Dynamic layering system for real-time interaction between entities and terrain

Game and Media Technology
Utrecht University

by

S.E. Pennings
sepennings@angrykitten.nl

Supervisors:

dr. ir. A.F. van der Stappen
prof. dr. M.J. van Kreveld

Table of Contents

1 ABSTRACT

This thesis describes a method which deals with terrain deformation and excavation in a virtual environment. Research into the subject has generally focussed on representation of the surface of the terrain, with methods based on triangle meshes with level of detail to voxel based surfaces, sometimes with deformation rules based on physics. However, most do not take sub-surface areas of terrain into account, nor do they consider multi-user interaction.

The method we present deals with terrain deformation as manipulation of volumetric data, with emphasis on retaining data describing the subsurface parts of the terrain. Manipulation of the data is caused by interactions between multiple entities and the terrain, where each entity is directed by a user within a multi-user network environment.

Experiments which were run to measure the performance of our methodology have shown that it is able to handle interactions at a lower than real-time rate, with its data structure best suited for scenarios which cause contiguous changes, such as a ball rolling through snow, as opposed to interactions causing highly random changes, which cause the data structure to grow rapidly. Experiments have also shown that client-side calculation of changes is possible, but it remains unclear if this is preferable to server-side calculation in terms of absolute calculation time.

2 INTRODUCTION

2.1 BACKGROUND

Throughout the years computing power has increased dramatically and the complexity of virtual worlds has grown with it. So-called 'open world gaming' is now the norm for triple A games rather than an exception. It is now commonplace for such games to consist of huge virtual worlds, complete with expansive landscapes and populated by characters consisting of tens of thousands of polygons. The possibilities for such worlds are as grand as the ideas of the people who create them. Their design choices decide what a player may and may not do within the world: which elements are dynamic and which are static. There are many examples of such dynamic elements: non-player characters, moveable objects, destructible geometry such as parts of buildings, the list goes on. Static elements may be as varied in appearance as dynamic elements, but are not interacted with beyond collision detection. Examples may include non-destructible buildings, trees, rocks, etc.

Terrain, by which we mean the world's underlying geography, generally also belongs to the latter group. Such a terrain may be modelled in a number of ways: it may be modelled by hand, which gives a creator the most control over the terrain's form, but it also demands very meticulous work. Computer generation is the other more scalable solution, which we may split into two groups: derivation from existing data sets and procedural generation. We will be discussing this in more detail later. Regardless of the way in which the terrain is created, like most elements the terrain is often represented as a polygon mesh since consumer graphics hardware heavily focusses on the fast rendering of polygons. In most gaming worlds this polygon mesh is sufficient. It is when we want the terrain to also be alterable, meaning allowing deformation, subtraction and addition of material, that the limits of using polygon meshes become apparent. The polygon meshes used merely describe the surface of the terrain. In other words: it is a shell with no thickness with nothing beneath beneath except a void, it is non-volumetric. Without volumetric data of what exactly is beneath this surface, it is nearly impossible to simulate a truly realistic deformation behaviour.

One may speculate on the reasons as to why volumetric terrains are used so little in games, but it is likely that data storage and computational considerations play a role, as does the balancing of resources. Furthermore, volumetric data often cannot be visualized in the same way as polygon meshes can and thus needs either a different rendering method or conversion to a polygon mesh in order to be visualized. Of course, there are examples of games where volumetric data is actually the main focus of the game, such as the popular 'Minecraft'.

Finally, terrain deformation may simply not add enough game play value to justify the resources it will need. This does not mean that terrain deformation is completely disregarded: many virtual worlds allow temporary and superficial alterations such as footprints or scorch marks to occur to add to the realness of the environment. More permanent changes, however, such as lasting deformation and addition and subtraction of material however are very rarely implemented, especially in mainstream games.

However, there are quite a few applications of virtual worlds where the inclusion of deformable volumetric terrain would be desirable or even necessary. For many years industries such as the mining, marine equipment and dredging industries have been using training simulators in order to teach their staff how to use multi-million Euro equipment. Such equipment, which includes excavators, dredging ships, pipe lays and drilling vessels, often interacts extensively with soil. If this soil can be represented by volumetric terrain which includes soil properties, it will be possible to make interactions between equipment and soil more natural, thus making the training simulation a richer learning experience.

2.2 OBJECTIVES

The primary focus of this thesis is the description, implementation and testing of a data structure and methodology for handling deformable volumetric terrain within a virtual world as found in typical 'open world' games. We want this terrain to be available in a network environment and contemporary, commercial hardware such as one would find in the average Dutch household should be enough to run it.

2.2.1 Deformable volumetric terrain

First and foremost, we want the terrain to be volumetric, so that the properties of the soil it describes can also be stored. Real world soil is almost never uniform in type and each type may have its own implications in terms of how the terrain deforms, but also in terms of what kind of effect the interaction between dynamic element and soil has on the dynamic element.

In order to get a general sense of what one could expect in soil, and thus the terrain, we take a quick look at the field of pedology, a branch of soil science which concerns itself with soil origin (pedogenesis), soil morphology and classification. Pedology subdivides soil into a multitude of layers, or horizons, of specific types of material. It describes a number of commonly occurring horizons, which are as follows: O Horizon) a layer of organic material in a relatively non-decomposed form; A Horizon) also called topsoil, this layer can be divided into layers of leaf-litter, fermenting leaf-litter, humus of several varieties and an eluvial layer; B Horizon) also called subsoil, a mix of particles like sand, silt and clay, which lacks the organic matter of the A horizon; C Horizon) or the parent material/rock, which is original rock from which the soil was formed. This may consist of large chunks of the R Horizon or Bedrock, a largely continuous mass of hard rock.

Our volumetric terrain must be able to accommodate similar structures of different types of material, which we will refer to as 'layers'. Layers can be of any shape and size, meaning it may also self-overlap.

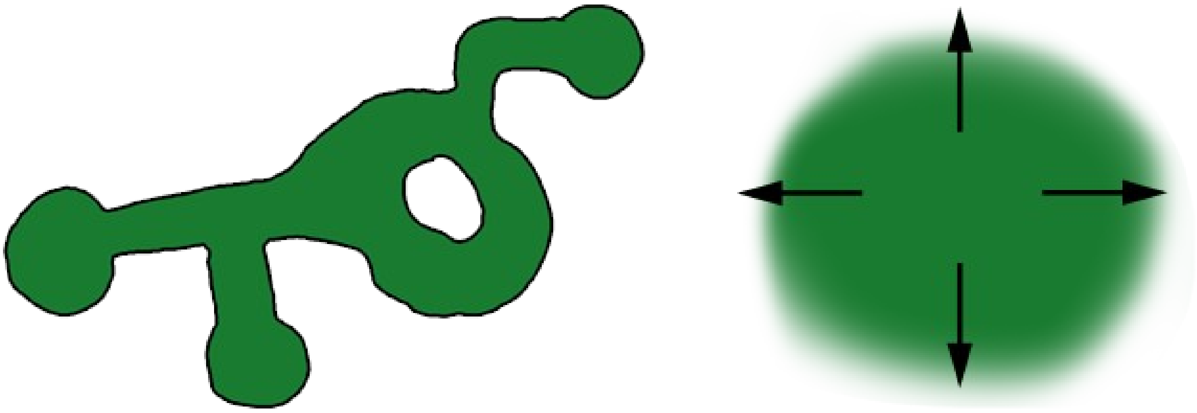


Figure 1: top-down schematic view of finite terrain (i) versus infinite terrain (r)

A terrain's shape and size will differ according to the overall goals of its designer: a linear first person shooter will demand a different kind of terrain than a flight simulator. We will look at these types of terrain use as finite terrain and infinite terrain (see Figure 1). The goal of finite terrain is generally to provide a small but detailed world and it may be non-contiguous and be oddly shaped. Examples of such terrain are any game that has levels which players need to traverse, like adventure games and first person shooters. Infinite terrain focuses more on expansiveness and will be largely contiguous. Flight simulators are an obvious example, but also games such as Minecraft, which uses procedural generation to create terrain on the fly so it is seemingly infinite. Of course, both finite and infinite terrains are technically finite because of technical limitations such as storage and calculation capacity of the hardware of the machines hosting them.

We want to be able to accommodate both types of terrain, although our focus will lie mostly on infinite terrain, as issues of data size and scalability are most prominently present there.

Alterations to the terrain will be initiated by the dynamic elements (excluding the terrain itself) which populate the virtual world. These elements must be able to change the terrain by adding or removing soil from it. We will assume that most interactions will be close to the upper surface of the terrain.

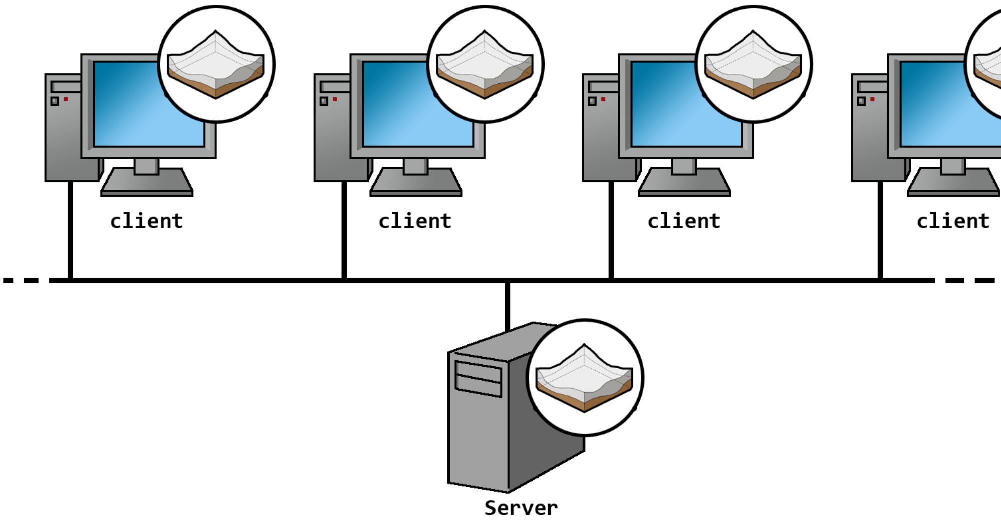


Figure 2: Computers in a client-server model. A terrain, originally defined by a server, is kept synchronous over a number of clients on a network. Each of these clients is able to change the terrain.

2.2.2 Network environment

We want the virtual world to run in a network environment and allow multiple users working on different physical computers to be able to look at and alter the same world and thus the same terrain. The type of networking environment we will consider is the well-known client-server model (see **Figure 2)**, whereby we make the terrain available from a central server to multiple client computers. This also means that the terrain needs to be kept synchronous between the server and each client: any changes to the terrain on one client should also appear in the server and all other clients as well.

Since the terrain may contain a large amount of data, we also need to limit the amount of data that needs to be communicated from server to clients and vice versa. In light of this, the data structure used should be able to communicate the terrain only partially and operations to the terrain's data should be kept to a minimum, meaning we must attempt to avoid repeatedly calculating the same operations.

2.3 RESEARCH QUESTIONS

From the requirements set in the previous section, we can describe a number of possible research questions, such as:

1. How can terrain property information (such as soil type) be stored?
2. How can data size be minimized in terms of storage and transfer over a network?
3. How do we keep data synchronized between server and client computers?
4. How do we handle interactions between dynamic elements and the terrain?

This leads us to the main research questions of this thesis:

1. Can volumetric terrain data be held synchronous in a client-server network environment with server-side interaction handling where multiple clients initiate such interactions, and is the amount of data needed to facilitate communication and calculation thereof acceptable?
2. In a client-server model where interaction handling is delegated to the clients and the server coordinates and communicates terrain updates, how often do two clients initiate interactions which conflict with one another?

2.4 STRUCTURE OF THIS THESIS

We will start with a literature review in Chapter 3, in which we will take a look at research concerning the use of terrain in games, polygon and tetrahedral meshes and volume reconstruction, visualization, storage and manipulation.

In Chapter 4 we will describe the methodology we have chosen to use and go into the details of its data structure and the various operations that need to be done, including generation of terrain, serialization and de-serialization of the structure for communication within the network environment and finding and calculating changes.

Chapter 5 will detail two experiments which we have run in order to test our methodology, including hardware set-up, software configurations, measures evaluated and of course results. These experiments are labelled I and II, I measuring general performance and II focussing on synchronization. Conclusions from the experiments in relation to our research questions will be discussed in Chapter 6.

Finally in Chapter 7, we will look at possible further research and cover any new question that we may have as a result of the experiments.

3 LITERATURE REVIEW

3.1 TERRAIN IN GAMES

Terrain, by which we mean the world's underlying geography, may be modelled in a variety of ways: modelling the mesh by hand gives a creator the most control over its form, but it also demands very meticulous work. Another way is to use height maps [1]. A height map is a regular two-dimensional grid holding values which describe height at a certain point. In other words, it is possible to use Jpeg, Gif and Bitmap image files and using the grey scale or colour values of each pixel to denote height. This method allows designers to work faster, but it is still not a scalable solution when dealing with very large terrains. A third way, procedural generation, alleviates this problem by an algorithm to determine the shape of the terrain. Such algorithms may be entirely random or have some forms of control applied to them [2] in order to give designers some measure of control over the final shape. A popular method is fractal based generation. An often stated downside to procedural generation, however, is that there is always a certain randomness to the end result which is beyond a designer's control. Like most elements, the terrain is often represented as a polygon mesh since consumer graphics hardware heavily focusses on the fast rendering of polygons. In some more recent games, such as 'Minecraft', the procedural generation approach is used to create a simple volumetric terrain consists of a regular grid of 'blocks' (containing soil characteristics) that are grouped into 'chunks'. Blocks have a relatively large size (around half the size of a player character) and its visible faces are rendered as polygons[3].

3.2 MESHES

Polygon meshes, or simply meshes are described by a number of connected elements. In two dimensions they can be described with either triangles or quadrilaterals, a form which we see in most terrain meshes, and in three dimensions by either tetrahedral or hexahedral elements, each of which is described by a number of vertices, or nodes, which connect to one another via edges. Adjacent elements share vertices and edges.

Meshes can be either structured or unstructured: structured meshes are generally simpler and faster because connections between each node and its neighbours is known beforehand. In unstructured grids each node needs to also store a set of pointers to each neighbour, which takes up additional storage. This does not mean that unstructured meshes are without use: there are many problems which cannot be modelled using structured meshes because of their shape or because the original data on which the mesh is to be based is itself irregular. In such situations unstructured meshes can result in more efficient triangulation as a smaller number of elements can closely fit an odd shape, while structured meshes may have to use many elements to both fit the odd shape and satisfy its regularity constraints.0

There is a great body of work concerning the construction of meshes, many of it incorporating Delaunay triangulation [4], such Chew's Constrained Delaunay triangles [5] and Shewchuk's Delaunay refinement [6]. Several works focus on the construction of meshes from regular and irregular data like point clouds [7][8] acquired via laser-range-scanning techniques and the subsequent creation of provably well formed triangle meshes.

Other, more simulation related studies focus on level of detail, such as ROAM [9][10].

3.3 VOLUME DATA

In the introduction we mentioned that volumetric terrain is not often applied in games and thus volume data is not taken into consideration in most virtual worlds created for entertainment purposes. The application of volume data is more prevalent in serious virtual worlds: one area in which volumetric data plays a major role is in medical imaging. In order to analyse a patient's health condition in a non-invasive way, medical personnel rely on several imaging techniques, such as Ultrasound, Thermography, x-Ray, Magnetic Resonance Imaging (MRI), Positron Emission Tomography (PET) and Computed Tomography (CT). Most techniques result in several slices of voxel or point set data which together form the entire scanned volume. The need for volume data can also be found in the widely varying field of engineering, where such data is needed for calculations ranging from force distribution through objects to fluid dynamics and heat transfer.

3.3.1 Volume Reconstruction

Data acquired via these imaging methods, or modalities, often requires reconstruction of some sort in order to make visualization possible. Two- and three-dimensional reconstruction of such data has been a research subjects for almost 30 years.

Most reconstruction methods used in the medical field [11][12][13] follow a number of basic steps from original data to a model ready for visualization:

Step 1) Image segmentation, meaning isolating pixels in each image slice which correspond with certain tissue types. The manner in which this is done varies by imaging modality: CT and MRI scans, for instance, are highly contrasted, allowing for easy segmentation.

Step 2) Region labelling, meaning the classification of tissue pixels belonging to the same group into anatomical structures. This may be done either by hand or using knowledge based methods.

Step 3) Global connection: this step connects the labelled regions on adjacent slices with one another.

Step 4) Local connection: once regions have been roughly linked, a proper connection must be made between them, which takes boundaries into account.

At the end of the entire process, we are left with proper volume data, which can then be visualized.

Within 3D construction methods, there are two main approaches: surface methods and volume methods. The former focusses mainly on the reconstruction of the surface of the volume and the efficient visualization thereof, while volume methods, as their name implies, focus on the volume itself. This is as much a visualization question as is a data structure question.

Surface methods focus on the visualization of the surface of the volume, which means that such a surface must first be constructed as well, for instance in the form of curved surfaces or a polygon mesh. This may be done via a number of techniques [7][8][14][15], including the point-cloud reconstruction methods mentioned earlier . The advantage of surface reconstruction over volume methods is that it can create meshes which can be easily visualized by consumer graphics cards. An obvious downside is the extra reconstruction that is needed.

3.3.2 Volume Visualization

Volume methods focus on the direct visualization of volume, and often use regular 3D voxel grids. There are three popular methods to do this, namely: Volume Ray casting, Splatting and Shear-warp [16].

Volume Ray casting [17] as its name implies, projects a ray through each of the pixels of the viewing, or image plane and continues through the volume data itself. This volume has certain colour and opacity properties which the ray then accumulates until it either becomes completely opaque, or it exits the volume altogether. This final state, along with shading, determines the opacity and colour of the resulting pixel.

Splatting [18][19] is another volume rendering technique. It takes volume data stored in a rectilinear grid and converts its voxels into two-dimensional representations, or footprints, which are then 'splatted' onto the viewing plane, much like throwing a snowball at a wall. The resulting footprints then combine to form a complete picture. Several factors influence the resulting rendering, including the size of the footprints and the ways in which they combine with other footprints. The method trades quality for speed, although many improvements have been made since it was originally proposed.

Shear-warp [20] tackles the problem of mapping object space, for instance a volume data grid consisting of several volume slices, to image space. In order to do so it creates an intermediary coordinate system, which is called 'sheared object space'. Because of the arbitrary angle at which the volume grid is viewed, viewing rays will most likely cut through the volume at an angle which is not

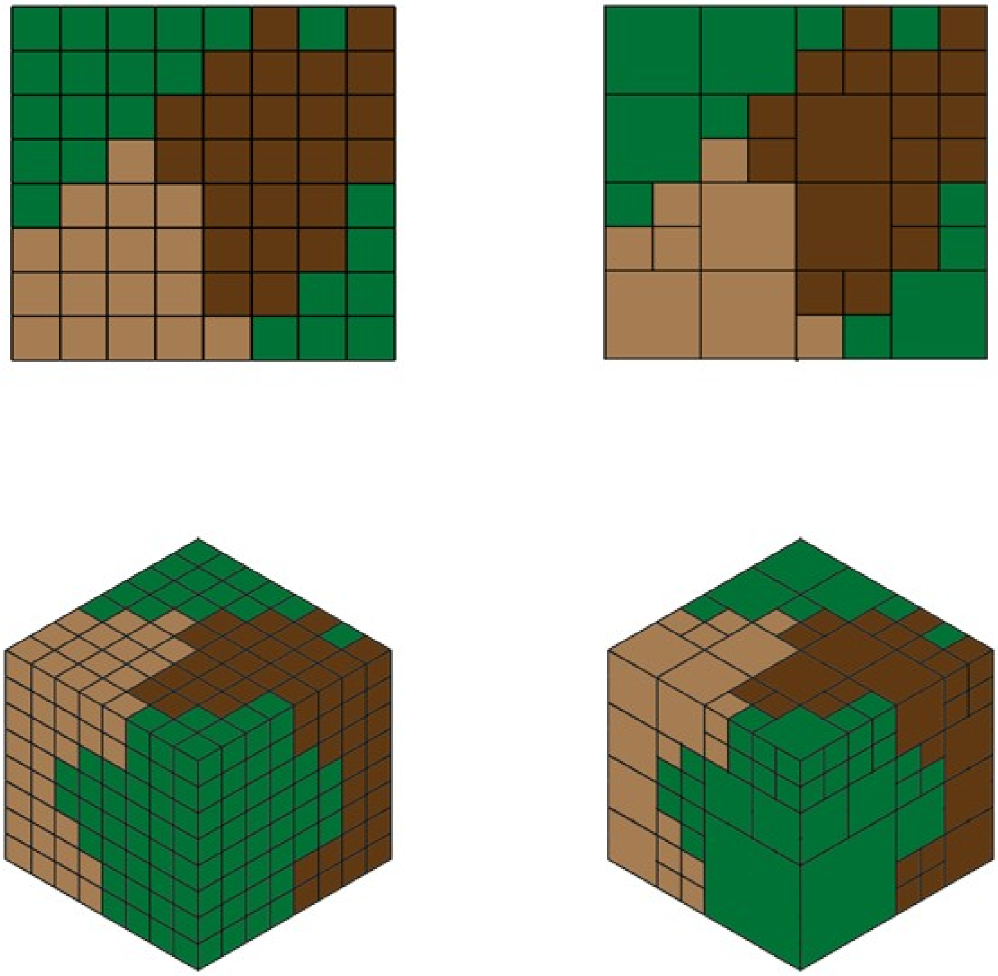


Illustration 3: A 2D regular grid (upper left) and a quadtree based on it (upper right), A 3D grid (lower left) and an octree based on it (lower right)

axis aligned with the grid itself. The shear-warp method transforms and scales the slices so that within the sheared object space, rays are axis aligned with the grid, which makes for more efficient compositing, which is then warped to match the image plane.

3.3.3 Volume storage structure

2D regular grids and quadtrees

Part of the volume reconstruction process for medical visualization is the isolating of pixels of certain tissue types and the classification thereof. All this takes place on a two-dimensional plane, which may be a two-dimensional regular grid. Each grid cell has the same dimensions and may be referred to as a pixel or unit square. Raster images, such as Jpeg, gif, png and Bitmap are regular grids and are useful in storing the two-dimensionally reconstructed data. However, since the regular grid may use many unit squares to describe a large contiguous tissue area, they are not particularly efficient.

One way which to store contiguous areas more efficiently is the quadtree [21]. Originally proposed by Finkel and Bentley [22] and others quadtrees are an often used tree structure which subdivides a two-dimensional space into four quadrants. Each of these quadrants can themselves be subdivided in a similar fashion until a granularity is reached in which subdivision is no longer possible.

In a quadtree, each quadrant can be seen as a node. Such nodes can either have no children, or exactly four. In other words, they are either a parent to four sub-quadrants, or they are leaf nodes. Each of these nodes can be reached by simply following a path down from the root node, which encompasses the entire area divided by the tree. Furthermore, the quadtree's logical structure means that it can be serialized using the same number of elements as there are nodes. We will discuss this further in Section 4.3.2 .

The basic quadtree idea has been expanded upon by various researchers, most notably Samet, who has written extensively [23][24][25][26][27] on this and directly related subjects. Work by him and others concerns quadtree construction from binary data, rasters, etc., neighbour node finding, conversion from and to chain/boundary codes [28][26] and quadtree compression [29].

3D regular grids and octrees

As we mentioned earlier when discussing 3D reconstruction, two-dimensional slices are combined to form a 3D grid which holds the complete volume data. One structure in which such data can be put is a three-dimensional regular grid, also known as a voxel grid. It suffers from the same efficiency drawbacks of the two-dimensional grid, only it adds another dimension to the problem.

Luckily, the Quadtree structure is easily translatable into a the third dimension, where it becomes an Octree. The octree is almost identical in properties to the quadtree, but instead subdivides the space into eight octants.

As with Quadtrees, a lot of work has been written by Samet and others on the subject [25][30][31].

3.4 VOLUME MANIPULATION (DEFORMATION, CUTTING AND ADDING)

There are many situations in which alteration of volume data is needed. As we mentioned earlier, within the field of engineering volume data can be used to make calculations relating to natural phenomena like force distribution, fluid dynamics, acoustics and more. Most of these calculations affect the volume data in some way, for instance deforming solids or changing certain properties of the volume data. Within the medical field, the need for alterable volume data can be found in surgical simulations created in order to train medical personnel. Deformation, elasticity, incompressibility and cutting of soft tissue is one of the main subjects, which is quite often combined with techniques concerning haptic feedback. Many studies, such as the works of Picinbono, Delingette and Ayache [32], Bro-Nielsen and Cotin [33] and Nienhuys [34] have delved into this subject and use a number of different methods.

A great number of studies have focussed on this problem and many have built on the idea of meshes, both regular and irregular. One method is that of Mass-spring models [35], which creates a mesh to fit the volume which it is to simulate and places masses at the nodes and damped springs on each edge. It is relatively computationally friendly and may allow real-time behaviour, but it is limited in its realism, as behaviour is heavily dependent by the topology of the mesh.

Another approach is Chainmail [36], in which volume elements are linked to six surrounding neighbours and behave in a way similar to that of the links in a chain, hence its name. Distance constraints are put on the links so that displacements are transferred to neighbouring links, which means that small displacements cause only local changes, but when stretched or compressed to the limit, it will cause the entire system to displace. Like mass-spring models, it is computationally friendly, but a bit simplistic.

There are also a number of discretization methods which deal with partial differential equations (PDEs) which are a mathematical way of describing natural phenomenon such as the ones we described at the beginning of this section. These are the finite difference method (FDM), finite element method (FEM), finite volume method (FVM) and the Boundary Element Method (BEM). What these methods have in common is that they attempt to break down a large complex problem into smaller, more manageable bits. The calculation of deformation, elasticity, heat conduction, fluid dynamics, etc. for a large volume or mass as a whole can be very difficult, if not impossible because of how all forces interact with the volume on a granular level. In other words: FEM, VFM, FDM and BEM break up problems concerning the behaviour of continuous mass (continuum problems) into smaller, discrete elements.

The finite difference method (FDM) [37][38] divides a volume into a regular grid called a space-time grid and creates a discrete finite-difference model, which is a way of approximating continuous derivatives, with approximations of the derivatives at grid points and boundary conditions at the end points. Because the resulting grid may not neatly fit the original volume, the method is less suitable to applications where correct geometry matters, for instance fluid flow, radiation and stress between moulds and castings. However, the method is often used in the fields of seismology, Earthquake ground motion modelling and exploration.

The finite element method (FEM) [34][32][39] is a popular within the field of computer-aided engineering. It brings together mathematical and engineering approaches by arranging discrete, non-overlapping elements (nodes) into a Finite Element Mesh. This 3D volumetric mesh, which can be a structured or unstructured and fits the original volume fairly closely, is key to the method but its creation can be a difficult process.

The finite volume method (FVM) [40][41] is similar to FEM in that it uses a mesh, which may be structured or unstructured, and is used for the discretization of conservation laws. It has been used extensively for the calculation of problems concerning fluid mechanics and heat and mass transfer. It divides the domain into discrete control volumes, which are a division of a volume into a grid, which can be regular or irregular.

Finally, there is also the Boundary Element Method (BEM) [42], which deals with boundary integral equations (BIE) [43], which are reformulations of boundary value problems for PDEs and is applicable to problems where Green's functions can be calculated. It does not divide the entire volume into a mesh of discrete elements like FDM, FEM and FVM do, but divides the boundary of such a volume into a number of discrete elements. In a 2D example, this means it applying line segments and in a 3D example surfaces. BEM deals well with complicated geometries.

3.5 NETWORKING AND SYNCHRONIZATION

Questions surrounding networking and synchronization have been around for a long time, with a lot of research emerging after the internet started growing to truly global proportions. At its core, all synchronization problems are problems concerning the chronologically correct sequencing of events between processes running at different locations. Much like the way we needed to synchronize our watches to a defined standard time, so too must these processes work according to a common time-line. Approaches such as described by Lamport [44] attempt to do just that.

In the games industry, the advent of online virtual worlds and a demand for low latency has driven research into the topics of networking and synchronization with a variety of approaches, based on a number of network models. The first is the common client-server model, which is often preferred because it centralizes event sequencing an thus makes synchronization simple. However, this means it is also a bottleneck and presents a single point where errors can disable the entire network. Peer-to-peer networking, another approach, decentralizes calculation by letting clients be responsible for calculation and update messages to other clients, but it also creates significant synchronization problems. Finally, there is also the mirrored servers approach [45], which uses multiple mirrored servers which are connected with separate groups of clients.

Research into volume data and networking focusses on a number of subjects: one such subject is design of systems which allow for the sharing of volume data for purposes of sharing and education and the transfer of such data. Another is distribution of calculation tasks, such as volume reconstruction and incremental [46] or parallel [47] processes for rendering of volume on a network.

4 METHODOLOGY

We want to have a virtual world which contains a deformable volumetric terrain as well as dynamic entities which may interact with it and we want this virtual world to work inside a network environment using a client-server set-up, meaning that alterations may be initiated by different clients and such changes must be synchronized between server and clients.

In this chapter, we will describe a methodology which aims to accommodate this. We will start by describing a data structure to hold the volumetric data, followed by how we deal with dynamic elements within the world. Finally, we will go through the operations that occur within the world before and during run-time.

4.1 DATA STRUCTURE

The terrain we wish to deal with may be of an arbitrary size and depth, and in the case of the infinite terrain example, may not have a set maximum size. It holds volumetric data describing soil of one or more different types, which are present in the form of layers. As we have discussed in Section 3.3.1 , this volumetric data can come in a variety of forms such as point clouds, stacked 2D images and voxel grids. Our methodology builds upon the idea of 3D regular grids, which means a conversion process is needed. We will go into more detail about this in Section 4.3.1.

The 3D regular grid, as described in Section 3.3.3 , consists of a collection of cubes. In graphical terms, we could refer to each of these cubes as a voxel. However, these terms are strongly connected to graphical representation, which is not the goal of this thesis. Instead we will refer to such cubes as *unit cubes* and we will refer to its 2D counterpart as a *unit square* (see Figure 4).

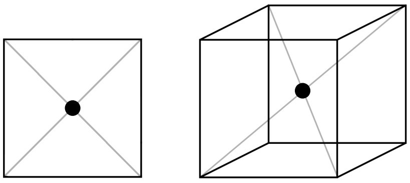


Figure 4: unit square (left) and unit cube (right). The dots indicate the centre of each.

This naming choice may seem arbitrary, but these terms more accurately describe functionality, as it is not inherently given how each unit relates to actual size of the terrain data that is represented in the virtual world. It could be that a unit square has dimensions in terms of cubic millimetres, metres or even kilometres. In other words: the size of a unit cube relative to the terrain it describes directly influences the resolution of the terrain. The more granular (i.e. smaller) the cube, the higher the resolution will be. Of course, this also means that more cubes are needed to describe the terrain and the size of the 3D regular grid will be larger.

It is obvious that as a terrain gets larger, the data needed to describe it also gets larger until at a certain point it reaches an unmanageable size in terms of both physical memory and computational limits. To alleviate this, we begin by dividing the terrain into smaller regions which we will call ***patches***. Doing this has a number of advantages: the spatial division of the terrain into subregions allows us to make interaction checking more efficient as we can check which patch an interaction relates to instead of checking the entire terrain. Secondly, it allows us to deal with each subset of the terrain separately, which opens up new possibilities for calculation and storage.

A patch, as illustrated in Figure 5, is a cuboid region that is axis aligned, meaning that it is aligned with the main axis of the virtual world in which the terrain exists. Its horizontal dimensions w are equal to each other and whose depth or height d is determined by the maximum depth or height of the subset of the terrain it encompasses. Relating back to the regular grid mentioned earlier, both values must 'fit' this grid. In other words, w and d can be seen to express a 'number of unit cubes', for instance 4x4x4 unit cubes. In relation to one another, patches are allowed to have different sizes in terms of both *d* and *w*, but the value of *d* may be any integer, while *w* must always be a power of two. The choice in actual size and placement of patches in relation to each other depends on the situation: in an infinite terrain the need for on the fly terrain generation may make similarly sized patches more preferable, while the odd forms of a finite terrain might be better off using varied sizes that more closely fit it. Examples of this can be seen in Figures 5 and 6.

The reason why we need *w* to be a power of two has to do with the way we subdivide each patch, whose data can be described as a 3D regular grid. Such a grid is inefficient in terms of storage and communication over a network, as the position and properties of each individual cube would have to be stored and communicated separately. In Section 3.3.3 we described ways in which 3D grid data can be stored more efficiently, namely via an octree. Therefore, using an octree to make a spatial subdivision of the grid would be a logical choice.

To recap: an Octree subdivides a cuboid region into eight equal cuboid regions recursively until a state is reached where division conditions are met and no further subdivision is possible or needed. One logical condition would be that all content of an octant needs to be of one type of soil, for instance. The octree has a very predictable structure, as a node has either no children or it has exactly eight, which is useful for serialization and de-serialization. In many ways it is a far more efficient way of storing volumetric data than the 3D regular grid. It is only in a worst case scenario that the grid is actually more efficient: the number of cubes needed to store a cubic area as a 3D regular grid can be defined as $N_{cubes}=w^3$, where N_{cubes} is the number of resulting unit cubes and w is the size of each side of the regular grid, which is a power of two, i.e. $w=2^n$. Thus, we can also write the formula as $N_{cubes}=(2^n)^3=8^n$.

Because an octree is a hierarchy of nodes, each child node has a parent that counts towards the total number of nodes needed to describe the unit cubes. The number of nodes in a worst-case octree, meaning the space is divided to its most granular level where no further subdivision is possible, would be:

$$N_{nodes}=\sum_{i=0}^{n} 8^i$$

where N_{nodes} is the number of nodes in the octree. Here, n also describes the depth of the tree. This situation will occur when the original terrain data is almost entirely non-contiguous. Choosing octrees as a method of spatial subdivision for our patches would be a logical choice. However, we choose to go with a bit more unorthodox approach. As we mentioned in Section 2.2.1 , we focus on interactions that mainly happen at the surface of the terrain and we want this terrain to exist within a network environment in which it is shared between a server and several clients. In such a network environment, only the server will initially need all of the terrain data. A client's user will be able to view the world via a camera, which may or may not be connected to an avatar representing the user within the world. In any case, the camera will have a certain starting position within the world and most likely, not all patches will be within viewing range of the client. This means that the client does not need to retrieve patches which lie beyond its viewing range. Also, the camera cannot look through the terrain's surface, so the client only needs to be aware of parts of the terrain which are exposed to or very near to the surface.

To facilitate this, we do the following: instead of subdividing via an octree, we will be subdividing the patch into regions that are perpendicular to the vertical axis. We will refer to these subsets as ***slices*** (See Figure 7). Slices have the same horizontal dimensions as the patch and its height is equal to a single unit cube, which is also how their vertical position within the patch is described. Slice 0 is positioned just above the virtual world's origin, meaning slices below it are expressed in negative numbers. By using slices, we can now choose to communicate only surface parts of the terrain to clients. Furthermore, we are able to extend a patch further up or down without having to make a new spatial subdivision for the patch as a whole, since we can simply add new slices to the top or bottom of the patch.

Of course, by slicing the patch up in this fashion we have basically reduced the 3D regular grid back to a stack of 2D regular grids (although each has a height of one unit cube in the third dimension). From each of these grid slices we create a quadtree. With this, the patch is now a stack of quadtree slices. This is

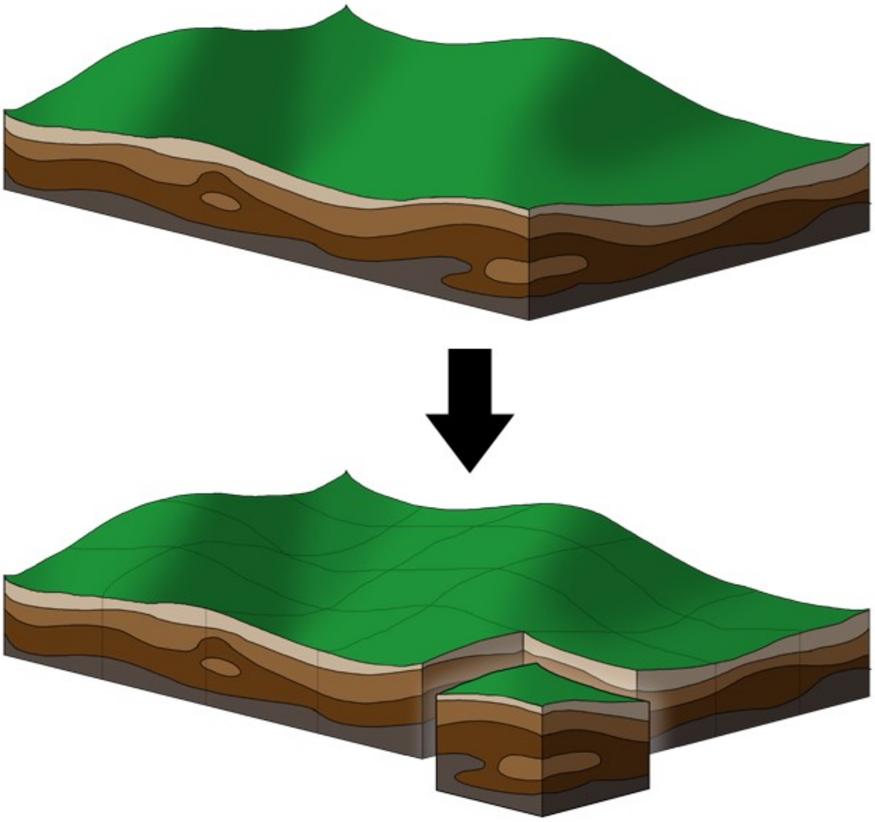


Figure 5: Dividing a terrain into patches of similar size. Below, a patch has been lifted out of the terrain for clarity.

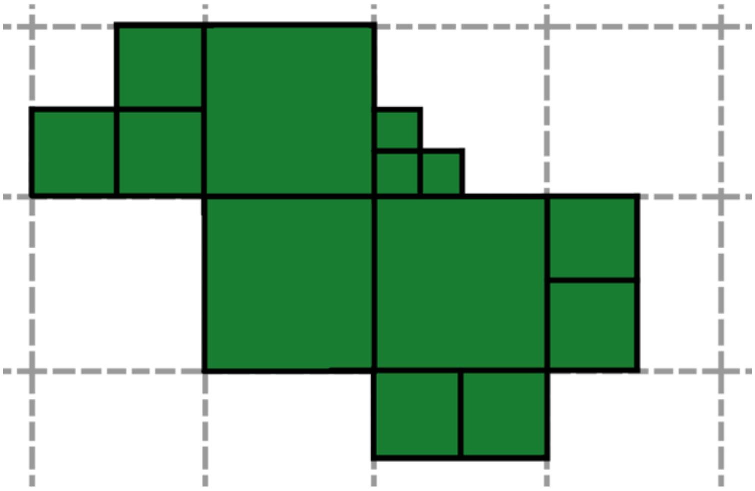


Figure 6: Example (top-down) of how differently sized patches may fit together to encompass a finite terrain.

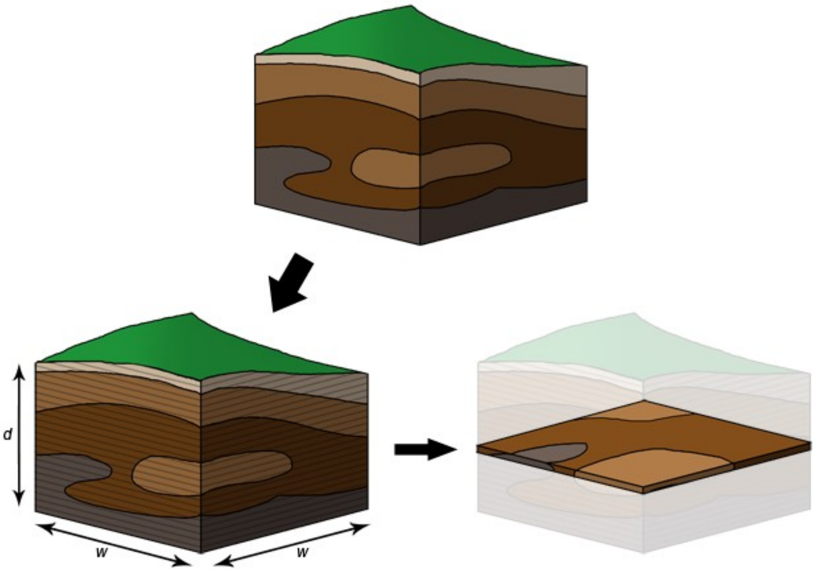


Figure 7: A patch vertically divided into slices of one unit cube height. Lower right shows a single isolated slice.

illustrated in Figure 8.

We use quadtrees because they have the same kind of properties that octrees have (octrees pretty much just adding a dimension to quadtrees): they have two types of node: internal nodes with exactly 4 children, and leaf nodes with no children. Its structure is predictable so it can be very easily serialized, stored and communicated over a network. This is something which we will discuss in detail in Section 4.3.2 . Quadtrees also have a good search time and work together well with the way in which we treat dynamic entities and the interactions they initiate, which we will explain further in the next section.

Using quadtrees does mean letting go of some of the data stored in unit cubes: Because groups of unit cubes of similar type are replaced by a single quadtree node, their positional data is lost. However, to

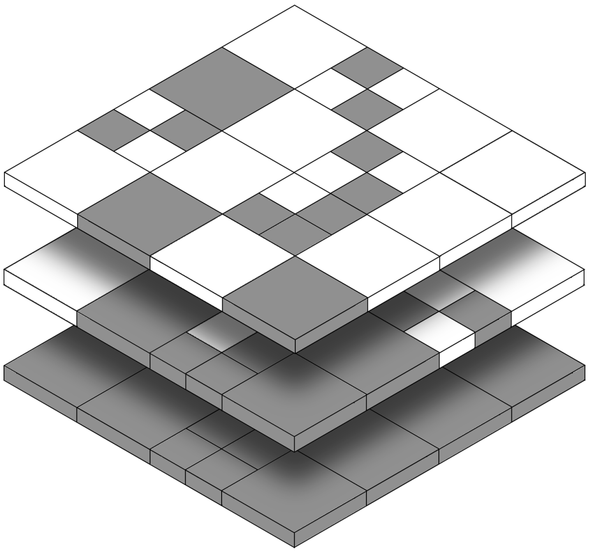


Figure 8: A stack of three slices, each described in the form of a quadtree. In this image, the quadtrees have been pulled apart and given thickness for better clarity. (Actual height is implicitly described by slices, which is a unit cube in height).

regain the original cubes, we can simply redivide the quadtree node.

As it is with octrees compared to 3D grids, the highest possible complexity of a quadtree is always less efficient than a 2D uniform grid describing the same area. We can describe the number of unit cubes in a 2D uniform grid as $N_{cubes}=w^2$, where again N_{cubes} is the number of cubes and $w=2^n$ so the formula can be written as $N_{cubes}=(2^n)^2=4^n$. The worst case for a quadtree can be described as:

$$N_{nodes}=\sum_{i=0}^{n} 4^i$$

where N_{nodes} is the total number of nodes required and n is also the depth of the tree. From this, we can conclude in the worst case situation even $n=1$ means that the quadtree is larger than the uniform grid. If we compare quadtrees and uniform grids over increasingly large values of n , we notice this efficiency difference rises sharply at first, but levels off at approximately 33%.

As we mentioned earlier with octrees, one situation in which such a worst-case scenario may occur is when material is distributed in an extremely granular and non-contiguous way, such as fine dust being sparsely sprinkled across the terrain. Figure 9 shows a number of patterns which result in a worst case scenario.

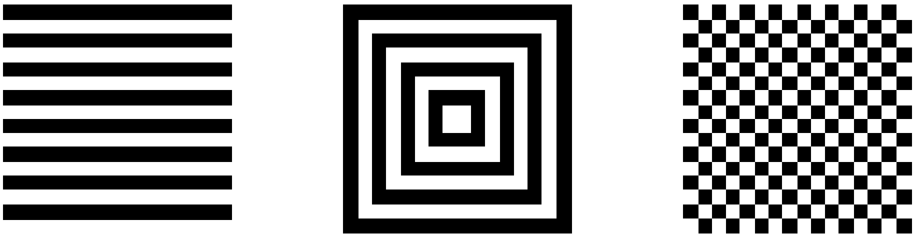


Figure 9: Three worst case examples for a quadtree. Each square in the example to the right is equal to a unit square

4.2 ACTING ENTITIES

Dynamic elements cause the terrain to deform by acting upon it. Of course, in our virtual world the terrain itself is a dynamic element, so we need to more specifically define which elements we mean. First, we exclude the terrain, since it will not interact with itself. Furthermore, we exclude any entity which is not controlled directly or indirectly by a client since such entities should be under the server's control, which means that any interaction is done on the server and only communicated towards the clients. We will refer to the elements which are directly or indirectly controlled by a client as ***acting entities** (AE).* Examples of acting entities would be an avatar controlled by a player, or units in a real time strategy game.

The shape and complexity of acting entities may vary widely, from a six-sided box to a character model consisting of tens of thousands of polygons. Especially in the case of the latter, it can be difficult to determine which parts of the entity make contact with the terrain. Luckily, we can look at solutions used in collision detection to solve this problem.

Because within a virtual world there may be many thousands of entities that might collide with one another, it is impossible to check for collisions between every entity pair. Therefore, the process is often split up into different levels of detail: first it needs to be checked whether entities are in close enough proximity to one another to collide. Tree structures may be used to map spatial divisions between the entities in order to make searching more efficient. But even if entities are close together, they might still not collide. In order to avoid immediately having to do detailed collision detection based on the polygons of the model, another rough check can often be made using an entity's bounding volume, which may be a sphere, cuboid, cylinder, etc. These bounding volumes may be rotated or axis-aligned.

We will be using an axis-aligned bounding box, which is the smallest cuboid possible that completely contains the entity and which edges are aligned to the axis of the virtual world. We do this because in our terrain's data structure, the stacked quadtrees are also axis aligned, which means that these bounding boxes can be directly used in tree searches.

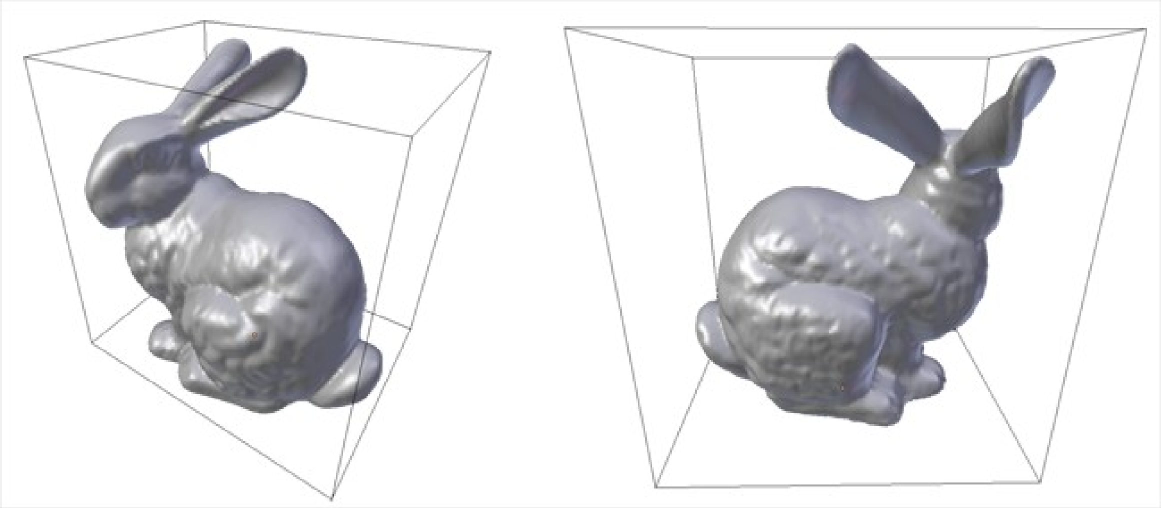


Figure 10: Bounding box surrounding a model of the Stanford bunny

The bounding box, together with the entity's geometry tell us which parts of the terrain are affected, while the physical properties and motion of the entity determine what this effect entails. We will discuss the details of this further in Sections 4.3.4 , dealing with finding affected nodes and calculating changes respectively.

4.3 OPERATIONS

Several operations are needed to facilitate storage, manipulation and communication of the data structure over a network. We will now describe these operations.

4.3.1 Creating the terrain data structure

As we mentioned in Section 2.2.1 , the basis for our terrain is a 3D regular grid. However, this does not mean the input volume data needs to be exactly that: a variety of collections of volume data can be used, provided that this data is described in a regular fashion such that it traversed as if it were a 3D regular grid. For example, a collection of stacked images such as described in Section 3.3.1 can also be used.

Irregular or incomplete data, such as for instance sparsely populated and unorganized point clouds should first be converted to a more regular format. As long as we have data points at regular intervals on which to base each unit cube, we can convert it into our data structure. Also, any absence of any data points is interpreted as empty space, i.e. a unit cube containing nothing.

The operation of creating the terrain from a 3D regular grid is relatively straight forward and moves through the following steps:

Step 1. Subdivision into patches

First, we need to decide whether the grid can be fitted into a single patch or that we need to divide its volume data among several patches, since particularly large sets of volume data may result in a patch of unmanageable size. This decision is made with the help of a pre-set value determining the preferred value for horizontal patch dimensions w_{max} : if the the grid's dimensions stay within this value, the entire grid is put into a single patch. If they exceed the value, we subdivide. w_{max} should be a power of two for reasons which we will explain in step 3.

As was shown in Figures 5 and 6, large 3D regular grids of volume data are subdivided horizontally into a regular grid of patches, each with a size equal to the pre-set maximum value or smaller. This subdivision is done by first making a rough subdivision into the aforementioned regular grid of patches. Sometimes, these patches will contain a lot of empty space due to the shape of the original terrain data. In such cases the patch is then subdivided into four quartiles, much like a quadtree is. This process is repeated until all patches contain either volume data or no data, or when further subdivision results in patches whose dimensions are smaller than a pre-set minimum value w_{min} . Such a minimum prevents the creation of patches which are too small (for instance a patch of 2x2 unit cubes).

Step 2. For each patch, divide vertically into slices.

Once patches have been created, they are each divided into slices, as is illustrated in Figure 7. This process is very straightforward: we simply subdivide the 3D regular grid vertically so that we are left with 2D regular grids which are one unit cube in height. Resulting slices are numbered according to there position relative to the origin of the virtual world, with which we simply mean that slices above zero will have positive numbers and those below will have negative numbers.

Step 3. For each slice, convert into a quadtree

In section 3.3.3 we mentioned that data in 2D regular grids can generally be stored more efficiently in the form of a quadtree. Because the horizontal dimensions of our patches and thus our slices are a power of two, we can easily convert each slice (which, although it has a height of one unit cube, acts like a 2D regular grid) into such a quadtree. We take the value of the top-left unit pixel (unit square) in the image and remember it. Then, we iterate through the values of each pixel in the image, comparing their values to our remembered value. If we encounter a value that doesn't match it means we need to split the node into 4 children.

For each of these children we repeat the scanning process, albeit now staring with the respective origins of each division, taking the top-left pixel within that division as an initial value and considering only the area of the image it encompasses. This process continues until no differences are found within a node, or we reach a logical node size of one pixel.

4.3.2 Slice serialization and de-serialization

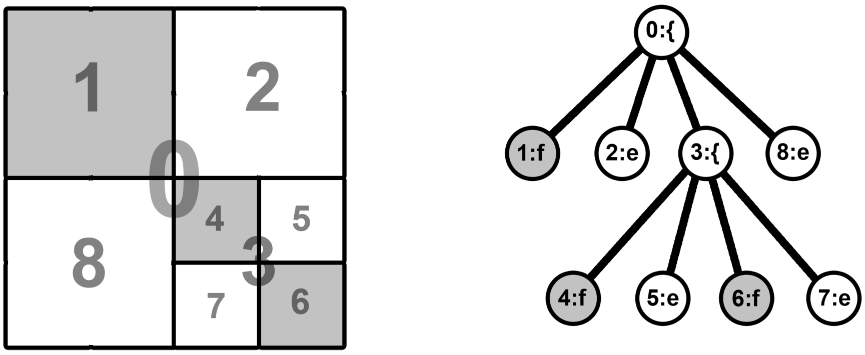


Figure 11: Serialization of a quadtree containing two layers: the left image represents a raster image divided into squares; the right image represents the resulting quadtree. This example results in the example string "{fe{fefee"

Because a quadtree node has no children or 4 children, we can serialize it efficiently by traversing the tree, starting with the root node and traversing its children in a set clockwise fashion. We move down the tree where nodes have children, repeating clockwise traversal until all nodes have been visited. This results in a serialized, ordered set of elements which can describe the entire quadtree with exactly as many elements as there are nodes.

De-serialization follows the same logic, but backwards: we start with the first element, which describes the root node. If the quadtree describes a slice that is wholly subterranean, this might be a leaf. Otherwise, this is a node with exactly 4 children, meaning the second through fifth elements describe children of said node. If we adhere to the same clockwise rule as done when we were serializing, we know at which position these nodes were. We continue through the set of elements, noting that each child's dimensions are exactly half that of its parent. Coupled with the knowledge of a parent's exact position we can build the entire tree in one traversal. This principle is shown in Figure 11.

It is worth noting that, because each branch within a quadtree is also a quadtree, we can choose to serialize and de-serialize only specific branches within the quadtree, which is useful for partial communication of terrain data.

4.3.3 Partial communication

We mentioned in Section 2.2.2 that we want to be able to only partially communicate our terrain and in Section 4.1 we discussed several reasons for why the ability to do this is needed, namely large terrain size and the local nature of clients and acting entities. Updates to the terrain caused by interactions will also have a local effect and such changes need to be communicated in a way which does not require us to send an entire patch or even a complete slice. In the next section, we will discuss how we can accomplish this.

As for initial connection between the server and a client, a client will communicate the current position and the bounding box of its acting entities. First, the server will determine which patches the client needs to acquire based on this. Secondly, it will determine which slices need to be communicated.

The number of slices, in other words the depth to which we take slices from the patch has consequences: If we make a shallow selection, this means only few slices need to be communicated to the client and thus the entire process generally takes less time and less client-side storage space is needed. However, this also means that in case of an interaction beyond the depth of the slices chosen, entire new slices need to be communicated towards the client, which may slow down the interaction itself. Conversely, choosing a deeper selection lessens the likelihood of having to communicate new slices, but it does mean more initial communication and it requires more client-side storage space.

4.3.4 Interaction handling

Interaction handling happens when an acting entity interacts with the terrain. In this section, we will describe the steps involved in this process.

Step 1 - Finding affected nodes

The first thing we do is determine which patches are within the reach of the acting entity, given its current position which should in most cases include no more than four patches (if the AE is at the spot where patch corners meet). This search can be done by checking whether or not the active entity's bounding box intersects with patches. Once we have determined the affected patches, we check which slices within these patches are affected. For this, we again use the active entities bounding box: only slices which lie between the bounding box's bottom and top planes are affected. Finally, we use the bounding box to do a quadtree search in order to find the affected quadtree nodes.

Step 2 - Acting entity geometry and slice intersection

Now, we must determine how the acting entity's geometry intersects with each slice and that slice's nodes. Because both acting entity bounding boxes and patches are axis aligned, this means that we can directly use the bounding box's horizontal dimensions to do a quadtree search in order to see which nodes of the tree are roughly affected by the interaction. This is illustrated in Figure 12.

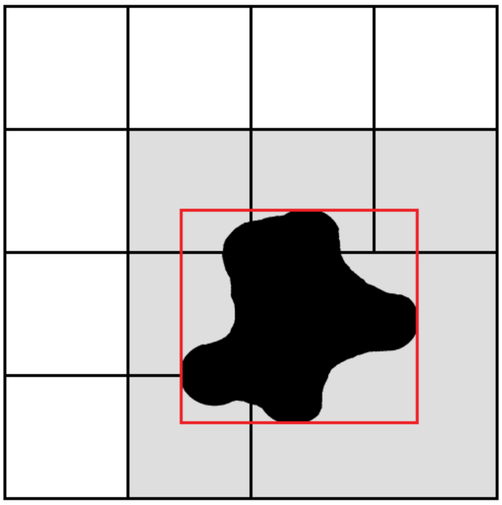


Figure 12: An Acting Entity's bounding box (red rectangle) holding the AE's stamp at a specific slice.

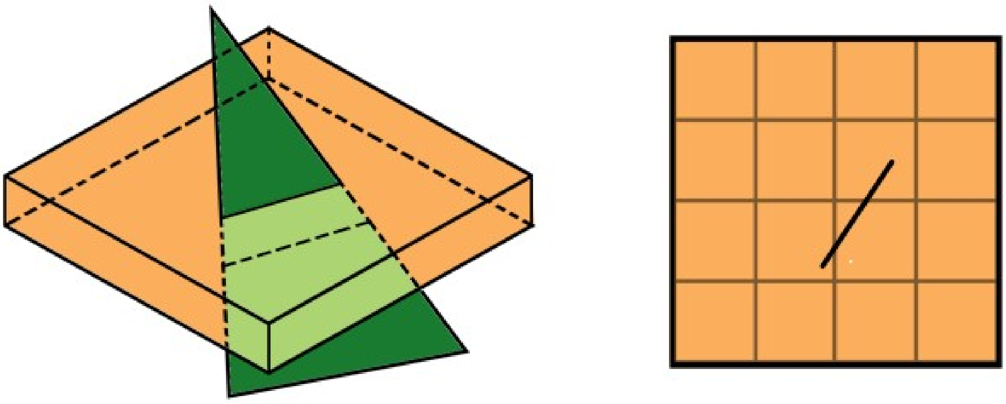


Figure 13: Left: a single polygon intersecting with a slice. Right: top down view of the same intersection.

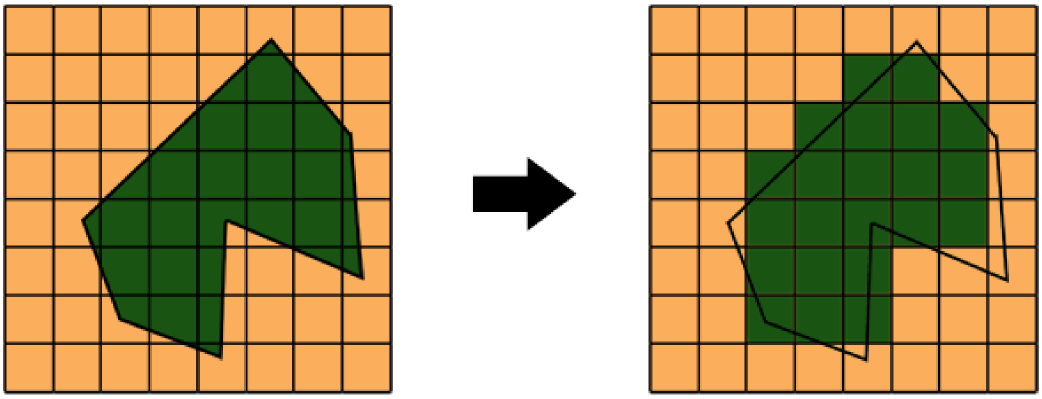


Figure 14: Approximating a grid from a polyhedron (or polygon in a 2D version).

The shape of an acting entity's geometry can vary widely and it most likely does not fill the entire bounding box, meaning that even if a part of the terrain falls within the bounding box this does not necessarily mean it is changed. In order to more accurately determine which nodes are affected and how, we need to know more about the acting entity's geometry. This geometry most likely consists of a collection of polygons, which will not neatly 'match' with the grid on which the quadtree is based (see Figure 13). Because of this, the effect of the interaction will never be completely precise. Therefore, we approximate the area which the geometry affects in the following way: first, we take the part of the acting entity's geometry which lies between the bottom and top of the affected slice, which is a polyhedron with a height equal to a unit cube. Then, we look at how this polyhedron fits within the slice.

Each slice's quadtree was created from a 2D regular grid of unit cubes. In order to fit the polyhedron into this grid, we must first convert it into such a grid of unit cubes. As illustrated in Figure 14, parts of the polyhedron may only partially occupy a unit cube, but a unit cube cannot contain multiple situations: it is either inside or outside of the acting entity's geometry. Therefore, we use a process

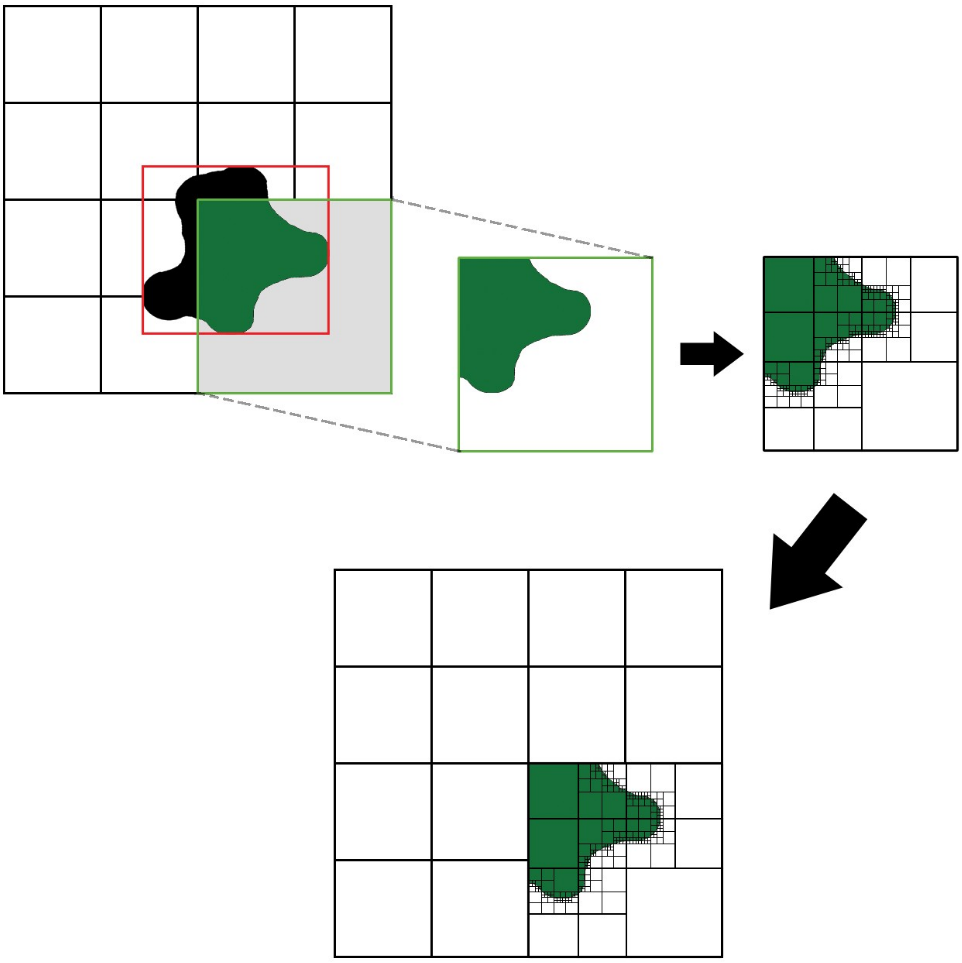


Figure 15: The stamp affecting the lower-right node, the part of the stamp that is within that node getting converted to a quadtree and the original node being swapped with the root node of the generated quadtree.

similar to the rasterization of vector graphics to make this decision.

A threshold value is used to determine how much of the unit cube needs to be occupied by the acting entity's geometry for it to be considered as being inside of it. This means that the value used greatly influences the outcome of the approximation. A low threshold will mean more unit cubes will be inside of the geometry, and thus the interactions effect will be larger in terms of size. This may lead to empty space between the acting entity geometry and the terrain. Conversely, a large threshold means more unit cubes will be considered outside, which may result in the terrain intersecting with the actual acting entity geometry.

One drawback of this method might be that these volume approximations may be very costly to calculate. An alternative is to slice the acting entity horizontally at a vertical position halfway between the bottom and top of a slice. This reduces the approximation to a two-dimensional question.

As a result of this step, we now have an approximation of the acting entity geometry which is a regular grid of unit cubes. We will refer to this as an acting entity's ***stamp*** on a slice. Of course, it is not necessary to actually generate a regular grid as an in-between structure because we can use the approximation rules directly in step 3, but for the sake of clarity in our explanation, we will deal with stamps as if we did.

Step 3 - Applying changes

Now, for each affected slice, we have a complete set of nodes that are affected by the interaction and we have the acting entity's stamp at that particular slice. From this we can calculate changes to the nodes.

Each affected node intersects with the acting entity bounding box. This means it may be fully within the bounding box or only partially within it. Likewise, the stamp may lie completely or partially within such a node. In each situation, we take the part of the stamp which lies within the affected node and we convert the regular grid within that part into a new quadtree. Now, in Section 4.3.2 we mentioned that a quadtree branch is in itself another quadtree. Therefore, we can use the resulting tree as a branch within our existing slice quadtree, meaning we simply swap the old node with the root node of this new branch to complete the change. This process is shown in Figure 15.

Step 4 - Communicating changes

Now, if we are using multiple clients in a network environment, changes caused by the interaction of one client must be communicated to all other clients in order for them to have the correct terrain. We can do this by sending out *change sets* to these clients. A change set is a collection of all the branches created by the interaction, coupled with a series of elements describing the path to the node which each of these branches is going to replace. These path descriptions include which patch and which slice the nodes are in. Once a client receives a changes set, it simply replaces the nodes at the given paths with the new branches, thereby updating the terrain data.

Step 5 - Pruning the quadtree

During the application of changes we do not prune the quadtree. Pruning is important for the efficiency of the quadtree as changes to the structure might introduce sibling nodes which all have the same content, meaning they can be combined into a single node.

The reason to wait until all changes are finished is that pruning between changes might cause parts of the quadtree to simplify into a parent node, while a second pending change might want to replace a child node of that node. As a result it would not be possible to make the change, which means the change is lost. Over time, this will cause otherwise synchronized quadtrees in different clients to grow increasingly dissimilar, which is not acceptable.

The pruning operation goes as such: for each changed node, we check if its siblings all have the same value. If so, they can be represented as one node: we delete the children and give their value to the parent, which is now a leaf. We then move up the tree to see if we can do the same thing again. This process is repeated until there is nothing left to prune (i.e. we try to move up at the root node).

5 EXPERIMENTS

In this chapter we will discuss a number of experiments which are aimed at testing our methodology, the implementation of which makes use of the well-known client-server model, namely with one server and a number of clients which connect to it. In this model, the calculation process can be distributed in different ways: the server can do all the calculation, calculation can be distributed over the various clients or a hybrid version of both. In our experiments, we will consider the first two situations: server-side calculation and client-side calculation.

Experiment I will focus on changes in terrain size and change set size in a server-side calculation situation, while experiment II will look at reservation conflicts between clients in a client-side calculation situation. Both experiments will run two interaction scenario's: random interaction and coherent path interaction and each of these scenario's will be run using different numbers of clients.

5.1 GENERAL EXPERIMENT SET-UP

Both experiments I and II use the same basic set-up in terms of hardware, terrain data and interaction scenario's.

5.1.1 Hardware

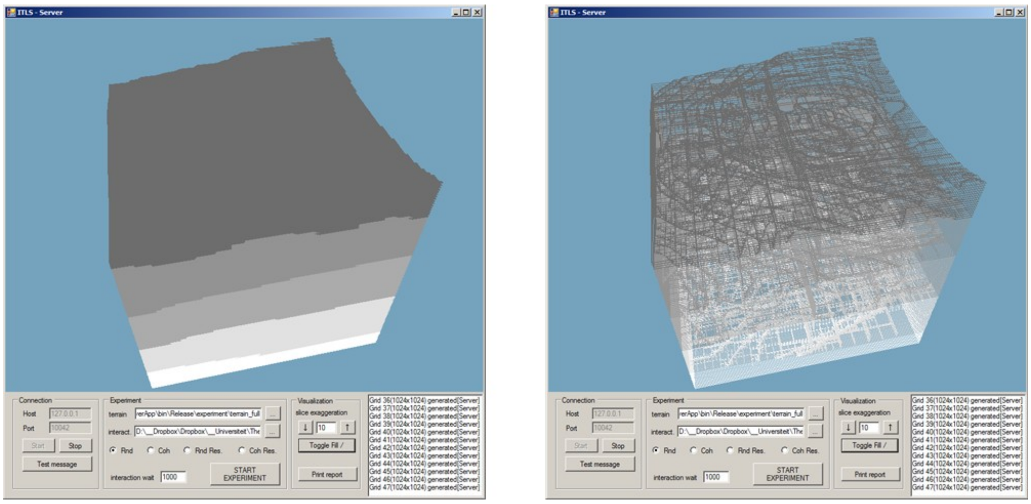
As we described in Section 2.2 , our methodology aims to allow calculation of interactions between acting entities and a volumetric terrain in real-time and to synchronize these changes via networks using commercial hardware, i.e. hardware found in the average Dutch household.

In order to run our experiments, we will be using the following pieces of hardware.

- **Server –** Desktop computer with the following technical specifications: AMD FX(tm)-8120 Eight-Core Processor 3.10 GHz, 16GB memory, Windows 7 64-bit.
- **Client(s)** - Samsung RC730, Intel Core I7-2670QM 2.2Ghz, 8GB memory, Windows 7 64-bit.

Clients and server will communicate with each other via a wired connection. All machines are set up to be in the same work group.

5.1.2 Terrain



Host 127.0.0.1
Port 10042
terrain verApp\bin\Release\experiment\terrain_full
interact. D:\__Dropbox\Dropbox\__Universiteit\The
● Rnd ○ Coh ○ Rnd Res. ○ Coh Res.
interaction wait 1000
slice exaggeration 10

```
Grid 36(1024x1024) generated[Server]
Grid 37(1024x1024) generated[Server]
Grid 38(1024x1024) generated[Server]
Grid 39(1024x1024) generated[Server]
Grid 40(1024x1024) generated[Server]
Grid 41(1024x1024) generated[Server]
Grid 42(1024x1024) generated[Server]
Grid 43(1024x1024) generated[Server]
Grid 44(1024x1024) generated[Server]
Grid 45(1024x1024) generated[Server]
Grid 46(1024x1024) generated[Server]
Grid 47(1024x1024) generated[Server]
```

Figure 16: Two screen shots of the Server program with one patch. Left, the patch is shown in solid colours. Right, the quadtree structures of each slice in the patch are shown, with empty nodes not drawn.

The same terrain will be used for each experiment.

The input data for this terrain is a collection of images of 1024x1024 pixels containing six possible grey-scale values, with each grey-scale value denoting a different type of soil for the terrain and a value of 0 describing the absence of soil, or empty space.

This creates a terrain consisting of a single patch which measures 1024x1024 unit cubes horizontally and contains 100 slices. Of these slices, 13 are (partially) exposed to the surface, meaning that the terrain surface has 13 different heights. 33 of them are described with a single quadtree node, meaning there is only one soil type at that particular slice.

There are 5 different types of soil in the terrain, which are stacked on top of each other in a layered fashion. For each of these layers, the height at different parts of its surface varies, and it contains some extrusions which overlap lower parts of the layer. An example of the resulting terrain can be seen in Figure 16.

Initially, the entire terrain is described using 183.476 quadtree nodes, with a potential to grow to a worst-case situation in which it would have $(\sum_{i=0}^{10} 4^i) * 100 = 139.810.100$ nodes. A 3D regular grid description of the same data would contain 104.857.600 nodes.

5.1.3 Interaction scenarios

In order to run our experiments, each client in the experiment must be able to trigger interactions. In order to do so, we need one acting entity per client. Human input (having a human user initiate change on said client) is one option, but it is very work-intensive, as we would need a person for each client. Furthermore, human input will be different in terms of both time and position of the change for each time that the experiment is run, meaning different behaviour on the part of human participants gives us different end results.

Therefore, we choose to automate this task by implementing a configurable acting entity that allows us to predefine where and when a change is made. This ensures that if we repeat an experiment, the input will always be the same.

This leads us to the following set-up: each client will control one acting entity that will interact with the terrain at predefined positions. These positions are given in a configuration file describing a list of points that is unique to each acting entity. All acting entities will use the same geometry as their shape, namely a sphere with a radius of 25 unit squares. The horizontal position of each acting entity at each interaction is determined by its point set. The vertical position for each interaction is at the surface of the terrain, meaning the place where non-empty volume data starts when moving through the slices in a top-down fashion (see Figure 17). This vertical position is determined prior to the actual interaction.

There are two types of effects that an interaction can have: the first removes soil from the terrain, while the second adds soil to the terrain. In the latter case, the soil added is the same as that of the top layer. These effects take place within the area occupied by the acting entity, i.e. within its spherical body. In each experiment, half of the clients involved have the first effect and half have the second.

Using such interactions, each experiment is split up into two interaction scenarios:

Random point interaction – this will use point sets generated as following: over the entire area of the patch, we will generate a random point set of 100 points. The scenario moves through these points, starting at t=0 (the first point) until it reaches t=99 (the last point). Interactions will be done with a time interval of 100 milliseconds, starting at point t=0. An example pattern for random point interaction is shown in Figure 18. Once t=99 is reached for all acting entities, the scenario is finished.

Coherent path interaction – here we will also use randomly generated point set. Each point set will contain 5 random points between which the acting entity will move in a straight line. Interactions will start at t=0 and the acting entity will move at a constant speed between the points until it reaches t=4. An example pattern for coherent path interaction is shown in Figure 19.

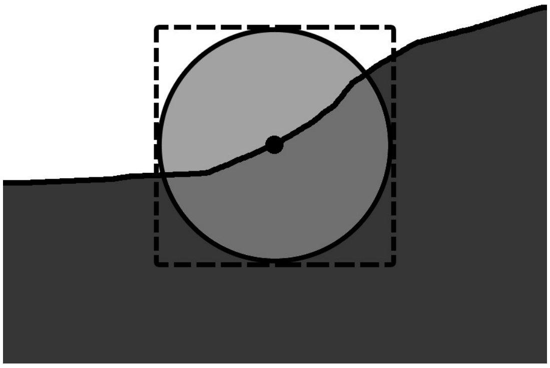


Figure 17: An acting entity and its spherical geometry just before an interaction is triggered (vertical cross-section).

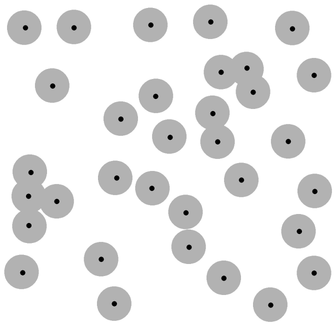


Figure 18: Random interactions.

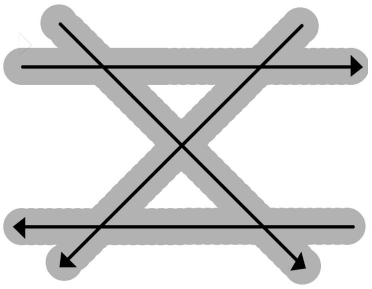


Figure 19: Coherent interactions.

At the beginning of each experiment, the server will send out a starting signal to all involved clients, telling them to start performing their interactions. This means that during the experiment all clients will be performing their interactions simultaneously.

5.1.4 Number of clients

Each interaction scenario will be run using different numbers of clients, namely 2, 4, 6, 8 and 10 clients. By running the experiments using different numbers of clients, we can see how an increase in clients affects the overall performance of our methodology.

5.2 EXPERIMENT I

As we mentioned at the beginning of this chapter, Experiment I focusses on changes in terrain size and change set size in a server-side calculation situation.

Server-side calculation, as the name implies, lets the server do most of the work. Clients will send it positions at which interactions happen, and the server will calculate changes and communicate these back to the clients.

The advantage of this configuration is that it makes the task of synchronization a simple one-way affair. Although clients initiate interactions, the server is the only holder of the source terrain (i.e. the terrain which can always be considered correct and current) and the only one making changes to it, thus controlling its correctness. As a result, clients will always receive the same terrain data and thus be synchronised. The disadvantage is that the server has to do all the work, meaning that its workload will increase according to the number of clients. And since each client initiates interactions and such interactions need to be communicated to all other clients this can, given enough clients, overload the server.

5.2.1 Interaction scenarios

The interaction scenarios for experiment I will be as they are described in Section 5.1.3 , except for the following: for each scenario, half of the clients participating will be adding material to the terrain, while the other half is removing material.

5.2.2 Measures evaluated

There are two measures which interest us in experiment I:

Terrain size – We will register the size of the terrain, i.e. number of nodes in the terrain before and after each interaction, after the terrain has been pruned. This way we can monitor how interactions affect overall terrain complexity. Will the terrain grow ever more complex to a point where it gets close to the worst case scenario described in Section 4.1 , or will complexity stabilize at a certain point?

Change set size – We will also register the number of nodes in each change set. Coupled with the number of times it gets sent over the network this should give us a good indication of how much strain each change set puts on the network, and how their size changes as the terrain itself changes over time.

The reason for measuring our method's performance in terms of amounts of data, rather than time, is that hardware configurations may vary widely with regard to processing power: is there a single, but powerful processor? Are there multiple processors, each of them mediocre in processing power? Other processes might be running, thereby influencing computing time. Network speed is similarly influenced. As far as hardware is concerned, a Gigalan network is obviously going to transfer data faster than a 65k modem. Network speed is also affected by other processes: if for instance another user on the network is streaming HD videos, this will slow down other transfers.

The amount and size of data generated and transferred is independent of processor and network speed. Processes might be executed at different speeds, but the amount of data remains the same, meaning that measuring these factors gives us a clearer picture of how our methodology behaves.

5.2.3 Experimental configuration

The terrain described in Section 5.1.2 will be used. As mentioned in Section 5.1.3 , the experiment will be run according to two different interaction scenarios: random interaction and coherent path interaction. Furthermore, we will run the experiments a number of times using a different number of clients as was described in Section 5.1.4 .

This results in the following experiments being run:

- random_interaction for 2, 4, 6, 8 and 10 clients.
- coherent_path for 2, 4, 6, 8 and 10 clients.

5.2.4 Results

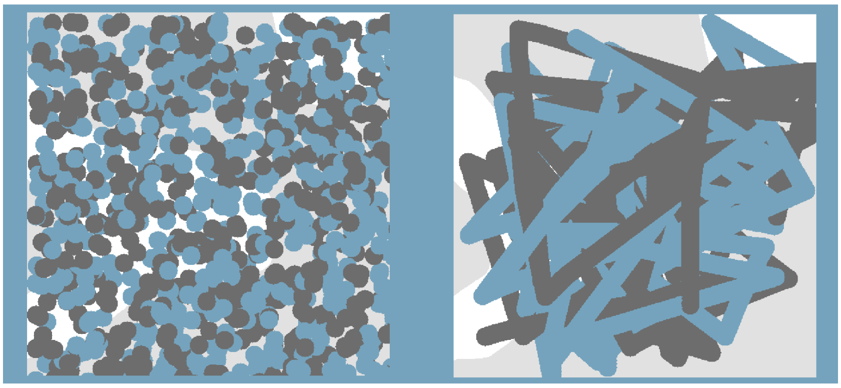


Figure 20: An example of the effects of each interaction scenario on a single slice. left, after random interactions. right, after coherent paths.

The results of experiment I using the random interaction scenario are shown in Figures 23 through 26 and Tables 1 and 2. Results for the coherent path scenario are shown in Figures 27 through 31 and Tables 3 and 4. For all graphs, the horizontal axis represents consecutive interactions, meaning interactions by all clients in the order in which they are received by the server (on a first-come-first-serve basis). For terrain size graphs, the vertical axis represents the number of nodes in the entire terrain and for the change set graphs, the vertical axis represents the number of nodes in a change set.

Looking at the results, we can see that terrain growth is affected not so much by the number of clients, but by the number of interactions caused by those clients. This becomes more clear when comparing the experiments for 2, 4, 6, 8 and 10 clients in each interaction scenario: if we overlay these results, we can see that the terrain growths they describe are very similar to one another. In fact, results from experiments using less clients almost seem to be subsets of experiments using more clients. In other words: if we were to repeat the experiments using only two clients, but having these clients perform a number of interactions equal to that caused by the version of our current experiments using 10 clients, results would likely be very similar.

If we look specifically at the results of the random interaction experiments, we can see that with 200 to 400 interactions, there seems to be a steady, almost linear growth. From 600 on to 1000 interactions however, we can see that the rate of growth slows down as more interactions are performed: this could indicate that terrain growth will eventually level out at a certain size, but the results of the random interaction experiments provide insufficient data to verify this. The resulting terrain is nearly 400% its original size after 200 interactions and well over 1200% after 1000 interactions. This large increase is most likely caused by the randomness of the interactions, which make the terrain more chaotic (see Figure 20) and so the quadtrees become more complex.

The size of the change sets varies throughout each experiment, with mean and median size per interaction increasing with the number of interactions performed. The maximum size is very similar for each experiment, which is to be expected since the acting entity's geometry is always the same.

The results for coherent path interactions paint a rather different picture. Although growth seems to follow a trend similar to that of random interactions, the rate of growth is much lower, with a mean and median growth per interaction that is around one tenth that of each experiment using the random interaction scenario. Maximum total growth per interaction is also much lower: 89% with 422 interactions (2 clients) and 343% with 2266 interactions (10 clients). It is worth noting that the number of interactions performed during the coherent path experiments is more than twice that of their random interaction counterparts, and so we have more data to work with. Growth rates, like we saw with random interactions, seem to slow down as more interactions are performed, almost flattening out at around 2200 interactions. However, it is still not clear if this means the terrain will stop growing.

The reason why coherent path interactions are so different from random interactions most likely lies in the nature of the interaction: random interactions are constantly influencing a different part of the terrain, while coherent interactions follow a certain path. This means that for each client, new interactions are always done close to where a previous one was performed, which results in more contiguous areas (see Figure 20) and contiguous areas can be described using less nodes.

This difference can also be seen in change set size. There are noticeable groups of peaks at several points in each experiment, which are most likely caused by two acting entities with differing actions (add/remove) encountering one another: since one would add soil and the other remove it, they are continuously causing large changes in the area where they intersect, thus causing large change sets. This type of conflict also causes the terrain to become 'shredded' as shown in Figure 21. Of course, it is worth noting that in most virtual worlds, such intersections would not occur, because acting entity geometry would be prevented from doing so by collision detection.

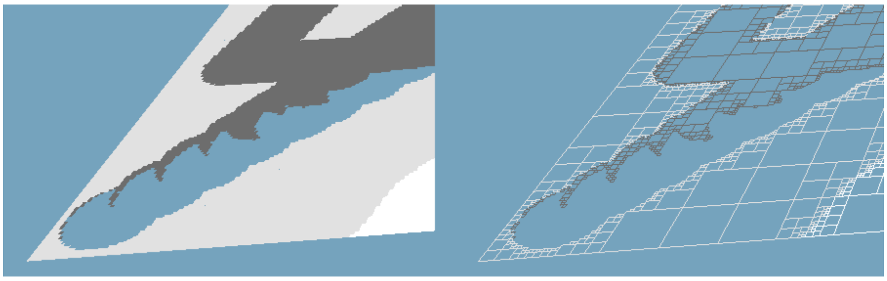


Figure 21: Terrain 'shredded' by two conflicting clients.

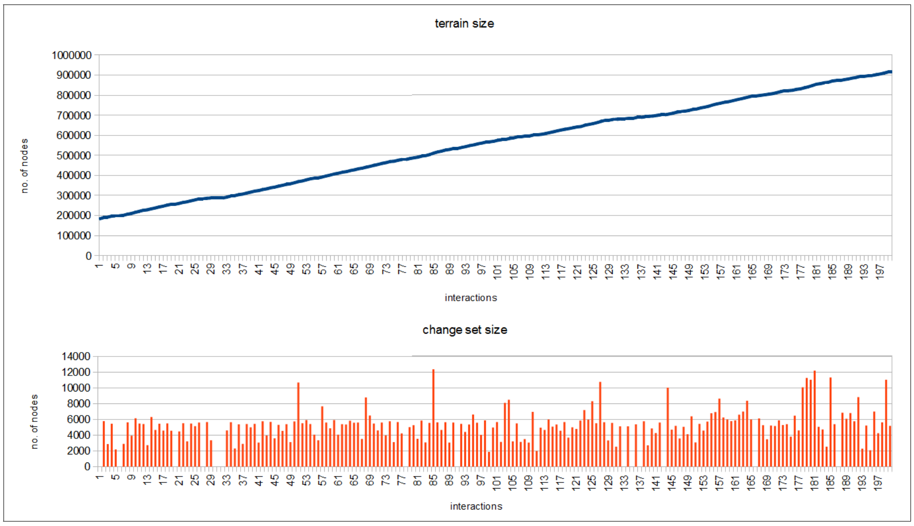


Figure 22: Experiment I - random_interaction, 2 clients.

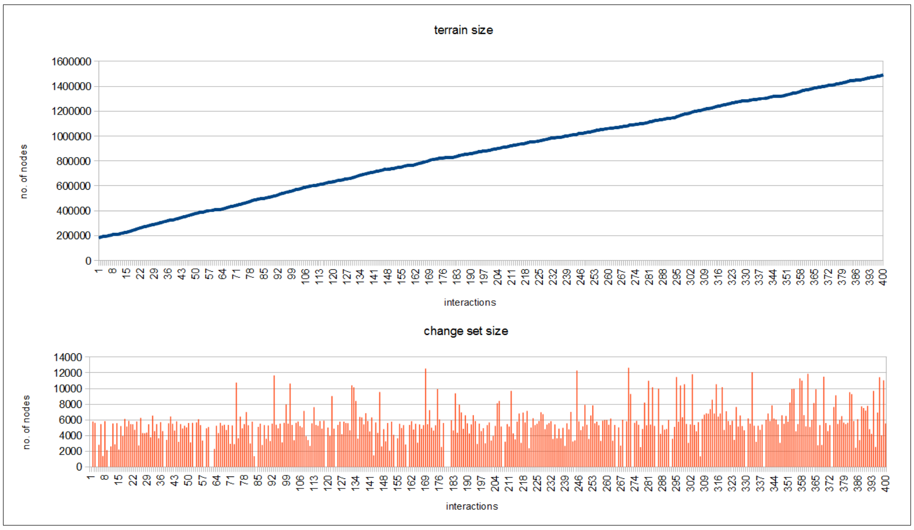


Figure 23: Experiment I - random_interaction, 4 clients.

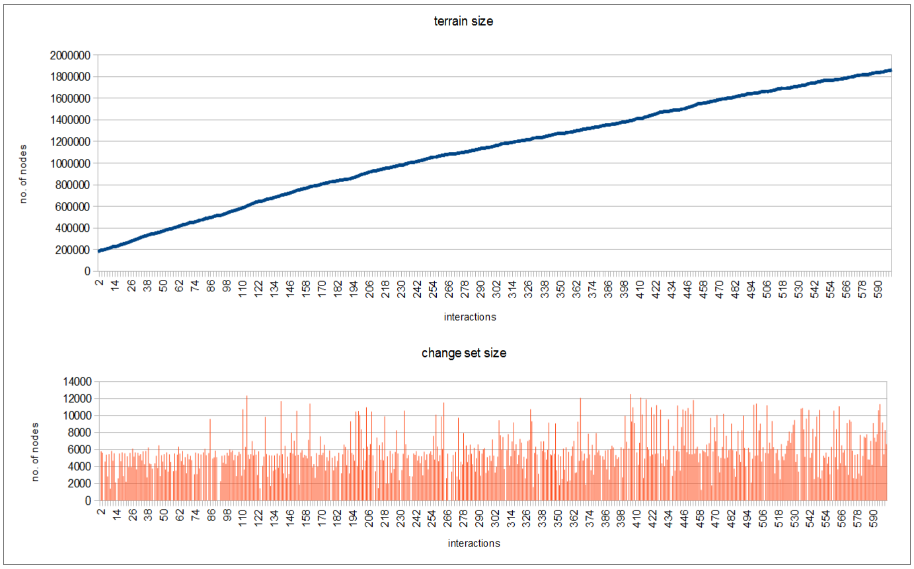


Figure 24: Experiment I - random_interaction, 6 clients.

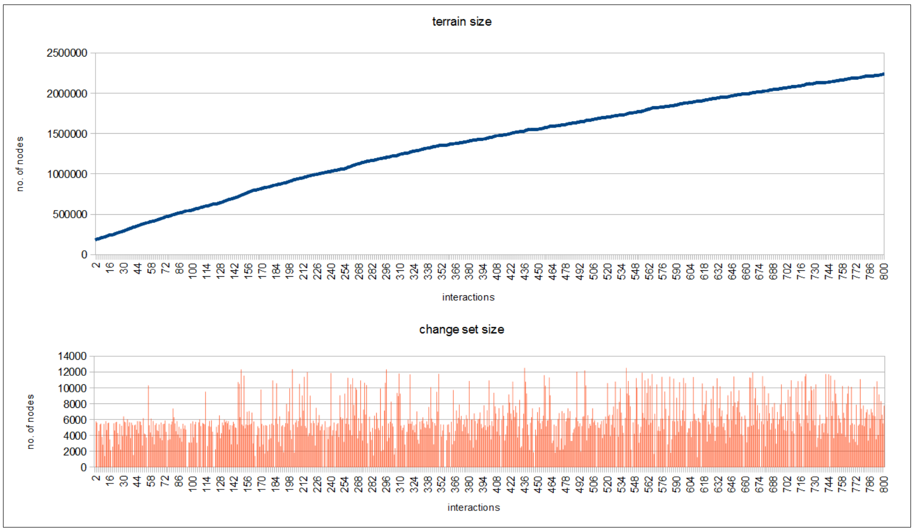


Figure 25: Experiment I - random_interaction, 8 clients.

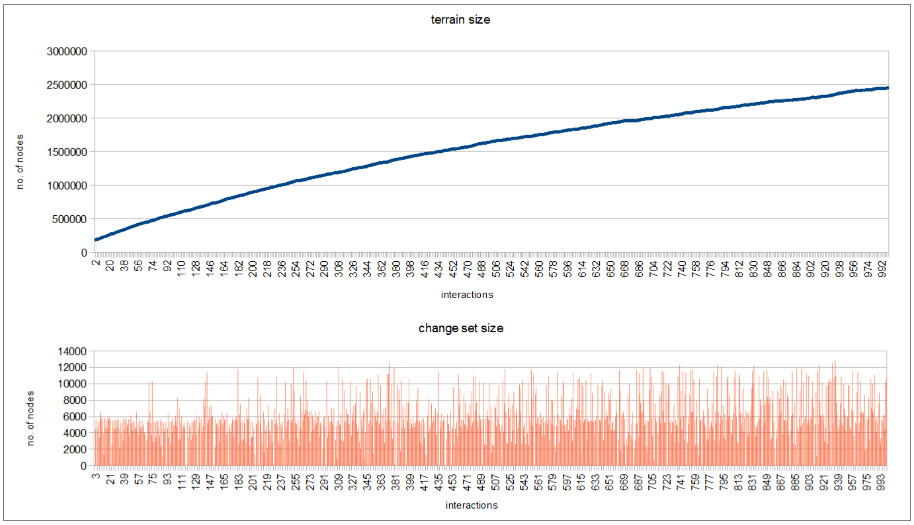


Figure 26: Experiment I - random_interaction, 10 clients.

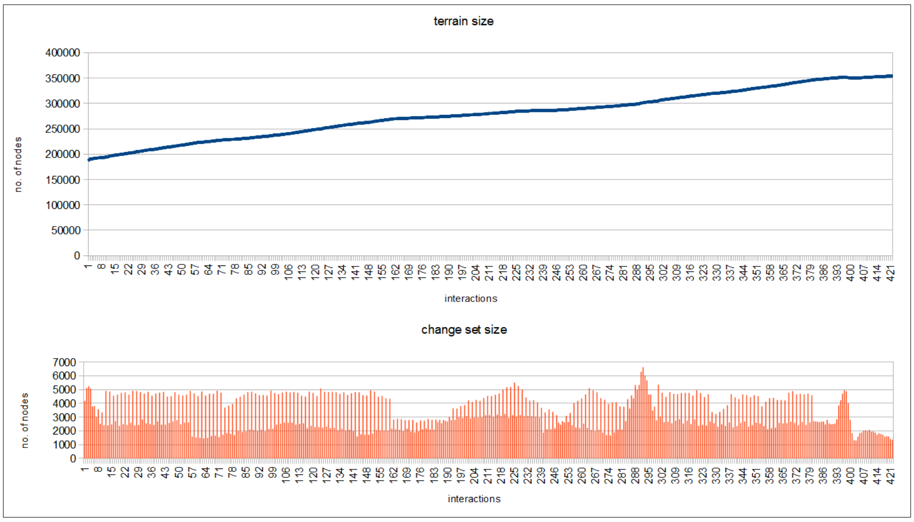


Figure 27: Experiment I - coherent_path, 2 clients.

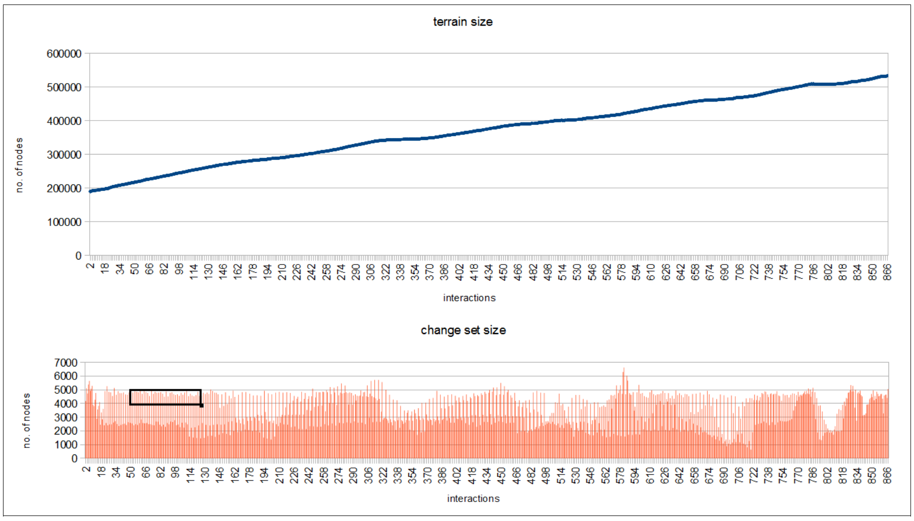


Figure 28: Experiment I - coherent_path, 4 clients.

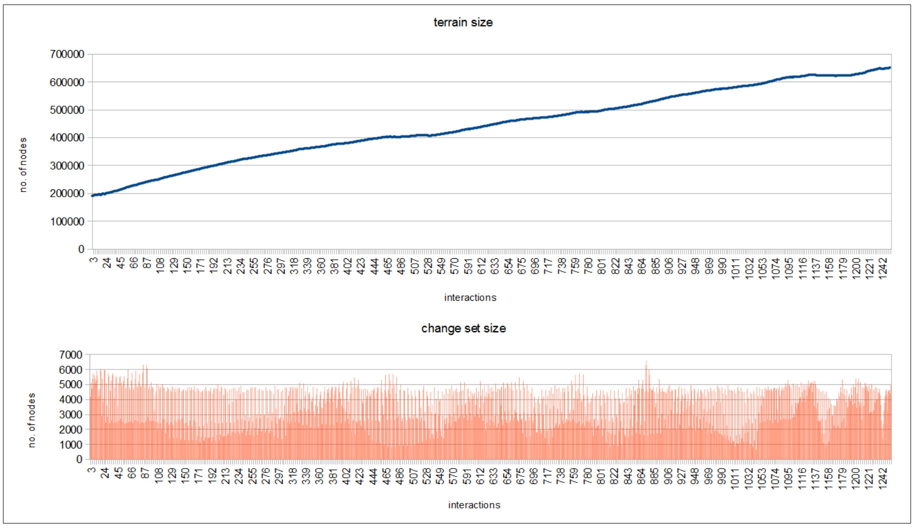


Figure 29: Experiment I - coherent_path, 6 clients.

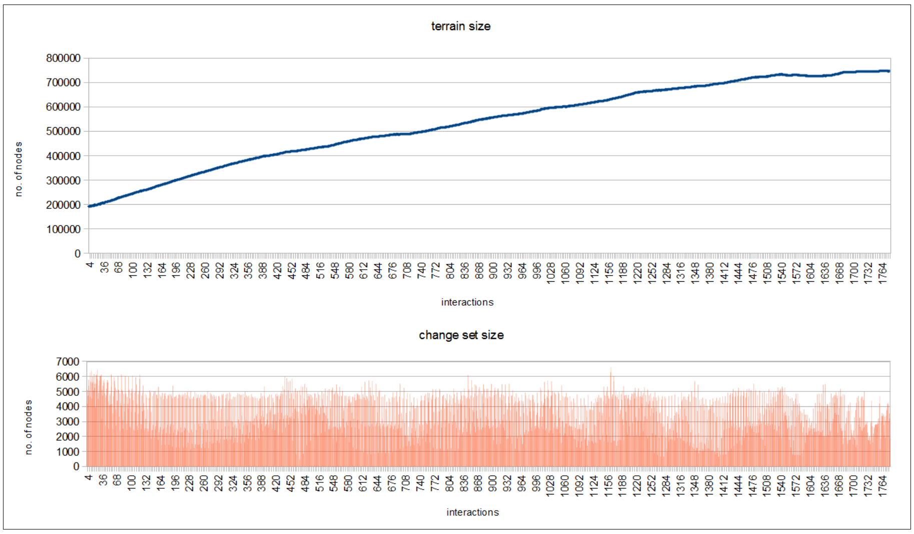


Figure 30: Experiment I - coherent_path, 8 clients.

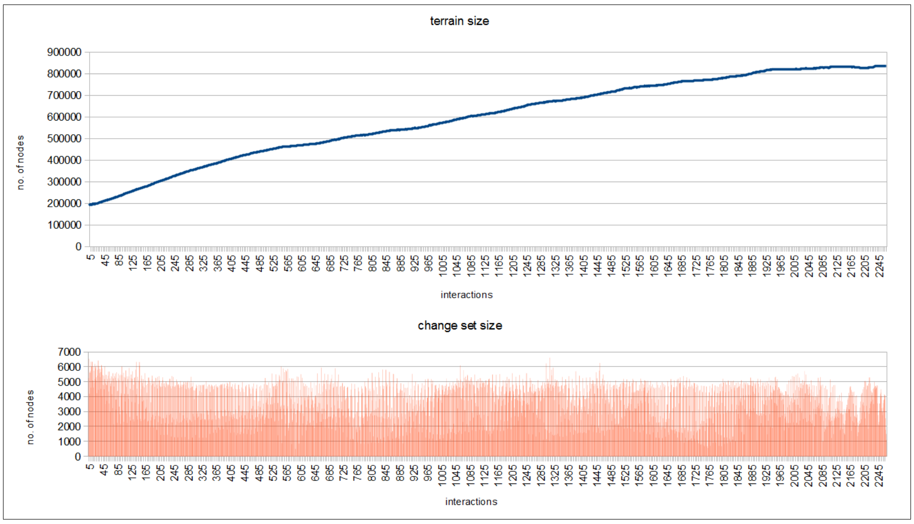


Figure 31: Experiment I - coherent_path, 10 clients.

		total growth		per interaction			
clients	**Combined Interactions**	**nodes**	**%**	**mean**	**median**	**max.**	**standard deviation**
2	200	733104	399.6	3665.5	4118	7580	1734.6
4	400	1307868	712.8	3277.9	3416	7576	1794.4
6	600	1679144	915.2	2803.2	2872	7864	1934.7
8	800	2051244	1118.0	2567.3	2492	7692	2013.2
10	1000	2273148	1238.9	2275.4	2176	7824	2051

Table 1: Experiment I – random_interaction, terrain size growth.

	change set size			
clients	**mean**	**median**	**max.**	**Standard deviation**
2	4993	5349.5	12347	2326.2
4	5347.4	5404.5	12650	2459.6
6	5457.5	5414.5	12530	2546.6
8	5787.3	5526.5	12525	2666.7
10	5832.8	5535	12829	2750.3

Table 2: Experiment I - random_interaction, change set size.

		total growth		per interaction			
clients	**Combined Interactions**	**nodes**	**%**	**mean**	**median**	**max.**	**standard deviation**
2	422	166528	88.9	395.6	352	3252	343.8
4	867	346652	185.1	400.3	382	3252	348.3
6	1255	463932	247.7	370.0	358	3252	423.7
8	1781	558368	298.1	313.7	312	3252	435.2
10	2266	645612	342.7	285.0	288	3572	459.3

Table 3: Experiment I - coherent_path, terrain size growth.

	change set size			
clients	**mean**	**median**	**max.**	**Standard deviation**
2	3331.5	2958	6625	1175.8
4	3511.3	3531	6625	1173.2
6	3604.3	3890	6625	1265.6
8	3546.5	3623	6625	1285.5
10	3680.1	3933.5	6611	1289.0

Table 4: Experiment I - coherent_path, change set size.

5.3 EXPERIMENT II

Experiment II focusses on interaction conflicts between clients in a client-side calculation situation.

Client-side calculation distributes calculation tasks across the clients, allowing them to handle the changes they make themselves and send the results to the server for distribution to other clients. The benefit of this method is that it relieves the server of its change calculation duties, thus allowing more clients to participate without overwhelming it. Furthermore, it should allow for greater changes to be made, as each client provides its own computing power. However, there is a drawback to this.

During the experiment, all clients will be performing their interactions simultaneously. This means that each individual client will attempt to change the terrain without regard for the actions of other clients. As a result, several clients could potentially make different changes to the same piece of terrain at the same time, resulting in multiple, equally valid changes which cannot be united into a coherent 'truth'. To prevent this, the server will need to enforce a system of reservation. What this means is that for every interaction, we need to check what part of the terrain the client is attempting to change, and if the client's version of the terrain is up-to-date for this part. Such a reservation attempt consists of an affected node search at both client and server side, with the client sending its results to the server and the server checking whether or not this matches its own results. If the results match and none of the affected nodes are already reserved by another client, the client may continue with its interactions. If the results do not match, the client's version of the terrain is not up to date, meaning it has not yet received available updates and it must wait for them. If there are already nodes reserved, the reservation also fails. In both cases, the client attempting the reservation must wait for updates before reattempting the reservation.

Unfortunately, in the time between a denied reservation and a renewed attempt by the client to make the same reservation, another client may have reserved and changed the terrain, meaning that the reservation request will fail again and the whole procedure is repeated once more. Especially when clients are in close proximity to one another this may mean that many reservations fail repeatedly. And, because each reservation attempt communicates a set of affected nodes to the server, the size of all reservation attempts combined might exceed the size of a single change set generated from a successful interaction.

The purpose of experiment II is to find out the amount of interaction conflicts in relation to the number of participating clients.

5.3.1 Measures evaluated

In this experiment, we are interested in the following measure:

Number of attempted reservations per interaction - We define this as a reservation attempt by an acting entity on one client, where some or all of the nodes it would affect are already in the process of being changed by an interaction caused by an acting entity from a different client. This measure will show us how many times a single interaction fails.

With this, we will be able to see how the number of conflicts per interaction changes over the course of the experiment, what the mean and median are for this and where extremes occur. Furthermore, we can also determine what kind of effect the number of participating clients has: is there a growth trend in the number of conflicts, for instance?

5.3.2 Experimental configuration

The terrain described in Section 5.1.2 will be used. Like experiment I, we will run both the random interaction and coherent patch scenarios using a different number of clients.

This results in the following experiments being run:

- random_interaction for 2, 4, 6, 8 and 10 clients.
- coherent_path for 2, 4, 6, 8 and 10 clients.

Each client will once again have one acting entity, which moves as described by the scenario type and its given point sets.

As was done in experiment I, at the beginning of experiment II, the server will send out a starting signal at which point the clients will start performing their interactions simultaneously. Reservations are handled by the server at a first-come-first-serve basis. Furthermore, in order to simulate the reservation and subsequent changing of a part of terrain, the server will lock such parts for a pre-set amount of time, defined as t_{lock}, which is the time in milliseconds that the lock will remain on the terrain. For our experiment, we will use 100 milliseconds.

We also set an amount of time a client has to wait before retrying an interaction after it fails, defined as t_{wait}, which is waiting time in milliseconds, where $t_{wait}=\frac{1}{2}t_{lock}$. We use half of t_{lock} because a client may encounter a conflict at any time between the other interaction's lock start and ending time, meaning that it may have to wait the full duration of the lock, or only a fraction of time thereof. The timing of each interaction is determined by a waiting time, which is 1/4th of t_{lock}.

5.3.3 Results

Looking at the results for random interactions, shown in Figure 32, we can see that most failed reservation attempts occur at the start of each experiment. The reason for this may be that in the beginning, the terrain has a lot of contiguous soil, which means that a lot of sub surface material can be described with a slice containing a single node. As the first few interactions are performed, many clients will try to reserve such single nodes at the same time, leading to many reservation failures. The amount of failures then rapidly decreases, because every applied interaction makes the slice more complex, thereby lowering the chance of acting entities working within the area of the same nodes.

Table 6 Shows us that the number of interactions that need multiple reservation attempts is fairly low compared to the total number of interactions, and the highest average attempts needed is only 1.4.

Coherent path interactions are again different. As shown in Figure 33, these types of interactions do not cause the same kind of peaks at the beginning of each experiment as random interactions did. Again, this is most likely because the client is following a path rather than acting randomly: it moves along a path, which means that each new interaction will be very close to the one before it. As such, once the first few interactions have been applied each client will work within an area which is already highly complex, thus limiting the chance that another client will intervene with its work. Such conflicts will only occur if two acting entities are very close to intersection.

Another noticeable difference, when we look at Table 5, is that the percentage of interactions needing multiple reservation attempts is much higher than with random interactions: it is almost 100%. This seems to be caused primarily by the clients themselves: because the area of each interaction largely intersects with the one before it, the server is likely to still be handling changes made in the previous interaction, thus denying a reservation of that part of the terrain. However, results also show a drop in the percentage of interactions that need multiple reservation attempts as more clients are used. It is not entirely clear why this happens, but it could be due to a slowdown of the program running the experiment as a result of becoming overwhelmed by the multitude of interactions.

One way to prevent this problem is to add information to reserved nodes on which client made the reservation. This way, when a client encounters nodes it reserved in a previous reservation, it can simply also reserve these nodes and continue its interaction, since it already knows what changes are made to the previous reservation. However, this also means that one client gets a reservation advantage over other clients, as it can continue to reserve along its own path, while denying others reservations along this path. When two client paths intersect, this may cause a prolonged failure of reservation for one of the clients.

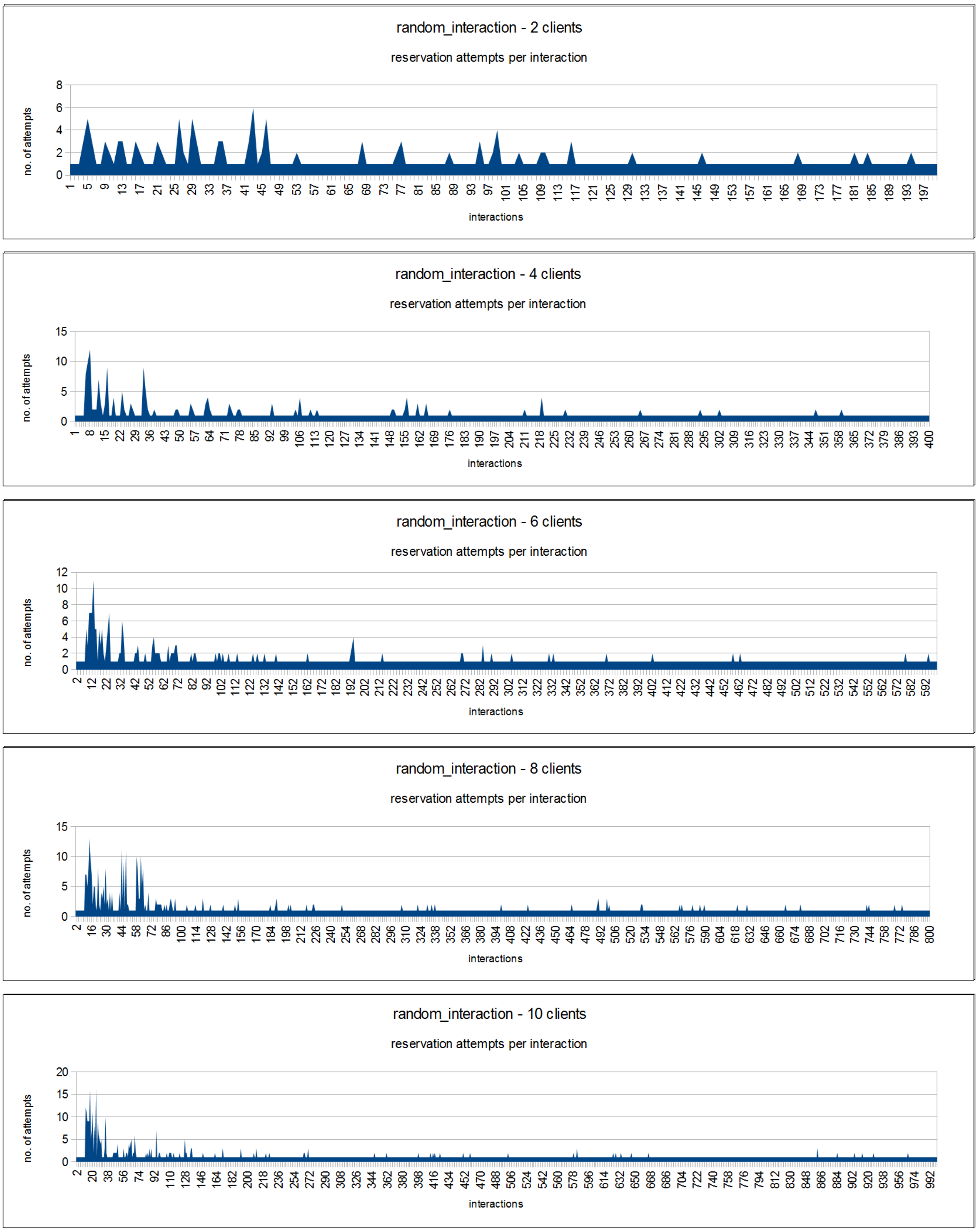


Figure 32: Experiment II - random interactions.

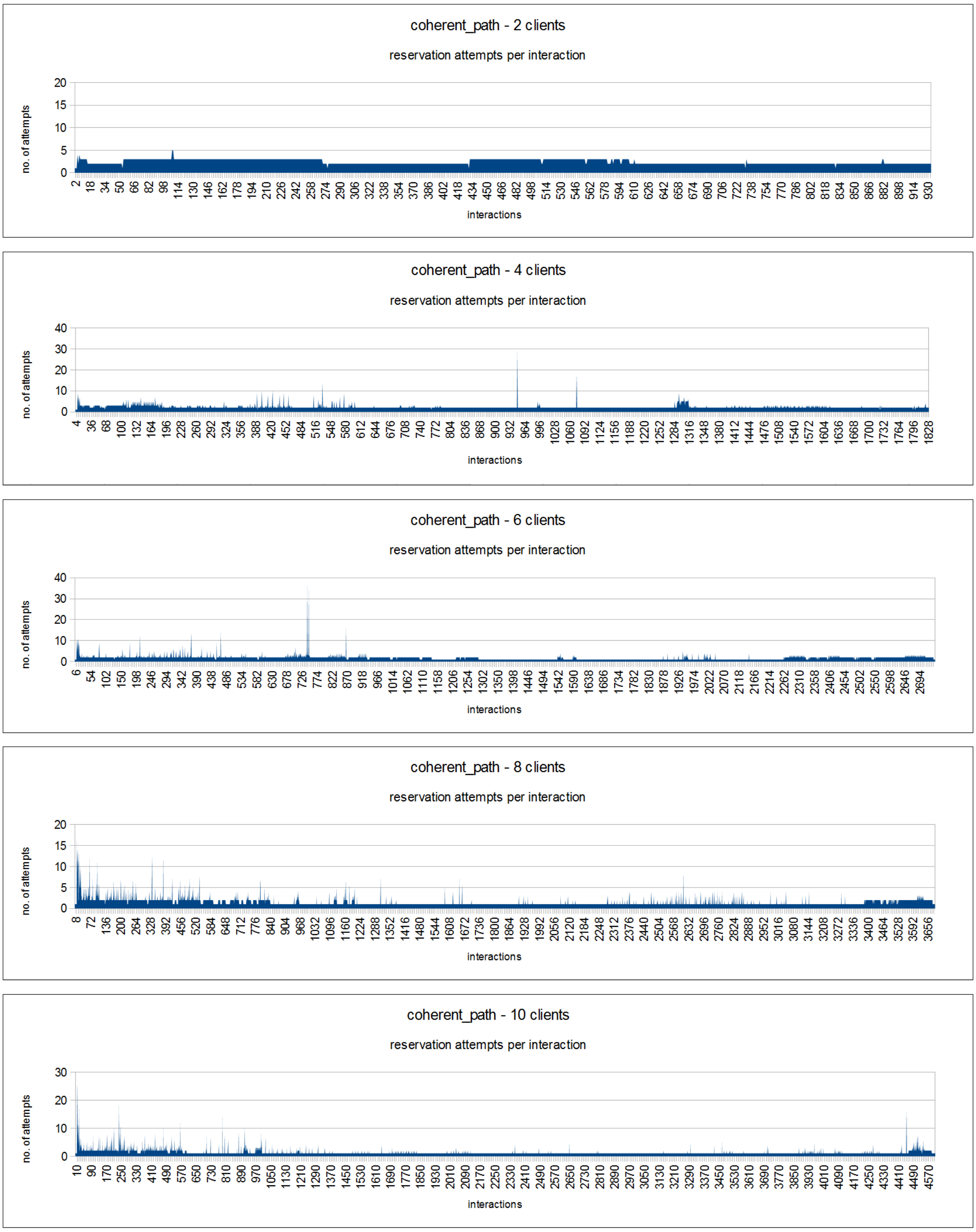


Figure 33: Experiment II - coherent paths.

		total			per interaction			
clients	Combined interactions	attempts	interactions needing multiple attemps	% of total interactions	mean	median	max	standard deviation
2	933	2258	925	99.1	2.4	2	5	0.5
4	1828	4235	1816	99.3	2.3	2	29	1.1
6	2741	4892	1654	60.3	1.8	2	36	1.3
8	3685	5358	1126	30.6	1.5	1	16	1.0
10	4604	6068	861	18.7	1.3	1	27	1.1

Table 5: Experiment II - coherent_path, results.

		total			per interaction			
clients	Combined interactions	attempts	interactions needing multiple attemps	% of total interactions	mean	median	max	standard deviation
2	200	272	39	19.5	1.4	1	6	0.87
4	400	518	50	12.5	1.3	1	12	1.14
6	600	733	66	11	1.2	1	11	0.86
8	800	1035	93	11.6	1.3	1	13	1.20
10	1000	1245	90	9	1.2	1	16	1.21

Table 6: Experiment II - random_interaction, results.

6 CONCLUSIONS

In this chapter we will discuss what conclusions we can draw from our experiments and how these relate to the main two research questions of our thesis.

While the data generated by the experiments is by no means conclusive enough to make any hard statements concerning the use of our methodology over a prolonged period of time, the results do shed light on a number of things which help us understand how it behaves under different scenarios.

In Section 4.1 we talked about the worst-case scenario for quadtrees, something which is caused by very granular, non-contiguous areas of soil. The random interaction scenario causes this kind of granularity (see Figure 20) and the consequences can be seen in the growth rate of the terrain during this scenario. As such we may conclude that, although growth rates seem to decrease once more interactions have been performed, the data structure used in our methodology performs badly when subject to random changes to its content, especially if such change increase the types of soil present in each slice. It performs better when subjected to a coherent path scenario, in which interactions resemble those caused by entities within a virtual world such as player characters. The results show that these interactions affect terrain size more positively due to the more contiguous nature of the changes, which results in larger contiguous areas that can be described with quadtrees that are less complex.

If we look at the question of server-side versus client-side calculation we may conclude that client-side calculation is indeed possible, as most conflicts happen only at the beginning of each experiment. However, although client-side calculation means the server doesn't need to calculate changes any more, it does need to enforce a system of reservation. In this system, each reservation causes both the client and server to do a search for affected nodes and the server needs to compare these results. If the results do not match, this causes the reservation to fail and the entire process will need to be repeated. Because of this added work, client-side handling may actually prove to be slower than server-side handling in terms of absolute calculation time. Unfortunately, neither experiment I nor II contain any data to make such a comparison and so it remains unclear which approach is more efficient.

With this in mind, we can now answer the main research questions stated in Section 2.3 .

Can volumetric terrain data be held synchronous in a client-server network environment with server-side interaction handling where multiple clients initiate such interactions, and is the amount of data needed to facilitate communication and calculation thereof acceptable?

Yes, this can be done, but only with a limited number of clients and results may vary according to the scenario: the best scenarios are those where interactions performed cause contiguous areas of the same soil type (or removal thereof) as this means less terrain growth and thus less data that needs to be communicated. The worst scenarios are those that cause granular, non-contiguous changes, in which case calculation and communication may become too slow to work in real-time.

In a client-server model where interaction handling is delegated to the clients and the server coordinates and communicates terrain updates, how often do two clients initiate interactions which conflict with one another?

The number of reservation attempts needed to do a single interaction depends on the terrain structure and the position of the clients. As described in Section 5.3.3 , a starting terrain may contain larger nodes increasing the chance of conflicts, but this problem is automatically solved after the first few interactions.

7 FURTHER RESEARCH

The experiments that were run as part of this thesis project have answered some questions, but have also given rise to new ones. Furthermore, there are also questions which have not been addressed in this thesis that may require further research. In this final chapter, we will discuss such questions shortly.

Prolonged usage of the methodology – the data produced by both experiments I and II showed that terrain growth seemed to slow down as more interactions were performed. However, this data was insufficient to draw any hard conclusions from. Continued research should test how the data structure used in our methodology behaves under the random interaction and coherent path scenario's when far more interactions are performed.

Client-side versus server-side calculation – in Chapter 6, we mentioned that the our experiments did not produce any data to show the differences in calculation time between client-side and server-side calculation. Therefore, further experiments should focus on measuring performance of both methods in terms of absolute calculation time.

Underground interactions – All experiments have focussed on interactions at the surface of the terrain, meaning underground interaction has not been considered. Such interactions, in a natural terrain, might cause effects which do not occur at the surface, such as cave-ins. Further research could focus on if and/or how the data structure used in our methodology might accommodate such effects.

Visualization of the terrain – This thesis project has focussed completely on data structures and network synchronization and ignores the issue of terrain visualization which is, as we mentioned in Section 3.3.2 , a well-researched subject. Continued research should therefore focus on uniting our methodology with one or more of these established methods.

Large terrains and dynamic terrain loading – although our methodology offers a way to subdivide large sets of volume data, it does not account for limits in memory size. The terrain used in the experiments consisted of a single patch containing only 100 slices, but once loaded into the server program, it took up over 750 MB of memory. In continued research, these issues should be addressed.

8 REFERENCES

[1] R. Smelik, K. de Kraker, S. Groenewegen, T. Tutenel, R. Bidarra . "*A Survey of Procedural Methods for Terrain Modelling*". Proceedings of the CASA Workshop on 3D Advanced Media In Gaming And Simulation, 2009.

[2] J. Doran, I. Parberry. "*Controlled Procedural Terrain Generation Using Software Agents*". Journal: IEEE Transactions on Computational Intelligence and Ai in Games , vol. 2, no. 2, pp. 111-119, 2010.

[3] Mincraft wiki."Chunk format".http://minecraft.gamepedia.com/Chunk_format.18 April 2014.

[4] M. de Berg, O. Cheong, M. van Kreveld, M. Overmars. "*Chapter 9 : Dalaunay Triangulations*". Computational Geometry - Algorithms and Applications, Third Edition. Pages 191-214. Springer-Verlag Berlin Heidelberg, 2008.

[5] L. Chew. "*Constrained Delaunay Triangulations*". Proceedings of the third annual symposium on Computational geometry, Pages: 215-222, 1987.

[6] J. Shewchuk. "*Delaunay Refinement Mesh Generation*". PhD thesis, Carnegie Mellon University, 1997.

[7] P. Jenke, M. Wand, M. Bokeloh, A. Schilling, W. Straßer . "*Bayesian point cloud reconstruction*". Computer Graphics Forum - CGF , vol. 25, no. 3, pp. 379-388, 2006.

[8] H. Hoppe, T. Derose, T. Duchamp, J.McDonald, W. Stuetzle . "*Surface reconstruction from unorganized points*". SIGGRAPH '92 Proceedings of the 19th annual conference on Computer graphics and interactive techniques, Pages 71–78, 1992.

[9] M. Duchaineau, M. Nijinsky, D. Sigeti. "*ROAMing Terrain: Real-time Optimally Adapting Meshes*". IEEE Visualization , pp. 81-88, 1997.

[10] Y. He, J. Cremer, Y. Papelis. "*Real-time Extendible-resolution Display of On-line Dynamic Terrain*". Conference: Graphics Interface - GI , pp. 151-160, 2002.

[11] K. Höhne, M. Bomans, A. Pommert, M. Riemer, C. Schiers, U. Tiede, G. Wiebecke . "*3D-visualization of tomographic volume data using the generalized voxel-model*". Conference: Volume Visualization and Graphics - VolVis , pp. 51-57, 1989.

[12] S. Dogan. "*3D reconstruction and evaluation of tissues by using CT, MR slices and digital images*". ISPRS Congress Istanbul 2004, Commission V papers, Vol. XXXV, part B5. Page(s) 323-327.

[13] B. Geiger. "*Three-Dimensional Modeling of Human Organs and Its Application to Diagnosis and Surgical Planning*". PhD thesis, Institut National de Recherche en Informatique et en Automatique, 1993.

[14] R. Mencl, H. Müller . "*Graph-Based Surface Reconstruction Using Structures in Scattered Point Sets*". Computer Graphics International - CGI , pp. 298-311, 1998.

[15] N. Amenta, M.Bern, M. Kamvysselis . "*A new Voronoi-based surface reconstruction algorithm*". Annual Conference on Computer Graphics - SIGGRAPH , pp. 415-421, 1998.

[16] M. Meißner, J. Huang, D. Bartz, K. Mueller, R. Crawfis. "*A practical evaluation of popular volume rendering algorithms*". Volume Visualization and Graphics - VolVis , pp. 81-90, 2000.

[17] M. Levoy. "*Efficient ray tracing of volume data*". ACM Transactions on Graphics - TOG , vol. 9, no. 3, pp. 245-261, 1990.

[18] L. Westover. "*Footprint evaluation for volume rendering*". Journal: ACM Siggraph Computer Graphics, vol. 24, no. 4, pp. 367-376, 1990.

[19] L. Westover. "*Splatting: a parallel, feed-forward volume rendering algorithm*". Doctoral Dissertation, University of North Carolina, 1992.

[20] P. Lacroute, M. Levoy . "*Fast volume rendering using a shear-warp factorization of the viewing transformation*". Conference: Annual Conference on Computer Graphics - SIGGRAPH, pp. 451-458, 1994.

[21] M. de Berg, O. Cheong, M. van Kreveld, M. Overmars. "*Chapter 14 : Quadtrees*". Computational Geometry - Algorithms and Applications, Third Edition. Pages 307-318. Springer-Verlag Berlin Heidelberg, 2008.

[22] R.A. Finkel, J.L. Bentley. "*Quadtrees : a data structure for retrieval on composite keys*". Acta Informatica - ACTA, vol. 4, no. 1, pp. 1-9, 1974.

[23] H. Samet. "*An algorithm for converting rasters to quadtrees*". IEEE Transactions on Pattern Analysis and Machine Intelligence - PAMI , vol. PAMI-3, no. 1, pp. 93-95, 1981.

[24] H. Samet. "*Neighbor finding techniques for images represented by quadtrees*". Graphical Models /graphical Models and Image Processing /computer Vision, Graphics, and Image Processing - CVGIP , vol. 18, no. 1, pp. 37-57, 1982.

[25] H. Samet. "*An overview of quadtrees, octrees and related hierarchical data structures*". NATO ASI Series, Vol. F40, Theoretical Foundations of Computer Graphics and CAD, Springe-Verlag Berlin Heidelberg 1988.

[26] C. Dyer, A. Rosenfeld, H. Samet . "*Region Representation: boundary codes from quadtrees*". Communications of The ACM - CACM , vol. 23, no. 3, pp. 171-179, 1980.

[27] C. Shaffer, H. Samet. "*Optimal Quadtree Construction Algorithms*". Graphical Models /graphical Models and Image Processing /computer Vision, Graphics, and Image Processing - CVGIP , vol. 37, no. 3, pp. 402-419, 1987.

[28] Z. Chen, I. Chen . "*A simple recursive method for converting a chain code into a quadtree with a lookup table*". Image and Vision Computing - IVC , vol. 19, no. 7, pp. 413-426, 2001.

[29] J. Woodwark. "*Compressed Quad Trees*". The Computer Journal - CJ , vol. 27, no. 3, pp. 225-229, 1984.

[30] H.Samet. "*Neighbor Finding in Images Represented by Octrees*". Graphical Models /graphical Models and Image Processing /computer Vision, Graphics, and Image Processing - CVGIP , vol. 46, no. 3, pp. 367-386, 1989.

[31] H. Chen, T. Huang. "*Survey of Construction and Manipulation of Octrees*". Graphical Models /graphical Models and Image Processing /computer Vision, Graphics, and Image Processing - CVGIP , vol. 43, no. 3, pp. 409-431, 1988.

[32] G. Picinbono, H. Delingette, N. Ayache. "*Non-linear anisotropic elasticity for real-time surgery simulation*". Journal: Graphical Models /graphical Models and Image Processing /computer Vision, Graphics, and Image Processing - CVGIP , vol. 65, no. 5, pp. 305-321, 2003.

[33] M. Bro-Nielsen, S. Cotin. "*Real-time Volumetric Deformable Models for Surgery Simulation using Finite elements and condensation*". Computer Graphics Forum - CGF , vol. 15, no. 3, pp. 57-66, 1996.

[34] H. Nienhuys. "*Cutting in deformable objects*". PhD thesis, Utrecht University, The

Netherlands. 2003.

[35] C. Paloc, F. Bello, R. Kitney, A. Darzi. "*Online Multiresolution Volumetric Mass Spring Model for Real Time Soft Tissue Deformation*". Medical Image Computing and Computer-Assisted Intervention — MICCAI 2002 Lecture Notes in Computer Science Volume 2489, pp 219-226, 2002.

[36] S. Gibson, J. Samosky, A. Mor, C. Fyock, W. Eric L. Grimson, T. Kanade, R. Kikinis, H. Lauer, N. Mckenzie, S. Nakajima, T. Ohkami, R. Osborne. "*Simulating Arthroscopic knee surgery using volume object representations, real-time volume rendering and haptic feedback*". Computer Vision, Virtual Reality and Robotics in Medicine - CVRMed , pp. 369-378, 1997.

[37] D. Causon, C. Mingham. "*Introductory Finite Difference Methods for PDE*". Ventus Publishing ApS, 2010.

[38] R. Leveque. "*Finite difference methods for differential equations*". Draft version for use in AMath, 1998.

[39] O. Zienkiewicz, R.Taylor, J. Zhu. "*The Finite Element Method: Its Basis and Fundamentals*". Elsevier Butterworth-Heinemann, 2005. ISBN-13: 978-0750663205.

[40] R.J. Leveque. "*Finite Volume Methods for Hyperbolic Problems*". Cambridge University Press, first edition, 2002. isbn-13: 978-0521009249.

[41] R. Eymard, T. Gallouet, R. Herbin. "*Finite Volume Methods*". Handbook of Numerical Analysis, P.G. Ciarlet, J.L. Lions eds, vol 7, pp 713-1020.

[42] Y. Liu. "*Fast Multipole Bounadary Element Method - Theory and Applications in Engineering*". Cambridge University Press, 2009. ISBN-13: 9780521116596.

[43] K. Atkinson. "*The Numerical Solution of Boundary Integral Equations*". The state of the art in numerical analysis, pages 223-259., 1997.

[44] L. Lamport. "*Time, Clocks, and the Ordering of Events in a Distributed System*". Communications of the ACM, Vol. 21, Issue 7, July 1978. Pages 558-565.

[45] E. Cronin, B. Filstrup, A. Kurc. "*An Efficient Synchronization Mechanism for Mirrored Game Architectures*". Multimedia Tools and Applications, Vol. 23, Issue 1, May 2004. Pages 7-30.

[46] M. Haley, E. Blake. "*Incremental Volume Rendering Using Hierarchical compression*". Computer Graphics Forum - CGF , vol. 15, no. 3, pp. 45-55, 1996.

[47] T. Günther, C. Poliwoda, C. Reinhart, J. Hesser, R. Männer, H. Meinzer, H. Baur. "*Virim A massively parallel processor for real-time volume visualization in medicine*". Computers & Graphics - CG , vol. 19, no. 5, pp. 705-710, 1995.