A Visual Editor for a Computational Fluid Dynamic Model

DIPLOMARBEIT

für die Prüfung zum

Diplom-Ingenieur (Berufsakademie)

Im Studiengang Informationstechnik
an der Berufsakademie Karlsruhe

von

Sean Farrell

September 2006

Bearbeitungszeitraum	3 Monate
Kurs	TIT03VIN
Ausbildungsfirma	Bundesanstalt für Wasserbau Karlsruhe
Gutachter der Ausbildungsfirma	Dr. Ing. Jann Strybny
Gutachter der Studienakademie	Dr. Ing. Jörg Matthes

A Visual Editor for a

Computational Fluid Dynamic Model

Diploma Thesis

Sean Farrell

Abstract

The computational fluid dynamics model *NaSt3DGPF* was recently developed at the *Institute for Numerical Simulation* at the *University of Bonn* and is currently extended in cooperation with the German *Federal Waterways Engineering and Research Institute*. Though the software is fully functional it lacks the commodity of a visual editor to create the input for simulations. This thesis describes the development of a visual editor for the computational fluid dynamics model *NaSt3DGPF* and the software engineering principals behind it.

Table of Contents

Illustration Index

Index of Tables

Chapter 1 Introduction

1.1 Computational Fluid Dynamics and the Federal Waterways Engineering and Research Institute

The German *Federal Waterways Engineering and Research Institute* was founded 1948 as a follow up of the *Prussian Research Institute for Water, Terrestrial and Ship Construction,* which in turn founded in 1903. Its main task is to provide expert advice for the construction and maintenance of waterways in Germany.

Since the beginning of the institute small and large scale laboratory models have been used for analyze the flow of water. With the advent of computers it has become possible to simulate computational fluid dynamics. In recent years computational fluid dynamics are becoming an economic alternative to laboratory models.

1.2 The Problem

Because of the special problem domain it has been proven that most commercial software is not suited for the applications of the *Federal Waterways Engineering and Research Institute.* The problem is that commercial software is optimized for the analysis of aerodynamics for cars and planes. As a result the software *NaSt3DGPF* was extended and adapted for this special problem domain in cooperation with the *Institute for Numerical Simulation* at the *University of Bonn,* the original authors of the software.

Although *NaSt3DGPF* is fully functional, it lacks the means to create a simulation in a user friendly manner. The input for the software has to be provided in a cryptic text format. In previous works two utilities where developed to overcome certain limitations in the simulation software. One utility is used to convert the geometry coming from the

computer aided design (*CAD*) software in to *NaSt3DGPF's* native format. The other utility is used to generate a definition for a computational grid, based on control points. In later development those two utilities where joined into one graphical tool. In addition it provided the means to preview the discretized geometry.

In this work the development process of a full editor for a simulation is described. That is to provide the means to create a full simulation including the obstacle geometry, the initial air and water conditions, create the grid and set all parameters that the application provides. To ease the creation the entities of the simulation should be displayed in a three dimensional visualization from which the situation can be rapidly understood.

1.3 The Name of the Application

The editor was named *visualnav*. This comes from the fact that the data represented in the *NAV* format, *NaSt3DGPF's* native format, is visually presented to the user. In this work the term *visualnav* and "the editor" or similar terms refer to the same application.

1.4 A Note about Libraries

The libraries *GTK*, *GDK*, *Glib*, *libglade*, *libxml* and *libxstl* are all implemented in C with a emulation of object orientation. All libraries have a C++ wrapping that put the pseudo object orientation in real objects. In the work to this thesis the C++ wrappings, for example *GTKmm* or *libxml++*, where used. Because the functionality is implemented in the C libraries and not the C++ wrappings of them, the C libraries are only states and it is implied that the C++ wrappings where actually used.

For *openGL* the C interface was used since it is implemented in a procedural way and there are not efficient wrappings for C++.

Chapter 2 Requirements

2.1 Target Users

The potential users of the visual editor are also the users of *NaSt3DGPF*. They are engineers of different disciplines related to hydraulic constructions and vessels.

2.2 The Anatomy of a Simulation

The software *NaSt3DGPF* describes a simulation with a grid, parameter and objects.

The numeric grid on which the simulation is discretized is a rectilinear grid that may be freely defined along each axis. The grid and so the simulation volume is defined between the origin and an arbitrary point lying in the first octant. In previous works a utility was developed that generated a definition for the grid by interpolating between control points. This improved definition is exposed to the user of the visual editor.

The simulation parameters define physical properties, such as viscosity of a medium, numeric settings, such as the iteration count of the numeric solver or application settings, such as which solver to use.

Parameters for volumes are defined by so called objects. In the case of *NaSt3DGPF* every object has a number of parameters that can be mixed and matched, even producing invalid or senseless behavior. Since the users are acquainted with *NaSt3dGPF* this mediocre definition is acceptable.

2.3 Functional Requirements

This section describes the functional requirements to the software in use cases. The illustration 1 shows the requirements recursively split up in single use cases.

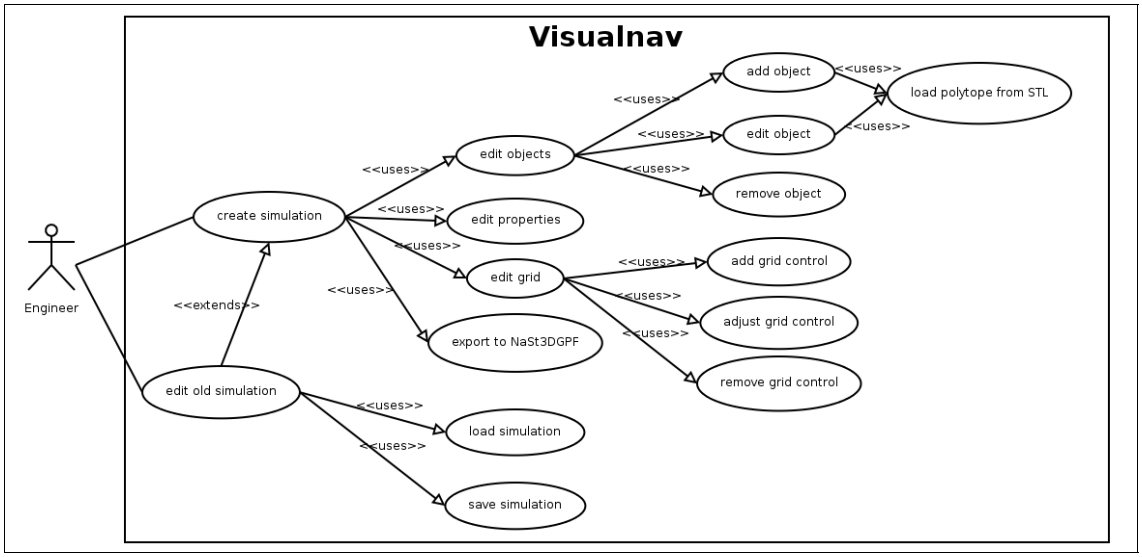

Illustration 1: Functional Requirement

Generally the application is used for two functions, to create a simulation *(create simulation)* and to edit an existing one *(edit old simulation)*. Whereas the second case is only a extension of the first with the added capability to store *(save simulation)* and retrieve *(load simulation)* the data for the editor.

To create a simulation the user needs to arrange objects (*edit objects*), define the grid (*edit grid*) and set the general simulation parameters (*edit parameters*). Since the parameters are unique they will be edited with the means of a dialog. After the simulation is complete it must be exported to the native format of *NaSt3DGPF*, the numeric simulation software *(export to NaST3DGPF)*.

Objects need to be added, edited and removed *(remove object)* from the definition. To add an object *(add object)* the user is prompted a dialog that displays the default attributed of an object and he may modify them before the object is added. The object is then only added if the dialog is successfully completed.

After selecting a object the user may edit that object *(edit object)*. To do so a dialog is raised that displays the attributs of an object. The changes the user made are committed to the definition only once the dialog is successfully completed.

Since a large number of geometry data comes from computer aided design tools it is necessary to import that geometry. The geometry is imported from the *STL* format *(load polytope from STL)* to be used as volume representation for objects.

The grid is defined by the means of grid controls. Grid controls are points along an axis that define the cell size at that position. For each axis it mus be possible to add *(add grid control)*, adjust *(adjust grid control)* and remove *(remove grid control)* single grid controls.

2.4 Software Environment

Since the targeted users use either *GNU/Linux* or *Windows*, the software must be written in a way that accommodates for these two operating systems. As a consequence it has to be ensured that the dependent libraries run on both systems. Because it is uncertain how often the software will be installed, free software is a simple solution to keep costs under control.

The graphical front end is implemented with the help of *GTK*. The *Gimp Tool Kit* (*GTK*) was developed for the image processor Gimp and has become the primary human

interface tool kit for the *Gnome Desktop*. It is published under the *GNU Lesser General Public License* and was ported from *GNU/Linux*[1] to *Windows*.

To efficiently store and retrieve the data the *XML* is used. To parse the data *libxml* is used. The library *libxml* is a simple implementation that is closely associated to *GTK*, witch guarantees proper interaction of the *GTK*.

The conversion of the data will be done with the help of *XSLT*. To convert the *XML* data to the *NaSt3DGPF's* native format within the application *libxslt* will be used.

All three libraries rely on *Glib*. It provides basic utilities such as UTF-8[2] strings, memory management or multi threading support. As a result *Glib* is extensively used and reduces the need for additional utility libraries.

Since the definition of a simulation should be presented graphically in real time, *openGL* is used to interact with common graphic hardware. On *Windows* systems *DirectX* could be used, but it is opted against since *openGL* is available for many operating systems, including *GNU/Linux* and *Windows* and the interface of *openGL* has proven to be stable over a long period of time. The gap between *openGL* and *GTK* is bridged with the help of *GtkGLExt.*

2.5 Hardware Environment

The software prenav will be deployed on average computer systems in the *Federal Waterways Engineering and Research Institute*. Because of its strong graphical nature a decent graphic card supporting *openGL* is required. These simple requirements are meet,

1 Actually the GTK was written for X11, a Unix standard graphical display mechanism that is also implemented for GNU/Linux.
2 UTF-8 is a Unicode standard that encodes multilingual characters in variable byte length.

since most computer systems in the *Federal Waterways Engineering and Research Institute* are graphical workstations.

2.6 Development Environment

The software is developed on a Siemens Celsius dual Intel Xeon 1.2 GHz computer, with a Nvidia Quatro4 graphic card. That computer system is a average system for the institute.

No integrated development environment (IDE) was used to develop the software, since experience has been proven that it is more efficient to utilize individual tools optimized for their task.

The *GCC C++ Compiler (g++)* was used to compile the code. The *GNU Compiler Collection* is one of the most conforming compilers with the *ISO C++ Standard* and is free software. The compiler runs and is able to compile code for both *GNU/Linux* and *Windows*.

To control the configuration and process of compilation the *GNU Autotools* where used. The three tools *autoconf*, *automake* and *libtool* are an efficient way to build multiple targets in one invocation. In addition it follows a common mechanic to build software on Unix and derivate systems. To use those tools under *Windows, cygwin* is used to provide a requirement for *autoconf*, a bin shell.

Source version control was performed with *Subversion*. It is seen as the follow up of *CVS* and is developed in parts by the same people. The repository is located on a network storage that is backed up daily. This implicitly solves recovery of sources in case of data loss. Though for performance considerations the development was done on local disk.

The quality of source code is ensured by an in house unit testing framework. The memory consistency is checked with the non intrusive memory profiler *valgrind*. In the seldom case that programs did not behave as expected *DDD* was used to debug the software. The *Data Display Debugger (DDD)* is a graphical front end for the console utility the *GNU debugger (GDB)*. A drawback of using *valgrind* and *DDD* is that both tools do not or only with restrictions operate on windows. But because no conditional compilation was necessary[3] this was not of a significant problem.

The documentation of the source code was done with the help of *doxygen*. It is a system similar to *JavaDoc* that integrate the interface documentation into the source code. *Doxygen* may create *HTML*, *man*, *PDF* or *XML* documents. Primarily *HTML* was used to view the documentation.

3 In some situations it is necessary to compile different bits of source code in order to adapt to the different interfaces of operating systems.

Chapter 3 Design Process

Although this project did not follow a strict process, it was necessary to apply certain methodologies to bring the project efficiently to its goal and to ensure that future development is possible. These two goals where important for the overall success of the project.

3.1 User Interview

Although the general direction for the project was clear, since previous projects lead to this one, a series of interviews with the future users was conducted.

The interviews where structured in the way to find out how the users would create a simulation and not how they thought the software should look. The inherent problem is that users tend to describe the features of their favorite software, thus preventing innovation. In this special case innovation is important, since only a hand full of software was developed for this problem and no common methodology has emerged that can be used.

As a result it became clear that the way entities are defined should not change since the user had already acquired profound knowledge of the simulation tool. An other point was that users wanted to work on the simulation in an arbitrary order, thus a more rigidly defined editor was out of question.

One user pointed out that it may be interesting to put the additional data required for the editor into comments within the *NaSt3DGPF's* native format. This approach would make the files that the editor outputs usable as if they where written for *NaSt3DGPF*. This approach was dropped since the *NAV* format is in its self not well structured and adding to

that would be difficult. In addition the *NAV* format is bound to change in the future thus potentially breaking any parser written for the current version.

3.2 Paper Prototyping

Paper prototyping is a rapid prototyping methodology in which the user interface is prototyped and so the final application. The interface is sketched on paper and is presented to a potential user. The prototype is then animated by a developer who simulates the application logic behind that software. The user then uses the software and thinks aloud his thoughts and what he tries to accomplish.

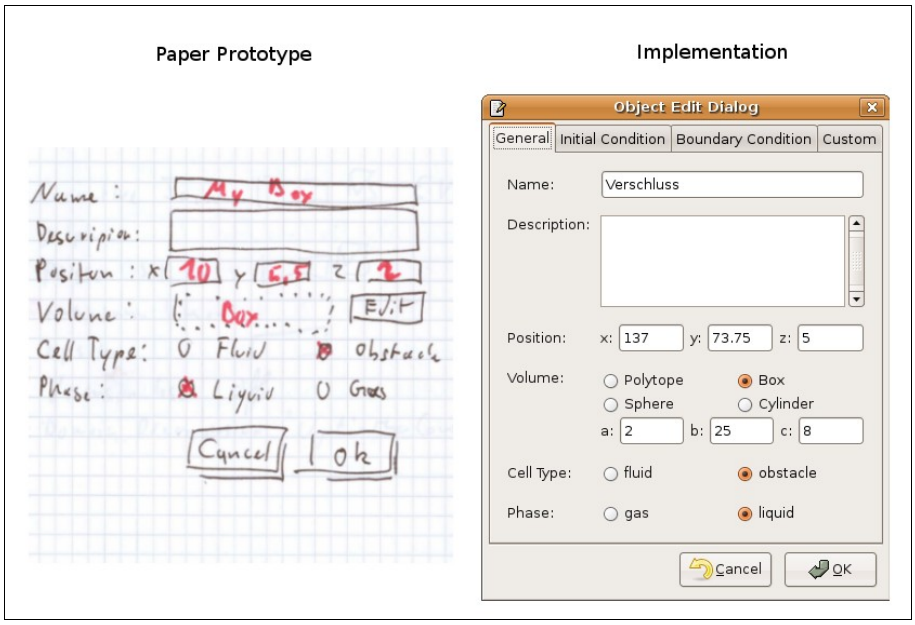

Illustration 2: Paper Prototype vs. Implementation

During the prototyping and execution process the prototype may be altered and so flaws in interface design can be discovered rapidly. As a result developers have a relative good idea of what the user wants and how the interface should look.

Paper prototyping originally came from usability testing, but can be used to refine functionality of software during design. Its strength is the turnover of information, since the prototype can be developed in a few hours and altered during the prototyping session. An implicit advantage is that project sponsors will not confuse a paper prototype with the finally product, which occurs often when using software prototypes.

In illustration 2, a except of a early prototype is displayed alongside of the concrete implementation. In this case it was planed to raise an other dialog to edit the volume of an object. This has proven to be cumbersome and was replaced by a simple solution. All widgets necessary to manipulate all types of volumes where integrate into the dialog and then hidden as they where not used. To further optimize the handling and prevent resizing of the dialog it was ensured that the widgets that manipulate one volume are all together the same size.

3.3 CRC Cards

Similar to paper prototyping, *CRC* cards is software design using paper and pencil. The main goal of this method is to design object oriented software and its focus is the internal design of classes and their interactions.

A *CRC* card is a simple stock card about the size of *A6*. On that card the essential information of a class, that is its name (class), its responsibility and collaborating classes are written in pencil on the card.

During the design phase a meeting with all the development team is held to design the software in a so called *CRC* session. Although it is recommended to work in a team, the design was done mostly by the only developer writing the software.

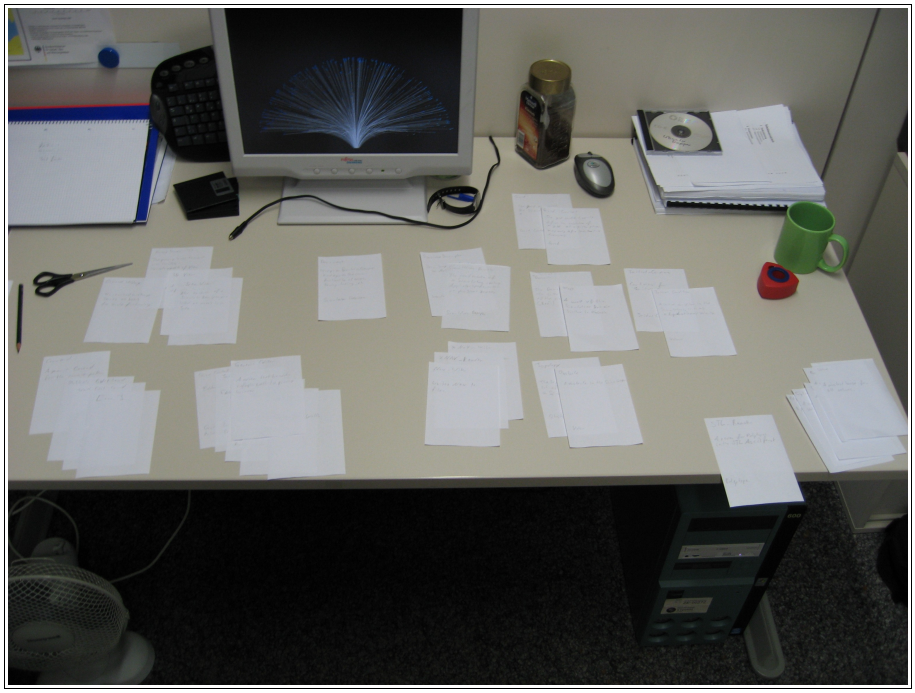

Illustration 3: CRC Design Session

During the *CRC* design phase it became clear that a simple design following the *Document–View* pattern was not sufficient but also that a a full M*odel–View– Controller* (*MVC*) pattern was to sophisticated. *MVC* would have required that the whole model would be warped and interaction with the *XML DOM*[4] prohibited.

3.4 Refactoring

Because it is impossible to capture the requirements of all software module at the beginning of a project a development schema was adopted that encouraged refactoring. The process behind refactoring is to change and improve the software when it is necessary to maintain a high level of quality. The strongest point behind refactoring is the ability to modify missinterpreted designs so that future development can proceed a swift pace.

4 The data storage method is described in more detail stating page 36.

3.5 Decoupling of Software Modules

To minimize the impact of changes the interfaces of classes where designed with a loose coupling. To achieve this goal two principles where used, the *Law of Demeter* and *Event Orientation*.

The *Law of Demeter* is a paradigm that dictates that only those objects associated with the current context may be accessed. The result is that the source code of classes, even single methods, depend on fewer classes and as a result change of one class impact only few. A drawback of the *Law of Demeter* is that it creates wide interfaces with some methods only calling methods on objects in there context.

Event Orientation is a paradigm that event further reduces the coupling of classes and may be used in conjunction with the *Law of Demeter*. The general idea is that a object exposes different events, also known as signals, that other objects can subscribe to by attaching a event handler, in the signal terminology a slot. The first object has no need to know about who is monitoring it and simply has to trigger a event once the condition occurs.

In this case the *libsigc++* signal and slot library was used. It is primarily used, since it is a dependency by the human interface took kit *GTK* and duplicating functionality would have been unwise.

3.6 Design by Contract

Although everyone learning how to program learns that algorithms have pre- and postconditions but this quickly forgotten and are not enforced once most start to program. By specifying requirement for algorithms and their result it becomes easier to detect errors and external developers know what are valid values for a given algorithm.

Just specifying what is valid for a algorithm is useful but it can significantly improve if those requirements are enforced in the source code. *C++* does not provide a mechanism for enforcing requirements such as the language *Eiffel* dose. Originally it was planed to use *nana* an improved assertion and logging mechanism for *C* and *C++* which provides a mimic of the *Eiffel* semantic. Two macros are defined, *REQUIRE* and *ENSURE* for pre- and postcondition check. The two macros insert code to check a given condition and interrupt the the application when that condition is not met. For performance reasons they may be disabled during compilation.

Since *nana* had some significant implementation flaws for a *C++* project and provided a relatively large amount of unnecessary functionality the two macros *ENSURE* and *REQUIRE* where implemented for this project. In addition the two macros *A* and *E* which represent the two quantifiers *all* and *exists* where implemented after the example of *nana*.

This approach significantly reduced the need for trivial unit tests. It was not necessary to check all variable access and most simple algorithms since that was done within the *ENSURE* macro. Both macros where also used as indicator for pre- and postcondition that where then extracted by the *API* documentation tool.

Chapter 4 The Data Model

The data model used for *visualnav* is derived from the *NaSt3DGPF's* native format *NAV*. Although it was extended to accommodate the extended functionality of the editor and to overcome some shortcomings and ambiguities in the format.

4.1 Storage in XML

Early in the development it was clear that the new format was to be stored in *XML*. The extensible markup language (*XML*) is a *W3C*[5] recommendation and a base for many web based technologies, such as *HTML*, *RSS* or *WebServices*. It is a derivate of *SGML* which was standardized in the early 80's.

The prime reason for using *XML* is that it is a standardized format with a number of implemented parsers. The abundance of parser is an advantage since the development of a parser is time consuming and a source of many errors. In addition it is strongly formated with no ambiguities, which can not be said about the *NAV* format.

4.2 The Simulation and Basic Structure

At the basis of the data model is the simulation structure. It encompasses all basic entities, as illustration 4 shows. In general it reflects the entities as described on page 12.

The simulation has two descriptive elements, that is its name and description. Both are used as information for the user, so that the simulation as such may be described. This information will be put into the output *NAV* file as comment.

5 The *WWW Consortium* (*W3C*) is a consortium that is the driving power in the standardization work of many web technologies such as *HTML*.

The simulation entity also has the two values, the size of the simulation domain and the time the simulation will run in simulation time. There is to note that NaSt3DGPF defines the simulation domain as a box starting at the origin and going to an arbitrary point in space, in the first octant.

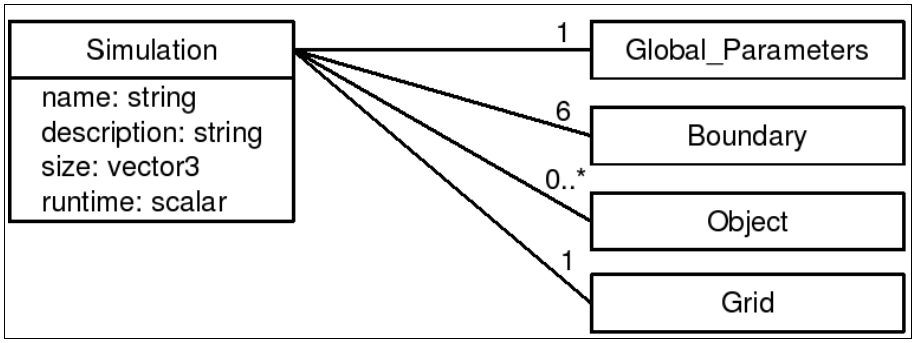

Illustration 4: Data Model: Simulation

In illustration 5 it is also visible the other parts of a simulation. That are the global parameters that are settings for the simulation core, the six boundaries, all objects and the simulation grid on which the simulation will be discretized.

4.3 Boundaries

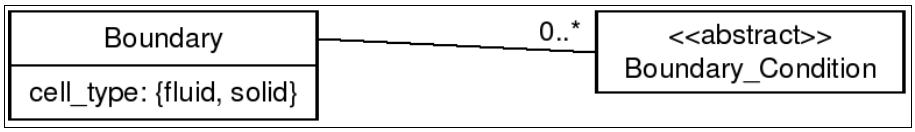

Illustration 5: Data Model: Boundary

There are six boundaries on the simulation domain. They are named *north, east, south, west, top* and *bottom* and are located as the table 1 describes. In general boundaries may either be *solid* or *fluid*. The behavior of boundary conditions are modified by boundary

conditions, that are described in more detail below. There is to note that is the boundary if flagged to be fluid a boundary condition regarding flow should be set.

North	$x = x_{domain}$
South	$x = 0$
East	$y = y_{domain}$
West	$y = 0$
Top	$z = z_{domain}$
Bottom	$z = 0$

Table 1: Position of Boundaries

In *NaSt3DGPF* boundaries are actually objects with a special flag, that is they are boxes with one of the keywords *north, east, south, west, top* and *bottom.* The distinction between the two was made so that the difference is clear and to prevent ambiguities.

4.4 Objects

Every entity that is defined by a volume is an object in *NaSt3DGPF*, this analogy was also used in *visualnav*. Illustration 6 portrays the data structure for objects. Similar with the simulation structure, objects have the two descriptive elements name and description.

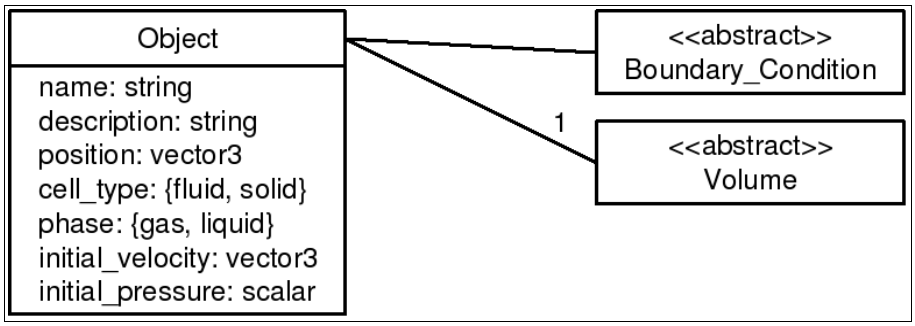

Illustration 6: Data Model: Object

Objects may be *fluid* or *solid.* If they are solid they are obstacles in the simulation and if they are fluid they may either be an initial conditions or a boundary condition. The initial values are set with the entities *phase*, *initial velocity* and *initial pressure*. They are optional but may be used to initialize the simulation.

4.5 Boundary Conditions

There are currently three different boundary conditions defined in *visualnav* as shown in illustration 7. That is the Roughness that defines the surface roughness of any objects or boundary. Roughness comes in three flavor, *slip*, *k-value* and *noslip*. In the case of slip the velocity at the surface is that of the flow and in the case of *noslip* the velocity is zero. The roughness value was shortly introduced to *NaSt3DGPF* and defines a surface with spheres approximately the size of *k*. The value of *k* is only sensible with the *k-value* condition.

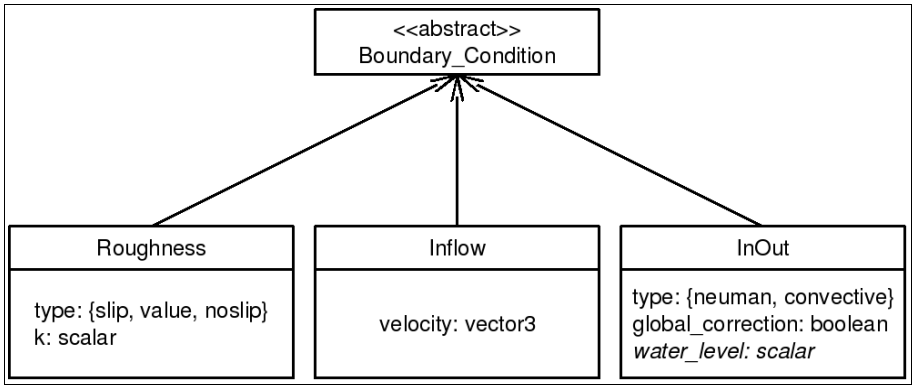

Illustration 7: Data Model: Boundary Conditions

The *inflow* condition defines a steady inflow into the system, defined by the velocity. The actual amount is defined over the size of the volume. The other condition regarding flow is the *inout* condition. With that condition set, the boundary will react as outflow, though also as inflow in certain conditions. Boundaries may have an additional parameter that

defines the water level. The fixed outflow condition will hold the water at a certain level. This condition is inspired by the way laboratory models regulate the outflow.

NaSt3DGPF defines two more conditions, *chenminit* and *temperature*. Those two conditions are currently not implemented since they have no use at the *Federal Waterways Engineering and Research Institute.*

4.6 Volumes

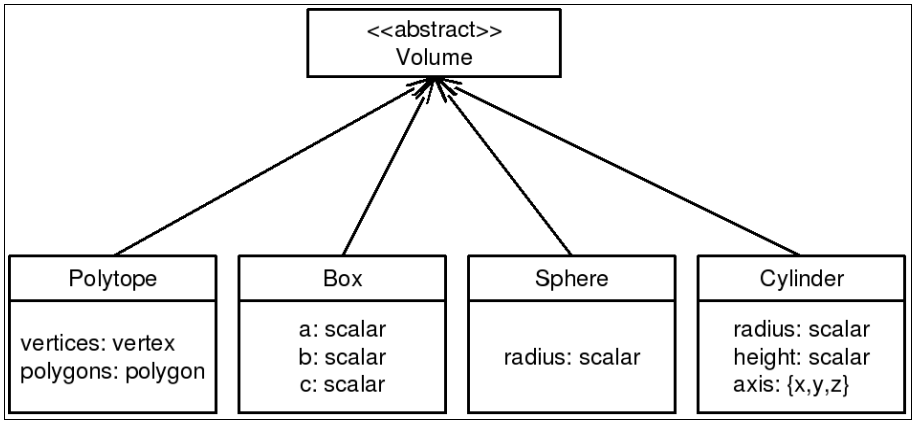

Illustration 8: Data Model: Volumes

All objects have a volume that describes what area of the simulation they affect. *NaSt3DGPF* defines four volume types. As can be seen in illustration 8, the four volume types are an axis aligned box, a sphere, an axis aligned cylinder and a polytope.

The definition of volumes was simplified and cleaned up from the way *NaSt3dGPF* defines the volumes box, sphere and cylinder. In *NaSt3DGPF* they are defined by the mans of minimum and maximum coordinates. This may be easily applicable to boxes but in the case of spheres and cylinder it becomes vague about the behavior when values indicate an ellipsoid rather than a sphere.

NaSt3DGPF also defines the two *constructive solid geometry (CSG)* operators *union* and *intersection*. Because they are rarely or never used they where not implemented.

4.7 The Grid

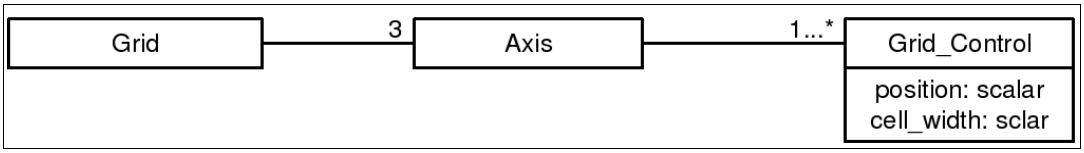

Illustration 9: Data Model: Grid

The grid in *NaSt3DGPF* is a rectilinear grid that may be scaled along the three axis.

4.7.1 Scaled Grid

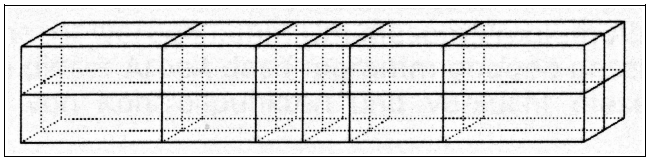

Illustration 10: Scaled Grid

NaSt3DGPF provides two ways to define the grid, either with a constant cell width or by specifying every cell position. In previous works tools where developed that provided the means to define a scaled grid with the help of control points. The cell width is then linearly interpolated between two control points. Illustration 10 shows a simplified scaled grid, that is scaled along the horizontal axis.

4.7.2 Grid Controls

As stated above the grid is defined with the means of grid control points or short grid controls. In *visualnav* the ability to define a constant scaled grid was dropped. The same

result can be achieved by only putting one grid control on that axis and as can be seen in illustration 9 there must be at least one grid control on every axis.

A grid control is simply defined by its position on the axis and the cell width at that position, as illustration 9 clearly shows.

4.8 Global Parameters

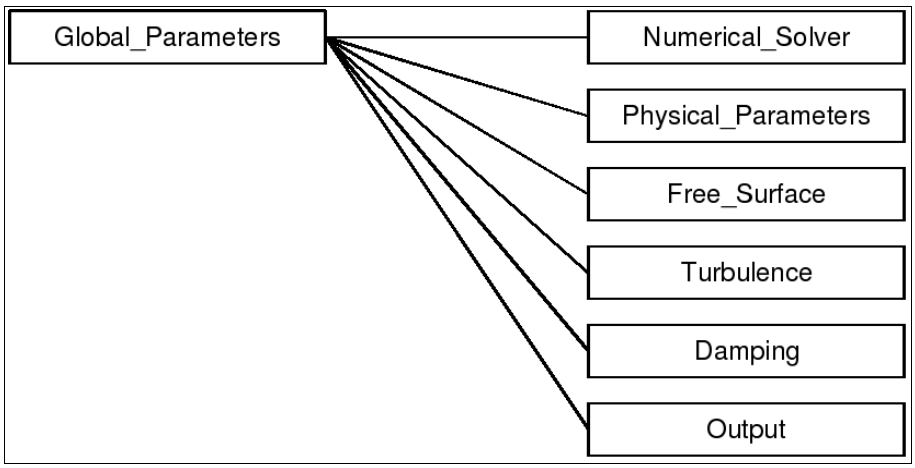

Illustration 11: Data Model: Global Parameters

The simulation parameters are settings for the simulator. In *NaSt3DGPF's* native format they are structured in a single block. As a result the block is fairly large and without the help of self imposed structure and comments this can easily lead to confusion. As can be seen in illustration 11, additional structure was introduced. The parameters where grouped together by there meaning.

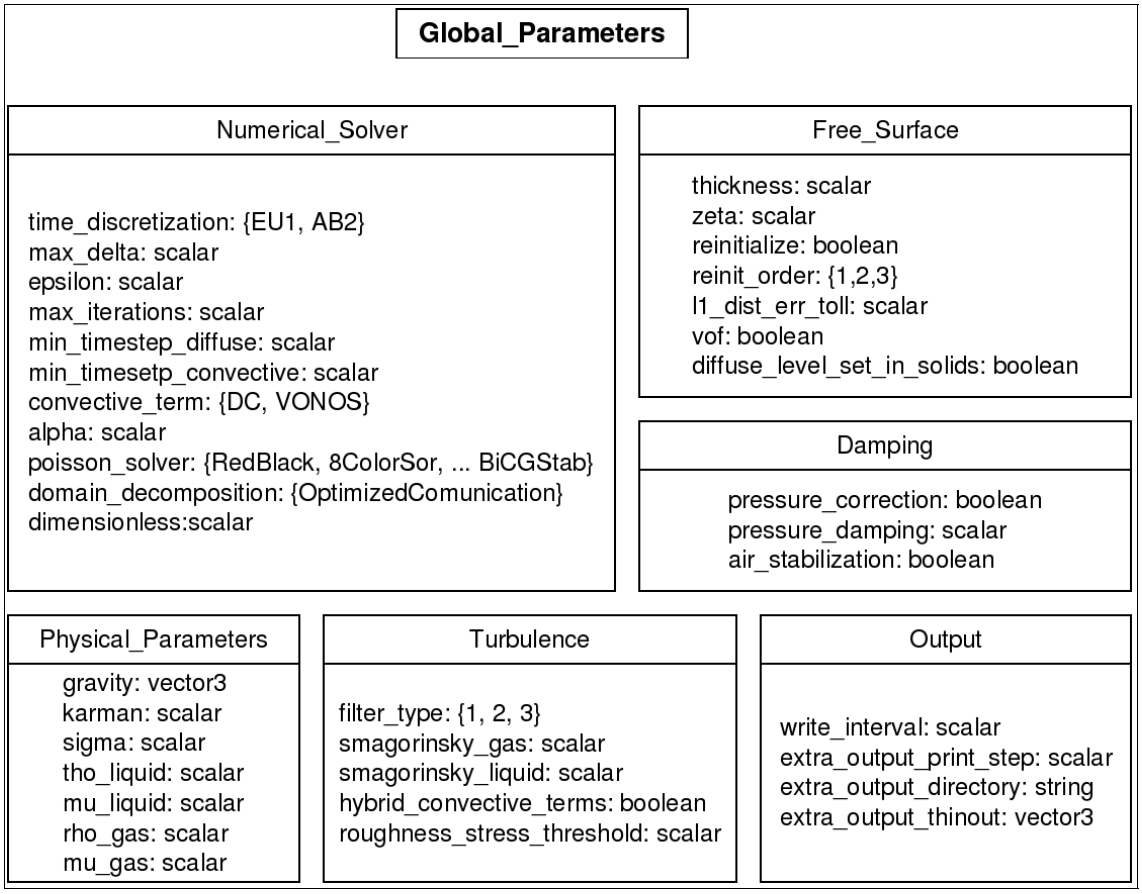

Illustration 12: Global Parameters

Chapter 5 The Editor

5.1 Application Structure

Illustration 13 shows the primary visual blocks of the editor. Common for all editors and most vial tools are the main menu (magenta) and the tool bar (blue). They are visual representations of the commands that the user may use to manipulate the data.

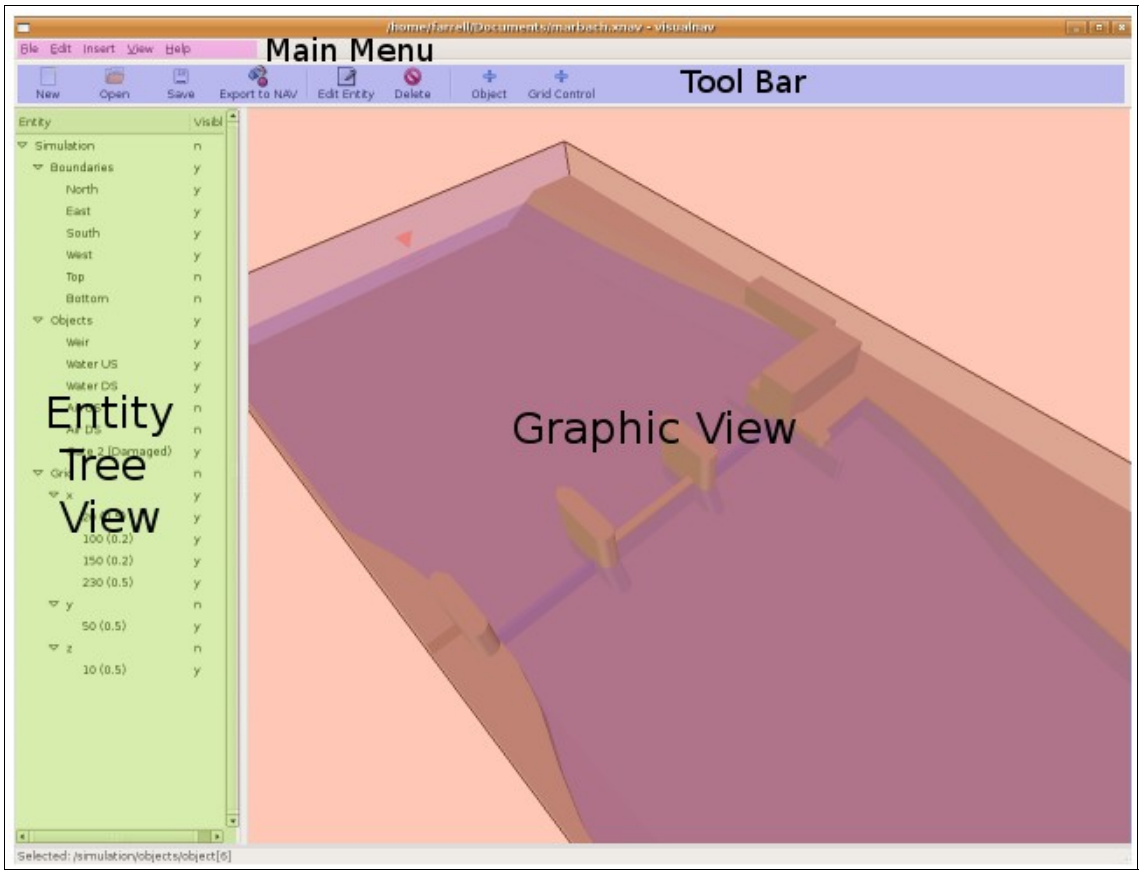

Illustration 13: Elements of the Editor

The two other blocks are the view, visual representations of the data. They provide the means to select and modify the data, although most modification in visualnav is done with the help of dialogs.

As can be seen in illustration 14, the application structure follows loosely the *Model-View-Controller (MVC)* pattern. The responsibility for data storage, visual representation and application control are split up. The Editor acts as controller the document as data storage and the views as visual representation.

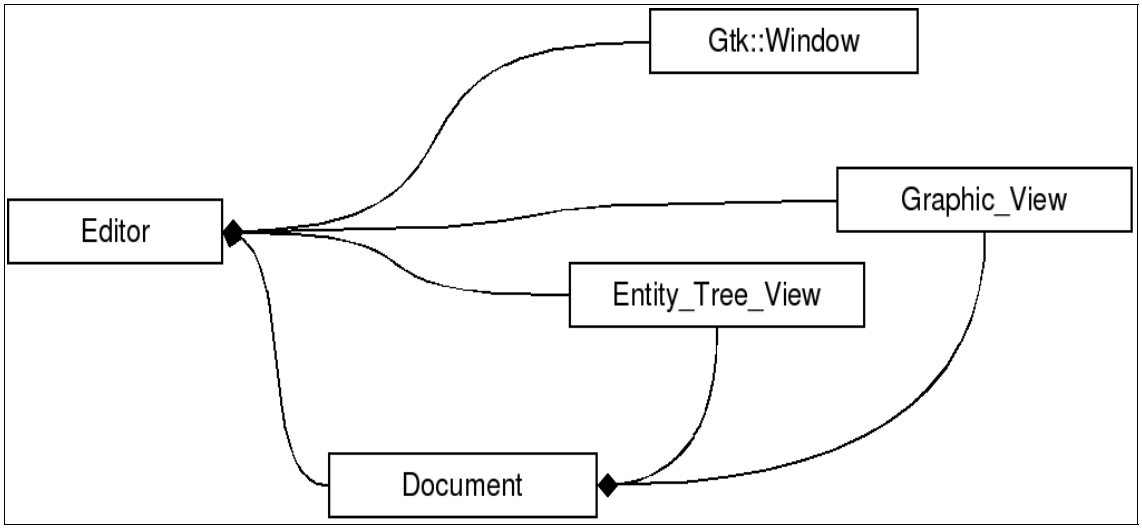

Illustration 14: Application Structure

There is to note that the implicit relation between the primary window and the views was omitted in illustration 14. Same as the relation built up by the call back mechanic of the signal and slots framework between the document and the views.

5.2 The Editor Class

The Editor, though not clearly labeled as such, is the controller. It houses the document, the primary window, all views and logic that drives the application. Because the *libxml's* document class dose not provide the means to signal on change and it was not warped, it is the editors task to notify all views that the data has changed.

After any modification to the data was made the method *notify_change* was called on the document class, thus informing any view that lessened to the change signal to update itself. The document class is described in more detail below.

5.2.1 Adding Entities

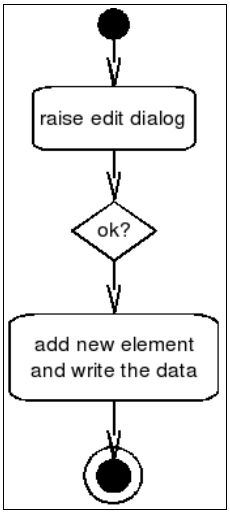

Illustration 15:
Adding Element

Entities are all added after the same schema. Every entity has its *Insert* entry in the main menu and tool bar and so it's own method. Illustration 15 displays the process by which an entity is inserted. First the corresponding dialog for that entity is raised and the user may insert the initial data for the entity. If the dialog was exited with the *OK* button, then a new entity is added to the definition and the data of the dialog is written into it.

Dialogs in general and specific dialogs for certain entities are described below in more detail.

5.2.2 Editing Entities

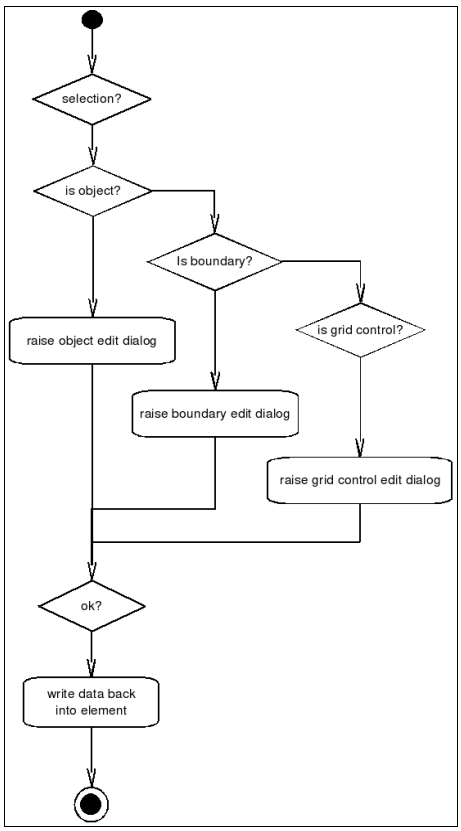

Illustration 16: Editing Entities

Similar to adding entities, the may be edited and there is a corresponding menu and tool bar entry. While editing entities the application flow passes by the same method for each entity and so the first step is to determine what entity is currently selected. as illustration 16 shows. After that the process is very similar to adding entities, the corresponding dialog showing the settings is raised. If the dialog was exited with the *OK* button the data is written back into it. Since the edit button may be hit at any time, even when there is no selection, the method is of course protected as illustration 16 shows.

5.2.3 Removing Entities

In the case of removing entities it is only check whether the user is allowed to remove them or not and then they are removed form the document.

5.3 Data Storage

5.3.1 Custom Model from libnav

In previews projects the library *libnav* was developed. It contained a basic data model that represented polytopes and a grid definition based on control points. It was used for example by the grid generator *visualgrid*.

This seemed to be a feasible solution at first. But the shortcomings where that there was no means implemented to store and load the data and for every extension to the model a good amount of time had to be invested.

This approach was dropped in favor of using the *extensible markup language document object model (XML DOM)* directly to store the data within the application. That approach was relatively straight forward since the data was to be stored in *XML* anyway.

5.3.2 XML DOM

Finally *libxml's document object model (DOM)* representation of *XML* was used. By doing so no format conversion was necessary and it is relatively efficient to load and to store.

An additional feature was that *libxml* preserved comments and visualnav was implemented to be fairly tolerant. As a result comments and unknown elements where

preserved within the editor. This feature keeps the possibility open for extensions within the format.

5.3.3 Extending the DOM

Using the *document object model* as sole model was not sufficient since for one it did not provide any means for change notification nor did it integrate the possibility to set a selection.

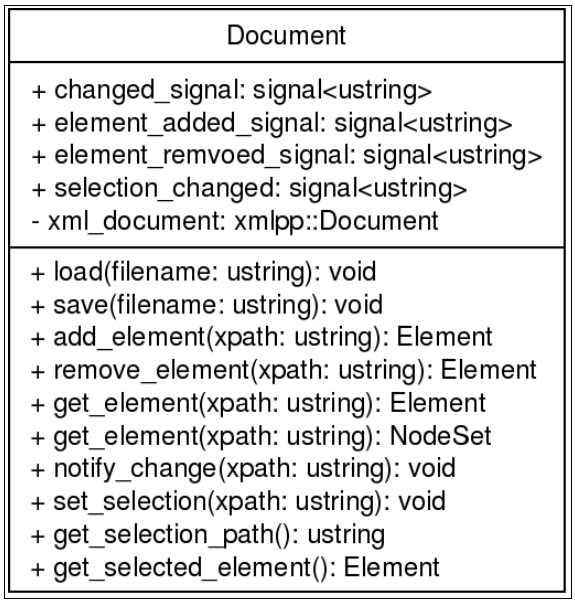

Illustration 17: The Document Class

The document class, as shown in illustration 17, is a wrapper around *libxml*'s *DOM* document class. The methods *load*, *save*, *add_element*, *remove_element*, *get_element*, *get_elements* are all actually just call through methods. The only extension is that they emit the corresponding signals, so that any view or other piece of code can update itself on that change.

A simple selection mechanism was added so that the selection is handled by a central entity. A view that has a selection will then call the *set_selection* method and by doing so implicitly notify any other views of the change. The selection or the selected element can than be retrieved without knowing of who or what set the selection. This is often used in the editor class.

5.3.4 Exporting to NaSt3DGP's Native Format

Since the data was stored in *XML* a *XML simple language transformation (XSLT)* style sheet was used to transform the data to *NaSt3DGP*'s native format. The imminent advantage to this approach is that it is possible to convert the data without the help of *visualnav*, like remote compute servers who do not have the software installed.

5.3.4.1 Generating the Grid

XSLT has a shortcoming, that it can only transform data from *XML* to an other format but not generate implicit data. In the case of *visualnav* where the grid is defined over grid control points, cell positions must be computed.

Since also other components of *visualnav*, mostly the graphical visualization, needed the grid data the calculation was centralized in one object. The *Grid_Generator* was created to generate the grid based on the grid controls. For each axis it reads the grid controls form the *XML* document and writes the the grid cells back into the document. This procedure is embedded within the refresh of the views.

There are two drawbacks to this approach. First the data is redundant within the file which has the consequence that it uses more disk space. The second is that if the grid controls

are modified by hand or an other utility the grid definition is not updated and so any export will not reflect the changes made to the grid.

5.3.4.2 Validating the XML

As recommended by the W3 Consortium, the developers of *XML*, a *XML* schema was developed for the format to validate the documents. This feature can be switched on or off when loading documents in *visualnav*. At the time of writing the schema was not fully complete, but will soon follow.

5.4 Views

5.4.1 Interface

All views inherit from the same super class, that is *View* as illustration 18 demonstrates and is rather a hint of intent than a necessity. Views all have the method refresh by which it may be forced to regenerate itself. In addition they hold a reference to the document.

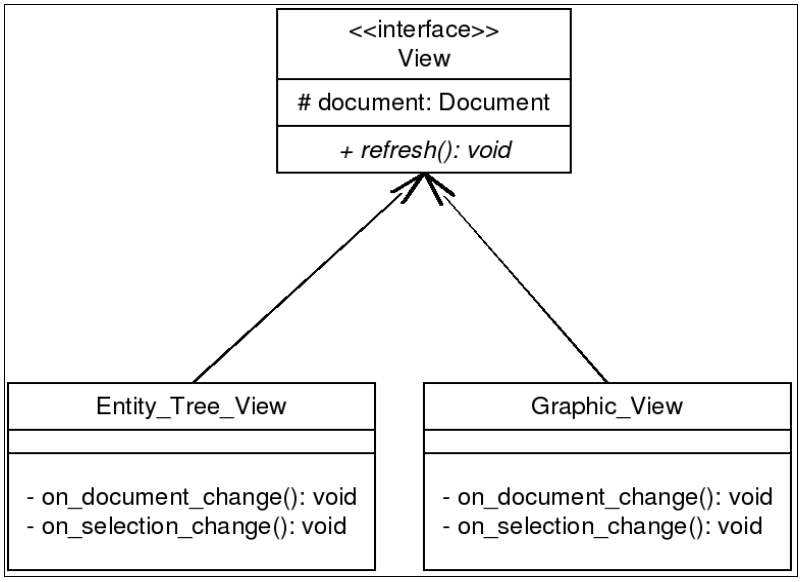

Illustration 18: Views

5.4.2 Notification of Change

All views register themselves at the document to receive at least the two signals, changed and selection_changed from the document. The signals are than handled by the two methods *on_document_changed* and *on_selection_changed*

5.4.3 The Entity Tree View

Illustration 19:
Entity Tree View

The entity tree view is a representation of all entities in the simulation in the form of a tree. To accomplish this a *TreeView* widget is used and the document is read out.

The tree is organized in the way that first all boundaries are displayed, than all objects and then the grid controls grouped by their axis, as can be seen in illustration 19.

5.4.4 The Graphic View

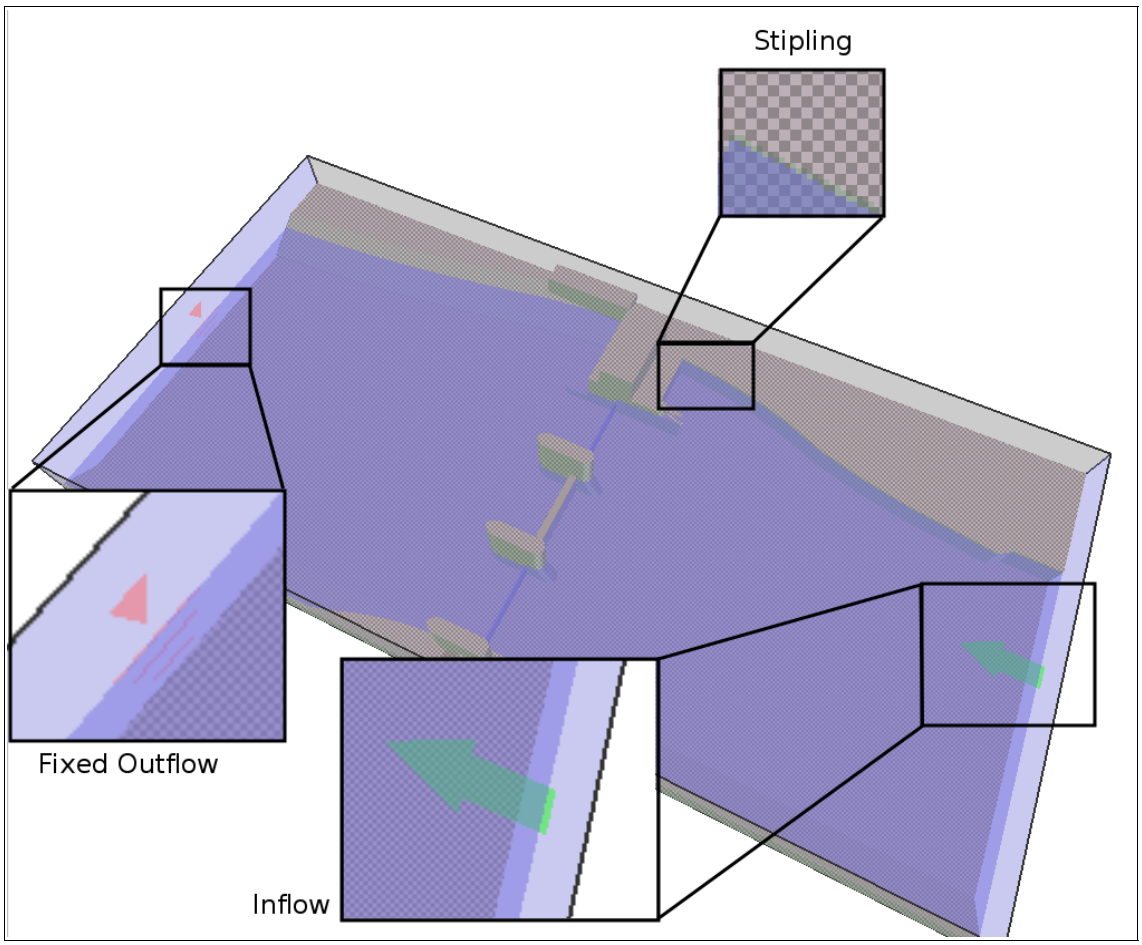

Illustration 20: Graphic View

The graphic view is the most noticeable view of the editor. It displays the simulation in a three dimensional rendered view. As can be seen in illustration 20 the solid objects are drawn in gray, the fluid and liquid are drawn in transparent blue and the fluid and gas are drawn in transparent yellow.

The surface roughness is indicated on solids by the color and the drawing method. Is the solid's surface defined as slip then it is drawn in light gray, is it noslip than in dark gray. If

a *k-value* is set then the surface is stippled in light and dark gray, that is drawn with every second pixed dark and light.

The boundary conditions *inflow* and *fixed-outflow* are also visually represented, as shown in illustration 20. The inflow boundary has a green arrow pointing in the direction of the flow. And the outflow boundary has a water level symbol drawn at the hight at which the water level is held, in red. Those two visual queues are essential to capture the essence of a simulation.

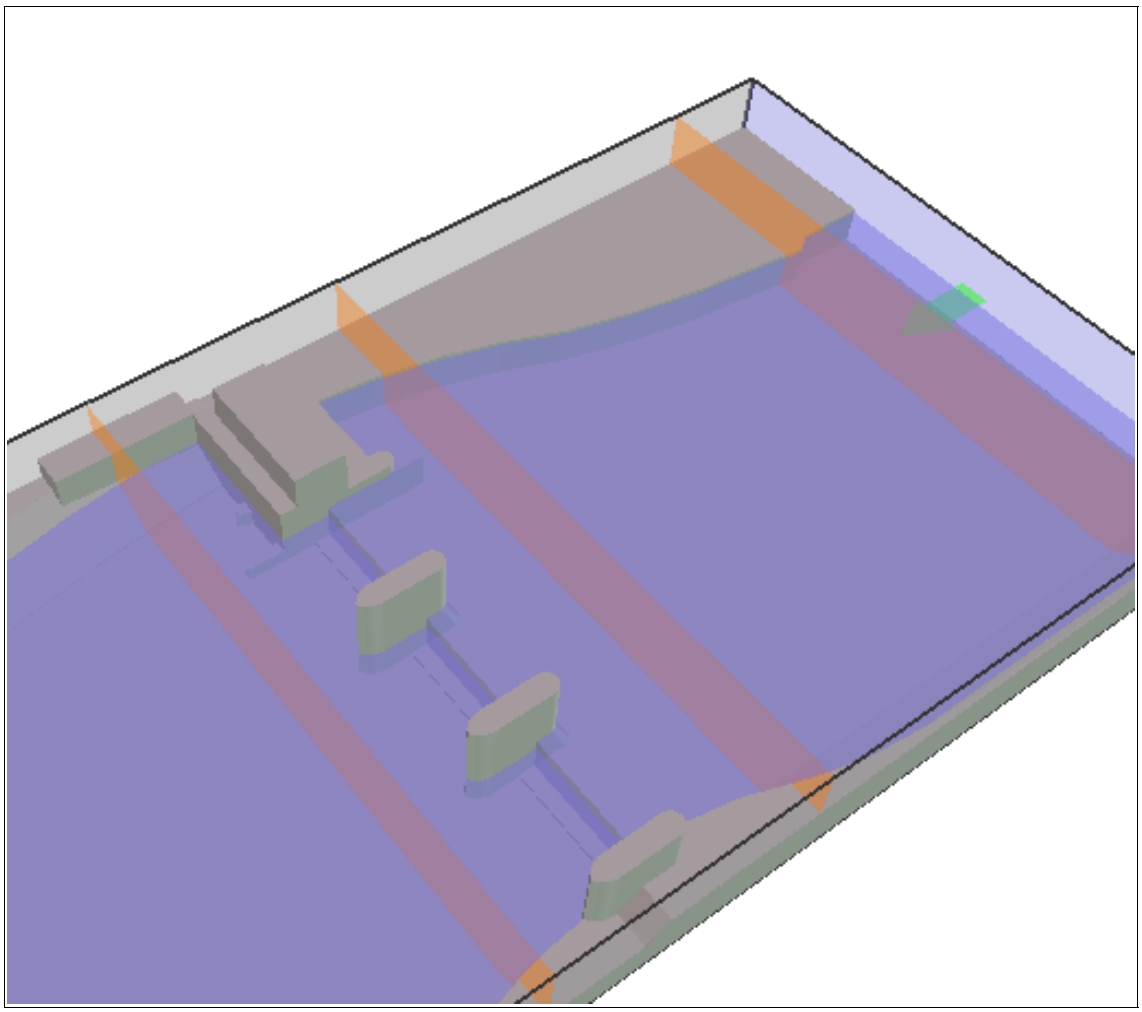

Illustration 21: Grid Controls

As illustration 21 shows, the grid controls are displayed as orange planes. This information is very helpful for quick feedback about the positions of those points.

5.4.4.1 Rendering Optimization

Because reading data from the model directly is relatively time consuming the rendering process, or rather the data acquisition had to be optimized. To do this the class visual was introduced as shown in illustration 22. The main optimization comes from the fact that the data for one visual is read one and modified only when the data has changed.

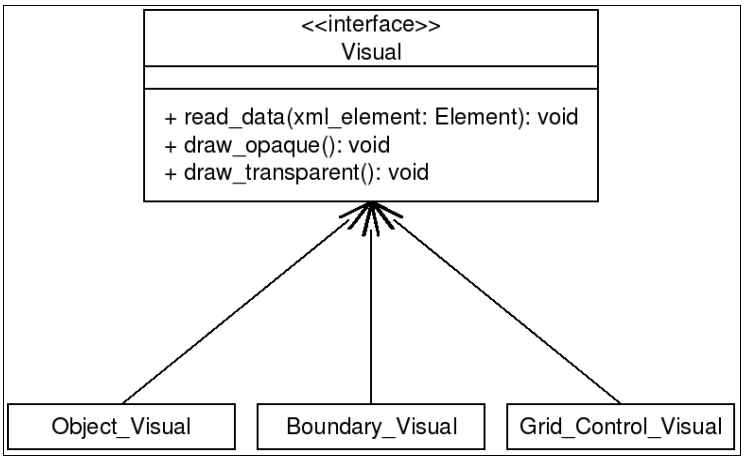

Illustration 22: Visual

5.4.4.2 Managing Transparency

To properly render transparent and opaque objects, the opaque objects must be rendered first and then the transparent. This is handled in the way that first the method *draw_opaque* on all visuals is called, there they may render opaque objects and then *draw_transparent* is called. This is a simple solution to ensure that opaque and transparent objects are properly rendered.

5.5 Dialogs

5.5.1 Interface

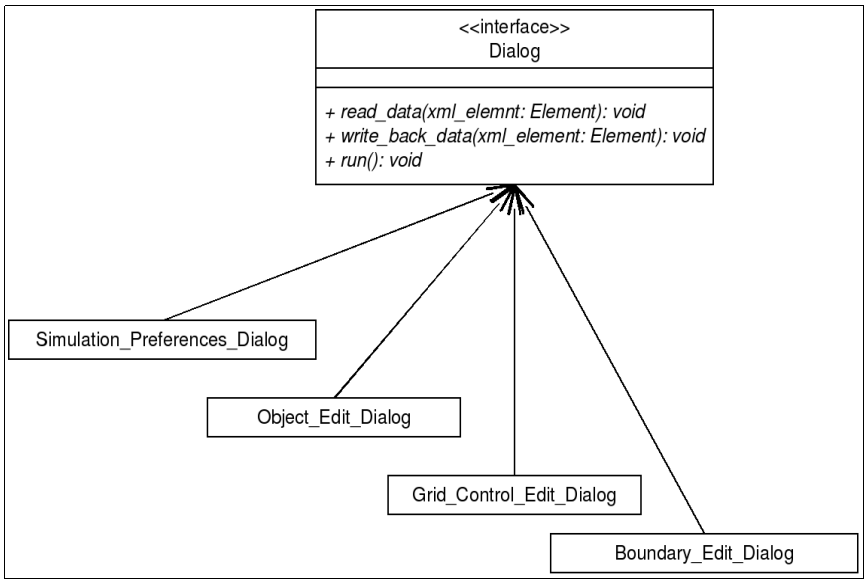

Illustration 23: Dialogs

All dialogs share the same interface. This was possible because of the use of *XML*, since all data shared the same representation. As displayed in illustration 23, dialogs can either be constructed either with empty data or with the data from a *XML* node. This was a compromise so that adding entities and editing them would not require different code.

Data can be loaded in two ways, either with the constructor or with the *read_data* method. It can be written back into a *XML* node by the means of the method *write_back_data*. The interface for writing data to the *XML* node is motivated by the fact that libxml prevents the use of *XML* nodes outside of a document, so that the node must be created within a document and then written by the dialog.

Finally the *run* method, which is a simple call through method, will run *GTK's* dialog in a separate main loop by calling the *run* method of *GTK*'s dialog.

5.5.2 Design With Glade

The Dialogs where designed with the help of the graphical interface designer *glade* and loaded at runtime with the help of *libglade*. This procedure did cut down the development time significantly.

To maintain a clean interface and implement custom behavior the dynamically loaded widgets, including the dialog, where wrapped in a separate class. By doing so the loading of the dialog and memory management could be tugged away into the wrapper and bound to only one location in the code, thus preventing duplication.

5.5.3 The Grid Control Edit Dialog

Illustration 24: The Grid Control Edit Dialog

The grid control edit dialog is the the dialog used to edit grid controls. It shows the two properties, position and cell width. In addition it shows on which axis the grid control lies, as can be seen in illustration 24.

5.5.4 The Simulation Preferences Dialog

As illustrations 25 and 26 show, the simulation preferences dialog provide the means to edit all global parameters for the simulation. As described on page 12 and following the parameters are grouped together into meaningful groups, this is done here with the help of tabs.

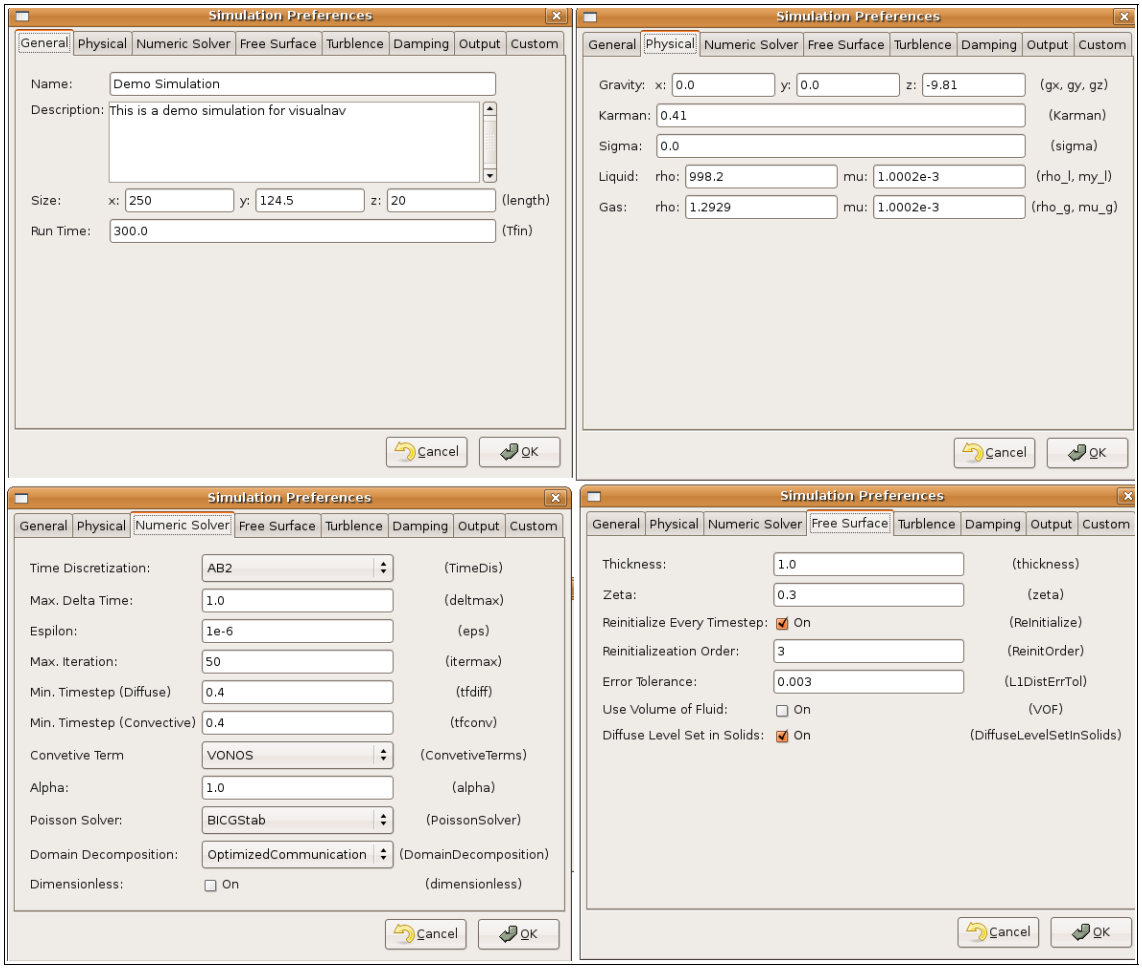

Illustration 25: The Simulation Preferences Dialog

For the user acquainted with *NaSt3DGPF* the parameters on which the values map are given on the right side as can be seen in the two illustrations. This feature is needed to ease the transition from one tool to the other.

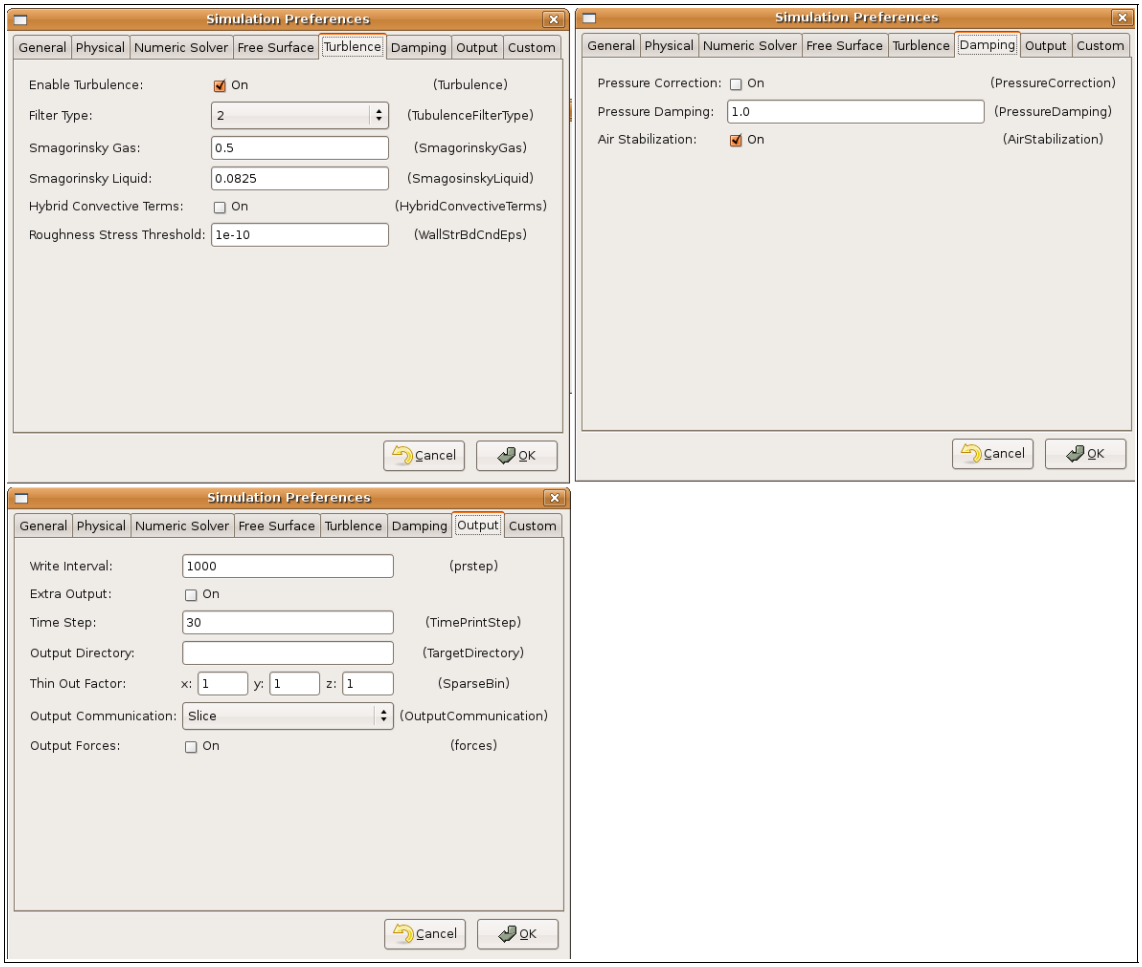

Illustration 26: The Simulation Preferences Dialog

5.5.5 The Object Edit Dialog

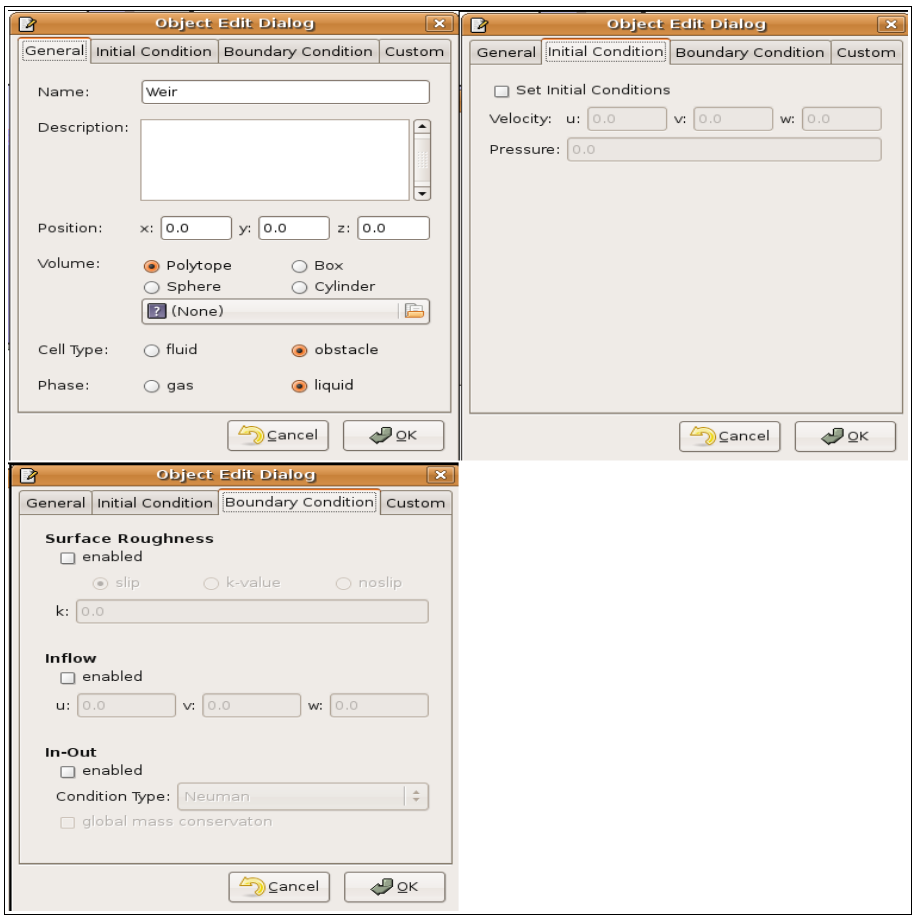

Illustration 27: The Object Edit Dialog

The object edit dialog provide the means to edit objects. As illustration 27 shows the dialog is structured in three tabs. The first tab, titled *General,* contains all the settings that must be set for an object and are commonly modified. The second tat holds the *Initial Conditions* and the third the *Boundary Conditions*. To guide the user those sections that currently make no sense to set in the current context are desensitized, that is they are "grayed out".

5.5.6 The Boundary Edit Dialog

The boundary edit dialog is similar to the boundary condition tab of the object edit dialog described above, though the boundary type and the fixed outflow was added. Same with the object edit dialog, those sections that currently make no sense to set in the current context are desensitized.

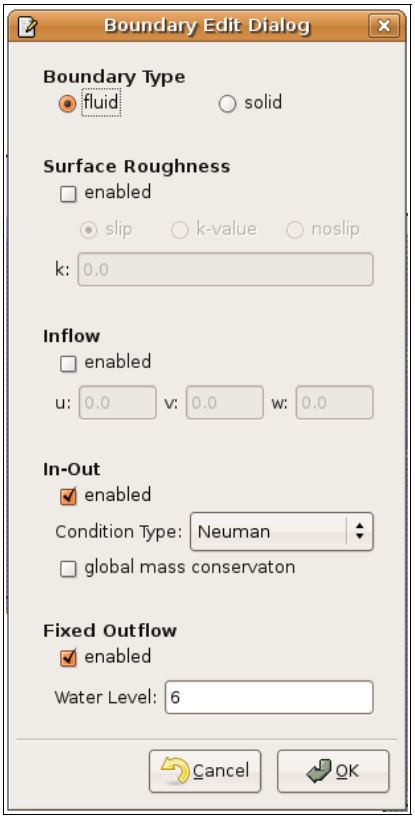

Illustration 28: The Boundary Edit Dialog

Chapter 6 Proof of Concept

In this chapter it will be shown that the editor can be used to create a real world example. The example portrayed here is based on recentinvestigation.

6.1 The Problem

A large number of waterways in Germany are rivers with a lock and dam system. In case of a flooding the weirs open their gates to let as much watter pass through as possible. The focus of the investigation is to determine the effect on the throughput in case one gate fails to open. This example is the investigation of the weir near Marbach, on the river Neckar, in the south of germany.

6.2 Simulation Domain and the Weir Geometry

After preliminary examination it has become clear that it is sufficient to model a 250 meter stretch of the river with the weir located at the middle.

The geometry of the weir and the river bed was prepared by the technical draftsperson and exported to the exchange format *STL*. In this case the river bed and the weir are exported as one geometry.

The two geometries of the weir and river bed where mended together so further processing with the existing methods is easier. For this editor separate geometries would have been of advantage, but the manpower to create them would have been out of proportion for this example.

The editor is opened with a empty definition as can be seen in illustration 29.

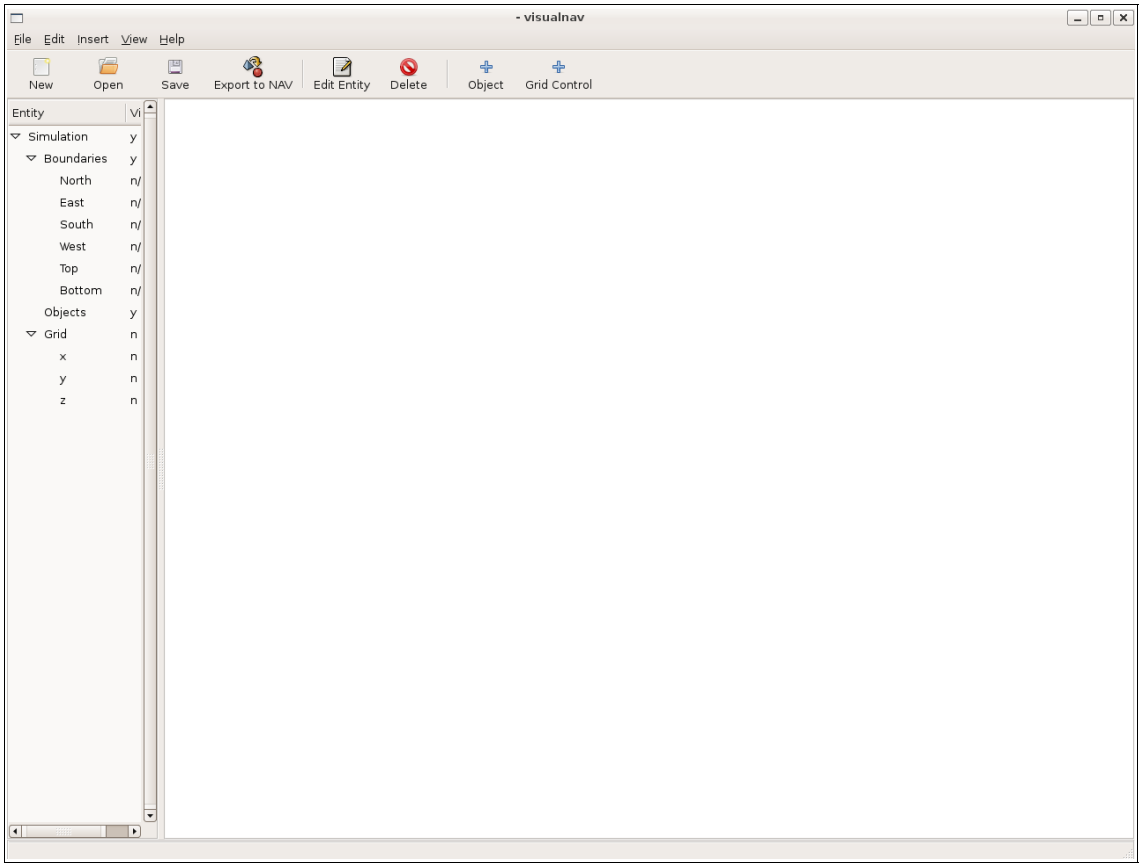

Illustration 29: Empty Editor

The first step is to define the simulation domain, that is the size of the simulation. This is done by selecting *Edit* and then *Simulation Preferences* in the main menu. As a result the *Simulation Preferences Dialog* is displayed, as can be seen in illustration 30. This dialog provides the means to edit all global parameters for the simulation, such as for example the physical parameters or the parameters for the numerical solver. The *Simulation Preferences Dialog* is described in more detail on page 46.

Here a meaningful name and description is given the simulation. The name and description actually have no real meaning and are solely there so the simulation can be documented along its definition. In addition the size of the simulation domain and runtime is set. As can be seen in illustration 30 the simulation domain is 250 meters long, 124.5 meters wide and 20 meters height and the simulation will run 300 seconds real time. The additional settings can be left by there default values.

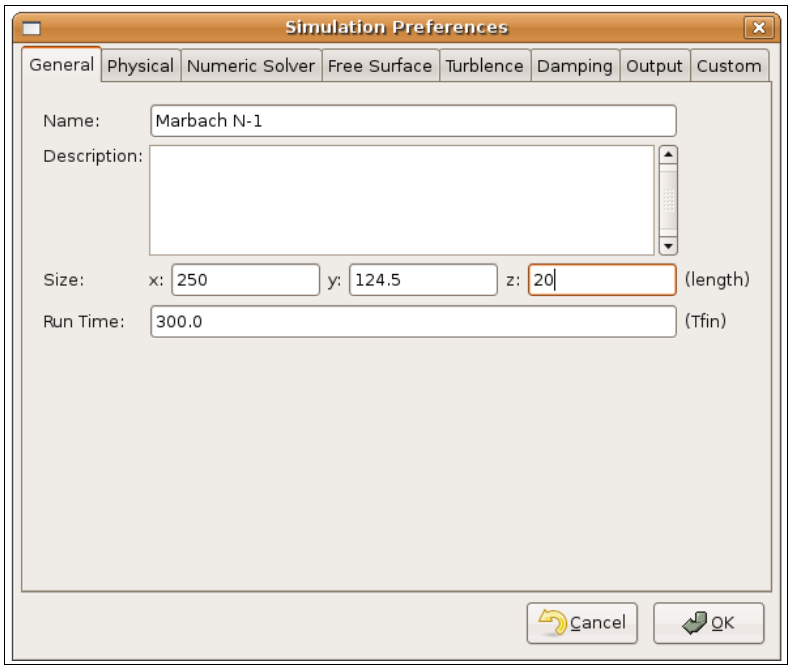

Illustration 30: Simulation Properties, General Pane

The editor now displays the empty simulation domain, as in illustration 31.

The next step is to load the geometry of the weir and river bed. This is done by selecting *Insert* and *Object* in the main menu and as a result the *Object Edit Dialog* is raised. The *Object Edit Dialog* may be used to edit all properties of object and is described in more detail on page 48.

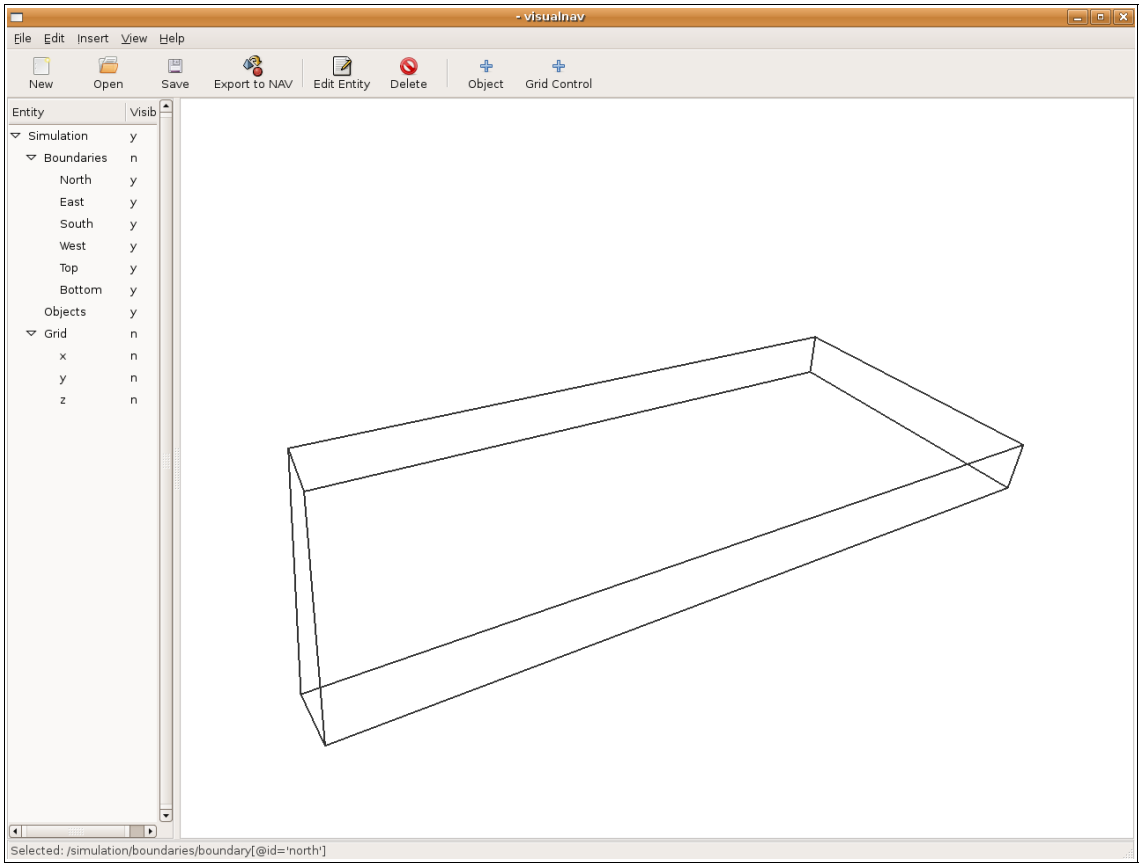

Illustration 31: Empty Simulation Domain

As with the simulation the weir gets a sensible name and description. Here only the name is set so the weir can easily be found among the objects. The geometry of the weir and river bed is selected with the help of a file chooser button, that is shown once the volume type polytope was selected.

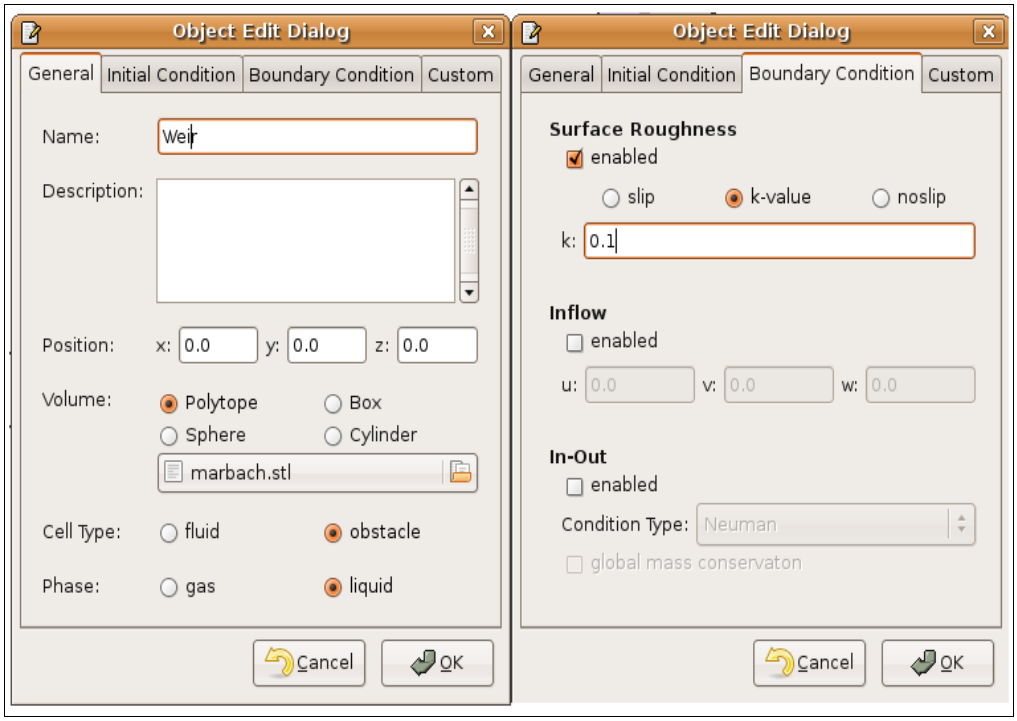

Illustration 32: Object Edit Dialog

Since the weir is clearly an obstacle the cell type is set to obstacle as can be seen in illustration 32. In *NaSt3DGPF* the level set function for the two phase flow is computed for all cells including obstacle cells. Since the river bed and weir are mostly surrounded by water the phase liquid is selected.

Since the river bed has a strong influence on the velocity of the flow and as a result on the volume of water that passes through the weir the boundary condition Roughness is set with a k-value of 0.1. This means that the surface is made up of approximately 10 cm large stones.

By exiting the dialog with *OK* the geometry is loaded into the editor and displayed. Illustration 33 show the geometry of the weir and river bed loaded in the editor. It is visible that the geometry and the simulation domain fit together. This is relatively important since gaps may produce errors in the simulation.

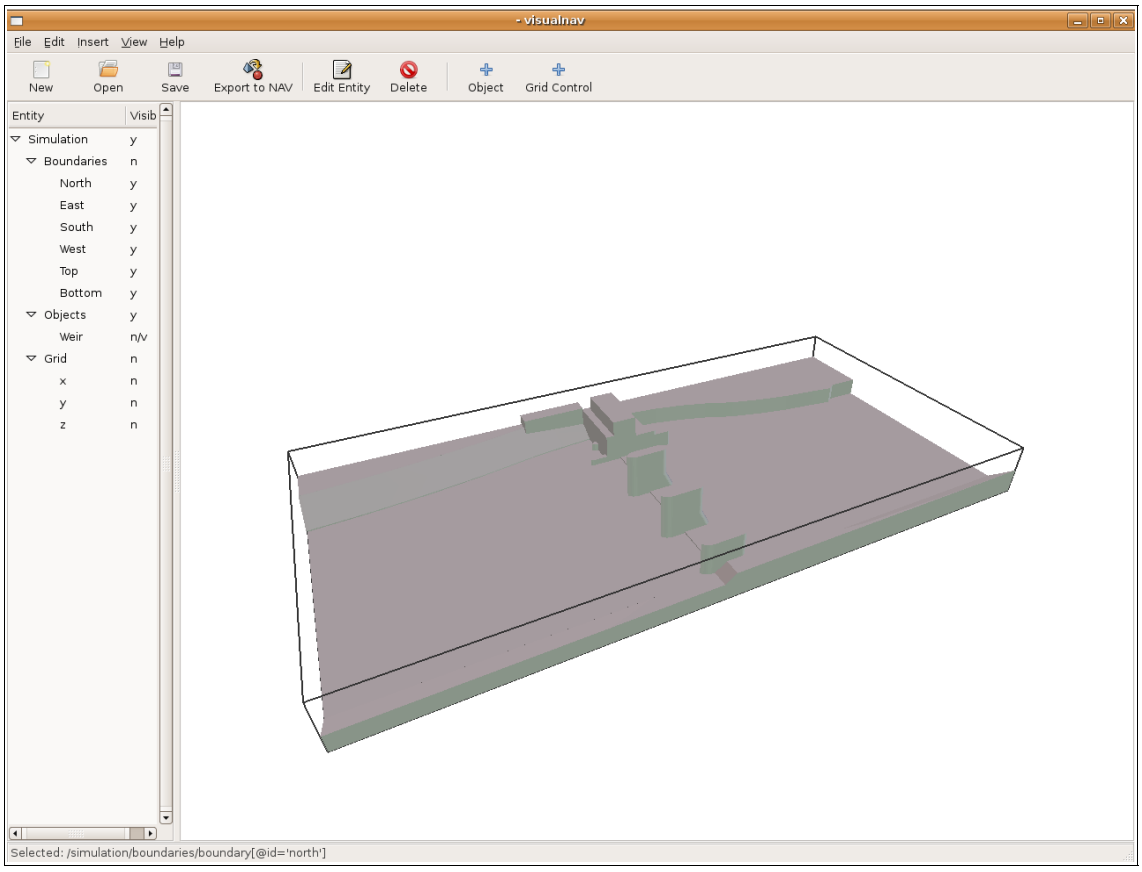

Illustration 33: The Weir and River Bed

6.3 Initial Water and Air

Since everything in *NaSt3DGPF* that has a volume is defined by an object, the initial water and air volumes are also set by objects. In this case boxes are used to define the volumes. The settings for the initial water and air are listed in table 2.

Object	*Position [m]*	*Size [m]*
Water Up Stream	(62.5, 62.25, 4)	(125, 124.5, 8)
Water Down Stream	(187.5, 62.25, 3)	(125, 124.5, 6)
Air Up Stream	(62.5, 62.25, 14)	(125, 124.5, 12)
Air Down Stream	(187.5, 62.25, 13)	(125, 124.5, 14)

Table 2: Initial Air and Water

Same as with the weir and river bed the water volume is inserted by selecting *Insert* and *Object* in the main menu. Illustration 34 shows the settings of the water volume upstream of the weir in the *Object Edit Dialog*.

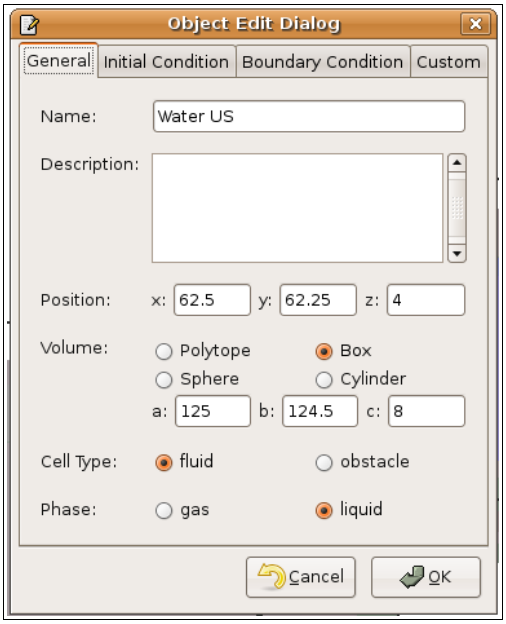

Illustration 34: Parameters of the Water Up Stream

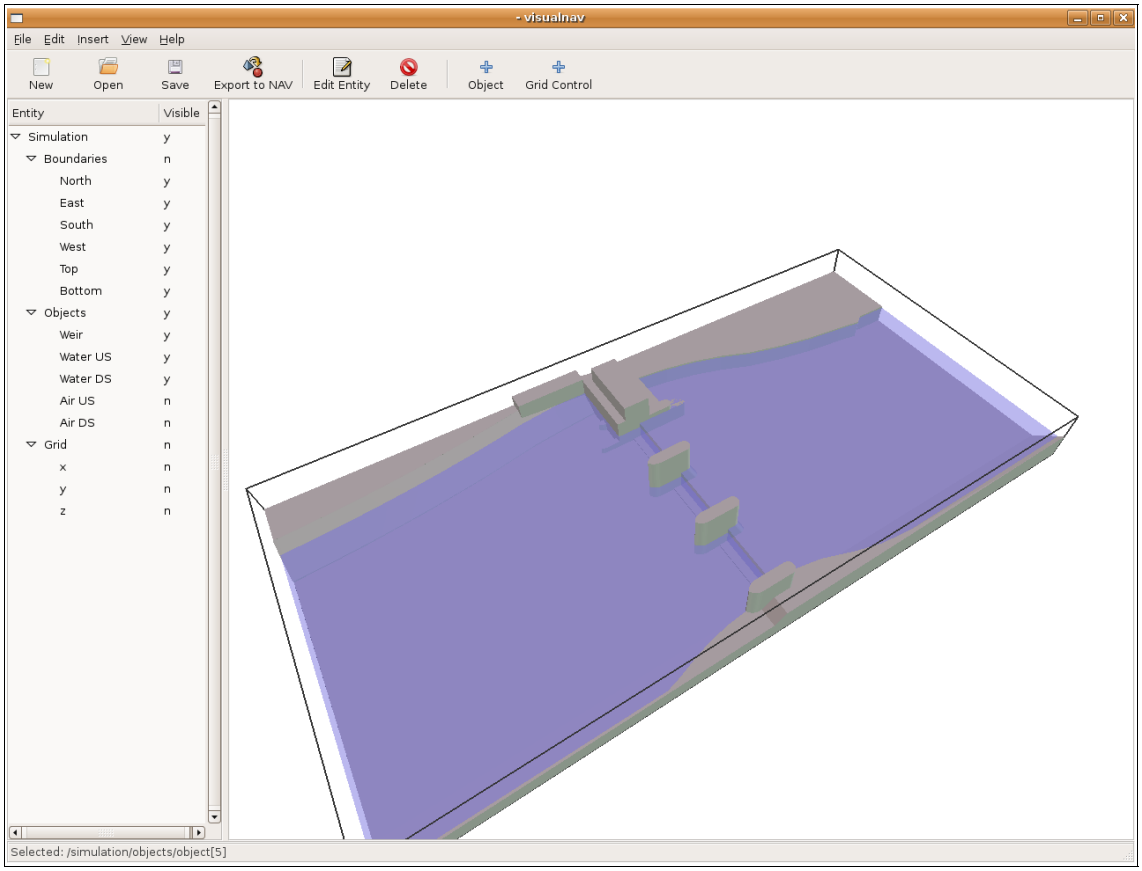

Illustration 35: Initial Water

Illustration 35 and 36 show the initial water and initial air. The editor displays the water volume as transparent blue and the air as a transparent yellow. Although the display of the air volume is relatively informative it occludes the following edit steps. It is advisable to hide the air volumes. This is done by selecting the volume and then *View* and *Hide* in the main menu or *Hide* in the right click pop-up menu.

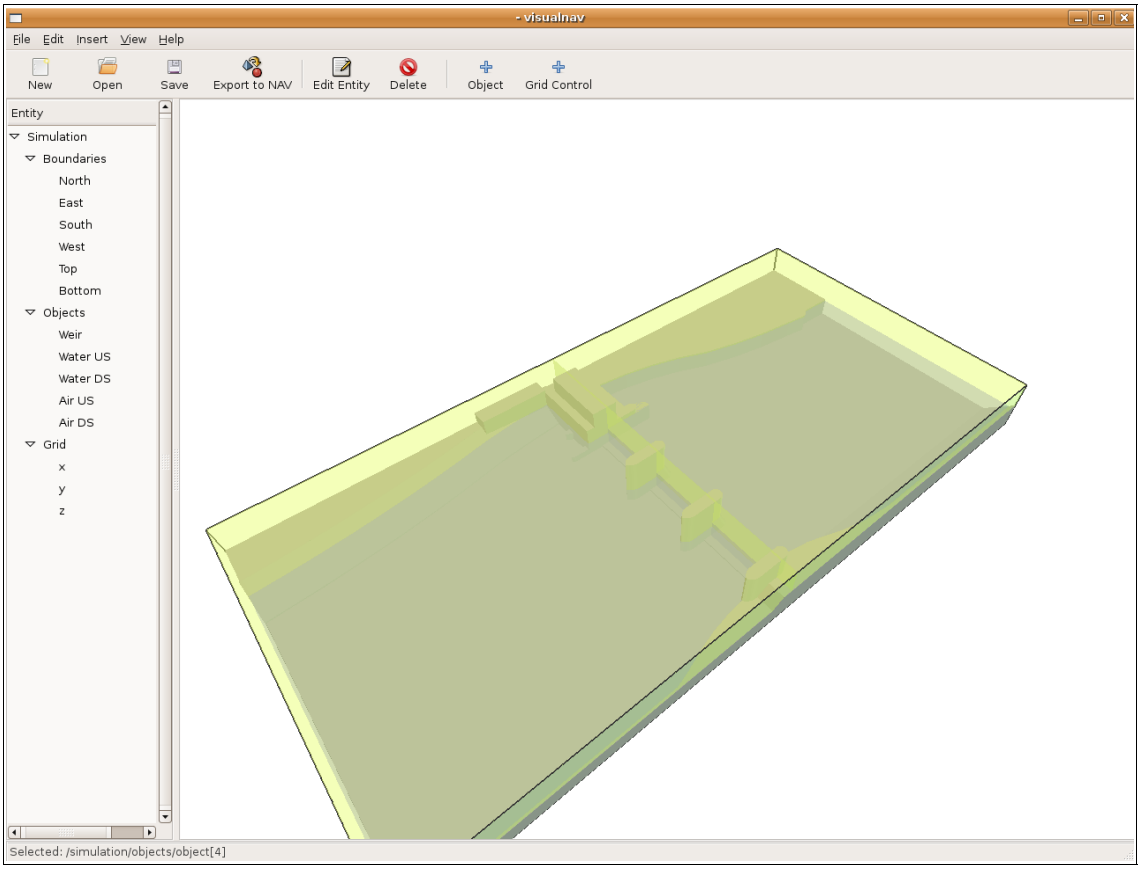

Illustration 36: Initial Water and Air

6.4 Simulation Boundary

Since the simulation dose not describe a close system boundary conditions must be set. The boundaries are defined in more detail on page 25. To edit boundaries a boundary must be selected and the *Boundary Edit Dialog* is raised by clicking on *Edit* on the right click pop-up menu or *Edit* and *Edit* in the main menu. Detailed information on the *Boundary Edit Dialog* can be found on page 49.

By definition the south boundary is a inflow boundary. It is upstream and thus the inlet for water. In this case the water and air flows in at 1 meter per second in the x direction as can be seen in illustration 37.

Same as the south boundary the north boundary is by convention the outflow boundary. As can be seen in illustration 37 the *InOut* boundary condition is set. In addition the fixed outflow condition is added to the outflow condition, indicating that the water level is to be held at 6 meters above the simulation floor.

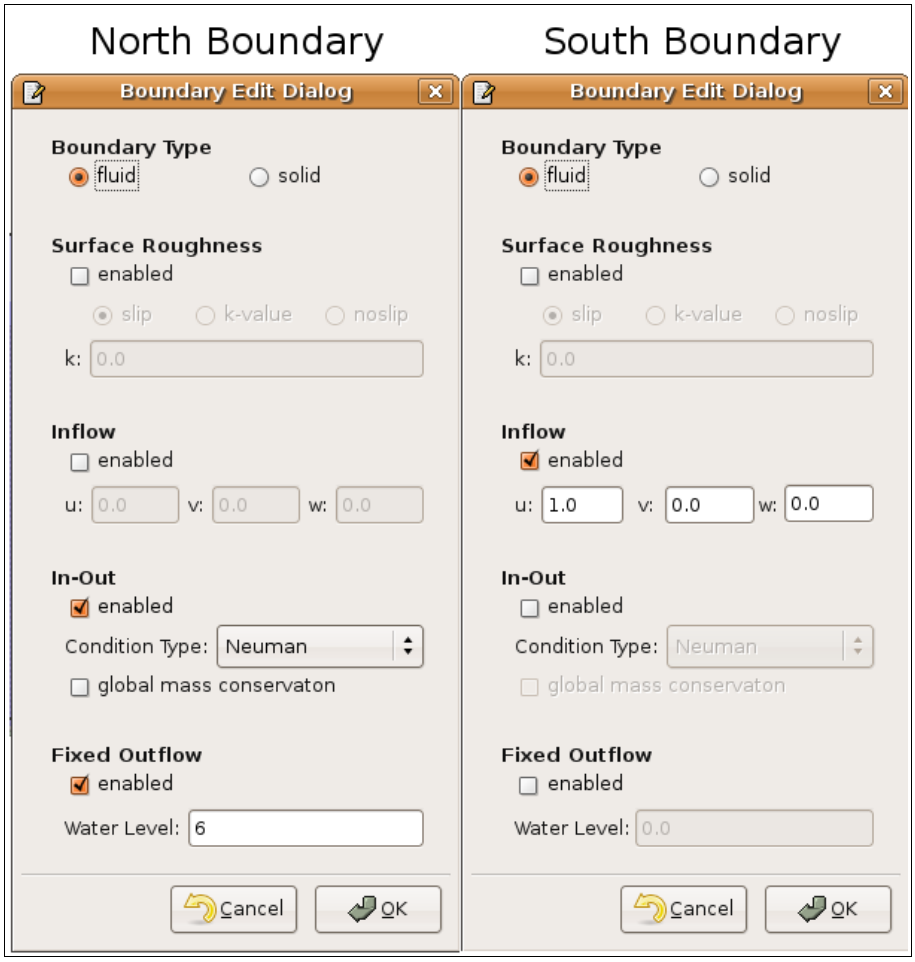

Illustration 37: South and North Boundaries

The other four boundaries are kept in their default setting to be *solid,* without the any discharge of watter and air.

The two boundary conditions are now visually presented to the user. In the case of the fixed outflow condition the symbol for the water level is displayed in red, as can be seen in illustration 38. It is drawn so that the indicated level is the level that was set. The inflow condition is indicated by a green arrow pointing in the direction of flow. Illustration 37 shows the inflow for the south boundary.

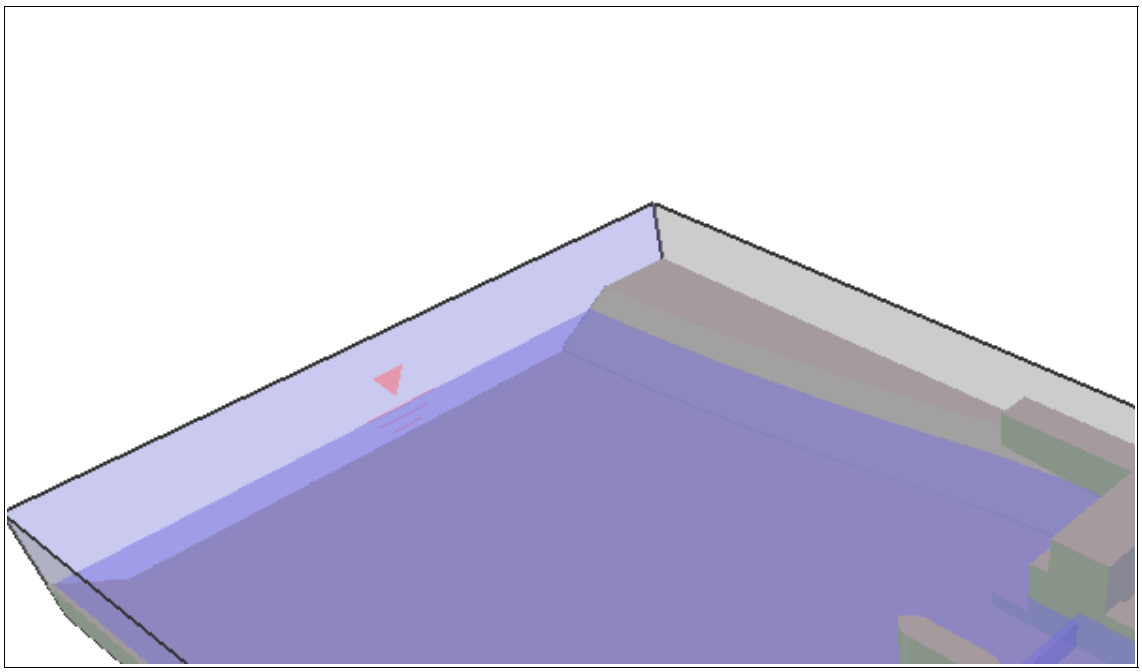

Illustration 38: Fixed Outflow Condition

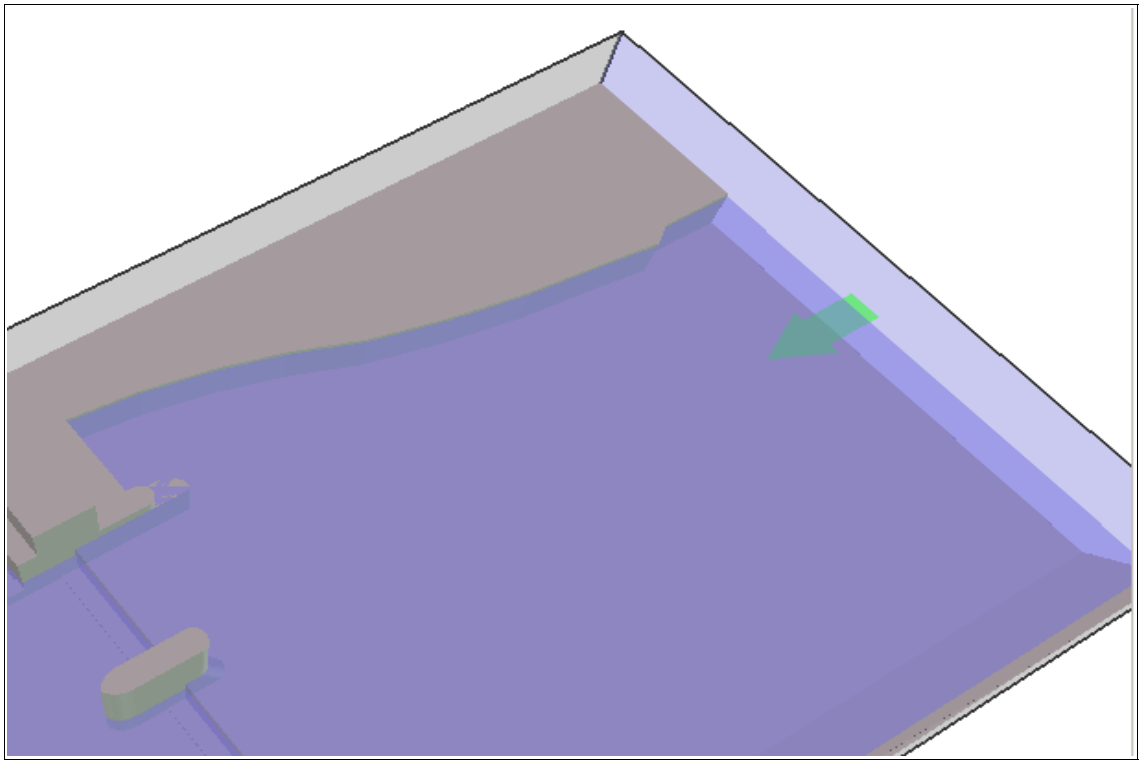

Illustration 39: Inflow Condition

6.5 Defining the Grid

The grid has to be defined. In NaSt3GPF the grid may be scaled along each axis and *visualnav* provides the means to define a grid with the help of grid control points, or short grid controls. The grid is then linearly interpolated between two grid controls and the remaining space at the border is filled up with the cell width of the closet grid control. This is explained in more detail on page 29.

Position	*Cell Width*
x = 20 m	0.5 m
x = 100 m	0.2 m
x = 150 m	0.2 m
x = 230 m	0.5 m

Table 3: Grid Controls on the x Axis

Since grid controls are located on an axis it must be clear on what axis the grid control is being inserted so an axis must be selected in the entity tree. After an axis is selected grid controls may be inserted by selecting *Insert* and *Grid Control*. This will raise the *Grid Control Edit Dialog,* which is explained in more detail on page 45. The *Grid Control Edit Dialog,* as seen in illustration 40, provides the means to modify the position of the grid control and the cell width at that position.

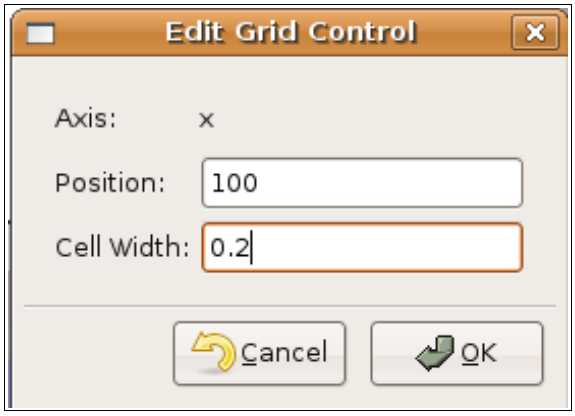

Illustration 40: The Grid Control Edit Dialog

It is always a trade of between accuracy and computation time. Since in this case only the discharge is of importance, the simulation may be relatively coarse. At the weir the cell width should be 20 cm. With this geometry it only makes sense to scale the grid along the

x axis. The grid is scaled as described in table 3. To reduce computation time the grid is stretch up to 50 cm at the edges and a certain padding is left so that the boundary conditions may properly be introduced into the simulation.

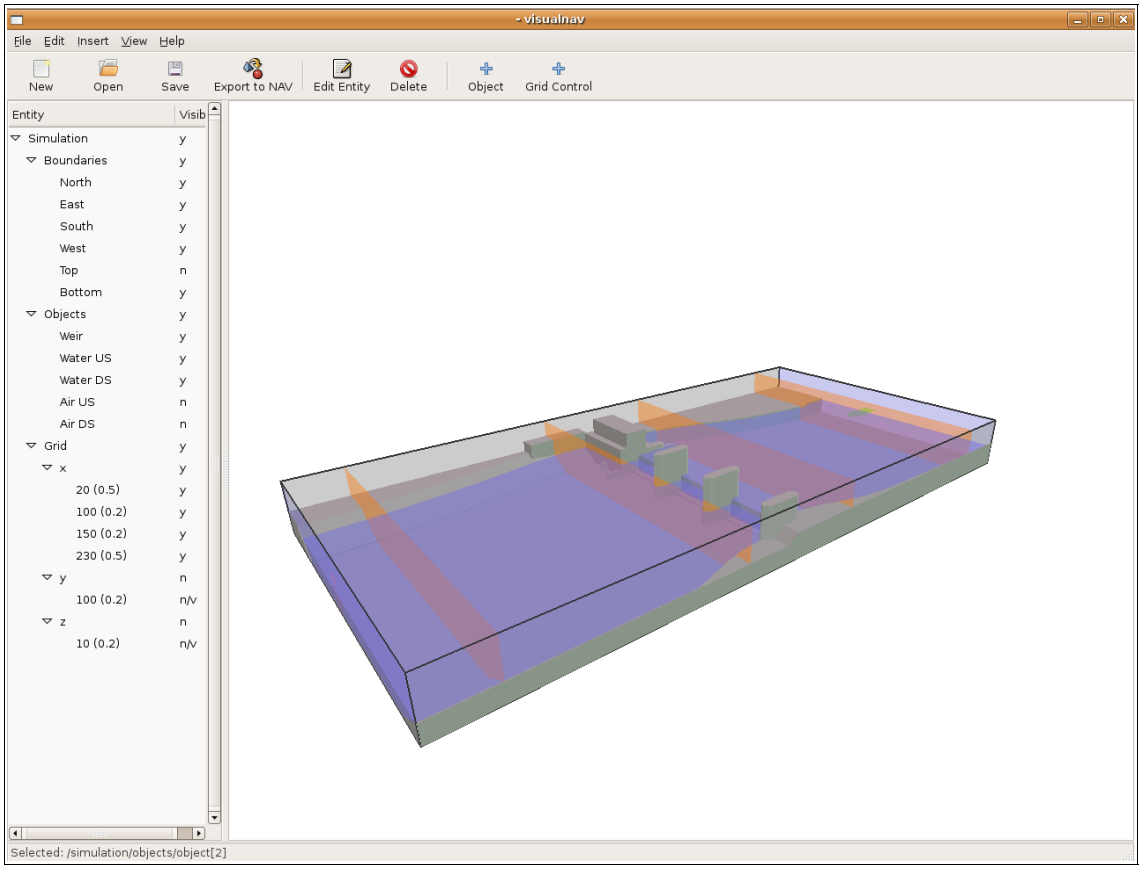

Illustration 41: Grid Controls on the x Axis

Illustration 41 shows the visualization of the grid controls for the x axis. It can be clearly seen that the the second is clearly before and the third clearly behind the weir. By showing the positions of the grid controls the user has immediate feedback of the positions of the grid controls.

On the y and z axis two grid controls are inserted so that the grid definition for a constant grid is generated on those axis.

6.6 Save and Export

Now that the simulation is finished it should be saved. This is done with the common method by selecting *File* and *Save* or *Save As*. The definition for the simulation will now be save to file in the *XNAV* format. The format is a *XML* based format defined for this editor.

Since the simulation should run in the numerical core *NaSt3DGPF* the definition must be exported to *NaSt3DGPF's* native format. This is done by selecting *File* and *Export to NAV*. This will trigger the conversion of the data stored in *XML* with the help of a *XSLT* style sheet. The conversion may also be done externally with the help of any *XSLT* processor, like for example the free *xsltproc*.

6.7 Modifying the Simulation: N – 1

The simulation that was just prepared has all gates open and is the reference against which the case with one blocked gate is compared.

The simulation may be loaded again from the file stored in the *XNAV* format. This is done by selecting *File* and *Open*.

For this case a box is sufficient to simulate the blocked gate, although the geometry of the gate may also be loaded. Similar to defining the air and water volumes an obstacle is inserted at (125, 73.5, 4) and the size (2, 25.5, 6).

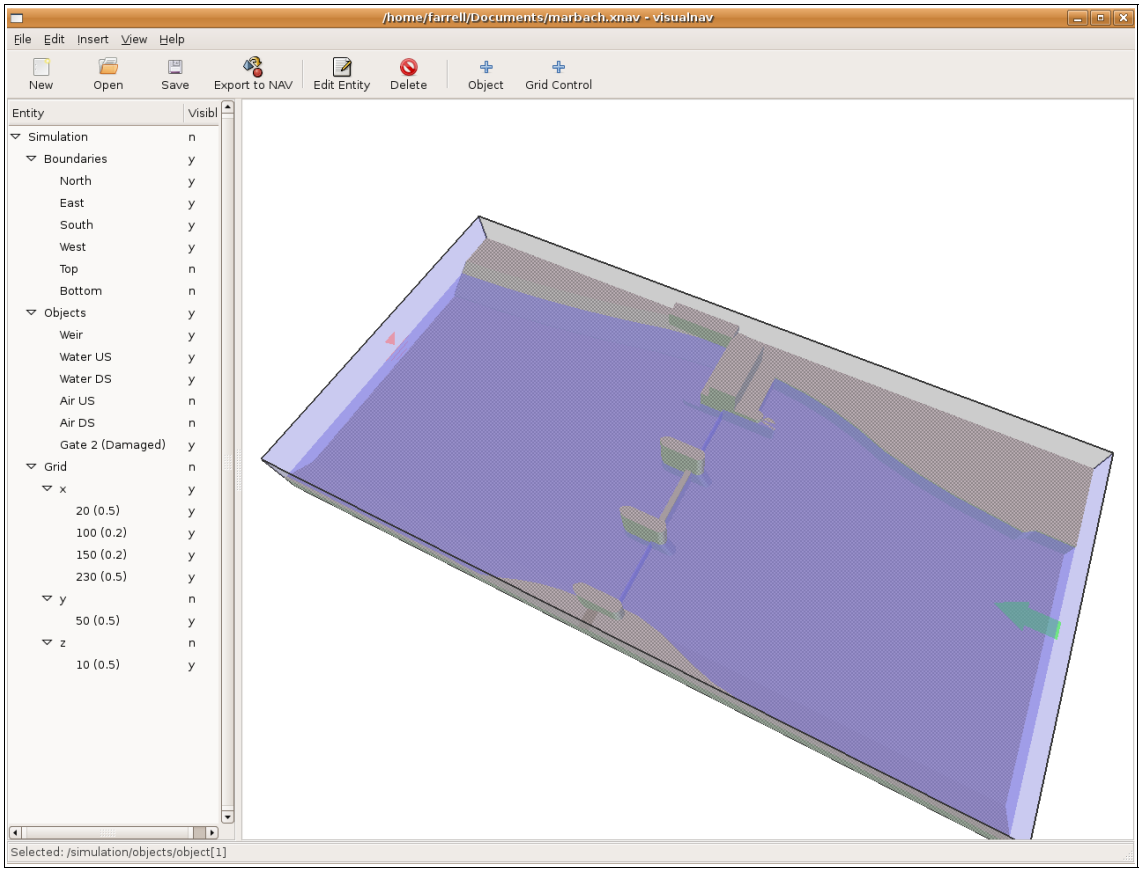

Illustration 42: Final Simulation with Gate

Illustration 42 shows the final simulation with the blocked gate. The weir and river bed geometry can be clearly seen as the two boundary conditions regarding the flow.

Chapter 7 Conclusion

The presented editor can be used to create the definition for a simulation in the computational fluid dynamics model *NaST3DGPF*.

In the first and second chapters the problem was analyzed and the requirements for a graphical editor where gathered. In the following chapter the design process from its beginnings was described, including the knowledge gained from user interviews, paper prototyping and design with CRC cards.

The data model, which is an extension to that provided by NaSt3DGPF was described and refined in chapter 4. In the following chapter the application was described that uses and wraps that model. In addition the visual representation and manipulation with the help of dialogs was presented there.

In the chapter “Proof of Concept” the use of the final product is demonstrated on a real world example that is part of the investigations conducted by *Federal Waterways Engineering and Research Institute.*

In conclusion it can be said that all functional requirements as stated on page 13 where met.

Chapter 8 Outlook

Although the *XNAV* format holds all required data, it is not optimized in its representation. As a result some constructs are not optimal and may be improved, such as *XML* elements that hold their single value as a attribute and not as a child text. An other example is the use of booleans for conditions, where the presence of an element would be less ambiguous.

In the application *visualgrid*, which provided the algorithm for the scaled grid by the means of grid control points, it is possible to preview the result of the grid generation algorithm. In addition the geometry discretized against that grid is visualitzed. This information is relatively useful and should be included in the visualization.

Although the flow of data and notifications of change has gratefully improved from the first designs, there is still some significant overhead. The application rebuilds implicit data to often. This can lead to some slowdown if the description starts to get large. This has the strongest impact in the graphic view regarding large polytopes.

Although the groundwork was laid with the *element changed*, *element added*, *element removed* and *selection changed* signals of the document, it was not used to its full extent.

The data model used in the editor is strongly coupled to *NaSt3DGPF's* representation with most of the ambiguities and shortcomings. In future work it may be possible to redefine some parts of the application. Objects could be broken up into solids, conditions and initial conditions and so that ambiguities are distilled out.

Bibliography

1. Erich Gamma et al. *Design Patterns* (1995)

2. Andrew Hund, David Thomas *The Pragmatic Programmer* (1999)

3. Calum Benson, Adam Elman, Seth Nickell, Collin Z Robertson *GNOME Human Interface Guidelines 2.0* (2004)

4. Tim Bary, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler, François Yergeau *Extensible Markup Language (XML) 1.0 (Fourth Edition)* 2006

5. James Clark, Steve DeRose *XML Path Language (XPATH) Version 1.0* (1999)

6. James Clark *XSL Transformations (XSLT) Version 1.0* (1999)

7. M. Griebel et al. *NaSt3DGPF User's Guide* (2005)

8. Charles Donnelly, Richard Stallman *Bison Manual* (2005)

9. Vern Paxson *Flex Manual* (1995)

10. Sean Farrell *Erstellung eines Generators für strukturierte und gradierte 3D Berechnungsgitter auf Basis von CAD-Daten* (2005)

11. Sean Farrell *Erstellung einer gaphischen Benutzeroberfläche für einen Gittergenerator* (2005)

12. OpenGL Architecture Review Board *OpenGL Programming Guide* (1993)