Diploma Thesis

Realistic real-time visualization of large-scale network-oriented multi-agent simulations

Mathias Walker

March 25, 2003

Advisor:
Duncan Cavens

Prof. Dr. Kai Nagel
Institute for Scientific Computing
ETH Zürich
# Contents

1 Introduction .......................................................... 1

2 Large-scale terrain visualization ................................. 3
   2.1 Terrain representation ....................................... 3
   2.2 View-dependent refinement .................................. 3
      2.2.1 Longest edge bisection ............................... 4
      2.2.2 Refinement criterion .................................. 6
      2.2.3 Error metrics .......................................... 8
      2.2.4 View frustum culling .................................. 10
      2.2.5 Run-time refinement .................................. 10
   2.3 Data layout and indexing ..................................... 13
      2.3.1 Linear data layout ..................................... 13
      2.3.2 Efficient index computation ........................... 14
      2.3.3 Memory mapping ...................................... 14
   2.4 Data preprocessing ........................................... 15
      2.4.1 Vertex representation .................................. 15
      2.4.2 Bottom-up propagation ................................. 15

3 Network and agent representation ............................... 17
   3.1 Network ....................................................... 17
      3.1.1 Nodes .................................................. 17
      3.1.2 Links .................................................. 17
   3.2 Agents ....................................................... 18

4 Integration .......................................................... 19
   4.1 Merging road network, agents and terrain ................. 19
      4.1.1 Difficulties ........................................... 19
      4.1.2 Solution ............................................... 21
   4.2 Multiresolution Texturing .................................. 21
      4.2.1 View-dependent texture quadtree .................... 22
      4.2.2 Other advantages to this approach .................... 23
   4.3 Changes to terrain creation ................................ 24
1 Introduction

Multi-agent simulations (MAS) are a relatively recent computer simulation method. MAS work done in the Institute for Scientific Computing (ETHZ) mostly deals with car and pedestrian traffic. The travelers are modeled as agents that move along a network (e.g. roads). In this work a tool for realistic real-time visualization of these scenarios has been developed.

In 3D graphics, objects are generally rendered as a series of triangles. Large-scale terrains need a large amount of triangles to render. The topography of e.g. Switzerland is available at a resolution of 25 m and has about $5 \cdot 10^8$ triangles, which is too much to render in real-time on current hardware. For this reason, an existing adaptive level-of-detail (LOD) algorithm has been implemented and where necessary adapted.

The network for the travelers is typically not given at the same resolution as the terrain mesh. This results in artifacts when the network is rendered on top without adjustments. In this work a technique to merge the terrain and the network has been developed. This network rendering technique has been integrated with the adaptive terrain.

The implementation has been evaluated regarding performance and visual quality.

This documentation assumes that the reader has a basic knowledge of computer graphics. Introductions to this topic can be found in [13], [14], [15], [16].
2 Large-scale terrain visualization

It is not feasible to render a terrain in the highest resolution on current hardware (10000 x 10000 vertices result in \( \approx 2 \cdot 10^8 \) triangles). This amount of data cannot be loaded entirely into memory. Even graphics accelerator hardware is not fast enough to render meshes of this size in real-time.

The goal of any level-of-detail (LOD) algorithm is to reduce the number of needed triangles to render a good approximation of the original, full resolution mesh. Only a subset of the original height field is selected. The mesh can be even more reduced by taking into account the position of the camera, the viewpoint. Basically we need more details in the foreground and for areas that are not so smooth.

The level-of-detail algorithm used in this work is based on the papers [1] and [2]. Some diagrams and figures in this chapter were borrowed from [2]. The papers provide a general framework for performing highly interactive view-dependent rendering that is simple to implement.

The refinement is done top-down and supports several error metrics. The data layout is independent of the refinement algorithm. Instead of complicated and explicit data paging techniques the built-in paging mechanisms of the operating system are used.

2.1 Terrain representation

The terrain surface is represented as an uniformly spaced rectangular grid of height values. For every grid point \((x, y)\) a height \(z\) is defined. A continuous surface \(z(x, y)\) is formed by linear interpolation which results in a piece-wise linear triangle mesh.

2.2 View-dependent refinement

The mesh construction is done at run-time. The mesh is updated whenever the viewpoint has moved a certain amount. If the camera stays in the vicinity the mesh will not change. Every refinement is made from scratch and is independent from
the previous mesh.

The difference between the original mesh and its approximation is called the *object-space* error $\epsilon$. It is commonly measured as the vertical deviation between corresponding points. This error is projected onto the screen to get a view-dependent measure of error $\rho(\epsilon)$. This algorithm refines the mesh top-down, i.e. it splits triangles of a coarser mesh to construct a finer mesh until the projected errors meet some tolerance.

### 2.2.1 Longest edge bisection

![split edge](image1)

**Figure 2.1**: split edge $e = \{v_l, v_r\}$ and diamond $T = \{t_b, t_t\}$ of a vertex $v$

In this refinement algorithm a particular type of subdivision based on *longest edge bisection* is used. The resulting meshes can be refined locally without the need to keep the same resolution for the entire mesh. In the edge bisection scheme, two smaller triangles are created by bisecting the hypothenuse of an isosceles right triangle. The bisected edge for the inserted vertex $v$ is called the *split edge* $e_v$ of $v$. The two triangles sharing $e_v$ are called the *diamond* $T_v$ of $v$ (Figure 2.1).

![edge bisection hierarchy](image2)

**Figure 2.2**: Edge bisection hierarchy. The arrows correspond to parent-child relationships in the DAG.

The algorithm starts with a coarse mesh of four triangles and makes an adaptive, recursive refinement. The decision whether to split an edge depends on a refinement criterion. This is based on the approximation quality of the current diamond
and on the position of the viewpoint.

![Figure 2.3: The terrain is enlarged to the next valid size $2^n + 1$ if the dataset does not match this constraint. Vertices outside the dataset are given the dummy value $z_{\text{outside}}$.](image)

The subdivision creates vertices that map directly to points on a regular, rectilinear grid. Thus it is very suitable to use the edge bisection hierarchy as a multi-resolution representation for approximating terrain surfaces. The dimensions of the grid are constrained to $2^{n/2} + 1$ vertices in each direction, where $n$ is the even number of refinement levels. The refinement levels are called alternatingly “white tree” and “black tree” according to the white and black vertices in Figure 2.2. If a dataset does not match this constraint, the terrain is enlarged to the next valid size. All vertices outside the dataset are given the dummy value $z_{\text{outside}}$ (Figure 2.3).

The advantage of refinement versus simplification is that its computational complexity and number of vertex accesses is linear in the size of the approximating mesh. Simplification starts with the highest resolution and its complexity depends on that size. Therefore, runtime refinement is faster.

Edge bisection creates a mesh that can be represented as a directed acyclic graph (DAG) of its vertices. A directed edge from $i$ to one of its children $j$ in the DAG corresponds to a triangle bisection, in which $j$ is inserted on the split edge and connected to $i$ at the apex of the triangle (Figure 2.1). All vertices that are not leaves and not on the mesh boundary are connected to four children in the DAG and have two parent vertices. Boundary vertices have two children and one parent. A vertex is active if it is included in a refinement $M$. $M$ is valid if it does not have any holes and T-junctions, so it forms a continuous surface. For $M$ to be valid it must satisfy

$$j \in M \Rightarrow i \in M \quad j \in C_i$$

(2.1)
where $C_i$ is the set of children of $i$ in the DAG. This means that for a vertex $j$ to be active, all its parents and ancestors must be valid (Figure 2.4).

![Figure 2.4: Example of adaptively refined mesh. The dotted edges must be added to avoid cracks (see on the right) in the mesh.](image)

To guarantee the validity of the mesh and satisfy Equation 2.1, the error terms used in the refinement criterion are nested so that all parent vertices are activated with their descendants.

### 2.2.2 Refinement criterion

For a given mesh vertex $p_i$ and a fixed viewpoint $e$, an increase of the object space error $\epsilon$ should lead to an increase of its projection $\rho(\epsilon, p_i, e)$. This means that if an terrain difference increases it should also give an increased difference from the observer at the viewpoint. The view-dependent error of the parents has to be greater than or equal to the error of its children and thus its descendants

$$\rho(\epsilon_i, p_i, e) \geq \rho(\epsilon_j, p_j, e) \quad \forall j \in C_i \quad (2.2)$$

This nested errors are sufficient for satisfying 2.1. Visiting every descendant of each active vertex at run-time is not practical for large terrains, since the set of descendants increases exponentially in size. A nested DAG of spheres is used to compute a conservative bound on $\rho_i$.

$\rho_i$ has two components: an object space error term $\epsilon_i$ and a view-dependent term that relates $p_i$ and $e$. The two are separated and a nesting for each term is guaranteed as follows. Let

$$\epsilon_i = \begin{cases} 
\hat{\epsilon}_i & \text{if } i \text{ is a leaf node} \\
\max \left\{ \hat{\epsilon}_i, \max_{j \in C_i} \{\epsilon_j\} \right\} & \text{otherwise}
\end{cases} \quad (2.3)$$
where $\hat{\epsilon}_i$ is the actual (not necessarily nested) geometric error measured by the object space metric. The error $\epsilon_i$ is greater than or equal to the errors $\epsilon_j$ of all descendants $j$ of vertex $i$. The monotonic relationship between $\rho_i$ and $\epsilon_i$ leads to $\rho(\epsilon_i, p_i, e) \geq \rho(\hat{\epsilon}_i, p_i, e)$, which means no loss in visual accuracy. Unfortunately $\rho(\epsilon_i, p_i, e) \geq \rho(\epsilon_j, p_j, e)$ for $j \in C_i$ is not necessarily true, as the viewpoint may be closer to $p_i$ than to $p_j$. The nested object space errors are not sufficient. Therefore one has also to take into account the spatial relationship between parent and child vertices.

\[
B_i = \{ x : \| x - p_i \| \leq r_i \} 
\]

The radius $r_i$ is then

\[
r_i = \begin{cases} 
0 & \text{if } i \text{ is a leaf node} \\
\max_{j \in C_i} \{ \| p_i - p_j \| + r_j \} & \text{otherwise}
\end{cases} 
\]

Then the sphere hierarchy is nested as $B_i \supseteq B_j$ for $j \in C_i$, i.e. each bounding sphere contains the bounding spheres of all its descendants. A 2D example is shown in Figure 2.5.
Large-scale terrain visualization

Figure 2.5: Each triangle is associated with the vertex at its right-angle corner. The bounding spheres contain their triangles on level 3 and above but not on the bottom two levels of the DAG. The maximum projected error is defined as

$$
\rho_i = \rho(\epsilon_i, B_i, e) = \max_{x \in B_i} \rho(\epsilon_i, x, e)
$$

Because $$\epsilon_i \geq \epsilon_j$$, $$B_i \supseteq B_j$$ and $$\rho$$ is monotonic, $$\rho_i \geq \rho_j$$ for $$j \in C_i$$. The projected error of a parent is always greater or equal than the projected error of its children. If $$j$$ is active then so is its parent $$i$$.

2.2.3 Error metrics

The error metrics consist of the object space error metric $$\epsilon$$ and the screen space error metric $$\rho$$.

Object space error metrics

The object space error used in this implementation is measured incrementally between two consecutive levels of refinement. The incremental error tells how much the mesh would change by removing the vertex.

Figure 2.6: The incremental error $$\hat{\epsilon}^{inc}_i$$ indicates how much the mesh would change by omitting vertex $$i$$.

Let $$z_t(x_i, y_i)$$ be the elevation of triangle $$t$$ at the point $$(x_i, y_i)$$ in the domain where the vertex $$i$$ lies. The vertical error between $$i$$ and $$t$$ is defined as

$$
\hat{\delta}_{i,t} = |z_i - z_t(x_i, y_i)|
$$

The incremental error can then be written as

$$
\hat{\epsilon}^{inc}_i = \max_{t \in T_i} \hat{\delta}_{i,t} = \left| z_i - \frac{z_l + z_r}{2} \right|
$$

where $$T_i$$ are the triangles that share $$i$$’s split edge (Figure 2.1). This is the vertical displacement from $$i$$ to the midpoint of its split edge $${v_l, v_r}$$ (Figure 2.6). The split
2.2 View-dependent refinement

edge is common to all $T_i$ if the mesh is valid.

**Isotropic error projection**

The screen space error $\rho(\epsilon)$ is the projection of the object space error $\epsilon$ onto the screen. The distance along the view direction is substituted with the Euclidean distance

$$d = \| \mathbf{e} - \mathbf{p} \|$$

between the viewpoint $\mathbf{e}$ and the vertex position $\mathbf{p}$. The same object error should appear smaller if it is further away from the viewpoint, due to perspective projection. A simple metric where the projected error decreases with distance from the viewpoint can be written as

$$\rho(\epsilon, \mathbf{p}, \mathbf{e}) = \lambda \frac{\epsilon}{\| \mathbf{e} - \mathbf{p} \|} = \frac{\epsilon}{d}$$

The projected error is the same in every direction at a fixed distance $d$ from the vertex and therefore isotropic. Equation 2.8 is a projection onto a sphere, so $\lambda = \frac{w}{\varphi}$, where $w$ is the number of pixels along the field of view $\varphi$. Then $\rho$ is compared against an user-specified screen space error tolerance $\tau$.

In the refinement procedure, the maximum projection $\rho(\epsilon, B, \mathbf{e})$ over a set of points $B$ has to be found (see 2.2.2). The maximum projection for Equation 2.8 occurs where $d = \| \mathbf{e} - \mathbf{x} \|$ is minimized. This term is zero for viewpoints inside $B$ and the vertex is activated. If the viewpoint is outside $B$ then the minimum is $d - r$ and the maximum screen space error becomes

$$\rho(\epsilon, B, \mathbf{e}) = \max_{\mathbf{x} \in B} \rho(\epsilon, \mathbf{x}, \mathbf{e}) = \lambda \frac{\epsilon}{d - r}$$

The vertex $i$ is active when

$$active(i) \Leftrightarrow (\nu \epsilon_i + r_i)^2 > d_i^2$$

where $\nu = \frac{1}{\lambda}$ is constant during each refinement. For spherical projection $\kappa = \frac{1}{\nu} = \frac{\varphi}{\pi}$ is the angular error threshold in radians. Expression 2.10 is very efficient to evaluate as it involves only six additions and five multiplications.
2 Large-scale terrain visualization

2.2.4 View frustum culling

The rendering performance is improved by view frustum culling, i.e. by not refining mesh triangles that fall outside the visible part on the screen. The view culling exploits the hierarchy of the subdivision mesh and discards large chunks of triangles high up in the mesh hierarchy whenever possible. The mesh resolution outside the view volume will be much lower (Figure 2.7).

The bounding sphere for a vertex \( i \) contains the vertices of all descendants of \( i \). This nested bounding sphere hierarchy can be exploited. If the bounding sphere is invisible then neither \( i \) nor its descendants will appear on the screen.

The parameters for the four planes of the view frustum (left, right, bottom, top) are passed along in the refinement. The parameters are \((n_k, d_k)\) where \( n_k \) is the normalized normal vector of the frustum plane, pointing inside the view volume, and \( n_k \cdot x + d_k = 0 \) with \( x \) a point on the plane (e.g. the viewpoint).

If the bounding sphere is inside a frustum plane then all descendant’s bounding spheres must also be inside and no further culling tests against this plane are necessary. If the sphere is inside all frustum planes the regular refinement without frustum culling is used. A vertex and its descendants are culled if the sphere is completely outside all frustum planes. View culling is done only for spheres intersecting the frustum planes.

The mesh always remains a continuous surface because the spheres are nested and thus a child is only visible if its parents are.

2.2.5 Run-time refinement

Here is a summary of the algorithm for top-down, recursive refinement and on-the-fly triangle strip construction. Pseudo code for the refinement is also listed here.

The refinement procedure builds a triangle strip represented as a list of vertex indices \( V = (v_0, v_1, v_2, \ldots, v_n) \). The procedure \texttt{tstrip-append} appends a vertex \( v \) to the strip. To form a valid triangle mesh the vertices are alternatingly on an even or odd refinement level. To ensure that this parity is alternating some vertices have to be swapped which is done in Line 5. In Figure 2.8 the sequence of triangles traversed is illustrated.

The recursive traversal of the mesh hierarchy is done in \texttt{submesh-refine} where \( c_l \)
2.2 View-dependent refinement

Figure 2.7: View frustum culling as seen from the viewpoint and from top

and $c_r$ are the “left” and “right” child vertices of $j$ in the DAG. The sense of left and right alternates between consecutive levels as shown in Figure 2.10.

The refinement starts with a base mesh of four triangles in mesh-refine (Figure 2.2). For each triangle submesh-refine-visible is called with $n = 2^m$ the number of refinement levels for a terrain of size $(2^m + 1) \times (2^m + 1)$ and the vertices named like in Figure 2.8.

\begin{verbatim}
tstrip-append(v, p)
1  if $v \neq v_{n-1}$ and $v \neq v_n$ then
2    if $p \neq \text{parity}$ then
3      \text{parity} = p
4    else
5      append $v_{n-1}$
6      append $v$
\end{verbatim}
2 Large-scale terrain visualization

Figure 2.8: Traversal of triangle strip. At the marked vertex swapping is needed.

```
submesh-refine\((i, j, l, v_{left}, v_{right})\)
1  \(refine = l > 1\) and \(active(j)\)
2  if \(refine\) then
3    submesh-refine\((j, c_l(i, v_{left}), l-1, v_{left}, i)\)
4    tstrip-append\((i, l \mod 2)\)
5  if \(refine\) then
6    submesh-refine\((j, c_r(i, v_{right}), l-1, i, v_{right})\)

submesh-refine-visible\((i, j, l, v_{left}, v_{right}, inside)\)
1  if \(inside\) all view frustum planes then
2    submesh-refine\((i, j, l, v_{left}, v_{right})\)
3  else
4    \(refine = l > 1\) and \(active(\text{ancj})\) and \(visible(\text{ancj}, inside)\)
5  if \(refine\) then
6    submesh-refine-visible\((j, c_l(i, v_{left}), l-1, v_{left}, i, inside)\)
7    tstrip-append\((i, l \mod 2)\)
8    submesh-refine-visible\((j, c_r(i, v_{right}), l-1, v_{right}, i, inside)\)

mesh-refine\((n)\)
1  init tstrip with \(\{i_{sw}, i_{sw}\}\)
2  \(parity = 0\)
3  \(inside_k = 0\) \(\forall k\)
4  for each \((j, k, l) \in ((i_s, i_{se}, i_{sw}), (i_c, i_{ne}, i_{se}), (i_n, i_{nw}, i_{ne}), (i_w, i_{sw}, i_{nw}))\)
5    submesh-refine-visible\((i_c, j, n, l, k, inside)\)
6  tstrip-append\((k, 1)\)

active\((i)\)
1  return \((v_{ci} + r_i)^2 > d_i^2\)
```
2.3 Data layout and indexing

Not all data can be loaded into main memory at once. One has to find a strategy that efficiently caches the data being used, and anticipates data that will be used in the next frame. There are two parts for efficient memory performance:

The first part is how to layout the terrain data on disk such that the access to the vertices is optimal. Data access is much faster if consecutive accesses are spatially close on the disk.

The second is the problem of out-of-core paging of the data. This is solved by using the virtual memory paging mechanism of the operating system with the `mmap` system call (under Linux/UNIX).

2.3.1 Linear data layout

In this implementation the simplest data layout of row by row indexing was chosen (Figure 2.9). Each row of the terrain is saved as a row on disk. Vertices on the same tree level are scattered over the whole terrain and are not spatially close. The vertices have the same spatial distribution on disk. This layout is not efficient and
results in a bad caching performance when accessing the vertices during refinement (see §5.1.5).

### 2.3.2 Efficient index computation

![Binary triangle tree formed by bisection. The arrows show the alternating triangle orientation on consecutive levels.](image)

An advantage of linear order indexing is the simple and efficient index computation. The child indices in the DAG can be easily computed by carrying along the indices \((v_l, v_m, v_r)\) of the current triangle \(t\) in the refinement (Figure 2.10). The two child triangles of \(t\) can then be written as \(t_l = (v_l, v_m, v_a)\) and \(t_r = (v_a, v_m, v_r)\) where \(v_m\) is the vertex at the midpoint of the split edge \(\{v_l, v_r\}\). The index of \(v_m\) is computed as the index average \(v_m = (v_l + v_r)/2\).

### 2.3.3 Memory mapping

A specific disk file is mapped to a part of the logical address space of the computer with \texttt{mmap}. The vertices are then accessed as if they were allocated in main memory and the operating system takes care of paging the data from disk.
2.4 Data preprocessing

Unfortunately there is a limitation of about 2 GB on the maximum addressable size. Up to 4 GB of memory could be addressed with a 32 bit architecture, but the operating system limits the usable memory size to 2 GB for each process. As the terrain files can easily be larger than this, they are split into several files of smaller size. Only one file is memory mapped at a time and is unmapped when a vertex of another file is accessed. There is no limitation in terrain size, at the cost of some loss in efficiency (see Section 5.1.4).

This problem was not mentioned by [1] and [2], though they used a dataset larger than 2 GB.

2.4 Data preprocessing

Before the refinement can be run, the terrain data has to be initialized with the information needed by the algorithm. The data preprocessing computes the error terms $\epsilon$ and radii $r$ for each vertex. This is not done at run-time but off-line and only once for a given terrain.

There are three steps in the preprocessing:

1. First the final terrain size is determined and the necessary number of empty files created so they can be memory mapped.

2. In a second step, one or several raw terrain patches are added to the terrain. If the patches don’t fill out the whole terrain, all outlying vertices are set to a fixed height $z_{\text{outside}}$.

3. The final step consists of the computation of the error and bounding sphere values for all vertices.

2.4.1 Vertex representation

For each vertex, its position $(x, y, z)$, the error term $\epsilon$ and the bounding sphere radius $r$ are stored. All of this fields are represented as 32-bit floating point numbers. This results in 20 Bytes of storage required for each vertex.

2.4.2 Bottom-up propagation

Because $\epsilon$ and $r$ are defined recursively, they must be computed and propagated bottom-up from the DAG children to their parents. The vertices are processed
level-by-level and each vertex is visited only once.

Starting with the leaves of the DAG all white and black trees are traversed. The white quadtree on refinement level \( l \) has a single square grid of dimensions \( 2^l \times 2^l \). The black quadtrees have two overlapping grids, one vertically and one horizontally aligned, of size \( 2^l \times (2^l + 1) \) (see Figure 2.2).

![Offset rules for a white resp. black vertex \( i \) on a terrain with side-length \( s = 2^n + 1 \). \( w = 2^{n-l-1} \) and \( b = 2^{n-l-2} \) on quadtree level \( l \).](image)

**Figure 2.11**: Offset rules for a white resp. black vertex \( i \) on a terrain with side-length \( s = 2^n + 1 \). \( w = 2^{n-l-1} \) and \( b = 2^{n-l-2} \) on quadtree level \( l \).

There are simple offset rules to reach the four children from a vertex \( i \) on a terrain with \( s = 2^n + 1 \) vertices width (Figure 2.11). Black neighbours of a white vertex on level \( l \) are at a vertical resp. horizontal distance of \( w = 2^{n-l-1} \) vertices. White neighbours of black vertices are \( b = 2^{n-l-2} \) vertices in \( x \)- and \( y \)-direction away.
3 Network and agent representation

One of the goals of this work is to visualize large road networks and simulated traffic. In traffic simulations, road networks are usually represented as graphs. The simulation is not part of this work. It uses a multi-agent approach to simulate cars [8].

3.1 Network

The road network is represented as graph consisting of a set of 2D points, the nodes, and a set of links which connect two nodes each [8] (Figure 3.1).

Figure 3.1: Nodes and links form a network.

3.1.1 Nodes

Each node has an unique ID number and geographical coordinates.

3.1.2 Links

Every link has also an unique ID number and the IDs of the nodes where the link starts and ends. There is additional information about the length, because a curved road is longer than the Euclidean distance. Other data are the number of lanes and the capacity, i.e. the number of cars that can leave the link per 12 hours. This is used for simulation purposes. For the rendering only the coordinates of start and end nodes are relevant.
3 Network and agent representation

3.2 Agents

An agent is equivalent to a single car or pedestrian.

The movement of the agents is determined in an separately developed large-scale network-oriented multi-agent simulation. This simulation is done offline and its results can be accessed through an agent player. The player uses the libagentcom library by Christian Gloor [9] which is not part of this work.

The visualizer can communicate with one or several players. The player sends IDs and positions of all agents that are inside a chosen area.
4 Integration

The last chapters described how to render large terrains and how to represent networks and agents. In those chapters, they were considered completely independently of each other. Now these two parts have to be merged to get a visualization that shows roads and cars on a 3D terrain. This chapter describes how to put these two parts together and how to implement the techniques.

4.1 Merging road network, agents and terrain

There are several difficulties in mapping a 2D network to a 3D terrain.

4.1.1 Difficulties

![Figure 4.1: Resolution differences between terrain and geometry lead to artifacts as seen on the left. On the right side, the same scene with correct mapping.](image)

One problem of mapping from 2D to 3D is to calculate the $z$-coordinates for a $(x, y)$ position. One has to find the triangle $(x, y)$ lies in and then getting the height $z(x, y)$ by interpolating the heights of its vertices.

The network usually does not have the same resolution as the underlying terrain. This leads to rendering artifacts if no adjustments are made (Figure 4.1). Non
Figure 4.2: The geometry has to be subdivided to fit the terrain. This leads to non trivial calculations and an increase in the number of geometric primitives.

trivial calculations are necessary to subdivide the network geometry, so that it fits the terrain mesh. The subdivision causes also an increase in geometric primitives (Figure 4.2). A change to the terrain mesh would be even more complicated because keeping this mesh valid is extremely difficult.

Figure 4.3: Z-buffer artefacts when rendering coplanar quads.

Coplanar geometries can result in z-buffer artifacts (Figure 4.3) as the renderer can not decide which geometry to render on top. Even if one geometry is slightly on top of the other there can be artifacts if they are far away from the viewpoint, because the z-buffer has only a limited resolution.

All calculations could be done in a preprocessing step on a static terrain. With a dynamic view-dependent terrain, the mesh is constantly recomputed. All triangle searches and subdivision calculations have to be done from scratch for each new mesh. Doing all this at run-time is not feasible.
4.2 Multiresolution Texturing

4.1.2 Solution

A solution is to add the network geometry to the terrain not directly but to render it to a texture which is then applied to the terrain mesh \cite{4}. The advantage of this approach is that the texture mapping automatically follows the height of the terrain. There are no z-buffer artifacts nor does the network geometry have to be subdivided.

A dynamic network, e.g. moving agents, requires the texture to be updated every frame. This has also be done if a view dependent texturing technique is used.

4.2 Multiresolution Texturing

The simplest way would be to render just one big texture. Unfortunately the texture memory on graphics hardware is limited. Typical memory sizes on graphics accelerators are 32 or 64 MB. Not all of it can be used for textures. A texture with e.g. 4096 × 4096 pixels and 4 Bytes per RGBA pixel uses 64 MB of memory. This resolution is not detailed enough for large terrains. So there should be used a hierarchical model similar to the terrain refinement where the system generates higher resolution patches used in the foreground.
4 Integration

Figure 4.5: View frustum culling of texture patches as seen from the viewpoint and from top. Checkerboard patches are invisible and no geometry is rendered into them.

4.2.1 View-dependent texture quadtree

Kersting and Döllner [4] propose a quadtree structure where every patch in the texture tree has the same resolution in pixels but has different sizes in world dimensions (Figure A.7). This approach is used in this implementation.

![Figure 4.6: Texture tree plane as seen from the side.](image)

The plane of the texture tree has the same dimensions as the terrain. It is fixed on a height $z_{\text{tex}}$ (Figure 4.6). This is only an approximation of the real terrain.

To generate the view dependent quadtree the following algorithm is used. It is basically the same idea as described in [5] only in 2D and with a variable distance metric.

A patch is refined if it is visible and if the distance $d$ between viewpoint and patch middle point $M$ is smaller than a threshold factor $t$ times the patch sidelength $s$. The refinement splits the patch at $M$ into four children nodes which are refined recursively. This is done up to a user defined maximal tree depth. An example is
4.2 Multiresolution Texturing

**Figure 4.7:** Top view of texture tree generation. Crossed out patches are outside the view frustum \( a \) and are not further refined or rendered into. If the distance from the viewpoint to the middle \( M_i \) of a visible patch is short enough (blue), this patch is refined.

shown in Figure [4.7](#). Only visible texture patches are rendered (Figure [4.5](#)).

One pixel in the coarsest resolution is \( \frac{t}{p^m} \) m and \( \frac{t}{p \cdot 2^m} \) m in the finest resolution, where \( t \) is the terrain sidelength in m, \( m \) the maximum depth of the texture tree and \( p \) the width and height of the texture patch in pixels. Example: Switzerland with 25 m resolution results in a terrain with 16385 \( \times \) 16385 vertices and \( t = 409600 \) m.

A 512 \( \times \) 512 pixel texture patch allows resolutions ranging from 800 m/pixel at the root up to 0.20 m/pixel on level 12 in the texture tree.

### 4.2.2 Other advantages to this approach

**Figure 4.8:** Additional data can be added as new layers. All layers are rendered to texture.
This solution has also other advantages. Because every geometry can be rendered to texture it is easy to change the network representation or add additional details. Every feature can be represented on a separate geometry layer (see \[\text{A.2.2}\]). All layers are then rendered to texture.

If one wants to represent e.g. the roads as polygons, there is no need to think of how to apply them to the terrain. Just change the road representation and then render it to the texture.

Additional data like lakes or cities can be easily added. This is done by creating new geometries in the scene graph that is rendered to texture (Figure 4.8).

4.3 Changes to terrain creation

The integration of the view dependent texture tree leads to some changes in the terrain mesh creation. There has to be found a way to apply the textures to the mesh. A geometry primitive cannot have multiple textures, so the one triangle strip from the refinement has to be split into several parts according to the texture quadtree.

Kersting and Döllner [4] assume that the terrain mesh has a quadtree structure. The triangle strip generated by the refinement algorithm does not. But it can be mapped in most cases to a quadtree if the smallest quadtree patch spans several triangles of the refinement (Figure 4.9). There are rare cases where a triangle spans several texture patches (see \[\text{A.1}\]). In the current implementation they are simply skipped. Fortunately this is not really a problem with real world datasets because
no landscape is perfectly flat. There are almost always enough triangles so that the texture tree can be mapped exactly. The problem can also occur if the texture tree is too deep. Then the texture patches span only a few mesh triangles.

Each vertex of this smaller strips is assigned the texture coordinates for the corresponding texture patch (see A.2.2).

There are no changes to the refinement algorithm itself. It is still independent of the network rendering.

4.4 Implementation

This work was implemented in C++. The Open Scene Graph 0.9.3 library (OSG) was used for the graphics part. The communication with the external agent player is handled by the libagentcom library by Christian Gloor.

OSG is a high level wrapper around OpenGL. It uses a scene graph which is a hierarchy of nodes. Nodes can contain geometry, light or manipulate other nodes. OSG handles the basic visibility functions, texture management, sorting, state management and windowing system among other things. This requires a well structured database. The library is designed for static databases but flexible enough to handle dynamic data. Other features can be added quite easily.

The implementation, except the memory mapping, is portable to common platforms. The memory mapping is encapsulated in a single class and can be easily changed.

4.5 Program structure

This section gives an overview of the program structure of the implementation. Additional details can be found in A.2. The program is organized in modules with different functionality, e.g. a module for the refinement algorithm or one for creating the texture tree. The modules correspond more or less the C++ classes.

In an initialization phase the user defined parameters are parsed (see A.2.3). The user can select which terrain and which network are rendered and chooses an error tolerance for the refinement. He can also define an optional starting viewpoint. The road network is loaded from disk and the communication with the agent player is
The main loop is described schematically in Figure 4.10 and Table 4.1. A more detailed overview is shown in Figure 4.11. First a view dependent refinement of the terrain from the current viewpoint is made. This is a list of vertex indices. In the next step the view dependent texture tree is created. Then the triangle strip is created from the refinement vertex list. The lighting of this strip is calculated before the strip is split into smaller triangle strips according to the texture tree structure. The corresponding texture patch of the tree is assigned to each of this new strips. Now the network and the agents are rendered to all visible patches of the texture tree. Finally the textured terrain mesh is displayed on screen.

Up to now the user had to wait. Now he can interact with the visualizer and move the camera. If the camera has not moved much in the next loop (frame), only the textures are rerendered to show changes in the agent positions, the mesh stays the same. If the camera has moved more than the given tolerance, a new refinement is made and the steps above repeated.

Figure 4.10: Interaction between the modules
4.5 Program structure

1. init
2. refine
3. create texture tree
4. make mesh
5. lighting
6. split triangle strip
7. apply textures
8. render textures
9. display
10. if camera inside tolerance goto 8
11. else goto 2

**Table 4.1:** Main event loop

**Figure 4.11:** Detailed overview of class interaction
5 Results

The goal of this work is to realistically visualize large-scale network-oriented multi-agent simulations in real-time. The visualizer can render large terrain data interactively in real-time on a standard PC. An example is the Switzerland dataset with a resolution of 25 m that uses 5.3 GB on disk (Figure 5.1). The road network and the agents are efficiently integrated into the terrain without artefacts. The overall visual quality is enhanced by using a satellite image as ground texture.

![Figure 5.1: Rendering of Switzerland with traffic. Roads are yellow, agents are red and nodes are blue. The satellite image only covers Switzerland. The outlying area is filled with a green color, as seen in the top left image.](image)

The paging speed is rather slow (Section 5.1.5) and the agent animation is not opti-
mal (Section 5.1.7). The slow agent animation is more a problem of the libagentcom library. The traffic simulation uses a more sophisticated internal road representation, so the agents are a bit offset. There is also an issue with the representation of tunnels and bridges (see Chapter 6). Tunnels do not go through the mountain but are rendered on top (Figure 5.2).

![Figure 5.2: Representation of the Gotthard tunnel.](image)

5.1 Benchmarks

5.1.1 System info

This computer system was used for benchmarking:

- P-III 600 MHz
- 256 MB RAM
- 1 GB swapfile
- GeForce 2 GTS 32 MB RAM
- RedHat Linux 7.3

5.1.2 Test data

For the benchmarking, terrain data based on the Switzerland dataset [18] with 25 m resolution was used. Due to memory map limitations (see 2.3.3) the terrain was split into smaller files with a maximum size of 1’000’000’000 bytes. All terrain data is accessed on a file server via the network.

The simplified network of swiss roads has a total of 10565 nodes and 28625 links. Only the nodes and links falling inside the terrain area are used.
5.1 Benchmarks

<table>
<thead>
<tr>
<th>number of vertices</th>
<th>size on disk</th>
<th>number of files</th>
<th>max. #triangles</th>
</tr>
</thead>
<tbody>
<tr>
<td>1025 \times 1025</td>
<td>21'012'500 Bytes</td>
<td>1</td>
<td>2'097'152</td>
</tr>
<tr>
<td>2049 \times 2049</td>
<td>83'968'020 Bytes</td>
<td>1</td>
<td>8'388'608</td>
</tr>
<tr>
<td>4097 \times 4097</td>
<td>335'708'180 Bytes</td>
<td>1</td>
<td>33'554'432</td>
</tr>
<tr>
<td>8193 \times 8193</td>
<td>1'342'504'980 Bytes</td>
<td>2</td>
<td>134'217'728</td>
</tr>
<tr>
<td>16385 \times 16385</td>
<td>5'369'364'500 Bytes</td>
<td>6</td>
<td>536'870'912</td>
</tr>
</tbody>
</table>

**Table 5.1:** Test data

The simulation used $\approx 43000$ agents. Due to problems with the libagentcom library not all agents from the agent player are updated by the agent viewer (see 5.1.7).

The resolution of a single texture patch in the texture tree is $512 \times 512$ pixel. The texture tree has a depth of $n - 2$ for a terrain with $(2^n + 1) \times (2^n + 1)$ vertices. A satellite image [19] with a resolution of 500 m per pixel was used as an example ground texture.

All measurements use view frustum culling for the refinement and the texture tree unless stated otherwise.

### 5.1.3 Measured values

This abbreviations were used for the table headers:

- *size*  terrain size in vertices
- *\tau*   tolerance of the refinement algorithm
- *fps*    frames per second, i.e. how many images are rendered on screen per second.
- *app*    time used by the application, i.e. updates that occur every frame (e.g. terrain refinement)
- *cull*   time used for culling the scene graph, i.e. the decisions whether something is visible or not; invisible geometry is not rendered
- *draw*   time used for rendering the scene graph
5 Results

<table>
<thead>
<tr>
<th>term</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>tristrip</td>
<td>number of triangle strip indices returned by the refinement algorithm</td>
</tr>
<tr>
<td>patches</td>
<td>number of rendered texture patches</td>
</tr>
<tr>
<td>total</td>
<td>total time for creating the new geometry</td>
</tr>
<tr>
<td>refine</td>
<td>time used refinement algorithm including disk access</td>
</tr>
<tr>
<td>preread</td>
<td>time used for transforming the triangle strip to geometry including disk access</td>
</tr>
<tr>
<td>lighting</td>
<td>calculation of the vertex normals for lighting</td>
</tr>
<tr>
<td>makesstrips</td>
<td>subdivision of the triangle strip according to the texture quadtrees and calculation of the texture coordinates</td>
</tr>
</tbody>
</table>

All durations are measured in milliseconds.

Most values come from a single measurement. Only the loading times for partially cached terrain are averaged over ten runs. The reason for single measurements is that some values are only shown inside the viewer as part of OSG. Another reason is that the benchmarking is not automated and required user presence.

The application has an internal time measurement for total, refine, preread, lighting and makesstrips. The values for framerate, app, cull and draw are calculated by OSG and shown inside the viewer window. The initialization time, e.g. reading the road network from disk, is not relevant. It is small compared to the loading time and only occurs at the start of the program.

5.1.4 Preprocessing

<table>
<thead>
<tr>
<th>sidelength</th>
<th>vertices</th>
<th>preprocessing time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1025</td>
<td>1'050'625</td>
<td>12</td>
</tr>
<tr>
<td>2049</td>
<td>4'198'401</td>
<td>37</td>
</tr>
<tr>
<td>4097</td>
<td>16'785'409</td>
<td>561 (9 min 21 sec)</td>
</tr>
<tr>
<td>8193</td>
<td>67'125'249</td>
<td>3004 (50 min 4 sec)</td>
</tr>
<tr>
<td>16385</td>
<td>268'468'225</td>
<td>14617 (4 h 3 min 37 sec)</td>
</tr>
</tbody>
</table>

Table 5.2: Preprocessing times (in seconds) for different terrain sizes.

The preprocessing time includes the creation of the empty files, adding all raw terrain patches and then preprocessing the whole terrain (see 2.4). It is roughly linear to the number of vertices (Table 5.2). There is some overhead from adding different numbers of terrain patches and if the file has to be split.


5.1 Benchmarks

<table>
<thead>
<tr>
<th>filesize (Bytes)</th>
<th>bins</th>
<th>preprocessing time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>83968020</td>
<td>1</td>
<td>37</td>
</tr>
<tr>
<td>80000000</td>
<td>2</td>
<td>45</td>
</tr>
<tr>
<td>45000000</td>
<td>2</td>
<td>43</td>
</tr>
<tr>
<td>20000000</td>
<td>5</td>
<td>42</td>
</tr>
<tr>
<td>10000000</td>
<td>9</td>
<td>45</td>
</tr>
</tbody>
</table>

**Table 5.3:** Preprocessing times for different file sizes on a $2049^2$ terrain with total 83'968'020 Byte.

Memory mapping limitations force one to split large files into multiple subfiles (see 2.3.3). Splitting the terrain data into several files results in about 20% longer preprocessing times (Table 5.3). This is because the files have to be memory mapped and unmapped a lot, sometimes after each vertex access (see 2.4.2). The preprocessing is done offline so the additional time does not matter. The run time refinement also uses memory mapping, so data access on multiple files will be a bit slower than on one single file.

5.1.5 Disk access and mesh construction

<table>
<thead>
<tr>
<th>$\tau$</th>
<th>tristrip</th>
<th>total</th>
<th>refine</th>
<th>preread</th>
<th>lighting</th>
<th>makestrips</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1</td>
<td>1899195</td>
<td>32327</td>
<td>15784</td>
<td>8216</td>
<td>4813</td>
<td>3223</td>
</tr>
<tr>
<td>0.5</td>
<td>534529</td>
<td>10670</td>
<td>8008</td>
<td>316</td>
<td>1300</td>
<td>941</td>
</tr>
<tr>
<td>1</td>
<td>274685</td>
<td>4451</td>
<td>3074</td>
<td>181</td>
<td>650</td>
<td>480</td>
</tr>
<tr>
<td>2</td>
<td>124843</td>
<td>3939</td>
<td>3276</td>
<td>117</td>
<td>287</td>
<td>319</td>
</tr>
<tr>
<td>3</td>
<td>70511</td>
<td>5004</td>
<td>4600</td>
<td>87</td>
<td>156</td>
<td>128</td>
</tr>
<tr>
<td>4</td>
<td>43465</td>
<td>4188</td>
<td>3923</td>
<td>80</td>
<td>95</td>
<td>80</td>
</tr>
<tr>
<td>5</td>
<td>29883</td>
<td>2216</td>
<td>2014</td>
<td>63</td>
<td>66</td>
<td>58</td>
</tr>
<tr>
<td>6</td>
<td>21545</td>
<td>3187</td>
<td>3028</td>
<td>63</td>
<td>45</td>
<td>43</td>
</tr>
<tr>
<td>7</td>
<td>15755</td>
<td>2840</td>
<td>2707</td>
<td>60</td>
<td>32</td>
<td>34</td>
</tr>
<tr>
<td>8</td>
<td>11601</td>
<td>2512</td>
<td>2399</td>
<td>57</td>
<td>23</td>
<td>21</td>
</tr>
<tr>
<td>9</td>
<td>8951</td>
<td>1818</td>
<td>1714</td>
<td>45</td>
<td>25</td>
<td>16</td>
</tr>
<tr>
<td>10</td>
<td>7321</td>
<td>452</td>
<td>358</td>
<td>43</td>
<td>14</td>
<td>14</td>
</tr>
</tbody>
</table>

**Table 5.4:** Loading times for uncached terrain with different $\tau$ and fixed viewpoint. Terrain size: $16385^2$ vertices.

In an interactive visualization, the data has to be displayed on screen after the shortest possible time. The user should have to wait as little as possible until the next view is rendered.
The measurements distinguish between the first run of a terrain, where the data has never been read before and the following runs where at least part of the data is still in the cache. Small triangle strips are cached entirely. Larger triangle strips don’t fit entirely into the cache but are still partially cached during additional runs of the same terrain. The uncached data takes obviously much longer to load.

To flush the cache, a large terrain that was not part of the test data was run at a low tolerance $\tau$. This fills the cache with independent vertex data and is done before measuring the uncached behaviour for each terrain. The measurements for partially cached data are averaged over ten consecutive program runs after the terrain has been loaded the first time. A run consists of starting the program and measuring the loading time ($total$, $refine$, $preread$, $lighting$, $makestrips$) for the first terrain refinement until the terrain is rendered on screen. This is done for a fixed
Table 5.5: Loading times for partially cached terrain with different $\tau$ and fixed viewpoint. Terrain size: $16385^2$ vertices.

Table 5.6: Loading times for uncached terrain with different terrain sizes and fixed viewpoint.

Table 5.7: Loading times for partially cached terrain with different terrain sizes and fixed viewpoint.

Results on a $16385^2$ vertices terrain for uncached data are in Table 5.4 for cached data in Table 5.5. A graphical representation can be found in Figure 5.3. The uncached performance shows a high dependence on the memory performance and
5 Results

a smaller influence by the mesh size. The partially cached performance can be approximately obtained by a horizontal shift from the *tristrip* curve, which corresponds to a constant factor because of the logarithmic plot. This indicates that the performance depends directly on the mesh complexity and not on the memory performance. For small $\tau$ performance drops because the uncached part of the data becomes larger.

Results for different terrain sizes are in Tables 5.6 and 5.7. The loading time can be reduced by using a smaller overall terrain, if one is interested only in a small area. Then there is less overhead from the terrain that lies outside the viewpoint.

The loading times result in a choppy behaviour during interaction. The user has to wait until the refinement and mesh construction is complete. By choosing the tolerance $\tau$ a compromise between waiting time and visual quality has to be found. See Chapter 6 for possible optimizations.

5.1.6 Rendering to texture

![Framerates with and without rendering to texture](image)

**Figure 5.4:** Framerate with and without rendering to texture.

The rendering of the network and the agents to the texture patches uses a lot more time than rendering the triangle strips alone. A comparison between terrain rendering with and without rendering to textures is given in Table 5.8 and Figures 5.4, 5.5. Without rendering to textures the framerate depends on the mesh complexity. For very small $\tau$ there is an additional overhead in the draw phase. When rendering to texture the framerate stays roughly constant for the same number of texture patches, but is lower than rendering the mesh alone. Only very large mesh sizes
5.1 Benchmarks

**Figure 5.5:** Cull and draw times with and without rendering to texture.

<table>
<thead>
<tr>
<th>$\tau$</th>
<th>tristrip fps</th>
<th>cull</th>
<th>draw fps</th>
<th>cull</th>
<th>draw</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1</td>
<td>1899195</td>
<td>35</td>
<td>0.2</td>
<td>30</td>
<td>420</td>
</tr>
<tr>
<td>0.5</td>
<td>534529</td>
<td>0.2</td>
<td>19</td>
<td>30</td>
<td>66</td>
</tr>
<tr>
<td>1</td>
<td>274685</td>
<td>0.2</td>
<td>10</td>
<td>30</td>
<td>49</td>
</tr>
<tr>
<td>2</td>
<td>124843</td>
<td>0.2</td>
<td>4</td>
<td>29</td>
<td>45</td>
</tr>
<tr>
<td>3</td>
<td>70511</td>
<td>0.2</td>
<td>2</td>
<td>30</td>
<td>43</td>
</tr>
<tr>
<td>4</td>
<td>43465</td>
<td>0.2</td>
<td>1.3</td>
<td>30</td>
<td>42</td>
</tr>
<tr>
<td>5</td>
<td>29883</td>
<td>0.2</td>
<td>1.0</td>
<td>30</td>
<td>42</td>
</tr>
<tr>
<td>6</td>
<td>21545</td>
<td>0.2</td>
<td>0.7</td>
<td>30</td>
<td>42</td>
</tr>
<tr>
<td>7</td>
<td>15755</td>
<td>0.2</td>
<td>0.6</td>
<td>30</td>
<td>42</td>
</tr>
<tr>
<td>8</td>
<td>11601</td>
<td>0.2</td>
<td>0.5</td>
<td>30</td>
<td>42</td>
</tr>
<tr>
<td>9</td>
<td>8951</td>
<td>0.2</td>
<td>0.4</td>
<td>30</td>
<td>42</td>
</tr>
<tr>
<td>10</td>
<td>7321</td>
<td>0.2</td>
<td>0.4</td>
<td>30</td>
<td>42</td>
</tr>
</tbody>
</table>

**Table 5.8:** Comparison between using and not using rendering to texture on a $16385^2$ vertices terrain with 15 visible texture patches.

lower the framerate.

There are some reasons for the relatively low performance when rendering the textures. One is the slow culling of the network geometry. Some performance can be won by arranging the geometry in a grid of $n \times n$ bins. There is a tradeoff between cull and draw times. The more bins, the faster the drawing but the slower the culling (Tables 5.9, 5.10, Figure 5.6).
5 Results

\begin{table}
\centering
\begin{tabular}{llllll}
patches & 1 bin & 4 \times 4 bins & 8 \times 8 & 16 \times 16 & 64 \times 64 \\
\hline
1 & 69 & 67 & 60 & 61 & 75 \\
4 & 300 & 83 & 32 & 31 & 42 \\
16 & 1210 & 132 & 46 & 44 & 41 \\
34 & 2580 & 214 & 51 & 45 & 46 \\
52 & 3940 & 300 & 61 & 51 & 52 \\
\end{tabular}
\caption{Draw times for $1024^2$ random points divided into different number of bins.}
\end{table}

\begin{table}
\centering
\begin{tabular}{llllll}
patches & 1 bin & 4 \times 4 bins & 8 \times 8 & 16 \times 16 & 64 \times 64 \\
\hline
1 & 0.2 & 0.2 & 0.3 & 0.6 & 6.2 \\
4 & 0.4 & 0.4 & 0.5 & 1.0 & 12.7 \\
16 & 1.1 & 1.2 & 1.4 & 2.4 & 36 \\
32 & 2.3 & 2.3 & 2.7 & 4.4 & 70 \\
52 & 3.5 & 3.5 & 4.0 & 6.5 & 105 \\
\end{tabular}
\caption{Cull times for $1024^2$ random points divided into different number of bins.}
\end{table}

\begin{figure}
\centering
\includegraphics[width=0.5\textwidth]{figure5.6}
\caption{Cull and draw times of $1024^2$ random points for different numbers of bins.}
\end{figure}

Another reason might be the implementation OSG uses to render to texture. Instead of working only in the graphic card memory the geometry is rendered on screen and then read back to memory. This is slow because typical graphics hardware is pipelined. It is very fast in rendering geometry on the screen but very slow in reading parts of the screen back to memory. More efficient implementations (which could be built into OSG) would take advantage of techniques that render
textures directly to texture memory (e.g. pbuffers [4]).

5.1.7 Agents

<table>
<thead>
<tr>
<th>tstrip</th>
<th>patches</th>
<th>no agents</th>
<th>with agents</th>
</tr>
</thead>
<tbody>
<tr>
<td>70511</td>
<td>15</td>
<td>12</td>
<td>30</td>
</tr>
<tr>
<td>5737</td>
<td>20</td>
<td>1</td>
<td>40</td>
</tr>
<tr>
<td>157</td>
<td>1</td>
<td>37</td>
<td>4</td>
</tr>
</tbody>
</table>

Table 5.11: Rendering with and without agents. Terrain size $16385^2$ vertices, $\approx 43000$ agents.

The slow rendering of the agents (Table 5.11) has partially the same reasons as the rendering to texture (see 5.1.6). The current implementation accepts all agents regardless whether they are inside the terrain area or not. It also does not arrange them in bins which results in very long culling times.

The communication between the agent player and the agent viewer is not designed for a large number of agents. Often the viewer can only process a part of all messages sent by the player and thus only a part of all agents will be updated. This is a problem of the libagentcom library and not part of this work.

5.1.8 View frustum

Using view frustum culling for the refinement and the texture tree gives a huge performance boost. Not only is the refined mesh much smaller (Table 5.12) and thus the loading and rendering much faster but there is also only a small number of texture patches rendered. It is possible to render more detailed terrains that could not be rendered without view frustum culling (e.g. an Out of memory error results for $\tau = 1$).

The framerate without culling is very low ($< 0.1$ fps) because the texture tree is also not culled and all 52 texture patches have to be rendered. With frustum culling only 15 texture patches are visible and rendered at about 12 frames per second ($> 100$ times faster).
5 Results

<table>
<thead>
<tr>
<th>$\tau$</th>
<th>no culling</th>
<th>culling</th>
</tr>
</thead>
<tbody>
<tr>
<td>error</td>
<td>274685</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>tristrip</td>
<td>tristrip</td>
</tr>
<tr>
<td>2</td>
<td>777393</td>
<td>124843</td>
</tr>
<tr>
<td>3</td>
<td>455739</td>
<td>70511</td>
</tr>
<tr>
<td>4</td>
<td>325547</td>
<td>43465</td>
</tr>
<tr>
<td>5</td>
<td>256555</td>
<td>29883</td>
</tr>
<tr>
<td>6</td>
<td>206173</td>
<td>21545</td>
</tr>
<tr>
<td>7</td>
<td>167433</td>
<td>15755</td>
</tr>
<tr>
<td>8</td>
<td>134847</td>
<td>11601</td>
</tr>
<tr>
<td>9</td>
<td>104611</td>
<td>8951</td>
</tr>
<tr>
<td>10</td>
<td>79803</td>
<td>7321</td>
</tr>
</tbody>
</table>

Table 5.12: Comparison between using and not using view frustum culling on a 16385$^2$ vertices terrain.

![Zoomout loading times](image)

Figure 5.7: Loading times during zoomout.

5.1.9 Interaction

As a simulation of waiting times during interaction all loading times for a linear zoom out done manually from a closeup position to a global terrain view were measured. The terrain size was 16385$^2$ vertices. A graph of all loading times at different camera positions is shown in Figure 5.7. The corresponding triangle strip size is shown in Figure 5.8. The fluctuations in the loading times and the mesh size are dependent on the terrain. There are more triangles if the viewpoint is near a mountain range than if it is far away from a plain. Higher $\tau$ result in less triangles and mostly lower loading times.
Figure 5.8: Triangle strip sizes during zoomout.

The triangle strip sizes at $z \approx 170000$ are mainly so large because the borders of the dataset are not smooth and result in lots of triangles in the refinement even at large distances (see A.1).

5.1.10 Refinement in every frame

<table>
<thead>
<tr>
<th>refine</th>
<th>static</th>
</tr>
</thead>
<tbody>
<tr>
<td>tristrip</td>
<td>fps</td>
</tr>
<tr>
<td>70511</td>
<td>1.6</td>
</tr>
<tr>
<td>3247</td>
<td>8</td>
</tr>
</tbody>
</table>

Table 5.13: Refinement done every frame vs. static terrain.

To watch the same area for a longer time the framerate should be high to see fluid animations. So the terrain is only refined if the camera has moved beyond a threshold.

The framerate drops drastically if the refinement is done every frame (Table 5.13). The scenegraph has to be rebuilt from scratch each frame which results in a lot of time in the app phase of the scenegraph. This is because OSG is designed for static geometries and not optimized for dynamically rebuilt geometries.

5.1.11 Visual quality

A comparison between visual quality and mesh complexity can be seen in Figures 5.9, 5.10, 5.11. The higher the tolerance $\tau$ the coarser and smaller the triangle
strip. There is some border where more triangles don’t mean better visual quality, especially if there are very small triangles in the background. The image e.g. for $\tau = 1$ looks about the same as for $\tau = 0.1$ though the former has less than 15% of the strip size.

There is a tradeoff between visual quality, loading time and rendering speed when choosing $\tau$. A good standard value is $\tau = 3$. When using smaller datasets $\tau$ can be smaller, with better visual quality at about the same speed.
Figure 5.9: Visual quality and mesh detail for different tolerance values $\tau = 0.1, 0.5, 1, 2$ from top to bottom
Figure 5.10: \( \tau = 3, 4, 5, 6 \) from top to bottom
Figure 5.11: $\tau = 7, 8, 9, 10$ from top to bottom
6 Further Work

The current implementation of this work is a good starting point for additional work. There are also some optimizations that could be made. In \[A.2.4\] is described where to possibly add extensions in the code.

Due to the bad locality of linear indexing the paging is much slower than a more sophisticated layout scheme (see \[5.1.5\]). Better paging behaviour is achieved with Interleaved Quadtrees \[1\] and \[2\] or \(\Pi\)-order Indexing \[2\].

Multithreading can be used to allow user interaction during the refinement phase. The refinement would run in a parallel thread to the rendering. The user can then interact with the current mesh, while the new mesh is calculated in the background. When the new mesh is ready, it is rendered on screen and the old mesh is discarded.

To reduce the number of triangles in the refinement (see \[A.1\]), the borders of the raw terrain data can be smoothed. The outlying terrain height is then not set to \(z_{\text{outside}}\) but interpolated from the dataset borders. This would be done during the preprocessing.

As described in \[4.2.2\], additional terrain info like lakes, cities and forests can easily be added to the scenegraph that is rendered to textures.

The representation of tunnels and bridges is not solved. Currently they are rendered on top of the mountain resp. lake. An additional problem is that the network data has no data about road types.

Two way roads are rendered on top of each other. A method to make the road network more realistic has to be developed. TRANSIMS \[10\] has a sophisticated road representation where this type of road is divided into two parallel lanes.

With the triangle strips corresponding to the texture tree the triangle search is more efficient. Only the strip in a texture patch area has to be searched. This can be used to render agents that are close to the viewpoint as 3D models and not as
6 Further Work

a 2D texture.

The user interaction can be enhanced by adding a better navigation interface. Examples would be to zoom to the point where the user clicks or to have a list of interesting viewpoints. Another interesting thing to do would be following an agent as it moves through traffic.
Appendix

A.1 Known issues

There are visual artifacts when the viewer window is resized or is overlapped by other windows. This is due to the implementation OSG uses to render into textures. The geometry is first rendered on screen and then read back. If there are changes in the video memory, they will be propagated back to the texture.

When a mesh triangle spans several patches in the texture tree it is skipped (see 4.3). This results in a hole in the mesh and/or texture distortions.

The plane of the texture tree is only an approximation of the terrain. In some cases the visible patches of a texture tree for a viewpoint will not match the view frustum entirely. Some texture patches will then not be rendered.

Figure A.1: Example for abrupt patch borders resulting in a large number of triangles at the top.

The terrain dimension is constrained to $2^n + 1 \times 2^n + 1$ vertices. If the raw data patches do not match this constraint, the terrain area is enlarged in the preprocessing. The area outside the raw data is initialized with a dummy height $z_{\text{outside}}$. This causes large height differences at the patch borders. This areas then contain
a large number of triangles in the refined mesh (Figure A.1). Chapter 6 shows a possible solution to this problem.

When adding raw data patches in the preprocessing, the program sometimes reaches a state where the memory mapping fails. Strangely there is no error if the same patches are added in a different order. The causes for this problem are not clear.
A.2 Implementation details

This chapter gives an overview over the file structure and the contents of the classes. The usage of the preprocessor and the visualizer is explained in the second section. There is additional information about where to add possible extensions in the last section.

A.2.1 Overview

File structure

![File structure diagram]

Figure A.2: File structure

Schematic overview

Figure [A.3] shows a schematic overview of the interaction between the classes. This is thought as a reference when reading the code. More information on the program structure can be found in [4.5].

A.2.2 Class description

Here are very short class descriptions to help navigating the code. Most of the details should be taken from the code and the comments.

The code for all classes is in the file with the same name as the class name if not mentioned otherwise.
Appendix

Figure A.3: Overview of class interaction (identical to Figure 4.11)

**VertexInfo**

Container for vertex data \(x, y, z, \epsilon, r\)

**VertexMemoryMapper**

This class is used to write the preprocessed terrain files and to access their vertex data (see Chapter 2.3).

The file size is given by the number of vertices. Very large files are split into several smaller files of size \(MAX\_FILESIZE\). The data is indexed linear row by row.

When accessing a vertex the corresponding file is memory mapped with \texttt{mmap} and unmapped (\texttt{munmap}) when a vertex in a different file is accessed.
A.2 Implementation details

NOTE:
Always use `VertexInfo::get...()` and `VertexInfo::set...()` just after `VertexMemoryMapper::getVertex()` because when the next `VertexMemoryMapper::getVertex()` is called, the old vertex is not guaranteed to be still mapped (except if its in the same bin).

`arcGridASCIIReader`
Reader for raw ASCII terrain data by Duncan Cavens.

`TerrainPreprocessor`
Preprocessing of the terrain as described in 2.4

`OUT_HEIGHT` is the height $z_{outside}$ of the outlying area.

`agentFactory`
Interface for agent creation used with `libagentcom`

`dcAgentCom`
Communication with agent player by Duncan Cavens

`XMLFile`
In files `filesxml.*`

XML network file reader

This class uses a C `xml` library which leads to some problems when dynamically allocating `vectors`. The auxiliary classes `tmpNode` and `tmpLink` are used for temporary node and link arrays.

`FrustumPlanes`
Container for view frustum planes `left,right,top,bottom`
Appendix

MyAgentFactory

Implementation of agentFactory. Every time the viewer gets a message from the agent player a new Agent object is created or, if an agent with the same ID exists, its position updated. All agents are grouped under a osg::Group that is added to the scene graph (Figure A.4).

Agent

In files MyAgentFactory.*

Contains the agent’s position $(x, y)$, ID and the geometry. An agent is represented as a quad and a point, so it is still visible from far away when the quad alone would be culled. The geometry is organized under a osg::MatrixTransform (Figure A.4) so the position can easily be updated.

NetworkGeometry

![Diagram of NetworkGeometry]

**Figure A.4:** Scenegraph of the geometry that is rendered to texture.

$N$: osg::Node, $Mt$: osg::MatrixTransform, $Gd$: osg::Geode, $Gm$: osg::Geometry, $Gr$: osg::Group

This class creates the geometry which is later rendered to texture. The geometry is organised in layers with different $z$ coordinates (Figure A.4) (see 4.2.2). There is a layer for the satellite image which is simply a textured quad. On the second layer are the links. The geometry of a link is a quad and a line. The quad keeps the link at a certain width even at close distance. The line is always visible even when the quad is culled at large distances.
The nodes are represented geometrically as a points on a third layer. This layers contain static geometry. The agent geometry is on a dynamic fourth layer and handled by MyAgentFactory.

**Node**

In files NetworkGeometry.*

Container for node data id, x, y.

**Link**

In files NetworkGeometry.*

Container for link data id, from, to.

**TerrainRefiner**

This class contains the refinement algorithm of chapter 2.2.5.

**TerrainRenderer**

This class gets the triangle strip indices from the refiner and transforms them to a triangle strip geometry (see 4.3).

The lighting is done for the whole triangle strip by assigning a vertex the average normal vector over all normal vectors of its adjacent triangles.

The triangle strip is split into smaller strips according to the texture tree structure (see 4.3).
The texturing step has to map all vertices inside the square \( x_{\text{min}} \leq x \leq x_{\text{max}} \) and \( y_{\text{min}} \leq y \leq y_{\text{max}} \) to the texture coordinates \((u, v)\) with \(0.0 \leq u \leq 1.0\) and \(0.0 \leq v \leq 1.0\) (Figure A.5). This is done for all triangle strips and the corresponding texture patches.

At the program start the viewer is initialized in this class.

**MyCallback**

In file `TerrainRenderer.cpp`

This callback is called each frame and decides whether to make a refinement by comparing the current viewpoint with the last one. If the camera has moved enough the view frustum is calculated and a new refinement is started.

![Enlarging of the view frustum depending on the viewpoint height](image)

**Figure A.6:** Enlarging of the view frustum depending on the viewpoint height \(z_{\text{cam}}\) above the texture plane \(z_{\text{tex}}\) and the tolerance angle \(\alpha\)

The camera movement tolerance is taken into account by enlarging the real view frustum. It is rotated and shifted in the tolerance area around the viewpoint. The distance tolerance \(t = f \cdot (z_{\text{view}} - z_0)\) where \(f = \text{DISTANCE\_FACTOR}\) is a linear factor and \(z_0 = \text{DISTANCE\_HEIGHT}\) the ground plane. If \(t < 0\) it is set to zero or a minimal tolerance distance. The rotation tolerance is a rotation of \(\alpha = \arcsin(\text{VIEW\_ANGLE\_SINE})\) around the viewpoint. To enlarge the view frustum each plane of the original frustum is first rotated \(\alpha\) in the direction pointing outside the frustum. Then the rotated planes are shifted \(t\) along the plane normals (Figure A.6).

**TextureQuadtree**

Creates a quadtree according to camera position (see 4.2.1).
A.2 Implementation details

Figure A.7: Structure of the texture quadtree

Figure A.8: Intersection test between the patch bounding sphere $c$ and a frustum plane $p$

The visibility is decided with an intersection test between the bounding circle of a patch and the view frustum planes (Figure A.8). The distance $d$ between the patch middle point $M$ and the frustum plane $p$ is $d = M \cdot n_p$ with $n_p$ the unit length normal vector of $p$. If $d < -r \cdot \sin \alpha$ the bounding circle $c$ of the patch is completely outside the frustum plane. The negative value is taken because the positive direction of $n_p$ points inside the view frustum.

The distance metric for the refinement is calculated by multiplying the patch side-length with a constant factor. This is the maximal distance from the middle of the patch to the viewpoint where the patch will be refined. $\text{DISTANCE\_FACTOR\_SQUARED}$ is the square value of this factor. Higher values generate finer quadtrees.

TreeNode

In files TextureQuadtree.*

Nodes of the texture tree with information about dimensions and visibility.

TextureRenderer

This class manages the texture rendering.
**Appendix**

MyCullCallback

In files `TextureRenderer.*`

In this callback the actual rendering to texture in OSG is made.

**A.2.3 Usage and config files**

Both programs, the TerrainPreprocessor and the TerrainRenderer, let the user choose a number of settings that are given as parameters at program start.

**TerrainPreprocessor**

The preprocessor has two arguments:

`TerrainPreprocessor <configfile> <mode>`

The `configfile` has this content:

```
<preprocessed terrain output>
<max number of patches in x direction> <in y direction>
<lower left patch>
<first patch>
<remaining patches>
```

The preprocessed terrain data is written to `preprocessed terrain output`. If the terrain data is too big and has to be split, the files are named by appending an increasing number starting at 1 to the filename (`filename.out, filename1.out, filename2.out, ...`). The next line contains two integer values, the first is the maximum number of terrain patches that will be added in `x`-direction, the second the number in `y`-direction. To calculate the final dimensions of the terrain the `lower left patch` is parsed for minimum `x` and `y` coordinates, the terrain resolution and the width and height of the patch. This can be a dummy patch if the raw data does not cover...
the lower left corner. The following lines are a list of patches that are added. All patches need to have the same dimensions.

NOTE:
The last line in the config file must not have a linebreak. If this is the case, the program tries to read a patch with an empty filename and aborts. Do not use nedit to write the config file because it always adds a linebreak at the end.

The second parameter of *TerrainPreprocessor* is a mode flag. When set to c only the files are created and all patches added. A mode value of p does only calculate the vertex errors and radii of existing files. If an invalid or no mode is given, both file creation and error calculation are done by default. This feature has been added to allow the user to do only part of the preprocessing and then continue at a later time.

The preprocessor generates an output config file containing the path to the preprocessed terrain and its dimensions. This file can be used as a base for the *terrainconfig* file of the visualizer.

**TerrainRenderer**

The visualizer is called as follows:

*TerrainRenderer* <terrainconfig> <networkconfig> <error tolerance>

The first two arguments are config files, the error tolerance is a positive *float* value.

*terrainconfig* contains information about the terrain. It has this structure:

```
<terrain file>
<terrain sidelength in vertices> <subtree depth>
<texturetree depth>
<satellite image>
<image offset u> <image offset v> <image scale>
```

The first two lines can be copied from the *.*.raw file that is written by the preprocessor. A good value for the *texture tree depth* is 2 – 3 less than the *subtree depth*. The following values are the image file path *satellite image* for the ground texture and the offset and scaling values. These *float* values are found by manually aligning the texture to the terrain with trial and error.
Appendix

The second config file networkconfig is independent of the terrain and has data for the agent viewer and the network file. Optionally, an initial viewpoint can be set.

<hostname>
<hostport>
<nodes and links xml file>
<view x> <view y> <view z> <look at x> <look at y> <look at z>

hostname and hostport tell the agent viewer where to look for agent players. The next parameter is the path to the XML files containing the network data. An initial viewpoint can be chosen by defining a camera position \((x, y, z)\) and the point where it looks at. If the default OSG initial view (global view from the side) is wanted, camera viewpoint \(x\) has to be set to none.

The third parameter is the screen space error tolerance \(\tau\) used by the refinement algorithm. The default value is 3.

A.2.4 Extensions

Here are some remarks where future additions (see Chapter 6) possibly could be added in the code.

All access to the preprocessed terrain is encapsulated in the VertexMemoryMapper class. Changes to the data layout have to be made in this class only.

Triangle search for 3D positioning of objects can be simplified by searching the texture patch where the \((x, y)\) coordinates of the object lies in. The corresponding triangle strip has then to be made available for searching in the TerrainRenderer class.

Smoother patch borders can be calculated in the preprocessing. An additional smoothing phase could be added in the TerrainPreprocessor class.

Additional terrain info rendered as texture is simply added to the scene graph in the NetworkGeometry class.

When porting the program to other platforms, the mmap system call has to be exchanged with the equivalent system call of the new platform. This is the only platform dependent code in the implementation and used in the VertexMemoryMapper class.
Bibliography


Bibliography

[12] OpenGL,  
http://www.opengl.org/


[14] *Graphische Datenverarbeitung I* lecture at Computer Graphics Lab, ETH Zürich,  
http://graphics.ethz.ch/main.php?Menu=4&Submenu=1&Hornav=1&Course=gdv1

[15] Real-Time Rendering Resources,  
http://www.realtimerendering.com/

[16] Graphics articles and resources on GameDev.net,  
http://www.gamedev.net/reference/list.asp?categoryid=40

[17] Federal Office of Topography (SwissTopo),  
http://www.swisstopo.ch/

[18] Digital heightmodel of Switzerland,  

[19] Satellite images of Switzerland,  

All URLs last visited in march 2003