FAKULTÄT FÜR !NFORMATIK

Faculty of Informatics

VR-Client for Scenario-based Response Training in Disaster Management

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieurin

im Rahmen des Studiums

Visual Computing

eingereicht von

Johanna Donabauer, BSc
Matrikelnummer 01226287

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Ao.Univ. Prof. Dipl.-Ing. Dr. techn. Eduard Gröller
Mitwirkung: Dipl.-Ing. Dr. techn. Jürgen Waser
Dipl.-Ing. Harald Steinlechner

Wien, 23. Jänner 2019

Johanna Donabauer	Eduard Gröller

VR-Client for Scenario-based Response Training in Disaster Management

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieurin

in

Visual Computing

by

Johanna Donabauer, BSc
Registration Number 01226287

to the Faculty of Informatics

at the TU Wien

Advisor:	Ao.Univ. Prof. Dipl.-Ing. Dr. techn. Eduard Gröller
Assistance:	Dipl.-Ing. Dr. techn. Jürgen Waser
	Dipl.-Ing. Harald Steinlechner

Vienna, 23rd January, 2019

Johanna Donabauer

Eduard Gröller

Erklärung zur Verfassung der Arbeit

Johanna Donabauer, BSc

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 23. Jänner 2019

Johanna Donabauer

Acknowledgements

Thanks to Eduard Gröller who agreed for supervising me and for his constructive feedback to improve my work. I want to thank VRVis Research Center of Virtual Reality and Visualization for giving me the opportunity to do my diploma thesis with them. Special thanks to Jürgen Waser for providing this interesting research topic and all his efforts on the project. Another very special thanks to Harald Steinlechner for the big support during the implementation and development of this work as well as for pushing me in the right direction. Warm thanks to Georg Haaser and Attila Szabo who were always available for any question regarding implementational details. Many thanks to Daniel Cornel for his help in the beautification process of the rendering and for discussing and eliminating all uncertainties. Thanks to Stefan Maierhofer, Michael Schwärzler, Agnes Knor, and Daniela Stoll and the employees at VRVis.

Special thanks to my parents for the whole support during my studies and their patience. Heartfelt thanks to my siblings and especially to my boyfriend for their patience and their general support in every situation. Last but not least, to all my friends and colleagues - thank you for making my life colorful.

Thank you!

Kurzfassung

Bei Naturkatastrophen wie Überflutungen zählt für das zuständige Personal jede Sekunde um Menschenleben zu retten und Gebäudeschäden zu minimieren. Um sich für den Ernstfall vorzubereiten, wird das zuständige Personal durch reale Trainingseinheiten geschult. Diese Trainingseinheiten können sowohl sehr kosten- als auch zeitintensiv sein. Eine mögliche Alternative dazu bietet Virtual Reality (VR). In der Vergangenheit haben bereits viele verschiedene Trainingsbereiche die Technologie für sich entdeckt. Zu den erhofften Vorteilen zählt eine flexiblere Gestaltung verschiedenster Trainingsszenarien sowie eine Kosten- und Zeitersparnis. Zusätzlich können VR Applikationen im Bereich der Öffentlichkeitsarbeit eingesetzt werden, um das öffentliche Bewusstsein zu stärken. Aus diesen Gründen ist das Ziel dieser Arbeit die Entwicklung einer VR Trainingsapplikation, um eine entfernte Überflutungssimulation zu steuern. Dadurch kann eine sichere und realistische Trainingsumgebung für das verantwortliche Personal geschaffen werden. Mit Hilfe verschiedener Szenarien können unterschiedliche Überflutungssituationen simuliert und trainiert werden. Um eine Überflutung zu verhindern oder zu mildern, können Schutzmaßnahmen gesetzt werden. Diese werden im weiteren Verlauf der Simulation berücksichtigt. Durch einen Operator-Trainee Aufbau können auch zwei Personen gemeinsam an einem Schutzplan arbeiten. Ein Experte arbeitet als Operator am PC, während der/die Auszubildende von diesem Instruktionen erhält und die Aufgaben in der virtuellen Welt erledigt. Um ein immersives VR Erlebnis zu bieten, werden eine hohe Bildfrequenz und zwei hochauflösende Bilder benötigt. Dafür muss der PC bestimmte Hardwareanforderungen erfüllen. Zusätzlich wird auch für die Überflutungssimulation eine hohe Rechenleistung benötigt, um diese schnell durchzuführen. Aus diesem Grund wird bei der implementierten VR Applikation auf eine Client-Server Architektur zurückgegriffen. Der Server ist für die Simulation zuständig. Der Client kümmert sich um die Visualisierung der Simulationsdaten in VR, sowie die schnelle und effiziente Verarbeitung der Daten. Um die benötigte Leistung zur Visualisierung zu gewährleisten, wird zusätzlich auf eine leistungsstarke Rendering Engine zurückgegriffen und mit passenden Grafikbefehlen, die auf die verfügbaren Daten abgestimmt sind, kombiniert. Die Evaluierung dieser Arbeit basiert auf der Messung der erreichten Wiedergabegeschwindigkeit, da diese maßgeblich für ein immersives VR-Erlebnis ist. Dafür werden zwei unterschiedliche Implementierungsstrategien verglichen. Da die Update-Geschwindigkeit des Wasserflusses von wesentlicher Bedeutung für die Überflutungssimulation ist, werden eine CPU und eine GPU Implementierung innerhalb der Evaluierung verglichen.

Abstract

In times of natural disasters like floods, the fast action of domain experts saves human lives and reduces high damages of the urban infrastructure. The training of different response plans of the responsible personnel should help in making the right decisions in time critical situations. As the creation of various physical training environments takes plenty of time, the use of virtual reality (VR) is a possible alternative. In recent years, different application domains with training purpose have been shifted to make use of the new developments in the field of VR. The desired benefits are a more flexible generation of different realistic training environments with low budget and material resources. Additionally, the VR application can serve as a public communication tool to raise the sense of awareness. Based on these considerations, the aim of this work is to create a VR training application to steer a remote flood simulation. The goal of the application is to provide a safe and realistic environment to train the responsible personnel. Through providing different scenarios, multiple flood events can be simulated and trained. The placement of barriers through interacting with the virtual environment offers possibilities to mitigate the results of the simulated floods. An Operator-Trainee setup enables the collaborative work between experts and trainees. While the expert works as an operator with a PC client, the trainee is able to perform instructions given by the operator within the virtual environment. VR applications demand for high and steady frame rates as well as two high resolution images for both eyes to provide an immersive VR experience. Based on these conditions, appropriate PC hardware is needed to run a VR application in general. Additionally, high computational power is needed to perform the different flood simulations in a fast way. In order to achieve the performance requirements, the VR application is implemented within a client-server architecture. The server is responsible for performing the flood simulation, while the client deals with the VR-related tasks. These tasks comprise the visualization of the simulation data in VR and a fast and efficient processing of the data. In combination with a high performance rendering engine and graphic commands suitable for the given data, the desired performance can be achieved. As the feeling of immersion is highly depending on the provided frame rates, the evaluation of this first prototype is based on the achieved rendering performance. This is measured and evaluated based on two different implementation strategies. Another important measurement is the update time of the water flow. A comparison of a CPU and a GPU implementation is presented within the evaluation.

Contents

CHAPTER 1

Introduction

From 1993 to 2012 floods were the most frequently occurring natural disaster world wide. More than 2000 million people were affected by such an event [Cenc]. Examining the annual reports about natural disasters, this trend does not seem to change [Cend], [Cene], [Cena], [Cenb], [Cenf]. Although floods are reoccurring natural disasters, the preparation against and mitigation of the results of such an event are still an ongoing process. The main long-term goals are to save human lives and the preservation of the urban infrastructure. Based on these reasons, it is necessary to be prepared for such an event and the associated consequences. For preparation, the responsible persons need to train for different kinds of possible flooding scenarios and their impact on the urban environment. Additionally, plans have to be evaluated, how to best protect a city and their inhabitants.

Although flood maps [flo] give a good overview to identify possible flood-prone areas they are not sufficient to train responsible personnel for different flooding events. For training purposes, physical training setups need to be created to provide a training environment. However, creating a physical artificial flooding scenario that models the real world conditions is practically not feasible. An alternative option is to use computer programs to simulate different flooding scenarios. For these simulations, real-world environments are replicated by computer models. Based on these models, different flooding conditions can be created to analyze possible effects on the environment. Through using computer supported flood simulations, we can create various different flooding scenarios by simply changing simulation parameters [SWR$^+$13]. This gives flood managers the possibility to create a pool of different flood plans. Depending on the given conditions an appropriate plan can be selected to mitigate the results of a flood. Such simulation programs are computationally expensive. Appropriate PCs with the desired computational power are needed that are able to calculate the different simulation data as fast as possible. Additionally, by using state of the art algorithms to compute the necessary data, the calculation performance can be further improved [HPW$^+$16].

A disadvantage of the use of computer programs to train responsible personnel is that the physical training to create protection mechanisms is shifted to pressing buttons, which reduces the sense of awareness of the threat [BM07]. Through the latest developments in the field of virtual reality (VR), a new possibility of training setup has evolved. The idea of setting up a virtual environment (VE) for different kinds of training purposes is not new. There are already various training applications on the market [vrsa], [adm], [SD18], [Goe16]. The available VR applications for flood simulation show predefined scenarios and do not let the user step through the simulation progress or make some changes in the flooding scenario. The main objective of this thesis lies in creating a VR application that steers a remote flood simulation and displaying it in the VE. To explore the virtual space, basic navigational interaction techniques are supported. In order to interact with and manipulate the simulation domain, the placement of protection mechanisms is provided. Furthermore, additional interaction methods let the user control the simulation progress as well as change between different scenarios. The successful development of such a system is depending on various factors.

1.1 Challenges

In order to create an immersive flood simulation application in VR, high computation power is required. This computational power is not only needed to provide the necessary capacity for the VR hardware in general, but also to do the desired simulation calculations for a flooding event. The incorporation of these two resource intensive application fields alone already poses a technical challenge as an interactive virtual flood experience in VR has to be provided. Additionally, the communication of the simulation data between the PC and the VR hardware has to be done in an efficient way, so that the user can explore the virtual world without any interruptions while scene updates are performed. Based on these considerations, the following challenges can be formulated to integrate these two high performance applications:

- Support different interactions within the VE to modify flood simulation parameters and add flood protection mechanisms
- Compute the flood simulation and visualize it on a VR display
- Provide constant high frame rates, even during simulation updates, while moving through the virtual space

The first point in the list is dedicated to provide the user possibilities to alter the simulation state. The goal of an immersive VR application crucially demands for interaction techniques applicable to explore the virtual world and reduce the impact of a flood. As a simulation visualizes a progress in time, appropriate control techniques are required to manipulate the simulation progress. The second point in the list is about to bring the two application fields of a flood simulation in a VE together. This is already a difficult task because of the high performance requirements of both fields. The third point concerns cybersickness, which is likely to occur if low frame rates are provided. Additionally, the sense of presence to the user can be decreased [WS98]. Presence is the feeling of being at the visualized place for real, while being physically at a different location [WS98]. For the user this means that he/she forgets to be physically in another place than he/she is seeing. The sense of presence is heavily depending on the feeling of involvement and immersion within the VE. The involvement within the VE depends on what extent a user focuses on the presented stimuli. If the user gets distracted by disturbing sounds outside the VE, or the wearing of the VR headset is not convenient, the sense of involvement suffers. Also the emotional state of the user influences the feeling of involvement [WS98]. In addition to involvement, the feeling of immersion is another big issue within the virtual world. Immersion describes to which extent a user feels like being included into the VE. This highly depends on how naturally he/she can interact with the surrounding environment as well as how much the user is included in the VE. However, the sense of immersion can also be influenced by the provided update rate of the VR display [WS98]. When having low frame rates, the sense of immersion decreases as well as feeling temporarily uncomfortable increases, resulting in a bad VR experience for the user [BM07].

1.2 Contributions

Creating a flood simulation for a VR setup is not only useful for responsible personnel. It can also be used to show non-specialists the work of flood managers. By seeing the 3D simulation within a 3D space, the depth perception can be retained. This can lead to a higher sense of awareness of the possible threat by a flood and makes the application usable as a public communication tool as well. Additionally, civilians can view what happens at different weather conditions. This might be useful to explain, why and which evacuation measures are done. Furthermore, such an application could also be useful in house planning. Through knowing possible flooding impacts, some prevention methods can be included.

The implementation of such an application comprises interaction challenges as well as technical challenges. In order to meet the presented challenges, as listed in the previous Section 1.1, different issues have to be considered for the implementation. Depending on the used VR hardware appropriate interaction methods have to be compared to determine, which ones suit best for the desired functions. The used VR hardware already states the desired performance requirements to ensure a proper working of the system. Through using existing systems, that are optimized for their application domain, and combining them, a better distribution of the computational intensive tasks is possible. This makes it possible to focus on an efficient data communication and representation. By incorporating this efficient communication with data dependent graphics commands the data can be sent to the graphic hardware in an optimized way. In combination with a high performance rendering engine for visualizing the simulation state, an interactive VR application can be created.

Based on these considerations, the implemented VR system can be described by the following characteristics:

- A client-server setup where the server is responsible for calculating the flood simulation data and the client is responsible for rendering the sent simulation data from the server in a VE.
- Two different VR application setups:
  - an Operator-Trainee application, for a collaborative working environment. The (more experienced) operator works with a separate PC client while the trainee is immersed in the virtual environment. Both can interact with the simulation scenario and see the changes the other one has made.
  - a standalone VR application. Only one user works in the virtual environment.
- A methodology to ensure high and steady frame times to provide a good feeling of immersion in VR. This is possible by implementing a state-aware client. Based on the given domain knowledge, data optimized graphics commands can be used combined with a high performance rendering engine for the rendering.
- A methodology to perform fast data updates in order to smoothly show a flood animation.
- Interaction possibilities within the VE to modify the simulation state. This includes the design of protection barriers as well as the investigation of alternative scenarios.

1.3 Structure of the Work

As the whole system integrates different research areas, the thesis has to deal with various different fields of related work. Therefore, Chapter 2 first gives detailed information about how flood simulations can be calculated at all and which decision support systems are currently available. The overview of different decision support systems is followed by some work about handling dynamic scene changes in an efficient way as well as some additional GPU optimization techniques. After that, some insights in possible VR training applications are given with the focus on flood simulations. The VR applications are followed by an introduction to VR interaction techniques and how to control the system state with menues. The related work finishes with an overview about movement techniques in VR.

Chapter 3 gives an overview on the implemented system. In Chapter 4, all possible interactions within the virtual world are explained. The implementation part in Chapter 5 explains more details about the underlying system implementation. This part of the thesis heavily deals with different kinds of data structures and how the creation of the scene has been done. In Chapter 6, a performance evaluation of different projects is presented. The last Chapter 7 gives some final remarks about the findings through implementing and testing the system as well as some possible extensions and improvements for the future.

CHAPTER 2

Related Work

Creating a flood simulation application comprises different requirements to create correct and realistic water flow behavior. Additionally, if the simulation visualization should be usable for domain experts as well as beginners and civilians, an understandable representation of the simulation results for both parties is necessary.

2.1 Interactive Flood Simulation

Through the use of an interactive flood simulation, flood events are visualized. An example of an interactive flood simulation is given by RiverFlow2D [riv]. With RiverFlow2D, flows of rivers and estuaries can be exhaustively visualized. In addition to dam break floods, the effects of bridges and other hydraulic structures can be considered and simulated. Through adding different modules to the basic application, the functionalities of RiverFlow2D can be further extended. Such extensions allow for the considerations of mud and debris or pollutants in the system. Another important extension considers erosion and deposition of rivers in the simulation[riv].

The basis for describing the flow of a fluid is formed by the Navier-Stokes equations. Because of their complexity, no schemes are available to solve the 3D Navier-Stokes equations in real-time for environmental applications. Approximations have to be used to yield a physically realistic flow behavior. In Computer Graphics, Smoothed Particle Hydrodynamics (SPH) is a basic method to simulate water flow in real-time [DG96]. SPH is based on the Lagrangian particle method. Water is represented by a discrete number of particles, each described by hydrodynamic properties. Based on these properties, water flow behavior can be simulated. The number of particles increases with the size of the simulation domain. Additionally, the more detailed the water flow is, the more particles are needed. But the increasing number of particles slows down the rendering performance. Even though a performance gain can be reached by implementing some optimizations like using the GPU [HKK07] or multiple GPUs [RBG+12], [VBDRC13], no test results are given for larger simulation domains.

In contrast to the Lagrangian particle based method for calculating the water flow behavior, another physical-based method is to use the Eulerian approach to solve the 2D Shallow Water Equations (SWEs). This method is based on using an underlying 2D grid structure with fixed points to calculate the fluid behavior. Thus it is more suitable to calculate the fluid flow for large-scale simulation domains. The water surface is described by a height field, where each cell of the underlying grid structure is assigned one height value. The fluid motion can be evaluated by solving the underlying hyperbolic partial differential equations. Through incorporating different optimization techniques [AA09], [KP07], [HWP+15], and exploiting the architecture of the used graphics hardware [HPW+16], simulation performance improvements can be achieved. This makes it possible to perform real-time flood simulations for large-scale simulation domains. The simulation framework used in this work bases on this method of calculating the water flow.

2.2 Decision Support Systems

Besides the efficient calculation of the water flow, software tools are needed, which support decision makers and flood managers in their work. Depending on the user group and the tasks that have to be performed, these applications have different requirements to fulfill. As different as the application requirements are, as different are the visualizations of these. Therefore, already a variety of tools and simulation programs exist.

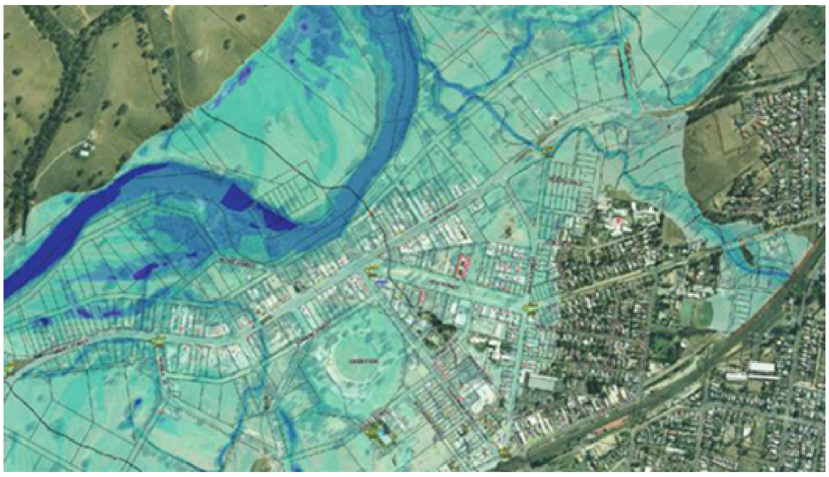


Figure 2.1: A flooding scenario produced by TUFLOW. Image taken from the Flood Modeller Website provided by Jacobs [tufa].

TUFLOW [tufb] is one type of program to simulate flood situations. Figure 2.1 shows a flooding scenario visualized with TUFLOW. TUFLOW is used to simulate different kinds of urban floodings. It is also able to model a storm tide or a tsunami inundation.

Furthermore, the modelling toolset of TUFLOW also comprises pipe networks. In order to speed up the computational processing time, another module exists that accesses the graphics processing unit (GPU) [tufb].

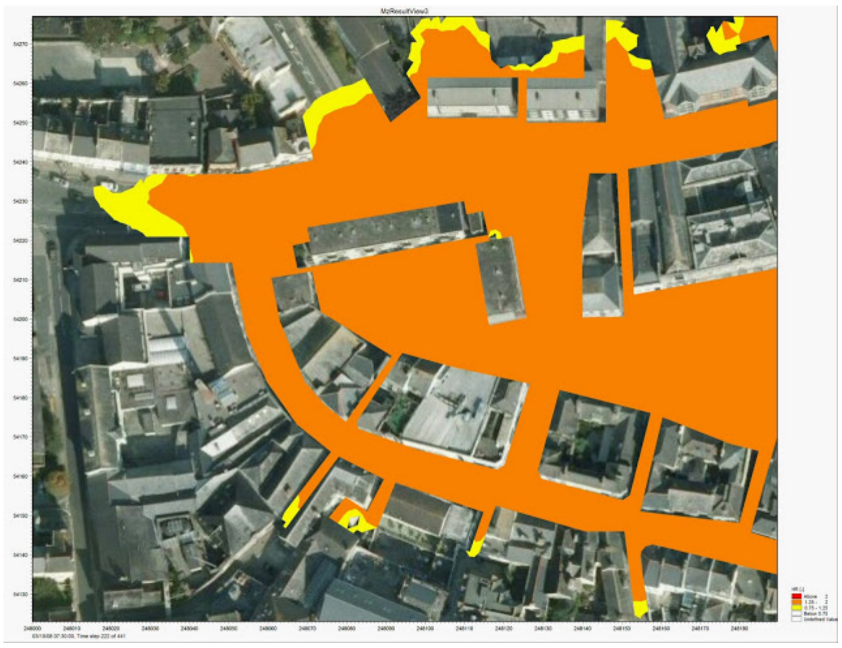


Figure 2.2: A hazard map used in Mike Flood. Image taken from the DHI UK and Ireland Blog [mika].

Mike Flood [mikb] is another available simulation program. It combines different tools supporting flood modellers in creating different kinds of flood situations. In this context, it does not matter if the flood happens at coastal or riverine areas. It is useable in various situations and works under different environmental conditions. The application fields of Mike Flood are manifold. It can be used to forecast floods and determine managing actions to mitigate the floods. Causes and consequences of dam breaches or flood defense failures can be studied. Additionally, Mike Flood can be used to do a risk analysis and create flood hazard maps, as visualized in Figure 2.2, for different regions. It is also possible to create evacuation plans and consider different infrastructures within the flood modelling [mikb].

When dealing with floods, different tasks can be done in order to prevent or mitigate the adverse effects of them. These tasks are influenced by various parameters, which have to be considered by the decision makers during flood management. Based on the high number of variable information affecting the impact of a flood, frameworks have evolved with the goal of helping decision makers in their tasks. DESMOF (**DE**cision **S**upport for **M**anagement **O**f **F**loods) is such a decision support system, developed in 2005 [AS06]. One functionality of the system is to support the user with an appropriate option depending on the area to reduce possible flood damages. The prediction of floods constitutes another main capability of the system. The forecast serves as an important information source for flood management in general, but also to reduce life losses and property damages. Another functionality of the system is the simulation of structures to control the flood. In order to influence the simulation result, different parameters are provided to modify the simulation [AS06].

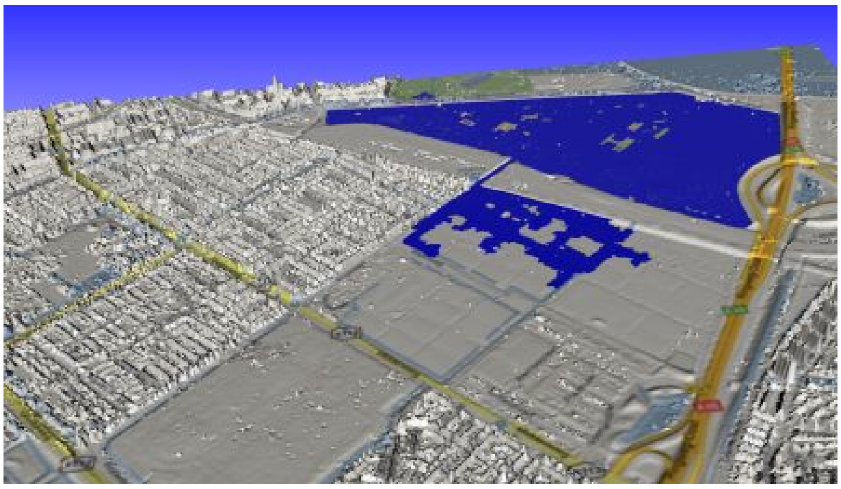


Figure 2.3: 3D Visualization of the Science Park in Amsterdam from UrbanFlood. Image taken from the project website [urb].

In 2011, an Early Warning System (EWS) named UrbanFlood [KSM+11] has been developed. Figure 2.3 shows a visualization of the Science Park in Amsterdam created with this application. The main focus of the system is to detect flaws and abnormalities of flood protection mechanisms which are already in use. Based on detected signals, that are given by integrated sensors of the defense system, probabilities of a possible failure can be calculated. These probabilities are the input source to do a simulation of possible scenarios, that might happen in case of a real failure. The results of the simulation form the basis for the interactive decision support system to support decision makers in fast reacting to failures of protection mechanisms [KSM+11]. At the same time another decision support system was proposed and is integrated into the framework of ArcGIS [QA11]. The system is based on a two-dimensional flood simulation and uses GIS features in order to calculate flood damage and estimate loss of life [QA11].

Figure 2.4 shows a visualization created by another decision support system called Visdom [vis]. Visdom combines flood simulation with analytic tools and different visualization methods to form one powerful software project. The system is not limited to only create one single simulation scenario, but to create multiple ones, which can be compared. This functionality builds the core component of the system. Each scenario can consider different parameters producing alternative simulation results. Through evaluating the results, appropriate mitigation plans can be established. While the simulation calculations are computationally very intensive, an interactive visualization of flooding events has to be provided. This is manageable by exploiting the computational power of modern GPUs [HPW+16] and optimizing the implementation of the solution of 2D shallow water equations. Additionally, the 3D visualization of the scene and different interaction techniques make the software usable for domain experts and non-experts. To evaluate the amount of possible damage, color codes are used to color the different objects within the flooded area. Further information can be viewed by selecting an object of interest for a detailed analysis. If adding protection mechanisms, the system does not only calculate the functionality, but also the quality of the placed barrier. The flood simulation is the most prominent application area of Visdom, but it is not restricted to this domain. Another use case of Visdom is to support logistical purposes like giving information about desired resources for barrier placement or how long it will take to build the protections [WKS+14].

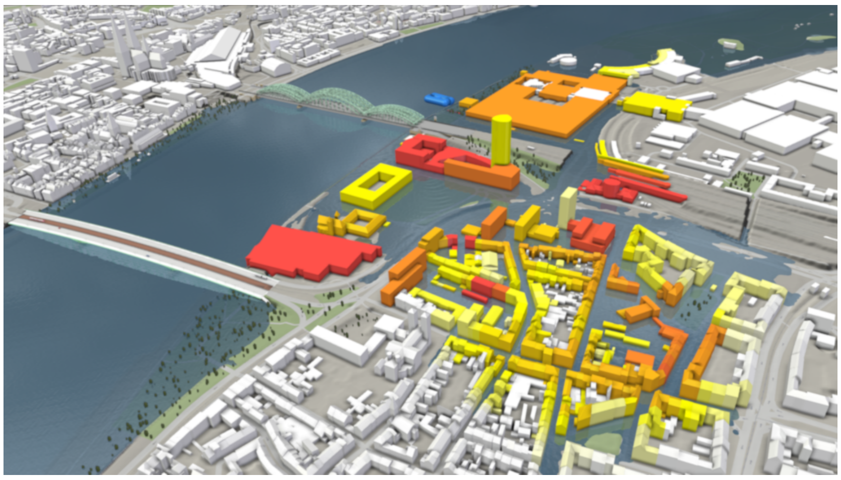


Figure 2.4: Visualization of a dam breach scenario produced by Visdom [vis].

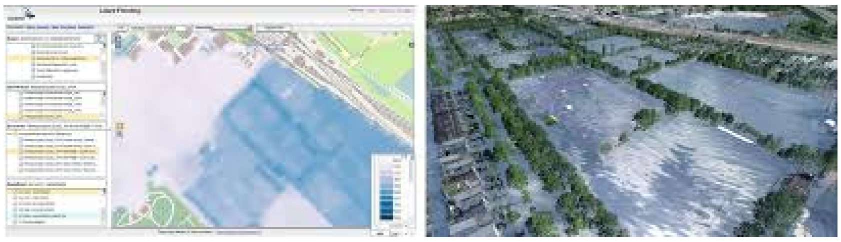


Figure 2.5: A visualization of a simulated flood hazard from two different perspectives. Image taken from Leskens et al. [LKT+17].

Leskens et al. [LKT+17] presented another decision support system for flood management. In their work they focus on creating a system that is suitable for domain experts as well as for practitioners. Based on the fact that the two user groups need different information for flood preparation, they developed a realistic 3D visualization. Figure 2.5 shows a visualization produced with their system. The main functionality of the system lies in simulating flood events based on heavy rainfalls or dam breaks. During the simulation process the user is able to change different infrastructural parameters like the elevation of the terrain, the type of land use or even adapt the canal system [LKT+17].

2.3 Dynamic Changes

When rendering a 3D scene, the underlying framework has to deal with different kinds of changes within the scene. The main extremes are either a fully static scene, meaning that no movement of the individual scene objects happens, or a fully dynamic scene, where everything changes between two consecutive frames. In the case of flood simulations, the spread of the water over the terrain is the most frequently occurring change between two time steps. Because of this dynamism in the simulation and the high importance of the water flow, the recalculations have to be done in an efficient way.

A possible solution to efficiently update simulation data is to use data-flow graphs [UFK+89]. Data-flow graphs are typically designed as graphs containing nodes as modules and directed input and output edges as input and output data. The system reacts individually to changes of connections as well as to adding and removing modules. Thus, the system can be tailored to specific needs depending on the situation. Through a demand-driven data-flow, each module only performs recalculations if the input data have changed and the output data are requested. This makes the calculation process more efficient [SWR+13].

Not only the data communication has to be done efficiently, but also the data rendering. A variety of graphic engines exist, each of them supports different functionalities. Based on the requirements of the developed application, the most appropriate one has to be selected. Every rendering engine implements some kind of optimization, like filtering out redundant draw calls. Depending on how this filtering is implemented, it can also lead to wasted calculation cycles. Additionally, the basic rendering engines continuously send every draw call over and over again for each frame, even if nothing has changed, instead of reusing already performed calls [HSMT15].

In order to save calculation and rendering time, an alternative option is to distinguish between unchanged and changed rendering calls between consecutive frames and only update those, which have changed. This type of optimization is similar to the introduced data-flow graph, where only the changed data are updated on demand. As in flood simulations only specific parts of the scene change, the exploitation of identical rendering calls in successive frames can increase the overall performances. In order to identify where changes happen in a scene graph, one option is to use a dependency graph like it was introduced by Wörister et al. [WSMT13]. The main task of a dependency graph is to track the changes made in a scene graph and map it to the associated rendering cache. The rendering cache provides information about the corresponding rendering calls to visualize the scene graph and can explicitly track input dependencies. As the performed changes are tracked, they can be used to efficiently update only the minimal necessary set of GPU resources. Thus, the rendering cache does not need to be destroyed and recreated if something changes. In addition, the updates are only performed for those scene-graph nodes that are visible at all. This lazy incremental update method only works for data changes, but not for structural changes of the scene graph [WSMT13].

As an extension to this limitation of the dependency graph, Haaser et al. [HSMT15] use the concept of a virtual machine (VM) that works on top of the graphics APIs. They use a dependency graph to convert a basic scene representation to a simpler representation. With this simplified description, it is possible to handle data changes as well as structural modifications incrementally. Based on the simple scene description, abstract machine code is created and optimized by removing duplicate code and ordering it by its state. In order to omit the expensive interpretation of abstract machine code, the abstract optimized instructions are directly compiled to executable machine code. This forms a good basis for an efficient high performance rendering engine [HSMT15].

In order to create the VR system proposed in this thesis, the above introduced concepts are used. The concept of the demand-driven dependency graph [SWR+13] is used within the simulation framework for fast data communication. The incremental rendering VM [HSMT15] is used for rendering the simulation data efficiently.

2.4 GPU Optimizations

In addition to efficiently perform scene updates, further improvements can be made by utilizing scene understanding and domain knowledge to improve the rendering performance. Similar to applications with high performance requirements, high and stable frame rates as well as low latency are of special importance for immersive VR applications [WS98], [Kol95], [vrsb]. One important factor is to use already known information about the underlying data before sending it to the GPU. Even though a rendering engine would also be able to analyze and exploit the structure of the data, this analysis process can be skipped by adding prior domain knowledge.

One of the most important optimizations is to render a huge number of instances of the same geometry with different transformations in one draw call. This is possible with prior domain information. Therefore, a special draw command called *GL_ARB_draw_instanced* has to be used. For the draw call not only the geometry information has to be passed to the GPU, but also a buffer storing all the transformation matrices, also called the instance buffer. Instead of manually iterating over all transformations and drawing the same geometry information with a different transformation matrix, the GPU takes on this task. Through shifting this iteration to the GPU, the performance is increased by fewer API calls and by reducing the amount of redundant data [ins].

Another possibility to improve rendering performance is to use indirect rendering. The basic concept of indirect rendering is to store the parameters to draw an object in buffers

[dra]. An extension to the basic indirect drawing is multi-draw indirect [mul]. The main objective of this extension is to pack multiple scene objects, sharing the same buffers, together and issue one draw call to the GPU. The main advantage of the commands is that the GPU is able to directly manipulate the parameters of the buffers. This saves time because the data do not have to be manipulated on the CPU and uploaded again to the GPU, so the CPU-GPU interaction can be reduced. Additionally, the data doesn't have to be recreated on the CPU. As for this thesis the rendering time is crucial, the usage of a high performance rendering engine is combined with suitable graphics commands depending on the underlying data. This implementation strategy helps in achieving the desired rendering performance.

2.5 Virtual Reality Applications

Visualizing a 3D world on a 2D screen can limit the view on certain properties within the scene. Therefore, different interaction techniques exist to change the point of view. To overcome this limitation, a 3D virtual world can be used to visualize the real 3D world. VR applications are not all about games but are also used in the fields of education, sports, business, or scientific visualization [vrsa]. The potential of using VR for disaster management has already been discovered by several parties, and will be discussed in the following.

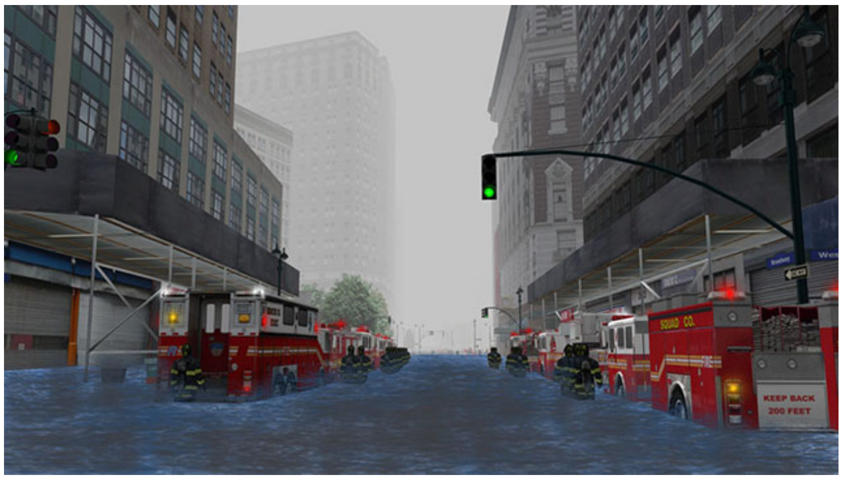


Figure 2.6: Visualization of a flooded scene in ADMS. Image taken from the project website [adm].

In the United States, already a powerful simulation training system is in use for years. It is called Advanced Disaster Management System (ADMS) [adm] and Figure 2.6 show an image from the system visualizing a flooded scene. ADMS is a modular system and its training capabilities reach from natural disasters to fire fighters as well as the police, to name a few fields. In the training environment, the user is able to slip into different roles like an emergency coordinator, incident commander, or vehicle operator. Additionally, the training purpose can vary from just receiving an alarm and driving to the event to response training or recovery or even debriefing the whole operation. Although the system comprises a variety of application fields for disaster management, no detailed description about the possibilities of specific actions in the flooding application can be found. The whole framework is a very powerful and complex system and can be tailored to the desired needs by just installing the appropriate modules. The system has to run on its own station and can either be used in full immersion theaters, projector screens, or computer screens. By connecting multiple stations, team trainings are available and optionally additional modules for driving purposes can be connected [adm].

Although technology is getting better over time, deciding when is the right time to inform civilians of a city about a possible upcoming flood is not an easy task. Based on that finding the Red Cross/Red Crescent Climate Centre developed a VR game [Goe16] to show people the tasks that have to be done by decision makers in case of a possible flood. In the game, the user is standing on a hill and has to observe the water level of the Nangbeto hydropower dam. His/her task is to decide when to ring a virtual bell to inform threatened people and to stamp the desired papers in order to get aid delivery and other sources. If the decision was made too early, the user gets warned with a stern voice about wasting money for approving aid for no reason. Additionally, the program shows possible improvements to the user in predicting possible future floods based on past experiences. Based on that, triggers can be extracted to speed up the emergency funding process to get aid for people in danger [Goe16].

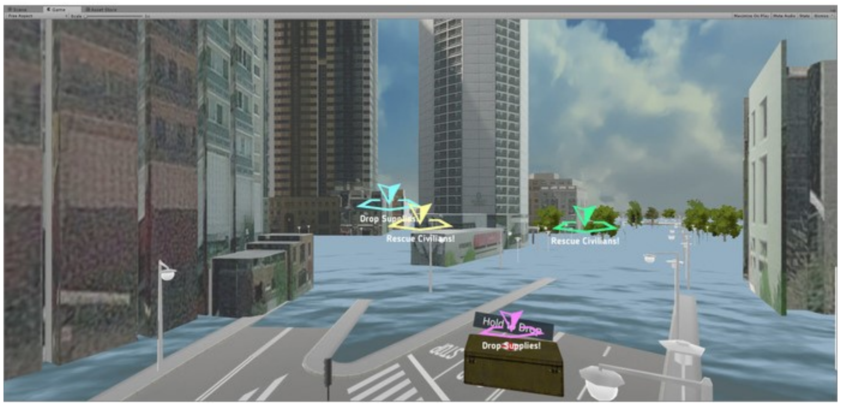


Figure 2.7: Visualization of an emergency supply delivery in Flood Action VR. Image taken from the project website [SD18].

Another VR game for flood simulation is Flood Action VR [SD18]. The goal of this game is to raise public awareness of the causes of a flood. Additionally, responsible personnel can use it for training purposes. The users are able to select different cities and current or historical weather conditions at the beginning of the game. After the selection is done, the appropriate scenario with the selected weather condition gets loaded.

During the simulation, the user has to fulfill different tasks like finding some objects, rescuing civilians, or escaping from the flooded area. Figure 2.7 shows the delivery of an emergency supply in the game. The tasks can also be done in a team, i.e., the game can be played as a multi-user application. The interaction with the VE works by giving voice commands and is also used to communicate with the remaining players. Furthermore, these commands are analyzed concerning the performed actions and how they are spoken to derive the emotional and psychological state of the users. Flood Action VR can be used with a Samsung Gear VR, but can also be adapted to other VR headsets [SD18].

In order to determine the influence of placed protection barriers to the water flow behavior of flooded regions, Winkler et al. [WZR17] introduced another type of VR application. The main purpose of the application is to use it as a communication tool to the public to raise the awareness of possible flood incidents. Furthermore, it can help to understand how protection mechanisms manipulate the water flow. The user is able to either look at a local flooded area of a village, as illustrated in Figure 2.8, or at the flooding of a subway station. Both scenarios are introduced in the paper. Additionally, the user is able to redirect the water flow by a predefined set of barriers [Inn17]. The introduced scenarios in the work of Winkler et al. show only a small scale simulation domain. No examples are given for simulation domains of greater size, like they are used in this thesis.

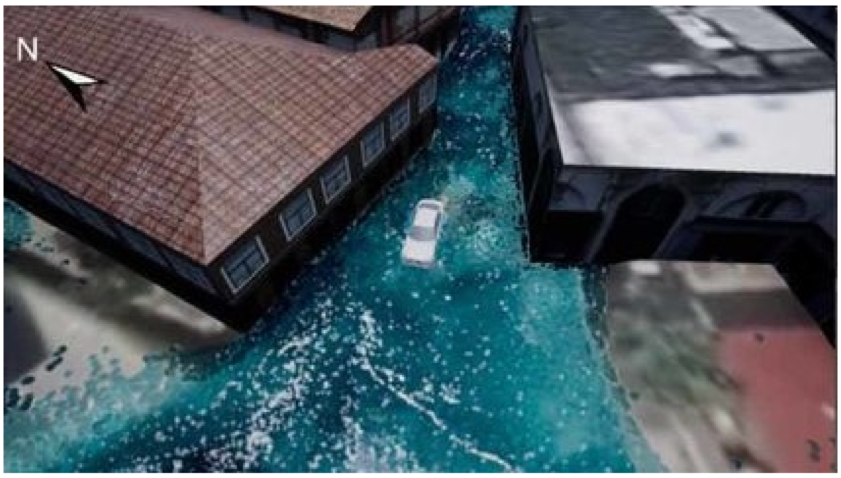


Figure 2.8: Visualization of a flooded area created by the introduced work from Winkler et al. [WZR17]. The image is a sub-image from their publication [WZR17].

In flood management, not only temporary protection mechanisms can be used, but also permanently installed ones for long-term protection. Flood managers and designers have to work together to determine how the protection mechanisms best fit into the urban landscape while fulfilling the desired task. This planning mostly happens on static miniature models, but virtual environments are a promising alternative of a more flexible planning environment [THF06]. An example of such a planning application is the CityEngine VR Experience [Ari18]. In the application, a traditional planning office is simulated showing a miniature model of a city as visualized in Figure 2.9. The users are able to move freely around and within the model. Different perspectives provide the possibility to analyze how new buildings fit into the city landscape. Multiple users can view different scenarios and compare them. As the user groups are not limited to city planners and designers, citizens can be integrated into the planning process.

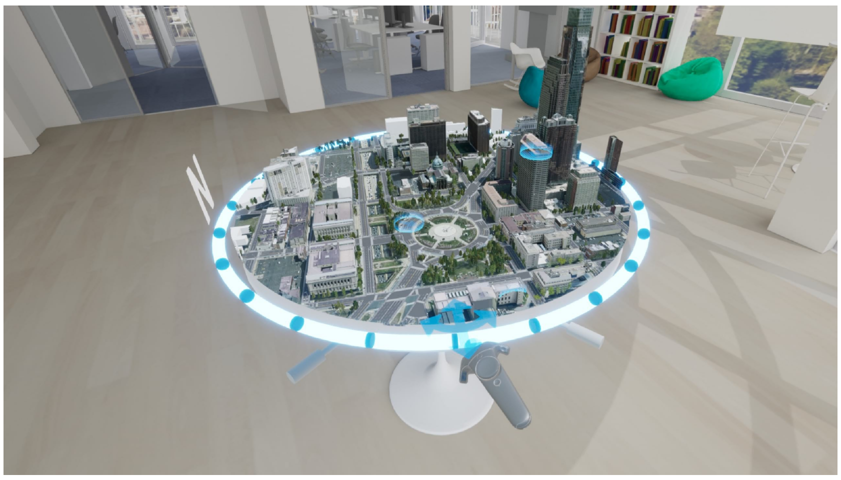


Figure 2.9: Image showing the CityEngine VR Experience application. Image taken from the project website [Ari18].

The presented VR applications for flood simulation are limited to one scenario. As no further information is given, the user has no control in influencing the simulation parameters. With the implementation of this work, a VR application is presented that is able to steer multiple scenarios of fast simulations. The user is able to switch between different scenarios and go back and forth in the simulation progress. New protection mechanisms can be placed into the scenario and are considered in the ongoing simulation progress. Additionally, two users can train together with different flooding events by using the Operator-Trainee setup.

2.6 VR Interactions

In order to interact with a VE, different interaction techniques have evolved. One of the first techniques is the Bat [WJ88]. It is based on the concept of the PC mouse, but is held in the hands. Thus, six degrees of freedom (DOFs) are available. The Bat allows the user to select an object in the virtual world and manipulate it. The manipulation of the selected object is possible by a linear mapping of the transformation of the Bat. As this kind of object selection and manipulation is limited to objects within the user's reach, extensions have been made. One extension is to use a ray (light beam) [Min95] to point to distant objects and select them for manipulation. The selected object sticks to the ray. This limits the DOFs. As an alternative serves the Go-Go technique

[PBWI96]. It virtually extends the length of the user's hand, so he/she is able to reach and manipulate distant objects. The manipulation of the objects is again a linear mapping of the user's hand transformation. An hybrid method called HOMER [BH97] combines the benefits of the ray-directed selection and Go-Go object manipulation. As an object under consideration is selected, it sticks to the hands of the user and reacts according to the hands' transformations.

The "World in Miniature" (WIM) metaphor is another kind of interaction technique [SCP95]. The basic idea of this interaction method is a smaller representation of the whole scene, which can be held in the hands. The scene objects can be manipulated by either altering them in the miniature representation or in the life-sized virtual environment. An additional interaction hardware like the Bat is used to select the different objects in the virtual world. Another possible interaction method is to use hand gestures. The user's hands can either be tracked by employing certain gloves, like presented by Cutler et al. [CFH97], or by a certain sensor like Microsoft's Kinect [SGH$^+$12]. Through recognizing different kinds of hand gestures, different interactions can be done.

The used VR hardware in this thesis is the HTC Vive [HTCb]. This hardware setup includes two controllers to interact with the virtual world. Depending on the performed interaction, different interaction metaphors are used. The interaction possibilities are described in Chapter 4 in more detail.

2.7 Virtual Menus

The presented interaction methods serve also as a selection tool in menus. As for 2D desktop applications, menus in the VE are used to provide some state control within the scene.

The floating menu is basically just taken one-to-one from the 2D screen into the 3D world [JE92]. Depending on how the floating menu approach is implemented, the list of menu items is displayed in front of the user. With the use of a ray, an item on the list can be selected and the menu disappears. Another menu type is pen & tablet [LSH99]. For this approach an additional physical device is used and gets tracked in the VE. The user has to hold some kind of object in his non-dominant hand, which should give the impression of holding a tablet or another physical entity. On this object the menu is displayed. With his/her dominant hand the user is able to select a menu item of a set of the displayed menu entries.

Bowman and Wingrave compared these two menu techniques and developed the "Three-Up, Labels in Palm" (= TULIP) menu system [BW01]. For this kind of menu, the user has to use special gloves as input device. These gloves are tracked and recognize the motion of the users hands. In general, the non-dominant hand and the dominant one have different meanings. The non-dominant one gives the overview of the menu. Therefore, each finger has one menu title assigned. Only the thumb has no entry assigned. With the thumb, a menu item can be selected by pinching it together with the finger having the appropriate title assigned. The dominant hand has the menu items of the selected menu title assigned. Again by pinching the thumb with the desired menu item, the selection can be done. With this kind of menu only four different menu items would be available. As a consequence, the basic menu selection technique has been extended. If more than four menu entries are available, the palm lists further entries. By pinching the thumb with the pinky finger, items from the list on the palm are assigned to the three residual fingers and can be selected.

As an alternative to the listed VR menu techniques, it is also possible to use speech recognition [MA96]. By using special voice commands, the user is able to directly speak with the VR application. The main advantage of this type of menu interaction is that the user does not have to navigate through a nested 2D menu in a virtual world to execute a desired task. Instead, he/she only has to voice the appropriate command, which will trigger the system to execute the same action as it would have been selected from a set of options. Furthermore, without using a 2D menu in a 3D world, the feeling of presence does not get impaired. However, the speech recognition system is limited to the underlying vocabulary and to the recognition rate of the used system to identify the spoken commands right.

Another possibility is to place special objects in the virtual world, which act as menus. As an example serves the game "Fantastic Contraption" [Nor]. In the game, a cat serves as menu. It carries different kinds of tools, which can be picked by the user. The selected tools can be used to interact with the environment or build vehicles. Because the user is able to move through the virtual environment, the cat follows the user wherever he/she goes. This has the advantage that he/she does not have to walk back to a specific place if something is needed from the tool palette. The menus implemented in this thesis are influenced by the WIM concept and a rotational list of items.

2.8 Moving through Virtual Space

In order to explore different regions of the simulation domain, the support of virtual movement techniques is needed. Exploring the VE by physically walking through space is very limited by the real tracking space of the VR system. As a result, different movement techniques have already evolved in 1990 to investigate larger virtual spaces. "Flying" is one of these movement techniques, which is still in use in today's VR applications. The main idea of flying is that the user steers his/her view point through the scene with a certain velocity [WO90].

The steering can either be controlled by the user's gaze, called gaze-directed steering, or by pointing. By employing pointing, the desired movement direction is obtained by the user's hand orientation. The main advantages of pointing-directed steering compared to the gaze-directed approach are a more comfortable exploration of the VE, as well as the ability to look around in the environment while moving in a certain direction [BKH97]. On the other hand, it is more difficult to learn and a bit inaccurate [BKH97]. A similar traveling technique in VR is "teleporting", where the user can directly move to a point of interest. This point can be indicated by casting a ray into the scene. The intersection point of the ray and the scene gives the point where the user can immediately move to [BKH97].

The main drawbacks of flying around the scene or teleporting to certain interest points, are unwanted side effects like motion sickness or disorientation. As a consequence, some approaches have evolved to support natural walking. One approach is called redirected walking [RKW01]. The basic idea is to redirect the walking direction of the user if he/she is about to move outside the tracking area or collides with a physical object in the real world. This technique exploits insensitivities in the human perception and slightly rotates the scene while the user walks in space. By placing waypoints to strategic selected points, even higher rotations can be added while the user turns his/her head. With this approach natural walking can also be done in limited spaces [RKW01]. Because placing predefined waypoints to guide a user through the virtual world is not always possible or even desired, further work has been done in the area of redirected walking. With the use of distractors, an improvement of the basic redirected walking approach has been made. Every time the user is about to leave the tracking area, certain distractors appear in his/her way. These objects should prevent the user hitting a wall or other items in the real world. While he/she is following the animated distractor, the virtual scene gets readjusted depending on the head's orientation. As soon as the user is able to walk on, the distractor disappears [PFW10].

Another technique to support natural walking in VEs are flexible spaces [VKBS13]. The idea is to dynamically recombine virtual rooms by allowing the system an overlapping of the environment to fit into the tracking space. The key concept is to connect the virtual rooms with corridors which are generated on the fly when the user wants to change rooms. With flexible spaces, the user may have one way to move from room A to room B, but typically has a different route to move back from B to A. If using flexible spaces, the VE should not depend on a specific spatial layout [VKBS13]. As the layout of a city is fixed, this kind of movement technique is not suitable for this application. Additionally, the exploration of a large-scale environment would benefit from other movement techniques, which allow the user a faster exploration. A hybrid movement technique is the use of pre-oriented teleportation with WIMs and previews [ESFT17] as a traveling strategy. The WIM gives an overview of the whole city. The user is able to change his/her view point by placing an avatar to the desired destination in the WIM. As an addition, he/she is able to determine his/her orientation before the teleportation happens. This kind of movement technique makes a fast exploration of large-scale environments possible. Through adjusting the user's orientation before teleporting, the user is able to immediately work on his/her further task.

CHAPTER 3

Flood Simulation and VR Integration - VisdomVR

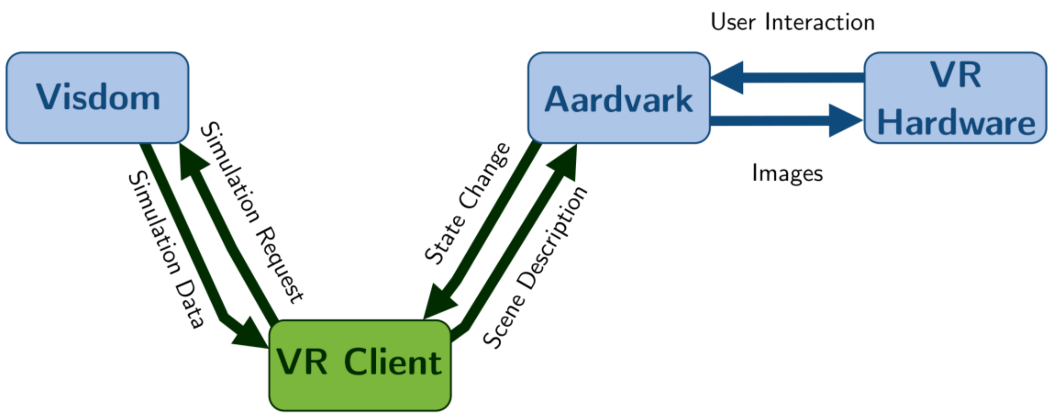


Figure 3.1: Illustration of the main system components. The blue boxes represent the already existing components. The green box illustrates the component implemented in the scope of this thesis. The labels between two components indicate the type of exchanged information. The arrows give the direction of the information flow.

VisdomVR is a system consisting of four components as visualized in Figure 3.1. Visdom and Aardvark are two independent components. Visdom is a flood simulation framework that supports decision makers in flood management. Aardvark is a high performance rendering engine visualizing data in the structure of a scene description as graphical representations. Additionally, Aardvark is able to communicate with a connected VR Hardware. Thus makes it is possible to create VR applications. The VRClient is the component implemented in the scope of this thesis and builds the connection between Visdom and Aardvark.

The four components exchange different data between each other, as labeled in the Figure 3.1. The arrows give the direction of the data exchange. In the data-flow direction from Visdom to the VR Hardware, Visdom sends Simulation Data to the connected VRClient. The VRClient processes the data and creates a Scene Description. Aardvark uses the Scene Description to create two renderable Images. These images are sent to and displayed on the connected VR Hardware. With the VR Hardware the user is able to interact with the VE. The User Interaction are retrieved by Aardvark and cause a State Change. The VRClient listens for a State Change and Depending on the state an appropriate Simulation Request is formulated. This Simulation Request is sent to Visdom. Visdom processes the request and the workflow starts over again. The result of this component integration is a VR application that is able to steer the remote flood simulation framework Visdom.

The following sections explain the different components of VisdomVR in more detail. The first section is dedicated to give an introduction to the main functionalities of Visdom. Additionally, the user interface of the main Visdom Client is explained as well as related terms. Section two describes the used rendering framework Aardvark. An explanation of the used VR Hardware is given in section three. Section four explains the main functionalities of the VR Client. The chapter closes with a description of the resulting VR application and a use case of it.

3.1 Visdom

Visdom [vis] is developed at the VRVis Research Center of Virtual Reality and Visualization in Vienna. It combines visualization, simulation, and analysis techniques in order to give support in decision making. The most prominent application concerns flood simulation and flood management. With the use of Visdom, alternative simulation runs can be created. The alternative simulation results serve as a decision input for creating protection plans in case of a flooding event [vis]. Depending on the specified parameters, different plans can be created and are stored in a pool of plans. In case of an emergency, decision makers can choose from the created pool of plans based on which one serves best for the given environmental conditions.

Visdom is implemented as a client-server architecture. The simulation calculations are done on the server. The corresponding Visdom Client is responsible to visualized the rendered simulation data. Before describing the user interface of the Visdom Client, different definitions have to be introduced, as presented by Waser et al. [WFR$^+$10]. They are incorporated as graphical representation into the user interface.

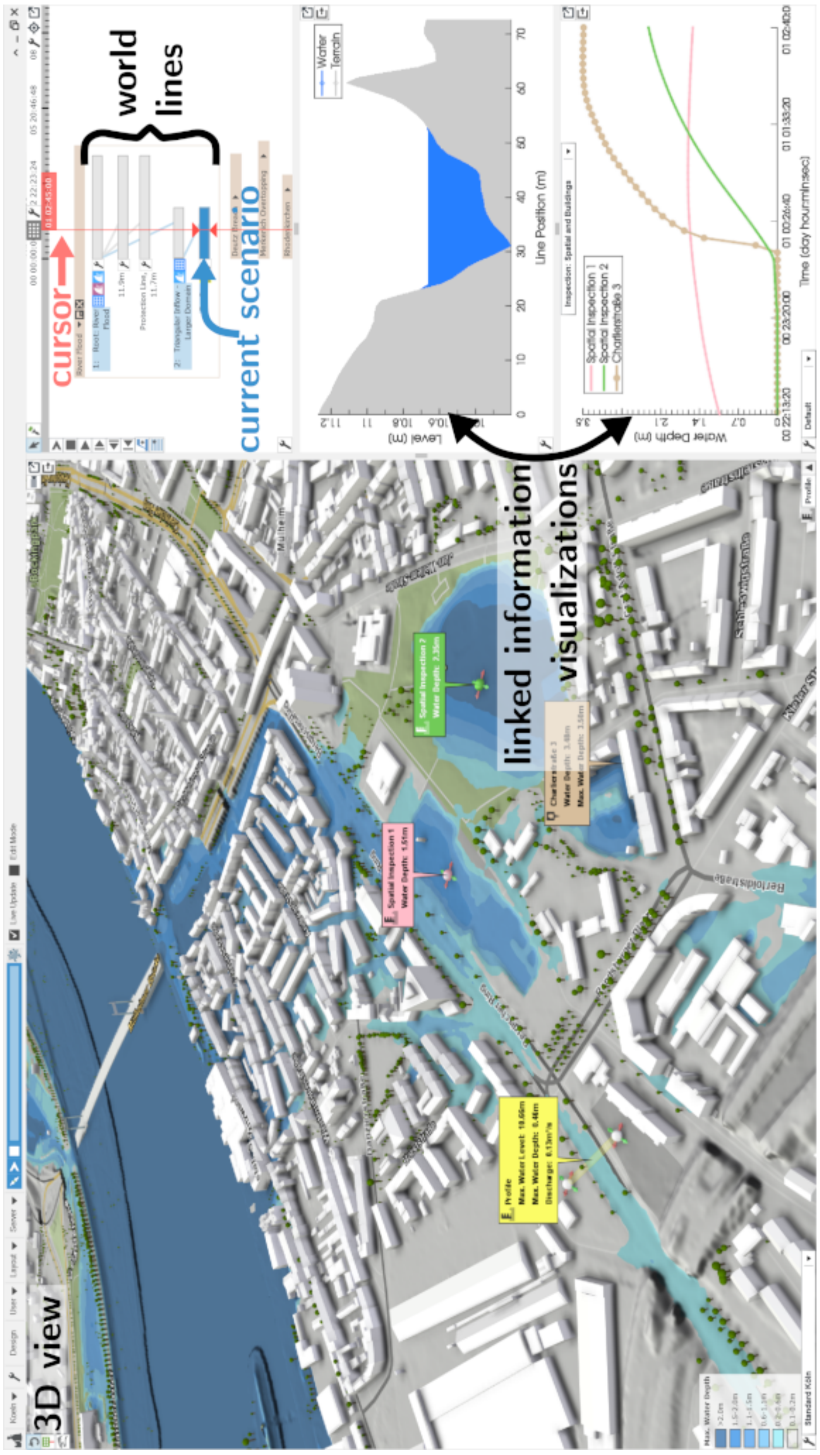


Figure 3.2: Screen shot of the Visdom Client used by flood managers [vis].

frame - a frame represents one specific time step within the simulation progress.

track - a track is a set of multiple frames. One track uses the same simulation parameters. If the simulation parameters are changed due to user interactions or other data updates, a branching of the current track happens into two. One still contains the original simulation parameters and the new track comprises the new simulation parameters.

world line - a world line (WL) consists of one or multiple causally related tracks.

scenario - a scenario is represented by one WL.

Figure 3.2 shows the Visdom Client used by flood managers to analyze different flooding scenarios. The 3D view shows the current simulation run, also called scenario or track. The right window uses WLs to visualize the different, created scenarios and shows additional information visualizations. The blue, highlighted WL is the current viewed scenario. The red cursor indicates the current time step of the visualized simulation. Visdom is not limited to the field of flood management alone, but also provides additional functionalities for other domains like logistics [WKS$^+$14]. In the scope of this thesis, only the flood simulation data are of interest.

3.2 Aardvark

Aardvark (**A**n **A**dvanced **R**apid **D**evelopment **V**isualization **A**nd **R**endering **K**ernel) [aar] is a high performance rendering engine that is used for rendering the simulation data from Visdom in a fast and efficient way [aar]. It combines several different features from the area of visual computing. The application fields range from basic visualizations to high performance graphics rendering. In order to render graphical visualizations, Aardvark needs a scene description. The scene description gives information about the objects that have to be visualized. This scene description is organized as a scene graph, as introduced by Strauss and Cary [SC92]. A scene graph is basically an acyclic graph consisting of nodes. Each node represents a different object that store certain shape information, groups, or certain properties. An entire scene is represented by multiple combined scene graph nodes. Based on the scene graph, Aardvark can render the scene correctly. The rendering engine is implemented as an incremental rendering VM [HSMT15] (see Section 2.3). Because of the efficient implementation, Aardvark can easily deal with large amounts of data and efficiently handles dynamic changes. A further functionality of Aardvark is the separation of visible and non-visible scene objects. Based on this distinction, the rendering engine only updates those objects that are visible in the scene. Due to its high rendering power, Aardvark serves as a good starting point for rendering the flood simulation data.

3.3 VR Hardware

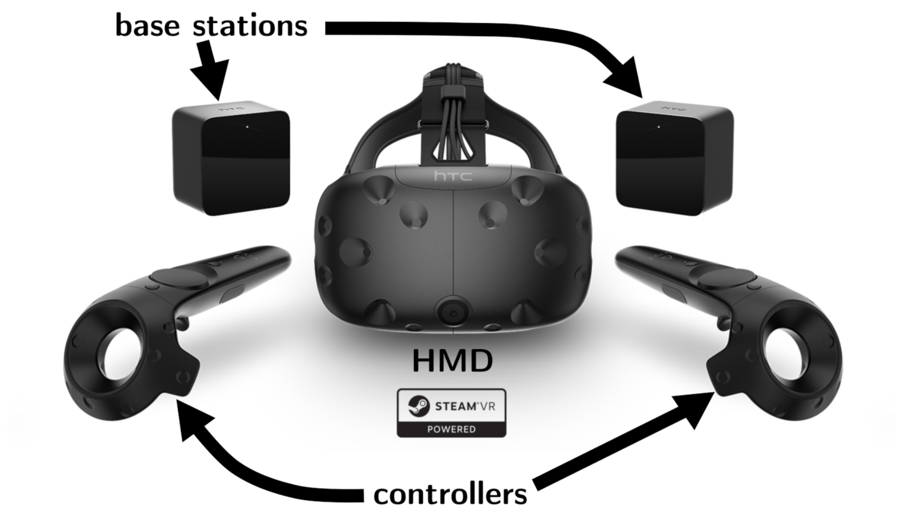


Figure 3.3: Components of the HTC Vive. Original image from Flickr [fli16] and edited by the author of the thesis.

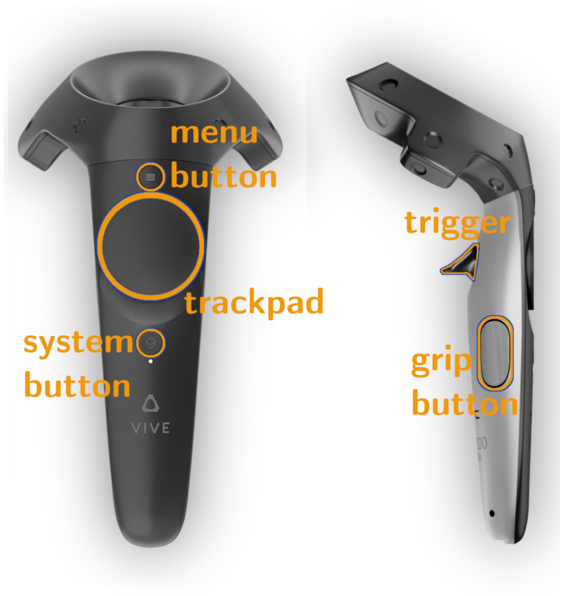


Figure 3.4: Detailed view of the HTC Vive Controller. Original image from Nick Hygens [Nic] and edited by the author of the thesis.

In order to run a VR application certain hardware is needed. In the scope of this thesis the HTC Vive [HTCb] is used for this purpose. The HTC Vive consists of two base stations responsible for the tracking, the Head-Mounted-Display (HMD), to show the virtual world, and two controllers for interaction. These components are illustrated in Figure 3.3. Additionally, Figure 3.4 gives a detailed view of the controller. The color-contoured buttons are used to perform different actions in the developed VR application. In the application, the buttons have different roles depending on the application state. These functions include, e.g., traveling in the virtual space, selecting between various options, or placing objects into the scene. Chapter 4 gives a more detailed explanation about the responsibilities of the individual buttons to perform different actions.

3.4 VR Client

The most important part to create a connection between Visdom and the rendering engine is the VR Client. The overall purpose of this component is to integrate Visdom into a virtual environment. This integration requires several different tasks that are performed by the VR Client. These tasks are:

- Implementing the communication protocol between the VR Client and the Visdom Server
- Parsing and interpreting the received simulation data
- Efficiently updating the scene graph
- Handling the VR interactions

The communication protocol between the VR Client and Visdom Server determines the structure of the exchanged messages in order to be understandable for both parties. The Visdom Server sends response messages to the VR Client and the client sends request messages to the server. The response messages give a description about the data that are sent along with the data itself. On the VR Client site, the data are parsed into the corresponding data types and stored in appropriate data objects (data interpretation). For each data object a corresponding scene graph node is created. This scene graph node again consists of different scene graph nodes containing information like the transformation, appearance and location of the object. If an object has already a corresponding scene graph node, the underlying data of this node are updated. For each scene graph node appropriate graphic commands are used for an efficient rendering. In combination with a rendering engine, that is optimized to deal with large and dynamic data, a fast scene rendering can be provided. The user is able to alter the simulation parameters through interacting with the VE. The interactions are converted to server requests. The requests cause the server to send scene updates. Depending on the type of sent request, different scene nodes are affected. Basically, the VR Client has to deal with two different kinds of updates:

per time step: the data are updated every time a new frame is requested by the VR Client.

per scenario: the data are updated every time a new scenario is requested by the VR Client.

Within this updating frequency (per time step or per scenario), individual scene graph nodes are updated. This can affect the appearance of individual scene graph nodes, or transformations as well as their location and geometry.

3.5 VR Application

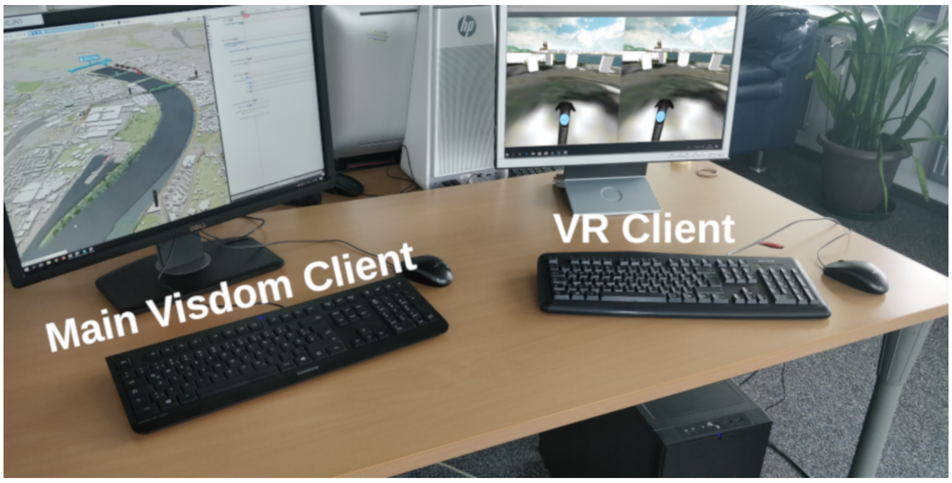


Figure 3.5: Basic Operator-Trainee setup. The operator-PC runs the Visdom Client and the other PC is responsible for running the VR application. Image has been taken by the author.

The implemented VR application of this thesis steers a remote flood simulation. Chapter 4 describes the supported actions in more detail. In general, these are:

- Moving through the virtual space
- Stepping through the simulation time
- Changing the simulation scenarios
- Placing barriers

The application can be used in a standalone setup or in an Operator-Trainee setup. With the standalone setup, only one person works with the application. The Operator-Trainee setup allows two persons to work together within the same simulation scenario.

Figure 3.5 shows the two clients used to run the Operator-Trainee setup. Both are connected to a remote server performing the main simulation calculations. The main work-flow in the Operator-Trainee setup is as follows:

The operator works with the Visdom Client application on the PC. He/she has the role of a flood manager who instructs a trainee. The trainee sees the same scenario as the operator, but uses the VR Client to view the scene. Figures 3.6, 3.7, and 3.8 show different views from the operator's perspective. Figure 3.9 shows different overview images from the trainee's perspective. Figures 3.10 and 3.11 show more immersed views from the trainee's perspective. In general, the operator has an overview of the scenario and more control in altering the simulation parameters as well as the appearance of the scenario. For example, the operator can change the texture that is mapped onto the terrain. Figures 3.9a, 3.9b and 3.9c show three different terrain textures applied from the trainee's perspective. Additionally, the operator can change the color codes of the buildings. The images in Figure 3.7 show the buildings colored by their priority from the operator's view. From the trainee's perspective the scenario looks like in the Figures 3.9 c and 3.9d. The colors range from light yellow to a saturated red. Yellow means low importance and red means high importance, like for a hospital. The white buildings have base priority. The images in Figure 3.8 show the scenario again from the operator's perspective, this time the buildings are colored by the level of damage caused by the flood. Figures 3.9e and 3.9f show the scenario from the trainee's view. The operator has to tell the trainee, what the current color codes of the scenario mean.

Additionally, he/she has to tell the trainee his/her task to perform. In this case it is to search for the hospital, colored in red like in Figure 3.6a from the operator's view, and to protect it from the flood. The trainee has to get a clear view of the flooding situation. This includes to determine when the flooding happens and where to place barriers in order to protect the building. Figure 3.9c shows the scenario from the trainee's perspective. At this time step, no flooding happened yet. As the the trainee steps through the simulation time, he/she reaches a time step, when the building under consideration gets flooded. This flooding is visualized in Figure 3.6b from the operator's view and in Figure 3.9d from the trainee's perspective.

The trainee is now able to identify the locations where the water comes to the building as visualized in the top image of Figure 3.10 at the origin of the arrows. Now he/she has to go back in time when no flooding happened to set protection mechanisms. As soon as the trainee thinks he/she is ready to place the barriers, the operator creates a new track originating from the current one (see Figure 3.7a) Now the trainee is able to place the barriers by using the provided interaction methods. Figure 3.10 illustrates this process. In Figures 3.10a and 3.10b the identification of the barrier locations is done. Figures 3.10c and 3.10d show the resulting barriers. After the placement of the barriers, the operator is also able to see the barriers within his/her view, as visualized in Figure 3.7b.

Now that possible protection mechanisms are placed, it is time to check if they hold against the flood. Figures 3.11a-f provide different temporal views during the flood animation of both barriers. In Figures 3.11a and 3.11b, the barriers are not present as the water is not close enough and no barriers are needed yet. With progress in time, the barriers are created and the local height of the barriers with the amount of construction material needed is derived from the simulation results. This progress is visualized in the Figures 3.11c-f.

In the given example the buildings are colored by priority. With this color code it is difficult to determine if a building got damaged by a flood. For this purpose the color codes have to be switched to encode the level of damage. Figure 3.8a shows the flooded scenario from the operator's view at the same time step as Figure 3.6b with the difference that the buildings are colored by the level of damage. The trainee's view is visualized in Figure 3.9e. Figure 3.8b shows the flooded scenario with the placed barriers from the operator's view. It shows the successful protection of the building, as it is now colored in base color. From the trainee's perspective this is visualized in Figure 3.9f. In general, the coloring of the buildings by priority is useful for identifying important buildings to protect. The coloring of the buildings by the level of damage is more useful in determining the impact of a flood on the urban infrastructure.

The next chapter explains the VR interactions in greater detail. Chapter 5 gives more implementation details of the following topics:

- The sent data
- The communication protocol
- Data parsing and interpretation
- Maintaining the scene graph

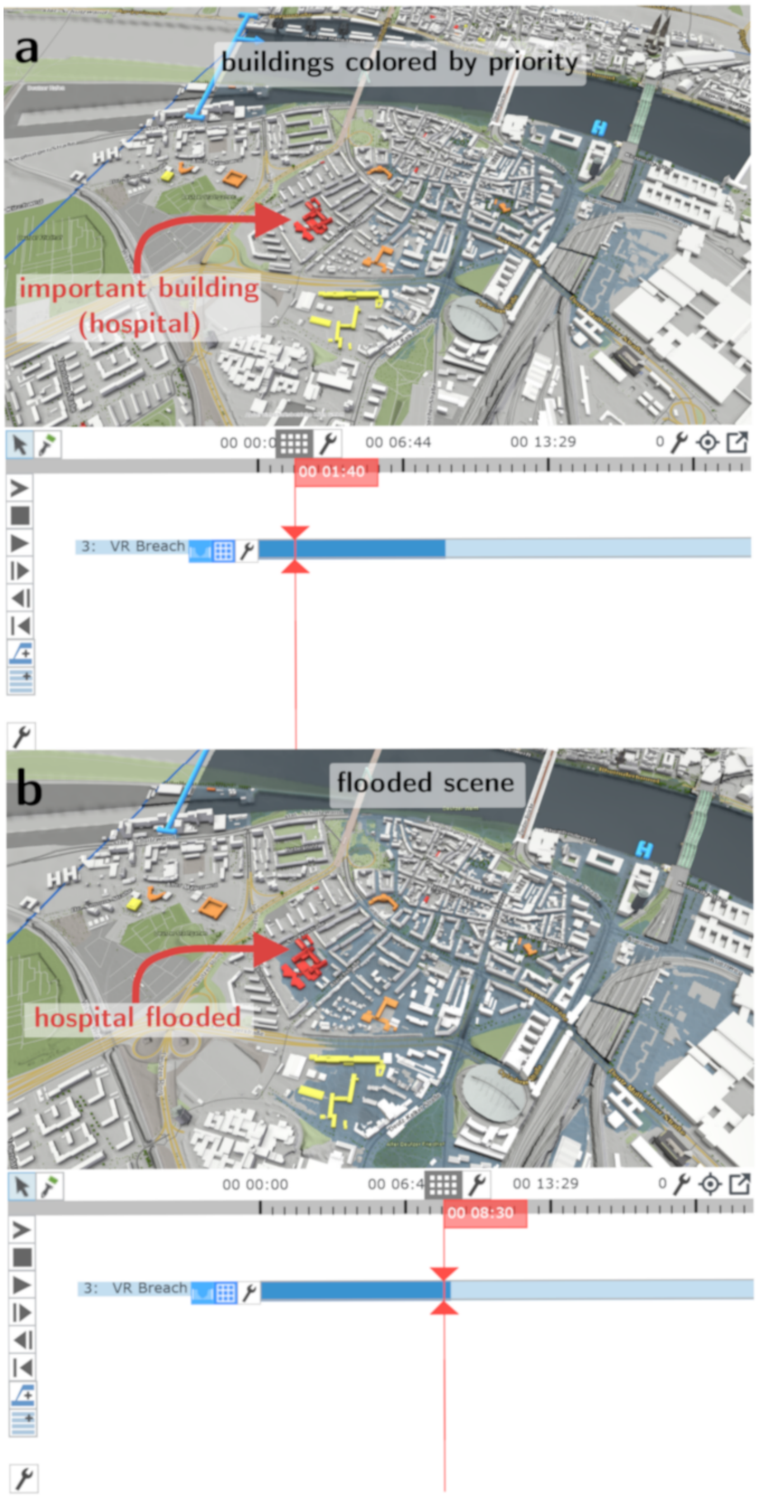


Figure 3.6: Screen shots of Visdom Client from the operator's view. In both scenarios the buildings are colored by their priority. (a) shows an overview of the flooding scenario. At the displayed time step, no flooding has happened. (b) shows the scenario at a later time step. At this time, the flooding has already happened.

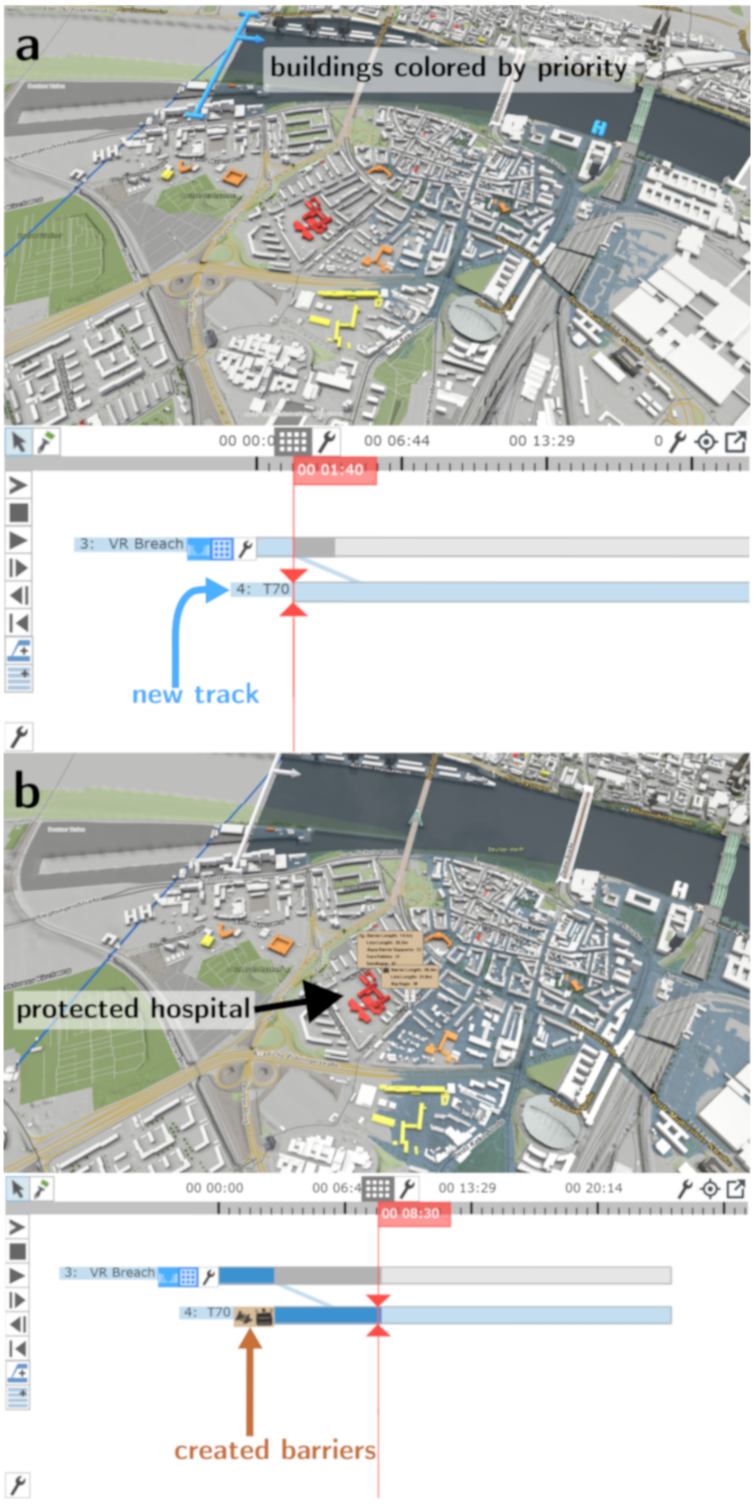


Figure 3.7: Screen shots of the Visdom Client from the operator's view. (a) shows the scenario at the same time step as Figure 3.6a, but with a new created track. (b) shows the flooding scenario, but this time protection barriers are placed to protect the hospital.

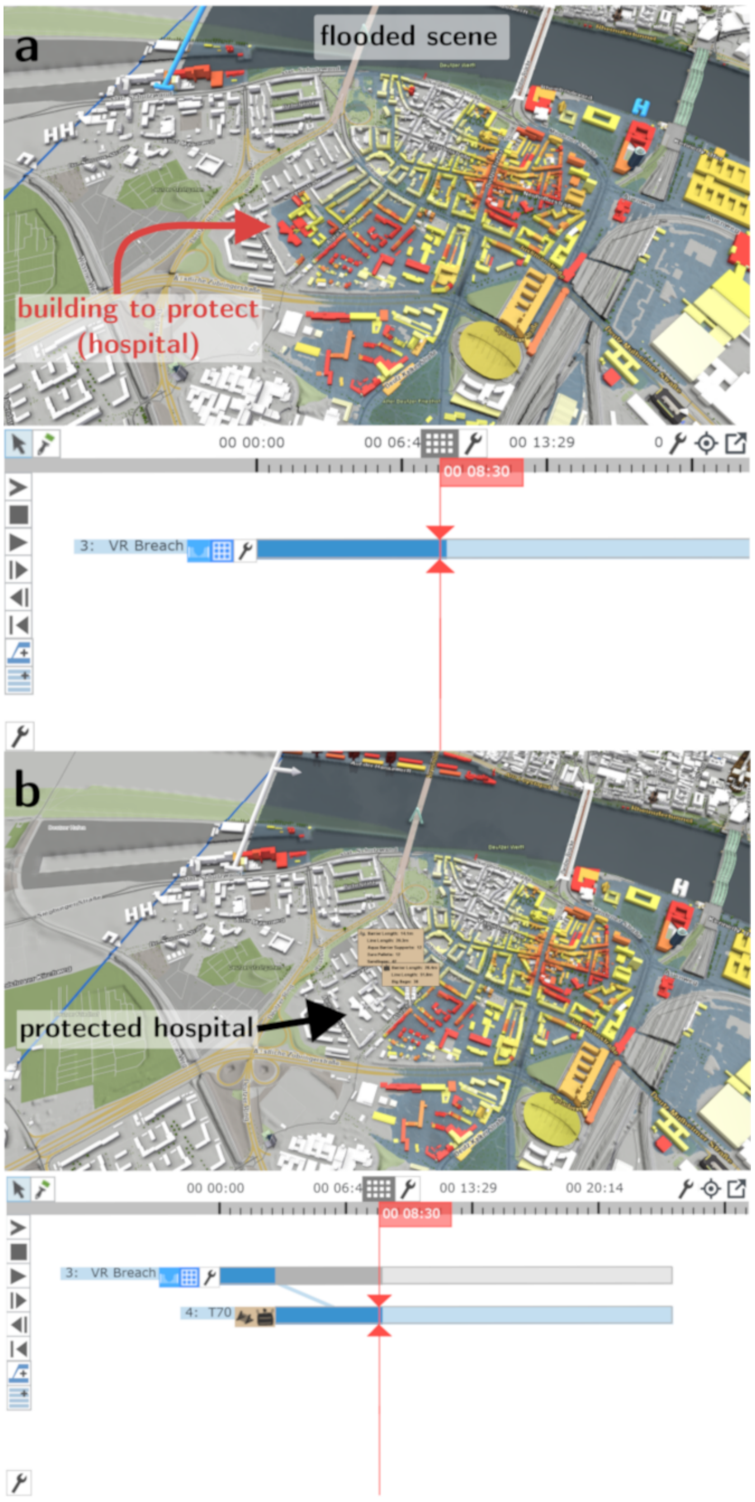


Figure 3.8: Screen shots of the Visdom Client from the operator's view. Both images show the buildings colored by their level of damage. (a) shows the flooded scenario like in Figure 3.6a. (b) shows the same flooding scenario, but protection barriers are placed to protect the hospital. As the building under consideration has now base color, it has not been damaged by the flood.

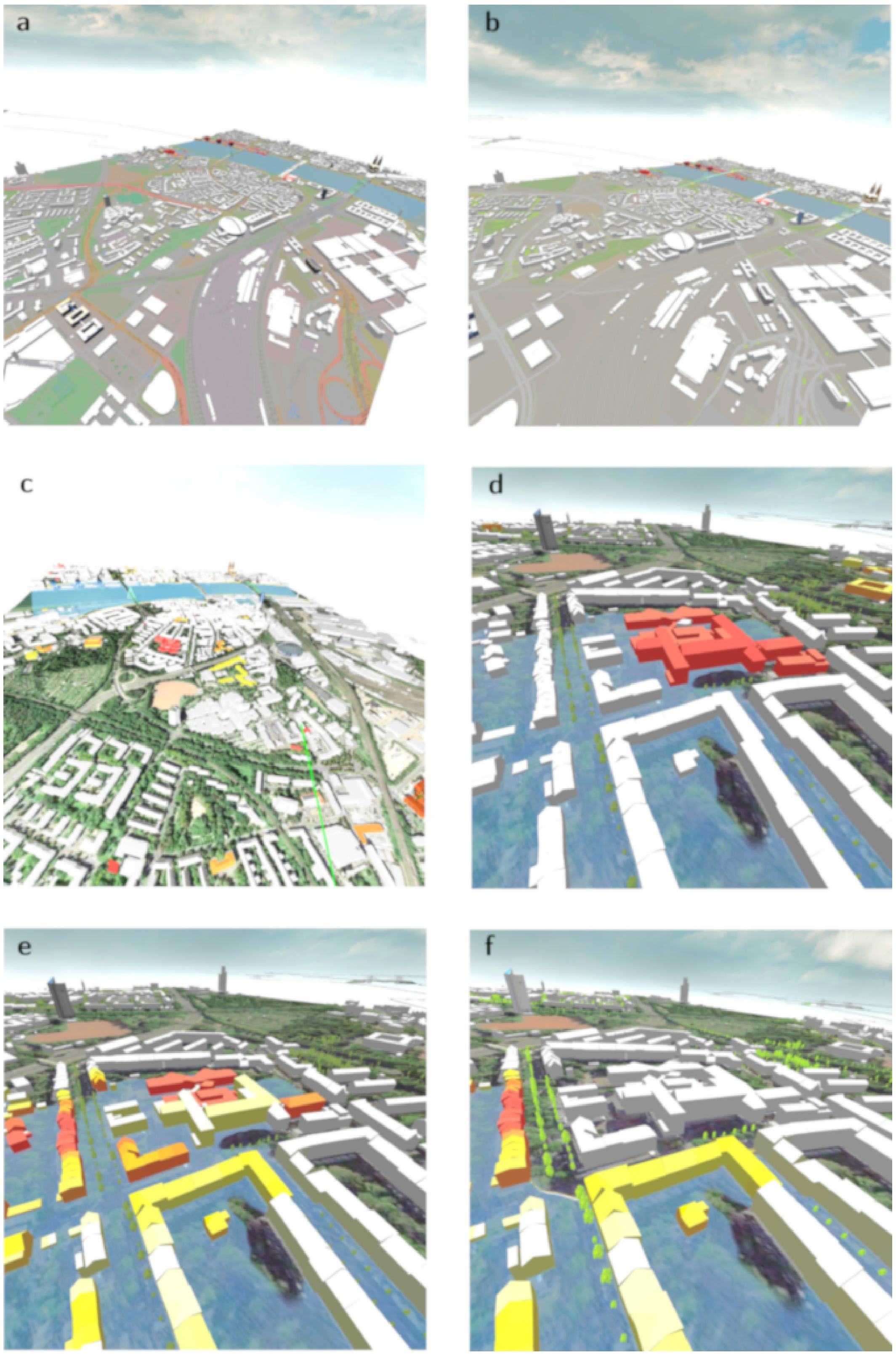


Figure 3.9: Overview images of the scenario visualized in Figure 3.6 from the trainee's point of view. (a), (b), and (c) show the scenario with different terrain textures applied. (d) shows the flooded hospital colored by its priority. (e) and (f) show the buildings colored by the level of damage. In (e) the hospital has been damaged by the flood. In (f) the hospital has been protected from the flood by placing barriers.

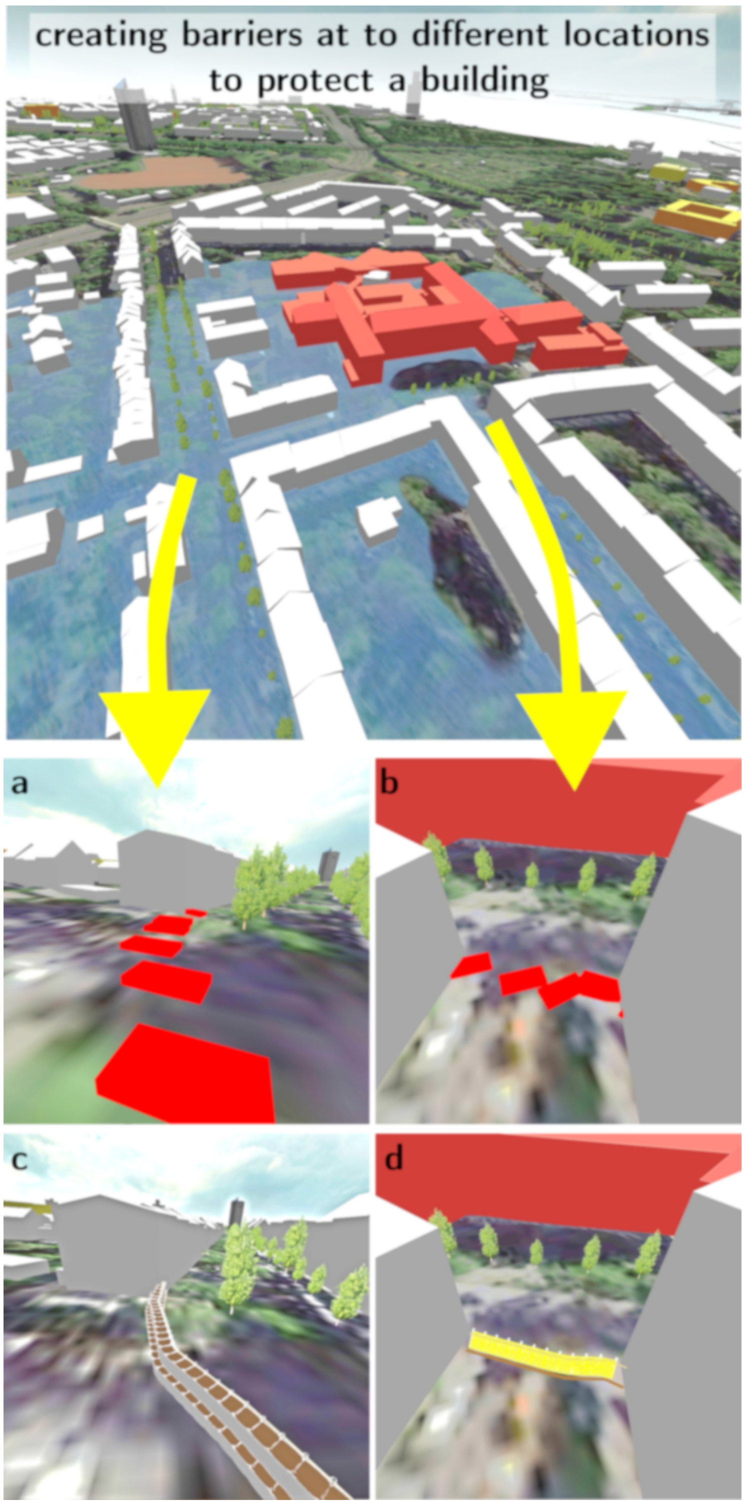

creating barriers at to different locations to protect a building
a
b
c
d

Figure 3.10: Barrier creation at two different places from the trainee's perspective. (a) shows the set control points for the first barrier. (b) shows the set control points for the second barrier. (c) and (d) show the resulting barrier based on the set control points.

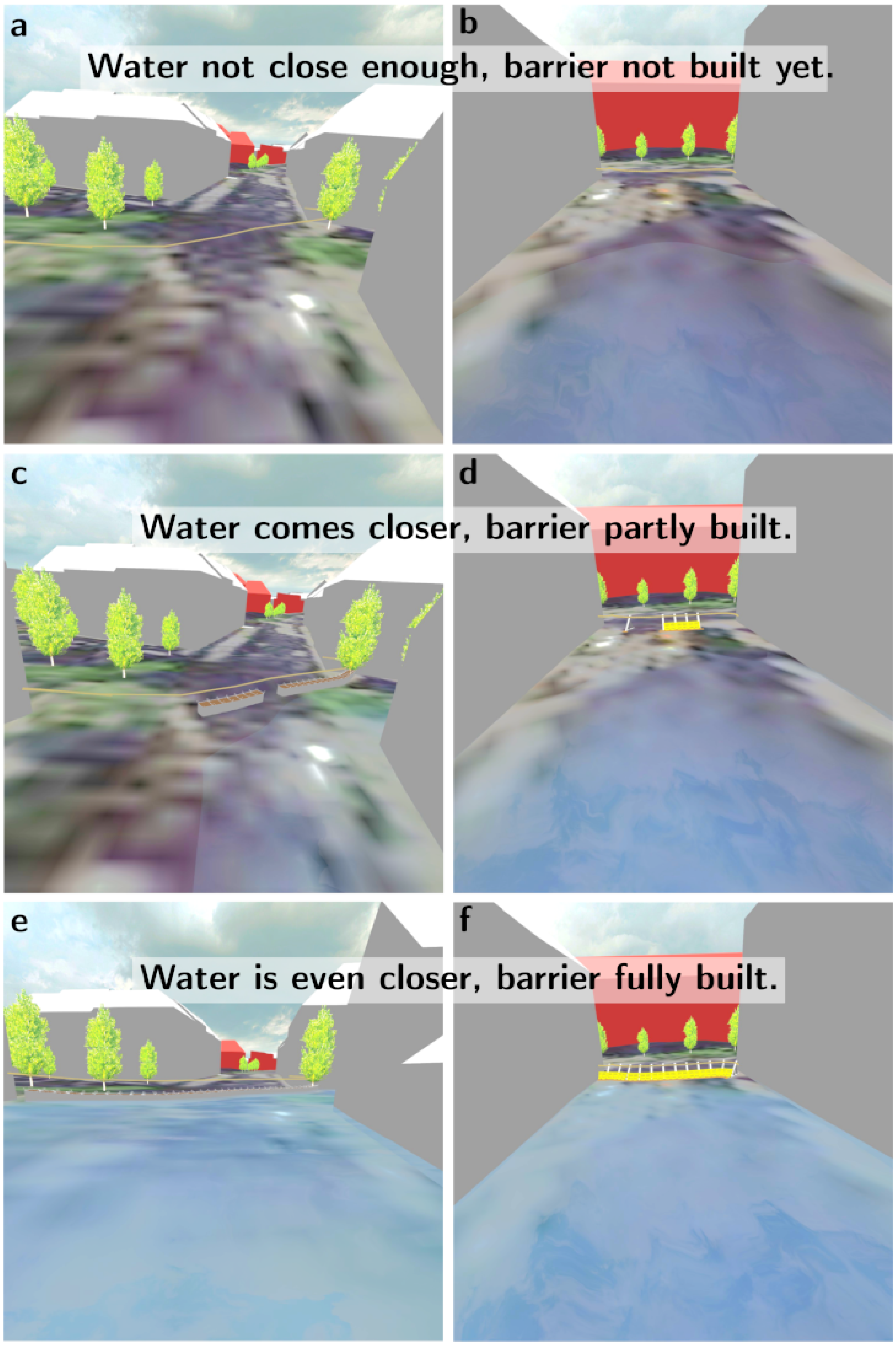

a
b
Water not close enough, barrier not built yet.
c
d
Water comes closer, barrier partly built.
e
f
Water is even closer, barrier fully built.

Figure 3.11: Different time steps of a flooding scenario from the trainee's perspective.

CHAPTER 4

VR Interactions

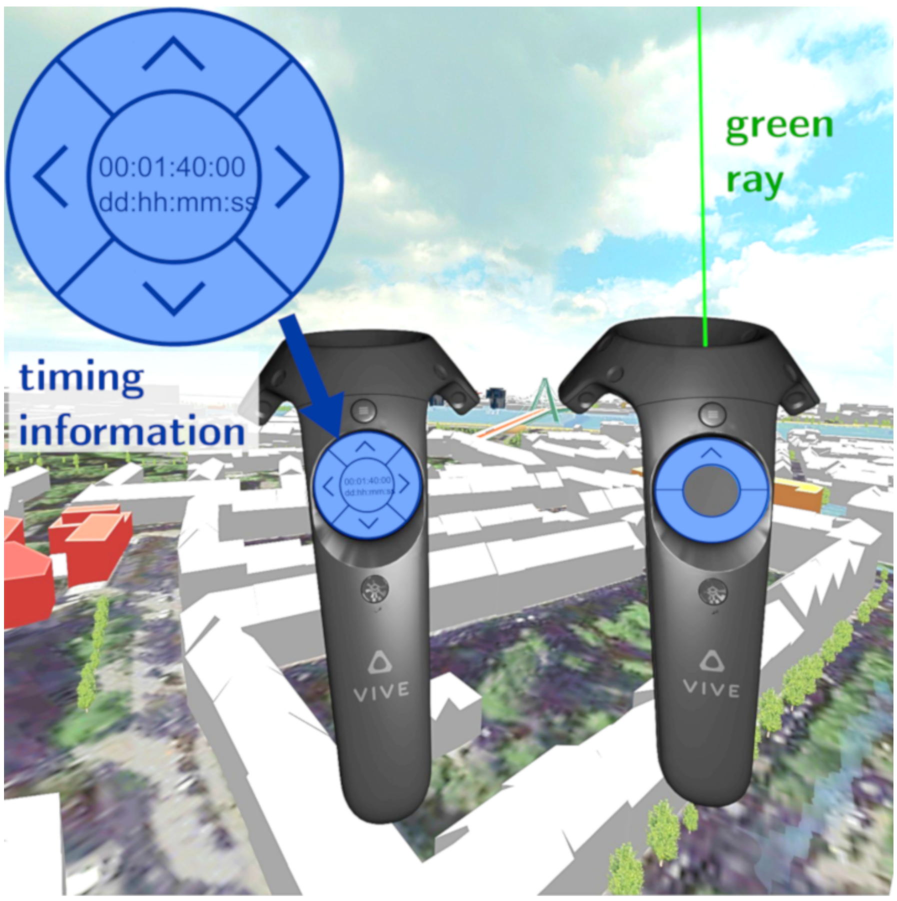


Figure 4.1: Screen shots of the controllers in the VE, extended by annotations. The left controller is the so called mode controller, the right one is the steering controller.

In order to interact with the virtual world, two controllers of the VR Hardware are needed. In this thesis, each of the controllers has a dedicated task to fulfill. Figure 4.1 shows the two controllers inside the VE. The controller with the timing information and the four arrows on the trackpad is the so called mode controller. The controller with the green ray associated is called the steering controller. The ray serves as a pointing device and gives the direction where the controller points at. The mode controller is held in the left hand and the steering controller is held in the right hand. Depending on what is more comfortable for the user, he/she can swap the controllers.

In general, the steering controller is responsible for the following tasks:

- Moving the user's view point through the virtual world
- Selecting a possibility out of a set of options and confirming this selection

The first task in the list is used during the spatial traveling, as described in Section 4.1. The second task is necessary to select a barrier type and a placement strategy in the context of a menu, as explained in Section 4.3.

The mode controller is responsible for the following tasks:

- Displaying time information
- Controlling the current time step and scenario
- Displaying the menu

The first task is for visualizing the current time step of the scenario. The second one is for progressing forward or backward in time or changing the current scenario. Section 4.2 describes the tasks in more detail. The last point in the list is necessary to be able to place barriers and is described in Section 4.3.

4.1 Spatial Traveling

As the HTC Vive is used as VR Hardware, natural walking to explore the VE is limited to the tracking area. The exploration of an urban environment by naturally walking through the streets could possibly give more immersed impression. Nevertheless, it would take some time to go to certain locations of interest or get an overview of a situation or the whole infrastructure. Because of these reasons, the pointing-directed steering and teleportation for traveling through the VE are implemented as movement techniques. By using these approaches, a fast and easy exploration of the environment is possible, after some interaction training time. In order to fly around the virtual world through pointing-directed steering, as described in Section 2.8, the green ray of the steering controller serves as the pointing device (see Figure 4.1). It gives the direction where the user is going to move. While pressing the trackpad on the upper part of the circle, the user is moving his/her viewpoint through the scene like he/she is flying. Figure 4.2a shows the texture displayed on the trackpad of the steering controller indicating the flying interaction. The arrow indicates the location where the user has to press to start the flying interaction.

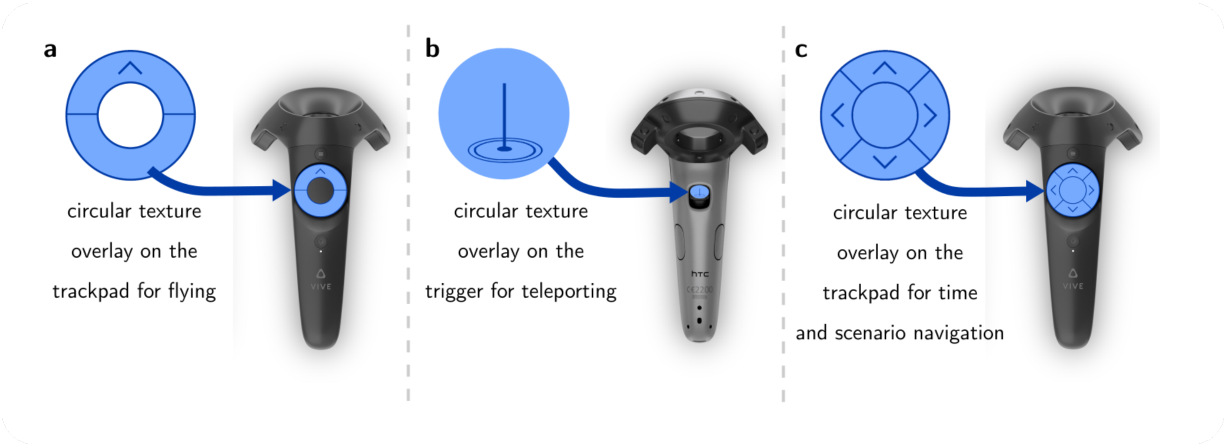

a
circular texture overlay on the trackpad for flying
b
circular texture overlay on the trigger for teleporting
c
circular texture overlay on the trackpad for time and scenario navigation

Figure 4.2: Images of the controllers showing three different texture overlays to represent interactions. (a) is applied to the trackpad of the steering controller. The arrow identifies the flying interaction. (b) is applied to the trigger of the steering controller. It identifies the teleporting interaction. (c) is applied to the trackpad of the mode controller. Depending on the pressed arrow, the user is able to change the current time step or the scenario. The original image of the controller is from Nick Hygens [Nic] and was edited by the author of the thesis.

Another supported movement technique is teleporting. Again the green ray of the steering controller serves as the pointing device. The location where the green ray intersects the terrain gives the position to where the user might move. To confirm the new position, the trigger of the steering controller comes into play. While pressing the trigger, nothing will happen. As soon as the trigger is released, the user moves to the location, where the ray intersects the terrain. The texture visualized in Figure 4.2b is displayed on the trigger of the steering controller and indicates this interaction. If the ray does not intersect the terrain as the trigger is released, nothing will happen and the user is going to stay at the current position. The teleporting movement is implemented in such a way, that it preserves the current height of the user. This means, that the view point of the user only changes in forward, backward, left, and right direction, but not up or down.

4.2 Time and Scenario Control

Visdom simulates flood scenarios over time. The visualized scene represents one time step in the simulation. In order to control the simulation progress in the VE, time control functions are required to go forward and backward in time. As Visdom can deal with multiple scenarios, interaction methods are provided to switch between them. For these interactions, the mode controller is responsible. The trackpad is used to go forward and backward in time of the current scenario as well as to change the scenario. Figure 4.2c shows an image of the texture overlay that is projected onto the trackpad of the mode controller. The arrow to the right triggers the application to go one time step ahead in time. The left one does the opposite action. Switching to the next scenario can be done by pressing the up arrow. Analogously, pressing the down arrow requests the previous scenario. By pressing into the middle of the trackpad, it is possible to directly go to the first frame of the scenario progress. Additionally, it is also possible to view a flood animation. This is done by continuously requesting the next frame. Through pressing the menu button, this is done continuously till it is pressed again.

4.3 Barrier Placement

One of the major goals in flood management is to protect human beings as well as to mitigate damages in the infrastructure. Based on this it has to be possible to add some protection mechanisms to the virtual world. This makes it possible to reduce the results of a flooding or prevent the flooding at all. The user has to perform the following steps to add protection mechanisms to the scene

1. Select one of the barrier types visualized in Figure 4.4.
2. Select the placement strategy as illustrated in Figure 4.5.
3. Determine the location where to place the barrier. Figure 4.6 shows how this is done.

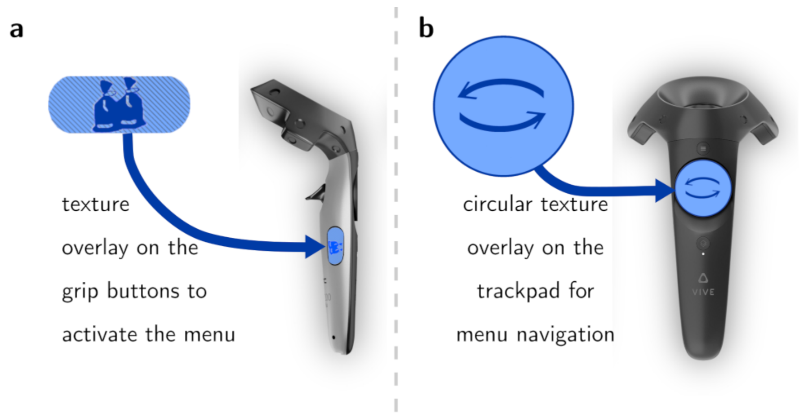


Figure 4.3: Images of the controllers with further texture overlays displayed on them. (a) is visible on the grip buttons of the mode controller. It activates the menu for selecting the type of barrier to place. (b) is visible on the trackpad of the steering controller if the menu is visible. The original image from the controller is from Nick Hygens [Nic] and was edited by the author of the thesis.

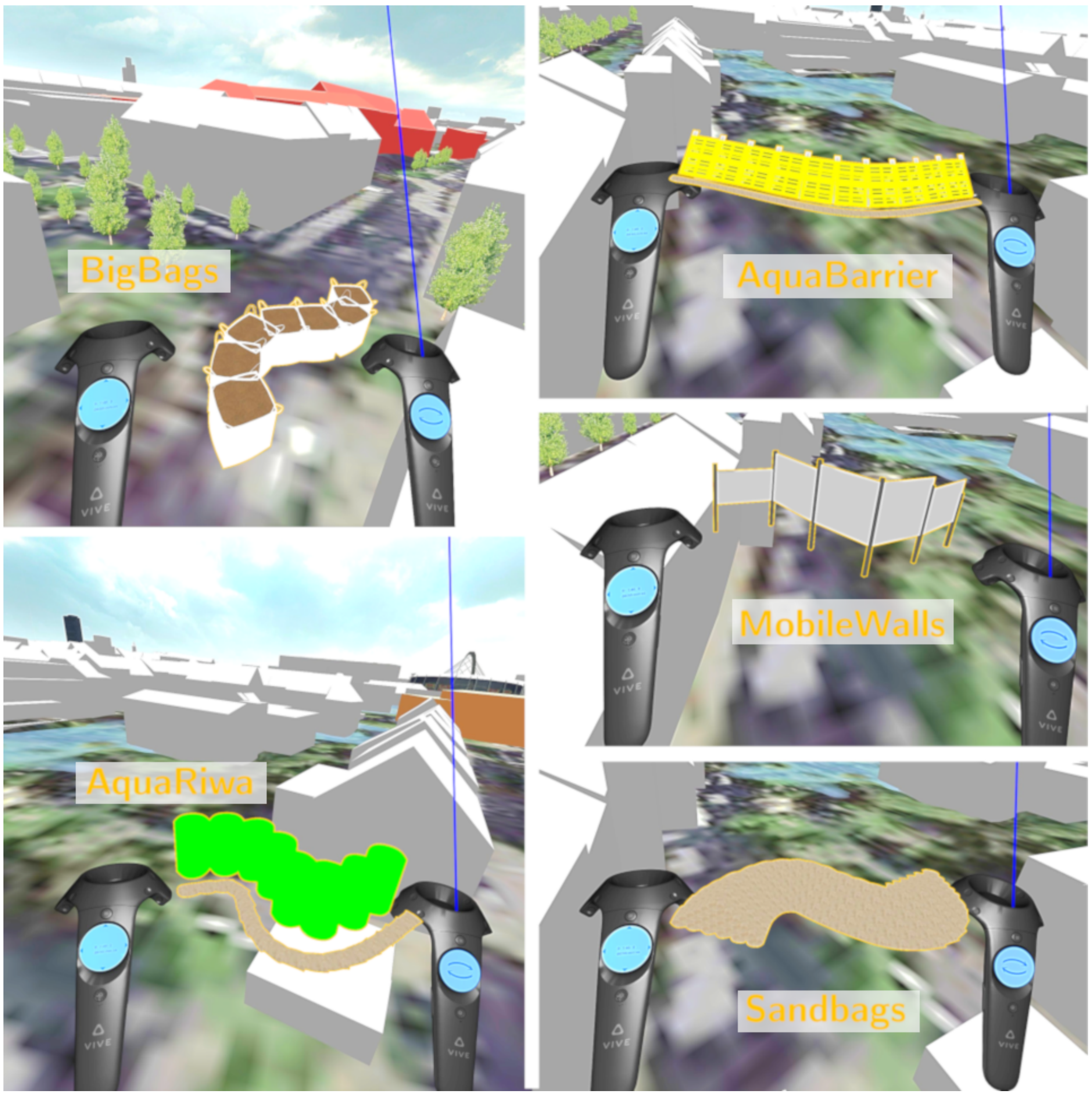


Figure 4.4: Screen shots of the five different barriers types that can be selected from the menu.

Before the barrier type can be selected, the user has to tell the system that he/she wants to place one. Therefore, the user has to press the grip buttons of the mode controller. These buttons are indicated by the texture visualized in Figure 4.3a. As the grip buttons are pressed, the green ray of the steering controller turns into a blue one, indicating that the user is currently in the mode of placing barriers. Furthermore, this mode means that the user is not able to move through the scene using teleportation or flying, but only by physically moving through the tracked space. Before the user is able to place a barrier, he/she has to select the protection type he/she wants to place. The user is able to choose between a predefined set of protection types. For this purpose a menu is used. It contains all the available barrier types as 3D model. As the user activates the menu by pressing the grip buttons, a model appears between the controllers. Figure 4.4 visualizes the described menu and the different barrier types contained in it. Only one model is visible at a time. In order to switch between all available barrier types, the user has to slide over the trackpad from left to right or the other way round. The texture in Figure 4.3b illustrates this interaction. It is applied to the trackpad of the steering controller, if the menu is active. Instead of sliding over the trackpad the user can also only touch and release the middle right or left of the trackpad. By pressing the trigger of the steering controller, the currently visible barrier type gets confirmed and is going to be created subsequently.

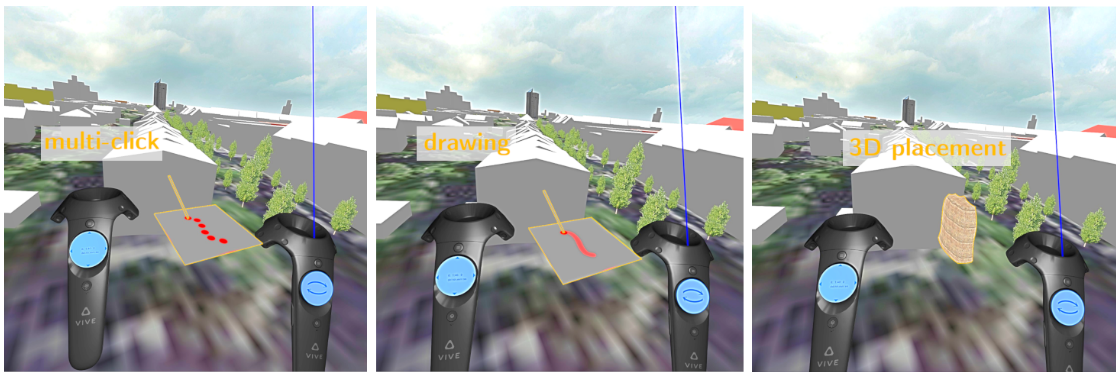


Figure 4.5: Screen shots of the three different placement strategies to identify the locations of the barriers.

In order to specify the location of the barrier, the user is able to choose between different placement strategies. For this purpose, three placement strategies are available: multi-click, drawing, or 3D placement. Figure 4.5 shows the corresponding 3D models used to represent these interaction methods in the menu. The switching between them and the selection of one strategy works in the same way as for the barrier type selection. For the multi-click placement strategy the user has to set individual control points indicating the contour of the barrier. These control points are created at the intersection point of the blue ray with the terrain as the trigger of the steering controller is released. Figure 4.6a illustrates the placement of these control points. As soon as the last control point of the barrier has been placed, the user has to finish the multi-click barrier placement by pressing the grip button(s) of the mode controller again. This time, the button event triggers the system to sent the positions of the control points to the Visdom Server. The Visdom Server uses them to calculate the selected barrier at this location and sends back the data to visualized them properly. Figure 4.6b hows the created barrier based on the defined control points of Figure 4.6a. With the drawing placement strategy, the user does not have to set each individual control point one after another. Instead, while he/she pulls the trigger of the steering controller and continuously moves the ray over the terrain, red boxes pop up above the intersection points. As soon as the trigger is released the control points are sent to the Visdom Server to calculate the barrier. As the user is working in a 3D world, a 3D interaction method for placing barriers is implemented as well. When selecting this interaction type, the steering controller transforms into a sandbag model. By tilting the sandbag about 90 degrees to the left or the right, red boxes drop out from the controller's position. Again, these boxes indicate control points for the barrier placement. After all control points have been placed, the positions are sent to the Visdom Server as the grip buttons are pressed.

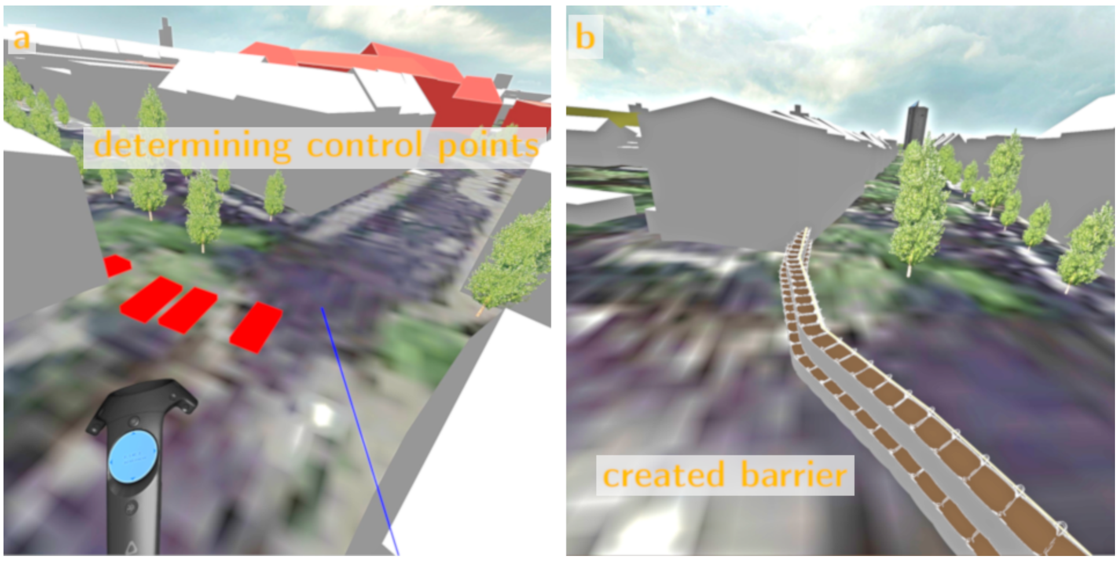


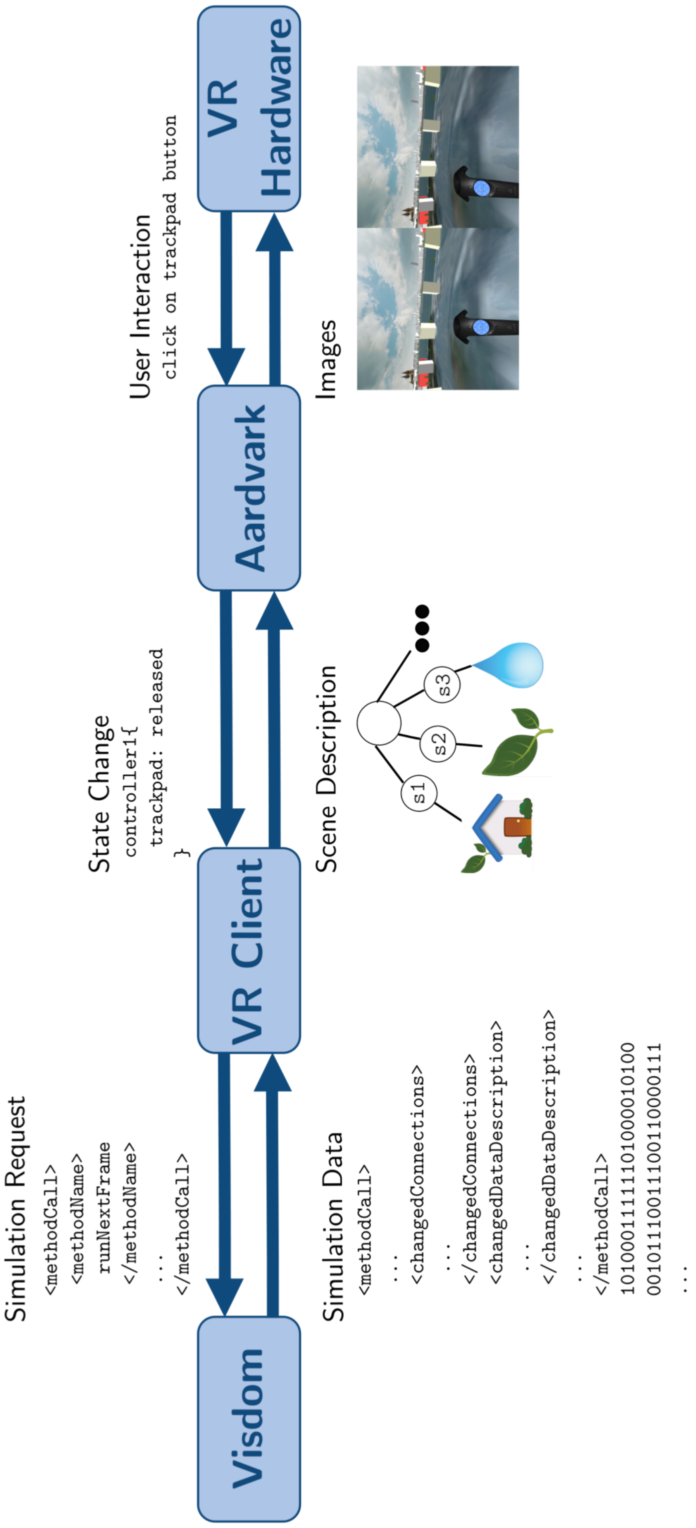
Figure 4.6: Screen shots showing how to place a barrier into the scene. (a) shows how the control points are defined. (b) shows the created barrier along the control points.

CHAPTER 5

Implementation

The previous chapters explained the main components of the system and its functions in a general overview. This chapter goes into more details on how the system is implemented. Figure 5.1 shows an extended illustration of the main components of VisdomVR. In comparison to Figure 3.1, the visualization in Figure 5.1 shows examples of the sent data between the individual components. The Visdom Server sends the Simulation Data in the structure of a XML document to the VR Client. Based on the implemented communication protocol between the Visdom Server and the VR Client, the client is able to process the received data. This processing results in a Scene Description in the form of a scene graph. This scene graph is the input for the rendering engine Aardvark. Aardvark uses the given scene graph to create to renderable stereo Images. These Images are sent to and displayed on the connected VR Hardware. With the VR Hardware the user is able to interact with the virtual environment. Aardvark records the performed User Interaction as State Changes. The VR Client can request the states of the different VR Hardware components. Depending on these states Simulation Requests are created that and sent to the Visdom Server. These requests either alter the simulation progress or change simulation parameters. The changes are considered in the simulation calculation and the data differences between the previous visualized scenario data and the updated one are sent to the VR Client. The workflow starts over again.

In order to understand the implementation details of the VR Client, further insights about the Visdom Server are needed. As a result, this chapter first describes some Visdom Server related implementation details. The section is followed by a description of the communication protocol used between the Visdom Server and the VR Client to exchange the Simulation Data. After the section about the communication protocol, the implementation details of the VR Client are explained. These details consist of the data parsing and interpreting part and the maintenance of the scene description.



```
<methodCall>
  <methodName>
    runNextFrame
  </methodName>
  ...
</methodCall>
```

```
<methodCall>
  ...
  <changedConnections>
    ...
  </changedConnections>
  <changedDataDescription>
    ...
  </changedDataDescription>
  ...
</methodCall>
101000111111010000010100
00101110011100110000111
...
```

```
controller1{
  trackpad: released
}
```

```
click on trackpad button
```

Figure 5.1: Illustration of the system components extended by examples of sent data types.

5.1 Visdom

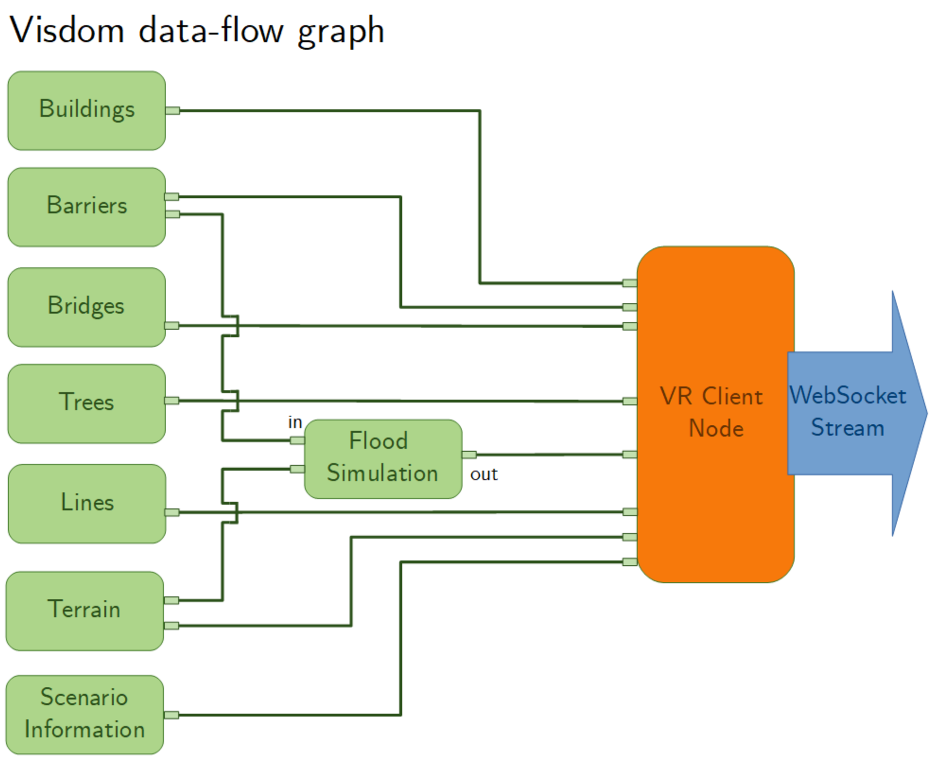


Figure 5.2: Schematic illustration of the Visdom data-flow graph showing the different nodes connected to the VR Client Node.

Visdom is based on the concept of data-flow graphs, as described in Section 2.3. More precisely, generic data-flow graphs are used in a multiverse [SWR$^+$13]. This kind of realization ensures an efficient recalculation and propagation of simulation data. As the implementation is done in a modular way, the system can be easily extended for this work by adding a new node for the VR Client. Figure 5.2 shows a schematic illustration of the used data-flow graph. The green colored nodes are the input connections to the VR Client Node (colored in orange). The VR Client Node determines the data that is sent as a stream over a WebSocket interface to the connected VR Client component of VisdomVR. The data of the input nodes are stored in corresponding data objects. In Figure 5.3 a detailed view of the input connections to the VR Client Node is given. The data objects used for the different input nodes are *Mesh, Water, Protection Line, Terrain, Scenario Information* (indicated by the *connector*). These data objects consist of different attributes (identified by the *associativeConnector*). Some nodes are represented by the same data object type. In order to distinguish them a *sequenceIndex* is used. The following subsections describe the individual data objects, indicated by the *connector*, and their attributes, indicated by the *associativeConnector*, in more detail.

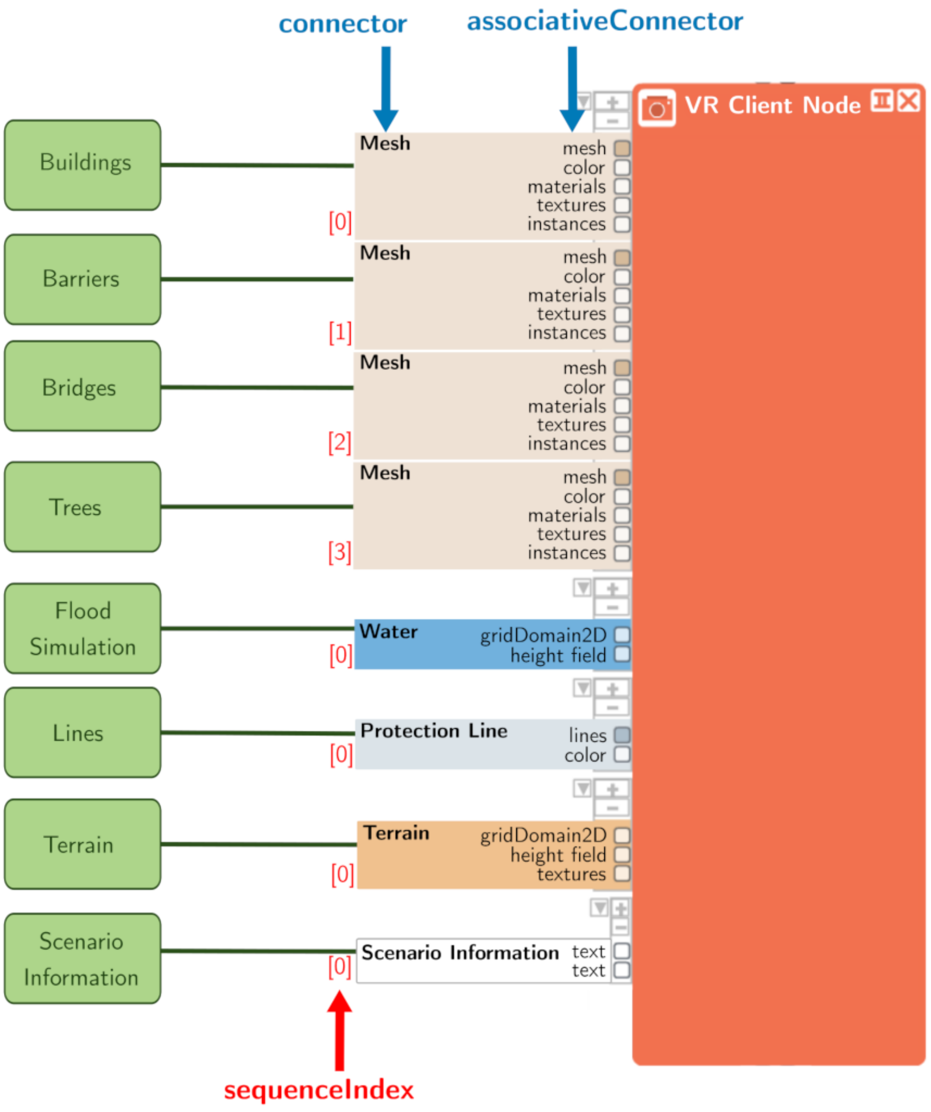


Figure 5.3: Detailed view of the VR Client Node (colored in orange) used in the Visdom data-flow graph model. The green boxes identify the different data graph nodes as illustrated in Figure 5.2. The nodes are represented by different data objects, determined by the *connectors*. The *associativeConnectors* specify the different attributes of these data objects.

5.1.1 Scenario Information

The scenario information contains some general information of the current scenario, like a label and the current time step. The time step of the scenario is displayed in the middle of the trackpad of the mode controller, like illustrated in Figure 4.1.

5.1.2 Mesh

The *Mesh* data object is used to represent multiple scene objects (Buildings, Bridges, Barriers, Trees). Because of this diversity, multiple attributes have to be considered to visualize the different scene objects properly.

```
type GroupInformation =
    {
        (*object ID*)
        id : int
        (*index of the first triangle of the object*)
        triangleStartIdx : int
        (*number of triangles belonging to the object*)
        elemCount : int
    }
```

Listing 5.1: Struct used to describe an object group.

`mesh` - The `mesh` attribute stores different data in a dictionary. The main components every `mesh` must contain are:

- `vertices`
- `indices`
- `normals`

The `vertices` contain 3D positions that define the geometry of an object in space. In combination with the `indices`, the triangulation of the mesh can be determined. The `normals` are the vertex normals of the object and are later necessary for lighting purposes in the rendering.

Additionally, a mesh can optionally contain:

- `uv-coordinates`
- `object groups`
- `material groups`

A `mesh` can contain all these attributes, only a subgroup, or none of them. The `uv-coordinates` are needed if texture information is present to properly map the given texture. `object groups` describe which of the given triangles of the whole mesh forms a group of one scene object belonging together. For example, the *Mesh* data object is used to describe buildings. Instead of just sending the geometry information of every individual building separately, all necessary geometry data of all buildings are sent in one array of `vertices`, one array of `indices`, and one array of `normals`. This has the advantage that only three buffers have to be used for uploading the geometry information for all buildings. This is much more efficient instead of creating thousands of buffers and uploading the individual geometry information for them. Additionally, to distinguish between the different buildings for rendering them properly, the `object groups` specify which triangles belong together and form one building. One `object group` is described by the information given in Listing 5.1 The `id` uniquely identifies a certain object. The `triangleStartIdx` specifies the index of the first triangle belonging to the object. The `elemCount` states how many triangles are needed to draw the object. Figure 5.4 illustrates how this information is used. Based on the `object groups`, the `indices` can be grouped together to identify individual objects.

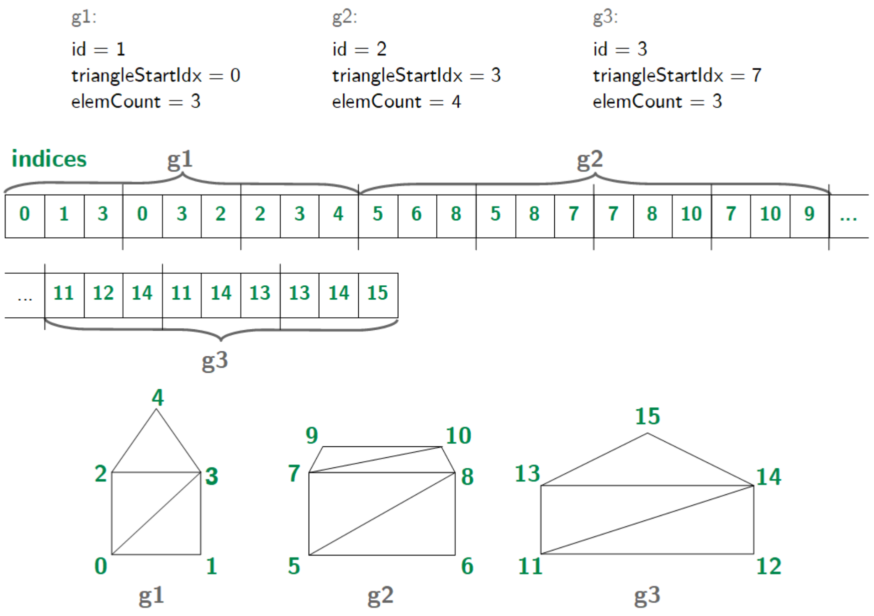


Figure 5.4: 2D example of `object groups` `g1`, `g2`, and `g3` indicating three different objects but using only one indices buffer.

A `material group` is described by the same structure as an `object group`. It is used to determine the appearance of an object. The `id` is used to identify the corresponding material in the `materials` attribute. The `triangleStartIdx` and `elemCount` specify how many triangles are covered by a certain `material`. Figure 5.5 gives an illustration, how `material groups` are used. Group `g1` describes the left building. Additionally, the material information `m1` states that a material with `id = 1`, represented by the color orange, has to be applied to two triangles starting at index `0`, belonging to the first group `g1`. The building in the middle is described by the group `g2` and material `m2`. The material `m2` with `id = 2` is represented by the yellow color. The third building has the same material as the first building, this time the whole group is textured by the given material.

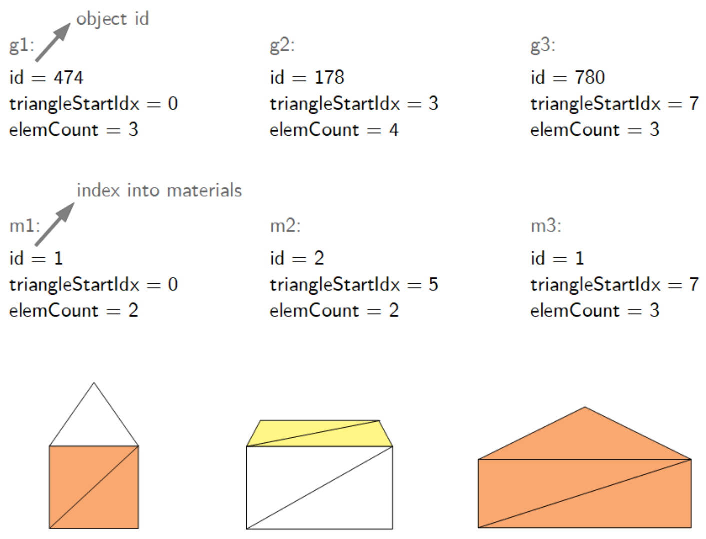


Figure 5.5: Example of `object groups (g1, g2, g3)` with additional `material groups (m1, m2, m3)`.

`color` - The `color` determines the color of either an `object group` or a `vertex`. With the per object group colors, for each `object group` one color value is available. An example of this coloring technique is to color buildings based on their priority in a scenario (like in Figure 3.9d). For per vertex colors one color value per `vertex` is available.

`materials` - The `materials` contain a set of additional properties about the appearance of the individual objects. This includes information about if an object is textured. If this is the case, every material in the `materials` contains an `id` that references to a texture in `textures`. Even if the `materials` provide more information than only about the texture of an object, no further information is used in the scope of this thesis.

`textures` - The `textures` contain the set of all textures. In combination with the `material groups` and the `materials`, the correct texture of an object can be determined and applied to it. If an object has already a certain color, the texture information is combined with the given color value.

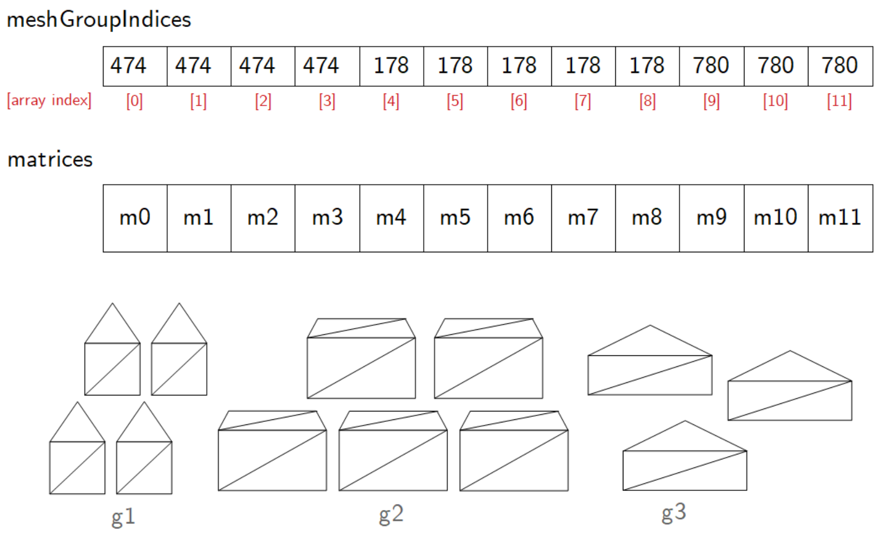


Figure 5.6: Illustration how `meshGroupIndices` are used to map the `matrices` per `object group`.

`instances` - The `instances` consist of multiple components. The most relevant ones are the `matrices` and the `meshGroupIndices`. The `matrices` are a list of transformations needed to render a given mesh geometry multiple times with different locations, scales, and rotations. This means that the geometry, described by the `vertices`, is rendered as often as the number of available `matrices`. If `object groups` are available, further information is needed to determine how often the individual objects are in the scene. For this purpose, the `meshGroupIndices` are provided. The `meshGroupIndices` are an array of the same size as the number of available `matrices` (see Figure 5.6). The index stored in each array cell indicates the `id` of an object. This `id` corresponds to the object id stored in the `object group` information. Figure 5.6 illustrates how `meshGroupIndices` are used to render specific objects multiple times. With the help of Figure 5.4 the `matrices` can be mapped to the corresponding `object group`. The first four elements in the `meshGroupIndices` array of Figure 5.6 correspond to group `g1`, array indices `[4]` to `[8]` to group `g2` and array indices from `[9]` to `[11]` to group `g3`. Based on this information, the matrices can be matched to the corresponding object groups based on the array indices. This means, that the `matrices` from array index `[0]` to `[3]` are used to render the `object group g1` four times, `matrices m4` to `m8` belong to group `g2` and `m9` to `m11` to group `g3`. With this information it is possible, that each individual object can be rendered multiple times with different locations, scales and rotations.

5.1.3 Terrain

The *Terrain* data object is used to represent the geographic environment of a scenario. In the following the different attributes are explained that describe this data object These are used to calculate the corresponding geometry information to create the scene graph node.

`gridDomain2D` - The `gridDomain2D` consists of an `origin`, a `dimension`, a `cellSize`, and a `validity`. With the given components a 2D regular grid can be created, as illustrated in Figure 5.7 `origin`, `dimension`, and `cellSize` consist of a x and y component each. The `origin` defines the location of the grid in the world. The `dimension` indicates how many cells are in the x direction and how many are in the y direction. The `cellSize` states the size of a cell.

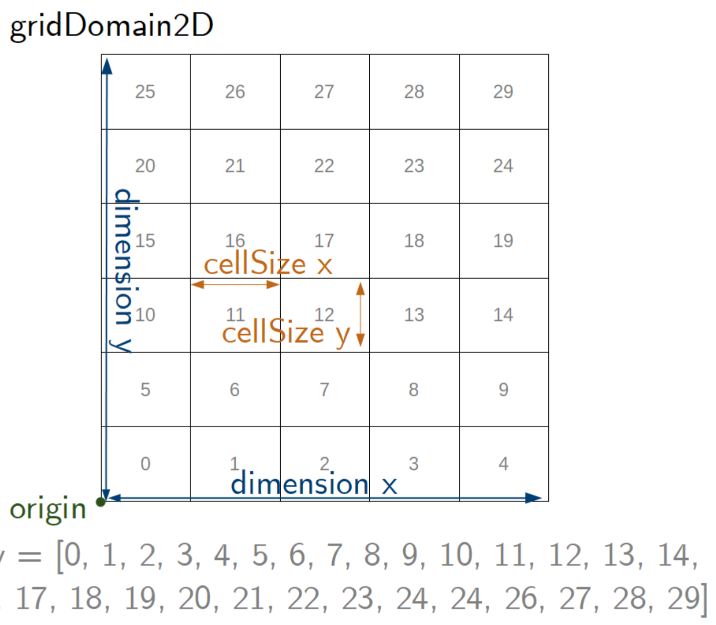

gridDomain2D
25
26
27
28
29
20
21
22
23
24
15
16
17
18
19
10
11
12
13
14
5
6
7
8
9
0
1
2
3
4
dimension y
cellSize x
cellSize y
dimension x
origin
validity = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 24, 26, 27, 28, 29]

Figure 5.7: Example of a 2D regular grid with `dimension` [5, 6] and `cellSize` [3, 2.5].

The `validity` stores the indices of those grid cells that are valid. For the terrain, all cells are valid.

`height field` - The `height field` stores for each valid grid cell a corresponding absolute height value. In the context of the *Terrain* data object, the height field values are called `terrain levels`. For the *Terrain*, as many `terrain levels` are available as the number of cells in the grid. As for each grid cell one terrain level value is available, the representation is called dense.

`texture` - The texture attribute stores the texture that has to be applied onto the terrain geometry. If no texture is available, the geometry is colored with a default color value. The correct coordinates are calculated based on the `dimension` and `cellSize` of the corresponding `gridDomain2D`.

5.1.4 Water

The *Water* data object has the same attributes as the *Terrain*, except of an `texture`. Additionally, the main differences can be encountered within the `validity` and `height field`. In the context of the *Water* data object, the `height field` are called `water levels`. `water levels` can rise and decrease in consecutive frames, whereas the `terrain levels` stay always the same in a scenario. Typically, not the whole terrain is covered by water. This implies that not all cells of the underlying grid are valid, determined by the `validity`. This special conditions have to be considered in rendering the water geometry.

5.1.5 Protection Line

The *Protection Line* is another type of data object. It consists of the following attributes

`lines` - One line is specified by a set of points in space. `lines` contain multiple sets of one line.

`color` - The color attribute specifies the color of each line.

5.2 Communication Protocol

In order to transmit the data from the Visdom Server to the VR Client, a communication protocol is necessary. This communication protocol gives information about the data that is transmitted. The data is transmitted from the Visdom Server to the VR Client over a WebSocket. The messages are based on a XML structure as illustrated in Listing 5.2. The text colored in orange, written in bold and surrounded by the square brackets, are placeholders and vary depending on the message type. In general, two message types are exchanged. The VR Client performs VR Client Requests to send messages to the Visdom Server. The Visdom Server sends Visdom Server Responses to send messages to the VR Client. The following subsections explain the different possible contents to replace the placeholders with.

```
<methodCall discardable="false" xsi:noNamespaceSchemaLocation="temp/
    MethodCall.xsd" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <methodName>[methodName]</methodName>
    <params>
        <param type=[paramType]>
            [paramContent]
        </param>
    </params>
```

```
</methodCall>
```

Listing 5.2: Basic XML Structure used to exchange messages between the VR Client and the Visdom Server.

5.2.1 VR Client Requests

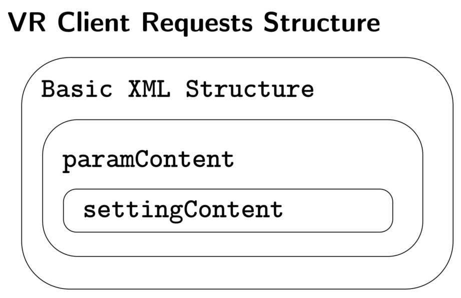


Figure 5.8: Schematic illustration of the VR Client Requests Structure.

For requesting data from the Visdom Server, the VR Client has to send specific requests. Figure 5.8 shows a schematic illustration of the VR Client Requests Structure. Every request from the VR Client is embedded in the Basic XML Structure from Listing 5.2. The placeholders in the XML are exchanged with one of the following content per placeholder.

[methodName]: `runStartFrame`, `runNextFrame`, `runPreviousFrame`, `runNextTrack`, `runPreviousTrack`, `interactorConfigurationPreset`, `interaction`

[paramType]: `node`

[settingType]: `BaseSettings_t`, `GenericSettings_t`

[paramContent]: is replaced by the XML document in Listing 5.3 if the `methodName` starts with `"interact"`.

```
<node id="ClientRenderingNodeId">
    <settings xsi:type=[settingType]>
        [settingContent]
    </settings>
</node>
```

Listing 5.3: XML document for the `paramContent` in the VR Client Requests.

Depending on the type of request, different combinations of the content cause certain actions. The main difference between the requests is the `methodName`. In general, the functionalities of the methods can be separated into two groups: one is needed to steer the simulation progress and the scenarios, the other one is needed to edit the simulation parameters. The methods starting with `"run"` control the simulation progress. The ones starting with `"interact"` are used for altering the scenario. The different requests per `methodName` cause different actions:

`runStartFrame` sets back the current time step of the simulation to 0.0.

`runNextFrame` requests the next time step of the simulation.

`runPreviousFrame` demands the simulation at the previous time step.

`runNextTrack` switches the currently visualized scenario. As Visdom is able to create multiple scenarios, it is possible to switch between them. This request switches to the next available scenario.

`runPreviousTrack` requests the previous available scenario.

`interactorConfigurationPreset` tells the server that a certain tool is selected for performing a specific interaction. This is mainly used to tell the Visdom Server the barrier type to place into the scenario. The Visdom Server supports further tools to select, but in this work, we are only able to add different barriers to the simulation domain. The barrier type to place is specified in the `settingContent` of the XML document. The desired XML to specify this is listed in Listing 5.4.

```xml
<content>
    <name>[barrierType]</name>
</content>
```

Listing 5.4: XML document in the settingContent to specify the barrier type to place.

The `barrierType` has to be replaced with the appropriate barrier type that is going to be added to the simulation domain. Additionally, in the same line, the type of placement strategy has to be specified. The placement strategy defines how the location of the barrier is going to be determined.

`interaction` has to follow the request with the `interactorConfigurationPreset` method. The information has to be substituted in XML for the `settingContent` placeholder. The example in Listing 5.5 shows how to tell the Visdom Server the locations where to place a barrier.

```xml
<content>
    <queuedInteraction>
        <type>WorldSpaceDrawing</type>
        <extension type=WorldSpaceDrawingInteraction>
            <objectId>0</objectId>
            <!-- world-space positions as evaluated by RayPicking in the
                 VR-Client application -->
            <worldPositions>
                <element>
                    <element>
                        <x>0</x>
                        <y>10</y>
                        <z>0</z>
                    </element>
                    <element>
                        <x>20</x>
                        <y>20</y>
                        <z>0</z>
                    </element>
                    <element>
                        <x>30</x>
                        <y>30</y>
                        <z>0</z>
                    </element>
                </element>
            </worldPositions>
            <!-- ------------------------------------------------ -->
        </extension>
    </queuedInteraction>
</content>
```

Listing 5.5: XML document in the `settingContent` to specify the barriers' location.

The bold text colored in orange ("WorldSpaceDrawing" and "WorldSpaceDrawingInteraction") varies depending on the type of interaction performed. Because currently the placement of barriers is the only supported interaction for the VR Client, this content is fixed. In the `worldPositions` tag of the `interaction` method, the control points for the location of a barrier are specified. The Visdom Server creates the barrier along the given control points.

For all methods starting with `"run"` the `settingType` is `BaseSettings_t`. For the methods starting with `"interact"` the `settingType` is `GenericSettings_t`.

5.2.2 Visdom Server Responses

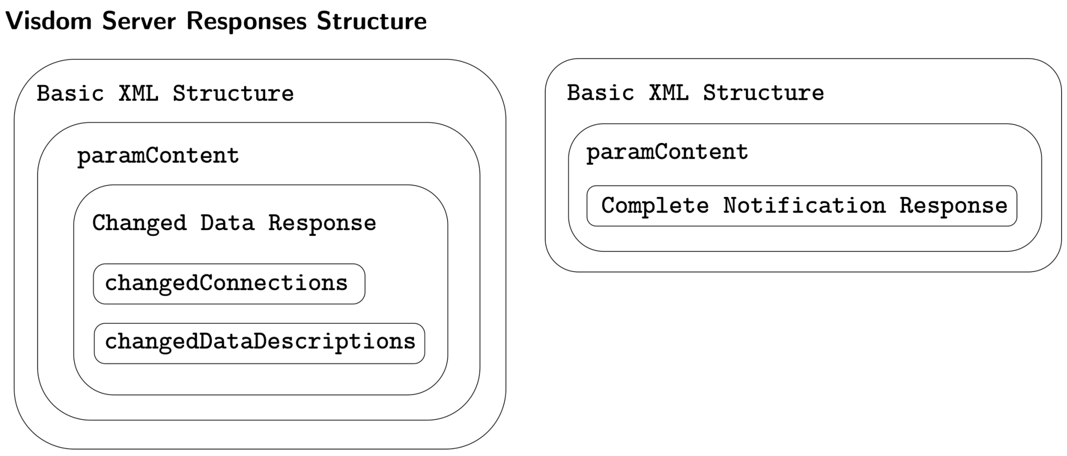


Figure 5.9: Schematic illustration of the Visdom Server Responses Structure.

The Visdom Server responses to a client request with two types of response messages. These messages follow the structure as illustrated in Figure 5.9. Both are embedded in the Basic XML Structure and the `paramContent`. The first message is labeled as the *Changed Data Response* and the second one as the *Complete Notification Response*. The *Changed Data Response* consists of a XML part and a binary part, while the *Complete Notification Response* consists only of a XML part. The XML part in the *Changed Data Response* describes the changed attributes of the data objects. This part again consists of two parts, named the *changedConnections* and the *changedDataDescription*. The binary part contains the raw data of these attributes. The *Complete Notification Response* indicates the end of the Visdom Server response to a client request. Both XML documents are embedded within the XML structure from Listening 5.2. The method name for the response is *parseResults* and can be found in the *methodName* tags of the XML document. Depending on the type of response message, different information is placed in the *paramContent* placeholder. Listing 5.6 shows the structure of the *Changed Data Response* and Listing 5.7 of the *Complete Notification Response*.

```
<results resultType=InputAdapterState>
    <success>true</success>
    <!-- scenario and time step info -->
    <scope global="false">
        <track>38</track>
        <timeValue>63600</timeValue>
    </scope>
    <!-- ---------------------------- -->
    <inputAdapterState>
        <inputAdapterState>
            <!-- changed data info -->
            <nodeId>ClientRenderingNodeId</nodeId>
            <changedConnections containsSequence="true">
```

```
                [changedConnections]
            </changedConnections>
            <changedDataDescriptions containsSequence="true">
                [dataDescription]
            </changedDataDescriptions>
            <removedConnections containsSequence="true" />
            <!-- ----------------- -->
        </inputAdapterState>
    </inputAdapterState>
</results>
```

Listing 5.6: XML document containing the *Changed Data Response*.

```
<results resultType="RequestComplete">
    <success>true</success>
    <errorReport>no error</errorReport>
</results>
```

Listing 5.7: XML document containing the *Complete Notification Response*.

Comparing the two XML documents, both are embedded in a *results* tag. This tag also contains an attribute indicating the type of the result. For the *Changed Data Response* the type is always *InputAdapterState* and for the *Complete Notification Response* it is *RequestComplete*. In the *Changed Data Response* document, the *InputAdapterState* tags contain the desired information to parse the sent data. This information consists of two parts, the *changedConnections* and the *changedDataDescription*. The content of the *changedConnections* is given in Listining 5.8

```
<changedConnections containsSequence="true">
    <element>
        <connector>[connector]</connector>
        <associativeConnector>[associativeConnector]</associativeConnector>
        <sequenceIndex>[sequenceId]</sequenceIndex>
    </element>
</changedConnections>
```

Listing 5.8: XML document specifying the *changedConnections*.

The *changedConnections* tag contains the information about the which data has changed. Each *element* tag contains the necessary information to distinguish between the individual data objects and the different attributes. This information corresponds to the connected nodes of the VR Client Node in the Visdom data-flow graph (see Figure 5.3). The *connector* identifies the data object. The *associativeConnector* identifies the changed data of an attribute. In combination with the *sequenceIndex*, a data object can be uniquely identified. The number of *element* tags in the *changedConnections* varies depending on how many attributes have changed since the last Visdom Server Response. For example, if the color values of a building have changed, the XML has to contain the information in Listing 5.9 to clearly identify the concerned data.

```
    <connector>mesh</connector>
    <associativeConnector>color</associativeConnector>
    <sequenceIndex>0</sequenceIndex>
```

Listing 5.9: XML document specifying a connected node.

The `connectors` and `associativeConnectors` in Listings 5.8 and 5.9 can be replaced by one of the following values:

[connector]: `mesh`, `terrain`, `water`, `lines`, `text`

[associativeConnector]: `mesh`, `color`, `texture`, `materials`, `instances`, `gridDomain2D`, `heightfield`, `text`

The `sequenceIndex` depends on the number of connected nodes in the data-flow graph of the same data object type. Listing 5.10 shows the structure of the *changedDataDescription*.

```
<changedDataDescriptions containsSequence="true">
    <element>
        <dataType>[dataType]</dataType>
        [dataDescription]
    </element>
</changedDataDescriptions>
```

Listing 5.10: XML document containing the *changedDataDescription*

The `dataType` of this XML document can be replace by the following values:

[dataType]: `GPUVector<T>`, `Images<T>`, `Mesh`, `MeshInstances`

The `dataDescription` is replaced by additional information about how many data values have to be parsed. The XML tags vary depending on the `dataType`. Listing 5.11 gives an example of a XML document if the `dataType` is `GPUVector<T>`.

```
<GPUVectorDescription>
    <numElements>20</numElements>
</GPUVectorDescription>
```

Listing 5.11: Example XML document for the `dataDescription` placeholder of the `dataType` `GPUVector<T>`.

For each *element* tag in the `changedConnections` document, a corresponding `element` tag in the `changedDataDescriptions` exists. This time, the `element` tag describes the underlying data type of a given attribute and the size of those data. The XML document in the *Changed Data Response* is followed by a binary stream. This stream contains the real data values in the same order as they are specified by the `changedConnections` and `changedDataDescriptions`. After all simulation data have been sent, the server sends the *Complete Notification Response*, as stated in Listing 5.7. The main objective of this message is to indicate that all necessary data have been sent. After the VR Client has received this message, it has all the desired data to correctly parse and visualize the current simulation scenario. Additionally, the Visdom Server is now able to process a new request.

5.3 VR Client

The VR Client constitutes the missing part in VisdomVR. The main work of this thesis is its implementation. The used programming language is F#. For shader programming the domain specific language FShade [fsh] is used. The OpenVR SDK [ope] provides access to the used VR hardware and serves as a communication interface between Aardvark and the HTC Vive. The simulation and scenario data calculated by the Visdom Server are transmitted over a WebSocket.

The Visdom Server is based on a generic data-flow graph [SWR+13]. This means, only those data, that has changed since the last request, is sent. Aardvark works on a similar basis and updates the rendering if changes of the underlying data happen. Instead of a generic data-flow graph, the concept of an incremental rendering VM [?] is used to implement the rendering engine. As the Visdom Server sends changes and Aardvark calculates changes to update only those parts of the rendering that are affected, one straightforward possibility is to directly pass the sent and parsed simulation data to Aardvark. This way, the VR Client does not know anything about the current simulation state and does not have to synchronize and update the old data with the new one. The second way is to interpret the parsed data and store them in appropriate data objects. This kind of integration technique lets the VR Client always know the current simulation state and the different objects can be rendered in an optimized way depending on their type and the available attributes. In our work, the second implementation approach is used. As the VR Client receives the responses from the Visdom Server, the received data stream is parsed. In a next step, the individual data are interpreted and organized into corresponding data objects. Based on these data objects, the scene graph can be created or updated and the resulting image is sent to the VR Hardware. The following subsections describe how these tasks are implemented.

5.3.1 Parsing and Interpreting Simulation Data

After the VR Client has received the data from the Visdom Server, the data is parsed and interpreted based on the communication protocol. Through iterating over the `element` tags within the `changedDataDescriptions`, the binary stream can be parsed. Each attribute of a data object has a certain `dataType`. The stated `dataType` in the XML (see Listing 5.10) is Visdom related. This each `dataType` is represented by a certain Visdom Data Structure. In order to be understandable for the used rendering engine, the Visdom Data Structures are parsed into corresponding VR Client Data Structures. Table 5.1 lists the different available Attributes and the used Visdom Data Structures to represent them. Each Visdom Data Structure is parsed into a corresponding VR Client Data Structure, as listed in the last column of Table 5.1. In order to determine the size of the data structure, a XML document, like in Listing 5.11, gives further information. The example shows a XML document for the `GPUVector<T>` data structure. The `numElements` gives the number of data values of type `T` in the `GPUVector`. Based on this information, the binary stream is parsed into the appropriate VR Client Data Structure with the given number of data values. For the given example this means that 20 elements of type `T` are stored in an array. If we assume that type `T` is a `V3d`, which is a vector structure storing three components of type double, $20*3*sizeof < double >= 480$ bytes of the sent binary stream are parsed into an array of type `V3d`.

Attributes	Visdom Data Structures	VR Client Data Structures
`mesh`	`Mesh`	`Dictionary<string, T>`
`color, materials, height field`	`GPUVector<T>`	`T[]`
`textures`	`Images`	`List<Image>`
`instances`	`MeshInstances`	`Dictionary<string, T>`
`gridDomain2D`	`GridDomain2D`	`Grid { origin : V2f, dimension : V2i, cellSize : V2f, validity : int[]}`
`lines`	`Lines`	`List<V3f[]>`

Table 5.1: Overview about the used Visdom Data Structures and corresponding VR Client Data Structures to store the data of a certain attribute of the different data objects.

After parsing the data into the correct data structures, they are organized into individual data objects. In analogy to the data objects in Visdom (`Terrain, Water, Mesh, Line`), the corresponding data objects for the VR Client are `VRTerrainType, VRWaterType, VRMeshType, VRLineType`. As the `connector`, `associativeConnector`, and `sequenceIndex` uniquely identify one data object, only one VR Client data object per unique Visdom data object is created. If a data object already exists, only the desired attributes are updated.

5.3.2 Scene Graph Creation and Update

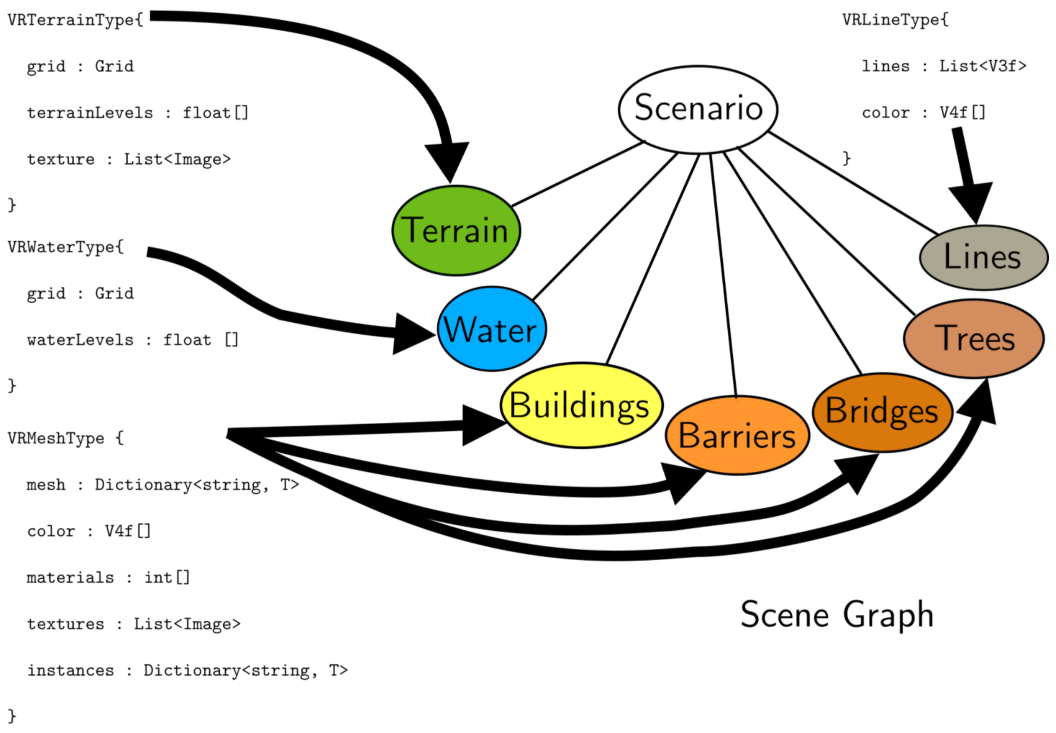


Figure 5.10: Simplified illustration of the scene graph. The *VRMeshType* is used to represent various scene objects. All other data objects are used to represent a particular scene object at a time.

After organizing the different data into the appropriate data objects, the scene graph has to be updated. For each unique data object one corresponding scene graph node exists. Figure 5.10 gives a simplified illustration of the scene graph. Each scene graph node represents a certain scene object. Only the *VRMeshType* is used for multiple scene objects. They can be distinguished by their *sequenceIndex*. A scene object again consists of different nodes containing information about the geometry, appearance and transformation. For the sake of simplicity, only a general scene graph is visualized containing the principal scene objects of a scene as a scene graph node. For each data object, which does not already have a corresponding scene graph node, a new node is created. Through adding new nodes to the scene graph, they get tracked by the rendering engine framework. If changes are made to the underlying data of a node, the rendering framework reacts and updates the rendering. For those data objects, which already have a corresponding scene graph node, no new node has to be created. In order to change the currently rendered scene based on the given data updates, only the underlying data are updated which relate to the scene graph node. As Aardvark tracks the changes of the data of a scene graph node, the rendering gets updated as changes are propagated. For each individual data object different data are available. Depending on the available attributes, different graphic commands are used. This enables a more efficient rendering of the scene. In order to understand, how this is done, the following subsections address each single data object. Each describes how the corresponding scene graph nodes are created.

VRMeshType

The *VRMeshType* is a little bit complex with respect to the available data and used draw calls for rendering. With the *VRMeshType*, any kind of scene object can be represented. In this thesis, it is used to render buildings, bridges, barriers, and trees. Depending on the scene object, different attributes are needed for rendering. A *VRMeshType* can represent a single scene object, like a bridge, or multiple different scene objects, like multiple different bridges. It is also possible to model different scene objects and each of these different scene objects appears as multiple instances in the scene. For example, different kinds of buildings and each of it appears multiple times in a scene at different places, scales, and rotations. Despite of the number of different objects, each of them can differ in their appearance. Some of them are just colored differently, others only textured or colored and textured.

The presented attributes in Subsection 5.1.2 contain all the necessary information to take this into account in the rendering. Depending on the available information for a given mesh, appropriate graphic commands are used to built the corresponding scene graph node. The most basic case is, if no optional data are available and the *VRMeshType* simply consists of the three required components: `vertices`, `indices`, and `normals`. This presents the simplest case because no further attributes have to be considered in creating the scene graph node. Thus, a basic direct indexed rendering is used. In general, the simplest case does not occur very often. Mostly, additional attributes are available that are taken into account for a proper rendering.

`matrices` are one of these attributes. If a *VRMeshType* contains `matrices`, instancing [ins] is used to render the object. If it also contains `object groups` and `meshGroupIndices`, further processing is necessary. One `object group` indicates a group of triangles as one related object in a set of multiple triangles. Figure 5.4 illustrates how `object groups` are used to distinguish between individual objects. As the objects can differ in their appearance, `material groups` are another attribute of a *VRMeshType*. With the use of `material groups` it is possible to texture individual triangle groups differently, as visualized in Figure 5.5. As the `material groups` and `object groups` partly overlap in their information, the two sets have to be separated. This is done by creating a set containing textured triangle groups and another set containing the untextured ones. Listing 5.12 shows how this separation into the two sets is done. `materialGroups` is the set containing the `material groups`. `objectGroups` contains the `object groups`. The code iterates over all `material groups` and checks if it overlaps with an `object group`. If this is the case, the groups are split up, so that the do not overlap. In the end, `resultMaterials` contains all textured groups and `generalWorkingSet` stores all untextured groups. Additionally, a bidirectional dictionary is created, that stores for each `material group` the overlapping `object group`. `objectGroupPerMaterial` stores this bidirectional dictionary.

```
let checkRangeOfMaterialGroups} (materialGroups : GroupInformation[]) (
    objectGroups : GroupInformation[]) =
    let objectGroupPerMaterial = Dictionary()
    let generalWorkingSet = List<GroupInformation>(objectGroups)
    let resultMaterials = List<GroupInformation>()

    (*go through all materials*)
    for materialGroup in materialGroups do

        (*function checking if the current material group is in the range of
            the object group*)
        let checkMembership (g : GroupInformation) : bool =
            g.triangleStartIdx <= materialGroup.triangleStartIdx &&
            (materialGroup.triangleStartIdx + materialGroup.elemCount) <= (g.
                triangleStartIdx + g.elemCount)

        (*check if the object groups contain the material group under
            consideration*)
        match generalWorkingSet |> Seq.tryFindIndex checkMembership with
        | None ->
                Log.warn "cannot find material group in object group"
                ()
        | Some i ->
                (*a object group contains the material information*)
                let objectGroup = generalWorkingSet.[i]
                generalGroupPerMaterial.Add(materialGroup, generalGroup)

                if ObjectGroup.triangleStartIdx = materialGroup.
                    triangleStartIdx then
                    (*the object group and material group start at the same
                        triangle*)
                    resultMaterials.Add materialGroup
                    (*check if the material group covers the whole object
                        group*)
                    if not materialGroupEntirelyCoversTheObjectGroup then
                        (*create a new object group with the residual
                            triangles not covered by the material*)
                        let restOfObjectGroup =
                                {   objectGroup with
                                        triangleStartIdx = objectGroup.
                                            triangleStartIdx + materialGroup.
                                            elemCount
                                        elemCount = objectGroup.elemCount -
                                            materialGroup.elemCount
                                }

                    generalWorkingSet.Add restOfObjectGroup
                    resultMaterials.Add materialGroup

                else
                    (*general group got empty*)
                    generalWorkingSet.RemoveAt i |> ignore
            else
                    (*material is sub-range of object group, split object group and create new groups*)
                    let left =
                            {
                                triangleStartIdx = objectGroup.triangleStartIdx
                                elemCount = materialGroup.triangleStartIdx - objectGroup.triangleStartIdx
                                id = objectGroup.id
                            }
                    let right =
                            {
                                triangleStartIdx = materialGroup.triangleStartIdx + materialGroup.elemCount
                                elemCount = objectGroup.elemCount - (left.elemCount + materialGroup.elemCount)
                                id = objectGroup.id
                            }

                    generalWorkingSet.RemoveAt i
                    resultMaterials.Add materialGroup
                    if left.elemCount > 0 then generalWorkingSet.Add left
                    if right.elemCount > 0 then generalWorkingSet.Add right


resultMaterials.ToArray(), generalWorkingSet.ToArray(), objectGroupPerMaterial
```

Listing 5.12: Matching `material groups` to `object groups`.

The two distinctive sets of textured and untextured groups form the basis for the rendering. Both groups use the *struct* from Listing 5.1 to describe the triangle groups. In order to determine if the individual objects appear multiple times in the scene or only once, the `meshGroupIndices` are used. The `meshGroupIndices` indicate the *object groups* that have to be rendered multiple times. Figure 5.6 illustrates how the `meshGroupIndices` are used to determine the groups that are rendered multiple times. This is done in an appropriate function. After all pre-processing steps are finished, the creation of the draw calls can be done. Instead of using direct rendering, indirect rendering is used to tell the GPU, which triangles belong together and form an object.

Listing 5.13 shows the used code to create the indirect draw call based on a given triangle group (= `groupInfo`). `instanceCount` states how often the object is in the scene. By default, this is one. If the group under consideration has matrices assigned, determined by the `meshGroupIndices`, this value is replaced by the correct number of instances for this group. `firstInstance` gives the index where to find the first instance of this object in an instance buffer. By default, this is zero if the object does not occur multiple times and no instance buffer is available. If the group under consideration occurs multiple times, the correct index into the instance buffer has to be set. The `FaceVertexCount` states how many vertices are used for the object. This value is calculated by the available `elemCount`, which is the number of triangles belonging to the object. As one triangle means one face and three vertices are needed to create one triangle, this value is calculated as stated in line three (=`facevertexcount`). `FirstIndex` states where to find the first index in an indices buffer. As the `triangleStartIdx` gives the first index of the triangles, the `FirstIndex` is calculated as stated in line five (=`firstIndex`). The `BaseVertex` gives the location of the first vertex of the object and is always zero in this application.

```
 1 (*creates a draw call based on a given instance count, first instance, and
       group information*)
 2 let createDrawCallWithBaseInstance (instanceCount : int) (firstInstance : int
       ) (groupInfo : GroupInformation) =
 3     (*number of vertices needed to create one face*)
 4     let facevertexcount = groupInfo.elemCount * 3
 5     (*location of the first index*)
 6     let firstIndex = groupInfo.triangleStartIdx * 3
 7
 8     (*return draw call information*)
 9     DrawCallInfo(
10         FaceVertexCount = facevertexcount,
11         FirstIndex = firstIndex,
12         InstanceCount = instanceCount,
13         FirstInstance = firstInstance,
14         BaseVertex = 0)
```

Listing 5.13: Example of how to create the draw call information for indirect rendering.

Now that all pre-processing steps are explained, they can be put together to create the scene graph node for the *VRMeshType*. Listing 5.14 contains the composed code. First, the separation of the `material groups` and `object groups` is done into textured and untextured groups (`texturedGroups, untexturedgroups` in the code). Afterwards, it is determined if the *VRMeshType* contains `matrices`. If this is the case, the `matrices` are matched to the corresponding `object groups` with the function `mapMatricesPerGroup` (see Listing 5.12). `matricesPerMeshGroupIndex` contains for each `object group` a list of indices into the `matrices`. This list of indices identifies which `matrices` are used to render a certain `object group` multiple times. The *VRMeshType* contains a color attribute to color the geometry based on per group colors or per vertex colors. The function `createColorBuffer` in line 22 is used to create the correct buffer storing the correct colors for the individual objects. After all the prerequisites are done, the draw calls are created. For this purpose, the `texturedGroups` and `untexturedGroups` are iterated and the appropriate indirect draw calls (instanced or not) are created. This is done from line 27 to 33 for the `texturedGroups` and from line 36 to 42 for the `untexturedGroups`. At this point, two distinctive sets of indirect draw calls exist: `texturedDrawCalls` and `untexturedDrawCalls`. These draw calls are combined to one set of draw calls (`drawCallsWithoutVertexAttributes`). In line 50 all the necessary attributes, like vertices, indices, normals, instances, uv-coordinates, and colors are added as buffers to the draw call set. Each individual draw call contains the information where to find the corresponding data in the attached buffers to render the geometry correctly. Through these preparations of the *VRMeshType* data only one draw call is necessary to send the whole data to the graphic hardware instead of multiple draw calls. This is possible through using the multi-draw indirect graphic command [mul]. In the Evaluation (see Chapter 6), a performance comparison between a naive implementation and this implementation is given. This implementation ensures an efficient rendering of the data and is the key for producing high frame rates.

```
 1 let createMeshScenegraphBasedOnGroups (meshdata : VRMeshType) =
 2     (*dictionary used to identify instanced object groups of the mesh*)
 3     let matricesPerMeshGroupIndex = ref(Dictionary())
 4     (*extract object groups from mesh*)
 5     let objectGroups = meshdata.objectGroups
 6     (*extract instances attribute from mesh*)
 7     let instances = meshdata.instances
 8
 9     (*match material groups to general object groups*)
10     let texturedGroups, untexturedGroups, objectGroupPerMaterial =
           checkRangeOfMaterialGroups materialInfos objectGroups
11
12     (*check if the current mesh has matrices as attribute -> instanced
           rendering*)
13     let hasMatrices =
14         if instances.ContainsKey "matrices" then
15             (*if the mesh has matrices, map the matrices to the corresponding
                   object groups*)
16             mapMatricesPerGroup instances objectGroups
                   matricesPerMeshGroupIndex
17             true
18         else
19             false
20
21     (*create a color buffer for the mesh scene graph based on the given
           colors*)
22     let colorBuffer = createColorBuffer meshdata.colors
23
24     (*split scene graph into textured and untextured draw calls*)
25
26     (*create textured draw calls*)
27     let texturedDrawCalls =
        if hasMatrices  then
            (*create for each textured object group an instanced indirect draw call*)
            createTexturedInstancedIndirectDrawCalls texturedGroups
        else
            (*if it is not instanced, create for each textured object group an indirect draw call*)
            createTexturedIndirectDrawCalls texturedGroups

    (*create untextured draw calls*)
    let untexturedDrawCalls =
        if hasMatrices then
            (*create for each untextured object group an instanced indirect draw call*)
            createUntexturedInstancedIndirectDrawCalls untexturedGroups
        else
            (*reate for each untextured object group an indirect draw call*)
            createUntexturedIndirectDrawCalls untexturedGroups

    (*combine the textured and untextured draw calls*)
    let drawCallsWithoutVertexAttributes =
        Sg.ofSeq [
            texturedDrawCalls
            untexturedDrawCalls
        ]

    (*add all necessary vertex attributes (vertices, indices, normals, instance buffer, uv coordinates, colors)*)
    let finalSg = addVertexAttributes drawCallsWithoutVertexAttributes meshdata colorBuffer

    (*add the final scene graph to the rendering*)
    addToRendering finalSg
```

Listing 5.14: Code for creating the scene graph node for a *VRMeshType*.

VRTerrainType

The terrain is one of the essential scene objects. The main task to create the corresponding scene graph node of the *VRTerrainType* consists in calculating the geometry data (vertices and indices). The creation of the node is based on a 2D regular grid. The terrain levels give the corresponding height values for each cell of the grid. Through combining the 2D grid information with the terrain levels, a 3D surface reconstruction can be performed. In order to combine this information, first the vertices of the grid are calculated. At this time, all vertices have the same height resulting in a flat surface. An additional single channel texture stores the corresponding terrain level values. In a vertex shader, these terrain levels are matched to the correct vertices.

Listing 5.15 shows how the creation of the vertices is done. The result is a list of all vertices for a grid with the given `dimension = (X_1, Y_1)`. The `origin = (X_2, Y_2)` gives the location where the terrain starts. As this does not have to be at `(0, 0)`, the vertices are translated by the origin to get the correct world position. Additionally, the distance between two vertices is specified by the `cellSize = (X_3, Y_3)`. Multiplying the `x` and `y` coordinate with the `cellSize` ensures that the cells of the grid have the desired size.

```
let vertices =
    [|
    for y in 0 .. dimension.Y
        for x in 0 .. dimension.X
            let vertexX = origin.X + (x * cellSize.X)
            let vertexY = origin.Y + (y * cellSize.Y)
            yield V4f(vertexX, vertexY, 0.0, 1.0)
    |]
```

Listing 5.15: Function for calculating the vertices of a regular grid.

The terrain levels are stored in a separate 1D-array. One possibility would be to match for each x and y value the corresponding terrain level in the 1D-array during the calculation of the vertices. This means, that the mapping is performed on the CPU. Another method is to put the terrain levels into a single-channel texture and pass it to the GPU. In an appropriate vertex shader, only a texture look-up is done to match a terrain level to the corresponding vertex. This method of mapping the terrain levels is based on the presented concept by Livny et al. [LSGES08]. Livny et al. use the GPU to map the height values to the vertices depending on the current view point and level of detail as it is faster than performing it on the CPU. Instead of a view-dependent mapping of the height values, the mapping of the terrain values is view-independent and has only one level. This means, for each vertex one terrain value is assigned and stored as the height of this vertex. As Aardvark only re-draws those parts of the scene, which have changed since the last rendering, this mapping is only done if the vertices of the terrain change. After matching the terrain levels to the corresponding vertex in the vertex shader, a reconstruction of the normals is necessary for proper lighting. This is done in the fragment shader.

```
let indices =
    [|
    for y in 0  .. dimension.Y-1 do
        for x in 0 .. dimension.X-1 do
            let tl =  ((y+1) * (dimension.X+1) + x)
            let bl =  (y * (dimension.X + 1) + x)
            let br =  (y * (dimension.X + 1) + (x+1))
            let tr =  ((y+1) * (dimension.X + 1) + (x+1))

            yield! [|V3i(bl, br, tr); V3i(bl, tr, tl)|]
    |]
```

Listing 5.16: Function for calculating the indices necessary for triangulation.

In addition to the vertices, indices are needed for the geometry triangulation. Listing 5.16 shows the corresponding calculations. For this, only the dimension of the grid is needed. In the double for loop, the corresponding indices for the produced vertices are calculated. In the Listing, the indices for two triangles per vertex are calculated and stored in two vectors of data type integer. These vectors are stored in an array containing all indices in the end.

In addition to the geometry information, a texture is available. This texture is used to color the terrain geometry. It is stored in an additional texture buffer and uploaded to the GPU. In the fragment shader, the texture is mapped onto the terrain geometry by a simple texture look-up. Aardvark tracks this texture buffer. The *VRTerrainType* represents one scene object. The vertices and indices are stored in corresponding buffers. Based on this information, a one single direct indexed draw call is used to create the scene graph node.

If the Visdom Server sends a new texture for the terrain, only the new data are uploaded to the correct texture buffer. As the changes are propagated, the rendering system updates the scene rendering accordingly. If the Visdom Server sends new terrain information, containing different grid data, the vertices and indices have to be recalculated. Through updating the corresponding buffers of the scene graph node, the scene rendering is updated too by Aardvark. This updating of the terrain geometry interrupts the VR experience.

VR WaterType

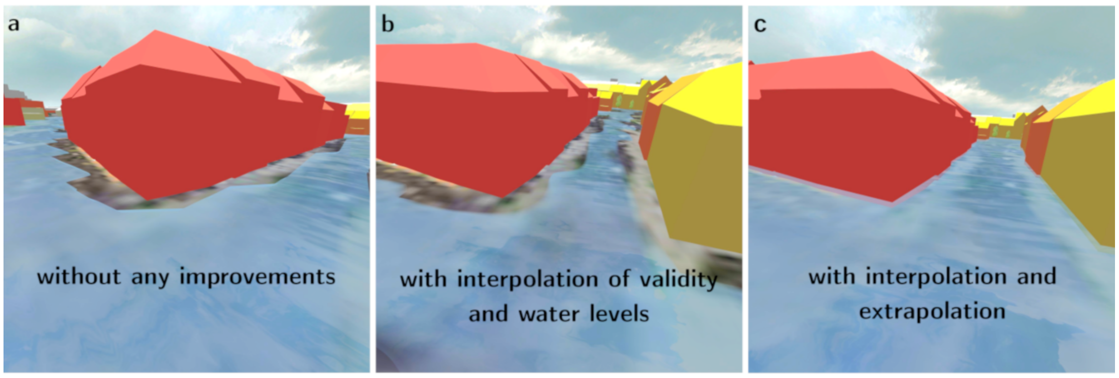

a
without any improvements
b
with interpolation of validity and water levels
c
with interpolation and extrapolation

Figure 5.11: Screen shots visualizing different types of water rendering. (a) is without any rendering optimizations. (b) shows smoother contours through interpolation. (c) uses interpolation and extrapolation of the boundary values. This closes the gap between water boundary and buildings.

The *VRWaterType* is very similar to the *VRTerrainType*. The vertices and indices are calculated the same way as it is done for the *VRTerrainType*. The water levels are stored in a texture and uploaded to the GPU. The water covers only parts of the terrain. In order to distinguish between water covered cells of the underlying grid and dry ones, the validity is used. As like the water levels, the validity is stored in a single-channel texture and uploaded to the GPU. In a fragment shader, the validity is used to identify the valid water geometry. The water levels are used to reconstruct this valid water geometry by adding the correct absolute water height values per fragment.

Visdom performs the simulation calculations on a discretized simulation domain. This causes two issues for the rendering of the water surface. Figure 5.11a illustrates the first issue. The rendered geometry follows an edgy contour if no rendering improvements are done. In order to improve the rendering, interpolation is used to produce smoother contours, as visualized in Figure 5.11b. With the use of interpolation the first issue can be removed but it does not eliminate the second one. The water does not reach the boundaries of the buildings, either in Figure 5.11a nor in Figure 5.11b, even if they are flooded. As this visualization breaks the plausibility of the water rendering, an additional extrapolation step is used to expand the rendered water surface. Figure 5.11c shows the rendered water geometry without the introduced issues. For the Visdom Client these interpolation and extrapolation steps are done for a visual plausible rendering, as introduced by Cornel et al. [CBKK$^+$]. For the VR Application, the rendering is done on the VR Client with a different rendering engine. In order to produce a similar water rendering result as for the Visdom Client, these interpolation and extrapolation concepts are used.

Before explaining the details on how these concepts are implemented, some pre-processing steps of the data are necessary. After processing, they can be uploaded to the GPU. The vertices and indices for the water scene graph node are calculated in the same way as it is done for the *VRTerrainType*. This results in a triangulated grid as it is illustrated in Figure 5.12d. The blue colored regions represent the water covered area. As the water covers only portions of the underlying grid, further processing steps are necessary. Figure 5.12a shows the non-triangulated grid structure used to represent the water. The blue colored cells represent the water covered ones (= wet cells). In the end, only the wet cells are rendered, as illustrated in Figure 5.12e. In order to distinguish between wet and dry cells the validity is used.

For the given grid in Figure 5.12a, with grid cell indices ranging from 0 to 29, the corresponding validity contains the following values `validity = [1, 2, 3, 18, 19, 23, 24, 26, 27, 28, 29]`. In this form the representation is called sparse, as only those cell indices are stored, which are valid. As for the rendering the vertices and indices are calculated for the whole grid, this representation is converted into a dense form. This means, for each cell of the grid a corresponding validity value is stored in a texture of the size of the grid. Figure 5.12b shows the dense representation of the validity. Wet cells are represented by a one. The cells containing a zero are invalid or dry. In order to obtain this dense representation as illustrated in Figure 5.12b, a conversion of the sparse validity is done. As only as many water levels are available as validity values, the same conversion from sparse to dense is done for the water levels. Instead of storing ones and zeros in the texture, the given water levels are stored. Figure 5.12c shows the dense water levels with exemplary data. This conversion step is done every time new validity values and water levels are available. As this is in the worst case every new frame, the conversion has to be done as fast as possible to provide a water flow animation in VR. Because of this performance requirement, the calculation is shifted to the GPU instead of using the CPU. In the evaluation in Chapter 6, a performance comparison of the CPU and the GPU implementation is given. Listing 5.17 gives the used compute shader to perform the conversion of the validity values and water levels written in FShade [fsh].

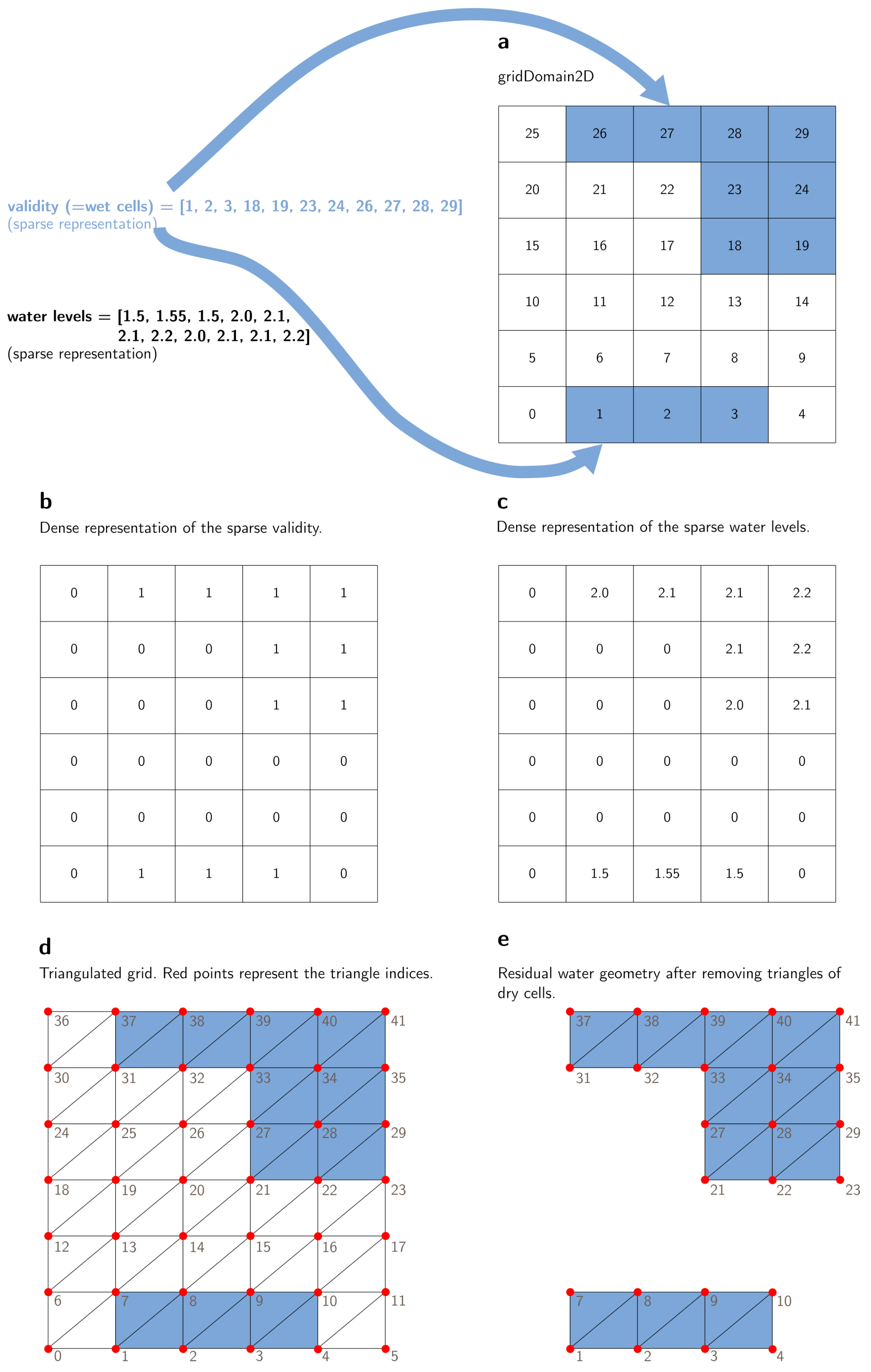
```
[<LocalSize(X = 64)>]
let expansionComputeShader (n : int) (sparseValidity : int[]) (waterleveldata
     : float32[]) (denseValidity : Image2d<r32f>)  (denseWaterlevels :
    Image2d<r32f>) =
    compute{
        let id = getGlobalId().X
        let dim = denseValidity.Size

        if id < n then
            let validIndex = sparseValidity.[id]
            let y = (dim.Y-1) - validIndex / dim.X
            let x = validIndex \% dim.X

        denseValidity.[V2i(x, y)] <- V4d(1.0f)
        denseWaterlevels.[V2i(x, y)] <- V4d(waterleveldata.[id])
    }
```

Listing 5.17: Compute shader for converting the sparse validity values and water levels.

The input parameters of the shader consists of five variables. `n` gives the number of available validitiy values as well as the number of available water levels. `sparseValidity` stores the validity as a list of integers. `waterleveldata` stores the corresponding water levels per valid index. `denseValidity` is the texture storing the dense representation of the validity (see Figure 5.12b). `denseWaterlevels` is the textures storing the dense representation of the water levels (see Figure 5.12d). Both, the `denseValidity` and `denseWaterlevels`, have the same size as the underlying grid used for the water. For both textures a single-channel texture is used and are initialized with zeros. The code itself is not complex. Based on the current `id` of the processing thread, one `validIndex` is extracted from the list of `sparseValidity`. As the indices are numbered sequentially, the x and y coordinate of the `validIndex` is reconstructed in order to get the coordinates of the corresponding pixel. The pixel with the calculated pixel position is then set to one in the `denseValidity` texture, as it represents a wet cell. Additionally, the corresponding water level is stored in the `denseWaterlevels` texture at the same pixel position. With the `denseWaterlevels`, the vertex positions are reconstructed like it is done for the terrain. The `denseValidity` is used within the fragment shader to determine, which fragments of the whole water geometry represent water. With this texture it is possible to reduce the water geometry as visualized in Figure 5.12d to only show the valid ones as visualized in Figure 5.12e.


a
gridDomain2D
validity (=wet cells) = [1, 2, 3, 18, 19, 23, 24, 26, 27, 28, 29]
(sparse representation)
water levels = [1.5, 1.55, 1.5, 2.0, 2.1, 2.1, 2.2, 2.0, 2.1, 2.1, 2.2]
(sparse representation)
b
Dense representation of the sparse validity.
c
Dense representation of the sparse water levels.
d
Triangulated grid. Red points represent the triangle indices.
e
Residual water geometry after removing triangles of dry cells.

25	26	27	28	29
20	21	22	23	24
15	16	17	18	19
10	11	12	13	14
5	6	7	8	9
0	1	2	3	4

0	1	1	1	1
0	0	0	1	1
0	0	0	1	1
0	0	0	0	0
0	0	0	0	0
0	1	1	1	0

0	2.0	2.1	2.1	2.2
0	0	0	2.1	2.2
0	0	0	2.0	2.1
0	0	0	0	0
0	0	0	0	0
0	1.5	1.55	1.5	0

Figure 5.12: Illustration of the necessary steps to process the validity and water levels for the water scene graph node. (a) shows the underlying grid representing the water domain. The blue cells are the wet cells, identified by the validity. (b) visualizes the used texture to identify the wet cells (= 1) and the dry ones (= 0). (c) contains the water levels for all cells of the underlying grid stored in a texture. (d) shows the triangulated grid used as basis for the water rendering. (e) shows the residual triangles, after removing the triangles of dry cells. This is done by using the texture of b.

The fast performance of this data conversion alone is not sufficient when updating the water scene graph node. In order to avoid flickering artifacts, the updating of the old data to the new ones has to be done efficiently. For this reason, we use double buffering [Rya]. The main idea of double buffering is to use two buffers, called the front buffer and the back buffer. Only the front buffer is used to visualize the data on the screen. If a scene has to be re-drawn, the back buffer is used for this update. As soon as the updating of the back buffer is finished, the two buffers are swapped. Thus, the back buffer containing the new data to draw the scene becomes the new front buffer and the former front buffer becomes the back buffer, ready for getting new data again. The same idea is used for updating the texture buffers storing the water levels and the validity values. This means, instead of two textures four are needed, as one pair of front and back buffer for the water levels and one pair for the validity values.

In order to reuse the back buffers again, the stored textures have to be cleared as they store the previous water levels and validity values. If this would not be done, the upcoming result would be wrong. An additional compute shader is used to clear the data stored in the texture buffers. The used compute shader looks similar to the compute shader used to convert the water levels and validity values (see Listing 5.17) and is given in Listing 5.18. By setting the content of the textures to zero, the buffers are cleared and can be refilled again without distorting new data of new frames.

```
[<LocalSize(X = 8, Y = 8)>]
let clearTextureComputeShader (denseValidity : Image2d<r32f>) (
    denseWaterlevels : Image2d<r32f>) =
    compute{
        let id = getGlobalId().XY
        let indicesDim = denseValidity.Size

        if id.X < indicesDim.X && id.Y < indicesDim.Y then
            denseValidity.[id] <- V4d(0.0f)
            denseWaterlevels.[id] <- V4d(0.0f)
    }
```

Listing 5.18: Compute shader for clearing the water level texture and the validity texture.

Through updating the water data with the introduced method, an interactive animation of a flooding scenario in the virtual world can be viewed without breaking the sense of immersion. Additionally, one special case has to be considered when updating the texture buffers. If the data of the underlying grid change, the sizes of the used texture buffers have to be updated too. This most likely happens at scenario changes.

After all geometry data are calculated, the next step is about rendering the data. In order to remove the edgy contour of the water boundary, as visualized in Figure 5.11a, a reconstruction of the water surface is done [CBKK$^+$]. Therefore, a piecewise bi-cubic interpolation with a cubic B-splines kernel of the water levels and the validity is performed in the fragment shader. Listing 5.19 shows the used code. Additionally, in the same function, the surface normal is reconstructed. The presented code could be further improved by exploiting this linear texture filtering already performed by the GPU, as presented by Sigg and Hadwiger [SH05]. This would reduce the number of texture look-ups and increase the performance of the filtering.

```
let textureBSplineValueAndNormal (texture : Sampler2d) (textureCoord : V2f) =
    (*calculate texture data*)
    let textureDimensions = texture.GetSize()
    let texelSize = 1.0f / V2f(textureDimensions)
    let unnormalizedTextureCoordinates = V2f(textureDimensions) *
        textureCoord
    let texelOrigin = floor unnormalizedTextureCoordinates
    let texelCenter = texelOrigin + 0.5f

    (*calculate coordinates*)
    let bottomLeftOffset = lerp (V2f(0.0)) (V2f(-1.0)) (lessThanEqual
        unnormalizedTextureCoordinates texelCenter)
    let bottomLeftTexelOrigin = texelOrigin + bottomLeftOffset
    let bottomLeftTexelCenter = bottomLeftTexelOrigin + 0.5f
    let bottomLeftTexelUpper = bottomLeftTexelOrigin + 1.0f
    let normalizedBottomLeftTexelUpper = texelSize * bottomLeftTexelUpper

    (*extract 4 x 4 neighboring texture values*)
    (*GatherOffset returns neighboring texel values with an offset,
      the values are returned in the order (x = topleft, y = topright, z =
          bottomright, w = bottomleft) and are stored in a vector*)
    let bottomLeftValues = texture.GatherOffset((V2d(
        normalizedBottomLeftTexelUpper)), V2i(-1, -1))
    let bottomRightValues = texture.GatherOffset((V2d(
        normalizedBottomLeftTexelUpper)), V2i(1, -1))
    let topLeftValues = texture.GatherOffset((V2d(
        normalizedBottomLeftTexelUpper)), V2i(-1, 1))
    let topRightValues = texture.GatherOffset((V2d(
        normalizedBottomLeftTexelUpper)), V2i(1, 1))

    (*calculate difference between actual texture coordinate and center*)
    let f = unnormalizedTextureCoordinates - bottomLeftTexelCenter
    (*calculate second and third powers for the filter weights*)
    let f2 = f * f
    let f3 = f2 * f;

    (*calculate additional values for the filter weights*)
    let oneMinusF = V2f(1.0) - f
    let oneMinusF2 = oneMinusF * oneMinusF
    let oneMinusF3 = oneMinusF2 * oneMinusF

    let ONE_OVER_SIX_VEC2 = V2f(1.0 / 6.0)
    let ONE_OVER_TWO_VEC2 = V2f(1.0 / 2.0)
    let TWO_OVER_THREE_VEC2 = V2f(2.0 / 3.0)

    (*calculate filter weights*)
    let w0 = ONE_OVER_SIX_VEC2 * oneMinusF3
    let w1 = ONE_OVER_TWO_VEC2 * f3 - f2 + TWO_OVER_THREE_VEC2
    let w2 = ONE_OVER_TWO_VEC2 * oneMinusF3 - oneMinusF2 +
        TWO_OVER_THREE_VEC2
    let w3 = ONE_OVER_SIX_VEC2 * f3
    let w4 = V2f(-0.5) * oneMinusF2
    let w5 = V2f(1.5) * f2 - V2f(2.0) * f
    let w6 = V2f(0.5) * oneMinusF * (V2f(3.0) * f + V2f(1.0))
    let w7 = V2f(0.5) * f2;

    let transposedXWeights = V4f(w0.X, w1.X, w2.X, w3.X)
    let transposedYWeights = V4f(w0.Y, w1.Y, w2.Y, w3.Y)

    (*perform interpolation, use dot products instead of sums and
       multiplications
      the values of the obtained texels in bottomLeftValues,
          bottomRightValues, topLeftValues, topRight values are in the order
           (x = topleft, y = topright, z = bottomright, w = bottomleft)*)
    (*filter in x direction*)
    let x = V4f(
        (Vec.dot transposedYWeights (V4f(bottomLeftValues.W, bottomLeftValues
            .X, topLeftValues.W, topLeftValues.X))),
        (Vec.dot transposedYWeights (V4f(bottomLeftValues.Z, bottomLeftValues
            .Y, topLeftValues.Z, topLeftValues.Y))),
        (Vec.dot transposedYWeights (V4f(bottomRightValues.W,
            bottomRightValues.X, topRightValues.W, topRightValues.X))),
        (Vec.dot transposedYWeights (V4f(bottomRightValues.Z,
            bottomRightValues.Y, topRightValues.Z, topRightValues.Y))))
    (*filter in y direction*)
    let y = V4f(
        (Vec.dot transposedXWeights (V4f(bottomLeftValues.W, bottomLeftValues
            .Z, bottomRightValues.W, bottomRightValues.Z))),
        (Vec.dot transposedXWeights (V4f(bottomLeftValues.X, bottomLeftValues
            .Y, bottomRightValues.X, bottomRightValues.Y))),
        (Vec.dot transposedXWeights (V4f(topLeftValues.W, topLeftValues.Z,
            topRightValues.W, topRightValues.Z))),
        (Vec.dot transposedXWeights (V4f(topLeftValues.X, topLeftValues.Y,
            topRightValues.X, topRightValues.Y))))

    (*reconstruct surface normal*)
    let n1 = (Vec.dot (V4f(w4.X, w5.X, w6.X, w7.X)) x )
    let n2 = (Vec.dot (V4f(w4.Y, w5.Y, w6.Y, w7.Y)) y)

    let normal = V3f(texelSize * (V2f(n1, n2)), 1.0f).Normalized

    (*reconstruct height value through filtering in y direction*)
    let height = Vec.dot transposedYWeights y

    (*return the interpolated height value combined with the reconstructed
       normal as one vector*)
    V4f(height, normal.X, normal.Y, normal.Z)
```

Listing 5.19: Function for interpolating a texture value and reconstructing the corresponding normal.

First some texture related values are calculated to compute the texture coordinates. Afterwards, the 16 neighbors of the current texture position are extracted. The values are stored in four four-dimensional vectors (`bottomLeftValues`, `bottomRightValues`, `topLeftValues`, `topRightValues`). These vectors store the texel value in the order (`x = topleft, y = topright, z = bottomright, w = bottomleft`). The next step calculates the B-Spline weights for filtering. The weights are used to calculate the filtering once in the x direction and once in the y direction. Through using dot products instead of writing the linear equations line by line, the calculations are performed faster. With this values the surface normal can be reconstructed. In order to obtain the interpolated height, an additional filtering in y direction is necessary.

The interpolation of the water levels and validity values creates smoother water boundaries. Nevertheless, because of the discretization of the simulation domain, the water surface does not reach the walls of the surrounding buildings, even if they are flooded. For this issue extrapolation is used to close the gaps between water surface and surrounding scene objects, as presented by Cornel et al. [CBKK$^+$]. The main objective is to consider all wet cells neighboring a dry cell. The water levels of these neighboring wet cells are summed. This value is divided by the number of neighboring wet cells contributing to the sum. The resulting value is an average water level value of the neighboring wet cells. This average water level value is applied to the dry cell under consideration. The used neighborhood for the extrapolation depends on the cell size of the underlying grid. Through this additional step, it is possible to extend the water covered area and create a more realistic water surface, especially next to buildings (see Figure 5.11c). Listing 5.20 shows the used compute shader. `checkValidityAndLevelOfNeighboringTexels` is a function to check the validity and water levels of the defined neighboring texels.

```
[<LocalSize(X = 8, Y = 8)>]
let waterExtrapolation (denseValidity : Image2d<Formats.r32f>) (
    denseWaterlevels : Image2d<Formats.rg32f>) =
    compute{
        let id = getGlobalId().XY
        let dim = denseValidity.Size

        if id.X < dim.X && id.Y < dim.Y then
            (*if current indices are invalid*)
            if denseValidity.[id].X = 0.0 then
                let mutable numberOfValidNeighbors = 0
                let mutable sum = 0.0

                (*checks the validities of the neighboring texels, increases
                    the numberOfValidNeighbors accordingly and sums the water
                     levels of the wet cells*)
                checkValidityAndLevelOfNeighboringTexels denseValitiy
                    denseWaterlevels numberOfValidNeighbors sum

                (*calculate the new water level value*)
                let originalWaterlevel = denseWaterlevels.[id]
                let avg =
                        if numberOfValidNeighbors > 0 then
                            sum / float numberOfValidNeighbors
                        else
```

```
                              originalWaterlevel.X

                  (*replace the old value with the new one*)
                  denseWaterlevels.[id] <- avg
    }
```

Listing 5.20: Compute shader function to extrapolate water level data from wet cells to dry cells.

VRLineType

The simplest data object used is the *VRLineType*. Aardvark already supports the creation of a line scene graph node based on a set of points. For each line that is visible in the scene, a line scene graph node is created based on the given list of points. Through providing this point list to Aardvark, an indexed geometry is created and is directly rendered.

CHAPTER 6

Evaluation

The evaluation of the implemented VR application is done by some performance measurements. For this purpose two different criteria are used. The first one is about the achieved frame times of the application in a steady condition. This means, that a scenario has been fully loaded and the user is able to move around the virtual world. The second criteria is about the processing time of the simulation data. This includes parsing and updating newly received simulation data. Both criteria are mainly influenced by the size of the scenario as well as the number of objects in it, like buildings, barriers, bridges, and trees. As a consequence, four different scenarios are used for the evaluation of the implemented system. Table 6.1 gives an overview about important data of the scenarios. Additionally, the Figures 6.1 and 6.2 show a screen shot for each scenario in the VE from an overview perspective.

Scenario	**S1**	**S2**	**S3**	**S4**
Terrain (Grid Area)	2,95 x 2,37 km^2	6,01 x 8,51 km^2	5,01 x 6,8 km^2	6,23 x 6,21 km^2
Cell Size	5 m	10 m	10 m	10 m
#Cells	279 660	511 451	340 680	386 883
#Wet Cells	76 848	90 391	67 994	22 651
#Buildings	5 696	55 487	37 889	5 038
#Bridges	6	6	6	0
#Trees	6 008	38 607	30 795	81 693

Table 6.1: Overview about the data of the different scenarios used for the evaluation.

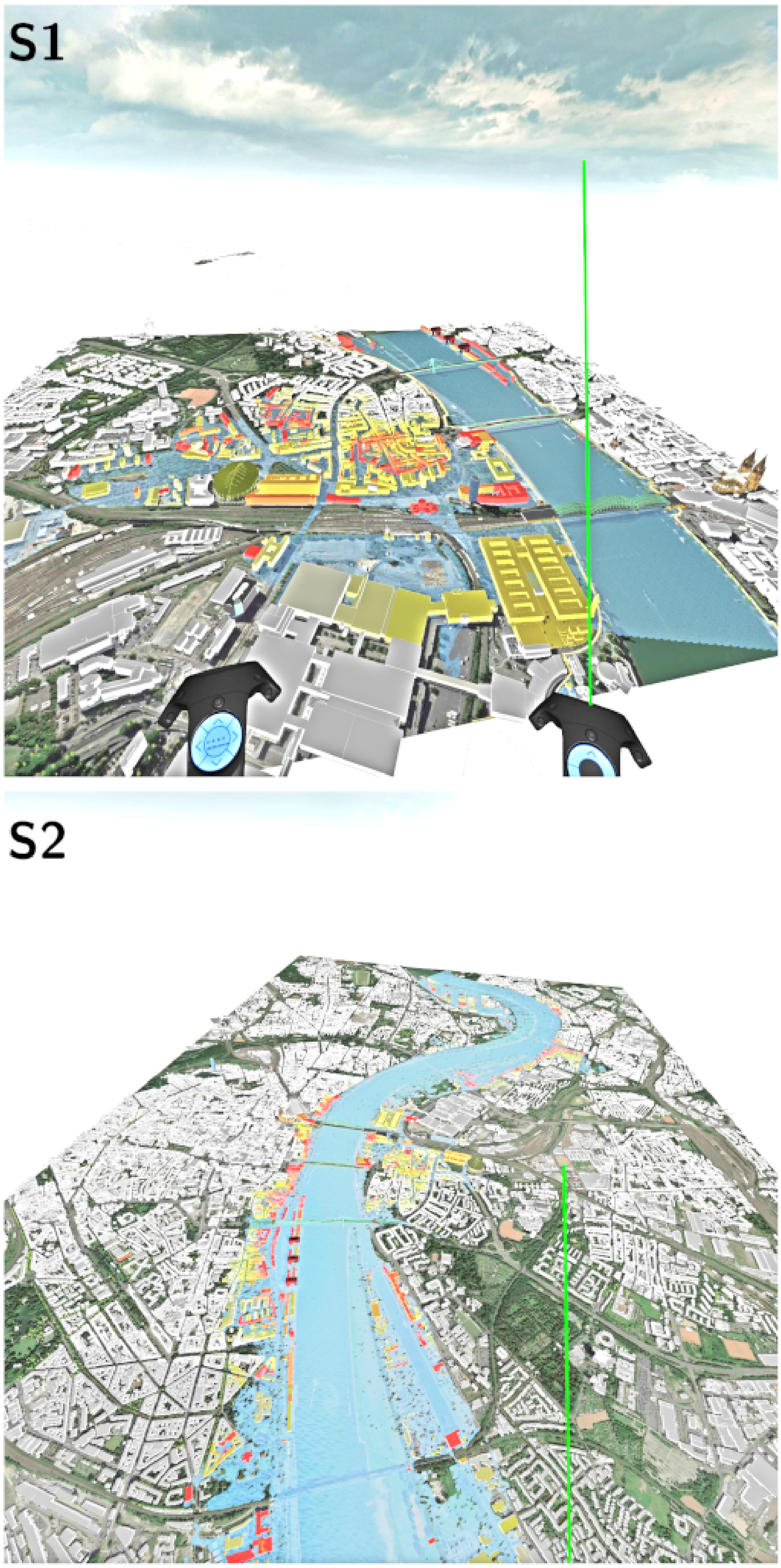


Figure 6.1: Screen shots of the Scenarios S1 and S2 from an overview perspective.

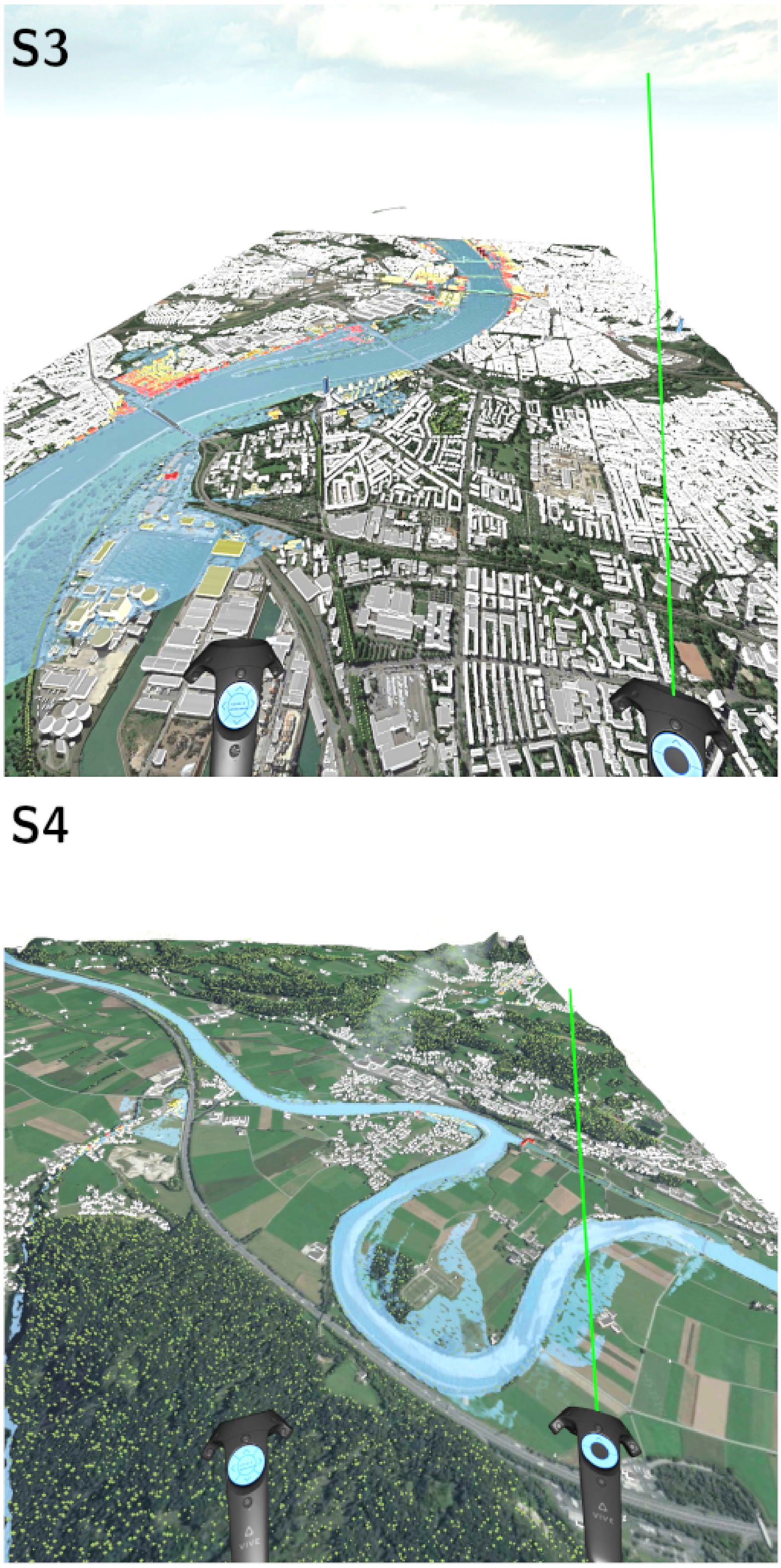


Figure 6.2: Screen shots of the Scenarios S3 and S4 from an overview perspective.

For all four scenarios the same measurements are performed by providing equal conditions. In a first step, the achieved frame times are investigated while the user explores the virtual worlds. Therefore, two different implementation strategies are used and the results are compared. In a second step, important processing times are examined in greater detail. The used test PC for the evaluation runs Widnows 10 Pro as operating system. It is equipped with 32 GB RAM and an AMD Ryzen Threadripper 1900X 8-Core Processor with 3.80 GHz and two Nvidia GeForce GTX 1080 Ti GPU. For the evaluation only one graphic card is in use.

6.1 Frame Times in the Virtual World

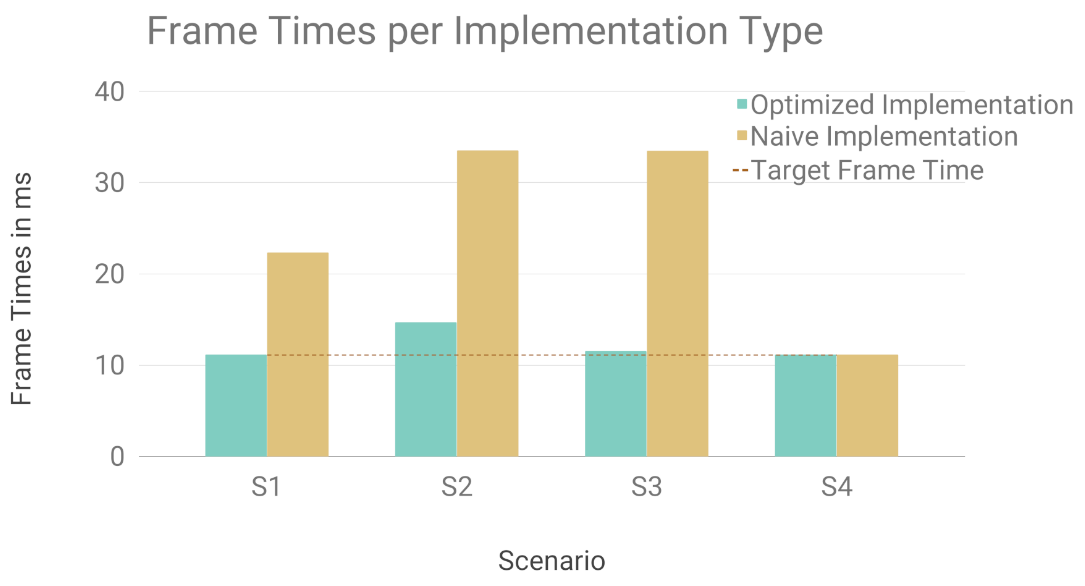


Figure 6.3: The achieved Frame Times per Scenario and Implementation Type. The Optimized Implementation uses multi-draw indirect [mul] to draw objects of the *VRMeshType*. The Naive Implementation issues for each `object group` of the *VRMeshType* one indirect draw call.

Low frame rates can reduce the sense of feeling immersed [vrsb]. In order to provide a good VR experience, high frame rates are desired. The displays of the HTC Vive have a refresh rate of 90 Hz [HTCa]. This means that about every 11th millisecond a new frame is provided. Therefore, the goal is to achieve this frame time in order to be able to explore the virtual world of the flood scenario without any disruptions. The focus of this part of the evaluation lies on the frame times of the VR application after a flood scenario has been loaded. As soon as the scene graph is fully created and rendered on the VR displays, the frame times are measured for about 30 seconds. For all four scenarios this is done twice, once for the Naive Implementation and once for the Optimized Implementation. The two implementation techniques concern only the *VRMeshType.* The Naive Implementation creates for each `object group` of a mesh an individual indirect draw call. The Optimized Implementation uses the multi-draw indirect [mul] command to render all available `object groups` of a mesh. This issues only one draw call to the GPU compared to the Naive Implementation. The Optimized Implementation is used for this VR application.

In Figure 6.3, the achieved frame times are visualized. The brownish colored dashed line represents the targeted frame time of 1/90th of a second. For the Scenarios S1 and S4 this target frame time could be achieved by using the Optimized Implementation. In comparison to the Naive Implementation, for Scenario S1 the frame time is higher than the targeted frame time, but for Scenario S4 also the Naive Implementation achieves the desired time. For the Scenarios S2 and S3 the Naive Implementation produces three times higher frame times than the targeted value. Although, it has not been possible to reach the target frame time with the Optimized Implementation for the two Scenarios, an improvement of the times can be recorded.

Scenario S1 has the smallest terrain from all used scenarios, but has the smallest cell size. A smaller cell size means a better resolution and more details. With this scenario characteristics, the desired frame time can be achieved. Scenario S2 has the largest terrain and the highest amount of additional scenario objects. With this scenario setup only a performance improvement can be achieved. Scenario S3 has the second smallest grid area in this evaluation with a resolution of 10×10 meters. The targeted frame time could not be entirely achieved with the Optimized Implementation. Scenario S4 has a slightly larger grid area as S3 but the same resolution. In comparison to Scenario S3, the targeted frame time could be achieved with the Optimized Implementation but also with the Naive one. This circumstance let us assume that the frame times do not necessarily correlate with the size of the grid area. In sum, Scenario S4 contains more additional scenario objects of the *VRMeshType* than Scenario S3, but S3 visualizes seven times more Buildings and six Bridges than S4. As Scenario S4 has a lower frame time than Scenario S3, this behavior could be possibly explained based on the complexity of the geometry necessary to render Buildings and Bridges compared to Trees. In order to make any conclusions, further investigations are necessary in the future.

During the evaluation of the frame times another insight could be gained. The different implementations seem to affect the Start-up Time of the application itself. Based on this finding, further measurements have been done to investigate this matter. For each scenario, the initial Start-Up Time to create the whole scene graph has been measured. Figure 6.4 shows the measured timings per scenario. It was assumed that scenarios containing a higher number of scene graph nodes of the *VRMeshType* yields a higher Start-Up Time. This assumption could not be confirmed. Scenario S4 was built faster than Scenario S1, but in sum contains more scene graph nodes of the *VRMeshType.* It seems that the Start-Up Time increases with a higher number of Buildings and Bridges in a scenario. This could support the assumption that the complexity of the geometry of the

VRMeshType plays a role for the time measurements. As no further investigation have been done to proof this assumption, this can be considered in future work. However, the visualization shows that all scene graphs per scenario are built faster using the Optimized Implementation.

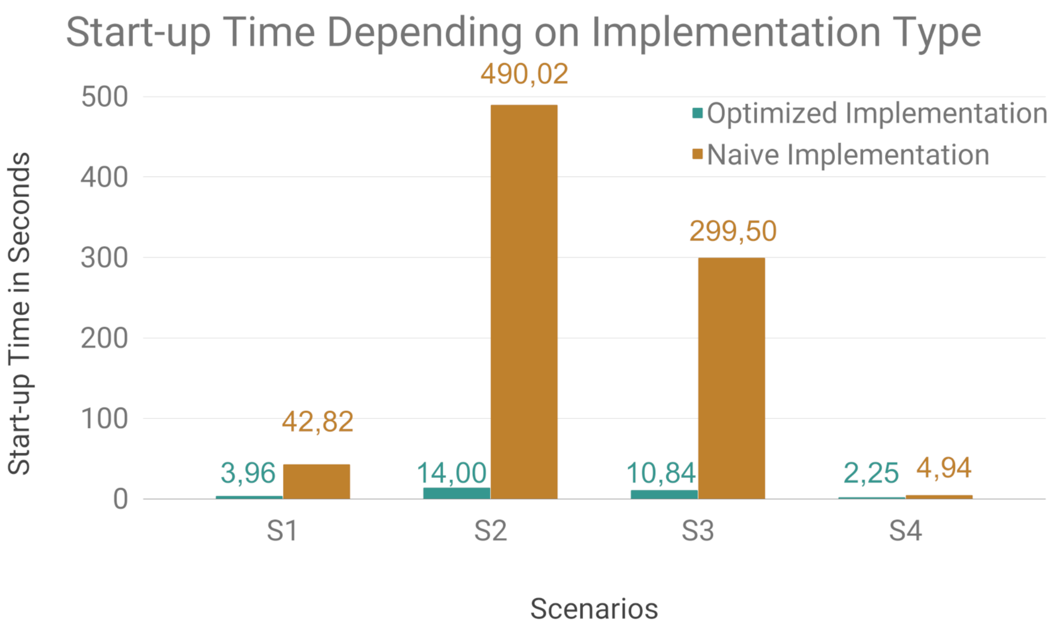


Figure 6.4: Measured Start-up Times to initially create the scene graphs for the individual scenarios.

6.2 Processing Times

The most expensive update of the VR application is an entire scenario update. This mainly affects the whole scene graph. The computational power is used to process the new data and update the old ones, while the VR application runs. Additionally, the entire visible scenario is exchanged and the user is placed in a different environment. This scenario change can hardly be done without breaking the sense of immersion. As a consequence, the focus of this work is to speed-up the processing times with respect on intermediate data updates. This concerns data updates after the main scenario data has been parsed and the scene graph has been created. Based on this considerations, this part of the evaluation focuses on the processing times of new data.

6.2.1 Scenario Data Processing Times

The first section of this evaluation part gives an impression how long it takes to process the scenario data the first time compared to only data changes. The first response from the Visdom Server to the VR Client is the initial response. It contains the Initial Data. This means, to this point no scenario data have been sent to the VR Client and thereof, no scenario data have been visualized in the virtual world yet. The Changed Data are the data of the following response after the initial one. This means, to this point the scenario data for a certain time step has been sent to the VR Client and is visualized in the virtual world. Figure 6.5 shows the measured timings for these two message types per scenario. For each scenario the whole application has been restarted to ensure equal conditions. Additional, only one subsequent frame has been requested after the initial message to measure the processing time of it. The measurements show that the processing of the Changed Data only needs a fraction of the processing time for the Initial Data. As the processing times do not give any information about how fast the corresponding scene graph nodes are updated, this visualization has only informational purpose. It should give an impression of the relation between the processing times of the Initial Data and the Changed Data.

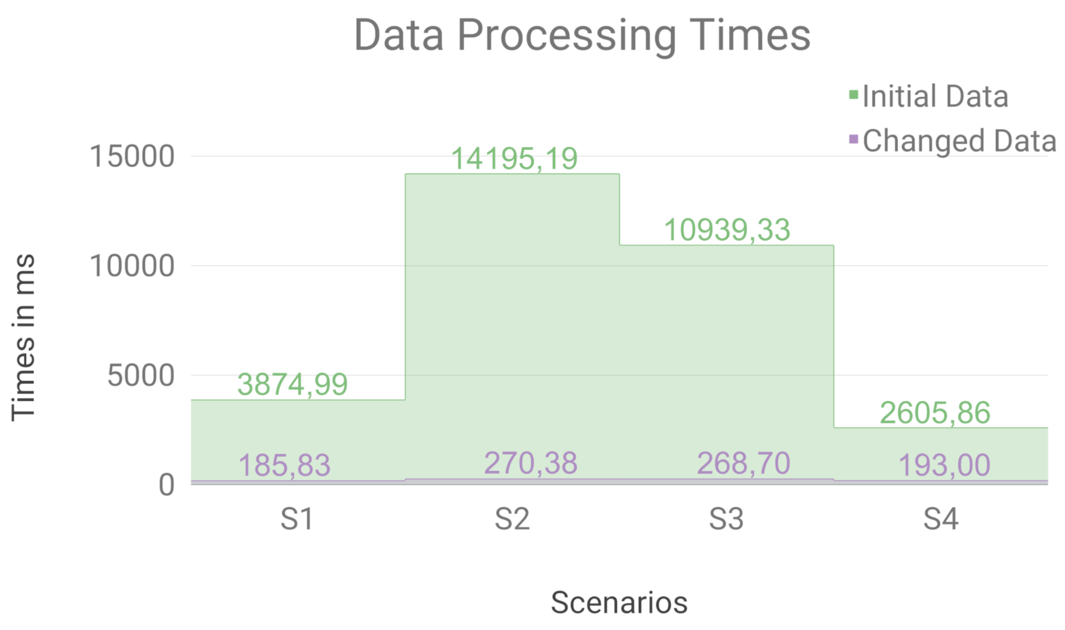


Figure 6.5: Comparison of the processing time of the Initial Data and the Changed Data.

6.2.2 Water Data Processing Times

The most important data that is contained in the Changed Data concerns the water surface. In order to provide an interactive simulation of a flooding scenario, an efficient update of the water scene graph node is inevitable. This updating includes the processing of the received data in a fast way. Based on this requirement, this section of the evaluation focuses on the processing times of the data concerning the water geometry. In order to eliminate a possible disruption by the running VR service during the measuring, the pure processing times of the water data are measured without the rendering time. As the updating of the water scene graph node is of high importance, a fast processing of the validity values and water levels, as described in Section 5.3.2, is necessary. For this purpose a comparison of a CPU and GPU implementation of the data processing is done. Figure 6.6 shows the performance measurements of both implementations per scenario. For all four scenarios, the GPU implementation always outperforms the CPU implementation. Based on this result, the VR Client only uses the GPU implementation for processing the data concerning the water.

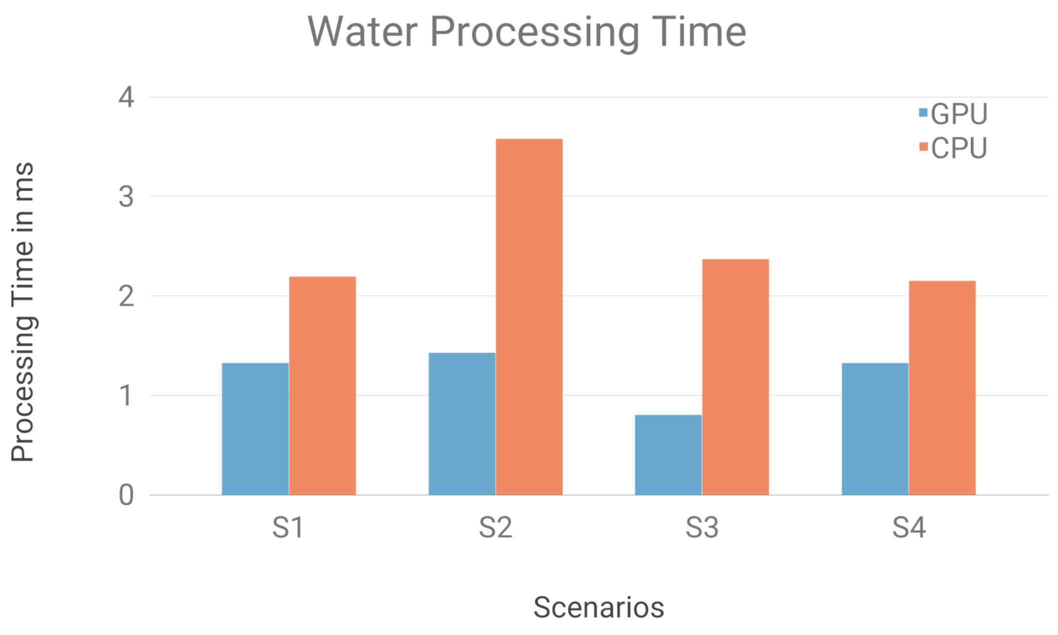


Figure 6.6: Measured times to process the validity values and water levels of the *VRWaterType*.

6.3 User Feedback

Although no explicit user evaluation has been done, feedback has been collected during several test sessions of the VR application. Most of the users complained about the complexity of the interaction interface. This applies especially for the task of placing barriers. Before a user is able to place a barrier, he/she has to decide between a variety of options. First, the user has to choose the type of barrier to place. Afterwards, he/she has to decide how to specify the location of the barriers. After these decisions have been made, the user is finally able to place the barrier. Most of the users were overwhelmed by making these decisions. None of the users, who tested the application, were a domain experts. Based on this feedback, a possible solution is to restrict the choice between different barrier types for non-experts and only let domain expert select in the set of options. For this distinction additional user studies with and without domain experts are necessary.

Additionally, a remark concerning the placement strategies of the barriers is given. The multi-click method and the drawing method were considered to be easier than the 3D placement strategy. Another disruptive factor was, that the user is not able to move through the virtual space with the implemented movement techniques while he/she determines the barrier location. Only the physical movement is possible, but this is limited to the tracking area. This restricts the region where to place the barrier to the current tracking area. Another user noted that some visual feedback would be helpful to determine where he/she touches the trackpad. In general, the distinction between a mode and a steering controller was experienced as too confusing. However, in the scope of this work too few tests have been done in the field of VR interactions to permit firm conclusions.

CHAPTER 7

Conclusion and Future Work

The following section summarizes the implemented VR flood simulation application and the used techniques how this is possible. This summary is followed by a discussion of encountered issues and new possibilities to improve the implemented system.

7.1 Conclusion

This thesis describes a first prototype of a distributed VR application for steering a remote flood simulation. As the basic performance issues have been eliminated, the application setup can be used to view a possible flooding in a 3D virtual world. Through providing different interaction mechanisms, the user is able to control the simulation progress. This is either possible by going back and forth in time or changing between consecutive scenarios. In order to be able to explore the virtual world, different movement techniques are supported. A major goal in flood management is the protection of civilians and the mitigation of negative impacts on the environment. As a consequence, the user is able to place barriers into the virtual world. For this purpose, virtual menus are used. First, the user has to select between different types of protection methods. Second, the location of the barrier has to be determined. As the barrier has been placed, it is taken into account in the further simulation progress. If it has been placed appropriately, eventually the environmental infrastructure will be prevented from getting flooded. The modifications of the scenario are visible in the VR Client but are also propagated to additional connected clients. This is possible as the main simulation happens on a remote state-aware server. As a result, the application can be used either as a single-user application or as a multi-user application. For the single-user application, the standalone VR Client is used. In the multi-user application, the Operator-Trainee setup makes a collaborative work between two users possible.

The main objective of this work was to create a VR application that is capable to visualize a flood simulation in a VE without having performance issues in the sense of low frame rates and thereof loss of immersion. This has been possible through integrating a flood simulation framework with a high-performance rendering engine with VR-Support. The integration happened by implementing a client-server architecture. The existing flood simulation performs the calculations on the server and send the data to the connected client. Through the implementation of a state-aware client it is possible to exploit the structure in the data and use data dependent graphic commands to optimize the rendering performance. In combination with a high-performance rendering engine the rendering performance could be further increased and an improvement in the rendering of different flood simulation events could be achieved. In addition to these technical contributions, user interactions are implemented to no only explore the virtual world but also to manipulate the simulation progress and add protection mechanisms to the simulation scenario for flood protection. Users can use the application in a single-user mode to get a feeling about the flooding situation, but also to train together with an experienced colleague in a multi-user mode.

7.2 Future Work

As the basic step in the direction of a VR flood simulation application has been done, the system can be used for further investigations and improvements in a next step. This includes methods for increasing the rendering performance. Based on the presented evaluation results, the performance seems to decrease with an increasing number of buildings and bridges in the scenario. This needs further investigations for additional optimizations in the future. Another possibility to increase the overall performance is to use two graphics cards instead of one and separate the tasks that have to be done. One card might be responsible for performing the computational tasks and one card for the rendering tasks. Through this separation, it can be ensured that the VR rendering does not get interrupted by other GPU calculation tasks. All the resources can be used to provide a high and stable frame rate.

Visdom is a very complex and powerful simulation framework and comprises a variety of functions. The support of all these functions in VR is not considered to be useful. Instead, the support of a subset of all possible functions has been implemented in this thesis. As the HTC Vive is the used VR Hardware the two controllers are used for the interaction. Therefore, the possible interactions are mapped to the different buttons on the controller. In order to distinguish between the provided functions, each controller has a certain purpose. One serves as the mode controller and the other as the steering controller. The idea was, that only one controller is responsible for interactions with the terrain and performing menu selections, while the other one has the task to control the simulation state and let the menu for barrier selection fade in. The intended benefit of this distinction was a simpler interaction interface, but the opposite effect happened. The interface was considered too complex. This was also said about the process for place barriers. The users criticize that too many choices have to be done before it is possible to place a barrier. Through making an extensive user evaluation with domain experts and non-experts, this interaction method can be further improved in the future.

Another issue is concerned with the controlling of the simulation progress. As the two controllers are responsible for different tasks, it is possible to switch the currently visualized scenario with the mode controller. This action is mapped to the trackpad of the controller. When starting the application, the functionality of the controllers is not clear, as they look identical. As soon as a scene has been loaded, the difference is visible through the different appearance in the VE. A problem that has been recognized is, that even before the scene has been loaded, the user is able to accidentally switch the scenario. This has happened several times during different tests. For small simulation domains this is not a big problem, as the scenario can be loaded relatively fast. For larger ones this can take a while, which can be very annoying. Because of this finding, the method to change between scenarios should be revised in the future. Furthermore, the VR application can be used in researching for new interaction techniques in virtual environments.

To summarize the discussed issues, the main tasks for the future are:

- additional research in the field of performance optimizations
- rethink the provided functionalities, if they are useful in the context of the VE and rework the user interface
- research in the field of user interactions, including a user evaluation with a representative number of people (domain experts and non-experts)

Bibliography

[AA09] Marlon A. Acuña and Takayuki Aoki. Real-Time Tsunami Simulation on Multi-node GPU Cluster. In *ACM/IEEE conference on supercomputing*, 2009.

[aar] An Advanced Rapid Development Visualization And Rendering Kernel. `https://www.vrvis.at/research/projects/aardvark/`, Last visited: 2018-08-16.

[adm] Advanced Disaster Management Simulator. `http://www.trainingfordisastermanagement.com/`, Last visited: 2018-08-04.

[Ari18] Stefan Arisona. The CityEngine VR Experience for Unreal Engine: A Virtual Reality Experience for Urban Planning Applications, 2018. `https://www.esri.com/arcgis-blog/products/city-engine/design-planning/ce-ue4-vr-experience/`, Last visited: 2018-10-27.

[AS06] Sajjad Ahmad and Slobodan P. Simonovic. An Intelligent Decision Support System for Management of Floods. *Water Resources Management*, 20(3):391–410, Jun 2006.

[BH97] Doug A. Bowman and Larry F. Hodges. An Evaluation of Techniques for Grabbing and Manipulating Remote Objects in Immersive Virtual Environments. In *Proceedings of the 1997 symposium on Interactive 3D graphics*, pages 35–39. ACM, 1997.

[BKH97] Doug A. Bowman, David Koller, and Larry F. Hodges. Travel in Immersive Virtual Environments: An Evaluation of Viewpoint Motion Control Techniques. In *Proceedings of the 1997 Virtual Reality Annual International Symposium (VRAIS '97)*, VRAIS '97, pages 45–, Washington, DC, USA, 1997. IEEE Computer Society.

[BM07] Doug A. Bowman and Ryan P. McMahan. Virtual Reality: How Much Immersion Is Enough? *Computer*, 40(7):36–43, July 2007.

[BW01] Doug A. Bowman and Chadwick A. Wingrave. Design and Evaluation of Menu Systems for Immersive Virtual Environments. In *Proceedings IEEE Virtual Reality 2001*, pages 149–156, March 2001.

[CBKK$^+$] Daniel Cornel, Andreas Buttinger-Kreuzhuber, Artem Konev, Zsolt Horváth, Michael Wimmer, and Jürgen Waser. Interactive Visualization of Adaptive Water Height Fields. Submitted to EuroVis 2019.

[Cena] Centre for Research on the Epidemology of Disasters CRED. 2015 disasters in numbers. `www.cred.be/sites/default/files/2015_DisastersInNumbers.pdf`, Last visited: 2018-05-23.

[Cenb] Centre for Research on the Epidemology of Disasters CRED. 2016 preliminary data: Human impact of natural disasters. `www.cred.be/sites/default/files/CredCrunch45.pdf`, Last visited: 2018-05-23.

[Cenc] Centre for Research on the Epidemology of Disasters CRED. Disaster Data: A Balanced Perspective. `cred.be/sites/default/files/CredCrunch32.pdf`, Last visited: 2018-05-23.

[Cend] Centre for Research on the Epidemology of Disasters CRED. Disaster Data: A Balanced Perspective. `www.cred.be/sites/default/files/Disasters-in-numbers-2013.pdf`, Last visited: 2018-05-23.

[Cene] Centre for Research on the Epidemology of Disasters CRED. Disaster Data: A Balanced Perspective. `www.cred.be/sites/default/files/CredCrunch37.pdf`, Last visited: 2018-05-23.

[Cenf] Centre for Research on the Epidemology of Disasters CRED. Natural disasters in 2017: Lower mortality, higher cost. `www.cred.be/sites/default/files/CredCrunch50.pdf`, Last visited: 2018-05-23.

[CFH97] Lawrence D. Cutler, Bernd Fröhlich, and Pat Hanrahan. Two-handed Direct Manipulation on the Responsive Workbench. In *Proceedings of the 1997 Symposium on Interactive 3D Graphics*, I3D '97, pages 107–114, New York, NY, USA, 1997. ACM.

[DG96] Mathieu Desbrun and Marie-Paule Gascuel. Smoothed particles: A new paradigm for animating highly deformable bodies. In *Computer Animation and Simulation'96*, pages 61–76. Springer, 1996.

[dra] ARB_draw_indirect. `https://www.khronos.org/registry/OpenGL/extensions/ARB/ARB_draw_indirect.txt`, Last visited: 2018-09-09.

[ESFT17] Carmine Elvezio, Mengu Sukan, Steven Feiner, and Babara Tversky. Travel in large-scale head-worn VR: Pre-oriented teleportation with WIMs and previews. In *2017 IEEE Virtual Reality (VR)*, pages 475–476, March 2017.

[fli16] HTC Vive Now Up For Pre-Order, 2016. `https://www.flickr.com/photos/bagogames/25845851080`, Last visited: 2019-01-08.

[flo] Flood Map: Water Level Elevation Map (Beta). `http://www.floodmap.net/`, Last visited: 2018-09-05.

[fsh] FShade: first-class shaders for F#. `https://www.fshade.org/`, Last visited: 2018-08-21.

[Goe16] Laurie Goering. Virtual reality game puts players in disaster driving seat, 2016. `http://news.trust.org/item/20161112223814-bpihw`, Last visited: 2018-08-07.

[HKK07] Takahiro Harada, Seiichi Koshizuka, and Yoichiro Kawaguchi. Smoothed particle hydrodynamics on GPUs. In *Computer Graphics International*, volume 40, pages 63–70. SBC Petropolis, 2007.

[HPW+16] Zsolt Horváth, Rui A. P. Perdigao, Jürgen Waser, Daniel Cornel, Artem Konev, and Günter Blöschl. Kepler shuffle for real-world flood simulations on GPUs. *The International Journal of High Performance Computing Applications*, 30(4):379–395, 2016.

[HSMT15] Georg Haaser, Harald Steinlechner, Stefan Maierhofer, and Robert F. Tobler. An Incremental Rendering VM. In *Proceedings of the 7th Conference on High-Performance Graphics*, HPG '15, pages 51–60, New York, NY, USA, 2015. ACM.

[HTCa] HTC Vive Specification. `https://www.vive.com/us/product/vive-virtual-reality-system/`, Last visited: 2018-03-09.

[HTCb] HTC. HTC Vive. `http://www.vive.com/`, Last visited: 2017-07-17.

[HWP+15] Zsolt Horváth, Jürgen Waser, Rui AP Perdigão, Artem Konev, and Günter Blöschl. A two-dimensional numerical scheme of dry/wet fronts for the Saint-Venant system of shallow water equations. *International Journal for Numerical Methods in Fluids*, 77(3):159–182, 2015.

[Inn17] Universität Innsbruck. Simulierte Überflutung, 2017. `https://www.uibk.ac.at/newsroom/simulierte-ueberflutung.html.de`, Last visited: 2018-10-16.

[ins] ARB_draw_instanced. `https://www.khronos.org/registry/OpenGL/extensions/ARB/ARB_draw_instanced.txt`, Last visited: 2018-08-14.

[JE92] Richard H. Jacoby and Stephen R. Ellis. Using Virtual Menus in a Virtual Environment. 02 1992.

[Kol95] Eugenia M. Kolasinski. Simulator Sickness in Virtual Environments. Technical report, Army research inst for the behavioral and social sciences Alexandria VA, 1995.

[KP07] Alexander Kurganov and Guergana Petrova. A Second-Order Well-Balanced Positivity Preserving Central-Upwind Scheme for the Saint-Venant System. *Communications in Mathematical Sciences*, 5(1):133–160, 2007.

[KSM$^+$11] Valeria V. Krzhizhanovskaya, G. S. Shirshov, Natalia B. Melnikova, Robert G. Belleman, F. I. Rusadi, B. J. Broekhuijsen, Ben P. Gouldby, Julien Lhomme, Bartosz Balis, Marian Bubak, Alexander L. Pyayt, Ilya I. Mokhov, Artem V. Ozhigin, Bernhard Lang, and Rober J. Meijer. Flood Early Warning System: Design, Implementation and Computational Modules. *Procedia Computer Science*, 4:106 – 115, 2011. Proceedings of the International Conference on Computational Science, ICCS 2011.

[LKT$^+$17] Johannes G. Leskens, Christian Kehl, Tim Tutenel, Timothy Kol, Gerwin de Haan, Guus Stelling, and Elmar Eisemann. An interactive simulation and visualization tool for flood analysis usable for practitioners. *Mitigation and Adaptation Strategies for Global Change*, 22(2):307–324, Feb 2017.

[LSGES08] Yotam Livny, Neta Sokolovsky, Tal Grinshpoun, and Jihad El-Sana. A GPU persistent grid mapping for terrain rendering. *The Visual Computer*, 24(2):139–153, Feb 2008.

[LSH99] Robert W. Lindeman, John L. Sibert, and James K. Hahn. Hand-Held Windows: Towards Effective 2D Interaction in Immersive Virtual Environments. In *Proceedings IEEE Virtual Reality (Cat. No. 99CB36316)*, pages 205–212, Mar 1999.

[MA96] Scott McGlashan and Tomas Axling. A Speech Interface to Virtual Environments. In *Proc., International Workshop on Speech and Computers*, 1996.

[mika] EXAMPLE: MIKE FLOOD FM. `http://dhiuk-demos.blogspot.com/2011/02/example-mike-flood-fm.html`, Last visited: 2018-12-16.

[mikb] MIKE FLOOD - Urban, coastal and riverine flooding. `https://www.mikepoweredbydhi.com/products/mike-flood`, Last visited: 2018-06-28.

[Min95] Mark R. Mine. Virtual Environment Interaction Techniques. Technical report, Chapel Hill, NC, USA, 1995.

[mul] ARB_multi_draw_indirect. `https://www.khronos.org/registry/OpenGL/extensions/ARB/ARB_multi_draw_indirect.txt`, Last visited: 2018-08-14.

[Nic] Nick Hygens. HTC Vive Controller. https://bigbadcat.artstation.com/projects/neqL4, Last visited: 2018-06-14.

[Nor] Colin Northway. Menus Suck. https://www.gdcvault.com/play/1023668/Menus/, Last visited: 2018-08-16.

[ope] OpenVR SDK. https://github.com/ValveSoftware/openvr, Last visited: 2018-08-18.

[PBWI96] Ivan Poupyrev, Mark Billinghurst, Suzanne Weghorst, and Tadao Ichikawa. The Go-go Interaction Technique: Non-linear Mapping for Direct Manipulation in VR. In *Proceedings of the 9th Annual ACM Symposium on User Interface Software and Technology*, UIST '96, pages 79–80, New York, NY, USA, 1996. ACM.

[PFW10] Tabitha C. Peck, Henry Fuchs, and Mary C. Whitton. Improved redirection with distractors: A large-scale-real-walking locomotion interface and its effect on navigation in virtual environments. In *Proceedings of the 2010 IEEE Virtual Reality Conference*, VR '10, pages 35–38, Washington, DC, USA, 2010. IEEE Computer Society.

[QA11] Honghai Qi and Mustafa S. Altinakar. A GIS-based decision support system for integrated flood management under uncertainty with two dimensional numerical simulations. *Environmental Modelling & Software*, 26(6):817 – 821, 2011.

[RBG+12] Eugenio Rustico, Giuseppe Bilotta, Giovani Gallo, Alexis Herault, C Del Negro, and Robert Anthony Dalrymple. A journey from single-GPU to optimized multi-GPU SPH with CUDA. In *7th SPHERIC Workshop*, 2012.

[riv] RiverFlow2D - A Breakthrough in 2D Numerical Modeling. http://www.hydronia.com/riverflow2d/, Last visited: 2018-06-28.

[RKW01] Sharif Razzaque, Zachariah Kohn, and Mary C. Whitton. Redirected walking. In *Proceedings of EUROGRAPHICS*, volume 9, pages 105–106. Manchester, UK, 2001.

[Rya] Ryan Smith. The NVIDIA GeForce GTX 1080 & GTX 1070 Founders Editions Review: Kicking Off the FinFET Generation. https://www.anandtech.com/show/10325/the-nvidia-geforce-gtx-1080-and-1070-founders-edition-review/13, Last visited: 2018-08-24.

[SC92] Paul S. Strauss and Rikk Carey. An Object-oriented 3D Graphics Toolkit. *SIGGRAPH Comput. Graph.*, 26(2):341–349, July 1992.

[SCP95] Richard Stoakley, Matthew J. Conway, and Randy Pausch. Virtual Reality on a WIM: Interactive Worlds in Miniature. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '95, pages 265–272, New York, NY, USA, 1995. ACM Press/Addison-Wesley Publishing Co.

[SD18] Yusuf Sermet and Ibrahim Demir. Flood Action VR: A Virtual Reality Framework for Disaster Awareness and Emergency Response Training, 2018. `https://csce.ucmss.com/cr/books/2018/LFS/CSREA2018/MSV3081.pdf`, Last visited: 2018-08-07.

[SGH$^+$12] Peng Song, Wooi Boon Goh, William Hutama, Chi-Wing Fu, and Xiaopei Liu. A Handle Bar Metaphor for Virtual Object Manipulation with Mid-air Interaction. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '12, pages 1297–1306, New York, NY, USA, 2012. ACM.

[SH05] Christian Sigg and Markus Hadwiger. Fast Third-Order Texture Filtering. *GPU gems*, 2:313–329, 2005.

[SWR$^+$13] Benjamin Schindler, Jürgen Waser, Hrvoje Ribicic, Raphael Fuchs, and Ronald Peikert. Multiverse Data-Flow Control. *IEEE Transactions on Visualization and Computer Graphics*, 19(6):1005–1019, 2013.

[THF06] Emine Mine Thompson, Margaret Horne, and David Fleming. Virtual Reality Urban Modelling - An Overview. In *CONVR2006: 6th Conference of Construction Applications of Virtual Reality*, 2006.

[tufa] TUFLOW. `https://www.floodmodeller.com/products/software/tuflow/`, Last visited: 2018-12-16.

[tufb] TUFLOW - Product suite of advanced numerical engines and supporting tools for simulating free-surface water flow. `https://www.tuflow.com`, Last visited: 2018-06-28.

[UFK$^+$89] Craig Upson, Thomas A. Faulhaber, David Kamins, David Laidlaw, David Schlegel, Jefrey Vroom, Robert Gurwitz, and Andries Van Dam. The Application Visualization System: A Computational Environment for Scientific Visualization. *IEEE Computer Graphics and Applications*, 9(4):30–42, 1989.

[urb] UrbanFlood. `http://www.urbanflood.eu/Pages/default.html`, Last visited: 2018-12-16.

[VBDRC13] Daniel Valdez-Balderas, José M. Domínguez, Benedict D. Rogers, and Alejandro J. C. Crespo. Towards Accelerating Smoothed Particle Hydrodynamics Simulations for Free-surface Flows on multi-GPU Clusters. *J. Parallel Distrib. Comput.*, 73(11):1483–1493, November 2013.

[vis] Visdom. `http://visdom.at/about/`, Last visited: 2018-06-28.

[VKBS13] Khrystyna Vasylevska, Hannes Kaufmann, Mark Bolas, and Evan A. Suma. Flexible Spaces: Dynamic Layout Generation for Infinite Walking in Virtual Environments. In *2013 IEEE Symposium on 3D User Interfaces (3DUI)*, pages 39–42, March 2013.

[vrsa] Applications of Virtual Reality. `https://www.vrs.org.uk/virtual-reality-applications/`, Last visited: 2018-08-10.

[vrsb] Human factors and user studies. `https://www.vrs.org.uk/virtual-reality/human-factors-and-user-studies.html`, Last visited: 2018-08-10.

[WFR$^+$10] Jürgen Waser, Raphael Fuchs, Hrvoje Ribicic, Benjamin Schindler, Günther Blöschl, and Eduard Gröller. World lines. *IEEE Transactions on Visualization & Computer Graphics*, (6):1458–1467, 2010.

[WJ88] Colin Ware and Danny R. Jessome. Using the Bat: a Six-Dimensional Mouse for Object Placement. *IEEE Computer Graphics and Applications*, 8(6):65–70, Nov 1988.

[WKS$^+$14] Jürgen Waser, Artem Konev, Bernhard Sadransky, Zsolt Horváth, Hrvoje Ribičić, Robert Carnecky, Patrick Kluding, and Benjamin Schindler. Many Plans: Multidimensional Ensembles for Visual Decision Support in Flood Management. *Computer Graphics Forum*, 33(3):281–290, 2014.

[WO90] Colin Ware and Steven Osborne. Exploration and Virtual Camera Control in Virtual Three Dimensional Environments. In *Proceedings of the 1990 Symposium on Interactive 3D Graphics*, I3D '90, pages 175–183, New York, NY, USA, 1990. ACM.

[WS98] Bob G. Witmer and Michael J. Singer. Measuring Presence in Virtual Environments: A Presence Questionnaire. *Presence: Teleoperators and Virtual Environments*, 7(3):225–240, 1998.

[WSMT13] Michael Wörister, Harald Steinlechner, Stefan Maierhofer, and Robert F. Tobler. Lazy Incremental Computation for Efficient Scene Graph Rendering. In *Proceedings of the 5th High-Performance Graphics Conference*, HPG '13, pages 53–62, New York, NY, USA, 2013. ACM.

[WZR17] Daniel Winkler, Jonatan Zischg, and Wolfgang Rauch. Virtual reality in urban water management: Communicating urban flooding with particle-based CFD simulations. *Water Science and Technology*, 77(2):518–524, 2017.