputs. To keep these stored values accurate, the design requires that when the status of the system is saved, it is created from the stored data. This ensures that the copy is kept accurate after a simulation run has ended.

The advantages of such an approach are twofold. As far as performance is concerned, while storing the settings and input separately carries some small overhead, there is no actual copying done. The overhead of the storage is insignificant compared to the cost of recreating an object. The other benefit, perhaps more significant, is the clarity of the design. Adding new settings or objects is relatively easy, and more importantly, not likely to break any other preexisting part of the system. Our experience has shown that many modifications were needed to actually use the node in the simulation scenario. We assumed that should the system be used for a new purpose, new modifications will have to be introduced as well. The presented design increases the reliability of the node and increases the chances that it will be usable for its original purpose even after being extended.

The bag dropping serves as an example of a behavior that was added later by modifying the PhysX simulation system. While it may have been possible to implement such a behavior as a separate node that changes its output in time, making such a node compatible with World Lines would require implementing functionality similar to that of the simulation nodes. The solution chosen consisted of a node which allowed the user to define the bags to be dropped within a track, while the simulation system would release them one at a time. This setup mimicked the breach closure study, where the bags used to seal the breaches were dropped by a helicopter one at a time, with a certain interval between bag drops [32].

Introducing such a behavior into the system requires the use of absolute time values. Unlike other behaviors modeled, the bag drops last for only a limited duration of time, and can be considered a gradual change in the state of the system. Normal changes of parameters are applied only once and require no additional actions, as the settings mechanism ensures that the effects of those changes will be present at the correct time. Unlike a normal parameter change, a gradual change must be integrated as part of the system state. The information stored for the bag dropping mechanism contains the start of the dropping process and the number of bags to be dropped. By using the absolute time value available to the simulation system along with the track start time, it is possible to determine when and how many bags need to be dropped yet. The transformation matrices of the barriers are always present in the system status, but they only get updated if the corresponding bags have been created in the PhysX engine. This allows the barriers to be seen before they are dropped, so that the user can know when the bag dropping is complete.

4 Scene rendering

Since this thesis concerns comparative rendering, a fair part of the work involved was related to the rendering subsystem of Visdom. Because of that, this section contains a description of the subsystem and the specific rendering modes used during the implementation of the scenario.

4.1 Rendering subsystem

The rendering subsystem of the Visdom framework consists of a single node - the OpenGL node. The functionality of the node is not spread out into multiple nodes because of the complexity of rendering. Each node in the framework can be instantiated on its own, and the rendering requires resources that would have to be shared among multiple nodes performing the rendering. Separating the OpenGL node into multiple smaller nodes would require tracking resources on a system-wide level, which would make it very difficult for the nodes to be intuitively used and modified, and would likely leave them prone to errors.

However, the single node design comes with some difficulties as well. The requirements placed on the node are not simple: the node must be able to:

- Process various types of input, some of which may not be present
- Control the behavior of rendering related to those inputs
- Be easily extendable

This means that all the complexity that would have been present in the multiple node design is mirrored in the underlying architecture of the node. Because of this, the basic concept of the OpenGL node is that of the `Renderable`. A `Renderable` is an object that can perform rendering on a buffer when provided with the appropriate input, settings and common resources. All rendering is done within the renderables, with all the other parts of the OpenGL node concerned only with the management of renderables. To explain the structure of the OpenGL node, various parts of the system will be explained in more detail. An accompanying UML diagram is provided in Figure 11.

The basic class which controls and invokes the renderables is the `OpenGLRenderer`. The renderer receives the settings and the input values, and based on the changes in the two, makes a decision on what renderables to use or remove. Almost all renderables are tied to inputs, and a renderable object usually lives as long as the input it visualizes is connected. The only renderables not tied to inputs are the gadgets used to help the user to orient himself by highlighting the origin or depicting the direction of the camera, which are controlled by the node's settings. For each renderable, the renderer loads a specification of the inputs it is to pass on to the renderable. The specification consists of required and optional inputs. A renderable is created only when all of the required inputs are present and it is enabled in the node settings.

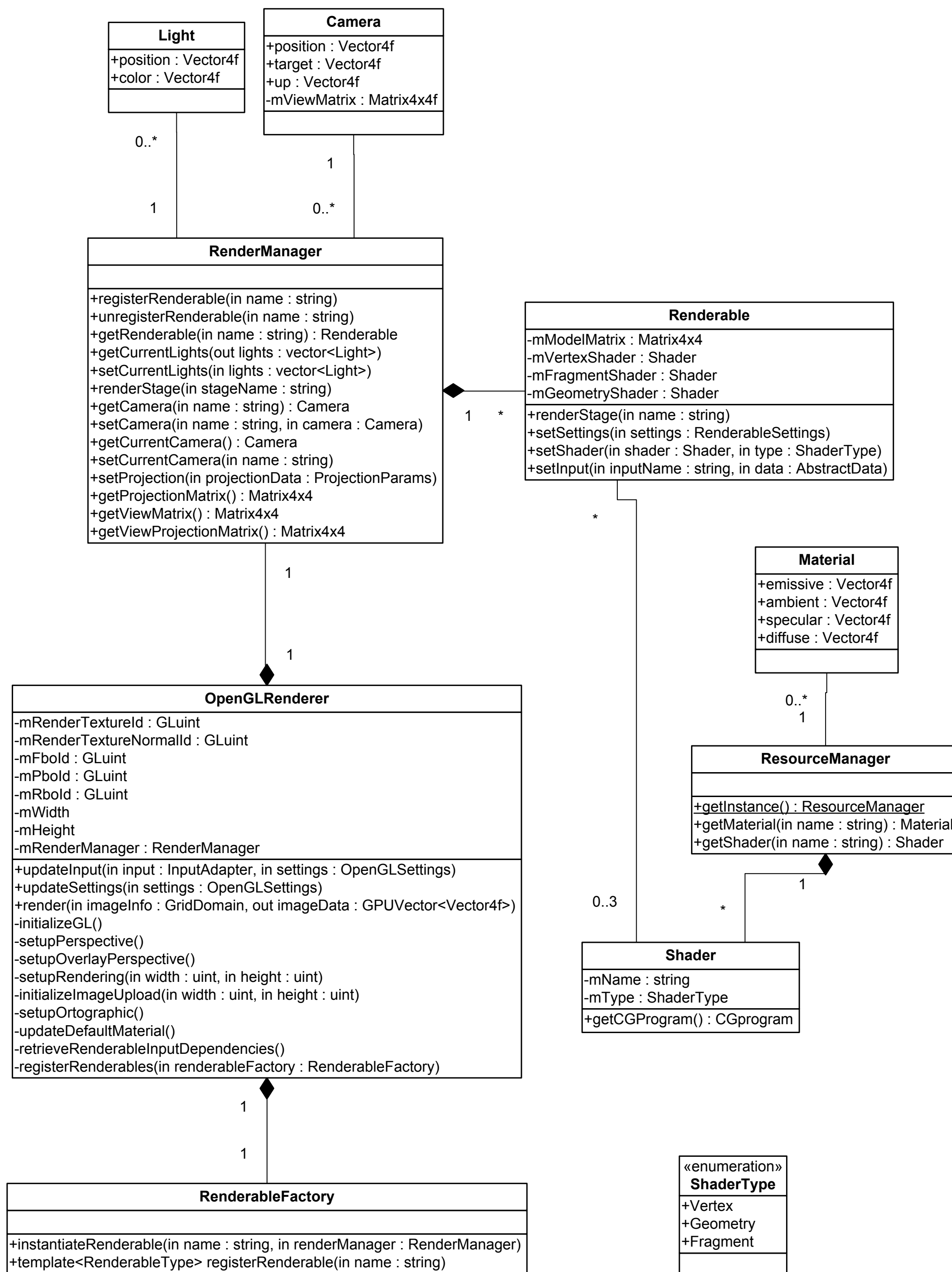


Figure 11: A class UML diagram of various classes used for rendering in the framework.

While the renderer does have control over the renderables, it does not store the information about which renderables are actually present in the system. Whenever the renderer requires a renderable object of a certain type, it requests it by name from the `RenderManager`. The `RenderManager` creates the object if necessary, and when the ren-

derer decides a renderable object is no longer needed, it may destroy it as well. Because the renderables are fetched by name only, no more than one renderable object of a certain type can be present in the system at the same time. While such a limitation may seem too restrictive, it is useful because it forces more complex methods of tracking input changes to remain within the renderables, making the already complex part of renderable initialization cleaner.

After handling the changes in the inputs and the settings, the renderer must perform rendering to make these changes visible. Since the rendering is not directly output to the screen but sent to the client instead, the results must be stored in an intermediate container. OpenGL provides such functionality in the form of framebuffer objects (FBO), which allow the rendering to be done to various separate buffers. One such buffer is used to accumulate the effects of renderables and retrieve the results as a byte stream.

The process of rendering is divided into phases that regulate the order in which the renderables are executed. Each renderable can register itself for participation in a phase, and it can participate in multiple phases in case multiple non-consecutive rendering passes are needed. Registration for a phase means that a renderable will be activated sometime during the phase and notified which phase it is participating in, allowing it to perform different actions according to the phase it is invoked from. The idea behind the phase system is to allow for a simple specification of execution order by supplying a number of self-explanatory phases and leaving room for user-made ones. Some common phases are the background phase, the rendering phase, the postprocessing phase, the overlay phase, etc. It should be noted that within a phase, there are no guarantees on the order of execution.

When invoked, each renderable may need access to common resources and settings, and there are two manager objects that supply them. The first one is the aforementioned `RenderManager`. Besides holding information about the renderables, it also provides information about the current rendering settings. While there are many renderable-specific settings, some settings like the camera and lighting data are widely used and should not be passed to each renderable individually. When needed, the renderables query the `RenderManager` to get the current values of these settings, assuring consistency. The `RenderManager` also provides a set of convenience functions that return values that could be extracted using OpenGL functions, such as current matrix values, FBO ids, etc. The `ResourceManager` is the second manager object, and its function is to provide storage for calculated results and manage resources. Its primary function is to store CG shader programs and load new ones when they are requested. Another of its intended purposes is the storage of results from other renderables. Since renderables that perform comparative rendering can use the data generated by other renderables, the idea of the `Resource Manager` is to allow intermediate results to be stored for future use.

While the OpenGLNode presents a powerful set of tools that allow for smooth component design, the main parts of the rendering system are the renderables themselves. Table 1 shows the renderables currently present in the framework, along with a short

Renderable name	**Renderable function**
Clone renderer	Renders multiple copies of the same triangle mesh
Coord gizmo	Shows the orientation of the camera
Fluid texture renderer	Renders the particle texture
Geometry renderer	Renders a triangle mesh
Origin sphere	Shows the origin of the scene
Particle point sprite renderer	Renders particles as a smooth fluid
Point renderer	Renders particles as points
Screen-space ambient occlusion renderer	Adds realistic ambient lighting

Table 1: The renderables currently present inside the Visdom framework

explanation of what each one does. The renderables used to show the simulation scenario will be explained in more detail.

4.2 Geometry rendering

The basic renderable is the geometry renderer, which is used to render triangle meshes in a quick and efficient way. The renderer uses Vertex Buffer Objects (VBO) to speed up the rendering process and custom shaders to provide lighting effects.

The triangle renderer is useful for huge non-changing meshes such as the terrain, but for objects that change often, it is inefficient. The large number of bags being dropped meant that another type of geometry renderer had to be introduced in order to efficiently render many copies of the same object. The other geometry renderer, called the clone renderer, accepts as input a triangle mesh and a vector of matrices which describe how the object's coordinates are transformed to the world. The triangle mesh is preprocessed and stored as a VBO and rendered once for each matrix supplied. Since it's usually not the triangle mesh that changes, the renderer can perform very efficiently.

4.3 Particle rendering

One of the first problems that had to be addressed during this thesis was the rendering of the particles produced by the SPH simulation. While there have been approaches relying on isosurface extraction to create geometry meshes that represent the fluid [31], their performance drops with the number of particles, reaching non-interactive levels at as little as 3000 particles, far beneath the number of particles needed for a flood in the simulation scenario. Other approaches capable of rendering a greater number of particles

try to render the fluid by finding sets of surface particles [5] or by creating and deforming mesh representations of the fluid [15]. As the particle rendering is only the basis upon which other comparative methods of rendering were to be built, the particle rendering is based on a fast splatting-based method. Splatting is a technique in volume rendering that generates a graphical primitive for every volume element and constructs the final image by rendering them together [39].

The screen space fluid rendering with curvature flow method [35] treats the particles as volume elements for splatting. The process happens in three stages. During the first stage, the particles are rendered as spheres using a shader, and their effects are accumulated into buffers which describe the surface of the fluid visible on screen. Because the rendered spheres do not look as smooth as a fluid surface, in the second phase the fluid surface is iteratively processed to create a smooth appearance. The third and final stage renders the buffers on screen, taking into account lighting and fluid transparency. Each of these stages will be explained in more detail. Figure 12 shows the basic idea of how the particles are rendered, while Figure 13 shows the images created from the intermediate buffers.

The rendering of the fluid begins with the extraction of depth and thickness data. For each pixel on screen, we imagine a ray emanating from the camera origin and passing through the pixel. The two values that need to be found for the later stages are the depth of the nearest particle encountered on the ray and the thickness of the particles intersecting the ray. To find these values, two buffers the size of the viewport are created and gradually filled up with information as the particles are drawn. The particles themselves are rendered as point sprites - camera-oriented quad shapes with equal sides. The advantage of point sprites is that only one vertex is used to specify the quad's location and orientation, as opposed to the four sprites that would usually be necessary, drastically reducing the number of vertices needed to render the particles. To actually render a sphere as opposed to a quad, a fragment shader is deployed to discard pixels that are outside of the circle in the middle of the quad. The shader also manually outputs the depth, increasing it depending on the pixel's position from the vertex, thus creating a sphere. If the depth calculated for a certain position on screen is smaller than the one already present, the new depth is stored instead. The thickness is calculated in a similar way and added to the thicknesses already accumulated.

The end result of the first phase of rendering is a new description of the fluid, where all the information from the particles has been reduced to two fields - depth and thickness. Because of the simplicity of this form, in certain cases information about the actual appearance of the fluid can be lost. If particles are overlapping, the measured thickness of the fluid will be greater than the actual one because there is no way to tell how the particles are placed from a single value, making a correction impossible. A similar problem can be found when there are gaps in a fluid along the ray, as there is no way to store or even detect the gaps. The disadvantages of the method also prevent the usage of transparent objects in the same scene as the fluid, as there is no way of telling how the fluid is positioned in relation to the transparent object. While it could be possible to modify the algorithm to extract one layer of fluid after another and use them in a depth peeling technique [**?**], the process would be highly expensive, both in terms of the time needed to render the result and the modificiations required in various shaders. There would also be no guarantee that the fluid surface produced would not vary greatly in

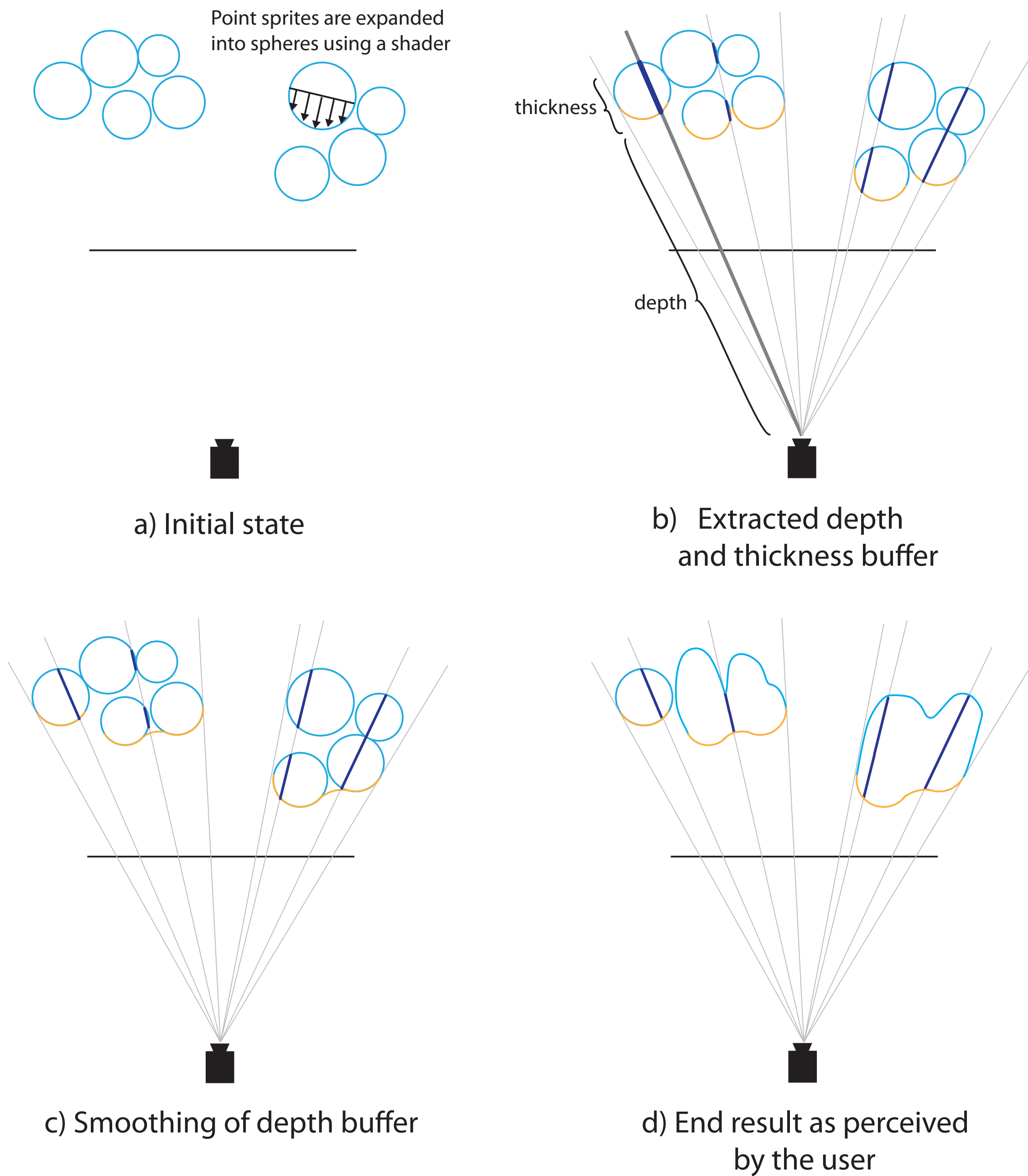
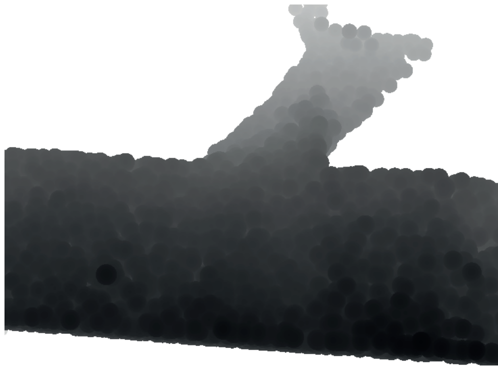


Figure 12: Various stages of the particle rendering algorithm

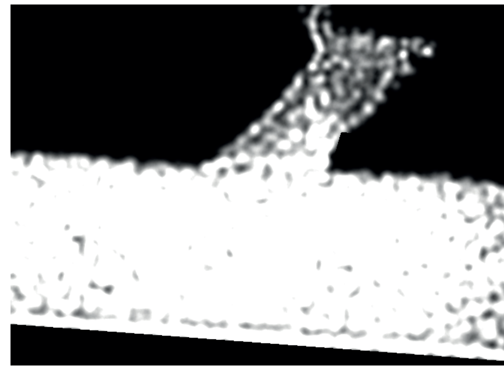
thickness depending on the number of layers peeled.

While the fluid surface from the first step could be rendered without any modifications, it does not look like an actual fluid surface due to the fact that it is just stitched together from particle spheres. In the second phase, iterative smoothing is applied to make the surface seem more realistic. The idea of the smoothing process is to minimize the curvature of the surface, where the curvature can be defined as the divergence of the unit normal. In other words, the changes between the surface normals need to be made more gradual, which will result in a smooth appearance. For each pixel, the curvature can be computed

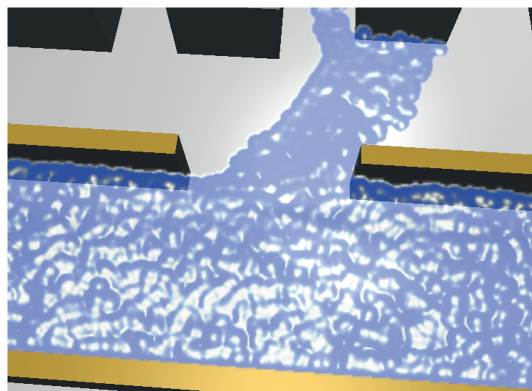
a) Scene without particle rendering



b) Depth buffer



c) Thickness buffer



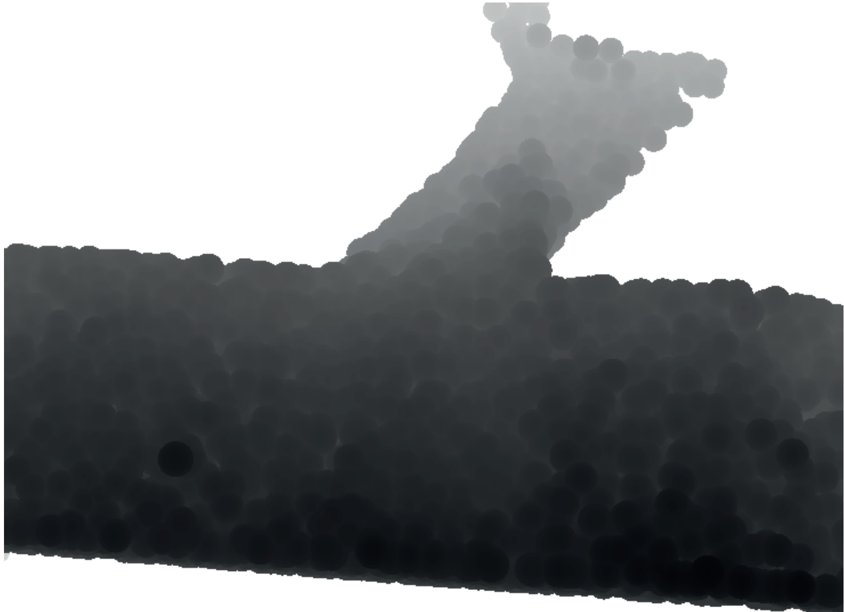
d) End result as perceived by the user

Figure 13: Images showing stages of the particle rendering algorithm

using spatial derivatives and double derivatives of depth. Since the curvature is the divergence of the unit normal, each iteration can be seen as a step of an Euler integrator that works towards minimizing curvature by changing the depth of pixels according to the calculated curvature values.

To lessen the number of iterations that need to be made, the depth smoothing is actually

done on a buffer half the size of the screen produced by minifying the original depth buffer. The changes in curvature propagate with greater speed, improving the efficiency of each iteration, but the buffer cannot be used directly, and must be blurred before being used in the rendering to avoid the appearance of artifacts. Figure 14 shows the difference in appearance between the initial depth buffer and the smoothed depth buffer.



a) Without smoothing, full resolution

b) After smoothing, halved resolution

Figure 14: The depth buffer before and after the depth smoothing.

The third and final phase of rendering involves combining the information from the buffers with the already rendered scene to create the final rendering. The opacity of the fluid grows exponentially with the thickness, while the color is influenced by several factors. The basic color of the fluid is determined by the thickness, ambient lighting, and diffuse lighting, with the color stemming from the reflected light being placed upon it. For the lighting calculations to function, each pixel's coordinates and estimated normal must be converted from the projected space to the world space in order to calculate the required vectors. To enhance the realism of the scene, a specific Perlin noise texture is added to each particle, creating a more realistic look and allowing for tracking of the fluid's movement over time steps. As the noise is unique to each particle, the movement of the surface patterns created by the noise offers a visual hint about the direction of the fluid's movement.

5 Aggregate renderings

Once the rendering of a single simulation scenario was completed, the next goal was the creation of comparative renderings that would be immediately useful in experimenting with the flooding scenario. To that end, two visualizations that enriched the rendering with information taken from the particle fields were created. These renderings are called aggregate renderings because they combine a large amount of data into one simple representation which gives he user some idea of what is common to all states being observed. The flood exposure aggregate rendering serves as a measure of how successful the solution was, and the particle attribute visualization helps with discovering explanations for the fluid's behavior that cannot be seen with just a rendering of the surface.

5.1 Aggregate rendering pipeline

Both of the visualizations designed for the simulation scenario have some common characteristics, and before describing their specifics, a model of how data is processed in these visualizations will be shown here. A diagram that describes a generic aggregate rendering can be seen in Figure 15. The goal of an aggregate rendering is to extract and combine data from multiple simulation states, and create a rendering that shows the combined data within the context of one. This happens within three distinct phases, each one of which processes the data in a different way.

The first phase is the data extraction. As the simulation states used as input can consist of many different types of objects, combining them into one single simulation state that can be rendered is very difficult or even impossible. The goal of the data extraction phase is to create output that describes a feature or a property of the simulation state, and that has a representation that is the same for all types of simulation states that can be encountered. Defining what property is to be described is very specific to the simulation at hand, but logical choices are sets of objects that are guaranteed to be present in all the simulation states, or properties that can be mapped to a specific area within the simulation. E.g., for the flood exposure aggregation, the building colors serve as the intermediate form. As the buildings are present in all worlds, the vector of building colors has the same dimensions in all worlds, and combining such vectors is a simple task.

The requirement that the representation of the extracted data is the same for all simulation states is understandable only within the context of the second phase, the data aggregation phase. While the design of the data extraction phase is closely related to the simulation and must take into account all the characteristics of the simulation to create meaningful output, the aggregation phase simply combines the data in a universal way. All the data that is given is combined using operators which produce only one value of the same type. When the representation of the data is the same for all data produced in the first phase, the operators are simply applied to all values, and another output, called the aggregate data, is produced. The aggregate data has the same format as the original data. When only one input is supplied to the data aggregation phase, the output is expected to be equal to the input value.

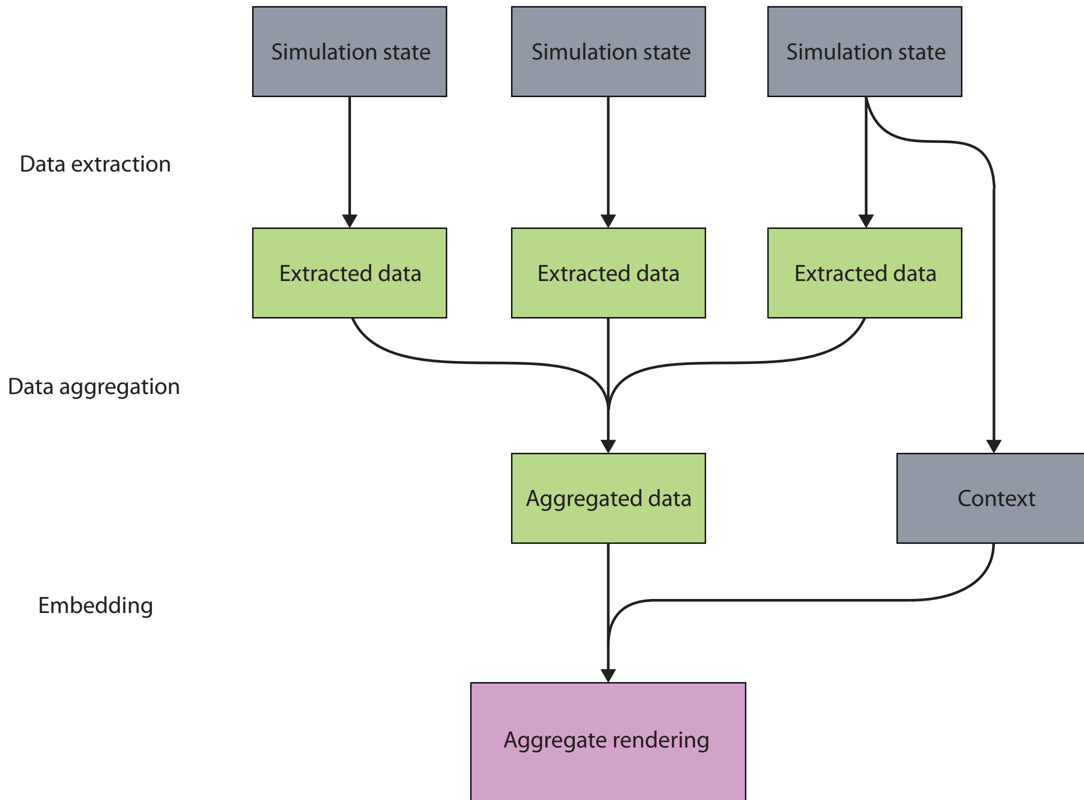


Figure 15: The various phases of a visualization that produces an aggregate rendering. Blue rectangles represent simulation states, green ones represent extracted features and data, and the pink rectangle is the final rendering.

The idea of using operators that combine data is similar to Woodring and Shen's operator-based comparative rendering method [38] that was described in Section 2.4.1, but the operators that can be used in this context are much more restricted. The main idea of Woodring and Shen's work is that the user crafts expressions that determine how various layers will be combined. In the context of World Lines, the problem with this idea is that it does not adapt well to the process of exploration. The number of data fields that could be used in the expressions changes dynamically because of the branching, requiring the expressions to be adapted often, stifling the process of exploration. If there was a way to reuse expressions and specify their inputs easily, possibly in a graphical way, it would be possible to use the approach with World Lines. However, the current way the tracks are visualized would only become more cluttered by introducing such an approach, possibly diminishing the usefulness of the system. Instead, we use only simple operators, such as the minimum, maximum and average operators, which produce the same output regardless of the ordering of the inputs, and are still useful despite their simplicity.

The final phase of aggregate rendering is embedding, which consists of mapping the aggregated data to primitives and rendering the scene. Since the aggregated data represents only a certain aspect of the simulation state, rendering the data by itself does not provide the user with much information. Because of this, the rendering is done as a part of the

rendering of one of the original simulation states, called the context. Using the context provides information required to interpret the aggregated data correctly and allows the user to compare the state of the world in the context with the aggregated state of all the simulations. By changing contexts, the user can perform multiple comparisons while using the same set of aggregated data, and visually detect interesting differences.

With the various phases of the aggregate rendering explained, the visualizations which are constructed according to these principles will be described in the following sections. This will provide a basis for the discussion of how the user can interact and learn from the visualizations based on aggregate rendering.

5.2 Flood exposure

In order to properly explain what flood exposure means, it is necessary to understand how the success of user actions in the simulation scenario is measured. In the original scenario, bags were deployed between houses to create makeshift barriers. These multibarrier systems proved to be more stable than the first attempt at closing the breach without buildings. Since the buildings used in barriers were left flooded and unusable, the goal of our experimentation was to try and find a solution which minimizes the number of buildings used as parts of the barrier or abandoned in the flooded area. We use the term flood exposure to describe the overall state of the buildings with respect to flooding. Alternative metrics such as the time needed to close the breach or the quantity of materials used were also considered. However, flood exposure was chosen as the important one because it gives the user a clear goal to aim for. The flood exposure is also tied to the other metrics, as more buildings lost causes the length of a barrier to increase, requiring more materials that take longer to deploy.

After defining the metric to be used, two problems become apparent. The first one involves finding a way to measure which buildings have been flooded. Once such a measure has been chosen and applied, it is also necessary to find a way in which to meaningfully show information about the buildings being flooded that stems from multiple system states.

5.2.1 Measuring exposure

Estimating the risk of flood exposure is an easy task for a human observer that is presented with the rendering of the buildings and the particles. When allowed to interact with a three-dimensional rendering of the simulated scene, the user can examine the buildings at will and see which buildings are surrounded by high water and thus flooded. However, this process is time-consuming and hard to extend to the analysis of multiple simulation scenarios, as it would require a single rendering that combines the states of all the simulation runs. To provide an alternative to manual examination, we define a measure of exposure that detects which buildings are flooded.

The basic assumption that was made while designing the measure is that flooding

happens when the height of the water reaches a certain level on either side of a building, causing the inside of the building to fill up with water. Should the water not reach a high enough level to flood the building, we still consider the building lost if it is surrounded by water. Thus, two height thresholds are used - one to determine if a side of the building is exposed to water, and the other to determine if a side is letting water into the building. Buildings exposed from all four sides and buildings flooded from at least one side are considered to be flooded, while buildings exposed from at least one side are considered to be in danger. With the basic state that signifies that the building was not exposed to water, the output of the measure is reduced to three possible warning levels per building: safe, in danger, and flooded

Since this measure maps a very large number of states a building may be in to just three categories, it could be argued that more warning should be added to allow a user to distinguish the state of the buliding more clearly by using just the measure. While this is a good idea for a single building, the simulation scenario contains between 15 and 20 buildings, each of which has to be assigned a value. The number of possible states of the buildings in the whole scenario increases exponentially with the number of values the measure can produce, and the adding of just one additional value can make the overview too complex for it to be useful to the user. The choice to make the number of different values equal three was a balancing act between providing enough information so that the user can see what is happening with a single building and not being overwhelmed when looking at the whole scenario.

The main problem with the implementation of the measure was the need to find the height of the water touching a side. If the simulation were performed using a classical twodimensional-grid based method, the extraction of the height would be simple and amount only to finding the height of the cells neighbouring the side. However, since a three-dimensional non-grid based method is used, the extraction relies on finding all particles that could be considered as touching the side, and finding out the height of those. While the rendering produces a depth texture that could be used to find some of the heights required, it cannot provide all the values needed. The buildings and barriers may obscure parts of the fluid, making the information about the occluded sides unavailable.

Finding out which particles touch a side of a building is not particularly complex, as it involves nothing more than finding all the particles positioned in a rectangular area in front of the side. How far the area extends from the side depends on the properties of the SPH simulation, as particles touching the side should be positioned about one kernel smoothing length away from the side. In our simulation scenario we ignored the expected distance in favor of a guessed value. This was done primarily because of the inacurracies introduced by the simulation. They were caused by the small number of particles in the scenario and the dynamics of particle-geometry interaction such as collisions. The inaccuracies would require us to account for the changes introduced by using an additional tolerance value, which would have to be guessed anyway.

Once the subset of particles that is of interest has been found, they are examined to find the maximum value. During the examination, special care must be taken to consider stray particles. Figure 16 shows a possible scenario in which some of the particles splash and fly upward, reaching a level higher than the one the user would percieve as the correct level from the rendering. Such behavior is problematic because it can cause the

maximum height to change rapidly when splashing occurs. This is undesirable given that it may distract the user, or worse yet, make him consider the introduced error as genuine behavior of the simulation and lead him to false conclusions. To remedy the problem, the average height of all the particles is recorded, and used to determine whether the found maximum value is acceptable. If the maximum height found is close enough to double the average height, it is considered to be acceptable, and otherwise, it is replaced with the double average height. This correction is based on the assumption that the particles will be distributed uniformly in the space that they occupy, which is a reasonable assumption given the repulsive forces present in the SPH model and the similar size of all the particles in the simulation. While there is still a possibility that the found height will be incorrect in cases of an uneven distribution of particles, the correction works well and does not require a significant increase in computational power, as the average height can be calculated at the same time as the maximum height.

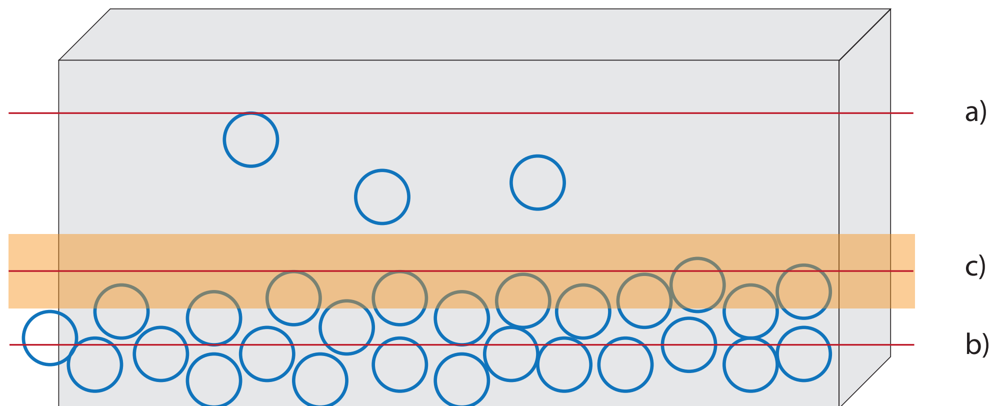


Figure 16: Various heights considered while determining the height of water touching a side of a building. a) Maximum height found. b) Average height. c) The acceptable range of heights, based on the average value.

Finding out which particles belong to which building is computationally intensive, as each one of up to 25 000 particles present in the scenario must be examined to see if it touches one of the sides of the 15 to 20 buildings. When rendering the results of the simulation in real time, the operation needs to be performed for each frame being displayed. If an interactive frame rate is assumed, the operation is called no less than 20 times per second alongside other tasks which also use up computational power. To allow for the measure to be used in real time, we implemented the search on the GPU using CUDA to take advantage of the GPU's processing power. This meant that the calculations needed to be parallelized, with the number of parallel tasks as high as possible. For each particle, one CUDA thread that checked it against all sides is created, and the heights of the particles that touch a side are gathered together using a fast parallel reduction procedure. The pseudocode of the procedure can be seen in Figure 5.2.1. Implementing the checks in CUDA caused the average time needed to perform the analysis fell from 120 ms to 50 ms, making it possible to use the measure in real time.

```
function findBuildingSideHeights(buildingData, particles)

set height array to zero with size noParticles * noBuildings * noSides
set particleExists array to zero with size noParticles * noBuildings * noSides

for each particle in particles in parallel do
   buildingData = getBuildingData()
   for each(building, side) in buildingData do
      if particle belongs to side then
         height[building][side][particle] = particle.height
         particleExists[building][side][particle] = 1
      end if
   end for
end for

for each (building, side) in buildingData do
   maxHeight[building][side] = maxReduce(height[building][side])
   averageHeight[building][side] =
   sumReduce(height[building][side]) / sumReduce(particleExists[building][side])
end for
```

Figure 17: Pseudocode of the CUDA height extraction

5.2.2 Results aggregation and visualization

The building flooding data extracted from the simulation state consists of a vector of values, one per building, which show if the building is in danger or flooded. We use this data to create an aggregate rendering. Our motivation came from the often encountered situation in which several ways of closing the barriers are tried, resulting in multiple simulation states. The user wishes to see which buildings were successfully protected in all of these states, and which buildings were exposed to danger or flooding in at least one of the states. If the results are ordered according to severity, with the safe state being least severe and the flooded state being most severe, the maximum operator can be used to combine the results from various simulation states.

Once the combined results were obtained, the embedding of the calculated results was done by substituting the color of the buildings with new ones that signify the warning level of the buildings. The colors used were chosen to be the same as the ones found on a semaphore, with safe buildings colored in green, endangered in yellow, and the flooded ones in red. Since these colors are usually associated with various levels of danger, the meaning behind them is apparent even to the user who knows nothing about the underlying mechanisms (see Fig. 18).

The purpose of this view is to allow the user a quick overview when observing just one simulation state and to give him tools that allow him to compare a simulation state with

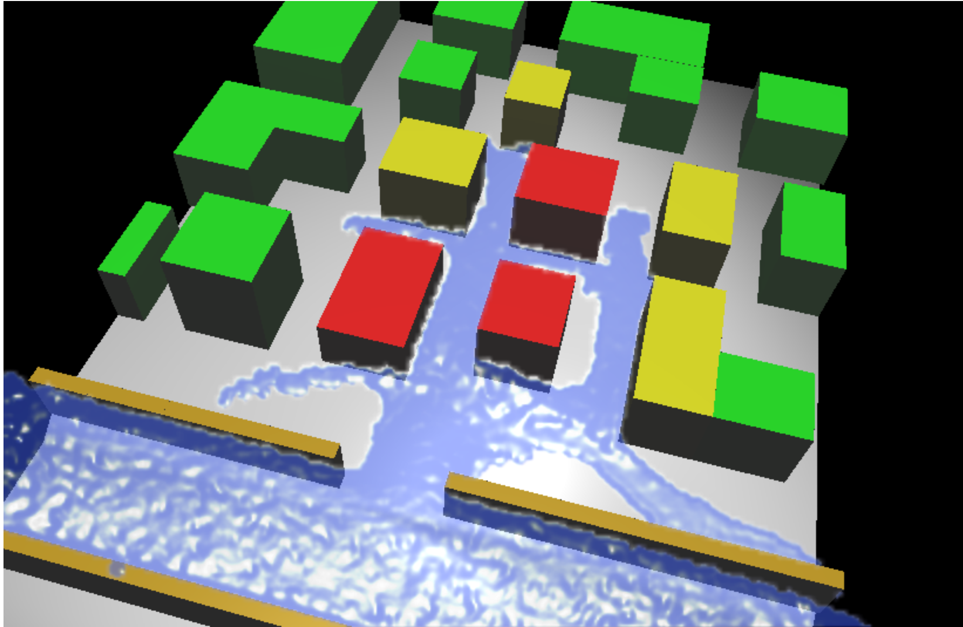


Figure 18: The flood exposure visualization showing which buildings are affected by the spreading water.

others. While the building colors allow a quick assesment of their state, the comparison relies on the user examining the flooding both by using the water levels and the colors, and noticing discrepancies.

5.3 Fluid property visualization

The flood exposure visualization has been made to solve the problem of letting the user know how successful his efforts were, and in this aspect it performs quite well. However, it did not help the user understand which changes to make in order to improve the situation. Using the visualization, the user could only try a number of parameters and pick the ones resulting in the best outcome. Exploring the scenario with only the flood exposure visualization available proved to be quite challenging, as events that were seemingly random happened often, even in tracks that were unchanged for a long time.

The most common of such events was the fall of an already placed barrier that was considered to be stable. When enclosing an area with barriers, the additional pressure created by the water that could not pour out of the area almost always caused the destruction of at least one barrier, showing hidden weaknesses in previously created barriers. To understand why the barriers fall and to predict when they will do so, a new type of visualization is needed, one that shows the hidden properties of the particles.

The idea behind the visualization involved rendering the fluid in such a way that changes in various properties of the underlying particles are seen by the user. The properties that the SPH model defines and uses affect the simulation and can show changes in the fluid that do not directly manifest themselves in the shape of the fluid or its appearance in the existing particle rendering. Showing these properties would allow the user to make

better informed decisions, and hopefully help him find causes of unexpected behavior in the simulation system.

The problem of showing various properties of the fluid is by no means new, as there exists a collection of dedicated approaches called flow visualization [19] [27] [28]. The most commonly used approaches such as streamlines and isosurfaces assume that the domain of the input data is a grid. The meshless SPH data cannot be used with these techniques directly, but they can be applied by converting the data into a grid-based representation. The conversion is based on using SPH interpolation to find out the value of a property at every point in the grid, making the process very slow and inadequate for interactive purposes. Some visualization techniques made specifically for SPH have been developed [9], but the authors admit that much work remains to be done before they can be compared to the techniques developed for mesh-based simulations.

Given that SPH visualization techniques are anything but standardized, our approach was created to fit the needs of the simulation and the comparative rendering. Since the user is already familiar with the fluid being rendered realistically in three dimensions, the familiarity can be exploited by using the property information in combination with the shape information to produce a new rendering. This approach borrows more from volume rendering than from flow visualization, as the rendering is created by casting vertical rays through the fluid to extract a two-dimensional texture and mapping it onto the fluid. In this way, the user is presented with both shape and property information, and the format of the extracted property texture allows it to be combined with textures generated for other simulation states.

5.3.1 Property texture extraction

Aggregating the information about the properties of the fluid into a texture can be done in many ways, and the choice of how exactly to do it depends on what sort of information the user needs, and how often the texture needs to be updated. The ray-casting approach used in volume rendering creates a two-dimensional image to be shown to the user by projecting a ray through the volume space for every pixel of the image. Values encountered along the ray are composited in a certain way to produce the final color value for the pixel. Such an approach could be made to work with the SPH particles, but even the most efficient algorithm would require a number of interpolations proportional to the number of pixels and the neighbours of a particle, making it too slow for normal use. Even if the cost of the interpolations was to be removed by performing a resampling of the data to a grid, the volume rendering itself would still slow down the interaction significantly.

To offset these problems, our approach casts vertical rays into the fluid to generate a two-dimensional texture, and then applies the texture to the previously extracted fluid shape information. The process can be seen in Figure 19. One texture needs to be generated for each simulation state. This process takes time, but since applying the texture is a simple operation, the interaction is smooth as long as the user does not modify the data. The disadvantage of the approach is that information about the three-dimensional structures within the fluid can be lost. For a general SPH simulation, this would be a problem, but in our simulation scenario, the height of the fluid is small in

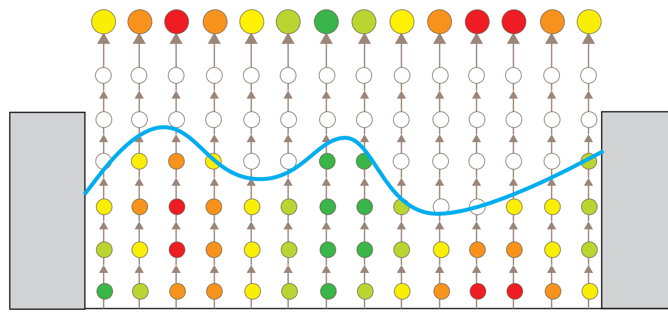
comparison to its other dimensions, and we assume that the lack of three-dimensional information does not hamper the user significantly.

The disadvantage is partially offset in the way the values are composited along the ray. The user can choose between several modes of compositing which are similar in concept to the ones used in volume rendering. An equivalent to front-to-back and back-to-front compositing does not exist, as it is impossible to make parts of the fluid transparent when the direction of the view need not match the direction of the rays. The first hit option functions much in the same way as in volume rendering, and extracts the value of the property on the surface of the fluid. The maximum and minimum options are similar to maximum intensity projection, extracting the largest or smallest property value encountered along the ray, while the average option takes the average value of samples found along the ray.

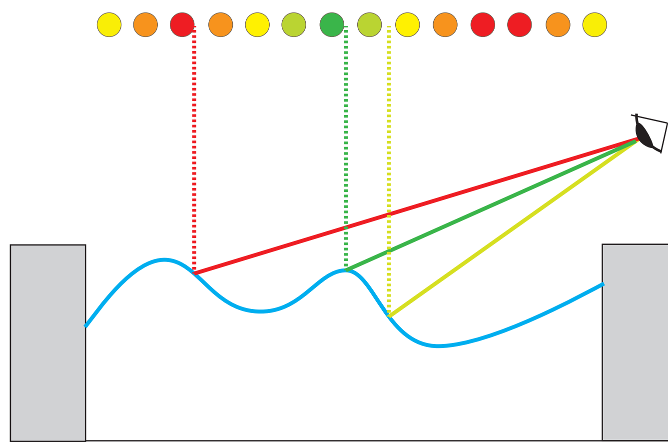
The difficulty of using these compositing techniques stems from the need to efficiently perform SPH interpolation. Since it would be grossly inefficient to use all the particles in every interpolation, space-partititioning structures are used to retrieve only a subset of the particles. The particles within are close enough to the target point that all the articles that need to be used in the interpolation are contained within it. The use of these structures drastically lowers the time needed to do SPH interpolation, but their construction takes a considerable amount of time. In cases where many interpolations have to be performed on the same set of particles, the additional time needed to construct the structure is small compared to the time saved by doing faster interpolation. In this approach however, interpolations are done only once to create the texture, making the construction time more important.

To find out which space-partitioning structure was best suited for our scenario, two such structures were used and tested. The first structure used was the octree, a tree data structure in which each node represents a part of space, and its eight children are created by subdividing the space of the node. Each leaf node contains all the particles that might be needed to perform SPH interpolation of a point within the node. Finding the particles needed for the SPH interpolation of a point amounts to finding the leaf that contains the point.

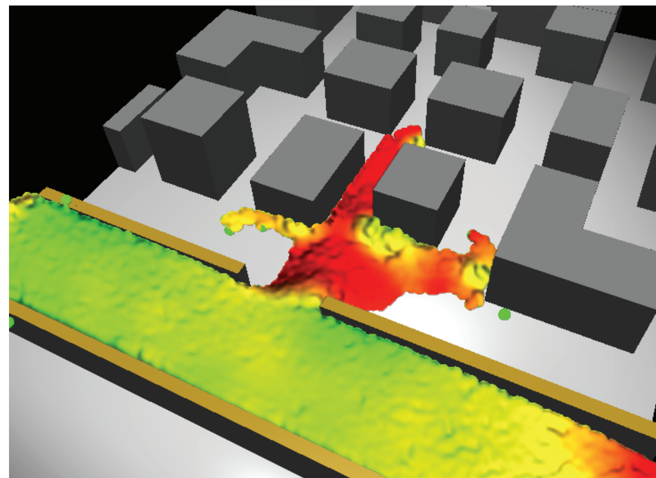
The second structure was a two-dimensional rectangular grid, which divided the area in which the simulation was performed into square-shaped cells. The cells are sized according to the smoothing length of the kernel, so that when inserting a particle, it needs to be placed only in the cell it belongs to and its eight neighbours. This allows the construction to be done quickly, and the particles are distributed almost equally thanks to the way the fluid particles are distributed. This structure was chosen with the specifics of the simulation scenario in mind, and it is likely it would not fare well in any other SPH simulation. If accuracy is to be sacrificed for speed, the cell size can be decreased so that some particles which should be used in the interpolation are not used. The loss in precision from using such a tradeoff can be small because of the way the cells are structured. With the required cell size, the neighbouring squares overlap an area bigger than the one that would be needed to cover all the particles. While smaller cells will not overlap the entire needed area, the extra space means that the lack of coverage will not be as severe as it would seem.



Texture extraction finding the maximum value



Mapping the texture to the surface



Final result

Figure 19: The extraction, mapping, and rendering of the texture.

Structure	Construction time (ms)	Interpolation time (ms)	Total time (ms)
Octree	1296	316	1612
Grid	7	261	268
Reduced cell grid (0.8h)	7	185	192

Table 2: Performance of various space-partitioning structures

The results of the test done to assess the peformance of various structures can be seen in table 2. The generated texture was made with 2500 values, 50 in width and 50 in length, spanning the area of the simulation scenario. Three structures were tested - the octree, the two-dimensional grid, and another version of the grid which uses a smaller cell size. The result that we expected was that the octree would perform the interpolation the fastest, but lag behind the other structures when the construction time is factored in. When the actual tests were performed, a different result was observed, as the octree performed the interpolation slower than either one of the grids. This behavior can have two causes: the octree performing more interpolations, or more particles being taken into account with each interpolation. Since the octree is constructed so that only particles close to the target point are taken into consideration when interpolating, it is unlikely that there is a big difference in the number of particles considered by the two data structures. Given that the octree can distinguish particles according to their Z coordinate, it should even have an advantage in this aspect.

One big advantage that the grid structure has over the octree is the ability to easily determine the maximum and minimum height of the fluid in one cell. Since the interpolations are done along a ray that extends vertically into the fluid, the heights at which the interpolations have to be started and stopped must be estimated. The estimate must be conservative enough so that no part of the fluid is missed, but the better the estimate, the fewer interpolations need to be done. The grid requires no additional efforts to find the minimum and maximum height, but the octree offers no easy way to get these estimates. Because of this, the octree must use estimates based on the bounding boxes of the fluid particles which are not as accurate.

Even if the octree was to perform the interpolation so quickly that the time spent was insignificant, it would still be inefficient in comparison to the grids because of its construction time. The construction of the grid structures is simple and takes up only a small amount of time, making them the ideal choice when the fluid is changed often. The time needed to construct the grid is dependent on the number of particles alone, and the grid size does not affect it. Unlike the construction, the interpolation is affected by the smaller number of particles taken into consideration in the reduced size grid, and takes less time to finish. Since the usage of the reduced size grid does not result in an image that differs significantly from the one created with the normal cell size, the reduced size grid turns out to be the best choice of all three tested structures.

Should the height estimates of the octree be improved, and its interpolation cost fall beneath the one belonging to the grid, in certain situations it might be better to use an octree. In the case that a more detailed texture has to be generated, the number of interpolations increases while the construction time stays the same. Given a large enough size of the texture, the octree might be the better choice, but it is not likely that such a large texture is necessary for the simulation scenario.

5.3.2 Texture application

After the texture has been prepared during the extraction process, it defines which property values correspond to xy coordinates in the scenario. This information can be used in the previously described particle rendering to color the fluid by using the property values along with a transfer function to generate new colors. To find out the correct property value for a pixel rendered on screen, it is first necessary to find out its three-dimensional coordinates. The particle rendering already calculates this value from the depth buffer as a part of the lighting calculations, making the xy coordinates of the pixel easy to retrieve. The xy coordinates are transformed into the texture space and the property value is extracted and converted into the color of the pixel.

The size of the texture determines the quality of the resulting image. The difference between various texture sizes can be seen in figure 20, which shows three screenshots of the same simulation state made with a varying texture size. The artifacts present on the low and medium resolution images appear because the values between the already calculated ones are linearly interpolated using texture-lookup mechanisms. Such artifacts may be sharply pronounced in highly sloped areas of the fluid. When the surface of the fluid is close to vertical, a very large number of pixel values has to be generated using only a small number of texture values, causing the surface to appear stretched in the final rendering.

To prevent non-existing values to be taken into account when interpolating the values at the edge of the texture, the shader does two interpolations instead of one. The first interpolation occurs in the property texture, which contains the attribute values where they were interpolated and zeroes elsewhere. The second is done in the mask texture. The mask texture is a very simple texture in which any texture element where the interpolation was successful contains the value 1, and the value 0 otherwise. Since both of the textures are two-dimensional, use the same type of interpolation, and are of the same size, when interpolated at the same location the texture elements taken into account and the factors they will be weighted by will also be the same.

When texture elements with undefined values are used in the property texture interpolation, the value will be different than it would be with those elements included. Since the undefined elements have the value zero, it is enough to normalize the interpolated value with the sum of all weight factors of the defined texture elements. Because the factors of the mask texture and the property texture are the same, the normalization value equals the interpolated mask texture value. The property value that should be used can be obtained by dividing the two interpolated values.

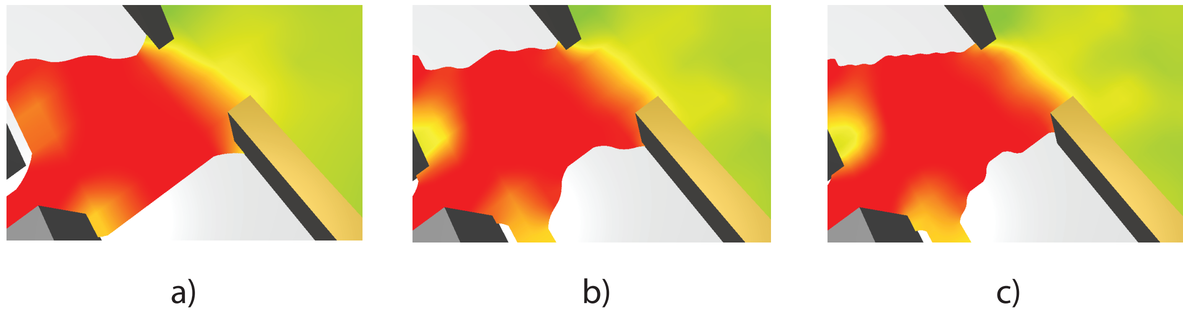


Figure 20: The fluid rendered with the velocity magnitude shown, using different texture sizes to store the data: a) low resolution b) medium resolution c) high resolution

As the purpose of this visualization is to allow the user to observe the properties of the fluid, smooth interaction is expected in order for the user to be able to explore the scenario. With this rendering method, the interaction is as smooth as in the original particle rendering, but only as long as the property texture does not need to be updated. The texture updates slow down the interaction because of the duration of the extraction process, but need to happen only when the input data or one of the settings related to the extraction process are changed. Unfortunately, this means that animating a sequence of simulated states is slow, as the texture needs to be rebuilt for every frame rendered. Using the visualization in such a way is possible, but not recommended, as its primary purpose is to allow the user to explore an interesting state of the system and not to show the changes of properties in time.

5.3.3 Comparative rendering

Despite the difficulties of using this visualization to show changes in time, it can still be used to compare multiple simulation states in an aggregate rendering approach. The idea of the approach is to combine the property textures of each of the states into one texture using an aggregation operator, and to show the aggregated data in a way similar to the one-state approach. With the properties combined, the user can detect areas where they take on interesting values. An example would involve using a minimum operator to find areas that have consistently high values of the measured property, or a maximum operator to find areas with consistently low values.

The aggregation operators used in this rendering differ from the previously mentioned ones because of the possibility that a texture contains values that can be interpreted as there being no data present at the point. Since the texture extraction process is applied to an area, it is performed even in places where there is no fluid present. The lack of a property to extract at such locations is denoted by a special value which is written into the property texture. When the aggregation operator is applied to combine the textures, only the values that signify the presence of data are used to create the final result. If no such value is present, the value for no fluid is written there instead. By combining the data in this way, the operators actually create a union of all the shapes. An example of the operator being used on textures can be seen in Figure 21.

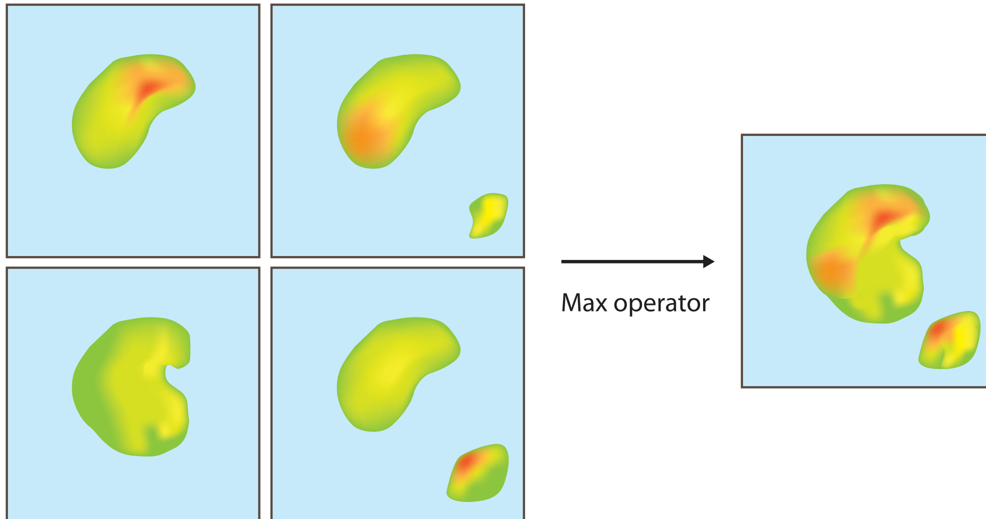


Figure 21: The process of combining multiple textures using the max operator. The color blue represents the undefined areas.

Embedding this information into the standard rendering is troublesome because a simple reuse of the rendering with the new property texture would show the proper values on the surface of the context fluid, but all the other information would be lost. Without an idea of the shape and the property values of the other fluids that are not shown, the rendering does not perform a comparison, making it only marginally more useful than the normal rendering. The problem was to find a way to represent both the shape of the context fluid and the additional information gained by the aggregation in the same rendering.

The first option is finding a way to show the shapes of all the fluids at the same time. Displaying the shapes in such a way that the human eye can recognize them is not an easy task, and there has been some research into it. Bair et al. [7] investigate ways in which two surfaces can be textured and colored in order to make them distinguishable from one another. The color and texture selections were chosen by using a genetic algorithm to find promising combinations, which were further improved after additional analysis. While the results are somewhat distinguishable, the use of more than two layers was not discussed in the paper. Even though the results of the paper were not directly usable in the thesis, they did give a clear sign that showing the shapes of all the fluids without restrictions is a very difficult task.

When using some restrictions, it is possible to show shapes using transparency and different colors only. In the earthquake visualization [16] described in section 2.4.2, the isosurfaces created for different magnitudes define shapes that are nested in one another. These shapes are made distinguishable by using transparency and varying lightness of colors. The outermost shapes are highly transparent and brightly colored, and the inner shapes have different, but darker colors with less and less transparency as they approach the innermost shape. By coloring the isosurfaces in this way, each layer of them can be distinguished. Applying this approach to the rendering problem is difficult because of the lack of a similar type of guaranteed nesting between the fluid shapes. However, if the

union of all the fluid shapes is considered to be a fluid shape itself, any individual fluid shape is guaranteed to be contained in it.

Based on the principles displayed in the earthquake visualization, two shapes could be rendered distinguishably - the context fluid and the fluid union. However, the problem of applying the property textures onto them would be difficult for two reasons: first of all, the earthquake visualization uses color to make the different shapes stand out. In the property rendering, the color is reserved for showing the values of properties. Using color to both highlight the shape and display the values would result in two different sets of colors used to show the same values of properties, confusing the user. The second and more important reason involves the user's inability to perceive two stacked colored textures separately. In many cases, the context fluid could only be seen through the transparent fluid union, making the property colors useless. Although this approach had not yielded any useful results as well, it served as another argument against using multiple shapes.

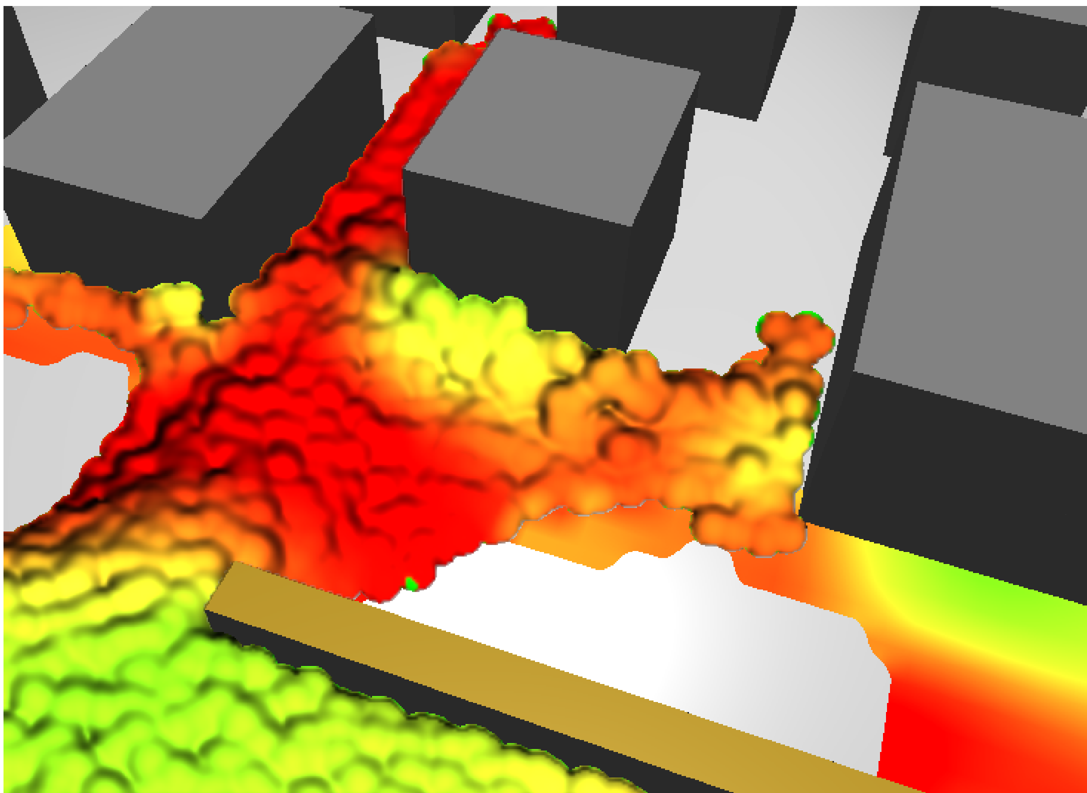


Figure 22: Aggregate rendering of fluid properties

The approach which had shown itself to be successful used a simple solution that discarded all shape information except the one from the context state, and managed to show the rest of the aggregated values despite that. Since the bottom of the simulation scenario is gray and even, it is possible to use the ground to show the aggregated values and the shape of all the fluids. To accomplish that, a new graphical primitive was introduced - a quad that floated just barely above the ground, colored according to the particle properties in much the same way as the surface of the fluid was colored. Since the shader that renders the data ignores the areas where the value of the data is undefined, the shape drawn looks like a two-dimensional projection of the fluid union. The context fluid

is rendered as well, using the aggregated texture to color its surface. As can be seen in Figure 22, the end result shows how the fluid has spread and contains all the aggregate information, and the context shape is easily distinguishable as well. To show how the various phases of the fluid property aggregation correspond to ones shown in the diagram pictured on Fig. 15, Figure 23 shows the diagram with screenshots of the intemediate stages.

While it is our desire to use shape information from more than one simulation state, it has shown itself to be impossible or very difficult. The chosen solution is a compromise. It relies heavily on the properties of the simulation scenario, but manages to show the aggregate information. The advantage of this approach is that the user is still free to switch the context state, and thus select the fluid whose shape is to be rendered on screen. An additional option the user has is to turn off particle rendering, causing only the quad colored with the aggregate data to be shown on screen.

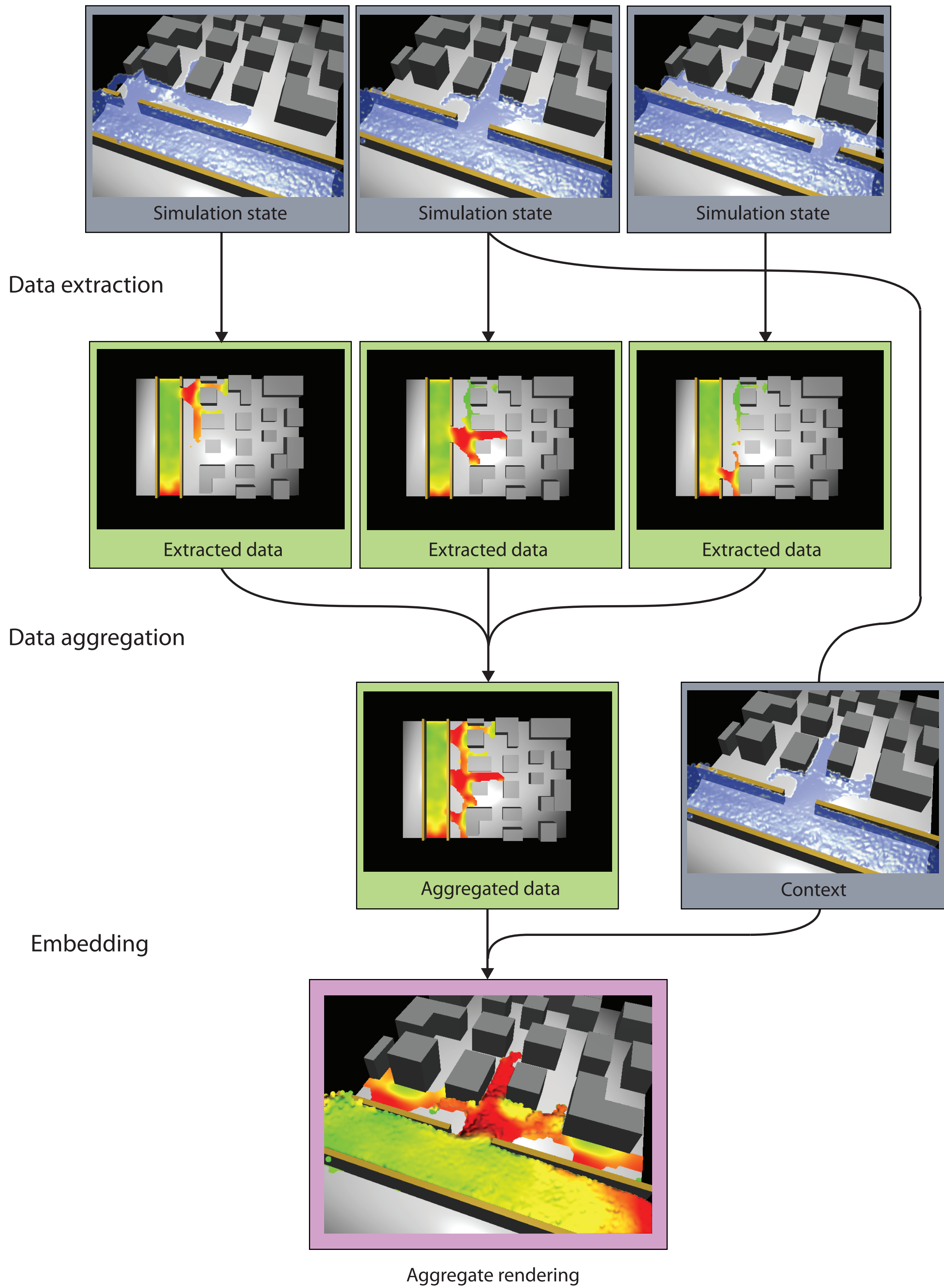


Figure 23: Diagram showing the stages of particle property aggregation.

6 Results and discussion

When examining the system as a whole, its usefulness to the user can be measured in terms of what it can do and how fast it can be done. The various types of interaction with the system have been described in detail in previous sections, so this section will focus on how the speed of execution affects the user's ability to interact with the system.

6.1 Performance

To be able to judge how well the user can interact with the system, it is necessary to know the time needed for the system to perform various actions and present the user with a response. The times stated here are averages of measurements made when the objects simulated were most complex. Assuming a frame rate of 25 frames per second is interactive, the server must produce and deliver one frame to the client every 40 ms for the user to percieve the interaction as smooth. The base overhead of any action performed by the framework stems from its distributed nature. Even when both client and server are running on the same computer, the cost of sending a request and receiving the results amounts to 10 ms on average. The delay is caused by the socket interface used to communicate data and the XML parsing of the requests and the results.

After the time taken by the framework is deducted, we are left with 30 ms in which the simulation and rendering must be performed. The performance of simulation varies greatly with the number of objects and particles in the scene, but in the most complex scene, approximately 650 ms are needed to simulate one second of the scenario's behavior. Assuming a real-time simulation, each 40 ms frame takes 26 ms to simulate, leaving only 4 ms for the rendering to be performed. This short period of time is not sufficient for the difficult task of rendering the scene, which takes up 22 ms on average. The large amount of time needed to render the scene is caused by the particle rendering, which takes up 17ms to render approximately 20000 particles, the maximum encountered when using the simulation scenario.

The sum of simulation and framework costs leaves an amount of time which is not sufficient to perform demanding rendering. When the simulation is performed in real-time, the frame rate which the system is capable of sustaining falls beneath the desired interactive one, clocking in at about 13 to 15 frames per second. However, it is not always necessary or possible to perform the simulation in real-time. Given that the flooding scenario takes a long time to unfold, it is practical to use larger time steps to see how the state of the scenario changes. We've found that a time step of 0.5 seconds fits the events well. When using such a time step, the sheer amount of time needed for the simulation to be done eliminates the possibility of the user interacting with the system as the simulation takes place. To allow the user to still be able to see how events unfold, we take advantage of the framework's ability to store data produced by the nodes. The results of the simulation are calculated in advance when the user selects tracks of interest using the World Lines. When the lengthy process of simulating the tracks is complete, the system can be interacted with at acceptable frame rates.

The interaction described so far involves only the base functionality of the system and not the comparative rendering. The performance problems are only amplified by its introduction, as the production of data used for comparative rendering takes up additional time. The flood exposure visualization requires an additional 50 ms per track for the building colors to be generated, and 200 ms per track are needed to create the attribute textures used to color the fluid. For both of the aggregations, the costs of aggregating the data are insignificant in comparison to the time needed to extract it.

While the 50 ms needed to calculate the flood exposure still allow for some degree of interactivity while the calculations take place, the 200 ms required to explore the properties of the fluid do not. This presents a problem similar to the one encountered when handling long simulation times, but the same solution can not be applied in this case. Unlike the data produced by the simulation, the extracted data is internal to the nodes, and the framework does not offer a way to automatically save and reuse it. Instead, the framework offers another solution for saving internal data by using the attribute system. Each data object can have additional attribute objects attached to itself that persist as long as the original data object. The attribute system can be used to associate the extracted data to the input data, but the caching process is not straightforward. The textures are generated from two data objects, the particles and the attribute vector, instead of one, and manual checks that associate pairs of objects to cached data must be added. When using the attribute system, it is possible to perform data extraction calculations for each data object only once. This makes it possible for the user to perform most of the calculations in advance and explore the results afterwards. Because there is no data object the aggregation results can be mapped to, they cannot be cached. The aggregation calculations must be performed every time the user changes the frame or a setting.

However, it should be pointed out that the caching procedure is not a solution for the problem, but only a remedy. In order to use the fluid property aggregation as a part of the interaction, the extraction procedure should be moved to the GPU, where the processing power should reduce the long calculation times.

6.2 Usability

While most of the advanced functionality takes a significant amount of time to be ready for use, these delays in the use of the system are not as noticeable as they may seem. The people using the World Lines interface to explore the simulation scenario followed a certain pattern of exploration. The pattern consisted of short simulation runs interspersed with parameter changes. While the simulation delay was present at this point, because of the shortness of the runs, it did not slow down the exploration significantly. The flood exposure visualization was used when multiple alternatives were compared, and while it took some time to generate results for every track considered, the delay was acceptable. Once critical tracks were identified, the fluid property visualization was used to compare them. All in all, the actions that took the longest to complete were used rarely, allowing the exploration to proceed relatively uninhibited.

An additional concern that surfaced during the testing of the system revolved around

the use of memory. The framework lacks a memory management system that could delete unused data and regenerate it if needed later on, meaning that all produced data continues to reside in the memory. Given that most of the data produced is particle information that takes up a lot of space, we wondered how much exploration the systen could support before running out of resources. During an extremely detailed session of experimentation in the simulation scenario that included over 25 tracks, the generated data exceeded two gigabytes of size. While such an amount of data can be handled on any modern computer, the event does show that in the case of a more complex simulation being handled by the framework, a better memory management system will be needed.

7 Conclusion

The goal of this thesis was to develop a solution that would allow the user to interactively explore a simulation scenario, and to find approaches that could be applied to many similar tasks, in terms of both simulation and visualization. While it is possible to extract models that describe how a generic process of simulation or comparative visualization works, the experience of designing implementations of these models has shown that the specifics of the modeled physical processes still have the biggest influence on the final solution. The only thing that can be said to be common to the various approaches are patterns of user interaction. The aggregate rendering model builds on this assumption by providing a way in which many different visualization techniques can be extended to accomodate such interactions. The simulation system is designed to extend existing simulation engines, allowing them to support the exploration of parameters and events without requiring significant changes to their internal structure.

In some ways, the system that was developed as a part of this thesis is a transitional system. It mediates between the steering capabilities that World Lines introduces and the classic visualization and simulation techniques. It may be possible that in the future, new visualization techniques better suited to comparative rendering will be developed. Right now, creating new techniques is hard due to a lack of knowledge about the needs of the user and the nature of the insight that the user can achieve by examining multiple simulation states. The users exploring the simulation scenario using our system help define the goals that the next generation of tools aims to achieve.

The ever-increasing complexity of simulations that researchers are performing suggests that sooner or later, there will be a need for tools that allow the user to steer incredibly complex simulations and filter the enormous amounts of data created. This thesis is just the beginning of the work that needs to be done to successfully tackle this challenge.

7.1 Future work

The further development of the system can proceed in many different ways. The idea of using aggregate renderings to perform visual queries is powerful, and may serve well as a tool that allows the user to quickly find frames with desired properties. The presence of such a tool in the framework would open up the possibility of doing parameter studies by having the system automatically create tracks that vary in selected parameters. Without a powerful filtering mechanism, the user would have only a limited use of the wealth of information generated by such an approach, as the aggregate renderings offer no way of linking a specific property to a frame.

The aggregate renderings currently exist only for physical objects from which it is easy to extract an intermediate representation. One big step towards proving that they are an approach that can work universally would be applying them to the barriers. The difficulty of that endeavour stems from the fact that the barriers are easily distinguishable, and a visualization makes sense only if they can be tracked. Tracking the barriers across multiple frames while aggregating the information about their positions is not an easy task, and

it would be the next logical step in the development of the visualizations related to the simulation scenario.

The simulation could benefit from better memory management. It would be interesting to explore whether the idea of a saving frequency could be replaced by dynamic data management that would be used to save memory in memory-critical systems by removing already present values from underused tracks.

As far as the performance issues of the system are concerned, there is unfortunately no easy way of dealing with them. Most of the overhead is caused by the simulation engine, and replacing it would be a very difficult task. Many faster SPH simulation engines exist, but they lack the support for interaction with objects that is vital to the simulation scenario. The fluid rendering is also not likely to be replaced until a new and better method of rendering SPH data is created. The aggregation process could be improved by attempting to use a different grid, as both a three-dimensional and a hexagonal grid would likely perform better than the current one. Additional improvements could be made by porting the sampling code to CUDA, and possibly getting direct access to PhysX-allocated particle buffer that resides in the GPU memory. It would be worthwhile to explore whether the performance can be improved by using better hardware.

7.1.1 Exploration in aggregate rendering

As the example of the particle property aggregate rendering has shown, adding a certain degree of interactivity to the rendering can help compensate for not being able to show all the data or to allow the user to experiment with the data. Since the way the user is supposed to interact with the aggregate renderings using World Lines has not been explained in more detail, this section will serve to explain how that interaction can occur, and what changes should be made to it in order for it to be more flexible and more powerful.

In the current setup of World Lines, the user can perform a selection of simulation states only by choosing a time value and a number of parallel worlds of the current time. This restricts the states that the user can select to the ones that have the same time value, which limits the flexibility of aggregate renderings considerably. Without this restriction, the same tools used to analyze parallel worlds could be applied to sequential time values or similar states that were not grouped around the same time value.

These restrictions are scheduled to be removed in the next version of World Lines, and replaced with a marker system which would allow the user to select an arbitrary number of states in World Lines, either by clicking on particular frames or brushing an entire sequence of frames on one or more tracks. However, once they are removed, the basic problem of interacting with the aggregate renderings remains the inability to connect aggregated results to the frames that influenced them. To find out what track caused a value that stands out, the user must manually add or remove tracks and guess what effects the changes will have on the rendering.

To counter this problem, a new type of interaction can be used, one which relies on the

ability of the rendering to detect how a primitive that the user has clicked on is connected to the extracted data. For both of the aggregate renderings presented here, that problem is solved easily. The flood exposure rendering needs only to detect which building the user has selected. The particle property rendering can perform an inverse transformation of the location the user has selected in screenspace into the space of the fluid, and map that three-dimensional location to the property texture. Once such a connection is made, it can be used to make a new selection of simulation states based on the properties of the selected area.

An example of the interaction that would be possible in this model involves a user examining a large number of states and trying to find out which ones of them have high pressures near one barrier and low pressure near another. The user begins by selecting the particle property rendering, the density property, and the maximum aggregation operator. The rendering shows the maximum values of pressure encountered in all simulation states he had selected. The user locates one of the barriers in question, finds an area in front of it, which should be marked as an area of high pressure if any simulation state has high pressure in that area, and clicks on it. The click is mapped to the exact location in the property texture, and a search of all the simulation states is carried out. The simulation states which have a value at that point similar to the one found in the aggregated texture are left in the selection, while all the others are removed. After the search is completed, the user has a selection of simulation states with high pressure in the specified area. To find those which have low pressure near another barrier, the user switches the aggregation mode to minimum and repeats the process with the other barrier, further reducing the number of tracks in his selection to contain only the ones which satisfy the conditions he has specified.

The example shows how the user can perform complex brushing of tracks by using no more than selections of aggregation modes and clicks on the rendering in question. Even though this type of interaction is interesting, it has not been implemented in the course of this thesis. The reasons are a lack of time and the current architecture of the system, which is distributed and assumes that all settings changes originate from the client. Since the selection of simulation states is a setting, this model of interaction would require that the server changes the settings too, something which is possible in this model only as a workaround at this time, thus making it hard to implement. Despite that, it is certainly a powerful addition to the set of analysis tools and and will be explored in the future should circumstances allow it.

7.1.2 Multiple gradual events use

The problem with the bag dropping and gradual changes in general is that they require the running process to be fragmented. Whenever a new bag must be introduced into the system, the object must be created and added using the appropriate SDK functions. That event happens at a fixed time, and should a simulation system be ordered to perform a run during which the event takes place, the run has to be split into two or more separate runs. In general, using multiple gradual changes at the same time could cause the run to be fragmented to the point where it becomes a danger to efficiency. Figure 24 shows the fragmentation that would result when more than one gradual change was to be introduced

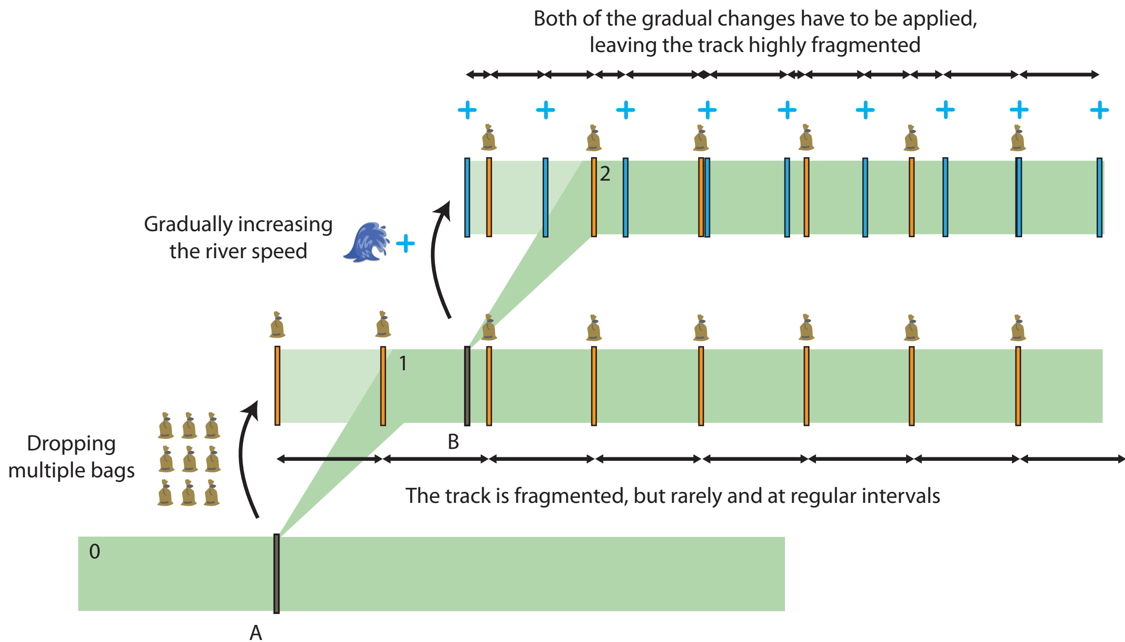


Figure 24: The occurence of fragmentation when using multiple gradual changes.

into the system. Two changes are introduced into the system - the dropping of the bags in track 1 and a gradual increase in the river speed in track 2. The first change lasts long enough that the bag dropping events are still present in track 2 along with the river speeed increase events. As can be seen in track 2, unless the periods of the changes are matched, the number of breaks that would have to be made in a run multiplies with every gradual change added. When a sufficient amount of gradual changes is applied, the run can be subdivided to the point where only a miniscule amount of time must pass before another event occurs.

A possible solution for this problem would require the ability to schedule events slightly earlier or later so that multiple events can be handled at the same time. However, care should be taken that these shifts do not reduce the quality of the simulation. Given that only one gradual change mechanism is currently present in the simulation system, such a modification is not necessary at the moment, but may have to be implemented if more gradual changes are added.

List of Figures

1 The differences and similarities of the procedures . . . 2
2 A property field of three particles extracted using SPH interpolation . . . 6
3 A screenshot of a data-flow setup . . . 9
4 The World Lines window after a period of experimentation with the simulation scenario . . . 10
5 Tracks colored according to how much danger the buildings are in . . . 12
6 A parallel coordinates visualization of car fuel consumption data. Taken from the Protovis website [33] . . . 13
7 The UML class diagram of the simulation classes . . . 18
8 The pseudocode of the abstract simulation node . . . 21
9 An example of a simulation run that has to be broken up into smaller runs 22
10 An illustration of how various parts of the system are connected. Arrows indicate how changes propagate through the system. . . . 24
11 A class UML diagram of various classes used for rendering in the framework. 27
12 Various stages of the particle rendering algorithm . . . 31
13 Images showing stages of the particle rendering algorithm . . . 32
14 The depth buffer before and after the depth smoothing. . . . 33
15 The various phases of a visualization that produces an aggregate rendering. Blue rectangles represent simulation states, green ones represent extracted features and data, and the pink rectangle is the final rendering. . . . 36
16 Various heights considered while determining the height of water touching a side of a building. a) Maximum height found. b) Average height. c) The acceptable range of heights, based on the average value. . . . 39
17 Pseudocode of the CUDA height extraction . . . 40
18 The flood exposure visualization showing which buildings are affected by the spreading water. . . . 41
19 The extraction, mapping, and rendering of the texture. . . . 44
20 The fluid rendered with the velocity magnitude shown, using different texture sizes to store the data: a) low resolution b) medium resolution c) high resolution . . . 47
21 The process of combining multiple textures using the max operator. The color blue represents the undefined areas. . . . 48
22 Aggregate rendering of fluid properties . . . 49
23 Diagram showing the stages of particle property aggregation. . . . 51
24 The occurence of fragmentation when using multiple gradual changes. . . 58

References

[1] *SPHysics*. http://wiki.manchester.ac.uk/sphysics/index.php/SPHYSICS_FAQ, September 2010.

[2] Adobe Systems Incorporated, *Adobe air system requirements*. http://www.adobe.com/products/air/systemreqs/, September 2010.

[3] Adobe Systems Incorporated, *Flex: An open source framework for developing web applications*. http://www.adobe.com/products/flex/, September 2010.

[4] O. Agertz et al., *Fundamental differences between SPH and grid methods*, Monthly Notices of the Royal Astronomical Society, 380 (2007), pp. 963–978.

[5] M. Alexa, J. Behr, D. Cohen-Or, S. Fleishman, D. Levin, and C. T. Silva, *Point set surfaces*, in VIS '01: Proceedings of the conference on Visualization '01, IEEE Computer Society, 2001, pp. 21–28.

[6] G. Amara, *N-body / particle simulation methods*. http://www.amara.com/papers/nbody.html, March 2000.

[7] A. Bair, D. H. House, and C. Ware, *Texturing of layered surfaces for optimal viewing*, IEEE Transactions on Visualization and Computer Graphics, 12 (2006), pp. 1125–1132.

[8] R. A. Becker and W. S. Cleveland, *Brushing scatterplots*, Technometrics, 29 (1987), pp. 127–142.

[9] J. Biddiscombe, D. Graham, and P. Maruzewski, *Visualization and analysis of SPH data*, ERCOFTAC Bulletin, 76 (2008), pp. 9–12.

[10] R. Bürger and H. Hauser, *Visualization of multi-variate scientific data*, in EuroGraphics 2007 State of the Art Reports (STARs), 2007, pp. 117–134.

[11] H. Doleisch, M. Mayer, M. Gasser, P. Priesching, and H. Hauser, *Interactive feature specification for simulation data on time-varying grids*, in In Conference on Simulation and Visualization 2005, 2005, pp. 291–304.

[12] R. Fuchs, J. Kemmler, B. Schindler, J. Waser, F. Sadlo, H. Hauser, and R. Peikert, *Toward a lagrangian vector field topology*, Computer Graphics Forum, 29 (2010), pp. 1163–1172.

[13] J. N. Ghazali and A. Kamsin, *A real time simulation and modeling of flood hazard*, in ICS'08: Proceedings of the 12th WSEAS international conference on Systems, World Scientific and Engineering Academy and Society (WSEAS), 2008, pp. 438–443.

[14] R. A. Gingold and J. J. Monaghan, *Smoothed particle hydrodynamics - theory and application to non-spherical stars*, Royal Astronomical Society, Monthly Notices, 181 (1977), pp. 375–389.

[15] R. Hoetzlein and T. Höllerer, *Interactive water streams with sphere scan conversion*, in I3D '09: Proceedings of the 2009 symposium on Interactive 3D graphics and games, ACM, 2009, pp. 107–114.

[16] T.-J. Hsieh, C.-K. Chen, and K.-L. Ma, *Visualizing field-measured seismic data*, in In Proceedings of Proceedings of IEEE Pacific Visualization Symposium, March 2010, pp. 65–72.

[17] A. Inselberg, *The plane with parallel coordinates*, The Visual Computer, 1 (1985), pp. 69–91.

[18] H. Janicke, A. Wiebel, G. Scheuermann, and W. Kollmann, *Multifield visualization using local statistical complexity*, IEEE Transactions on Visualization and Computer Graphics, 13 (2007), pp. 1384–1391.

[19] R. S. Laramee, H. Hauser, H. Doleisch, B. Vrolijk, F. H. Post, and D. Weiskopf, *The state of the art in flow visualization: Dense and texture-based techniques*, Computer Graphics Forum, 23 (2003), p. 2004.

[20] L. B. Lucy, *A numerical approach to the testing of the fission hypothesis*, Astronomical journal, 82 (1977), pp. 1013–1024.

[21] J. J. Monaghan, *Smoothed particle hydrodynamics*, Annual review of astronomy and astrophysics, 30 (1992), pp. 543–574.

[22] NVIDIA, *Cuda reference manual 2.3*. http://developer.download.nvidia.com/compute/cuda/2_3/toolkit/docs/CUDA_Reference_Manual_2.3.pdf, September 2010.

[23] NVIDIA, *NVIDIA CUDA architecture: Introduction & overview*. http://developer.download.nvidia.com/compute/cuda/docs/CUDA_Architecture_Overview.pdf , September 2010.

[24] NVIDIA, *PhysX features*. http://developer.nvidia.com/object/physx_features.html, September 2010.

[25] S. G. Parker and C. R. Johnson, *Scirun: a scientific programming environment for computational steering*, in Supercomputing '95: Proceedings of the 1995 ACM/IEEE conference on Supercomputing, ACM, 1995, p. 52.

[26] U. U. I. performance evaluation taskforce (IPET), *Final report*. https://ipet.wes.army.mil/, Mar 2007.

[27] F. H. Post, B. Vrolijk, H. Hauser, R. S. Laramee, and H. Doleisch, *Feature extraction and visualisation of flow fields*, Eurographics 2002 State-of-the-Art Reports, (2002), pp. 69–100.

[28] F. H. Post, B. Vrolijk, H. Hauser, R. S. Laramee, and H. Doleisch, *The state of the art in flow visualisation: Feature extraction and tracking*, in Computer

Graphics Forum, vol. 22, John Wiley & Sons, 2003, pp. 775–792.

[29] K. Potter, A. Wilson, P.-T. Bremer, D. Williams, C. Doutriaux, V. Pascucci, and C. Johhson, *Visualization of uncertainty and ensemble data: Exploration of climate modeling and weather forecast data with integrated visus-cdat systems*, Journal of Physics: Conference Series, 180 (2009), p. 012089.

[30] J. C. Roberts, *State of the art: Coordinated & multiple views in exploratory visualization*, in Proceedings of the 5th International Conference on Coordinated & Multiple Views in Exploratory Visualization (CMV2007), IEEE Computer Society Press, 2007.

[31] I. D. Rosenberg and K. Birdwell, *Real-time particle isosurface extraction*, in Proceedings of the 2008 symposium on Interactive 3D graphics and games, 2008, pp. 34–43.

[32] A. M. A. Sattar, A. A. Kassem, and M. H. Chaudhry, *Case study: 17th street canal breach closure procedures*, Journal of Hydraulic Engineering, 134 (2008), pp. 1547–1558.

[33] Stanford Visualization Group, *Protovis.* `http://vis.stanford.edu/protovis/`, September 2010.

[34] E. R. Tufte, *The Visual Display of Quantitative Information*, Graphics Press, second ed., 2001.

[35] W. J. van der Laan, S. Green, and M. Sainz, *Screen space fluid rendering with curvature flow*, in I3D '09: Proceedings of the 2009 symposium on Interactive 3D graphics and games, ACM, 2009, pp. 91–98.

[36] J. Waser, R. Fuchs, H. Ribičić, B. Schindler, G. Blöschl, and E. Gröller, *World lines*, in IEEE Transactions on Visualization and Computer Graphics 15(6), to appear, 2010.

[37] C. Weaver, *Cross-filtered views for multidimensional visual analysis*, IEEE Transactions on Visualization and Computer Graphics, 99 (2009), pp. 192–204.

[38] J. Woodring and H.-W. Shen, *Multi-variate, time varying, and comparative visualization with contextual cues*, IEEE Transactions on Visualization and Computer Graphics, 12 (2006), pp. 909–916.

[39] M. Zwicker, H. Pfister, J. van Baar, and M. Gross, *Ewa volume splatting*, in VIS '01: Proceedings of the conference on Visualization '01, IEEE Computer Society, 2001, pp. 29–36.

Comparative Rendering of Simulation Scenarios

In the course of this thesis, a software solution that can be used to handle the simulation and visualization of multiple flooding scenarios was developed. Multiple alternatives can be simulated at once, and the user can jump between various simulation states and explore them at will. New alternatives can be explored by changing parameters or introducing events into the simulation. The thesis is centered around the study of breach closure procedures used to repair the damage done to the levees of New Orleans after hurricane Katrina. A simulation scenario resembling the original study was made and used to explore alternative ways of closing the breach. The results of the simulation are rendered to the user using the screen space fluid with curvature flow rendering method. To allow ease of exploration, additional views that allow the user to compare simulation states were developed. These views show which buildings are threatened in various states, and the properties of the simulated fluid across multiple states. It is shown that these views follow a common approach that can be used to create new comparative renderings.

Keywords: Fluid simulation, SPH, comparative rendering, visualization, fluid rendering, computational steering, World Lines

Usporedni prikaz simulacijskih scenarija

U sklopu ovog rada je razvijena softverska podrška za simuliranje i vizualizaciju više scenarija poplavljivanja. Korisnik može ispitivati alternative mijenjanjem parametera ili poticanjem dogaaja u simulaciji. Inspiracija za korišteni scenarij je studija različitih načina zatvaranja proboja nasipa, temeljena na događajima koji su uslijedili nakon što je tornado Katrina poharao New Orleans. Napravljen je simulirani scenarij koji sliči uvjetima iz studije, i korišten da se testiraju različiti načini zatvaranja proboja. Rezultati simulacije se iscrtavaju korištenjem "Screen space fluid with curvature flow" metode. Radi olakšavanja istraživanja, razvijeni su dodatni prikazi koji omogućavaju korisniku da usporedi simulirana stanja. Pogledi pokazuju koje su zgrade u opasnosti i svojstva simulirane tekučine. Pokazujemo da pogledi dijele zajednički pristup koji može biti korišten za stvaranje novih usporednih prikaza.

Ključne riječi: simulacija tekućina, SPH, vizualizacija