Design, Analysis, and Evaluation of a Data Structure for Distributed Virtual Environments

Dissertation von Matthias Fischer

Schriftliche Arbeit zur Erlangung des Grades eines Doktors der Naturwissenschaften

Heinz Nixdorf Institut und
Institut für Informatik
Fakultät für Elektrotechnik, Informatik und Mathematik
Universität Paderborn

Paderborn, im Februar 2005

Für meine Eltern,
meine Frau Hanne und
meine Kinder Miriam & Carolin

Herzlichen Dank
für ihre ideelle Unterstützung,
ihre Geduld, Entbehrungen und ihr Verständnis

Contents

1 Introduction to a System for Networked Virtual Environments

Three-dimensional computer graphics, a field of great importance in industrial applications and in scientific research, is a means for depicting and representing kinds of data. Typical applications are the architectural visualization of buildings and cities. The user of such systems wishes a realistic impression of designed buildings that do not exist yet. In industrial simulations of manufacturing plants, the computer graphics system is used for the visualization of simulation data. For example, the data derives from material-flow simulations or planning systems. In order to investigate the dynamics of fluids, computer graphics is used for the visualization of the flow of fluids. Applications in industrial construction and design use the computer graphics systems for the visualization of vehicles, clothes, tools, household appliances, and so forth (CAD systems). Further fields of applications are entertainment (games, film industry), marketing (product presentation), education, history, archaeology, and medicine.

The data of the applications can be solids (e.g., vehicles or clothes), physical parameters or quantities (e.g., pressure, temperature, and flow), and artistic works of art. There arise two different questions and problems which must be solved by the applications. The first problem, the *visualization problem* [228, 216], is the question of how to visualize the data, e.g., temperature or pressure. The goal is to search for methods to intelligibly and clearly arrange complex data. The second problem, the *rendering problem* [95], is the fast and efficient computation of one or more high quality images of the data. In this work, we develop efficient data structures and algorithms for the fast rendering of three-dimensional data, i.e., we focus on the rendering problem.

The first important step towards real-time rendering was the implementation of Catmuls z-buffer algorithm [35, 36, 37] in hardware. Consisting of simple and basic operations, it is to date the basic concept of today's real-time rendering graphics workstations. However, the growing demands for more complex and realistic models in some applications make the main drawback evident: the running time of the algorithm, $O(n \cdot a)$ [121], is linear in n and a where n denotes the number of triangles drawn and a the number of pixels (without consideration of occlusion).

In this work, we tackle the problem by a distribution of the data across a network of workstations and rendering only parts of a scene, namely the neighbourhood of a visitor. Accordingly, the running time of our rendering algorithms depends on the complexity of the neighbourhood of the visitor, but is independent of the size of the scene.

Due to the lack of practical alternatives, the effort to render complex scenes and to get a realistic impression results in two trends. On one hand, hardware producers optimize the technique so that the rendering pipeline is capable of rendering more and more polygons at the same time. After a phase of improvements on higher triangle and rastering performance, the manufacturers focus more on realistic effects than on higher geometry performance. With techniques such as textures, pixelshader, and reflection mapping, the realistic impression of the scene is improved. Today many applications use these new techniques instead of highly complex models in order to get a realistic image of the scene.

The second trend is based on the insight that the rendering pipeline can be disburdened by reducing the number of polygons sent to the graphics pipeline. Among other rendering techniques, two main streams of algorithm developments are established that work like a prefiltering step before the graphics pipeline. The approximation of the image (simplification and replacement of polygon meshes) is the one and the computation of invisible polygons (visibility culling) is the other one. The concept of *level of detail* (LOD) started this direction of research and it is an important one in the field of computer graphics, dating back to James Clark's 1976 seminal paper [55].

In the following, we sketch the main ideas and results of the architecture and data structure. Our main contributions are summarised as follows:

1. We propose a data structure for the management, rendering, navigation, and manipulation of a distributed virtual environment. This data structure supports
   - distributing the scene across a network of workstations.
   - a walkthrough of the scene, concurrently for many users.
   - a manipulation of the scene, namely insertion and deletion of objects, by many users at runtime.
   - a conceptually unbounded extension and size of the virtual environment.
2. A theoretical analysis of the data structure that proves that each of the operations (navigation, manipulation, rendering) can be executed in a time that only depends on the complexity of the neighbourhood of the user, and is independent of the total size of the scene.
3. A prototypical implementation of a walkthrough system for a distributed virtual

environment based on our data structure. The objects of the scene and the data structure are stored across a network of remote hard disks.

4. An experimental evaluation of this implementation confirms the theoretical results and gives further details about the properties of the system. The evaluation investigates how fast the user can manipulate and navigate through the scene.

Distributed Multi-User Walkthrough System

We present a walkthrough system for the real-time rendering of distributed virtual environments. The complexity of the scenes is scalable and can exceed the capacity of main memory and hard disk. The scene is distributed across a network of workstations each having a local hard disk. Only parts of the scene are stored in the main memory. We develop a system architecture, a data structure, and algorithms that enable a user to walk through a distributed scene in real time. The system avoids extensive upload times for the whole virtual scene. Each workstation stores only a small part of the data in its main memory, or on its hard disk.

The system is a multi-user and multi-system environment. Multi-user environment means that many users use the virtual scene for a collaborative work. They work independently on different parts of the scene. Typical work is insertion, deletion, and viewing of objects. Multi-system environment means that the scene can be manipulated by several graphic systems that are placed at distinct places and which may be incompatible with each other with respect to software and hardware. Our simple interface consists of the virtual objects and a data structure (graph) that connects the objects.

Our walkthrough system aims at scenes with a large spatial range such as landscape scenes consisting of cities, countries, and areas of vegetation. The scenes are 2.5-dimensional because the diameter of the scene is much larger than the height of the scene. We develop special featured operations and data structures that allow navigation and manipulation.

Abstraction of a Virtual Scene and Architecture

We introduce a simple theoretical model for modelling virtual scenes. In our underlying abstraction, the virtual scene is composed from simple *balls* (*objects*). The balls have unit size and must not overlap. We show that the abstract model, despite its simplicity, can be applied to typical virtual scenes consisting of houses, cars, and trees. On one hand, the model is a suitable means for the development of efficient algorithms, and otherwise it is suitable for practical use.

The virtual scene can be visited by *visitors* and *modellers*, each sitting at a graphics workstation. The visitor walks only through the scene and moves to arbitrary positions. The modeller has additional capabilities; he inserts objects into and deletes objects from the scene. Our architecture enables many viewers and modellers to view and to model the scene simultaneously, but at the same time independently of each other.

Both, the viewer and the modeller see the objects in a sufficiently large neighbourhood. More precisely, they see all objects inside the circle with centre x and radius t if x is the position of the viewer or modeller. We denote the circle and all objects inside the circle by *bubble*. The bubble must implement some properties that we denote by *bubble requirement*. The bubble requirements feature an easy duplication of parts of the scene. For example, remotely stored parts of the scene can be copied easily to a local hard disk, and from the hard disk, parts of the scene can be copied into the main memory. The bubble requirements feature an easy combination of different parts of the scene, e.g., city models can be combined easily to a whole country.

The walkthrough system must enable the movement of the bubbles, and the insertion and deletion of objects. We use the four operations $\textsc{Search}(x, t)$, $\textsc{Move}(x, \Delta x)$, $\textsc{Insert}(x, o)$, and $\textsc{Delete}(x)$. The $\textsc{Search}(x, t)$ operation computes all objects in the bubble starting from one object in the bubble. The centre of the bubble is always the position of viewer and modeller. The $\textsc{Move}(x, \Delta x)$ operation moves the bubble through the scene if the viewer and the modeller move. The modeller uses the operations $\textsc{Insert}(x, o)$ and $\textsc{Delete}(x)$ to insert and delete an object o at his current position x.

Locality of Data

An important requirement for our architecture is the *locality of data*. More precisely, the spatial locality in the scene must correspond to the locality of the data. We use a non-hierarchical, two-dimensional data structure that features the locality of data. Hierarchical data structures, which are mainly used in computer graphics, do not feature the locality of data. Spatial locality means that two modellers can insert and delete objects at distant locations concurrently, but independently of each other. Locality of the data means that a modification of the data by one modeller does not affect the data of the other modeller and vice versa.

We give an example: two modellers, each sitting at a graphics workstation, modifies a model of the landscape of Germany independently of each other. The one inserts and deletes objects in Paderborn and the other one modifies Munich. We require a separation of the data structure: a part M of the data structure manages the objects of Munich and a part P manages the objects of Paderborn. The modification of the scene involves a modification of the data structure. The data structure supports the locality of the data if the manipulation of the modeller in Munich only involves updates of the part M and the manipulation of Paderborn only involves updates of part P. Not only updates of the data structure are locally restricted, but also references to or knowledge about far away parts of the scene are limited.

What is the advantage of such a data structure? The unbounded spatial size of the scene implies a growing number of visitors and modellers. This in turn implies that the users employing different systems may be standing at different places. Heterogeneous systems prefer an interface that is as small as possible. Our data structure supports an interface consisting only of links (edges) that point from one object to its neighbour. We must neither refer to nor change the data of Munich if we add new objects to Hamburg and Aachen. Moreover, the system used by a modeller, who renders and manipulates data of Munich, can be completely incompatible with a system of a modeller who modifies Paderborn. Spatially separated parts of the scene can be added to a single scene by simple paste operations such as composing the parts of a jigsaw puzzle. Therefore, the locality of data makes the system scalable.

Weak Spanner and Sectorgraph

We introduce a new class of graph, the *weak spanner* [91], this is a special variant of a spanner. A graph G of a two-dimensional point set is a f-weak spanner if for each node u, v of G there is a path from u to v in G so that for each point s on the path $dist(u, s) \leq f \cdot dist(u, v)$ holds. We propose a new non-hierarchical, flat data structure, namely the *sectorgraph*, based on a well known graph construction for spanners [241, 135, 180]. The sectorgraph is defined as follows: for each point u draw k rays (line segments) from u such that these rays subdivide $\mathbb{R}^2$ into k equal cones (*sectors*) around u. For each u and each sector s, u has a directed edge to the closest object from u lying in the sector s.

We show that the sectorgraph is a weak spanner [94] and use it in our walkthrough system to manage the objects of the scene. We prove that the running time of our algorithms for the navigation and manipulation in the scene only depends on the number of objects in the neighbourhood of the visitor, which is independent of the size of the scene. The sectorgraph fulfils the bubble requirements so that the graph is usable for a circular range query, the movement of bubbles, and the insertion and deletion of the objects.

The reason for our choice of data structure is the property that the weak spanner combines the *spatial locality* with the *locality of the data*. The weak spanner features the property because all edges of an object point only to some few objects of the neighbourhood. Changes in the data of far away parts of the scene do not involve updates of the data of nearby parts of the scene.

In order to reflect the locality in the runtime analysis, we introduce the notion of the *neighbourhood sensitive* running time. As an output-sensitive algorithm, the running time depends on the output of the algorithm. In our case, this is all objects of a bubble. Typically, our algorithms must search the neighbourhood of the bubble, too. This implies costs that depend on the number of objects in the neighbourhood of the bubble. Consequently, the running time is good if the neighbourhood is small, and the running time is worse if the neighbourhood is larger than the bubble. The cost measure reflects the spatial locality of the algorithmic operations well. We show that our algorithms are neighbourhood sensitive.

Implementation and Evaluation

We implement the algorithms into a prototypically walkthrough system. The system makes the manipulation and movement through a distributed virtual scene possible. Our system stores the objects and the sectorgraph in the main memory, on a local hard disk, and across the network on remote hard disks (*storage types*). For each storage type, we create bubbles that contain duplicated objects of the scene. If we slot the bubbles of different storage types into each other, we build a *spatial hierarchy of caches.* Depending on the access time of the storage type, we can move the bubbles with varying speed. We use the system for the evaluation of our methods and show their practical use. For this, we investigate the running time for the navigation and manipulation in the virtual scene. We measure how fast a user can move the bubbles of different storage types through the scene.

Overview

We survey the state of the art in Chapter 2. In Chapter 3, we present our abstraction of a virtual scene and the architecture of the system. In Chapter 4, we describe the algorithms for the management and manipulation of the scene based on the sectorgraph. The implementation and evaluation follows in Chapter 5.

2 Background and State of the Art

In the following Subsections, we introduce acceleration methods and data structures for walkthrough systems of complex models. The work on parallel rendering and networked virtual environments (Section 2.2) is closely related to our approach because our walkthrough system renders a scene that is distributed across a network of workstations. Most of these work is a result of computer graphics. Because our data structure is a spanner is used for range searching, the work on range searching, spatial data structures, and geometric spanner is also closely related to our approach (Section 2.3). Section 2.1 gives a wide overview of the fields and general approaches that improve the rendering time. Prior to that, we informally describe the ability and features of a walkthrough system, and we explain demands, problems, and bottlenecks.

Input to a Graphics System

Typical 3D data derives from a CAD system, simulator, medical apparatus, and 3D scanner [27]. The exchange of 3D data between different applications (e.g., CAD and walkthrough system) is complicated because applications use their own proprietary file format. The format must be converted in a suitable *3D model* that can be processed by a graphics system. Widely used formats of 3D models are established by 3D rendering libraries (e.g., OpenInventor [233]), application interfaces of the World Wide Web (e.g., VRML [34, 203]), as well as by 3D modelling programs for the construction of 3D models (e.g., 3DStudioMax [26, 78]). The handling and conversion of file formats has up to now been a time intensive work process and an insufficiently solved problem in computer graphics. The main reasons are buggy transformation programs and incompatible features of the formats. A curious transformation A to C to D to B may be better than directly A to B.

Two common techniques for modelling three dimensional data are surface modelling and solid modelling [95, 119, 229]. Solid modelling allows a distinction between the inside, outside, and the surface of a model. These properties are necessary in many applications, e.g., CAD/CAM. Today's commercial computer graphics systems mainly use surface models. They model the surface of the objects by means of a mathematical description, e.g., polygon mesh surfaces, parametric surfaces, and quadratic surfaces. A wall can be modelled as a single equilateral (one-sided wall) or as flat cuboids (two-sided wall). Rounded objects need many flat surface elements to be modelled realistically.

Real-Time Versus Offline Rendering

A traditional classification of rendering methods distinguishes, between *real-time rendering* and *offline rendering*. Offline rendering defines the motion of the camera and computes an image for each position of the camera path in a preprocessing step. Afterwards, all images are put together into a movie that shows the scene along the defined path. After completion, a modification of the orientation and the position of the camera is impossible. Classical offline rendering methods are *radiosity* [204, 61] and *ray tracing* [201, 103]. The computation of each image needs a few minutes up to hours.

Real-time rendering computes the image during the walkthrough such that the viewer can move the camera to arbitrary positions of the scene. The images must be recomputed if the viewer moves to the same position several times. To get smooth rendering of the images, fast real-time rendering methods are necessary. The rendering of a single image consists of the three basic steps transformation, illumination, and hidden surface removal: the transformation computations transform the three-dimensional scene space to the two-dimensional screen space of the viewing device [95]. The second step, the illumination computation, computes a realistic appearance of the scene. Fast local illumination methods such as Goraud Shading [105, 106] and Phong Shading [171, 172] are used in walkthrough systems. In contrast to global illumination methods (radiosity, ray-tracing), a loss of image quality is the consequence because shadows and reflections are missing. However, the newest generation of graphics hardware reduces this lack of image quality using techniques as textures, pixelshaders, and reflection mapping [165]. The third rendering step is the removal of the hidden surfaces. Hidden surface removal is a time consuming step and it is one of the classical rendering problems [82].

Rendering in real time has been available since the successful hardware implementation of the *z-buffer algorithm*. The algorithm goes back to Catmull's famous dissertation [35, 36, 37]. The algorithm removes all hidden surfaces by the transformation and rasterization of surfaces. For each pixel of the rastered surface, the distance to the viewer (z-value) is also stored to decide if other surfaces are in front or behind of the surface. The running time of the algorithm grows linearly $O(n + a)$ with the number of surface polygons n, and the number of pixels a that are used for the rendering of a surface (independent of occlusion) [121]. The simple hardware implementation of basic algorithmic operations makes this algorithm successful. Today, all three basic rendering steps are hardware supported. They are implemented in a kind of pipeline that processes the surface elements, the polygons, one after another.

Walkthrough Systems and Their Requirements

A walkthrough system of virtual scenes allows the arbitrary modification of the position and orientation of the camera by the mouse and other additional devices, so that the user is able to move to arbitrary positions. The system has to compute a new image for every new camera position; each rendered image is denoted as *frame*. A system is denoted as *walkthrough system* if it allows this kind of navigation.

The ability to visit things that do not exist in the real world makes the technology so interesting. The practical value of a walkthrough system is the view from, and the move to "things" that we would otherwise never experience without the help of the system. The new stadium in Atlanta for the 1996 Centennial Olympic Games was far from completion as the opening approached. There was no time to interrupt the construction for visits and tours by international officials and local planners. A walkthrough system of the whole stadium, implemented by IBM's 3-D Interaction Accelerator (3DIX) [131], enabled international officials and local planners to simulate a walkthrough tour of the stadium before, during, and after the construction [125]. The walkthrough system was used in classrooms to help to train the employees and the volunteers who worked later at the Olympic venues; the security people planned access points, surveillance, and tactics. The stadium was built parallel to the training and schooling of the service teams.

The question remains what are the essential demands on a walkthrough system? The example shows that we cannot assume a technical background of the users. This implies that the user does not suffer a poor realistic impression of the system. In consequence, the walkthrough system must fulfil two requirements, first high quality modelling, lighting, and shading of the scene, and second guaranteed real-time rendering. *Real-time rendering* means that the system computes a new image with at least ten *frames per second (fps)* if the user moves to a new position. To avoid jerky movements of the camera at least 20-30 fps are necessary, i.e., 5ms for the computation of one image. Computer games try to get 30-60 fps, and first-person shooters need more than 60 fps because of the fast movement of the actors in the game. More problematical than jerky image generation is the *navigation* of the user. For middling navigation, at least 10 fps are necessary. Frame rates below ten prevent good navigation because the position of the rendered image and the position of the input device differ. The user permanently tries to correct the difference, but the walkthrough-system cannot follow and so the difference gets larger and larger. The result is wide jumps of the camera and the user cannot reach the desired position. Therefore, high frame rates have the highest priority compared with other demands like good image quality. Otherwise, navigation is impossible and the system is unserviceable.

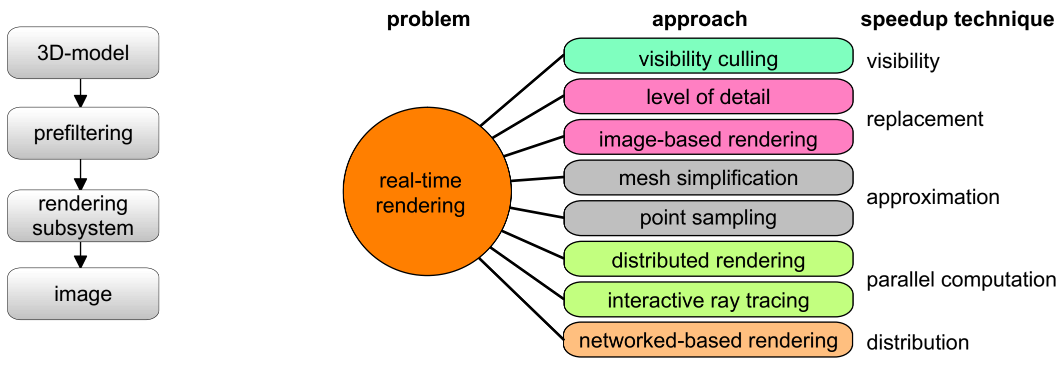


Figure 2.1: Some prefiltering approaches and their underlying mechanism.

Bottlenecks and Acceleration Methods

Many applications need visualization of scenes with millions of polygons. The real-time rendering of small scenes up to several hundred thousand polygons is possible with today's graphics hardware. The *fill rate* (6.4 billion texels/sec), the *memory bandwidth* (35.2 GB/sec), and *geometry transformation* (600 million vertices per second) are relevant parameters that limit the complexity of the scene (performance specification NVIDIA GeForce 6800 Ultra [164]). If the scene complexity exceeds more than a million polygons, the *large scene* cannot be rendered in real-time because of the linear running time of the z-buffer algorithm [121]. Thus, the graphics pipeline is one of the bottlenecks of the system. A further bottleneck occurs in front of the graphics pipeline. The slow memory access and the limited transfer rate of the bus system prevent a fast transfer from the CPU/memory unit to the graphics unit and prevent the graphics pipeline from working at full capacity. In addition, the limited memory capacity is a bottleneck. Large scenes must be modelled by instantiation in order to fit into the main memory. Otherwise, they have to be stored on hard disks and out of score rendering techniques that slow down the frame rate are necessary.

We need sophisticated algorithms in order to achieve high frame rates. Some *acceleration methods* start from the situation that the end of the rendering process is made up of the graphics pipeline, which is based on the conventional z-buffer algorithm. The algorithms work like a *prefiltering step* between the model and the rendering subsystem (see Fig. 2.1, left). The acceleration methods reduce the number of polygons that they send to the graphics hardware. Some basic approaches have been established. *Visibility culling* algorithms [82] compute a subset of the invisible polygons and avoid their rendering by the graphics hardware. *Level of detail*, *surface simplification*, and *multiresolution modelling* [153] are approximation methods that achieve the acceleration by rendering less complex models, whereas the simplified models should look like the highly complex original. *Textures* [120] or *image based rendering* [202] substitute large parts of the scene with fixed images. Some methods must exempt from the "pre-filtering" classification, e.g., interactive ray tracing [223] or some point sampling work do not work on top of a z-buffer architecture, because they solve the hidden surface removal problem without the help of the z-buffer algorithm. Figure 2.1, right, shows some approaches and their underlying mechanism.

Our work is most related to networked-based rendering and distributed rendering. We do not use level-of-detail, approximation, replacement, and point sampling techniques because we bound the scene complexity by rendering only the objects lying within a short fixed distance. However, a combination with our system is useful and possible. With the help of approximation methods, we render more objects in the neighbourhood of the viewer. In the same way, integration of visibility culling is possible resulting in the rendering of fewer objects. However, the goal of our system is to show the usefulness of a new data structure for the management of distributed scenes. Therefore, we neglect other kinds of speed up techniques. A good introduction to walkthrough systems that integrates several acceleration techniques is given by the MMR system of Aliaga et al. [10, 11], and the book by Möller and Haines [159]. Although a combination of different techniques is worthwhile, not all algorithms can be combined in the same system. In addition to the time and memory efficiency of the algorithms, the capability of the integration is an important feature of the rendering algorithms.

2.1 Classification of Methods for Scene Complexity Reduction

2.1.1 Level of Detail Concepts

The concept *level of detail* (LOD) is an important one in the field of computer graphics, dating back to Clark's seminal paper [55]. Clark recognizes the redundancy of using many polygons to render an object covering only a few pixels. He described a hierarchical scene graph structure that incorporated not only level of detail, but also other now common techniques, such as view-frustum culling. The paper discusses advantages of a hierarchical scene organization for computer graphics.

The Hierarchy

The hierarchical scene organization makes it possible to render a model in different levels of detail. The smiley in Fig. 2.2 consists of three LOD models, each consisting of different resolutions and quality. High quality models are exchanged by low quality

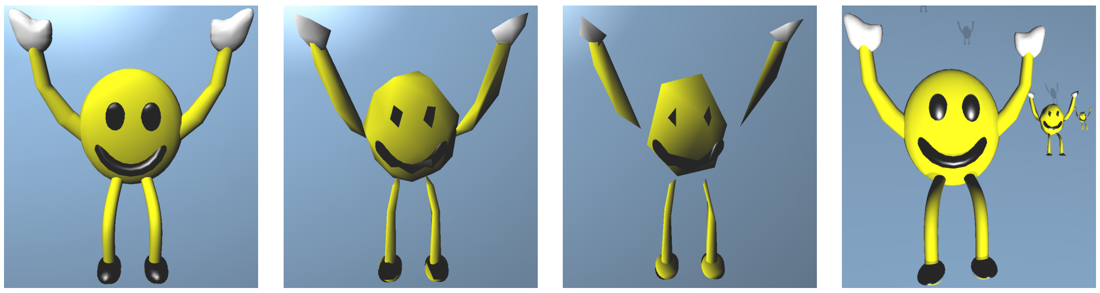


Figure 2.2: High quality models for short distances, low LOD models for far distances. From left to right: LOD1 (5976 triangles, 3135 vertices), LOD 2 (366 triangles, 308 vertices), LOD 3 (264 triangles, 257 vertices), and all of them.

models if the user moves farther away from the model (see Fig. 2.3, left). The rendering of the low quality models is faster than the rendering of high quality models because the low quality models need fewer polygons. The visual perception of distant low quality models is only slightly disturbed (see all LOD models at distinct distances in Fig. 2.2 right).

How does a data structure that supports this concept look? Clark suggests a hierarchical organization of the scene modelled by an acyclic graph. The root node of the scene represents the whole scene (see Fig. 2.3, right). Children of inner nodes represent refined definitions of upper nodes, e.g., a coarse model of a smiley (level 1). The arcs in the tree represent either transformations, necessary for the placement of objects in the world, or references (links) to the children that contain a more refined model representation. The three children of the smiley node are a refined model representation for head, legs, and arms (level 2). The three children of the head node contain further

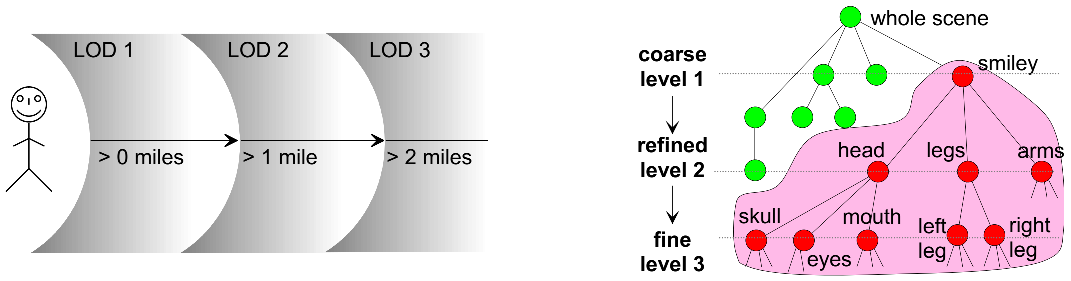


Figure 2.3: Different LOD's are chosen for different distances with respect to the viewer (left). LOD hierarchy (right): vertices at higher levels contain coarse models and vertices at lower levels contain refined models.

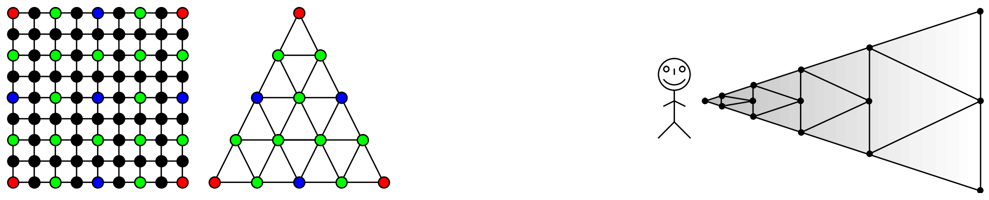


Figure 2.4: Continuous LOD (left): one single model contains different resolutions. The coarse level use only the red vertices, refined levels additionally use the blue (green, black) vertices. View-dependent LOD (right): near to the viewer, the object is rendered with small polygons and far from the viewer with large polygons.

refined models of the head, both eyes, and mouth (level 3). Each of the three levels contains a full model representation of different resolutions.

Level of Detail Frameworks

Clark's hierarchical LOD idea has been refined because of problems with the fixed granularity. The following classification into three frameworks is used in the literature [153].

Discrete LOD: The original scheme, denoted by *discrete level of detail*, creates multiple versions of every object during an off-line computation in a preprocessing step. At runtime, the appropriate LOD model is chosen to render the object.

Continuous LOD: The problem of the discrete LOD is the granularity of the objects; usually a fixed number of models (2-5) is used for switching the objects. Switching a model implies disturbing popping effects. *Continuous level of detail* solves these problems by encoding a continuous spectrum of detail. Many different resolutions of the object are encoded in one single representation; the desired LOD is extracted at runtime. The better granularity leads to a better balancing of the number of polygons for each object. The quadratic mesh of Fig. 2.4, left, shows four levels of detail in one single representation. The coarse mesh contains the four red vertices (a single square). The first refined level uses additionally the five blue vertices (four squares), the second refined level uses the green vertices (16 squares), and the finest level uses all vertices (64 squares). Other geometric subdivisions are possible; see the triangle mesh of Figure 2.4, left.

View-Dependent LOD: *View-dependent level of detail* extends continuous LOD using a view-dependent simplification to dynamically select the most appropriate level for the current viewpoint of the viewer. Nearby portions of the object are shown at higher resolution than distant regions. A single object spans multiple levels of details for one single view. View-dependent LOD is useful for objects with large extent, e.g. trains,

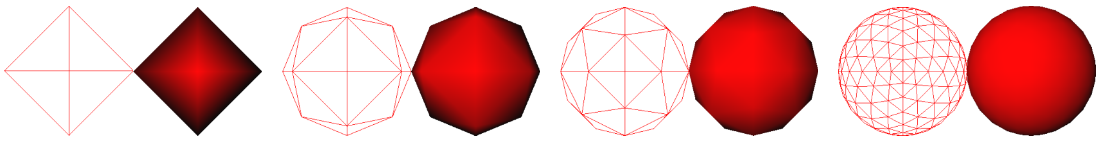


Figure 2.5: Spheres with distinct resolutions (wire frame and shaded). From left to right: 8, 32, 72, and 521 triangles.

aircraft, and terrain. If the viewer is standing at the front of the train, we render the head of the train with a high LOD and the back of the train with a low LOD. The viewer of Fig. 2.4 looks from left onto the triangle mesh. Near the viewer, we render many small triangles, and far away, we render the triangle mesh with a few large triangles.

Although today only cited in the context of systems that use simplification and replacement methods, Clark's paper set the cornerstone for many visibility culling techniques, too. This spatial scene organization is basically used for many approximation and culling algorithms. Shade et. al [199] used LOD based hierarchies to cache rendered images and reuse the images in subsequent frames. Funkhouser and Séquin [98] used an LOD hierarchy in conjunction with Potentially Visible Sets (PVS). Maciel and Shirley [156] used an LOD hierarchy of meshes and textures to reduce the complexity of the scene. Chamberlain et al. [40] used an LOD hierarchy consisting only of coloured boxes, where the colour of the box is computed by an average of the colours in the box.

Looking back from today's point of view, the idea of building up a hierarchy is simple, but its fundamental character has influenced the development of new algorithms and systems for real-time rendering until today. Therefore, Clark's paper was selected for a book on seminal publications [56].

2.1.2 Polygonal Surface Simplification

Highly complex polygonal 3D models occur in technical applications. 3D scanners generate 3D polygon models from an existing object in our real environment. The scanning devices use a high resolution to get a smooth surface for plumb objects. CAD programs use abstract mathematical high-level description of the objects. To make the models readable for a walkthrough-system, they have to be converted into complex polygonal models. The high resolution is unnecessary for objects with large flat surfaces and causes slow rendering times, high memory consumption, and long network transmission times. *Simplification algorithms* substitute the many polygons in flat regions for only few polygons. The input is a mesh with n polygons. The algorithm computes a simplified mesh with n' polygons, whereby $n' \leq n$ on condition that the rendered simplified

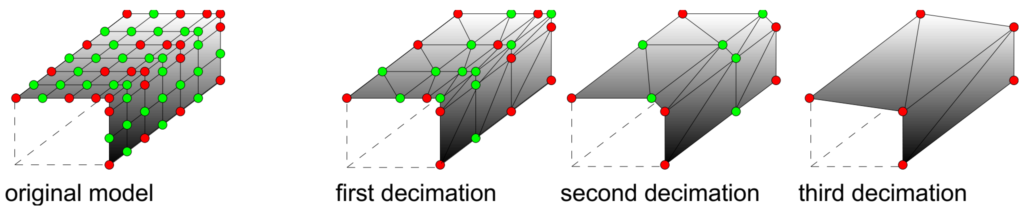


Figure 2.6: Decimation: the green vertices are systematically removed and the red vertices remain. After each removal, the whole mesh is retriangulated.

mesh looks similar to the original mesh (approximation quality). Fig. 2.5 shows four spheres with distinct resolutions. Spheres with fewer triangles show more corners. For a sphere, approximately 200 triangles are necessary to avoid recognizable corners.

We describe four polygon-removal algorithmic operations that are common for surface simplification algorithms: *decimation*, *sampling*, *vertex merging*, and *adaptive subdivision*. Some simplification algorithms use combinations of the four mechanisms. A detailed introduction can be obtained from several surveys [100, 176, 54, 88, 122, 151] and dissertations [149, 154, 101].

Algorithmic Operation Decimation: Decimation iteratively removes vertices or faces from the mesh and retriangulates the resulting hole after each step. The operations continue until they reach a degree of reduction of the polygons specified by the user. The algorithm mostly defines some kind of error metric. Some decimation algorithms try to preserve the local topology of the mesh. They do not permit a vertex or face removal that will change the local topology. Consequently, the algorithm may be unable to effect high degrees of simplification.

The meshes of Fig. 2.6 show two sides of the surface shape of a cube. The left mesh is the original mesh; the other three meshes show the mesh after three consecutive decimations steps. Each decimation step removes some rows and columns of vertices (green points) from the mesh of the preceding step. Afterwards, the resulting holes are retriangulated. From step to step, the error between the original mesh and the simplified mesh gets larger because the corner of the box is sloping. Some results of this approach can be obtained from the literature [196, 59].

Algorithmic Operation Sampling: In contrast to the decimation techniques, the sampling approach need not use a subset of the original vertices in the simplified mesh. In Fig. 2.7, the left mesh with the red vertices is the original model. The sampling operations start with a sampling of additional vertices of the original model (green points of Fig. 2.7). The samples can be points of the 2D manifold surfaces or voxels in a 3D grid superimposed upon the original model. After the removal of the original vertices (red

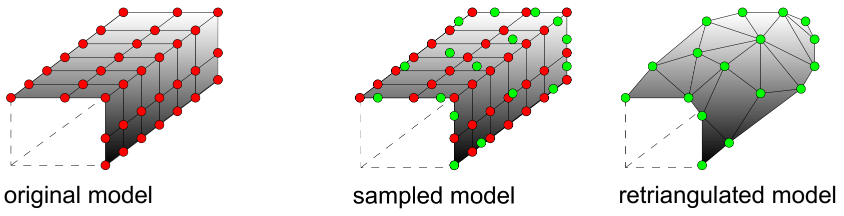


Figure 2.7: Sampling: vertices are sampled on the surface with the desired density (green). The original vertices (red) are removed and the remaining vertices are retriangulated.

points), the algorithm triangulates the new vertices so that the surface closely matches the original model. Varying the number of samples taken regulates the accuracy and degree of simplification of the original model. Errors can occur, so that the straight edges of the box are replaced by a plumb-looking shape (see Fig. 2.7). Some results of this approach can be obtained from the literature [221, 128, 118].

Algorithmic Operation Vertex Merging: *Vertex merging* algorithms take two or more neighbouring vertices and merge them to one new vertex, e.g., see the two green vertices of Fig. 2.8. One or more adjacent triangles collapse and these triangles must be removed. The resulting area triangulates the same space using fewer triangles than before the vertex merging step (see the yellow triangles of Fig. 2.8). The edges of the new triangles are connected to one of the original points. This property prevents the new edges from moving far away from the surface of the original mesh. This algorithm makes it possible to reduce some interesting areas of the surface mesh, independently of other parts of the surface where a high degree of triangulation is desired. This makes the method more flexible than the other techniques. Some results of this approach can be obtained from the literature [179, 126, 150, 127, 102, 174, 239].

Algorithmic Operation Adaptive Subdivision: Adaptive subdivision algorithms start from a coarse base mesh. Afterwards, they try to refine the base mesh by the addition of new points such that the distance to the original mesh is reduced. The distance is measured by an error metric. The Fig. 2.9 shows an original mesh, a base mesh,

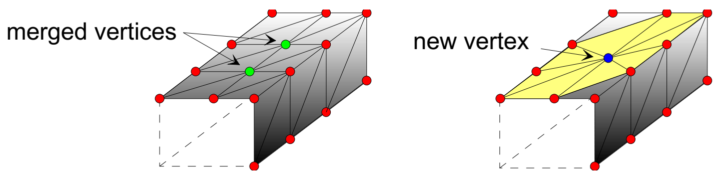


Figure 2.8: Vertex merging: two or more vertices are merged. Afterwards, collapsed triangles and vertices are removed.

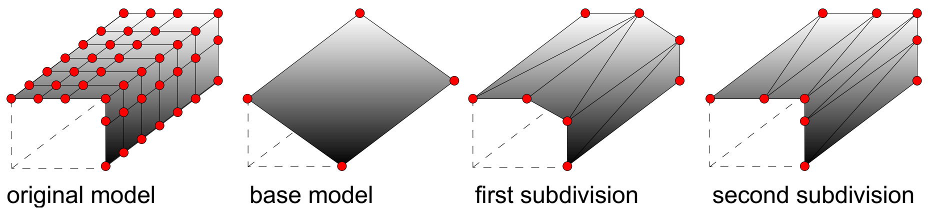


Figure 2.9: Adaptive subdivision: a coarse base model is computed. Afterwards, the mesh is refined until a desired polygon number or error is reached.

and two refined meshes. The additional points are placed at the edges of the cuboid. The algorithm can be classified as the inverse operation of the decimation approach. The main problem is the computation of the base mesh. Mostly, an other kind of simplification algorithm is used to compute the coarse base mesh. The advantage is the observation that one can better adjust the error by the addition of points than by the removal of points. This is why the added points can be at arbitrary positions, but the removed decimated points are fixed by the original mesh. Some results of this approach can be obtained from the literature [85].

2.1.3 Point Sampling

Instead of triangles, point sampling uses points as rendering primitives. The points can be obtained by sampling orthographic views on an equilateral triangle lattice [112] or by random throw of balls on the surface of the objects [227]. The basic idea of point sampling consists of the two steps *sampling* and *image reconstruction*. The house (1) of Fig. 2.10 is sampled and reconstructed out of the sampled points showing differences between the position of a sampled point and the position of the corresponding coloured pixel (2). If the density of points is sufficiently high, the rendered points will show a correct image of the object (4), otherwise holes occur (3) so that the shape (roof and outside wall) is unclear. Holes of false reconstructions cause misleading renderings, especially in the case of nearby parallel surfaces (see Fig. 2.10 right). From the top, the viewer sees a green surface in front of a red surface. Due to holes in the green surface, parts of the red surface shine through the green surface. The holes occur because some pixels contain only red sample points missing at least one green sample point.

The construction of an image out of a set of surface sample points makes point sampling independent of the input topology, thereby overcoming the main problem of polygon based rendering. Point sampling is used to speed up the rendering of complex scenes. Objects of complex scenes cover only a fraction of the pixels of the screen. The advantage of polygon-based scan-line coherence is lost because triangles or objects are

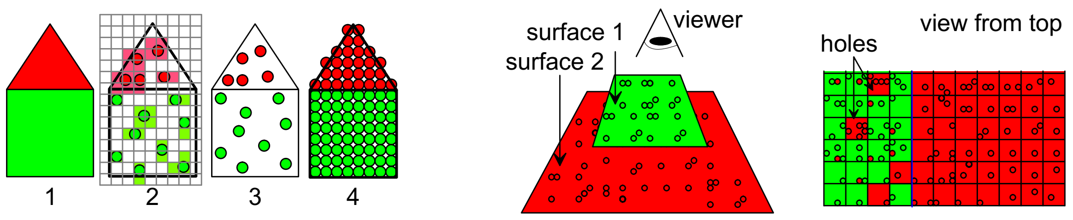


Figure 2.10: Left: A house (1), sampled and reconstructed (2) with holes in the surface (3), and a correct reconstruction (4). Right: Incorrect reconstruction of two nearby parallel surfaces showing holes in the green surface.

smaller than a pixel. The rendering time of polygon based methods is proportional to scene complexity (the number of polygons and the projected area) [121]. Point sampling achieves logarithmic rendering times with certain conditions [227].

Point sampling was introduced by Levoy and Whitted [146] to render surface models for the display of smooth three-dimensional surfaces. Also, points were used for the sampling and control of implicit surfaces [235]. Grossmann and Dally [112, 111] obtained point samples by sampling orthographic views on an equilateral triangle surface. They solve two problems at the same time, the reconstruction of continuous surfaces, and the efficient rendering of the data. Unlike points as input, Rusinkiewicz and Levoy [181] and Pfister et al. [170] convert polygon based rendering primitives into a uniform point representation. They build up a level-of-detail representation of points so that points of appropriate level can be selected depending on the screen projection. The simplicity of the hierarchy of points as a rendering primitive speeds up the rendering times. Several work use point sampling to speed up the rendering [211, 132, 49, 60, 8, 246, 234]. We ourself contribute the *Randomized z-Buffer* algorithm [227, 226]. We render the scene with a random set of surface sample points. Far away, objects are rendered with few points only and nearby the objects are rendered with many points. The number of points is proportional to the projected area of the objects. A careful choice of the sample set guarantees a high quality of the rendered image. We improve the randomized approach

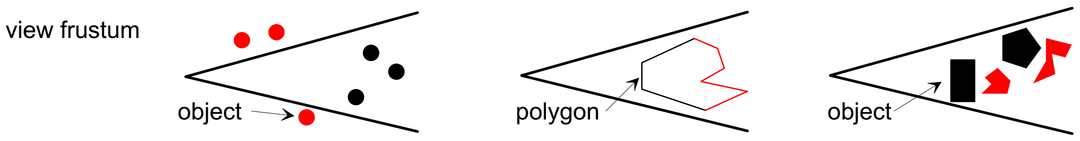


Figure 2.11: Visibility culling: view-frustum culling (left), back-face culling (middle), and occlusion culling (right). The red objects are invisible.

such that the randomization only performs in a single preprocessing step [137]. Our *Randomized Sample Tree* allows the I/O efficient access to the sampled data of scenes that are stored on a hard disk. A detailed introduction to point sampling can be obtained from several surveys [245, 244].

2.1.4 Visibility Culling

Visibility Culling reduces the number of polygons by a computation of a set of invisible polygons that are excluded from the rendering process. The literature distinguishes among *view-frustum culling, back-face culling*, and *occlusion culling* (see Fig. 2.11). View-frustum culling computes polygons that are completely outside the view frustum. The algorithms use spatial data structures such as octrees (see Section 2.3.3). For each new position, all nodes of the octree are traversed that lie inside of the view frustum. All octree nodes outside the view frustum are invisible and excluded from rendering. Back-face culling computes all polygons that face away from the viewer assuming that each polygon has only one visible side. The algorithm tests if the angle between the normal of the polygon and the viewing direction exceeds 180 degree. Occlusion Culling computes all invisible polygons that lie inside the view frustum.

The occlusion depends on the view direction, e.g., the scene is highly occluded in front of a row of houses, but above the houses the scene is densely occluded (see Fig. 2.12 right). The usage of visibility culling algorithms is practical only if the scene consists of sufficient invisible polygons because searching and testing for invisible polygons costs computation time. It is insufficient to only maximize the number of computed invisible polygons. More important is the time for the computation of invisible polygons. We save no time if it exceeds the time for the rendering of the invisible polygons. Some occlusion culling algorithms compute a large number of invisible polygons quickly, but they need much more time for the computation of all invisible polygons. There is a tradeoff between the rendering time and the computation time of the invisible polygons. In the schematic diagram (see Fig. 2.12, left), we assume a linear rendering time of the polygons

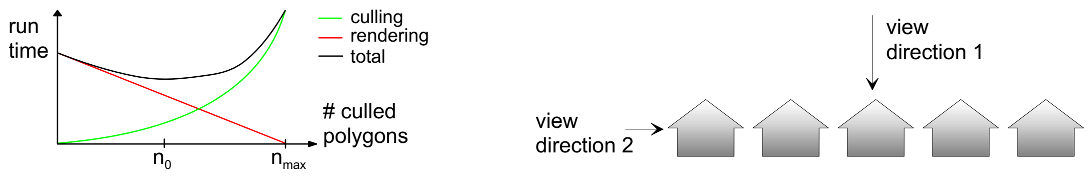


Figure 2.12: Left: Tradeoff of rendering time versus computation time of culled polygons (schematic diagram). Right: Different views of a scene with high occlusion.

(red line) and a non-linear time (green line) for the visibility culling computations. The total running time gets a minimum for n_0 computed invisible polygons.

Occlusion culling is used for densely occluded environments such as buildings, although the algorithms suffer from the knowledge of rooms and floors. Teller et al. [218, 99] compute *Potentially Visible Sets (PVS)* that contain all visible polygons and only a few invisible polygons (*conservative visibility*). A dynamic computation of the PVS is presented by Luebke and Georges [152]. We distinguish *image space based culling* algorithms that compute the polygons to be culled by the coordinates of the image space, and *object space based culling* algorithms that use the 3D coordinates of the objects to be culled. Coorg and Teller [65] present an object space culling algorithm that computes large occluders in the preprocessing step and use these occluders during the walkthrough for the culling of hidden objects. Greene et al. [108] present an image based algorithm that renders a spatial octree of the scene from front to back. Each octree cell containing a polygon is tested for visibility. If the box is invisible, all polygons are culled. Theoretical bounds of visibility are given by the aspect graph of Plantinga and Dyer [173]. They classify the ways in which the topology of the image of a polyhedron changes with the viewpoint. They show that a scene consists of $O(n^9)$ (perspective viewing model) maximal regions of viewpoints of the same aspect. Durand et al. [83, 84] introduced the 3D visibility complex that subdivides the visibility in line space and claimed a complexity of $O(n^4)$. The *approximate visibility* [214] allows little error due to missing visible polygons. This is practical in the densely occluded scenes where only overlapping objects result in an occluding map [243]. Occlusion culling algorithms work in common with replacements techniques, e.g., textures are used to “close” portals like doors and to efficiently cull all objects behind an open door or window [12]. A problem for the algorithms is to find a suitable occluder if the scene consists of many small objects that all together overlay to a large occluder. Bittner solved this problem by dynamically computing a suitable occluder [24]. A detailed introduction can be obtained from several surveys [62, 82, 23, 7] and dissertations [81, 242, 217].

2.2 Parallel Rendering and Networked Virtual Environments

Distributed and parallel rendering aims at the acceleration of the rendering process by many workstations. *Parallel rendering* is distinct from the *distributed* and *networked virtual environments*. The main objective of networked environments is the common interaction of multiple users in a distributed environment, but the goal of parallel rendering is the acceleration of the rendering process by means of parallelism. We see our work in both areas because we introduce a new data structure for the efficient distribution of a virtual scene and for the independent navigation of multiple users. The connection between the two is given by the kind of data structure and the kind of distribution. Our method provides for the efficient rendering of the scene for each user.

Parallel Rendering

Parallel rendering has been applied in computer graphics for a long time; for an introduction see serveral surveys [22, 38]. It is used for time consuming tasks such as ray tracing, radiosity [66, 177, 39], and volume rendering [240, 133, 236]. Most applications are off-line computations that need minutes to hours for the computation of a single image. However, recently more and more systems for real-time rendering have become available.

Parallelism is used with different approaches, e.g., for parallel occlusion culling [107], and for parallel ray-tracing used both to render off-line images of a film as well as interactive real-time ray-tracing on a cluster of workstations [223]. The typical problems of parallel processing occur. The methods must offer efficient solutions for the combination of the parallel computed partial results.

Since the underlying problem of rendering is a sorting problem, Molnar et al. [160] classify the parallel rendering approaches as *sort-first*, *sort-middle*, and *sort-last* architectures. The three classes distinguish where the sorting of objects takes place: before the geometry processing, between geometry processing and rasterization, and after the rasterization in the graphics pipeline.

Architectures

Parallel graphics architectures are grouped into three classes. The first type are *special purpose architectures* that are specialized in the solution of a specific problem, e.g., ray-tracing [148]. The pixel-plane architecture [89], starting from a more general raster based approach, also belongs to this group. The pixel-plane architecture is a SIMD (*Single Instruction Multiple Data*) machine where each processor rasters exactly one column of pixels of the display.

The second class of architectures are *tight coupled parallel systems* that are not designed for special kinds of rendering problems. In this group are SIMD machines such as the connection-machine (CM-2) that is used for ray-tracing [67] and MIMD machines (*Multiple Instruction Multiple Data*) such as the IBM SP/2 used for the rendering of methods that improve the image quality by incremental steps of refinement [178].

The third class of architectures is *distributed networks of workstations.* A network of 117 Sun workstations rendered the movie “Toy Story” [123] and 40 Amiga Workstations rendered the television series “SeaQuest” [238]. The workstations were networked and each of them was used to render parts of filming sequences. The first class, the special purpose architectures, has the best performance, followed by the second class of tightly coupled systems. The trend is going towards distributed systems because of improved network performance, low hardware costs, and the flexibility of a networked architecture.

Distributed Virtual Environments

A *networked virtual environment* is a virtual environment that is distributed across a network, e.g., the Internet. Multiple users interact with each other in the same virtual scene in real time. Each user is represented in the scene by a so called *avatar* or *entity* such that the user can be recognized by other users. The action of each user can be observed by other users in real time. In some systems the users can communicate with each other and exchange video messages [205]. The users share the common scene and run an interactive program on distinct workstations connected to each other via a network. The interface program is a walkthrough system that renders images of the environment as perceived from the viewpoint of the user. Applications are networked walkthroughs of large information spaces, networked multi-player games, virtual meetings, distributed training simulations, and computer supported cooperative work.

The challenge of such systems is maintaining a consistent state, e.g., the current position of each user, among a large number of users and workstations distributed across a wide network. Whenever a user changes his state (e.g., movement, interaction with other objects), an appropriate update must be applied to the system in order to maintain a consistent state. Despite the improvements in network technology, the network is the most constrained resource of the system. Architectures and data structure are required that minimize the bandwidth consumption when users navigate through a distributed virtual environment. If the scene is very large and stored on many disks across a network, we need a data structure for the spatial organisation of the scene that allows efficient access to the objects of the scene. If the user accesses neighbouring objects, the data structure should access only the disk where the object is stored and avoid references to other hard disks. The book by Zyda and Singhal [205] and several dissertations [2, 198, 194] give an overview.

2.2.1 Real-Time Rendering on Clusters

Recently, parallel real-time rendering has become popular because of increasing network and graphics hardware performance. Typically off-line rendering problems are becoming more and more rendered in real time in an attempt to make each frame run faster. In addition, the classical triangle and z-Buffer based rendering methods were improved by cluster based systems. They handle the triangle stream to move geometry and imagery across a network as required.

The main problem of cluster based real-time rendering is the balancing of the rendering tasks. Each workstation should be assigned an equal part of the rendering load. Otherwise, some workstations finish their rendering sooner than others, and thus the system is not optimally load balanced. There are different approaches to distributing and combining the rendering tasks. We can tile the display such that each tile has to be rendered by a workstation. In addition, we partition the scene such that each workstation has to render an equal part of the scene. Not all types guarantee a balanced load of the workstations because the current number polygons to be rendered depends on the topology of the scene and the viewpoint.

Examples

One example of cluster based rendering is interactive distributed ray tracing [223]. Wald et al. [224] render an image on a cluster of seven PCs that are connected by a Gigabit Ethernet network. For scenes of 50 million polygons at a resolution of 640x480 pixels, they achieve 4-5 frames per second on a client that is connected with the cluster. They bundle many rays in order to save network bandwidth [225]. Compared to other raster-based approaches, ray tracing has the advantage that it is easily to parallelise.

The scalable graphics system WireGL [129] uses a cluster that is connected by a Gigabit network. The systems consists of application nodes, rendering nodes with graphic acceleration (pipe servers), and many displays. On the application nodes, the programmer uses an OpenGL-like [237] library interface that replaces the standard OpenGL interface. Each application node generates a part of the virtual scene and starts preprocessing the data. Afterwards, the data is sent across the network to the rendering nodes. The data is distributed such that each rendering node renders a tile of the display. The system is scalable because of the three independent parts application, rendering nodes, and displays. The parallel access of the application nodes to the rendering nodes avoids the classical bottleneck between application and graphics hardware. Similar approaches are used in Chromium [130] that provide a sort-first and sort-last architecture that is similar to WireGL, but it is more flexible because its stream processors can be interchanged and combined arbitrarily. Thus, the system can be load balanced for several kinds of scenes.

The main goal of the SAGE graphics architecture [77] is the computation of high quality, aliasing reduced images. The fixed raster of the frame buffer is replaced by a sample buffer that can place up to 16 non-uniformly placed samples on each output pixel. In order to reduce aliasing errors, various filters are supported and the reconstruction process uses up to 400 samples per output pixel. Each SAGE board contains four parallel rendering subunits and several of the boards can be tiled together in order to scale the system for higher fill rates and resolutions.

The main problem is the partitioning of the rendering load. The first systems partition the 3D model or the 3D screen statically resulting in worse efficiency because some of the workstations render more triangles than others. The unbalance is dependent on the topology of the scene and the current viewpoint of the camera. Improved systems use a view-dependent partitioning of the 3D model and the 3D image [187]. Some of the methods require either replicating the entire 3D scene on every workstation or they require the redistribution of parts of the scene if the viewpoint changes. Thus, such kinds of systems need high memory capacity and network bandwidth. An interesting approach to trade off these demands is the *k-way replication* of Samanta et al. [186]. They duplicate copies of parts of the scene to k of the n workstations ($k \ll n$) in order to reduce the communications overhead. During the rendering, exactly one of the k servers renders each object. Thus, the system can be balanced if the viewpoint changes by dynamic partitioning.

Hardware Supported Image Reconstruction

One problem of a parallel rendering system is the combination of all images to a single image. This can be done by either a software and a hardware solution [129]. The software solution uses a server that collects all images from the rendering clients across a fast network. The hardware solution uses dedicated hardware components that get the images directly from the rendering hardware of the clients and combine them to a single image. Typically, the hardware solution is faster than the software solution because a large amount of data has to be sent across the network. WireGL achieves frame rates of 8Hz with a software solution and a frame rate of 90Hz with the hardware solution Lightning-2 [129]. The Lightning-2 system is scalable for the number of rendering nodes and the number of displays [215]. One important feature is that they use only standard graphics hardware to ensure a high degree of flexibility. The combination of the partial images cannot be performed by a simple overlay of the video signals because more complex information like depth values is necessary [215]. Also, the access by the AGP bus is too slow. However, today's graphics hardware can do this using digital outputs (digital display interface, DVI [113]). Further examples of hardware solutions are the MetaBuffer [25], the “VISUALIZE fx” graphics hardware of Hewlett-Packard [68] and the PixelFlow machine of Molnar et al. [161].

2.2.2 Remote Rendering

Man and Cohen-Or [157] perform remote rendering where the scene is stored remotely from the client, but the rendering process is done jointly by client and server. The client stores a locally available geometric model of the scene. They assume that the textures to improve the realistic impression are expensive to transmit over the network. Therefore, the server stores all textures and renders a high quality image of the scene. Dependent on the current viewpoint, the server sends the rendered image to the client. The client maps the texture on the model and renders the image. Only the differences between the images are sent across the network between successive images.

In contrast to preceding image-based remote rendering systems, the QSplat system of Rusinkiewicz and Levoy [182] performs a view-dependent transmission of the geometric data across the network such that the client holds only the data that is valid for the current position. The whole scene is stored as a point-based data structure on a server.

Remote rendering is used for the improvement of the image quality. Stamminger et al. [212] use a workstation for the real-time rendering of a scene that consists of a moderate number of polygons. The image quality is improved by a powerful parallel remote workstation that renders high quality images via ray tracing. The computed images, used as textures, are sent to clients to improve the image from time to time.

Cohen-Or et al. [63] assume that the virtual scene is texture-intensive and that the size of the geometry is significantly smaller than the size of the textures. The client does not store the whole scene and loads a view-dependent stream of geometry and textures that is rendered for nearby views. These textures can be compressed well because of the temporal coherency, thus avoiding expensive loading times from the server.

2.2.3 Visibility-Based Approaches

Other approaches for network-based walkthrough systems address the visibility problem. They overcome the network bottleneck by avoiding transmission of invisible objects. The modified state of a user (e.g., position) is only sent to users that can see each other. Cohen-Or and Zadicario [64] generalize the Potentially Visible Sets (PVS) [218] for the use in distributed virtual environments. The classical PVS algorithms cannot be directly adapted for a network-based system. The server will need to permanently update the visible set of the clients because of the movement of the viewpoint. The authors introduce a so-called *ϵ-neighbourhood* of a given viewpoint. The client renders the model independently of the server as long as the user moves inside the ϵ-neighbourhood. During the walkthrough, the server transmits the view-dependent superset of the visible set of the ϵ-neighbourhood. The superset includes all the visible primitives that can been seen from a viewpoint of the ϵ-neighbourhood. The RING system [97] of Funkhouser is a further well-known distributed system that uses visibility culling for the reduction of network traffic (see discussion below).

2.2.4 Networked Virtual Environments

Interest Management

The heart of a distributed system is the *entities* of the system. An entity can be a user of the system or objects and active things that some users are interested in. A problem is the scalability of the system. The more users come into the distributed scenes, the merrier the communication traffic occurs, especially if the users request many entities of the system. A common approach is the filtering of the entities. Filtering means that the system knows that some clients of the system are only interested in some of the entities that are available in the system. Therefore, the system can reduce the communications overhead. For example, each client needs only to update the positions of those entities he is interested in. This approach is known as *interest management.* The systems can be classified by the part of the system where the filtering of entities is done (*server-based filtering*, *sender-based-filtering*, and *region-based filtering*) [2].

In a *server-based filtering system*, all clients are connected to one or more servers. The servers control the actual state of the entities and decide which state changes should be sent to which clients. The RING system [97] of Funkhouser is an example for server based filtering. RING manages the communication between multiple users interacting in a shared virtual environment. Its client-server architecture uses visibility-based message culling algorithms to reduce the message traffic. Server-based visibility algorithms compute potential visual interactions between entities representing users in order to reduce the number of messages required to maintain a consistent state among the distributed workstations. When an entity changes its state, the updates are sent only to clients with entities that can see the updated objects. A further example of server-based filtering is the MASSIVE system of Greenhalgh and Benford [109]. Each entity has a so-called *aura* that defines the extent to which interaction with other objects is possible. The interaction of two auras depends on the position of the objects and other attributes. If two auras collide, the server notifies the two objects and peer-to-peer communication between the two objects is enabled. This mechanism reduces the communications overhead because only entities that are interested in each other communicate.

In *sender-based filtering systems*, the entities decide which other entities the messages are sent to. The strength of the system is that all entities get only the needed messages and must not throw away useless data. The drawback of such types of systems is that each single entity must know all other entities that are interested in the entity. The Minimal-Rendering Toolkit [200] is an example for a sender-based system. Each entity of the MR toolkit holds a list of all other entities. The entity sends an update to all other entities if the state of the entity changes.

Region-based filtering systems use a tessellation of the virtual scene. Although mostly tessellated by the three-dimensional space of location, other kinds of shapes are possible. Entities of each region send the state changes to a central point that is responsible for that region. The receiving entities must subscribe to all regions which they are interested in. The drawback of such kinds of systems is the crowdedness of a region if many entities are placed in the same region.

Most systems use fixed local shapes for the partitioning of the scene in order to perform interest management. Typical shapes can be hexagons or grids. Abrams et al. [1] differ from the fixed shape by introducing a tree-tiered interest management system in order to avoid clumping. The main problem of a distributed system is scalability. A distributed system with thousands of users requires a high demand on the network capability because of the large number of messages that must be sent across the network. The general approach to achieve this is area of interest management (AOI). If there are n users in the scene, there are $O(n^2)$ potentially interesting relationships. A spatial subdivision reduces the relationships so that users must provide notification of state changes of users of the same subdivision and of users of neighboured subdivisions. A regular subdivision leads to an unbalanced load on the servers. Steed and Abou-Haidar [213] used a record of aggregate behaviour of the participants and constructed an irregular subdivision in order to minimize server and network costs. The CyberWalk system [163] uses multiple servers and algorithms to dynamically partition the entire scene into regions. A server manages each region. If a viewer visits a region, the server of that region is responsible for all requests. If the viewer crosses the boundaries of the regions, the servers of all neighbouring regions serve the request of the users. In order to maintain a uniform workload of the servers, they use an adaptive region-partition scheme. The VELVET [76] system uses an adaptive area of interest management that supports heterogeneity amongst participants with varying system performance. The system manages users with high-speed networking and supercomputer performance as well as users that work with a single workstation behind a slow dial-up connection. The user may elect to unilaterally reduce or increase his own view of the world. It depends on the performance of the system that the client uses. The area of interest can be enlarged and reduced dynamically so that, upon increase in load, the system automatically reduces the area of interest.

Replicated Geometry

One of the problems of a distributed virtual environment is access to the 3D data. Some applications, e.g., fast computer games, have all the data stored at each client. They use a locally stored copy of the virtual environment that is available before the application starts (DOOM [210], NPSNET [155, 247]). If the data is not available, we download it using VRML based browsers [34, 117]. Thus, fast access and rendering of the data is possible, but this solution is space wasting. Another reason for the distribution of the data is the situation that the client is not allowed to access all the data; so the distribution is the only solution. Systems that distribute the geometry data over the network are denoted *geometry replication systems*; they must transfer parts of the scene to each client. The question is how to organize the data and the storage of the scene such that fast access is possible.

One approach is the *Remote Rendering Pipeline* from Schmalstieg [194]. To avoid a single geometry replication before the application starts, Schmalstieg and Gervautz [195] use a geometry database maintained by a server. Users share the common scene across a network. The problem of long download times is addressed by the AOI concept. The client only has the data for those objects that are contained in a spherical area. If the client moves through the scene, the AOI is moved and additional objects are loaded from the server. In order to compensate for the delay introduced by the network transmission, the system uses prefetching. The LOD algorithm selects finer LOD objects than needed for the rendering when the objects are still relatively far away. Sewell [198] followed a similar concept of *data base manager*. He proposed that the virtual scene is managed separately from the graphics application. The rendering workstation operates only on a subset of the entire model. Methods for the reduction and culling of objects are performed on the data base manager. The manager sends only a partial subset of the virtual scene to the rendering workstation. Another solution proposed Hesina et al. [124] with the *Distributed Open Inventor*. Their implementation extends the standard library Open Inventor [233] so that the programmers can use a familiar software interface. The scene is distributed among several servers that are unable to render the scene, but they manage the scene. Parts of the scene graph are stored on different servers. The approach relies on the replication of the scene graph at every workstation. The workstation and the servers must keep all replicas synchronized.

2.3 Data Structures from Computational Geometry

We survey related algorithms and data structures from computational geometry. The problems deal with points, lines, and boxes. The algorithms search subsets, compute intersections and arrangements. Mostly, the space dimension $\mathbb{R}^d$ is arbitrary and constant. The two and three-dimensional space is interesting because of its practical use. Our fundamental data structure problem is range searching (Section 2.3.1). The basic data structure for our walkthrough system is a weak spanner. Weak spanners have strong similarities with spanners (Section 2.3.2). Standard spatial data structures for computer graphics systems are octrees and BSP-trees (Section 2.3.3).

2.3.1 Range Searching

Given a point set, a range query computes all points which lie in a specified region. The shape of the region can be arbitrarily formed and influences the running time of the algorithms that solve the problem. In two dimensional space, commonly used shapes are circles and rectangles; in the three dimensional space spheres and cuboids are used. Rectangles and cuboids are special cases of orthogonal range queries. An overview of the different types of range searching problems can be found in several surveys [158, 3, 5] and books [75, 4]. Typically, a complex data structure is computed in a preprocessing step. Afterwards, we compute one or more range queries very quickly. In one-dimensional space, we construct a balanced binary search tree in time $O(n \log n)$ and $O(n)$ space, such that the a range query can be reported in time $O(k+\log n)$, where k is the number of reported points. An orthogonal range query in two-dimensional space uses a more complex data structure, a kd-tree. A *kd-tree* splits the set of points into two subsets of roughly equal size using horizontal and vertical lines, so that a balanced tree is generated. The kd-tree can be constructed in $O(n \log n)$ time and $O(n)$ space. A rectangular range query can be performed in $O(\sqrt{n} + k)$ time, where k is the number of reported points. Another well-known data structure is a *range tree.* A range tree consists of several nested balanced trees. In the two dimensional space, the tree consists of a balanced tree for one dimension. Inside the tree, several balanced trees exist for the other dimension. The tree can easily extend to arbitrary dimensions. In d dimensions, a range tree of n points needs $O(n \log^{d-1} n)$ space and it can be constructed in $O(n \log^{d-1} n)$ time. Rectangular range queries can be reported in time $O(\log^d(n+k))$, where k is the number of reported points [75]. A sophisticated technique is *fractional cascading* [47, 48]. Fractional cascading improves the results by one dimension; queries can be reported in time $O(\log^{d-1}(n + k))$.

Some applications need algorithms that process input that is more complicated than points. Circles, spheres, and lines have been used as the shape of the input [114]. In our model, the input consists of spheres and balls. A technique to improve the running time of the algorithms is to restrict the input model, e.g., small and thin triangles in a set of triangles. This is useful because in some applications the worst-case constructions do not occur. An approach to the solution here is the work of Schwarzkopf and Vleugels[197]; they define *low-density environments.* A scene is a low-density environment if any bounding box of an object cannot intersect more than a constant number of objects of the same or larger size.

2.3.2 Graph Spanners and Geometric Spanners

Let $G = (V, E)$ be a weighted graph with n nodes and positive edge weights. A subgraph $G' = (V, E'), E' \subseteq E$, is an *$f$-spanner* (*graph spanner*) of G if for each node $u, v \in V$ there is a path from u to v in G' with $dist_{G'}(u, v) \leq f \cdot dist_G(u, v)$. The distance $dist_G(u, v)$ of a path is the sum of the weights of all edges of the path. We denote f by *stretch factor*. Spanners are used and motivated by applications in communications networks, distributed systems, and network design theory. If a set of n objects is given, we search for a good network that connects all objects, in such a way that special properties are maintained or optimized. For example, the path along the edges of the network should be minimized, i.e., we need a small stretch factor. Other interesting parameters of spanners are the construction time, the size and weight of a spanner, and the maximum vertex degree. An introduction and overview of different types and construction can be found in several surveys [208, 87] and dissertations [207, 42, 115].

Graph spanners were first introduced by Peleg and Ullman [169, 168], where they used the spanner for the synchronization of networks. For general graphs, it is desirable that the spanner be as sparse as possible, i.e., have few edges. Let $S_k(G)$ denote the number of edges of a k-spanner for a graph G. Peleg and Schäfer [167] demonstrate the problem of determining, for a given graph G and a integer m, whether $S_2(G)$ is NP-complete. Therefore, it is interesting to look for approximation algorithms that approximate a given bound and compute a construction. For a given graph G, Kortsarz and Peleg [138, 140] compute the sparsest 2-spanner G' with mostly $|E'| = O(S_2(G) \cdot \log(|E|/|V|))$ edges, i.e., the approximation ratio is $\log(|E|/|V|)$.

Geometric Spanner

Geometric network problems approximate the distances among the points of the Euclidean space. The input is a set of points; wanted is a *geometric spanner G* such that for each pair (u, v) of nodes $dist_G(u, v) \leq f \cdot dist(u, v)$ holds. The problem can be solved by a reduction of the graph spanner problem. We construct the complete geometric graph where the edge weight is the Euclidean distance. We search for a sparse subgraph, namely the *geometric spanner*, that approximates the complete graph hereby. For our type of walkthrough problem, a suitable class of graphs is geometric spanners. We use geometric spanners to perform a local bounded range query: if a single node of the graph is given, we perform a breadth first search that is bounded by the distance $f \cdot t$, where t is the maximum distance of objects to be computed.

For our walkthrough system, a small edge degree and small stretch factor f is relevant. Soares [209, 206] presents a bounded degree spanner. For every point set and stretch factor f, an f-spanner G of the point set exists such that the degree of G is bounded by f

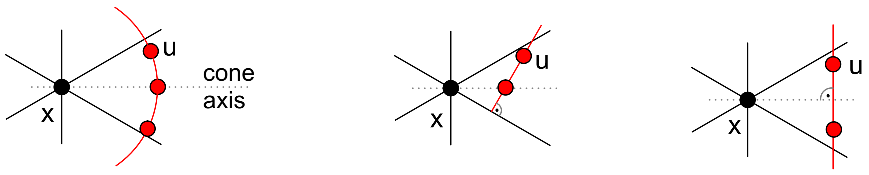


Figure 2.13: Distance measure of Yao [241] (left), Keil and Gutwin [135] (middle), and Ruppert and Seidel [180] (right).

and d, where d is the dimension. A simple greedy algorithm achieves this: subsequently, we add all edges (u, v) such that $dist_G(u, v) \leq f \cdot dist(u, v)$ holds. The edges (u, v) must iterate in increasing order with the distance $dist(u, v)$. The algorithm terminates if $dist_G(u, v) \leq f \cdot dist(u, v)$ holds for all edges (u, v). The degree of the resulting spanner is bounded by $O(d^{\frac{3}{2}} \cdot (\frac{\sqrt{2} \cdot f}{f-1})^d)$. The dimension $d = 2$ is fixed in our walkthrough system, but the we must adjust the stretch factor f during the walkthrough. In consequence, we must recompute the graph for every adjustment of the stretch factor. Therefore, the construction shown [209] is inappropriate for our application needs. We need a spanner construction that has a constant out degree, independent of the stretch factor.

Yao [241] presents a more usable construction that is the basis for our data structure: let x be a point in the plane. Like pieces of a cake, divide the plane into k uniform regions relative to x, where x is the centre of the region. The regions are formed by k lines each ending at point x; see the six sectors of the point x in Fig. 2.13. For each region, x gets an edge pointing to the next neighbour of the region. Yao [241] used this construction for the construction of a minimum spanning tree. In a d-dimensional space ($d \geq 3$), he presents algorithms such that the minimum spanning tree can be computed in time $O(n^{2-d'}(\log n)^{1-d'})$, where $d' = 2^{-(d+1)}$. The graph was named after the author as *Yao graph* [87]. Although the Yao graph may have unbounded degree, the important property for our data structure is the small bounded out degree.

The Yao graph is used for several spanner constructions: Keil and Gutwin [135] used the graph for a spanner construction that has only $k \cdot n$ outgoing edges, where k is the number of sectors. The ingoing degree of the graph is unbounded. The graph has a stretch factor of $1/(\cos\theta \cdot (1 - \tan\theta))$, where $\theta = 2\pi/k$ is the degree of each sector. Keil and Gutwin denoted the graph by *θ-graph*. The significant difference between the Yao graph and θ-graph is the type distance measure of the nearest neighbour in each sector (see Fig. 2.13). The Yao graph used the Euclidean distance, i.e., all points u are equidistant from a point x that are placed on are circle around x. In the θ-graph, all points u are equidistant that have the same perpendicular projection onto one of the sector boundaries. In consequence, the point x of the θ-graph will not necessarily be connected to its nearest neighbour in each region. Keil and Gutwin [135] computed the θ-graph in time $O(n \log n)$. At the time of publication, the result was faster than the construction time of the Yao graph ($O(n^{2-1/8} \cdot \log^{2-1/8}(n))$, [241]). Later, Wee et al. [232, 231, 230] presented divide-and-conquer algorithms that construct the graph in time $O(n \log n)$.

Rupert and Seidel [180] improved the stretch factor of the θ-graph by using a different distance measure. All points u are equidistant that are placed on the perpendicular of the line halving the sector of point x (see Fig. 2.13). The stretch factor of the graph is $1/(1 - 2\sin\frac{\pi}{k})$, where k is the number of sectors. The graph can be constructed in time $O(n(\log n)^{d-1})$ for dimension $d \geq 2$. In this work, we use the Yao graph and θ-graph, respectively, for the construction of our data structure. We use the same distance measure as Ruppert and Seidel, but for a different type of graph. In contrast to Ruppert and Seidel, we construct a weaker kind of spanner, a so called *weak spanner* (see Section 4). We show a smaller stretch factor as the factor for the spanner of Ruppert and Seidel; this in turn is an important feature for our walkthrough system. However, note that our factor and that from Ruppert and Seidel are not comparable because they describe different features of the same graph construction and distance measure. In order to make a clear distinction, we denote our graph by *sectorgraph.*

Spanner Constructions Based on the Delaunay Triangulation

Other ways to get a spanner construction are the usage of well-known graph types or data structures, e.g., the Delaunay triangulation. The Delaunay triangulations are good candidates for the approximation of the complete Euclidean graph since they contain a linear number of edges and can be computed quickly $O(n \log n)$ [175, 75]. The results of the stretch factors depend on different distance measures used by the authors. Chew [52] shows that the Delaunay triangulation of a point set in the L_1 metric is a spanner with a stretch factor $f = \sqrt{10}$. Afterwards, Chew [53] used a slightly modified measure of the Euclidean measure, the *triangle distance*, and reached a stretch factor $f = 2$ for the Delaunay triangulation. In Euclidean metric, Dobkin et al. [79, 80] show a stretch factor $f = \pi(1 + \sqrt{5})\frac{1}{2} \approx 5.08$ for the Delaunay triangulation. Keil and Gutwin [135, 134] improved this bound and show that each Delaunay triangulation of a point set approximates the complete Euclidean graph by a stretch factor $\frac{2\pi}{3\cos(\pi/6)} \approx 2.42$. Despite the good results for the stretch factors of Delaunay triangulations, the graph does not optimize all parameters that are important for good spanners. One can construct a set of n point sets where the Delaunay triangulation has a weight of order n times the weight of a minimum spanning tree (MST) [136]. To overcome this drawback, one can slightly modify the Delaunay triangulation. Levcopoulos and Lingas [142, 143] modify the Delaunay triangulation in time $O(n)$ and show that for an arbitray $r > 0$, the resulting graph approximates the Euclidean graph by a stretch factor of $f = (1 - \frac{1}{r})\frac{2\pi}{3\cos(\pi/6)}$. The weight of the graph is bounded by $(2r + 1) \cdot wt(MST)$.

Sparse Spanners with Low Weight and Low Size

The size $size(G)$ of a spanner G is defined as the number of edges and the weight $wt(G)$ of the spanner as the sum of the edge weights. *Sparse spanners* have a low size or weight. The minimum spanning tree (MST) is, by its definition, the sparsest possible spanner in terms of both size as well as weight. However, the stretch factor can be worse [14, 13]. Therefore, the sparseness of a good spanner is compared to the size and weight $wt(MST)$ of the minimum spanning tree. A further important parameter is the diameter; it is defined as the maximum number of edges of a spanner path. For spanners of bounded degree, a better diameter than logarithmic is difficult and comes only at the expense of an increasing degree. All parameters are interdependent; it is difficult to optimize all parameters at the same time. Constructions with a good stretch factor and a good size may have a worse weight. We give some examples of the research.

Optimal spanner constructions, in terms of weight, for two-dimensional Euclidean space are presented in [14, 142, 143, 72]. There exists $O(1)$-spanner with size $O(n)$ and weight $O(1) \cdot wt(MST)$. The results exploit properties that hold only for two-dimensional Euclidean space, so that new techniques were developed for higher dimensional spaces. For the d-dimensional space, Chandra et al. [46, 44, 45] achieved a spanner of weight $O(1)$ times the weight of the minimum spanning tree and with a stretch factor $O(\log^2(n))$. For the special case of a three dimensional Euclidean space, the stretch factor $O(\log^2(n))$ can be improved. For any $t > 1$, Das et al. [69] produce a t-spanner with $O(n)$ edges, and weight $O(1) \cdot wt(MST)$. Chandra uses a simply modified algorithm of Kruskal (to compute the minimum spanning tree) in order to show that for dimension $d > 3$ and a randomly choosen set of the points, an optimal spanner with weight can be constructed [41, 43].

Some research bounds the degree of the spanner. For example Salowe [185, 184] constructs a $t(k)$-spanner with vertex degree at most four, where $t(k)$ is a constant dependently from k for every $k \geq 2$. Das and Heffernan [70, 71] construct spanners bounded by degree three that additionally achieve an optimal weight $O(1) \cdot wt(MST)$. Bounding stretch factor and the degree of the spanner to be computed can result NP complete problems [139, 141].

Another important property of a spanner is the time required for its construction. The construction of a t-spanner with $O(n)$ edges is possible in time $O(n \log n)$ [183, 222, 32]. The spanner of Callahan and Kosaraju [32] used the so called *well-separated pair decomposition* of Callahan [30]. It consists of a binary tree whose leaves are points, with internal nodes corresponding to subsets of points in the natural way. In addition to the tree, they used a list of pairs of nodes, such that the sets corresponding to each node are geometrically separated, and each distinct pair of points is covered by exactly one of the pairs of nodes. Originally, they introduced this data structure for the nearest neighbour and the n-body problem [33, 31].

Arya et al. [15] concentrate on building spanners that optimize many spanner properties. They build spanners in time $(n \log n)$ and space $O(n)$ with bounded degree and an optimal weight of $O(wt(MST)$. Their spanners achieve optimal tradeoffs among several combinations of the spanner properties weight, degree, and diameter. They used a data structure, the *dumbbell tree*, that provides a method of decomposing a spanner into a constant number of trees, so that each $O(n^2)$ spanner path is mapped in one of the trees. The work improves the so far best known algorithm of Arya and Smid [19, 20] that constructs a bounded spanner in time $O(n \log^k n)$. Also, the work improves the construction time $O(n \log^2 n)$ of the spanner of Das and Narasimhan [73] that has optimal weight $O(wt(MST))$.

Other Directions of Spanner Research

As for other kind of problems, several techniques are used to improve the algorithms using randomized solutions [162, 57, 58, 17, 18] and approximate queries [16]. Another interesting type of spanners is fault-tolerant spanners [144, 145]. If in fault-tolerant spanners at most k edges or vertices are removed, then each pair of points in the remaining graph is still connected by a short path. Another kind of spanners is designed for a special kind of graph, e.g, a *grid spanner* for a grid of points [147]. The restricted topology of the points is useful in constructing parallel computers. In contrast to the restriction of the input topology, we can restrict the output, namely the topology of the spanner. For example, *tree-spanners* are spanners that are trees [116]. This restriction can result in hard decision problems, e.g., to decide if a given graph admits a tree t-spanner is NP-complete for $t \geq 4$ [29]. Also, other kinds of optimal spanners can result in NP-completeness, e.g., finding a t-spanner for planar weighted graphs with minimum weight is NP-complete [28].

Lower bounds were given by Chen et al. [51, 50]. They show that the problem of constructing a t-spanner for $t > 1$ takes time $\Omega(n \log n)$ in the algebraic computation tree model.

2.3.3 Spatial Data Structures

Spatial data structures are one of the fundamental types of data structure for the rendering algorithms of computer graphics. They are important because a good spatial subdivision is necessary for efficient access to parts of the data of the scene. Although many methods use the same spatial data structure for the same purpose, the methods are different. The difference is because how they use the data structure. We only examine briefly some basic work about the spatial data structures.

One important data structure is *quadtrees/octress*, they were introduced by Finkel and Bently [90]. Since then, many researchers have investigated quadtrees. This phase was completed by the books by Samet [190, 189] and surveys by him and Webber [192, 193, 191, 188]. The quadtrees/octrees find not only their applications in real-time systems for the spatial subdivision of the scene, but also in various other applications such as finite elements methods, VLSI design, robotics, and in computer graphics as means for the adaptive surface meshing together with wavelets and LOD [110]. A drawback of a quadtree is that the balance of the tree depends on the arrangement of the points. Therefore, we compute the depth of the tree depending on c, the smallest distance between two points, and s, the length of the bounding box of the point set. The depth of the quadtree for a point set is at most $\log(\frac{s}{c}) + \frac{3}{2}$. The tree of depth d and p points can be constructed in time $O(n(d+1))$.

A further important data structure for computer graphics is *Binary Space Partitions* Trees (BSP). They were introduced by Fuchs et al. [96] for the removal of hidden surface using the *painters' algorithm*. To subdivide the spatial area of a set of polygons, the BSP tree uses polygons of the scene as part of the dividing plane, whereby other polygons are split. This kind of subdivision results in $O(n \log n)$ expected fragments of the polygons; and the construction of a two dimensional BSP tree of size $O(n \log n)$ can be computed in expected time $O(n \log n)$ [75]. Although the construction time is worse, many applications use the BSP tree because they compute the tree in the preprocessing. If we assume certain properties on the topology of the scene, we get better results. De Berg [74] constructs BSP trees of linear size and $O(n \log n)$ construction time for *uncluttered scenes*. Similar improved size and construction time can be obtained if the aspect ratio of the rectangles is restricted [6]. BSP became important for the hidden-surface-removal problem [166]. They were used for the rendering of geometric data [220, 104]. Also dynamic variants of BSP trees are developed where the objects of the scene can be moved and deformed [219].

3 Architecture and Functionality of the System

We introduce a new scene model that we need for the data structure and architecture of our walkthrough system (Section 3.1). The model defines a virtual scene consisting of *objects*, a *viewer*, and a *modeller* (Section 3.1.2). The objects are basically for our scene model (Section 3.1.1). The user navigates in his environment, which we denote by *bubble* (Section 3.1.4). The user perceives only objects of a bubble; all objects outside the bubble are invisible. In Section 3.1.3, we describe our kind of distribution of the scene across a network. We define the available operations of the walkthrough system in Section 3.2. As a result of the scene model and the operations of the walkthrough system, we define the requirements to our data structure in Section 3.3.

3.1 Underlying Abstraction of Dynamic, Fully Distributed Scenes

3.1.1 Scenes Composed from Abstract Objects

Our architecture and data structure are designed for large and complex 2.5-dimensional scenes. A complex scene means that the 3D model consists of many polygons and a large scene means that the spatial extent is large. We assume that width and depth are large with respect to the height. Typical scenes are outdoor scenes as landscapes, cities, and whole countries. There are no further restrictions on the topology of the scene. We allow occluded scenes such as architectural models, and disconnected scenes such as trees, forests, and meadows. The height of large objects, such as buildings and mountains, is unrestricted. More important is that the diameter of the scene is much larger than the height of the objects. An exact limit is difficult; our experiments show that the system works well if the ratio of diameter to height is at least ten.

The reason for the restriction to 2.5 dimensional scenes is our data structure, which handles two-dimensional spaces only. The memory consumption of more-dimensional variants of the data structure is too large for practical applications. Our experiments show that the two-dimensional data structure works well for landscapes scenes.

Virtual scenes are semantically structured by the term *object*, e.g., a car, a house, and a tree are objects. The term is imprecise because an exact definition does not exist and objects are defined differently in applications. Objects may be modelled with a few as well as many polygons and they may contain other objects, e.g., a whole house is an object, and the window and the roof of the house could be objects. For our theoretical analysis, we need an exact object definition in order to compute exact runtime bounds. Our kind of object definition influences the runtime bounds of our algorithms strongly. We distinguish between objects of the theoretical model and objects of the virtual scene.

Objects of the Theoretical Model

The scene consists of m simple objects $o_1, \ldots, o_m$ of unit size. We denote the objects by *balls*. The balls are arbitrarily distributed without overlaps, i.e., the balls can touch each other, but one ball cannot enclose another ball. The balls have a fixed constant diameter r.

The requirement that the objects do not overlap is important for our data structure. Maximal $O(t^2)$ non-overlapping objects can be placed on a two-dimensional disk with radius t. This condition has an effect on the runtime bounds of our algorithms (for details see Chapter 4).

Objects of the Virtual 3D Scene

In contrast to the simple non-overlapping balls of the theoretical model, the basic units of the 3D scene are polygons that form surfaces with arbitrary topology. The problem arises of how to map the simple theoretical model onto the complex structure of a virtual scene. Each ball corresponds to an object of the virtual 3D scene. Typical objects of a virtual landscape scene are houses, trees, traffic lights, street lamps, crossroads, and pedestrian crossings (see Fig. 3.1).

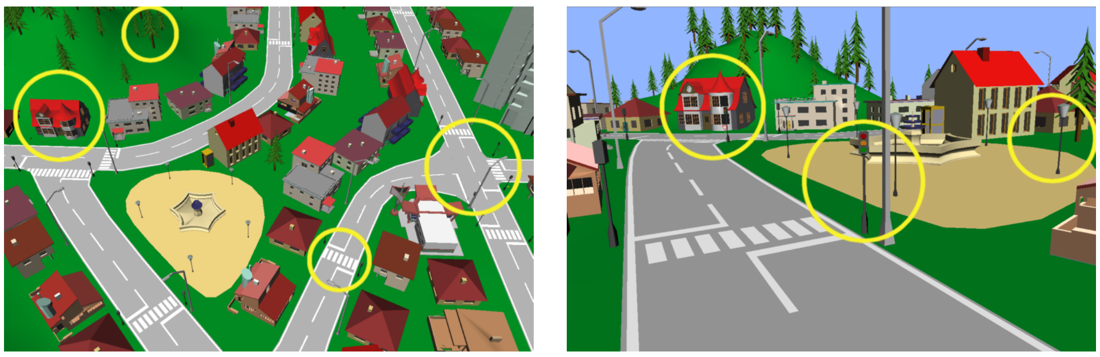


Figure 3.1: Typical objects of a landscape scene: houses, trees, traffic lights, street lamps, crossroads, and pedestrian crossings.

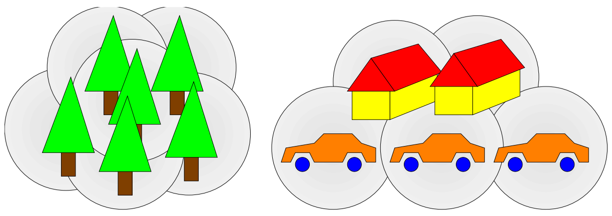


Figure 3.2: A small overlapping of the enclosing spheres $S(o_i)$ of all objects o_i is allowed.

Let $S(o_i)$ be the smallest enclosing sphere of Object o_i. The diameter r_i of the spheres $S(o_i)$ is bounded by a small fixed range $[r_{lower}, r_{upper}]$. A small range means that the difference between upper and lower bound is small in comparison to the diameter r_{scene} of the scene, i.e., $r_{upper} - r_{lower} \ll r_{scene}$ holds.

We allow that the enclosing spheres $S(o_i)$ overlap each other by a small constant value (see Fig. 3.2). However, the condition must hold that on a two-dimensional disk with radius t we can place maximal $O(t^2)$ enclosing spheres. Otherwise, the condition of our theoretical model is violated and the runtime expectation of our data structures does not hold.

The question remains whether it is possible to structure arbitrary virtual scenes with a number of slightly overlapping spheres. We achieve one simple construction with a regular grid of spheres in which all empty spheres are removed. This construction works for arbitrary scenes, but it results in degenerated objects, e.g., a car could be divided. There is a good chance to model a landscape scene without such types of artefacts. A house, a chair, and a table are good choices for an object if these objects do not overlap. If a table and a chair are placed inside a house, we choose only one single object.

3.1.2 Interactive Multi-User Navigation and Manipulation

We design a multi-user system for real-time navigation and manipulation in a virtual scene. Many persons use the system for a collaborative work. They navigate through the scene to arbitrary positions and work on different parts of the virtual scene at the same time. Each user is sitting at a rendering workstation that allows free navigation in the scene. In our model, we represent the collaborative work with a *viewer* and a *modeller* (see Fig. 3.3).

The ability of the viewer is restricted to the navigation in the scene. The viewer moves the position and orientation of his camera in order to navigate to arbitrary objects of the scene. The movement is slightly restricted: if the viewer moves from his current position x to a new position y, he must go the complete path to position y, e.g., viewer

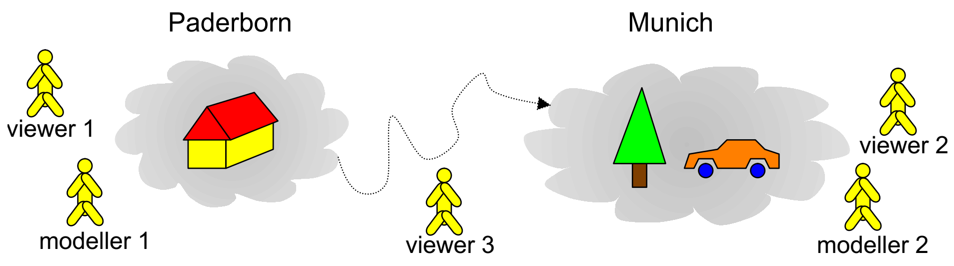


Figure 3.3: Modeller and viewer work on far away places of the scene. Viewer 3 moves from Paderborn to Munich

3 in Fig. 3.3. The system cannot beam the viewer down to a new location. Some walkthrough systems are able to beam the viewer to arbitrary positions.

Like the viewer, the modeller navigates through the scene. Additionally, he manipulates the scene and inserts and deletes objects of the scene. The manipulation is restricted to the insertion and deletion of complete objects. A manipulation of a single object is impossible, e.g., the insertion of a single polygon into an object. A single object is the *smallest editable unit*. The car, the tree, and the house of Fig. 3.3 are objects.

Although we support the manipulation of a scene, the system is not a 3D modelling program like *3d Studio Max* [78] or *Softimage* [21]. The system is designed for the collaborative work of many users on one single virtual scene. A typical application is the interactive visit to and design of large landscapes and cities. The user is interested in a manipulation of complete objects, i.e., he exchanges some models, e.g., houses, cars or trees. He uses a modular system of objects in order to build the scene.

3.1.3 Distributed Virtual Scenes

Our walkthrough system is a distributed system. The virtual scene is complex so that the objects cannot be stored in the memory of the computer. The data is distributed and stored on several disks that are connected by a network. In a preprocessing step, we build the scene such that spatially close objects are stored on the same hard disk (see Fig. 3.4). During the walkthrough of the users, newly inserted and deleted objects ought to balance among the disks. Other kinds of distribution are thinkable and intended, e.g., a random distribution.

The data structure, responsible for the navigation and management of the objects, is distributed among the hard disks, too. The objects of each partition are managed by a subset of the data structure stored on the same disk as the objects (see the green edges

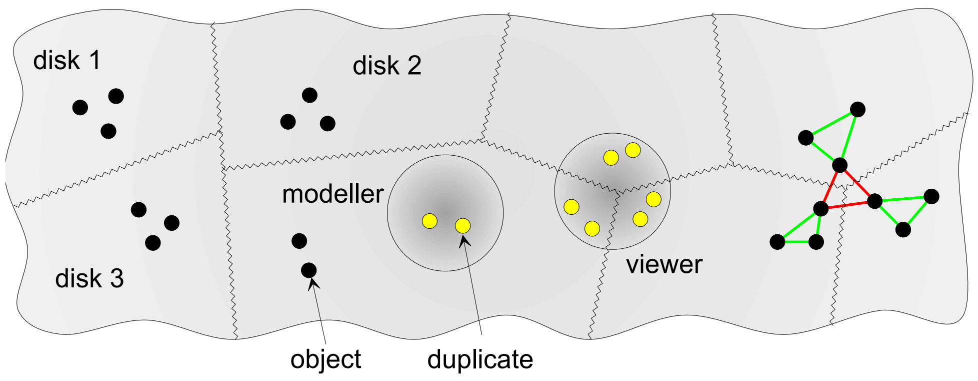


Figure 3.4: Distributed walkthrough system: the objects of the scene (black balls) are distributed among several hard disks. The bubble of the user stores duplicates of the objects (yellow balls). The data structure is stored on the hard disk (green edges). Links (red edges) connect the data structure between two hard disks.

in Fig. 3.4). Parts of the data structure are connected via links (red edges). These links refer from an object on one disk to an object on another disk.

The user walks through the scene without any knowledge of the partition of the scene. It does not matter if the bubble overlaps one or more objects that are stored on different hard disks or partitions (see the bubble of the viewer in Fig. 3.4). The user moves his bubble from one partition to another. The modeller inserts and deletes objects even if his bubble contains objects from more than one disk. He manipulates the scene without the knowledge that the objects are stored on several disks. He sees only one single virtual scene.

The bubble of the user contains a *duplicate* of the objects stored on the rendering workstation of the user. The black objects in Fig. 3.4 are the distributed objects of the scene and the yellow objects are duplicates of the objects. The objects of the scene are loaded from the hard disk and sent to the user via the network. The object is stored as a duplicate on the rendering workstation of the user. The duplicates are necessary for fast access and rendering of the objects. We remove the duplicates from the rendering workstation if the position of the objects is outside the bubble. We describe the four kinds of duplicates in more detail in Section 3.3.3.

3.1.4 Bubbles: A Spatial Hierarchy of Caches

Each user sits at a graphics workstation and has his own environment. An environment contains all objects of the users neighbourhood. We denote the environment of a user

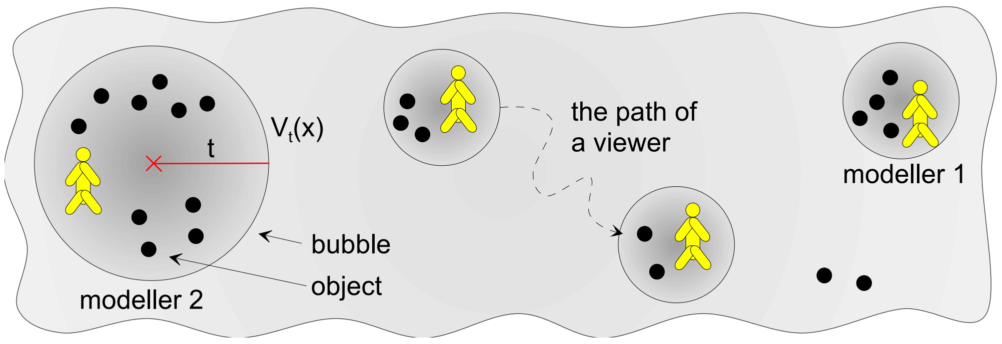


Figure 3.5: The basic model of the scene consisting of bubbles and objects: each bubble is a circle and contains all objects of the corresponding area.

by *bubble*. A bubble is a circle with *radius* t. Let x be the current position of the user, then $V_t(x)$ denotes the set of all objects with position y so that $|y - x| \leq t$ holds. The *centre* of the bubble is the current position of the user. The walkthrough system executes the movement of the user through the scene by the movement of the bubble (see Fig. 3.5).

The user sees only objects of the bubble; all other objects are invisible. We must choose the radius t of a bubble to be large enough. Otherwise, the user misses some far away objects that are outside the bubble. In typical scenes such as landscape scenes, this seldom occurs if we choose t large enough.

Requirements to the Data Structure

The bubble defines a spatially bounded environment of the user. Many viewers and modellers work independently of each other on different parts of the scene at the same time if their bubbles do not overlap. We want to transfer the property of the independent work to the property of the data structure.

See Fig. 3.5 for an example: modeller 1 and modeller 2 are working on two distant locations of the scene; x_1 and x_2 are the positions of modeller 1 and modeller 2, respectively. Both modellers work completely independently of each other and each bubble contains a subset of the data structure that manages all objects of the scene. Assume now that modeller 2 manipulates the scene. The insertion and deletion of objects of modeller 2 must not induce changes or references to the data structure of modeller 1. Modeller 1 does not know the positions of the objects $o_i \in V_t(x_2)$ of modeller 2. Therefore, modeller 1 must not update his data structure if modeller 2 manipulates the scene.

Just the same applies to a viewer moving through the scene during the walkthrough (see Fig. 3.5). The viewer must update his data structures if he moves to a new position. For these updates, he knows only the position of the objects of his neighbourhood. He needs no access to the objects of the bubbles of modeller 1 and modeller 2.

Hierarchical data structures, e.g., octrees, BSP-trees, $kd-$trees (see Section 2.3) create a hierarchy of the objects. At higher levels of the hierarchy (e.g., a tree), the data structure makes a coarse-grained subdivision. This subdivision is refined at lower levels of the hierarchy. That is, the hierarchy needs the references of all objects. If some objects are inserted and deleted anywhere in the scene, the hierarchy must be updated and parts of the hierarchy must be rebuilt depending on the type of data structure. Thus, two far away modellers cannot make their insertion and deletion independently from each other. Therefore, these data structures are unsuitable to fulfil our desired requirements. We present a data structure that fulfils these requirements in Chapter 4. The requirements are specified in more detail in Section 3.3.

3.2 Elementary Operations for Navigation and Manipulation

We describe the operations that we need for the movement of a user, and for the insertion and deletion of objects. We discuss some features of the operations and those our system supports, and explain why we do not support other kinds of operations. All operations are described in more detail in the following sections.

Dynamic Properties of the Virtual Scene

Some walkthrough systems support only static scenes. A *static scene* consists of a fixed number of polygons and objects. A modification of the scene is impossible. We can neither insert and delete a single polygon nor an entire object. Independent of the static behaviour of the scene, the visitor moves unrestrained to arbitrary positions of the scene. Sophisticated walkthrough systems use algorithms that replace a complex object by a less complex object in order to save rendering time. Also these walkthrough systems render a static scene because the user cannot manipulate the scene.

A scene is *dynamic* if the user inserts and removes parts of the scene during the walkthrough. The 3D-modelling program *3d Studio Max* [78] is an editor for full dynamic scenes. Our system allows the insertion and deletion of objects: we use precomputed models that are stored on the hard disk. The modeller starts with an empty scene and inserts the precomputed models in the scene. This feature gives the modeller a good degree of flexibility in order to build a scene from scratch. However, our system does not allow the modification of a single object, e.g., the modification of polygons.

Movement of Objects
A walkthrough system allows movements if the objects are able to move to arbitrary positions. For example, car and human models can move through a virtual scene. The degree of mobility is different: the system supports a high degree of flexibility if a large number of cars or human models are able to move to arbitrary positions. The rotating blade of a windmill is an example of a low degree of flexibility because the movement takes place in a spatial bounded area. The different degree of movement makes different demands on the algorithms. Spatially bounded movement is easier to support than the movement of many objects to arbitrary positions. Our system supports the movement of objects that take place in a spatially bounded area, but the movement of many objects to arbitrary positions is impossible. Our system does not support the free movement of objects in the scene.

Operations of the System
Our system uses four operations that are necessary for the movement of the user in the virtual scene. The operations compute the bubbles, move the bubbles, and allow the manipulation of the objects. Furthermore, the operations are necessary in order to iterate all objects of a bubble for the rendering of the objects.

SEARCH(x, t):	Let be y the position of an object o_i. Search all objects o_i with $\lvert x - y \rvert \leq t$.
MOVE$(x, \Delta x)$:	Move the bubble from position x to position $x + \Delta x$.
INSERT(x, o):	Insert object o at position x if x is the centre of the bubble and the user, respectively.
DELETE(x):	Delete the object nearest to position x if x is the centre of the bubble and the user, respectively.

We describe the operations in the following Sections.

3.2.1 Reporting from the Scene

The data structure must offer two report operations: the computation of a bubble and an iterator for a sequential listing of all objects. The operation SEARCH(x, t) computes a bubble with centre x and arbitrary radius t. For the computation of the bubble, a single object (seed) must be given that is inside the bubble (see Fig. 3.6). The operation SEARCH(x, t) is used if we create a new bubble or if we change the radius of the bubble from t_1 to a greater value t_2. We need this operation during the walkthrough if the scene becomes sparsely populated and the user sees the border of his bubble.

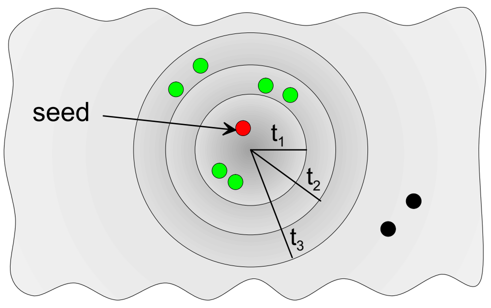


Figure 3.6: Report operation: an object (seed) is used in order to compute a bubble with an arbitrary radius t.

The second report operation is an *iterator* that computes a sequentially unordered list of all objects. This operation is necessary for the rendering of the scene. The listed objects are sent to the rendering pipeline in order to render an image for the user. We need the iterator after each insertion and deletion of objects, and after each movement of the visitor.

3.2.2 Insertion to and Deletion from the Scene

Our data structure supports fully dynamic operations for the insertion and the deletion of objects. The operation Insert(x, o) inserts a new object at position x, where x is the position of the modeller (see Fig. 3.7). The modeller inserts only whole objects. The objects are loaded from the hard disk. The centre of the bounding box of the object is placed at the position x. The operation Delete(x) deletes the object that is next to

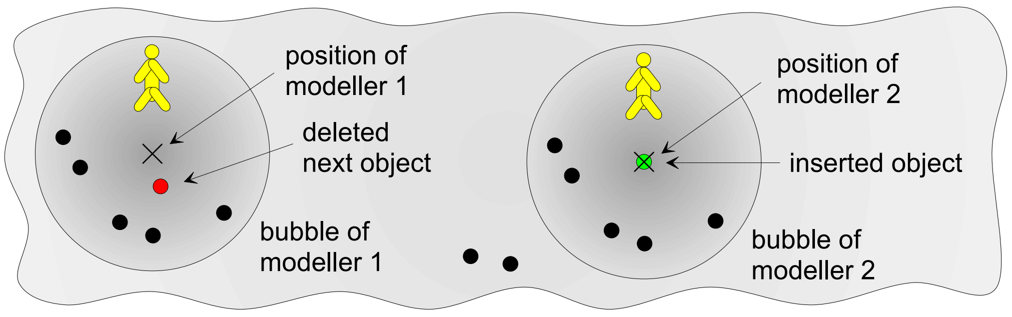


Figure 3.7: Insertion and deletion of objects: modeller 1 deletes the next object; modeller 2 inserts an object at his current position.

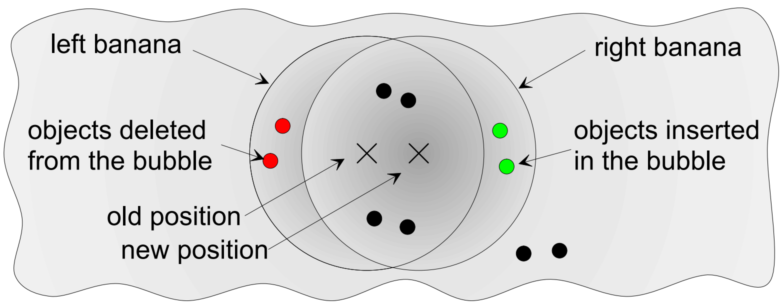

left banana
right banana
objects deleted from the bubble
objects inserted in the bubble
old position
new position

Figure 3.8: Movement of a bubble: the objects of the left banana are deleted from the bubble (red) and the objects of the right banana are inserted (green).

position x, where x is the position of the modeller (see Fig. 3.7). The modification of an object is impossible. If the modeller wants to move an object, he can do this with successive deletion and insertion operations. Delete and insert operations do not move the position of the visitor and the centre of the bubble.

3.2.3 Incremental Motion of Bubbles

The operation $\textsc{Move}(x, \Delta x)$ moves the bubble from position x to position $x+\Delta x$. This operation is necessary for the movement of the user. If the user changes the position of his camera, we have to move the bubble of the user too, because the position of the user is always the centre of the bubble. No operation is necessary if the user changes the orientation of his camera. If the centre of the bubble moves, some objects are deleted and some are inserted (see the two *bananas* in Fig. 3.8). The objects of the *left banana* are outside the bubble at the new position; they are deleted from the bubble. The objects of the *right banana* are inside the bubble at the new position; they are inserted. The system must compute all objects of the left and right banana.

Despite the modification of the bubble, the scene remains without any changes. The objects are neither inserted into nor deleted from the scene. The objects of the bubble are duplicates of the objects of the scene. We delete and insert only the duplicates of the bubble. Compare this with the $\textsc{Insert}(x, o)$ and $\textsc{Delete}(x)$ operations. Both delete and insert the objects completely from the scene.

3.3 Resulting Requirements to the Data Structure

The efficiency of the data structure is important for a fast execution of the operations and a successful usage in a walkthrough system. Furthermore, we need a simple handling

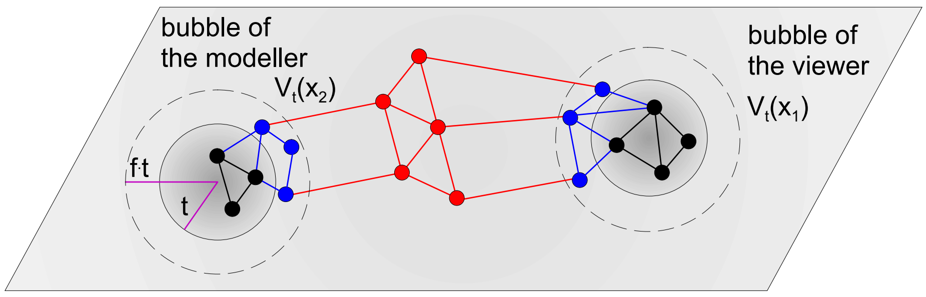


Figure 3.9: Locality: manipulation of objects of the set $V_t(x_2)$ does not affect data structure of other objects. Only parts of the data structure for the black and blue objects are referred and modified.

and the ability to implement easily. For this, we need four requirements that are the key observations for our choice of data structure. We denote the four requirements by *bubble requirement* and describe them in the following sections.

3.3.1 Spatial Locality

Our virtual scene is shallow and has a large diameter, for example a large landscape scene consisting of many towns and buildings. Many modellers and viewers use the scene simultaneously. Our walkthrough system supports independent operations for all users who are standing widely apart in the scene. For this purpose, we need a *locality of the data structure*. Locality means that just as we can view and model two parts of the scene independently of each other, we also change the appropriate parts of the data structure independently. The data structure of far away objects is well separated.

See Fig. 3.9 for an example: the scene contains two bubbles $V_t(x_1)$ and $V_t(x_2)$, one for the viewer and the other one for the modeller. If the modeller manipulates the objects of his bubble, all changes of the data structure are supposed to be bounded to the *spatial area* of his bubble. Objects and parts of the data structure outside of $V_t(x_2)$ are neither *referred to* nor *changed* (in Fig. 3.9 the red objects). It is unnecessary to refer to objects $o \notin V_t(x_2)$ if the modeller modifies objects $o \in V_t(x_2)$. *Locality of the data* means the spatial boundary of references to parts of the data structure. No references means that the modeller does not know the positions and number of objects outside his bubble. Moreover, he does not know if a scene outside his bubble exists. If the modeller inserts or deletes an object, he modifies the data structure only by local bounded references and changes to the data structure. The avoidance of reference to parts outside of $V_t(x_2)$ makes all parts outside of $V_t(x_2)$ completely independent of the

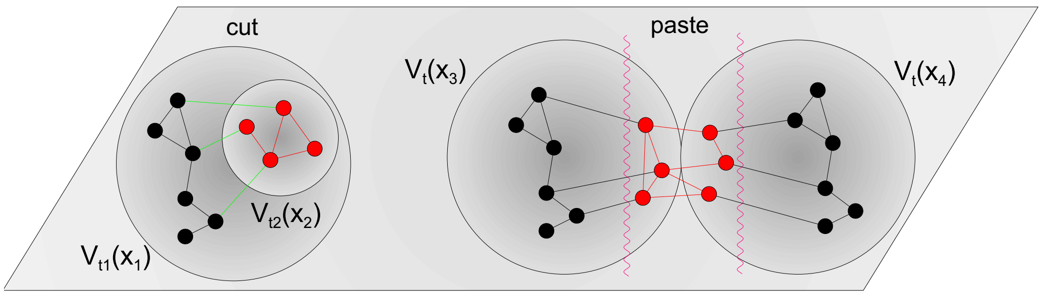


Figure 3.10: Cut: the extracted part of the data structure (red objects and edges) supports all operations without any recomputation. Paste: connect two independently modelled parts of a scene to a single scene by a recomputation of the borderline.

manipulation by the modeller. In the same way, the viewer of $V_t(x_1)$ does not take care of any changes of $V_t(x_2)$ if he moves through the scene.

We must slightly modify what we mean by *spatially bounded references*. If we modify the number of objects inside a bubble with radius t, we allow a reference and modification of the data structure to objects that are inside a circle with radius $f \cdot t$ for a small constant f (see Fig. 3.9). If the modeller inserts objects into his bubble $V_t(x_2)$ (black objects), he modifies and refers to objects of the data structure that are inside the dashed circle (blue and black objects and edges).

Locality makes the system fault tolerant to any changes or failures of far away parts of the scene. Furthermore, groups of modellers and viewers can operate independently in the scene if they are far away from each other. They need no synchronisation, data exchange, and communication in order to make their changes. For example, no parts of the data structure that are used for a walk through Paderborn are affected if the modeller inserts houses in Munich.

3.3.2 Support for Cut and Paste

Let $V_{t_1}(x_1)$ be a sufficiently large bubble with radius t_1 and let $V_{t_2}(x_2)$ be a bubble with radius $t_2 < t_1$ (see Fig. 3.10, left). The bubble $V_{t_2}(x_2)$ is completely inside of bubble $V_{t_1}(x_1)$. The objects are a subset of bubble $V_{t_1}(x_1)$, i.e., $V_{t_2}(x_2) \subseteq V_{t_1}(x_1)$ holds. If this condition holds, we say that we *cut* bubble $V_{t_2}(x_2)$ from bubble $V_{t_1}(x_1)$.

Our aim is to transfer the locality of the objects to the locality of the data structure. Just like the cut of the objects of bubble $V_{t_1}(x_1)$, we need a *cut of the data structure* for bubble $V_{t_2}(x_2)$. A cut of the data structure means that we extract and isolate a subset of the data of the bubble $V_{t_1}(x_1)$. This subset allows the movement and the

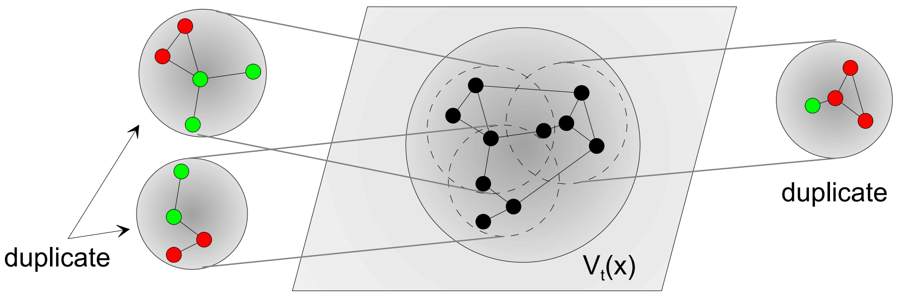


Figure 3.11: Duplication: the bubble $V_t(x)$ contains three bubbles, each consisting of duplicates of the objects. The green objects have more than one duplicate; the red objects have only one duplicate.

insertion and deletion of objects in the bubble $V_{t_2}(x_2)$. All data structure operations of the bubble $V_{t_2}(x_2)$ should be carried out with the extracted part of the data structure. We allow no further recomputation for the isolated parts of the data structure. In order to fulfil this requirement, we need a data structure that has the same locality as the locality of the objects in the scene.

Similarly to the *cut property*, we need a *paste property* in order to connect two independently modelled scenes together to form a single scene (see Fig. 3.10, right). The example shows two bubbles $V_t(x_3)$ and $V_t(x_4)$. We place the two one beside the other. Before we connect the two parts to a whole, we assume that each part has computed its own data structure. We connect the two parts of the scene by a recomputation of objects that are at the border of the two bubbles (red objects). We allow neither a recomputation nor a reference to the other objects (black objects and edges). If the data structure has this property, we denote it by *paste property*.

3.3.3 Support for Duplication

The walkthrough system supports independent movement and modification of the bubbles in the scene for many users. The users are sitting at several rendering workstations. We distribute the virtual scene over a network of hard discs. Therefore, every user must load a copy via the network. Each user has a copy of his own; they do not share the same data. We need a mechanism for copying and loading the objects from the distributed virtual scene. We denote such copies by *duplicates* because we have four different kinds of copies (see Section 3.3.4) and not every type is a simple copy of the data. Duplicates are self-contained instances of the same object.

See Fig. 3.11 for an example: the bubble $V_t(x)$ contains the distributed objects and three included bubbles (e.g., for three viewers) that consist of duplicates of the bubble $V_t(x)$. The bubbles can overlap, so that one or more duplicates can be descended from the same object. The objects of the green duplicates in the example have more than one duplicate; the objects of the red duplicates have only one duplicate.

If a modeller deletes an object of his bubble, the appropriate object in the distributed scene is removed and the data structure must be refreshed. Afterwards, the system looks for the bubbles that contain duplicates and the accordingly duplicates of the objects are removed. If the modeller inserts an object in his bubble, at first a new object of the distributed scene is generated and the distributed data structure is updated. Afterwards, the system makes duplicates for all bubbles that contain the new object.

3.3.4 Support for a Combined Bubble and Storage Hierarchy

The objects of the virtual scene, the polygon data, and the data structure are stored on several hard disks that are connected by a network. Different problems arise if a viewer or modeller wants to access the scene. The user must load and store the necessary objects of his bubble. The viewer needs a buffer if his neighbourhood is large, and if he wants to explore different parts of the scene. Otherwise, he would load and remove the same parts of the scene if he moves away and comes back again. For this kind of problem, we have four types of duplication: `TuNetwork`, `TuDisk`, `TuMemory`, and `TuReference`.

`TuNetwork`: A `TuNetwork` bubble stores all objects distributed over several disks connected by a network. Every rendering workstation gets access to the whole scene. The rendering workstation does not know where the object is stored. The `TuNetwork` bubble decides itself where a newly inserted object is stored.

`TuDisk`: A `TuDisk` bubble stores all objects on a single hard disk. Only the rendering workstation that is connected to the hard disk can access the objects of the scene. The objects cannot be accessed by other rendering workstations.

`TuMemory`: A `TuMemory` bubble stores the objects and the polygon data in the main memory of the rendering workstation. Only the rendering workstation can access the objects.

`TuReference`: A `TuReference` bubble stores instances that contain references to other duplicates.

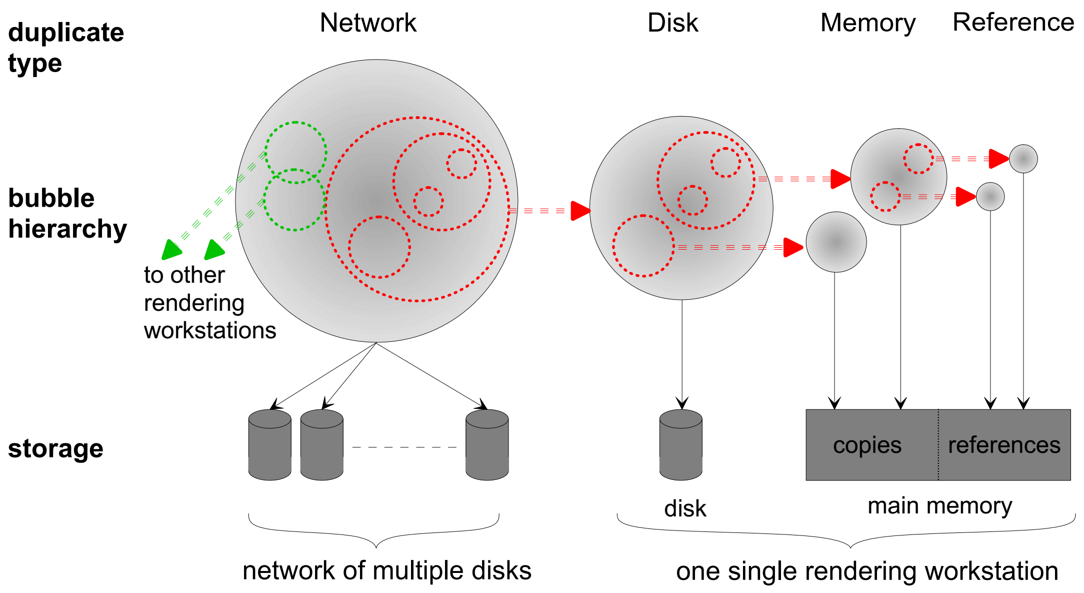


Figure 3.12: A hierarchy of bubbles: extracted parts of the data structure are slotted into each other. We store each extracted part on different types of storage (memory, disk, network, references).

Each bubble can contain one or more duplicates of different types. In Fig. 3.12, the large `TuNetwork` bubble contains three `TuDisk` duplicates, the large red dashed circle, and the two small green dashed circles. The bubbles may be included one in another. The `TuDisk` bubble includes two `TuMemory` bubbles. One of the `TuMemory` bubbles includes two `TuReference` bubbles. The bubbles form a *storage hierarchy* of bubbles that works as a *spatial hierarchy of caches.*

Only the `TuNetwork` bubble is distributed and stored on many disks. The other three types are stored on a single workstation, whereby a single workstation can store more than one type of bubble. In the example, the `TuDisk` bubble, the two `TuMemory` bubbles, and the two `TuReference` bubbles are stored on the same rendering workstation.

The question remains, why do we need such a spatial hierarchy of caches? We need the `TuNetwork` bubble in order to distribute the objects over many disks. The `TuDisk` bubble works like a buffer and cache, respectively, for a single rendering workstation. If a user stays in the same place for a longer time, all objects of the neighbourhood are stored on the hard disk of the rendering workstation. We need the hard disk if the objects do not fit in the main memory. The `TuMemory` bubbles are used for fast access to render the scene. The `TuReference` bubble is necessary if the modeller uses one or more rendering windows at the same time, for example, if he models simultaneously at different places. He looks for something in one window, and he models in the other window.

3.4 Summary and Discussion

We presented a model for the management of a scene that is distributed across a network of workstations. The scene consists of non-overlapping objects, so that it is impossible to put an arbitrary number of objects into a fixed area. This requirement of the model is consistent with practice because in typical scenes, the models overlap only to a small degree.

A visitor moves through scene, and a modeller manipulates the scene by insertion and deletion operations. The visitor, as well as the modeller, must move from his current position to a new position, i.e., he must walk along the whole way. Beaming to a new position that is far away is not supported. This requirement is not a restriction to many practical applications, because although technically easy to achieve, beaming confuses the visitors so that they lose their way in the scene. For the navigation and manipulation in the scene, we need the four operations $\textsc{Search}(x, t)$, $\textsc{Move}(x, \Delta x)$, $\textsc{Insert}(x, o)$, and $\textsc{Delete}(x)$ that must be supported by the data structure.

The extent of the scene is very large, so that the we assume that the visitor sees only a small part of the scene, namely all objects inside a circle that we denote by bubble. The bubble creates four requirements on the data structure:

1. To enable an arbitrary high scalability of the scene complexity, we require that the data structure used for the navigation of a visitor in a part of the scene must not be updated if far away parts of scene are manipulated, e.g., for insertion and deletion (*locality of operations and data*).

2. To enable an easy combination of some independently modelled parts, the combination of the parts into a single large one needs only objects of the border of the scene parts (*paste property*). To avoid the loading of whole data structure of the scene, we can extract a part of the scene as well as a part of the data. The extracted part of the data supports all operations without any recomputation of the data structure (*cut property*).

3. The scene is stored on remote workstations or on local hard disks. To enable fast local access in the main memory of the rendering workstation, the data structure must support easy duplication of data without any recomputation of the data structure (*duplication property*).

4. To enable buffering of the scene on hard disk and in main memory, and to enable the rendering of distinct parts of the scene at the same time, the data structure should support bubbles being slotted into each other. The slotting of bubbles forms a storage hierarchy that works as a spatial hierarchy of caches (*bubble and storage hierarchy*).

In the next chapter we present a data structure that fulfils these requirements.

4 Our Data Structure and Algorithms

We present and analyse a data structure that manages the storage, access, and manipulation of a virtual scene. The scene is 2.5-dimensional and scalable in two dimensions, i.e., all objects are three-dimensional, but the height of the objects is small in comparison to the diameter of the scene.

First, we think about the type of data structure that we need for our system. The problem of computing all objects inside a circle with radius r and centre x looks like a range query. Many data structures and algorithms exist for range queries (see Section 2.3.1), but only some data structures are practical and fulfil all the requirements of our bubble (see Section 3.3). We use a special class of graphs, the so-called *spanners* (see Section 2.3.2). A spanner has the property of locality that we need for our bubbles. For our data structure we use *weak spanners* that have weaker properties than a spanner (see Section 4.1). In Section 4.2, we define a weak-spanner that we use for the implementation of the walkthrough system. We discuss the problems that a weak spanner approach involves in Section 4.3. In Section 4.4, we use the weak-spanner to implement movement, insertion, and deletion. We discuss and compare our approach in Section 4.5 at the end of this chapter.

4.1 Weak-Spanner Approach

Before we investigate our new weak spanner approach, we define a f-spanner, a well known data structure in the computational geometry (see Section 2.3.2).

Definition 1 (f-spanner). *Let O be a set of n nodes (objects) in $\mathbb{R}^d$ and $d \in \mathbb{N}$ be a constant. Furthermore, let $G = (O, E)$ be a graph whose edges $(u, v) \in E$ are straight-line segments with Euclidean distance $dist(u, v)$ between two points u, v of the set O. The* length *$length(P)$ of a path P in G is the sum of the length $dist(u, v)$ of all edges of P. Let $f > 1$ be a real number. The graph G is a f*-spanner *for O if for each point u, v there is a path from u to v in G with $length(u, v) \leq f \cdot dist(u, v)$. We denote f by the* stretch factor *of the f-spanner.*

In other words, the spanner guarantees a bounded path length from every node u to

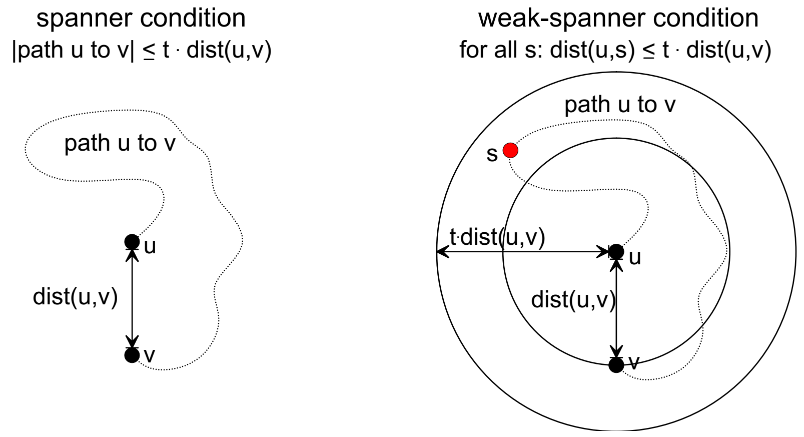


Figure 4.1: Spanner versus weak spanner: a spanner guarantees a bounded path length from every node u to a node v. A weak spanner guarantees only that the distance to every node s on the path from u to v is bounded. Therefore, every spanner is a weak spanner, but not the other way round.

a node v in G if we move along the edges of G. Note that the graph G must not be the complete graph.

A f-spanner is a good candidate for the computation of all objects of a bubble: starting with one object u of the bubble, we compute all other objects of the bubble by a simple breadth first search with the given point as starting point. Our breadth first search stops the search at all nodes s if the nodes are further away than $f \cdot dist(u, s)$. We compute all neighboured objects by a local bounded complete search of the neighbourhood.

Although we can implement a bubble with a f-spanner, we introduce a new graph type, the so-called *weak spanner*, in order to get a better data structure for our problem.

Definition 2 (f-weak spanner). *Using the notations of $O, G = (O, E), f > 1$ as in the definition of a f-spanner, we define a f-*weak spanner *as follows: the graph G is a f-weak spanner for O if for each point u, v there is a path from u to v in G so that for each point s on the path $dist(u, s) \leq f \cdot dist(u, v)$ holds.*

In other words, a weak spanner guarantees that the distance to every node s on the path from u to v is bounded. The path can be longer, like $t \cdot dist(u, v)$. Therefore, every spanner is a weak spanner, but not the other way round (see Fig. 4.1). A weak spanner fulfils our requirements for the bubbles. Compared with a normal spanner, the advantage is the size of the stretch factor. In the following section, we show smaller

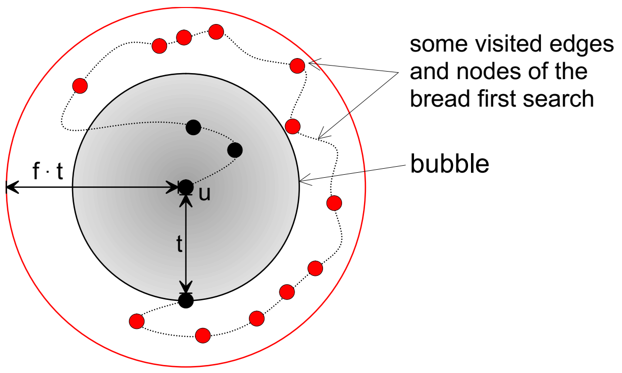


Figure 4.2: Neighbourhood sensitive versus output sensitive: output sensitive running time depends from the number of objects of the bubble (black objects). The neighbourhood sensitive running time depends on neighbouring objects that are outside of the bubble (red objects).

stretch factors for weak spanners than for normal spanners. This is important because the stretch factor has an effect on the running time. We introduced the notion *weak spanner* later [91, 92] when we define and use the graph [94].

In this work, the dimension d of the nodes is always two. We look only at two-dimensional spanners, although higher dimensional variants of spanners and weak spanners exist and we developed variants for $d = 3$ [93]. However, the two-dimensional construction is sufficient and easier to handle for typical landscape scenes.

For the classification of the running time, we need a definition that reflects locality.

Definition 3 (neighbourhood sensitive, neighbour factor). *Let S be a set of n objects of the virtual scene, $b > 1$ a real number, x the centre, and t the radius of a bubble $V_t(x)$. We call an algorithm b-*neighbourhood sensitive *if the running time of the algorithm depends only on the number of objects of the bubble $V_{b \cdot t}(x)$, i.e., the running time is a function of $|V_{b \cdot t}(x)|$. The number b is called the* neighbour factor.

The definition does not allow the dependency from n, i.e., the running time is independent of the size of the scene. The running time does not only depend on the size $V_t(x)$ of the bubble, but also on the neighbourhood of the bubble. If a bubble contains a few objects and the neighbourhood consists of many objects ($|V_t(x)| \ll |V_{b \cdot t}(x)|$), we get slow query times for the computation of the bubble, although the number of objects of the bubble is small (see the red objects in Fig. 4.2). This implies that the neighbour factor of our algorithms should be as small as possible.

What does this definition mean compared to an output-sensitive algorithm? An algorithm is *output-sensitive* if the running time of the algorithm is sensitive to the size of the output [162, 75]. If k is the output size of an algorithm, the running time depends on k and is mostly a small term of the input size n. On the one hand, the neighbourhood sensitive definition is harder because no dependency from n is allowed. Otherwise, the definition is weaker because it depends not only on the size of the output k, but also on the number of objects of the neighbourhood of the bubble.

Why do we need this kind of definition? Our main objective is to build a system and to develop algorithms that support locality of the user in the scene. The definition reflects locality because it measures the degree of locality. If the neighbour factor is much larger than one, the algorithm has a low degree of locality. A neighbour factor of one is optimal because a 1-neighbourhood sensitive algorithm does not refer to objects outside of the bubble $V_t(x)$.

4.2 The Sectorgraph

Now we define the weak spanner that we use for the navigation in our scene. The data structure makes the movement of the bubble and the insertion and deletion of objects in the scene possible. We define the graph, compute the stretch factor, present a scan-line algorithm for the computation of the graph, and finally show how one can use the graph for the implementation of the operations.

Definition 4 (γ-sector, γ-angle graph, sectorgraph). *Let k be an integer, $\gamma = \frac{2\pi}{k}$, and u in $\mathbb{R}^2$ be a position of an object in the scene. The k γ-*sectors *of a position x are defined as follows: draw k rays (line segments) from u such that they form angles $\frac{2\pi(i-1)}{k}, i = 1, \dots k$, with the vertical line through u. These rays subdivide $\mathbb{R}^2$ into k sectors around x.*

*The directed γ-*angle graph *$G_\gamma = (O, E_\gamma)$ on a finite point set $O \subseteq \mathbb{R}^2$ has the vertex set O and edges that are constructed as follows: for each $u \in O$ and each γ-sector, u has a directed edge to the closest object from O lying in the sector. The edge of the sector is non-existent if no object lies in the sector.*

*The function $S_v(i) : \{1, \dots, k\} \mapsto S \subseteq \mathbb{R}^2$ defines the space of each sector i for each object v. We define a function $sec_u(v) : O \mapsto \{1, \dots, k\}$ and write $i = sec_u(v)$ if object v lies in sector i of the object u. Also, we denote the edge of object u pointing to the next object v of sector $sec_u(v)$ by "*edge *$sec_u(v)$".*

We define the distance $d_\gamma(u, v)$ from u to v as follows (see Fig. 4.3 a)):

Definition 5 (distance measure, bisector line, orthogonal-bisector line). *Consider the γ-sector S of a node u containing the node v. Let the* bisector line *be the*

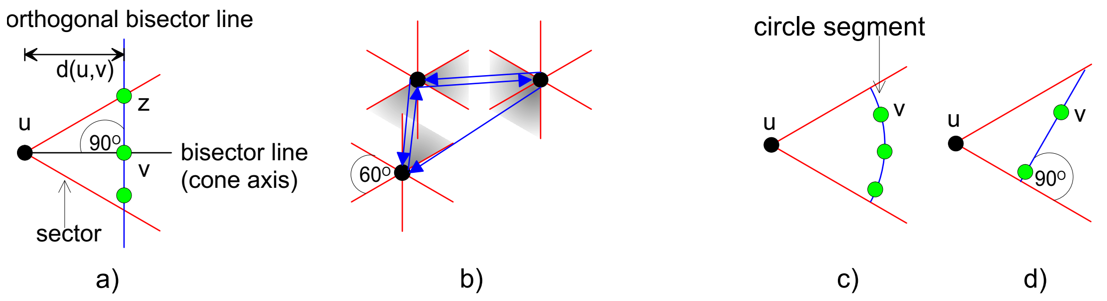


Figure 4.3: a) Construction of the sectorgraph ($d(u, v) = d(u, z)$), distance measure of Rupert and Seidel. b) example of the sectorgraph with $k = 6$ and $\gamma = 60°$. c) Distance measure of Yao d) Distance measure of Keil and Gutwin.

line halving the angle in S at u and let the orthogonal-bisector line *be the line orthogonal to the bisector line. Then $d_\gamma(u, v)$ is the (Euclidean) distance from u to the orthogonal-bisector line through v.*

The graph has out-degree k and the minimum number of edges is one. The graph is connected because each node "sees" at least one of the other objects through one of its sectors if the graph consists of more than two nodes. If $k \equiv 2 \pmod 4$ holds, the orthogonal bisector line is also a boundary of a γ-sector of v. See Fig. 4.3 b) for an example with $k = 6$ and $\gamma = 60°$. The example shows that two nodes can be connected in a unidirectional or bidirectional way. We assume that exact one line segment of a node belongs to one sector of the node, e.g., the left line segment bounding a sector belongs to the sector.

This kind of graph was first introduced by Yao [241], but he used the graph for the construction of a minimum spanning tree. The Yao graph used the Euclidean distance, i.e., all points v are equidistant from a point u that are placed on are circle around u (see the green objects of Fig. 4.3 c)). Afterwards, this kind of construction was used for various spanner constructions with modifications in the distance measure. Keil and Gutwinn [135] denote the graph by *θ-graph* and show that the undirected graph is a $(\frac{1}{\cos(\gamma)} \cdot \frac{1}{1-\tan(\gamma)})$-spanner for the Euclidean metric if $k \geq 9$ holds. In the *θ-graph*, all nodes v of a sector of a node u are equidistant that lie on a line orthogonal to one of the sector boundaries of the node u (see the green objects of Fig. 4.3 d)). Rupert and Seidel improved the stretch factor of the spanner by a modification in the distance measure. They show [180] that the γ-angle graph is a $(\frac{1}{1-2\sin(\gamma/2)})$-spanner if $k \geq 7$ holds. With that, it is shown that the distance measure influences the stretch factor of the sectorgraph construction. For more details on these results, see the discussion in

Section 2.3.2 on page 35.

We use the Yao graph and θ-graph, respectively, for the construction of our data structure. We use the same distance measure as Ruppert and Seidel, but for a different type of graph. In contrast to Ruppert and Seidel, we construct a weaker kind of spanner, the *weak spanner*. We compute a smaller stretch factor as the factor for the spanner of Rupert and Seidel; this in turn is an important feature for our walkthrough system. However, note that our factor and that of Ruppert and Seidel are not comparable because they describe different features of the same graph construction. For a clear distinction, we denote the graph by *sectorgraph*.

4.2.1 Circular Range Query in Output-Sensitive Time

We show how to compute a local range search in order to compute of all objects of a bubble. We need the algorithm for the movement of a bubble. We show that the γ-angle graph is a weak spanner with stretch factor f_γ. If one single object of the bubble is given, we show how far we must search the graph in order to compute all objects of the bubble. Afterwards, we modify a simple breadth first search in order to compute a simple range search.

Theorem 6 (weak spanner). *If $k \geq 6, \gamma = \frac{2\pi}{k}$, and $x \in \mathbb{R}^2$ is the position of an object, it holds: each object of $V_t(x)$ is reachable via one or more directed paths from x in the γ-angle graph. At least one path contains no objects farther away from x than $t' = f_\gamma \cdot t$. The bound for f_γ is tight for even k.*

$$f_\gamma := \max\left\{\sqrt{1 + 48 \cdot \sin^4\left(\frac{\gamma}{2}\right)}, \sqrt{5 - 4 \cdot \cos(\gamma)}\right\}$$

Proof. First, we assume that $k \equiv 2 \pmod 4$ holds. Let u, v be objects in the scene with $d(u, v) = t$. In order to lead us far away from u before we find v, an adversary places objects that we must choose as our neighbour. Our strategy to reach v is as follows: when having reached a ball w, we choose as next the edge going in that sector around w containing v. The strategy of choosing the objects and the choices of the adversary is shown in Fig. 4.4.

The left illustration of Fig. 4.4 shows the simpler, special case where $\gamma = \frac{\pi}{3}(\hat{=}60°)$ and $k = 6$. Note that, for $k \equiv 2 \pmod 4$, the rays from w_i through v, $i = 1, 2, \ldots$, are boundaries of γ-sectors of w_i. It is easy to check that v is always reached, and that w_2 or w_3 is always the object on the path far away from u, i.e., it holds $f_\gamma = \max\{d(u, w_2), d(u, w_3)\}$.

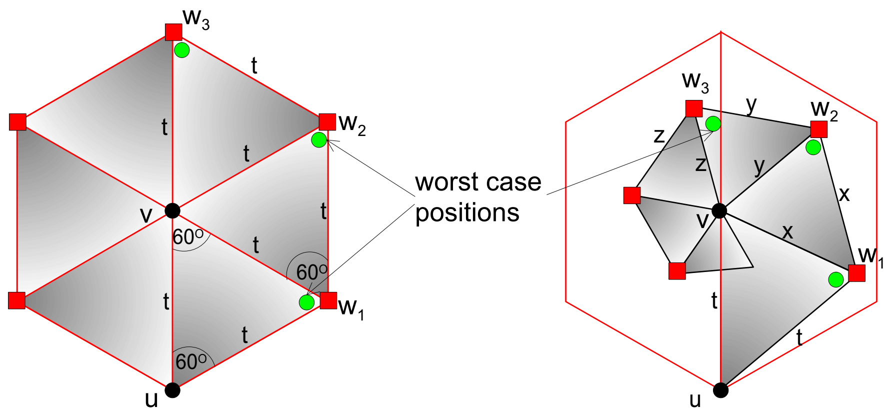


Figure 4.4: Search environment of the γ-angle graph for $k = 6$, $\gamma = 60°$ (left) and for $k > 6$ (right).

The following two calculations compute the exact distance from u to w_2 and from u to w_3. With $x = 2t\sin(\frac{\gamma}{2})$ and the cosine theorem we get for $d(u, w_2)$:

$$\begin{aligned}
d(u,w_2) &= \sqrt{x^2 + t^2 - 2xt\cos\left(90° + \frac{\gamma}{2}\right)} \\
&= \sqrt{4t^2\sin^2\left(\frac{\gamma}{2}\right) + t^2 - 4t^2\sin\left(\frac{\gamma}{2}\right)\cos\left(90° + \frac{\gamma}{2}\right)} \\
&= t\cdot\sqrt{4\sin^2\left(\frac{\gamma}{2}\right) - 4\sin\left(\frac{\gamma}{2}\right)\sin\left(180° + \frac{\gamma}{2}\right) + 1} \\
&= t\cdot\sqrt{8\sin^2\left(\frac{\gamma}{2}\right) + 1} \\
&= t\cdot\sqrt{8\frac{1}{2}(1-\cos(\gamma)) + 1} \\
&= t\cdot\sqrt{5 - 4\cos(\gamma)}
\end{aligned}$$

Moreover, with $z = 8t\sin^3\left(\frac{\gamma}{2}\right)$ and the cosines theorem we get for $d(u, w_3)$:

$$\begin{aligned}
d(u,w_3) &= \sqrt{z^2 + t^2 - 2zt\cos\left(90° + \frac{3}{2}\gamma\right)} \\
&= \sqrt{64t^2\sin^6\left(\frac{\gamma}{2}\right) + t^2 - 16t\sin^3\left(\frac{\gamma}{2}\right)t\cos\left(90° + \frac{3}{2}\gamma\right)}
\end{aligned}$$

Algorithm 1 LRS(t, o_s): Computes all objects of a bubble.

Require: γ-angle-graph of n Objects $O = \{o_1, \ldots, o_n\}$, radius t, and one object $o_s \in O$
Ensure: All objects $V_t(o_s)$ of a bubble

```
$Que.push(o_s); V_t(o_s) = o_s$;
while $Que \neq \emptyset$ do
  $u = Que.pop()$;
  for all neighbors $v$ of $u$ do
    if $dist(o_s, v) \leq f_\gamma t$ and $v$ is univisited then
      $Que.push(v)$;
      if $dist(o_s, v) \leq t$ then
        Add $v$ to $V_t(o_s)$;
      end if
    end if
  end for
end while
```

$$= t\sqrt{64\sin^6\left(\frac{\gamma}{2}\right) + 1 + 16\sin^3\left(\frac{\gamma}{2}\right)\sin\left(\frac{3}{2}\gamma\right)}$$

$$= t\sqrt{64\sin^6\left(\frac{\gamma}{2}\right) + 1 + 16\sin^3\left(\frac{\gamma}{2}\right)\left(3\sin\left(\frac{\gamma}{2}\right) - 4\sin^3\left(\frac{\gamma}{2}\right)\right)}$$

$$= t\sqrt{1 + 48\sin^4\left(\frac{\gamma}{2}\right)}$$

□

The above construction also shows the optimality of our bound for f_γ if k is even. If k is odd, the stretch factor f_γ is an upper bound. Now it is easy to execute a local range search, i.e., we compute all objects of a bubble if one object is given that lies in the centre of the bubble. We execute only a simple breadth first search. See Algorithm 1 for a detailed description; $Que()$ is a First-In-First-Out Queue.

Corollary 7 (local range search, LRS). *Let $o_1, \ldots, o_n$ n Objects and $G = (O, E_\gamma)$ be a precomputed γ-angle-graph with stretch factor f_γ for these objects. We compute the set $V_t(o_s)$ of all objects of a bubble with radius t in time $O(|V_{f_\gamma \cdot t}(o_s)|$ if the reference of an object o_s of the sectorgraph is given. The centre of the bubble is the position of the object o_s. The algorithm is f_γ-neighbourhood sensitive.*

Proof. We find all objects of $V_t(o_s)$ by starting a simple breadth first search (BFS) from o_s, without ever leaving $V_{f_\gamma \cdot t}(o_s)$. For each object v found during the search, we check whether $d(o_s, v) \leq t$ holds. The running time follows from the fact that we never refer to objects outside of $V_{f_\gamma \cdot t}(o_s)$ and that all objects are visited only once. □

All our efforts to find a weak-spanner with a small stretch factor are aimed at this simple and efficient algorithm that under no circumstances refers to any objects outside of $V_{f_\gamma \cdot t}(o_s)$. Above all, the existence of objects outside $V_{f_\gamma \cdot t}(o_s)$ and the current state of the data structure is completely irrelevant for the computation of $V_t(o_s)$.

In order to make the really interesting property clearer, assume that we get the object o_s and compute $V_t(o_s)$ at the time t_1, i.e., we get the subset of the data structure (perhaps a copy) that contains all objects of $V_t(o_s)$ and a little bit of $V_{f_\gamma \cdot t}(o_s)$. After the time t_1, some parts of the scene outside $V_{f_\gamma \cdot t}(o_s)$ are manipulated, i.e., objects are inserted or deleted, and the corresponding parts of the data structure are updated. After these modifications at time t_2, we use our local part of the data structure (a copy) of the bubble $V_t(o_s)$ again. At that time, no update of our part of the sectorgraph is necessary. Moreover, there is no necessity that we know anything about the preceding modification. Our data structure has this property because the locality of the objects in the scene is the same as the locality of the data structure. This property of sectorgraph was the key motivation for our development.

In addition the above-mentioned results for spanners [135, 180] yield stretch factors similar to Theorem 6. The spanners would fulfil our requirements too. However, their results have worse stretch factors f. For example for $k = 9$ $(10, 11)$, the result from Keil and Gutwinn [135] yields stretch factors $f = 8.11$ $(4.52,\ 3.32)$, the one from Ruppert and Seidel [180] $f = 3.16$ $(2.61, 2.29)$, whereas we obtain the stretch factors $f = 1.39$ $(1.32, 1.27)$. As our stretch factor is defined for $k \geq 6$ $(k = 6 \rightarrow f = 2)$, we can use a graph with small out-degree for our implementation. It is reasonable to choose a weak spanner with a small out-degree and a low stretch factor f for the implementation in order to save running time and memory.

4.2.2 Constructing the Sectorgraph in O(n log(n))

The efficient computation of the sectorgraph is possible by the means of the scan-line technique in time $O(k \cdot n \cdot \log(n))$ if n is the number of objects and k the number of sectors (the simple brute force algorithm needs time $O(n^2)$). The following algorithm works for $\mathbb{R}^2$ and with minor modifications for higher dimensions. The algorithm computes the edges of the sectors in consecutive passes. At first, in each pass we project the points onto the bisector line of the sector to be computed. Afterwards, we sort the points

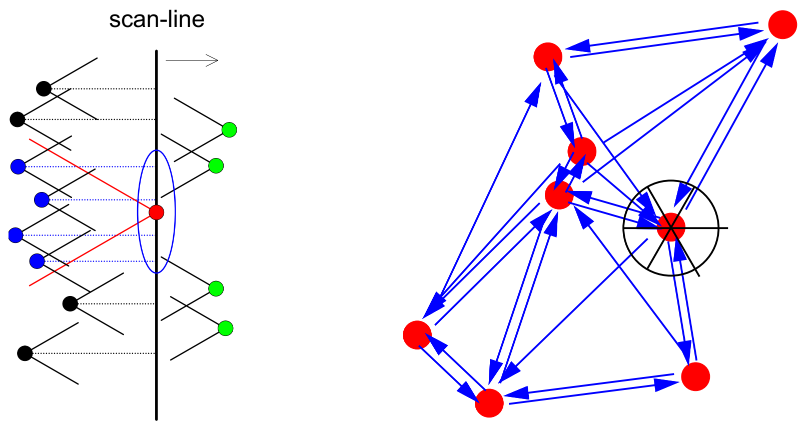


Figure 4.5: A situation of the scan-line during the computation of the γ-angle-graph (left). An example of the graph consisting of eight objects (right).

according to their projection onto the bisector line.

The scan line starts and iterates the points in increasing order (see the left illustration of Fig. 4.5). We keep in mind all objects that have an unconnected sector and store it in the data structure $SortSeq()$ (blue and black nodes left from the scan line). A single step of the scan line, from one object to another, looks like this: we query $SortSeq()$ for all objects that contain the object of the scan line in their sector (the blue objects left of the scan line). We connect all these objects with the current object of the scan line (the red object) and remove it all from $SortSeq()$ afterwards. Last, we insert the scan line object into $SortSeq()$. As usual for scan line algorithms, the non-processed objects are right of the scan line (the green objects). In the right illustration of Fig. 4.5, we see an example of the γ-angle-graph consisting of eight objects computed by our algorithm.

For the detailed description in Algorithm 2, we use the following notations and functions: $proj(sl, o_i)$ computes the projection of the point o_i onto the scan line. "o_i in sector s_j of o_c" means that the node o_i lies in the sector s_j of the node o_c. The data structure $SortSeq()$ keeps all inserted values in a sorted order. The operation $SortSeq.find(x)$ finds the next element to x and sets a reference to the element found. We use $SortSeq.nextup()$ and $SortSeq.nextdown()$ to iterate to the next smaller or next greater element of the sorted elements of $SortSeq()$.

Theorem 8 (γ-angle-graph computation). *Let n denote the number of objects and k the number of sectors. The algorithm computes the γ-angle graph in time $O(k \cdot n \cdot \log(n))$ and space $O(n)$.*

Algorithm 2 `CSG`($\{o_1, \ldots, o_n\}$): Computes γ-angle-graph

```
Require: $n$ Objects $O = \{o_1, \ldots, o_n\}$
Ensure: $\gamma$-angle-graph
  for all $k$ sectors $s_j$ of the origin $\vec{o} = (0, 0)$ do
    for all objects $o_i \in O$ do
      compute the projection onto the bisector of $s_j$;
    end for
    $Sort_p(O)$ := sort all objects according their projection onto $s_j$;
    $SortSeq := \emptyset$;
    for all objects $o_i \in Sort_p(O)$ in increasing order do
      $o_c := SortSeq.find(proj(sl, o_i))$; {$sl$ is the scan-line}
      while $o_i$ in sector $s_j$ of $o_c$ do
        connect $o_c$ with $o_i$; {edge for sector $s_j$}
        remove $o_c$ from $SortSeq()$;
        $o_c := SortSeq.nextup()$;
      end while
      $o_c := SortSeq.find(proj(sl, o_i))$;
      while $o_i$ in sector $s_j$ of $o_c$ do
        connect $o_c$ with $o_i$; {edge for sector $s_j$}
        remove $o_c$ from $SortSeq()$;
        $o_c := SortSeq.nextdown()$;
      end while
      put $o_i$ in $SortSeq()$;
    end for
  end for
```

Proof. Correctness: note that all objects left of the scan line either have found their nearest neighbour, or the objects are stored in $SortSeq()$. The invariant of the algorithm is that all objects of $SortSeq()$ have no nearest neighbour left of the scan line. Look now where the scan line reaches the next object. All objects of $SortSeq()$ are computed containing the scan-line object in their sector s_j. The scan-line object is the nearest neighbour for the objects of $SortSeq()$ because all other objects right of the scan-line object are farther away. This holds because the objects are scanned in the sorted order (see $Sort_p(O)$) and the scan line is the orthogonal bisector line of the sector s_j. As noted above, all objects of the scan line are equidistant for the sector s_j (see definition 5).

Time and space: the scan line iterates all objects with n steps. For each step, we have the following costs: the data structure $SortSeq()$ can be implemented with red-black trees. If n is number of elements of $SortSeq()$, the insertion, deletion, and query ($SortSeq.find()$) of an element can be executed in time $O(\log(n))$. As we insert and remove every object only once to $SortSeq()$, a single scan line step causes time $O(\log(n))$. The query of the next smaller or next greater element ($SortSeq.nextup()$ and $SortSeq.nextdown()$) costs constant time. The iteration of the objects during the scan steps adds up to time $O(n)$ for the whole computation because every object is iterated and removed only once. All iterated objects form a consecutive subset of elements of the sorted sequence $SortSeq()$. See the blue ellipse in the left illustration of Fig. 4.5. The algorithm sorts and projects the objects in time $O(kn\log(n))$. The distance measure implies the logarithmic running time of the algorithm. A Euclidean distance measure would cause more problems, although there exist algorithms with the same running time (see the discussion in the literature 2.3.2).

Each element is stored only once for the projection, once in $SortSeq()$, and once in the resulting list. Therefore, we need only space $O(n)$. □

The algorithm for $\mathbb{R}^d$ works similarly to the two-dimensional case. The problem is an efficient computation of all unconnected nodes left of the scan line that contain the scan line node in their sector. In two dimensions, we use a sorted sequence. Alternatively, we can use a more dimensional range query for the mirrored sector of the scan line node; see the red cone of the scan line node in the left illustrations of Fig. 2. All unconnected nodes left from the scan line lying in the mirrored sector of the scan line have the scan line node as their nearest neighbour.

4.3 Limitations and Overcoming Them

In our model, the bubbles use only movement operations that do not refer to far away points. The quadtree data structure (see Section 2.3.3) refers to far away points of the scene because it builds a tree of all objects of the scene. On the highest level, the construction algorithm decides which parts of the scene are far away and which parts are nearby. Our data structure can move the bubble without such a hierarchical structure and without references to far away points.

An alternative to a hierarchical data structure is a two-dimensional graph that connects the objects in the form of a network, e.g., our weak spanner. The connections are the edges between the objects. In order to ensure the locality of the data structure, we connect the objects that are neighbouring (e.g., see cluster one of Fig. 4.6). If the bubble moves through the scene, we compute the neighbouring objects with a breadth first search of the graph. In the following two Sections (4.3.1, 4.3.2), we describe some

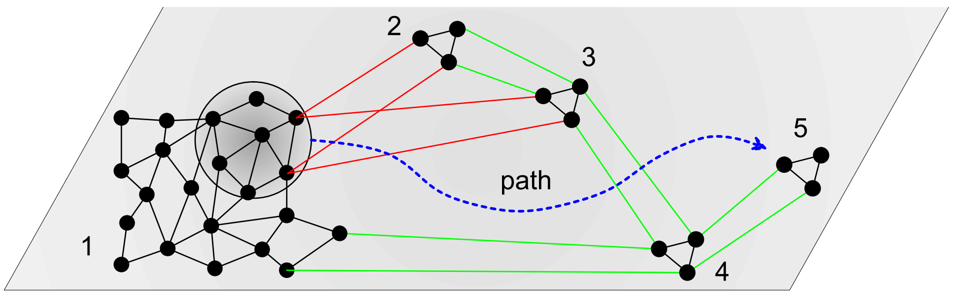


Figure 4.6: Deserts and long edges: the crossing of a desert is impossible without a search of the whole scene.

problems that result from the weak spanner approach. The solution to overcome the problem is described in Section 4.3.3.

4.3.1 Deserts and Long Edges

A problem of the movement of the bubbles is the navigation across large empty parts of the scene. A part is large and empty in comparison to the diameter of the bubble if the part is at least as large as the bubble.

The scene in Fig. 4.6 consists of five clusters (groups 1-5). The edges between the objects can be a graph that connects the objects such as our sectorgraph. The moving bubble starts in the large cluster one and moves along the path to cluster five. During the movement to cluster five, the bubble crosses parts of the scene where the bubble does not contain objects. Therefore, it is difficult to compute objects that will form a way through the cluster. At the starting position of the bubble, in cluster one, the bubble knows only the edges of clusters two and three (red edges). The bubble has no information about the edges of cluster one and three that lead to cluster four and from four to cluster five. Generally, large and empty parts of the scene are a problem for a weak spanner missing a hierarchical structure to make point location. We denote such large empty parts of the scene by *deserts*. The question arises, how we can bridge over large deserts. The long edges between the clusters do not help. We cannot guarantee that we will quickly reach the desired cluster if we cross the desert containing only long edges.

4.3.2 Unbounded Accumulation

A further problem of the weak spanner approach is the arbitrarily high density of points. Let every object be represented as a point of the sectorgraph. If the objects are far

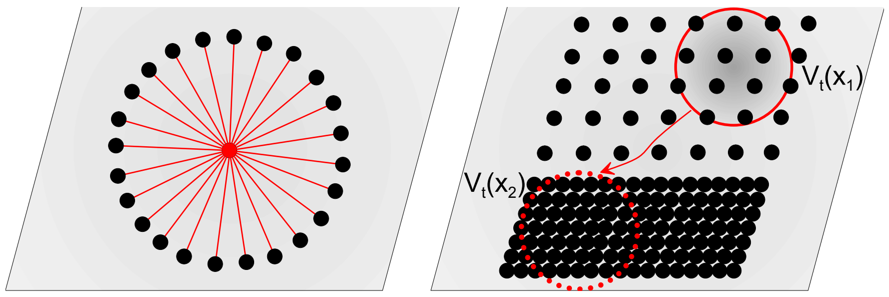


Figure 4.7: High Point density. Many objects are neighbours of the bubble in the centre (left). The bubble moves from an area with low density to an area with high density (right).

away, a single object has many neighbouring points. This is a problem for the efficiency of the insertion or deletion operations. We cannot guarantee a simple small bound because we can easily construct a worst case with many neighbours (see Fig. 4.7 for a worst-case example). In the centre of a circle of points (black), we have a single object (red). The red edges are edges of the neighbours pointing to the red object. We have to modify many edges if this object is inserted or deleted.

The point density is also a problem for the movement of the bubble because each bubble can arbitrarily include many points. This causes unsteady response times of the movement operation. In Fig. 4.7, the red bubble moves from an area of low density to an area of high density. As soon as the viewer arrives at the area of high density, the walkthrough system will slow down.

4.3.3 Crowded Scenes, Dummy Balls, and Non-Overlapping Objects

We use two techniques, *dummy points* and *non-overlapping objects*, to overcome the two problems "high point density" and "deserts"; both together solve the problems. Dummy points are auxiliary inserted objects in the scene. As an example, the objects are arranged on a regular grid in Fig. 4.8 (red squares). The dummy points are invisible to the user of the walkthrough system. The points prevent long edges because each object has a nearby neighbour. The bubble can move across the deserts because the desert consists of at least these dummy points. The bubble contains at least one object that is necessary for the movement of the bubble.

The condition of *non-overlapping objects* is defined by our abstraction of the scene and means that the objects have a fixed size and that only a small overlapping is possible (see the definition of our model in Section 3.1.1). The condition guarantees that on a square or circle with diameter r, we place only $O(r^2)$ objects (see Fig. 4.8

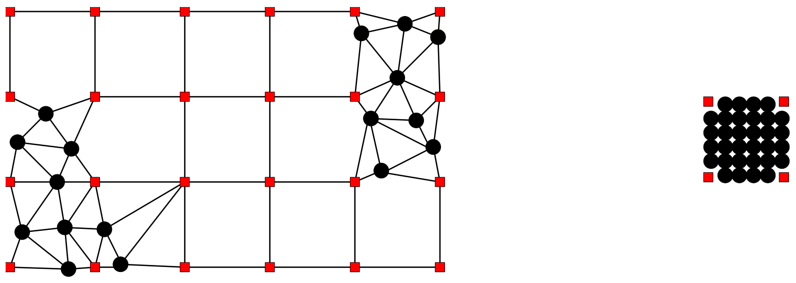


Figure 4.8: Dummy points (left) and non-overlapping objects (right) overcome the problems of high-point density and deserts.

right). Without this condition, we can place an arbitrary number objects on a square or circle with diameter r. This in turn involves an inefficient insertion and deletion of objects. The condition of non-overlapping objects is the reason for the efficiency of our algorithms. In our effort to model the problem, we must parameterize the dummy points technique. A parameterized technique gives us the possibility to make a runtime analysis. Dummy points are a means to increase the density of the scene. Therefore, we introduce a measure of the *density* of a scene.

Definition 9 (c-crowded scene). *We denote a scene by* (γ, c)-crowded *if for each point* $z \in \mathbb{R}^2$ *each* γ*-sector around* z *contains either no object or an object within distance at most* c *from the position* z.

In other words, a c-crowded scene gives us the guarantee that we find a next neighbour in every direction within a maximum distance, or the point is at the border of the scene and contains no neighbour in that direction. One simple way to make a scene (γ, c)-crowded for a given c, is a simple grid of dummy points.

Lemma 10 (grid of dummy points). *A grid of dummy points with edge length* $d = c/(1 + \frac{1}{2 \cdot \tan(\frac{\gamma}{2})})$ *is a* (γ, c)*-crowded scene.*

Proof. To make the crowd factor c as large as possible for a given d, we must set the top of the sector to the position as shown in Fig. 4.9. In this general position, we maximize the distance between the points p_1 and p_2 and the top of the sector without including the points p_3 and p_4 in the sector. For the distance d holds:

$$\tan\left(\frac{\gamma}{2}\right) = \frac{d/2}{c-d} \quad \Rightarrow \quad d = 2(c-d)\tan\left(\frac{\gamma}{2}\right) = \frac{2c\tan\left(\frac{\gamma}{2}\right)}{1 + 2\tan\left(\frac{\gamma}{2}\right)} = \frac{c}{1 + \frac{1}{2\tan\left(\frac{\gamma}{2}\right)}}$$

□

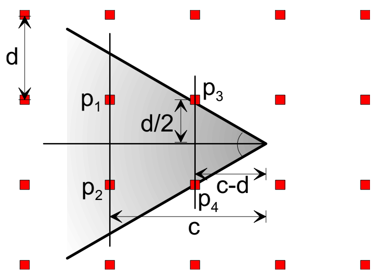


Figure 4.9: A grid of dummy points.

Therefore, we can turn every scene with unknown crowdedness into a c-crowded scene. For different angles $\gamma = 30°(53.13°, 60°, 90°, 160°)$ we get factors $a = 2.87(2.0, 1.87, 1.5, 1.09)$ if $c = a \cdot d$ holds. As we use our weak spanner for $\gamma \geq 60°$, it follows that long edges are "broken" at least after a second cell of regular grid. The auxiliary dummy points need space requirements quadratic in n in a grid-like manner. Of course, we cannot allow that in practical use. We can reduce the requirements to linear space if we use hashing. We store only the dummy points lying in the neighbourhood of objects, all objects lying in deserts were not stored [94]

For c-crowded scenes, we can strengthen the stretch factor of our weak spanner if the stretch factor is less than the distance between source and destination object.

Corollary 11 (stretch factor for c-crowded scenes). *Let x, y be objects of a (γ, c)-crowded scene with $t = d(x, y)$ and $c \leq t$. At least one path from x to y exists such that for every node s on the path holds $dist(x, s) \leq t + \Delta(\gamma) \cdot c$.*

Proof. The result follows from theorem 6 by a modification of the proof as follows: if hold $c \leq t$ hold, our adversary can place the objects only within maximum distance c instead of distance t. Therefore, we substitute t by c in the formulas of $d(u, w_2)$ and $d(u, w_3)$ and get the claimed bound. □

4.4 Implementation of Bubbles Using the Sectorgraph

This section shows how we use the sectorgraph to implement the operations of the bubbles, which are defined in Section 3.2. The problem is the computation of $V_t(x)$ for general positions. The computation requires that x is also the position of an object and the search starts from it. However, the condition does not hold for general positions. We give algorithms for the reporting, movement, search (Section 4.4.2), insertion, and deletion (Section 4.4.3). Finally, we show how to compose parts of a large scene in Section 4.4.4.

4.4.1 Cutting a Subgraph of the Sectorgraph

We introduced and defined objects and their duplicates (see Section 3.3.3), which are the basic units of the scene. We make the requirement that the data structure of a duplicated bubble $V_t(x)$ should be extracted from the data structure of the whole scene and that the duplicated subgraph should keep all properties such that all of our operations can work on the duplicated subgraph. We show how we extract a duplicated bubble from the γ-angle graph.

Definition 12 (subgraph of the sectorgraph). *Let $G_\gamma = (O, E_\gamma)$ be the γ-angle graph and $V_t(x)$ a bubble with centre x and radius t. We denote $G_{V_t(x)}$ by* subgraph *of G_γ if $G_{V_t(x)} = (V_t(x), E'_\gamma)$ is the subgraph of G_γ that is induced by $V_t(x)$, i.e., for all edges $e = (u, v) \in E'_\gamma$ it holds: $e \in E'_\gamma \Leftrightarrow u, v \in V_t(x), e \in E_\gamma$.*

In other words, $G_{V_t(x)}$ contains all edges that lie completely in the circle with centre x and radius t. The circle of the bubble cuts a subset of objects and edges from the γ-angle graph. Edges that are cut through by the circle are not in $G_{V_t(x)}$, i.e. all edges $e = (u, v)$ with $u \in V_t(x), v \notin V_t(x)$ and $v \in V_t(x), u \notin V_t(x)$, respectively.

We need the subgraph because the Delete(x) Move($x, \Delta x$), Insert(x, o), and Search(x, t) operations work not only on the complete γ-angle graph, but also on the bubble without any access to the whole graph. Like the objects of the bubble $V_t(x)$, the subset $G_{V_t(x)}$ also belongs to the bubble $V_t(x)$.

By this definition, the duplication of the bubble is an easy operation: we must compute and duplicate all objects of a bubble $V_t(x)$ in a first step. Afterwards, we duplicate all edges of $G_{V_t(x)}$ in a second step.

4.4.2 Algorithms for Reporting and Incremental Motion

We have all means in order to implement the operations Search(x, t), Move($x, \Delta x$), Delete(x), and Insert(x, o) for *general positions* in the scene, i.e., positions that are unequal to the positions of the objects. The essential three means are a parameterized c-crowded scene (Section 4.3.3), the algorithm for the computation of the sectorgraph (Section 4.2.2), and the algorithm for the local range search (Section 4.2.1). The algorithms use the sectorgraph to search the objects. Some of them start the search from the *reference* to an object. References to objects are pointers to the corresponding nodes of the sectorgraph.

Algorithm 3 SEARCH(x, t): Computes the bubble $V_t(x)$.

```
Require: $\gamma$-angle graph, radius $t$, reference to object $o_s \in V_t(x)$.
Ensure: Objects of bubble $V_t(x)$ are given as data structure $G_{V_x(t)}$.
  Compute $V_{t+d}(o_s)$ by algorithm LRS$(t + d, o_s)$;
  for all objects $o_i \in V_{t+d}(o_s)$ do
    if $|pos(o_i) - x| \leq t$ then
      $V_t(x) := V_t(x) + o_i$;
    end if
  end for
```

Search

Let x be the centre of a bubble $V_t(x)$ to be computed. If a single reference to one object $o_s \in V_t(x)$ of the sectorgraph is given, we execute the SEARCH(x, t) operation as follows: we execute the local range search of Section 4.2.1 in which the reference to the object o_s is used as the starting point. In order to find all objects of $V_t(x)$, we must compute $V_{t+d}(o_s)$ where $d = dist(x, o_s)$ is the distance between position x and object o_s, and $d < t$ holds (see Fig. 4.10). Afterwards, we discard all objects $V_{t+d}(o_s) \setminus V_t(x)$. A detailed description of the operation is shown in Algorithm 3.

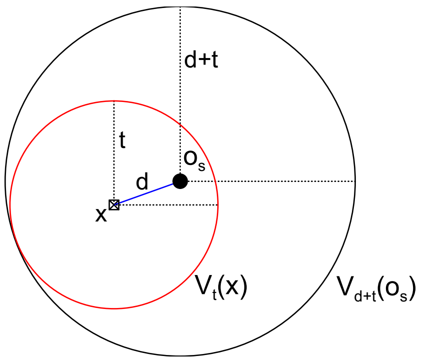


Figure 4.10: SEARCH(x, t) operation: in order to find the objects of the bubble $V_t(x)$ we must compute the bubble $V_{t+d}(o_s)$ starting a search from object o_s.

Theorem 13 (operation search). *Let $k = \frac{2\pi}{\gamma} \in \mathbb{N}, k \geq 6$, t the radius of a bubble, and let x, y be two positions of a bubble where $\Delta x = dist(x, y)$. We consider a (γ, c)-crowded scene of n objects in $\mathbb{R}^2$ organized as a sectorgraph where $t \gg c$. The* SEARCH(x, t) *operation can be done in time $O(|V_{2f_\gamma t}(o_s)|) = O(t^2)$ if an object $o_s \in V_t(x)$ is given.*

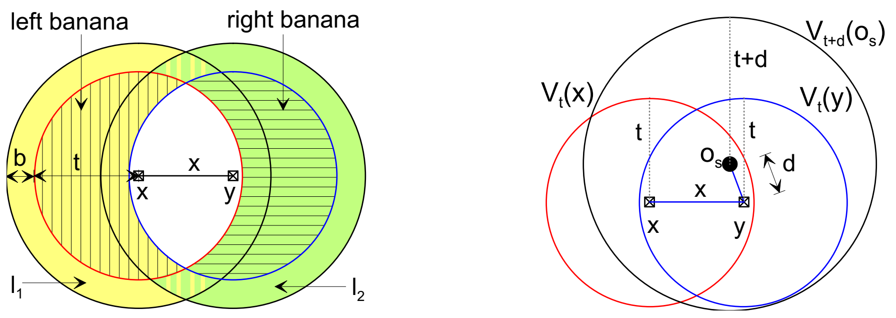


Figure 4.11: The MOVE$(x, \Delta x)$ operation must insert objects of the right banana (horizontal hatching) and delete the objects of the left banana (vertical hatching).

The algorithm is $2f_\gamma$-neighbourhood sensitive. By corollary 11, we can strengthen the neighbour factor up to $2 + \frac{f_\gamma \cdot c}{t}$. The data structure needs space $O(n)$ for all operations.

Proof. By the $\texttt{LRS}(t + d, o_s)$ algorithm, we search all objects of $V_{t+d}(o_s)$ and some, not necessary all, objects of $V_{f_\gamma(t+d)}(o_s)$. Due to $o_s \in V_t(x)$, it holds $d = dist(x, o_s) \leq t$. Hence, $\texttt{LRS}(t + d, o_s)$ execute a breadth first search up to a distance of $f_\gamma 2t$, so that the running time $O(|V_{2f_\gamma t}(o_s)|) = O(t^2)$ and the neighbour factor $2f_\gamma$ hold. The iteration of $V_{t+d}(o_s)$ and the removal of the objects $V_{t+d}(o_s) \backslash V_t(x)$ cost time $O(t^2)$, too. By corollary 11, the $\texttt{LRS}(t + d, o_s)$ algorithm searches the objects only up to a distance of $2t + f_\gamma c$, so that the neighbour factor $2 + \frac{f_\gamma c}{t}$ results. □

Movement

The visitor of the scene uses the MOVE$(x, \Delta x)$ operation in order to move the bubble from position x to y where $\Delta x = y - x$ is the distance of the movement. Two parts of the bubble $V_t(x)$ and of the data structure $G_{V_t(x)}$ must be updated, namely the two *bananas*, shown in the left illustration of Fig. 4.11. The *left banana* (vertical hatching) contains the objects to be removed from the bubble and the *right banana* (horizontal hatching) contains the objects to be inserted; both bananas are of width Δx. As our local range search algorithm must search up to a distance of $\Delta x + b$ (for some fixed b), the area to be searched is a little bit larger (yellow and green area of Fig. 4.11). We describe two algorithms for the computation of the two bananas and the bubble $V_t(y)$, respectively.

The first algorithm of the MOVE$(x, \Delta x)$ operation (see Algorithm 4) is similar to the algorithm of the SEARCH(x, t) operation. We perform a local range search $\texttt{LRS}(t+d, o_s)$, starting from the object $o_s \in V_t(x)$ that is next to the position y, in order to search all

Algorithm 4 MOVE$(x, \Delta x)$: Moves the bubble $V_t(x)$ from position x to y.

```
Require: The $\gamma$-angle graph, radius $t$, bubble $V_t(x)$.
Ensure: The bubble $V_t(y)$ is given as data structure $G_{V_t(y)}$.
  By iteration of $V_t(x)$, compute the object $o_s$ next to position $y$;
  $d := dist(o_s, y)$;
  Compute $V_{t+d}(o_s)$ by algorithm LRS$(t + d, o_s)$;
  for all objects $o_i \in V_{t+d}(o_s)$ do
    if $dist(o_i, y) \leq t$ then
      $V_t(y) := V_t(y) + o_i$;
    end if
  end for
```

objects of $V_{t+d}(o_s)$ where $d = dist(o_s, y)$ is the distance between object o_s and position y (see the right illustration of Fig. 4.11). We must search all objects within distance $3 \cdot t$ because $d \leq 2t$ holds. To get the object o_s, we can iterate the objects $o_i \in V_t(x)$ and check if o_i is the object next to position y. A detailed description is shown in Algorithm 4.

The running time of our first MOVE$(x, \Delta x)$ algorithm (Algorithm 4) is independent of the distance Δx of the movement. If Δx is small compared to t, the running time is not linear in terms of the moved area, i.e, the size of left banana $|V_t(x) \backslash V_t(y)|$ and the size of right right banana $|V_t(y) \backslash V_t(x)|$. Therefore, for small values Δx, we use a second algorithm that achieves a running time depending on the objects of the left and right banana only.

We slightly modify the local range search algorithm LRS(t, o_s) (see Algorithm 1). The algorithm works as follows (see Fig. 4.12). To get the new bubble $V_t(y)$, we must compute the right and left banana. We insert the objects of the right banana into $V_t(x)$ and remove the objects of the left banana from $V_t(x)$ resulting in the bubble $V_t(y)$. To compute the objects of the right banana we must search up to a distance $\Delta x + c$. Therefore, we search the objects of $V_{t+\Delta x+c}(x)$ starting a local range search from the *border* of the bubble $V_t(x)$. The border of $V_t(x)$ is defined as follows: $B_t(x) = \{v \in V_t(x) \mid \exists\ (v, u) \in E_\gamma : u \notin V_t(x)\}$. In other words, the border is a subset of the bubble $V_t(x)$, consisting of nodes that have at least one edge pointing to a neighbour outside of the bubble $V_t(x)$. Searching up to a distance $t + \Delta x + c$ ensures that we get the objects of the right banana. To get the objects of the left banana, we could iterate the objects $V_t(x)$, but this would cost time $O(t^2)$. Therefore, we modify our local range search algorithm such that we avoid the search of inner objects of $V_t(x)$, namely the objects $V_t(x) \cap V_t(y)$. The input of the modified algorithm LRS$(t, \Delta x, c, B_t(x))$ are the

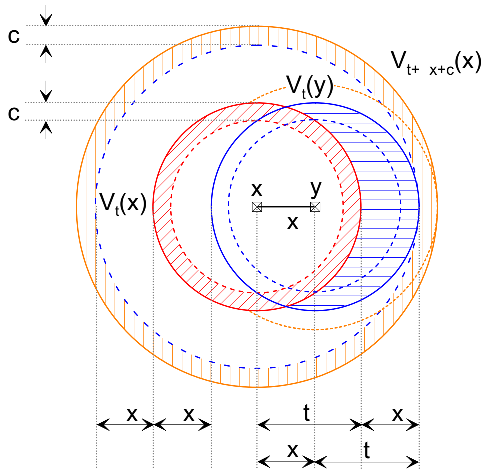


Figure 4.12: The movement of the bubble from position x to y. The objects to be inserted are in the right banana (horizontal blue hatching) and the objects to be deleted are in the left banana. The objects $V_{t+\Delta x+c}(x)\backslash(V_t(x) \cap V_t(y))$ must be computed in order to get the objects of the left and right banana.

radius of the bubble t, the distance Δx of the movement, the crowdedness c, and the border $B_t(x)$ of the bubble $V_t(x)$. The detailed description is shown in Algorithm 5.

Theorem 14 (operation movement). *Let $k = \frac{2\pi}{\gamma} \in \mathbb{N}, k \geq 6$, t the radius of a bubble, and let x, y be two positions of a bubble where $\Delta x = dist(x, y)$. We consider a (γ, c)-crowded scene of n objects in $\mathbb{R}^2$ to be organized as a sectorgraph where $t \gg c$.*

If $V_t(x)$ is given as data structure $G_{V_t(x)}$, the Move$(x, \Delta x)$ *operation can be done in time $O(|V_{3f_\gamma t}(o_s)| + |V_t(x)|) = O(t^2)$ by Algorithm 4 where $o_s \in V_t(x)$ is the object that is next to position y. The algorithm is $3f_\gamma$-neighbourhood sensitive. With corollary 11, we can strengthen the neighbour factor up to $3 + \frac{f_\gamma \cdot c}{t}$.*

If $V_t(x)$ is given as data structure $G_{V_t(x)}$, the Move$(x, \Delta x)$ *operation can be done in time $O(t \cdot \Delta x + \Delta x^2)$ by Algorithm 5.*

Proof. Move$(x, \Delta x)$ operation by Algorithm 4: The iteration of the bubble $V_t(x)$, to compute the object o_s that is next to position y, costs time $O(|V_t(x)|) = O(t^2)$. Since $o_s \in V_t(x)$, it holds $d = dist(y, o_s) \leq 2t$. Hence, the `LRS`$(t + d, o_s)$ algorithm performs a breadth first search up to a distance of $f_\gamma 3t$, so that the running time $O(|V_{3f_\gamma t}(o_s)|) = O(t^2)$ and the neighbour factor $3f_\gamma$ results. The iteration of $V_{t+d}(o_s)$ and the removal of the objects $V_{t+d}(o_s)\backslash V_t(y)$ costs time $O(|V_{t+d}(o_s)|) = O(|V_{3t}(o_s)|) = O(t^2)$, too. By corollary 11, the `LRS`$(t + d, o_s)$ algorithm searches for the objects only up to a distance

Algorithm 5 Move$(x, \Delta x)$: Moves the bubble $V_t(x)$ from position x to y.

```
Require: The $\gamma$-angle graph, crowdedness $c$, radius $t$, bubble $V_t(x)$, border $B_t(x)$.
Ensure: The bubble $V_t(y)$ given as data structure $G_{V_t(y)}$.
  Compute $V_{t+\Delta x+c}(x)\backslash(V_t(x) \cap V_t(y))$ by algorithm LRS$(t, \Delta x, c, B_t(x))$:
    1. The breadth first search (BFS) starts with a queue that is filled with all objects
       of the border $B_t(x)$.
    2. Restrict the BFS to the upper distance $t + f_\gamma(\Delta x + c)$, such that no objects
       outside of $V_{t+f_\gamma(\Delta x+c)}(x)$ are visited.
    3. As "lower distance", restrict the breadth first search to objects $o_i$ where
       $dist(o_i, y) \geq t - c$ and $dist(o_i, x) \geq t - c$ holds.
  $V_t(y) := V_t(x)$ {reference only, no copy of elements}
  for all objects $o_i \in V_{t+\Delta x+c}(x)\backslash(V_t(x) \cap V_t(y))$ do
    if $dist(o_i, x) > t$ and $dist(o_i, y) \leq t$ then
      $V_t(y) := V_t(y) + o_i$; {Add the objects of the right banana to the bubble}
    end if
    if $dist(o_i, x) \leq t$ and $dist(o_i, y) > t$ then
      $V_t(y) := V_t(y) - o_i$; {Remove an object of the left banana from the bubble}
    end if
  end for
```

of $3t + f_\gamma c$, so that we can strengthen the neighbour factor to $3 + \frac{f_\gamma c}{t}$.

Move$(x, \Delta x)$ operation by Algorithm 5: in fact, our modified local range search algorithm LRS$(t, \Delta x, c, B_t(x))$ searches through an area that is similar to a degenerated torus. The outer radius is $t + \Delta x + c$ and the inner "radius" is a shape formed by $V_t(x) \cap V_{t-c}(y)$. The algorithm starts the search from the boundary $B_t(x)$ of $V_t(x)$. The objects of the boundary $B_t(x)$ must lie in $V_t(x) - V_{t-c}(x)$, i.e., all objects lie in a torus that has centre x, inner radius $t - c$, and outer radius t. Thus, the size of $B_t(x)$ is bounded by $O(t)$ since c is a constant and sufficiently small compared to t. The time of the Move$(x, \Delta x)$ operation depends on the size of the degenerated torus. The witdh of the degenerated torus is smallest, $\Delta x + 2c$, at the right banana, and at most, $2\Delta x + 2c$, at the left banana. So we get the running time of $O(t \cdot \Delta x + \Delta x^2)$. □

4.4.3 Algorithms for Insertion and Deletion

We describe the algorithms for the manipulation of the scene. The modeller inserts objects in the scene using the Insert(x, o) operations and deletes objects from the scene using the Delete(x) operations. The modeller must stand near the position of the objects to be deleted. The modeller inserts the object at his current position. The operations have the same difficulties as the movement operations because our local range search algorithm starts the search from an object of the scene. Additionally, the following problems must be solved.

The deletion of an object is simple, but the update of the graph is difficult. The update consists of two steps. First, we compute all objects that point to the deleted object. This can be achieved easily if each object stores an indegree list. If we do not use indegree lists, we compute the objects to be updated by a local range search started from the object to be deleted. We must search sufficiently widely in order to find all objects. The second step is to compute the new neighbours of the objects. This can be achieved by a local range search, where we search all objects of the neighbourhood. Afterwards, we use the algorithm for the construction of the sectorgraph (see Algorithm 2) in order to update the edges of the objects pointing to the object to be deleted.

The insertion of an object works similarly, but it is harder. We must compute all neighbouring objects that get new edges pointing to the inserted object. The first problem is that we have no object that we can start our search from in order to explore the neighbourhood. The second problem is that we do not know how wide we must search in order to find all objects. If the scene is c-crowded and we are inside the scene, the answer is easy. Nevertheless, if we are at the border of the scene, we have difficulties because we do not know the shape of the border.

Delete

Let x be the position of the object to be deleted. The algorithm of the DELETE(x) operation consists of two steps: first, we delete the object and its outgoing edges. Second, we compute the set U of objects that have an edge pointing to the object x and we redirect the edges $sec_u(x)$ for each $u \in U$ to the new closest neighbour of u (see left illustration in Fig. 4.13).

As our scene is (γ, c)-crowded, the objects u to be updated are a subset of $V_c(x)$, i.e., $U \subseteq V_c(x)$ holds. The set U is even smaller, because at most $O(c)$ objects in the set $V_c(x)$ can have x as their closest neighbour in one of the k directions of the sectors, i.e., $|U| = O(c)$ holds. The worst case would be to cluster the border of $V_c(x)$ with objects u so that each object points to object x. As the density of the objects cannot be arbitrarily high, the condition $|U| = O(c)$ follows.

After the computation of the set U, the second step of the DELETE(x) algorithm starts. We recompute all edges $sec_u(x)$ for each $u \in U$. The new closest neighbour n of the object u is at most at distance c from x or there is no other object in the sector $sec_u(x)$ because the scene is (γ, c)-crowded (see a more detailed proof in theorem 15).

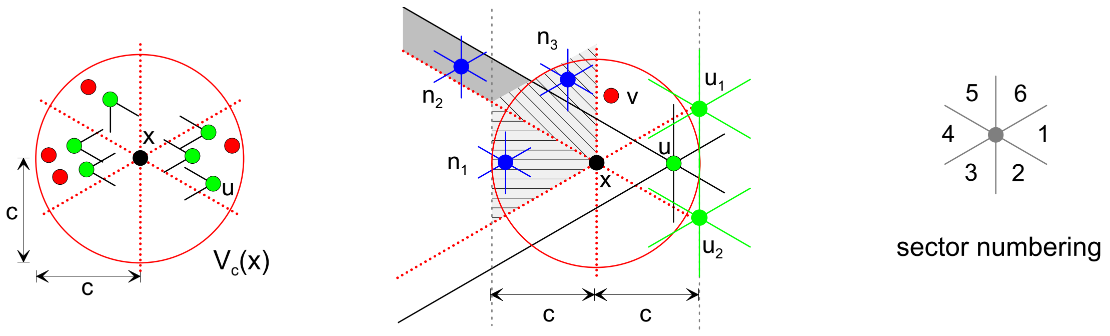


Figure 4.13: We must search the neighbourhood $V_c(x)$ of the object x in order to find all objects $u \in U$ (green balls) that have edges pointing to the object x (left). The new next neighbour of object u in sector $sec_u(x)$ is at most at a distance of c or the sector is empty (right).

Thus, when we search all objects $V_c(x)$ and afterwards, we use the algorithm for the computation of the sectorgraph $\texttt{CSG}(V_c(x))$ (see Algorithm 2) in order to recompute the edges. However, we slightly modify Algorithm 2. The algorithm must retain unchanged both sectors of objects $u \notin U$ as well as sectors s of objects $u \in U$ where $s \neq sec_u(x)$. The algorithm of the DELETE(x) operation is sketched in Algorithm 6.

Algorithm 6 DELETE(x): Deletes the object x from the bubble $V_t(x)$.

Require: The γ-angle-graph, radius t, reference to the object $x \in V_t(x)$ to be deleted, object x is centre of the bubble.

Ensure: The bubble $V_t(x) \backslash x$.

```
Compute $V_c(x)$ using the local range search algorithm $\texttt{LRS}(c, x)$;
for all objects $u \in V_c(x)$ do
  if $u$ has an edge pointing to $x$ then
    $U := U + u$;
  end if
end for
For all $u \in U$ recompute all edges $sec_u(x)$ by the algorithm $\texttt{CSG}(V_c(x))$;
```

Theorem 15 (operation deletion). *Let $k = \frac{2\pi}{\gamma} \in \mathbb{N}, k \geq 6$, $c \leq t$ be the radius of a bubble, and let x be the object to be deleted. We consider a (γ, c)-crowded scene of n objects in $\mathbb{R}^2$ organized as a sectorgraph. The* DELETE(x) *operation can be done in time $O(c^2 \cdot \log(c))$ if a reference to the object x is given and the position x of the object is the centre of the bubble $V_t(x)$.*

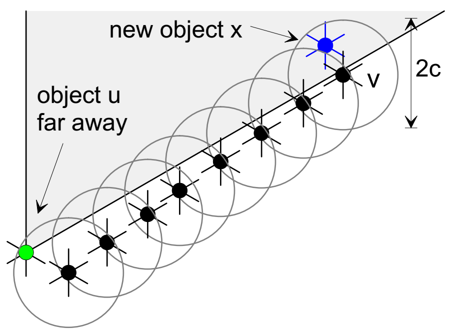
Proof. We compute all objects of $V_c(x)$ using the $\texttt{LRS}(c, x)$ algorithm. The algorithm performs a breadth first search at a distance of $f_\gamma c$ so that the running time $O(c^2)$ holds. The redirection of the edges by the algorithm $\texttt{CSG}(V_c(x))$ costs time $O(c^2 \cdot \log(c))$ because all $O(c^2)$ objects of $V_c(x)$ are possible neighbours.

It remains to be shown that all possible new neighbours n of the objects $u \in U$ are placed at a maximum distance of c. First, we look at the case of a graph with $\gamma = 60°$, i.e., an object has six sectors. See the middle illustration of Fig. 4.13; a numbering of the sectors is shown in the right illustration of Fig. 4.13. The result of other values γ follows if $k \equiv 2 \pmod 4$ holds. We look only at one single sector $sec_u(x) = 4$ of the object $u \in U$. The result for other sectors concurs because of the symmetry of the sectors. Object u points to object x in sector $sec_u(x) = 4$ if the object is placed in the triangle of the three points x, u_1, and u_2. We show that the next neighbour n of object u is at most distance c from x or there is no other object in the sector. We have to distinguish different cases depending on the possible position of the new neighbour n. First, we assume that at least one object n is placed in sector 4 of object x, i.e., $sec_x(n) = sec_u(x)$ holds and sector $sec_x(n)$ is a subset of sector $sec_u(x)$. It follows that at least one object $n = n_1$ is placed at a distance of c from x because otherwise the sector $sec_x(n)$ of object x is not (γ, c)-crowded. Assuming that sector 4 of object x is empty, we consider two further cases. In the first case, sector 5 of object x contains at least one object n. In the second case, sector 3 of object x contains a neighbour, is symmetrical. At least one object $n = n_3$ is placed at a distance of c from x because otherwise the sector $sec_x(n_3)$ of object x is not (γ, c)-crowded.

However, this observation is insufficient for the proof. In fact, we must distinguish whether the object n_3 is placed in sector $sec_u(x)$. We are finished if n_3 is in sector $sec_u(x)$. However, if n_3 is not in sector $sec_u(x)$, the following problem appears. Let $S(sec_u(x))$ be the area of sector $sec_u(x)$. Either the area $S(sec_u(x)) \cap S(sec_x(n_3))$ is empty, then we are finished, or one object $n_2 \in S(sec_u(x)) \cap S(sec_x(n_3))$ exists that is placed far away from object x (see n_2 in the dark grey parallelogram of Fig. 4.13). This would not violate the crowdedness of sector $sec_x(n_3)$. However, this object must be placed at a distance of c because otherwise the sector $sec_{n_2}(x)$ of object n_2 is not (γ, c)-crowded, i.e., point n_2 cannot be placed as shown in Fig. 4.13 (inside the grey parallelogram). Note that in the latter case the sector 4 of object x is empty. $\square$

Insert

Let x be the position of the object o to be inserted. The execution of the $\textsc{Insert}(x, o)$ operation consists of two steps. First, we have to insert the object and to compute its outgoing edges to the closest neighbours. Second, we must find the set U of objects u



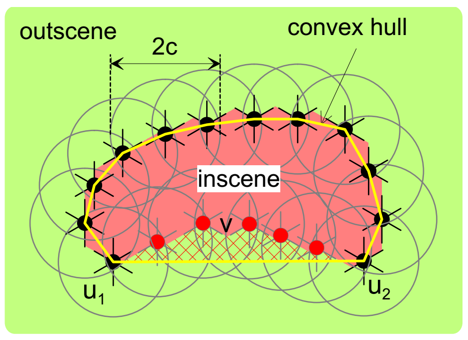


Figure 4.14: Both far away objects (e.g., u) and nearby objects (e.g., v) point to the inserted object x (left). The space of the scene is partitioned into the two parts *outscene*, and *inscene* depending on the shape of the convex hull (right).

having sectors $sec_u(x)$ that points to a closest neighbour that is farther away than the inserted object x. We redirect these edges onto the object x. As our scene is (γ, c)-crowded, the objects U are a subset of $V_c(x)$, i.e, $U \subseteq V_c(x)$ holds. Thus, we compute the set $V_c(x)$ using the $\texttt{LRS}(c, x)$ algorithm, similar to the DELETE(x) and SEARCH(x, t) operation. Afterwards, we insert the new object x and iterate all objects $u \in U$ in order to compute the closest neighbour of the object x, and to redirect the edges for each object $u \in U$.

The update of objects $u \in U \subseteq V_c(x)$ is sufficient in the majority of cases, namely for all positions "inside" a c-crowded scene where each sector has an edge. However, the update may fail if a sector points to x that does not have an edge so far. The situation arises that a far away object u points to the inserted object x lying in a sector of u that is empty so far. The left illustration of Fig. 4.14 shows a c-crowded scene consisting of the green object u and some black objects. After the insertion of the new object at position x, the object becomes a new closest neighbour for a sector of the object v and u. Only object v remains c-crowded. Object u violates the condition because it is too far away from x. Therefore, the insertion algorithm fails because it does not find all far away objects, here in the example, object u, and cannot point the edges to x.

In the following, we describe how we can identify such kinds of situations. First, we classify those parts of the scene that have objects with empty sectors. We denote the sector of an object that does not contain an object by *empty sector*. We partition the two-dimensional space of the scene in two parts (see the right illustration of Fig. 4.14): let $S_{out} = \{S_v(i) \mid \text{ sector } i \text{ of object } v \text{ is an empty sector}\} \subseteq \mathbb{R}^2$ be a region of the scene that contains no objects. We denote S_{out} by *outscene* and the rest of the scene by *inscene*, where $S_{in} = \mathbb{R}^2 \backslash S_{out}$. Accordingly, we denote all objects that have at least one empty sector by the set O_{out} and we denote the set of objects having no empty sectors by O_{in}.

Observation 16 (Property of inscene and outscene). *The space S_{in} is a single contiguous region shaped as a polyhedron. The vertices of the polyhedron are the set S_{out}. All objects O_{in} are inside the polyhedron, i.e., $\forall u \in O_{in} : u \subseteq S_{in}$*

Proof. We assume that two discontinuous regions exists and look at the leftmost objects $u \in O_{out}$ that are in the first region. Some of their sectors would contain objects of the second region and therefore, the two discontinuous regions cannot exist. All objects O_{out} lie on the polygon of the polyhedron because otherwise an object $o \in O_{out}$ inside the polyhedron would contradict the definition of S_{in}. Therefore, the shape of S_{in} is a polygon formed by piecewise parts of sector boundaries of the objects $u \in O_{out}$. The objects O_{in} must lie inside S_{in} because an object outside S_{in} would contradict the definition of the empty sectors. □

In the next step, we compare and characterize the shape of the inscene with the convex hull of the scene. Large parts of the inscene are shaped by the empty sectors of objects lying on the convex hull of the scene (see the black objects in the right ill. of Fig. 4.14). Only small parts of the inscene are lying outside the convex hull. These small parts of the inscene are made from non-empty sectors of objects of the convex hull. On the other side, some parts of the outscene lie inside the convex hull. The objects with empty sectors lying inside the convex hull contribute empty sectors to the outscene that are lying both outside as well as inside the convex hull (see the red objects and the chequered region in the right ill. of Fig. 4.14). Now we ask how far away can an object of the set O_{out} be from an edge (u_i, u_j) of the convex hull.

Observation 17. *Let (u_i, u_j) be the edge of the convex hull, h be the length of the edge, and u_i and u_j the vertices of the edge of the convex hull. The maximum distance of objects $v \in S_{out}$ from the edge of the convex hull is* $\max\{\frac{1}{2}h\tan(\gamma), \frac{h}{\tan(\gamma)}\}$.

Proof. To make the maximum distance of an object $v \in O_{out}$ from the edge of the convex hull, we must place the object v between the two objects u_i, u_j, so that at least one of its sectors looks through the two nodes u_i, u_j. Two different situations can occur depending on the orientation of the sectors. The two situations are shown in the middle and right ill. of Fig. 4.15. In the middle illustration, the vertices v_1, v_2 and v_3 can look through the two vertices u_1 and u_2, so that at least one of its sectors is empty. In the right illustration, the vertex v_4 can look through the two vertices u_3 and u_4. The maximum distance follows by the height of the triangle consisting of the vertices u_i, u_j

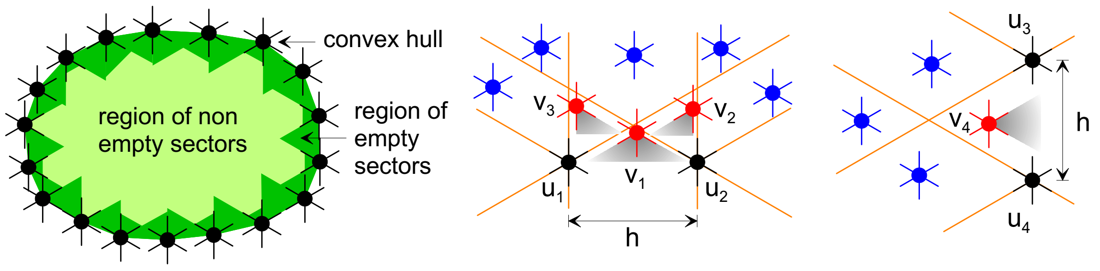


Figure 4.15: Only the sectors of objects v_i at a distance of $\max\{\frac{1}{2}h\tan(\gamma), \frac{h}{\tan(\gamma)}\}$ look through two objects u_j.

and the sector boundaries. These triangles shape a zone near the convex hull of the scene. The left ill. of Fig. 4.15 shows an example of the zone. Inside this zone, we can place objects that get empty sectors. □

The insertion of objects in the inscene is simple. We describe the algorithm for the INSERT(x, o) operation if the object x to be inserted is in the inscene, i.e., we know that there are no far away objects that will get the object x as their next neighbour. The algorithm is sketched in Algorithm 7.

Theorem 18 (operation insertion). *Let $k = \frac{2\pi}{\gamma} \in \mathbb{N}, k \geq 6$, $c \leq t$ be the radius of a bubble, and let x be the position of a object to be inserted where x is the centre of the bubble. We consider a (γ, c)-crowded scene of n objects in $\mathbb{R}^2$ organized as a sectorgraph. The* INSERT(x, o) *operation can be done in time $O(c^2)$.*

Proof. We can insert a new object without any problems in the inscene. It is sufficient to search and update all objects of $o_i \in V_c(x)$ as described in beginning of this section. It holds $U \subseteq V_c(x)$ if $V_c(x)$ is "inside" a (γ, c)-crowded scene. All objects of the inscene have degree k, i.e., they have no empty sectors and each sector of the object points to another object. The running time follows from the local range search of the bubble $V_c(x)$ and the iteration of all objects in order to update the edges. □

The insertion of objects into the outscene is a more difficult case because in the worst case, we must search far away to find the objects that will point to the inserted new object (see the example of Fig. 4.14). The first question is, how can we identify such a situation? If we search the bubble $V_t(x)$ and find a ball in each sector of x, we can assume that no far away objects can point to x because the objects of the sector work as so called *blocking balls* (see the left ill. of Fig. 4.16). A set of blocking balls prevent objects on one side of the set from pointing to an object x that is placed on the opposite

Algorithm 7 INSERT(x, o): Inserts an object at position x into the bubble $V_t(x)$ if x is also the centre of the bubble.

```
Require: The $\gamma$-angle-graph, radius $t$, object $x$ to be inserted.
Ensure: The bubble $V_t(x) \cup x$.
  Compute the object $o_s$ that is next to position $x$;
  $d := dist(o_s, x)$;
  Create a node for object $x$;
  Compute $V_{c+d}(o_s)$ by algorithm LRS$(c + d, o_s)$;
  for all $o_i \in V_{c+d}(x)$ do
    if $o_i$ is next neighbour of $x$ in sector $sec_x(o_i)$ then
      edge $sec_x(o_i) := o_i$;
    end if
    for all sector $i$ of $o_i$ do
      if $x$ is next neighbour of $o_i$ in sector $sec_{o_i}(x)$ then
        edge $sec_{o_i}(x) := x$;
      end if
    end for
  end for
```

side. The set of blocking balls v prevents objects u_1 from pointing to object x. Between the blocking balls and the *blocking line* is a small area where an object can look through the blocking balls, so that it can point to x.

However, it is insufficient to examine only some balls of one sector in order to exclude that other objects of the same sector do not point to x. In the middle ill. of Fig. 4.16, the two objects v work as blocking balls for the new object x. However, the border of the sector is critical because a far away object u can point to x. In the situation

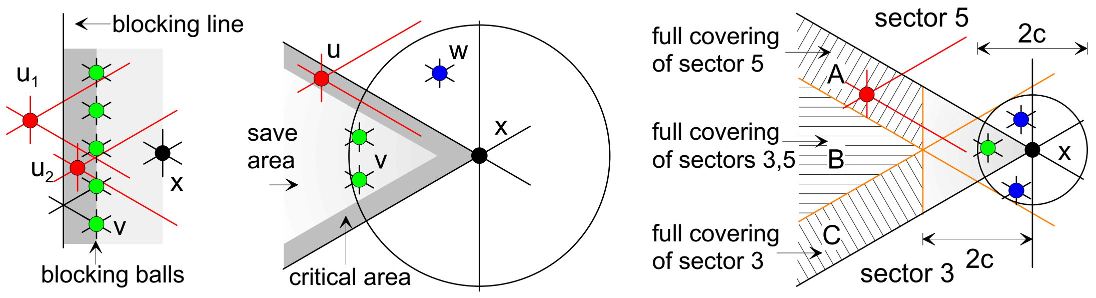


Figure 4.16: Blocking balls prevent distant objects from pointing to object x.

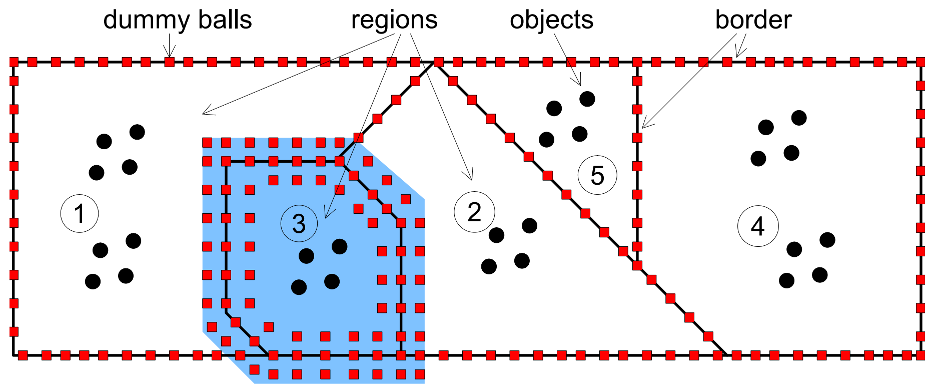


Figure 4.17: The composing of a large scene. Five regions are modelled and managed by different groups. Three rows of dummy balls are placed at the border of the regions.

in the example, the object u cannot point to x because a neighbouring sector contains the object w. Therefore, we must search in the direction of sectors of the object x that have no objects in the bubble $V_t(x)$ in order to find far away objects. The right ill. of Fig. 4.16 shows that a far away object of sector four cannot point to object x if the neighbouring sectors three and five contain objects. The areas A, B and C cover the corresponding sectors.

4.4.4 Composing a Large Scene

We outline how one can construct a virtual scene that is divided into independent regions. A *region* is a part of the scene that is modelled and stored by a group of modellers. For example in Fig. 4.17, the scene consists of five regions. Each region is bounded by a *border* (black line). The region can be arbitrarily shaped.

Many *groups* work on the same virtual scenes. A group can be a firm or a company. Therefore, each group manages and stores the objects of its region on its own hardware system. Thus, the spatial partition of the regions corresponds to the groups and their storage. Typically, the virtual scene consists of more than the five regions and groups, as shown in the example. The system is extensible, i.e., it is possible to extend the virtual scene whereby we must update the sectorgraph only for a few groups. We allow only updates of regions that are next to the inserted region. For example, if region 3 of Fig. 4.17 is newly inserted, only updates of regions 1 and 2 are necessary.

Our weak spanner fulfils these requirements because the locality of the data is the same as the spatial locality of the scene. We use dummy points (see Section 4.3.3) in order to break long edges of the weak spanner construction. Lemma 10 shows that for

$\gamma = 60°$, a long edge cannot cross more than two grid cells of a grid with edge length d. Thus, our strategy is as follows. We place a zone that consists of three parallel lines of dummy points (red points in Fig. 4.17) at the border of each region. Because of lemma 10, we know that the edges of the objects of the neighbouring regions cannot cross the zone. Therefore, after the insertion of a new region, we must only recompute the sectorgraph for the objects of the border of neighbouring regions and we must compute the sectorgraph for the objects of the new region. In Fig. 4.17, we compute only the sectorgraph for the objects of the blue area if region 3 is newly inserted.

4.5 Summary and Discussion

If our visitor wants to visit a far away part of the scene, he must move from his current position to a new position, i.e., he cannot beam to a far away position and must walk the whole way. In most applications, the user moves mostly through the scene without beaming to arbitrary positions because most users are unfamiliar with a beaming function, and thus, beaming would confuse the visitors, such that they would lose their way in the scene. Beaming to every desired position can easily be achieved with a hierarchical data structure.

In this work, we take advantage of this restriction. We use the weak spanner to connect two independently modelled parts of the scene. All we need is a recomputation of the graph at the border of the scene parts. This is difficult for a hierarchical data structure because the hierarchy must be updated and, possibly, parts of the hierarchy must be rebuilt depending on the type of data structure. Thus, two far away modellers can make their insertions and deletions independently of each other. Furthermore, it is possible to partition a scene into arbitrary tiles, each modelled by a distinct firm. Each firm stores the tiles on its own server. To connect all parts to a single large scene, neighbouring firms must only know the objects of their borders in order to compute the edges pointing to the objects of the neighbouring firm. If we use a hierarchical data structure, we have at least one central instance, namely the top of the hierarchy, that knows the position of all objects. This central instance must be updated if the scene changes. In our system, we can exchange one part of the scene (e.g., a firm) for another without notifying most of the tiles stored by other firms.

A further property of our model is the bounded density of the objects in the scene. Typically, virtual scenes do not need an arbitrarily high point density. A high density implies that at some positions of the virtual scene we have a very high number of objects. However, in the real world, all objects have an extent, and typically, they are not arbitrarily small. On a fixed space, we place only a fixed number of objects.

Therefore, our abstraction of a scene seems more reasonable than a scene model with an arbitrarily high density of objects. The bounded density is a necessary condition for the proofs of our runtime analysis. At this point, we see that a useful and practical restriction of the model leads to provably correct algorithms.

The main objective of the model and system design is to construct a system that offers completely independent work by many users in the scene. For this, we require the locality of the data structure and present a class of data structure, namely the weak spanner, which fulfils this requirement. The question remains, what is the price for this degree of flexibility? The first restriction is the abandonment of point location. Second, we show that we need some kind of parameter for the density of the scene. For this, we introduce crowded scenes and the parameter c. Without this parameter, the insertion and deletion of objects remains difficult. The search and move operations work well without the requirements of a c-crowded scene.

5 Implementation and Evaluation

This chapter consists of three parts. Section 5.1 describes the functionality of the system implemented. Section 5.2 briefly introduces the details of implementation. We confine ourselves to a short description of the software layers and the interfaces of the system. The third and last part of the chapter (Section 5.3 and et seqq.) contains our evaluation of the system.

Goals of our Evaluation

Our evaluation aims at the investigation of the performance of the MOVE$(x, \Delta x)$, SEARCH(x, t), INSERT(x, o), and DELETE(x) operations: for practical usage of the system, the execution of the operations must be fast enough for interactive movements and manipulations. The operations are implemented for the storage types `TuMemory`, `TuNetwork`, and `TuDisk`. Operations executed on a `TuMemory` bubble are computed in the main memory. Operations executed on a `TuDisk` bubble are computed on the local hard disk, i.e., accesses to objects and edges are executed by loading the data from the hard disk. Operations executed on a `TuNetwork` bubble are computed on remote hard disks, i.e., accesses to objects and edges are executed by loading the object from the remote hard disk via TCP. We investigate the performance of the operations for all storage types.

We analyse the basic algorithms used in DELETE(x), SEARCH(x, t), INSERT(x, o), and MOVE$(x, \Delta x)$ operations (see Section 4.4). For this we proceed as follows: all four methods are composed of two basic algorithms of the sectorgraph (Section 4.2). The one is the algorithm to construct a scene (see Section 4.2.2), and the other one is the breadth first search algorithm to execute a circular range query (see Section 4.2.1). The breadth first search algorithm is used in all four operations. The construction of a sectorgraph is only used in INSERT(x, o) and DELETE(x) operations and to construct a sectorgraph from scratch. Therefore, we investigate the construction time of the scene in Section 5.4, depending on our three storage types. The breadth first search algorithm is important for the MOVE$(x, \Delta x)$ operation. Therefore, we investigate this operation in the context of the implementation of the 3D viewer in Section 5.5. We measure the speed of motion of a user navigating through the scene.

5.1 Functionality and User Interfaces

We describe the functionality of the software implemented and demonstrate the construction of a virtual scene (2D viewer, Section 5.1.1), the usage of the bubbles, and the generation of a bubble hierarchy (Section 5.1.2). We show by means of examples that our implementation supports the generation of an arbitrary hierarchy of bubbles that works as a *spatial hierarchy of caches*. Furthermore, we demonstrate how viewer and modeller walk through a virtual scene (3D viewer, Section 5.1.3).

5.1.1 Scene Construction and 2D Viewer

We implemented two ways of creating a virtual scene. One possibility is to start with an empty scene, so that the modeller inserts objects into the scene one after another. An alternative is to start with a predefined scene consisting of the 3D data and the positions of the objects. In a preprocessing step, we compute the weak spanner for the whole scene from scratch with the algorithm presented in Section 4.2.2.

Both objects, as well as the weak spanner, are stored and distributed across a network. The network consists of a number of workstations that we denote as *manager* (see Fig. 5.1). Each manager stores a part of the scene. Notice that the position of the objects does not depend on the servers where the objects are stored. With other words, objects, placed at the same position, can be stored either on the same server or on distinct servers. Either the manager stores the scene temporarily in his main memory or permanently on his local hard disc; we use the latter configuration for all our measurements). Each object consists of a position and a polygonal 3D-model. We do not use any instantiation of the objects [233].

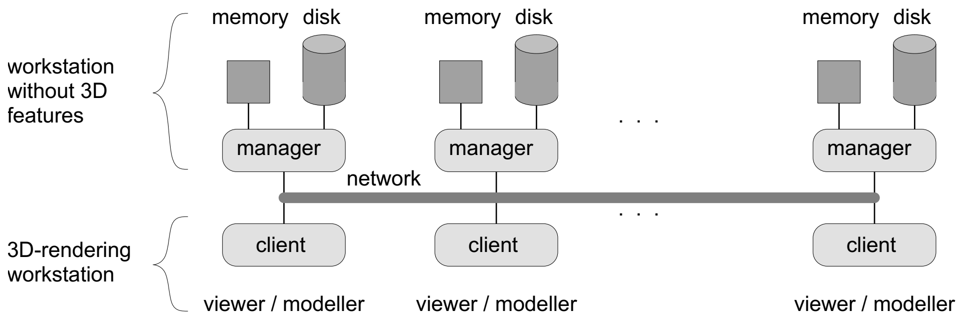


Figure 5.1: Viewer and modeller use a rendering workstation (client) to walk through the scene. The scene and the sectorgraph is distributed across a network of workstations (managers) stored on local hard disks.

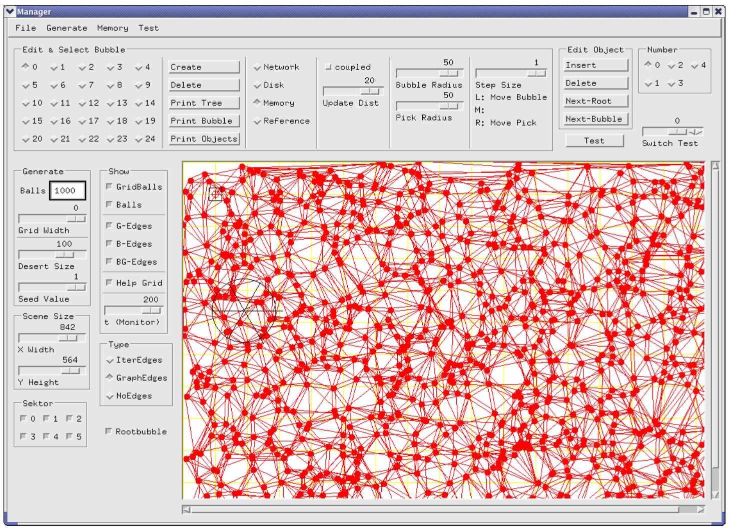


Figure 5.2: The graphical user interface of the 2D viewer. The drawing widget visualizes the sectorgraph consisting of vertices (red points) and edges (red lines). The top part of the window contains control widgets for the generation of an arbitrary bubble hierarchy, and for the insertion and deletion of objects. The left part of the window contains widgets to configure display options.

We implemented a 2D viewer and a 3D viewer for the navigation and manipulation of the scene (the latter is described in Section 5.1.3). The 2D viewer executes the following tasks:

1. Generation of a predefined scene from scratch.
2. Visualization of the data structure, i.e., vertices and edges of the sectorgraph.
3. Real-time generation of an arbitrary bubble hierarchy.
4. Visualization of the bubble hierarchy.

Figure 5.2 shows a screenshot of the graphical user interface of the 2D viewer. The drawing widget visualizes the sectorgraph, which consists of vertices (red points) and edges (red lines). The top part of the window contains control widgets for the generation of an arbitrary bubble hierarchy (see Section 5.1.2) and for the insertion and deletion of objects. The left part of the window contains widgets to configure display options.

We designed the system such that only the rendering clients (3D viewer see Section 5.1.3) need the ability to process the 3D file format of the objects in order to render the

3D data. The managers, as well as the 2D viewer, treat the polygonal 3D data of the objects as an abstract block of data. Therefore, the system is able to mix different 3D file formats of the objects without any changes to the implementation of the managers and the 2D viewer. Furthermore, it is conceptually possible to model different parts of the scene using different file formats. The 3D clients can visit only those parts of the scene whose file formats they can render. Each object of the 2D viewer that is visualized as a red point contains the polygonal 3D data.

5.1.2 Generation of an Arbitrary Bubble Hierarchy

In this section, we demonstrate the capability and flexibility of the system. Our system is able to slot arbitrary bubbles of different storage types into one another. First, we discuss some useful configurations for a rendering workstation. Second, we give examples of a bubble hierarchy that is created by the 2D viewer. The 2D viewer can arbitrarily slot bubbles of different storage classes (`TuNetwork`, `TuMemory`, `TuDisk`) into one another in real time. The implementation of the 3D viewer uses an arbitrary bubble hierarchy that is fixed at compile time.

Typical Configurations for a Rendering Workstation

As described in Section 3.3.4, we need four kinds of duplication for a bubble, namely `TuMemory`, `TuDisk`, `TuNetwork`, and `TuReference`, depending on the type of storage. Each of them stores the objects on a different storage type. We can slot each duplication type into each other. In the following, we discuss typical instances for a rendering workstation.

Every bubble hierarchy consists of at least one bubble, namely the root bubble. The root bubble is one of storage type `TuNetwork` if the scene is distributed across the network. The root bubble is one of type `TuDisk` if the scene is stored on a local disk. The root bubble is one of type `TuMemory` if a process generates a scene in its main memory. If the root bubble is one of storage type `TuNetwork`, we can configure different kinds of sub trees that work as a spatial hierarchy of caches.

A possible configuration is to slot a `TuMemory` bubble into a `TuNetwork` bubble (see option 1 in Fig. 5.3). The rendering workstation accesses the distributed scene using the `TuNetwork` bubble. Local references to `TuNetwork` objects are stored in the `TuReference` bubble. This option is worse because each movement of the visitor causes the `TuReference` bubble to access the `TuNetwork` bubble in order to load a new object. We must wait because of the long loading time for each new object.

A better configuration uses a bubble on the rendering workstation that is larger than the bubble that the visitor needs. We slot a `TuReference` bubble into a `TuMemory`

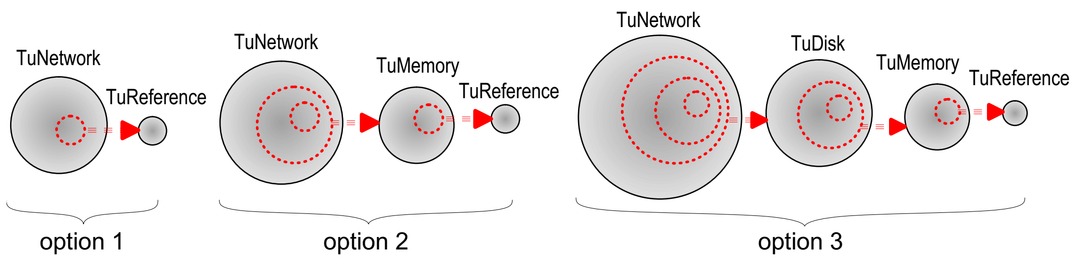


Figure 5.3: Some configurations to slot the bubbles into each other. The bubble hierarchy forms a spatial hierarchy of caches.

bubble, and the `TuMemory` bubble into the `TuNetwork` bubble (see option 2 in Fig. 5.3). The `TuMemory` bubble works as a cache for the `TuReference` bubble. The size of the cache depends on the radius of the `TuMemory` bubble. The user sees only the objects of the `TuReference` bubble. We use the `TuReference` bubble instead of a `TuMemory` bubble to avoid copies of the same object in the memory of the rendering workstation.

An alternative configuration slots a `TuReference` bubble into the `TuMemory` bubble, the `TuMemory` bubble into the `TuDisk` bubble, and finally the `TuDisk` bubble into the `TuNetwork` bubble that accesses the distributed scene. This configuration is suitable if the main memory is too small to store the objects of the scene. We use the `TuDisk` type to obtain a cache that uses the local hard disk (see option 3 in Fig. 5.3). We have a double cache consisting of the local hard disk and the main memory of the rendering workstation.

Our implementation of the bubble is able to slot arbitrary bubbles into each other. Therefore, other options are thinkable and useful, e.g., if the modeller models one part of the scene and visits another part of the scene at the same time: we slot two `TuMemory` bubbles into a `TuNetwork` bubble on the rendering workstation, one for viewing and the other one for modelling. The discussion shows that the system is flexible, so that arbitrary configurations can be constructed.

Examples of a Bubble Hierarchy Created by the 2D Viewer

Below, we show some examples (screenshots) of bubble hierarchies that are generated by our 2D viewer. Figure 5.4 shows a distributed virtual scene consisting of 5000 objects (red points) that are stored on three workstations (managers). The edges of the sectorgraph are drawn as red lines. The objects of the scene are managed by a `TuNetwork` bubble (root bubble). If we access the objects of the `TuNetwork` bubble to draw the sectorgraph, the objects must be loaded from that workstation (manager) which stores the object.

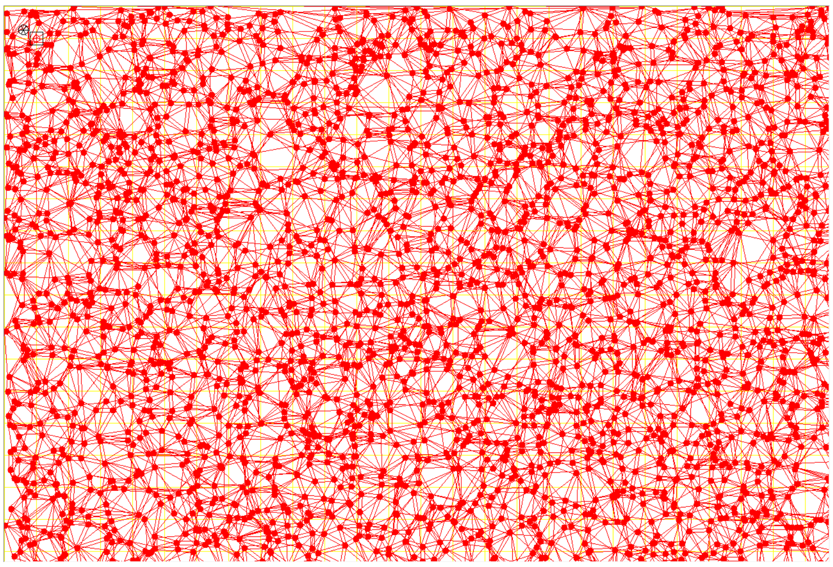


Figure 5.4: A distributed virtual scene consisting of 5000 objects (red points) that are stored on three workstations (managers). Edges of the sectorgraph are drawn as red lines. The scene can be accessed by a `TuNetwork` bubble (root bubble).

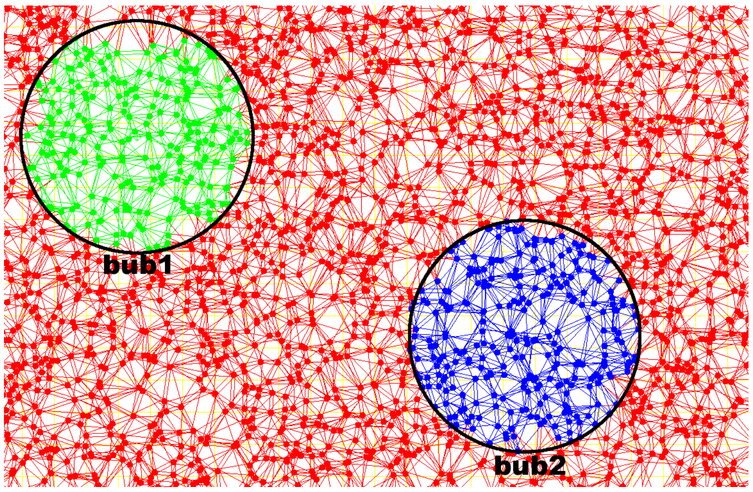


Figure 5.5: The same scene as in Fig. 5.4: additionally, two `TuMemory` bubbles (bub1, bub2) are inserted. The bubbles are children of the `TuNetwork` root bubble.

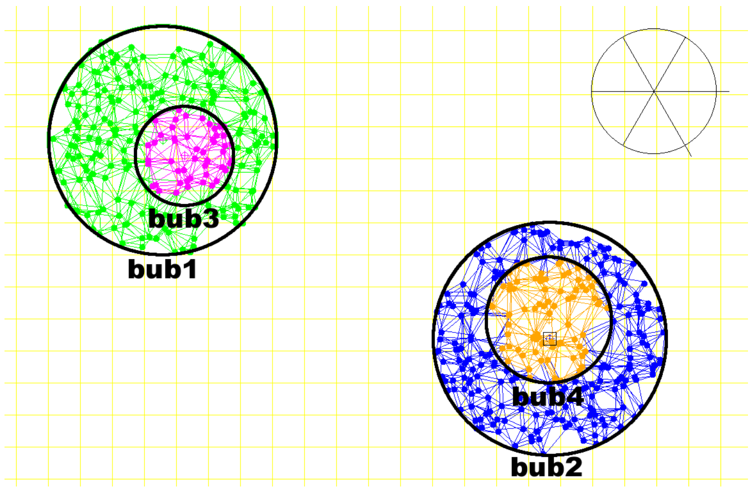


Figure 5.6: The same scene as in Fig. 5.5 (without visualizing the root bubble): additionally, a new `TuMemory` bubble (bub3) is slot into bub1, and a new `TuMemory` bubble (bub4) is slot into bub2.

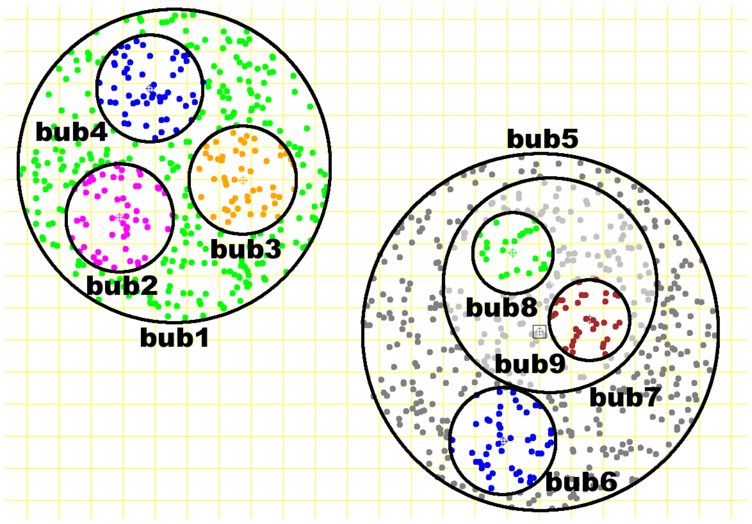


Figure 5.7: The 2D viewer generates arbitrary bubble hierarchies: the root bubble (not visualized) contains the two bubbles (bub1, bub5). Bubble bub1 contains the three bubbles (bub2, bub3, bub4). The bubble bub5 contains two bubbles (bub6, bub7). The bubble bub7 contains the two bubbles (bub8, bub9). The storage type of each bubble is arbitrary (`TuNetwork`, `TuDisk`, `TuMemory`). The bubble hierarchy forms a spatial hierarchy of caches.

Because the access to the `TuNetwork` bubble is slow, two users of the scene avoid direct access and rather slot a `TuMemory` bubble into the root bubble in order to cache a local duplication of nearby objects. Figure 5.5 shows the same scene as in Fig. 5.4. Additionally, two `TuMemory` bubbles (bub1, bub2) are inserted, one bubble for each user. The bubbles are children of the `TuNetwork` bubble (root bubble).

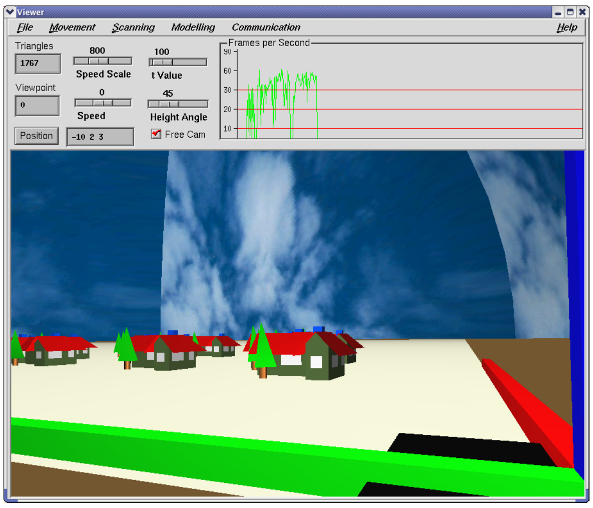
The two inserted `TuMemory` bubbles, only make fast access to the objects possible if the user does not move. For each movement of the bubble, new objects must be duplicated from the root bubble (`TuNetwork`) into the `TuMemory` bubble (bub1, bub2). The slow access to the `TuNetwork` bubble prevents smooth movement of the user. Therefore, each user slots another small `TuMemory` bubble into his bubble. Figure 5.6 shows the same scene as in Fig. 5.5 without visualizing the root bubble. Additionally, one user slots a `TuMemory` bubble (bub3) into bub1, and the other user slots a `TuMemory` bubble (bub4) into bub2. The bubble hierarchy of Figure 5.6 shows a typical configuration of two users that moves through the scene using the 3D viewer (see Section 5.1.3).

Notice that the bubbles bub1 and bub3 in Fig. 5.6 not only contain duplicated objects of the distributed scene, but also the duplicated data structure, namely the edges of the sectorgraph. Therefore, the $\textsc{Move}(x, \Delta x)$ operation of the innermost bubble bub3 uses only the cut sub graph of the preceding bubble in the hierarchy, namely bubble bub1. This is the reason why local updates of the scene do not affect the rendering workstations of far away users.

Arbitrary bubble hierarchies can be generated by the 2D viewer, Figure 5.7 shows an example: the root bubble (not visualized) contains two bubbles (bub1, bub5). Bubble bub1 contains three bubbles (bub2, bub3, bub4) and bubble bub5 contains two bubbles (bub6, bub7). The bubble bub7 contains two further bubbles (bub8, bub9). The storage type of each bubble is arbitrary (`TuNetwork`, `TuDisk`, `TuMemory`) and can be mixed in arbitrary order. The implementation can generate all thinkable combinations. The bubble hierarchy forms a spatial hierarchy of caches.

5.1.3 3D Navigation and Manipulation of the Scene

One or more users can walk through the 3D virtual scene. Each user uses a 3D viewer (graphical user interface) running on a rendering workstation (see client in Fig. 5.1). The 3D viewer uses a bubble configuration as shown in Section 5.1.2, typically at least one `TuNetwork` and two `TuMemory` bubbles. Figure 5.8 shows a screen shot of the 3D viewer. The 3D viewer stores the objects of the two `TuMemory` bubbles (see bubble bub1, bub3 of Fig. 5.6), but only the objects of the innermost bubble (bub3) are rendered and are visible to the user. The `TuNetwork` bubble holds the connection to the network and loads the required parts of the scene from the manager which stores the objects.



Triangles 1767
Viewpoint 0
Speed Scale 800
Speed 0
t Value 100
Height Angle 45
Position -10 2 3
[✓ Free Cam]

Figure 5.8: The graphical user interface of the 3D viewer.

The system manages an arbitrary number of viewers and modellers. The 3D viewer is used both by the viewer, who walks only through the scene, as well as by the modeller who inserts and deletes the objects of the scene. The position of a viewer is always the centre of his innermost bubble (see bubble bub3 in Fig. 5.6). If the viewer moves his position, the 3D viewer moves the centre of the innermost bubble. The 3D viewer moves the centre of the second `TuMemory` bubble (see bubble bub1 in Fig. 5.6) only if the innermost bubble crosses the border of the bubble bub1. To insert an object, the modeller loads a new object from the disk. The newly inserted object is placed at the current position of the modeller. In order to delete an object, the modeller must move near the position of the object. The system selects the object nearest to his current position. The insertion and deletion of the objects goes from the root bubble down to the leaves of the bubble hierarchy. We update the root bubble first and afterwards the lower levels of the bubble hierarchy. For measurements and presentations, the 3D viewer records and stores the path of motions on disk. After loading a stored path, the 3D viewer moves the position of the user automatically along the recorded path.

5.2 Implementation of the Bubbles

We describe the basic ideas for implementation. To distribute the objects across a network of workstations, we implement a virtual distributed memory. The graph, consisting

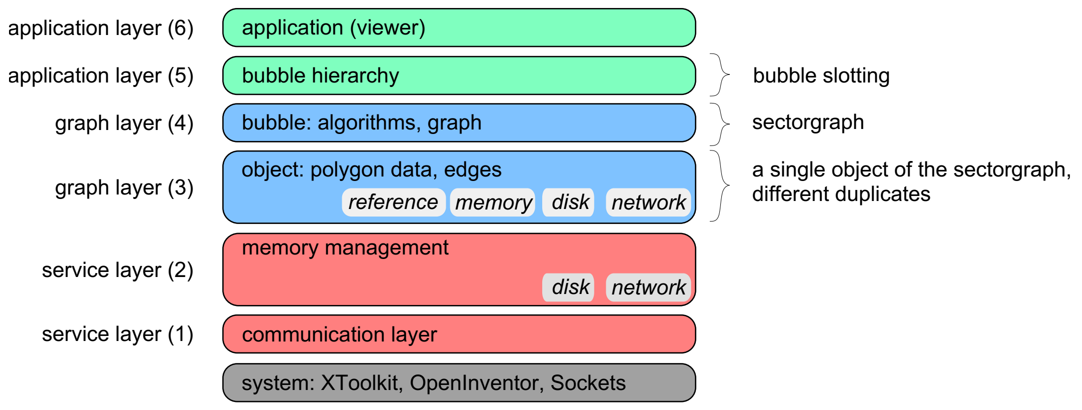


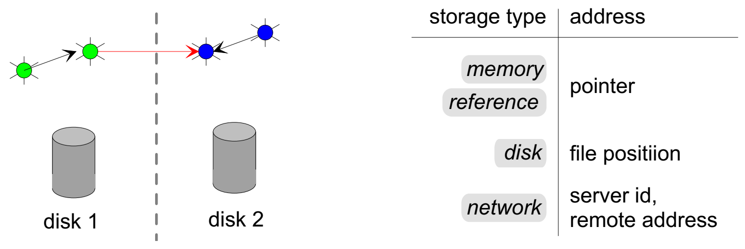
Figure 5.9: The software layer of the system. The algorithms are separated from the memory management. Layer 3 implements the virtualisation the distributed memory.

of objects and edges, is the interface of the virtualisation. Each object is implemented for `TuMemory`, `TuDisk`, and `TuNetwork`. The implementation of pointers, necessary for the edges of an object, uses an extended address space consisting of the identifier of a remote workstation, and the disk address of the remote disk. The implementation of the extended address space enables the addressing of objects that are stored on remote workstations. The access to a virtualised object hides the information about where the data of the object are stored. We describe the software layers of the system in Section 5.2.1, and the idea of the interfaces in Section 5.2.2.

5.2.1 Software Layer

The core idea of the implementation is the separation of the algorithms and the memory management. We implement the system in six distinct software layers using the system libraries at the lowest level (see Fig. 5.9). Layers 6 and 5 are the application layers. Layer 6 is the viewer that iterates all objects and renders the objects. Therefore, this layer must understand and process the file format of the 3D objects. All other layers are independent of the format of the 3D data. Layer 5 allows the construction of a bubble hierarchy. At this level, we slot a bubble into another bubble as described in the preceding Section 5.1.2. The bubble hierarchy is managed with a tree data structure.

The core of the system consists of layers one to four. Layers 4 and 3 are the graph layers. The algorithms are completely implemented in layer 4. The graph can be stored in the main memory, on the local hard disk, and across a network of workstations. Even though we have three different storage types, there is only one implementation of the



storage type	address
memory *reference*	pointer
disk	file positiion
network	server id, remote address

Figure 5.10: Four types of memory addresses. To point to a remote object, the network type consists of a server identifier and a remote disk address. The red edge points to a remote address.

algorithms, e.g., single breadth first search implementation for all three storage types. The abstract interface of the objects in layer 3 makes this possible. The algorithms do not know if they are searching a graph stored in the main memory, or if they are searching a graph stored on a remote workstation. The object layer (3) hides the storage type.

The implementation of the objects depends on the storage type. For each kind of bubble, namely the types `TuNetwork`, `TuMemory`, and `TuDisk`, we have an implementation of the object (see Fig. 5.9). The implementations lie in layer 3. In order to make it easy and fast to implement the objects, we enclose the basic operations of the communication in the communication layer, and we enclose the basic operation of the memory management for a disk and network access (`TuDisk`, `TuNetwork`) in layer two.

5.2.2 Interfaces

Our most important interface is a virtual interface to the storage management. Typically, the operating system offers a virtual interface to the main memory where parts of the memory are stored on the hard disk. The algorithms access the data without any knowledge of where the data is stored. The advantage is an easy access because of the simple interface. The disadvantage is the fact that the algorithms cannot distinguish between a cheap local access, and an expensive remote access.

In order to keep the advantage and to avoid the disadvantage for our virtualisation, we move the storage interface to the objects of the scene. As shown in Fig. 5.9, the graph layer 4 works independently of the storage type. The graph layer 3 implements four different types, namely `TuMemory`, `TuDisk`, `TuReference`, and `TuNetwork`.

In order to achieve a virtual memory interface for objects, we use a technique similarly to smart pointers, a well known C++ programming technique [9, 86]. We implemented pointer classes that correspond to the four storage types (`TuMemory`, `TuDisk`,

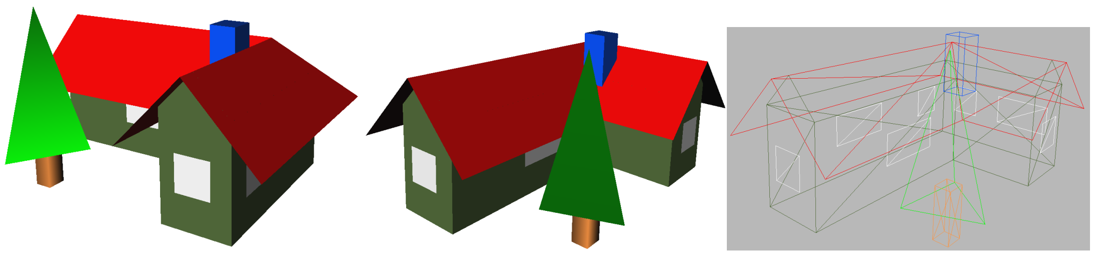


Figure 5.11: A house and tree used as objects for the benchmarks. The object consists of 61 triangles and needs 2917 Bytes storage. The object is of size 6 (depth), 8.5 (width), and 4 (height).

`TuReference`, `TuNetwork`). To implement a pointer that points to remote workstations, we extend the address space of a normal memory address. The memory address of the network type consists of a server identifier that identifies the remote server, and a remote address. The remote address can be either a disk type (file position) if the object is stored on a local hard disk, or a memory type if the object is stored in the main memory of the remote machine. Furthermore, we have a class for disk addresses, and the normal pointer addresses for the access to the main memory (see right illustration of Fig. 5.10).

The left illustration of Fig. 5.10 shows two green objects and two blue objects each stored on different hard disks connected by a network. If an algorithm accesses the red edge of the green object pointing to the blue object stored on a remote disk, the system automatically loads the green object from the remote workstation.

5.3 Benchmark

In the following subsections, we measure the running time for movement and construction of a virtual scene. Our benchmarks are defined and described in the corresponding subsections. Common to all benchmarks is the following object of the scene (see Fig. 5.11): the object is a simple modelled house and a tree consisting of 61 triangles. The object is of size 6 (depth), 8.5 (width), and 4 (height). The polygon data of the object needs 2917 Bytes storage. An object of the graph stored on the disk and on a remote workstation needs 3082 Bytes storage because of additional data for graph edges, position and pointers to duplicated objects.

We use the object in our benchmarks for the generation of a randomly distributed scene. The objects to be inserted and deleted can be of arbitrary type and shape. Objects of a virtual scene can be more complex. Typically, values range from 1000 to 5000 triangles. Our benchmark house has comparatively few polygons. There is a tradeoff between the complexity of objects and the number of objects. If we use complex objects, we can place only a few objects in the scene because the system supports no methods for polygon reduction.

Memory Consumption and Overhead

The object in our benchmarks needs 2917 Bytes of storage for the polygon data. A generated object needs 165 Bytes additional data for the storage of edges, position, and pointers to the parents and children of duplicates. Therefore, we have a memory overhead of 5.6% which is a good value for the objects used in the benchmark. We should avoid larger overheads of smaller objects, i.e., the 2917 Bytes is the lower limit for a reasonable size.

Hardware and Software Used

We implemented the system on Irix (SGI) and Linux (PC). The source code is written in C++. For the graphical user interface, we used X11, Xlib, and Motif libraries. The 3D scenes were rendered with the OpenInventor library [233]. All objects are coded with the OpenInventor file format. For all measurements, the 3D viewer runs on a SGI O2 with 256MB main memory and 160MHz clock speed. For all measurements, the 2D viewer and the managers run on Linux 2.4. For the measurements of different storage types (see Section 5.4.1), the 2D viewer runs on a Pentium 3 PC with and 933Mhz clock speed and 1GB main memory. All network communications are executed via TCP connections. For our measurements of scalability (see Section 5.4.2), the managers run on a cluster of 20 PCs. The first eleven PCs are Pentium 4 systems with 1.7GHz and 1GB main memory, and the last nine PCs are Pentium 3 systems with 933 MHz and 512 MB main memory.

5.4 Construction and Recomputation of a Scene

In the following subsections, we investigate the construction time of a virtual scene that is constructed from scratch. For the construction of the scene, we use the algorithm described in Section 4.2.2. The algorithm is essentially not only for the construction of the scene, but also for the algorithms of the Insert(x, o) and Delete(x) operations (Section 4.4.3). For insertion and deletion, we execute a circular range query and recompute the sectorgraph for all objects lying inside a circle with fixed radius.

The algorithm to compute the sectorgraph must be fast to make an interactive manipulation of the scene by the modeller possible. Therefore, we investigate the construction time of the sectorgraph in the following subsections. In Section 5.4.1, we investigate the

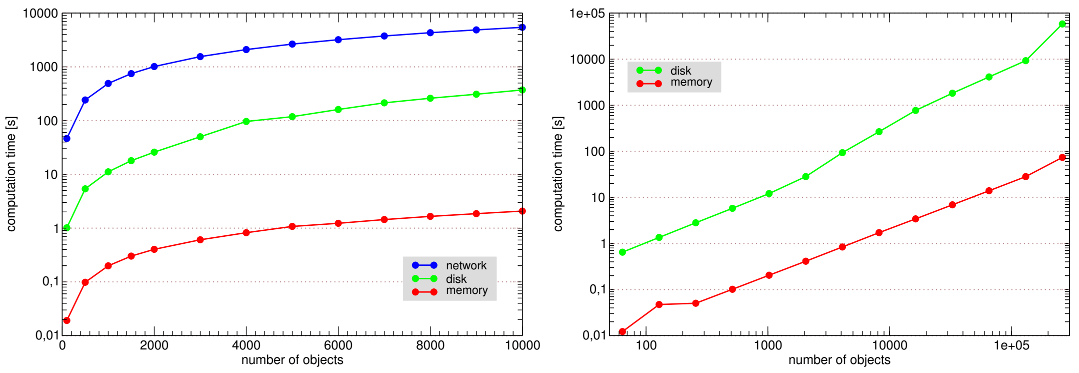


Figure 5.12: Computation time against the number of objects. Red curves for a construction in Memory (`TuMemory`), green for a construction on hard disk (`TuDisk`), and blue for a construction across the network (`TuNetwork`).

running time for the storage types `TuMemory`, `TuDisk`, and `TuNetwork`, and in Section 5.4.2 for a construction across a network of multiple servers.

5.4.1 Construction Time for Different Storage Types

The dimension of the scene is of size 2500x1500 units. We place a fixed number of objects chosen randomly into the scene and use the object described in Section 5.3. We vary the number of objects from 100 to 10000 and measure the construction time of the scene, i.e., the time for the computation of the sectorgraph. We plot the running time against the number of objects in Figure 5.12. We repeat this measurement for the computation in the main memory, on the local hard disk, and across the network (`TuMemory`, `TuDisk`, `TuNetwork`). The three curves are shown in the left diagram of Figure 5.12. For storage types `TuDisk` and `TuMemory`, the right diagram of Figure 5.12 shows the same measurements for larger scenes ($n = 64 \dots 262144$). We doubled the size of the scene in each step.

Running Time Versus Asymptotic Bound

The running time of our algorithm, described in Section 4.2.2, is of order $O(n \log n)$. The logarithmic factor is due to unconnected objects (objects without a next neighbour) lying left of the scan line (see the discussion of the algorithm in Section 4.2.2), and because of the sorting algorithm, which is executed three times. The logarithmic factor ranges from four to nine for scenes of size 100 to 10000. Despite this multiplier, the running time curves show a linear characteristic. The reason for the nearly linear running time is the equal distribution of the randomly chosen objects and the

n	`TuMemory` $[s]$	`TuDisk` $[s]$	`TuNetwork` $[min]$	$\frac{\texttt{TuDisk}}{\texttt{TuMemory}}$	$\frac{\texttt{TuNetwork}}{\texttt{TuMemory}}$	$\frac{\texttt{TuNetwork}}{\texttt{TuDisk}}$
100	0.019	1.01	0.77	53	2430	46
500	0.098	5.38	4.03	55	2460	45
1000	0.199	11.14	8.18	56	2469	44
1500	0.302	17.99	12.47	60	2479	42
2000	0.403	25.87	16.88	64	2511	39
3000	0.607	50.04	25.83	82	2553	31
4000	0.823	96.35	34.88	117	2544	22
5000	1.074	118.44	44.19	110	2468	22
6000	1.229	160.89	53.31	131	2602	20
7000	1.443	213.88	62.49	148	2599	18
8000	1.653	260.82	71.84	158	2608	17
9000	1.857	309.26	81.06	167	2620	16
10000	2.074	371.69	90.57	179	2621	15

Table 5.1: Running time and ratio for the construction of a scene.

fixed length of the scan line. The scan line moves from left to right across the scene. Therefore, the density of objects on each side of the scan line is approximately of the same size for each position of the scan line. A higher scene density does not increase the number of unconnected objects. The number of unconnected objects depends on the length of the scan line if the objects are equally distributed. Also the logarithmic multiplier of the sorting algorithm is not apparent.

Running Time Versus Storage Type

Tables 5.1 and 5.2 show the values and ratios of computation times plotted in Fig. 5.12. The first and last column shows the ratios of computation on the disk to computation in the memory. Further columns show the ratio of network to memory, and network to disk. The ratios show the additional costs of the computation on different storage types. The computation of 64 objects on disk takes 53 times longer than the same computation in the memory.

The ratios of $\frac{\texttt{TuDisk}}{\texttt{TuMemory}}$ range from 53 ($n = 64$) to 328 ($n = 131072$). Noticeable is the steady increase in the ratio, i.e., the computation on disk is more expensive for large scenes than for small scenes. This is caused by the file caching mechanism of the operating system. Free parts of the main memory are used for file caching (1G Byte in our tests). For large scenes, the computation needs more memory such that the file cache becomes smaller and the inexpensive access to the file cache must be replaced by expensive disk accesses. The ratios 29 ($n = 128$) and 788 ($n = 262144$) seem to be irregularities.

The ratios of the other columns bear out our observation. The ratio $\frac{\texttt{TuNetwork}}{\texttt{TuMemory}}$ shows

n	TuMemory $[s]$	TuDisk $[s]$	$\frac{\texttt{TuDisk}}{\texttt{TuMemory}}$
64	0,012	0.01	53
128	0.047	0.02	29
256	0.050	0.05	56
512	0.101	0.10	57
1024	0.205	0.20	59
2048	0.412	0.47	69
4096	0.840	1.56	111
8192	1.718	4.44	155
16348	3.420	12.85	225
32768	6.895	30.41	265
65536	13.936	68.47	295
131072	28.254	154.36	328
262144	73.738	969.02	788

Table 5.2: Running time and ratio for the construction of a scene.

the same increase, ranges from 2430 ($n = 100$) to 2621 ($n = 10000$), but the relative increase is smaller. The increase is due to the file caching of the remote workstation. The relative increase is smaller because of the latency of the network transfer. We have no caching mechanism for the network transfer. This has an effect on the ratio $\frac{\texttt{TuNetwork}}{\texttt{TuDisk}}$. The values decrease steadily to 22 ($n = 4000$), but after $n = 4000$ the relative increase is smaller.

Now it is possible to answer the question how much more time the computation costs on hard disk than in main memory, and how much more time the computation costs across the network than on hard disk. The ratios $\frac{\texttt{TuDisk}}{\texttt{TuMemory}}$ and $\frac{\texttt{TuNetwork}}{\texttt{TuDisk}}$ of the table show that for small scenes $n < 4000$, we have nearly equal ratios memory to disk and disk to network. The ratio of memory to disk of a scene with $n = 4000$ objects is 117, and 22 for disk to network. This means that we lose the most time for the disk access. The difference increases for larger scenes. The ratio memory to disk is 179 for a scene of size $n = 10000$ objects, and the ratio disk to network is 15.

5.4.2 Multiple Managers for a Storage Across a Network

We investigate the construction time of a virtual scene that is distributed across the network and make the following measurement for it: the dimension of the scene is of size 2500x1500 units. We place a fixed number of objects chosen randomly into the scene and use the object described in Section 5.3. All objects are distributed on remote hard disks (storage type `TuNetwork`). We vary the number of managers from 1 to 20, and

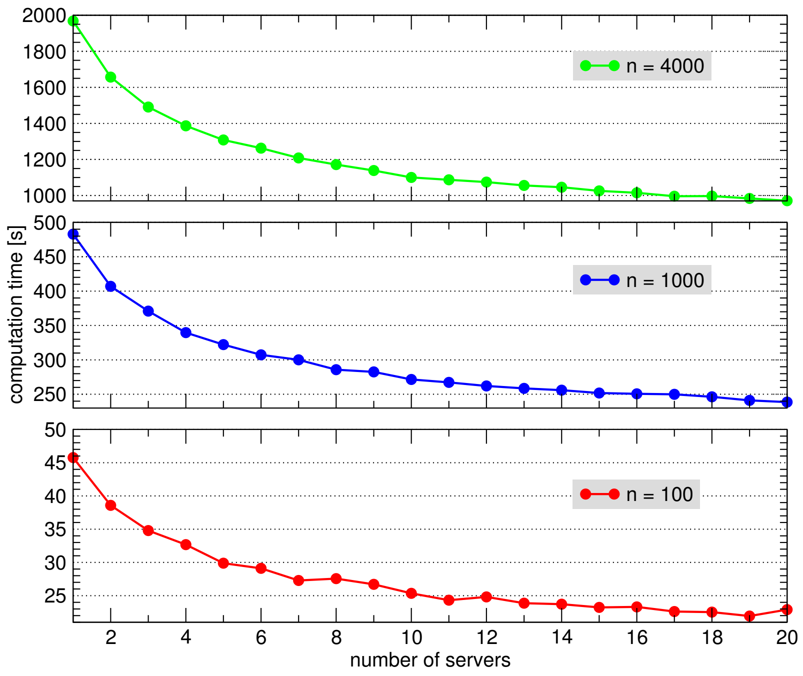


Figure 5.13: The computation time across the network (`TuNetwork`) plotted against the number of managers (servers). The red curve is for a scene of size 100 objects, the green curve for 1000 objects, and the blue curve for 4000 objects. The speedup converges against two.

measure the construction time of the scene. In Figure 5.12, the running time is plotted against the number of managers (servers). We repeat the measurement for different sizes of the scene $n = 100$ (red curve), $n = 1000$ (green curve), and $n = 4000$ (blue curve).

The computation of the graph is performed by a single workstation. The implemented algorithm (Section 4.2.2) executes the computation steps sequentially. Therefore, we expect the same running time independent of the number of managers. The three curves of Fig. 5.12 show that we achieve a speedup if we use more managers. The speedup converges against two for 20 managers. The relative speedup is maximum 0.59 for two managers. Each additional manager accelerates the computation time, however, the relative speedup falls off.

The reason for this speedup is the asynchronous implementation of the *write* and *free* operations. The write operation sends a packet from the client to a manager. After the message is transferred, the client program continues and does not wait until the data is written to the disk of the manager. The same holds for the free operation. The free operation frees an allocated storage on the remote disk. After the message is sent out, the client program continues. Subsequent operations to distinct remote disks can be executed simultaneously.

5.5 Motions Through the Scene

In this section, we investigate the motions of a user through the virtual scene. We are interested in how fast a visitor moves through the scene. The speed of the visitor depends on the running time to move a bubble. Therefore, we measure the running time of the $\textsc{Move}(x, \Delta x)$ operation dependent on the storage type of the bubble. We distinguish between a user who slots only one `TuMemory` bubble into the scene bubble (Section 5.5.3), and a visitor who uses two bubbles that work as a cache for fast movement of nearby positions (Section 5.5.4). For the latter, we investigate the frame rate of the walkthrough system in order to analyse the influence of remote accesses on the running time of the client.

5.5.1 The Radii of Bubbles

A slight restriction of our system is that we render only the objects of the neighbourhood of the user. No objects outside the bubble are rendered. As soon as the moving bubble reaches objects lying outside the scene, the objects pop in the rendered image. The missing objects outside the bubble and the popping effect can be neglected if the radius of the bubble is large enough so that the border of the bubble is far away. However, we cannot choose the radius to be arbitrarily high because the complexity of the objects to be rendered increases. Furthermore, the radii of the bubbles have an effect on the number of updates of the bubble in the bubble hierarchy, and on the running time of a movement operation. Therefore, we must choose the radii thoroughly.

Figure 5.14 shows a scene consisting of 5000 objects (benchmark house). The four screenshots show a view to the scene for four different radii of the bubble (50, 100, 300, and 400 units). The bubble with radius 50 contains 15 objects, the bubble with radius 100 contains 47 objects, the bubble with radius 300 contains 387 objects, and the bubble with radius 400 contains 662 objects. Note that the view frustum renders only a few objects of the bubble. The radius of size 50 shows an observable error because of the missing objects outside the bubble. The error is less observable for the bubble of size 100. We can neglect the error caused by missing objects for the bubble with radius 300. The differences to the image with radius 400 are hard to observe. Of course, this impression is relative to the position of the viewer. If the viewer moves a little bit

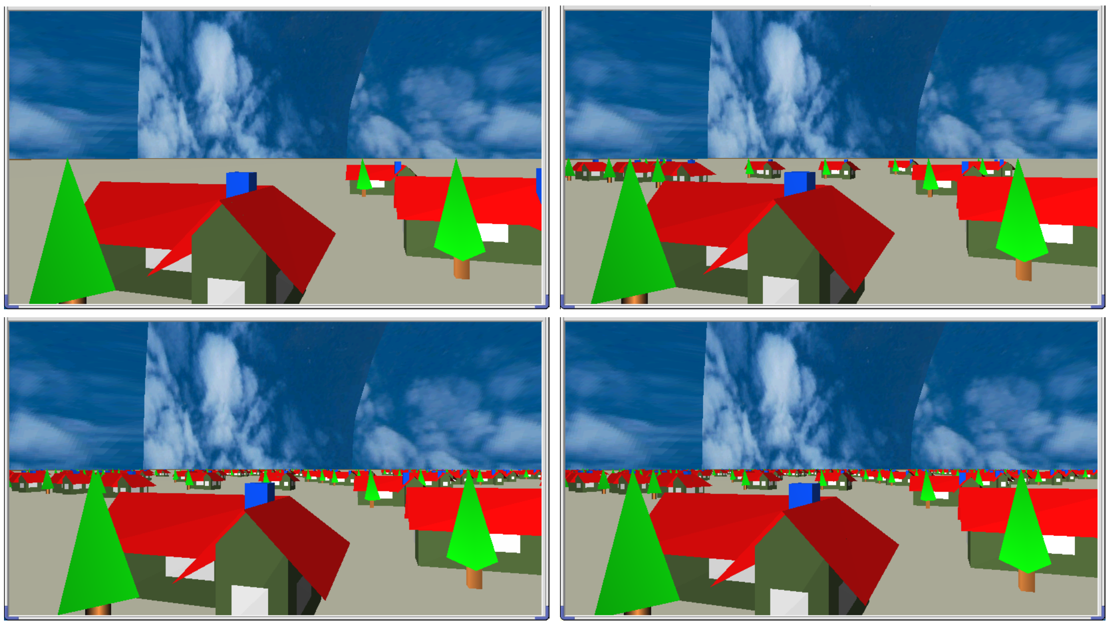


Figure 5.14: A walk through our benchmark scene: the house with the tree is a single object. The three images show a view of the scene for three different radii of the bubble (50, 100, 300, and 400 units).

higher, he sees better the border of the bubble. Therefore, for higher positions of the viewer, we must increase the radius of the bubble to reduce the artefacts.

5.5.2 Benchmark

The benchmark scene is of size 2500x1500 units and consists of 5000 randomly chosen objects (the house described in Section 5.3). We define a path of the moving bubbles and visitors (red line in Fig. 5.15). Four markers are positioned on the path (1, 2, 3, and 4). For benchmark *scene 1*, the markers have the coordinates (x,y) = (300, 300), (500, 500), (700, 700), and (900, 900). For the benchmarks *scene 2* and *scene 3*, the four markers have the coordinates (300, 300), (800,800), (1400, 800), and (1400, 1500).

The value r denotes the radius of a bubble. The extent of scene, bubble, and path are true to scale. We denote a bubble that stores the whole scene by *scene bubble*. A bubble that is rendered in the 3D viewer is denoted by *view bubble*. The viewer sees all objects of the view bubble; other objects are not rendered. If the viewer moves through the scene, the 3D viewer executes his motion by a movement of the view bubble. However,

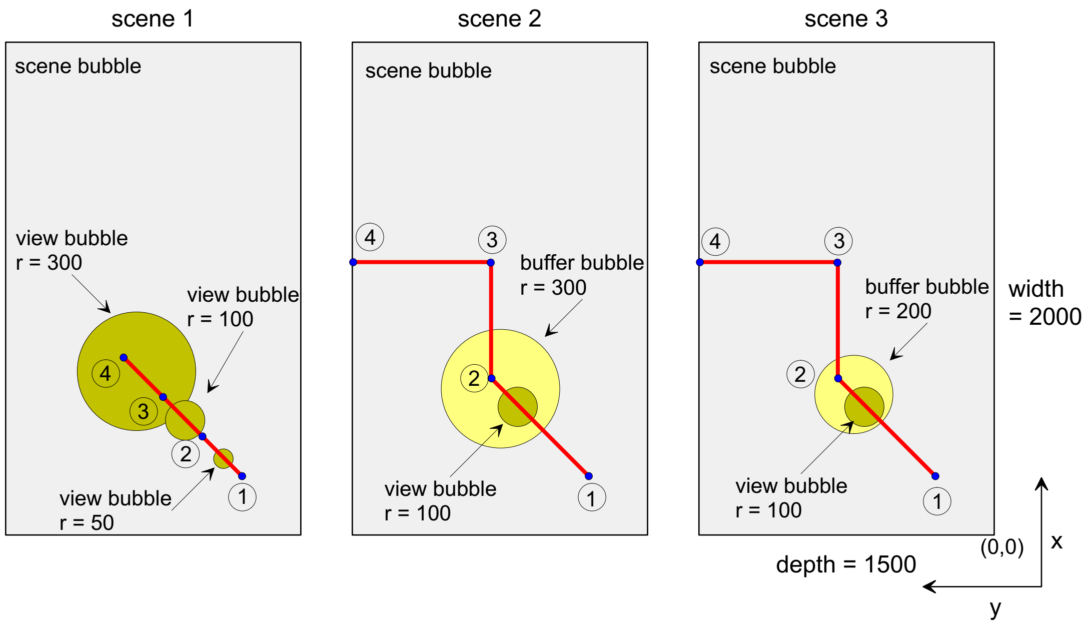


Figure 5.15: The red line is the path of the bubbles. For scene 2 and scene 3, the coordinates of the markers 1, 2, 3, and 4 are (x,y) = (300, 300), (800,800), (1400, 800), and (1400, 1500). For scene 1, the coordinates are (x,y) = (300, 300), (500, 500), (700, 700), and (900, 900). The value r denotes the radius of a bubble. The extent of scene, bubble, and path are true to scale.

the system does not move the bubble for small motions of the viewer. The system uses a threshold of 10 units. If the viewer is 10 units away from the centre of the bubble, the system executes the MOVE$(x, \Delta x)$ operation. We use this threshold to save execution time and because it is visually undisturbing to fix the centre of the bubble for small movements. If a third bubble is used to cache the scene, as in scene 1 and scene 2, we denote the bubble by *buffer bubble*. The buffer bubble is moved only if the view bubble crosses the border of the buffer bubble. Otherwise, the view bubble moves freely inside the buffer bubble without updating the buffer bubble.

5.5.3 Movements with one Bubble

The goal in this section is the investigation of an unbuffered movement through a scene. We investigate the frame rate and the running time of the MOVE$(x, \Delta x)$ operations for the storage types `TuDisk` and `TuMemory`. The results of both test series are shown in Fig. 5.17 and 5.16, and Table 5.4, 5.3, and 5.5).

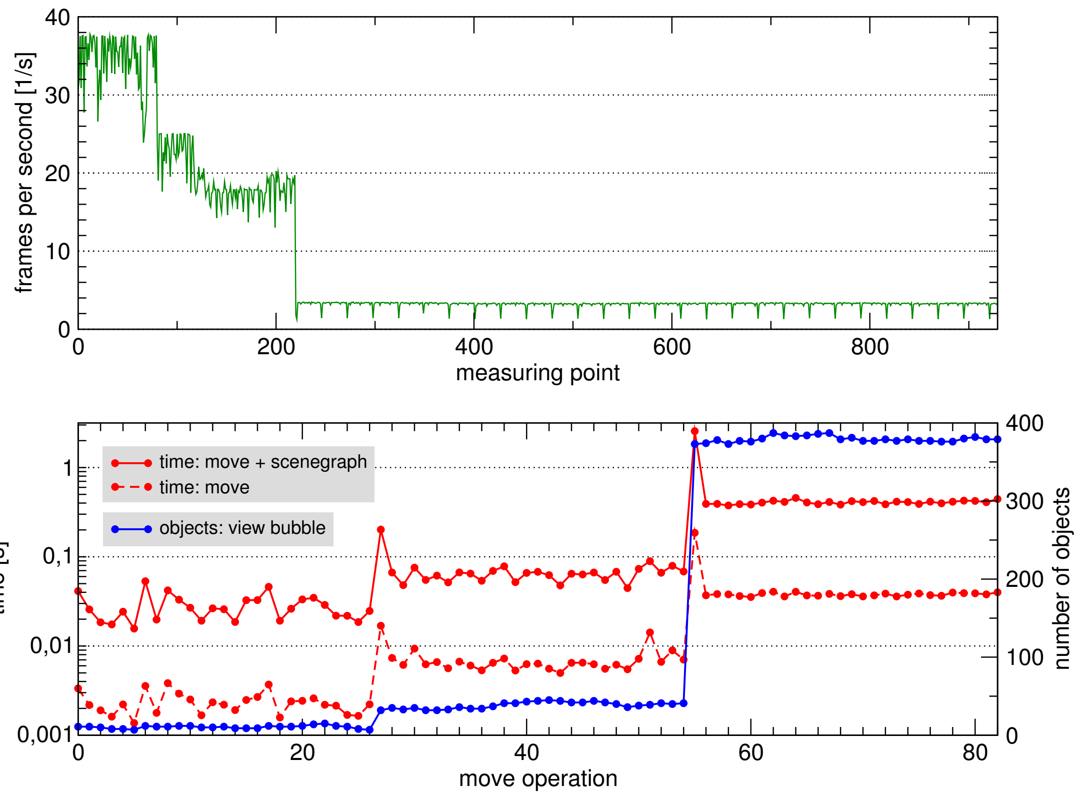


Figure 5.16: Benchmark scene 1: the scene bubble and the view bubble are stored in the main memory (storage type `TuMemory`).

We use the benchmark scene 1 shown in Fig. 5.15. A scene bubble stores the whole scene and a view bubble contains the objects to be rendered. We execute two test series: the one uses a `TuDisk` bubble to store the scene on the hard disk, and the other uses a `TuMemory` bubble to store the whole scene in the memory. In both cases, the view bubble stores the objects in the memory (`TuMemory`). We move the view bubble along the path from marker 1 to marker 4. The radius of the view bubble changes when we reach the next marker. The radius is of size 50 from marker 1 to 2, 100 from marker 2 to 3, and 300 from marker 3 to 4.

The blue curve shows the number of objects of the view bubble. The curve is formed as a three-stepped stair caused by the changing radius of the view bubble. The average number of the objects is 10 for radius 50, 38 for radius 100, and 379 for radius 300. The dashed red curve shows the running time for the MOVE$(x, \Delta x)$ operation. After

	radius 50			radius 100			radius 300		
	o	t_{mo}	t_{mo+su}	o	t_{mo}	t_{mo+su}	o	t_{mo}	t_{mo+su}
minimum	7	0,0014	0,0157	32	0,0050	0,0445	373	0,0354	0,3762
average	10,52	0,0023	0,0277	38,43	0,0071	0,0688	379,18	0,0431	0,4840
maximum	15	0,0038	0,0532	45	0,0168	0,2015	387	0,1858	2,5570

Table 5.3: Statistical data of Fig. 5.16, the scene is stored in memory. o: number of objects. t_{mo}: running time for move operations. t_{mo+su}: running time for move operation plus updating the scenegraph.

	radius 50			radius 100			radius 300		
	o	t_{mo}	t_{mo+su}	o	t_{mo}	t_{mo+su}	o	t_{mo}	t_{mo+su}
minimum	7	1,40	1,41	31	6,07	6,12	372	53,13	53,49
average	10,56	1,97	2,00	38,36	7,28	7,34	379,36	55,80	56,23
maximum	15	2,72	2,76	45	9,43	9,50	387	58,38	59,91

Table 5.4: Statistical data of Fig. 5.17, the scene is stored on disk. o: number of objects. t_{mo}: running time for move operations. t_{mo+su}: running time for move operation plus updating the scenegraph.

	$t_{su}^{\texttt{TuMemory}}/t_{mo}^{\texttt{TuMemory}}$	$t_{mo}^{\texttt{TuDisk}}/t_{su}^{\texttt{TuDisk}}$	$t_{mo+su}^{\texttt{TuDisk}}/t_{mo+su}^{\texttt{TuMemory}}$
minimum	5,29	25,09	23,43
average	9,77	120,67	109,07
maximum	13,89	218,79	194,43

Table 5.5: Averaged running time ratios of $\textsc{Move}(x, \Delta x)$ operations to updating the scenegraph (data from Fig. 5.16 and Fig. 5.16). t_{mo}: running time for move operations. t_{su}: running time for updating the scenegraph. $t_{mo+su} := t_{mo} + t_{su}$. In each case for storage type `TuMemory` and `TuDisk`.

executing the operation, the polygon data of the objects must be inserted into the scenegraph. The solid red line shows the running time for the $\textsc{Move}(x, \Delta x)$ operation plus the time for updating the scenegraph. The test with storage type `TuMemory` (see Fig. 5.16) shows that updating the scenegraph is 9.77 times more expensive on average than the $\textsc{Move}(x, \Delta x)$ operation. The ratio ranges from 5.29 to 13.89 (see Table 5.5). The results are inverse for the test with storage type `TuDisk` (see Fig. 5.17). The time for the $\textsc{Move}(x, \Delta x)$ operation is 120.67 times more expensive on average than the time for updating the scenegraph. The ratio ranges from 25.09 to 218.79 (see Table 5.5). The time for the $\textsc{Move}(x, \Delta x)$ operation is so large that the sum of the $\textsc{Move}(x, \Delta x)$ operation plus updating the scenegraph has nearly the same values. Therefore, in Fig. 5.17, we see only one red curve although both curves are plotted. The red curves of Fig.

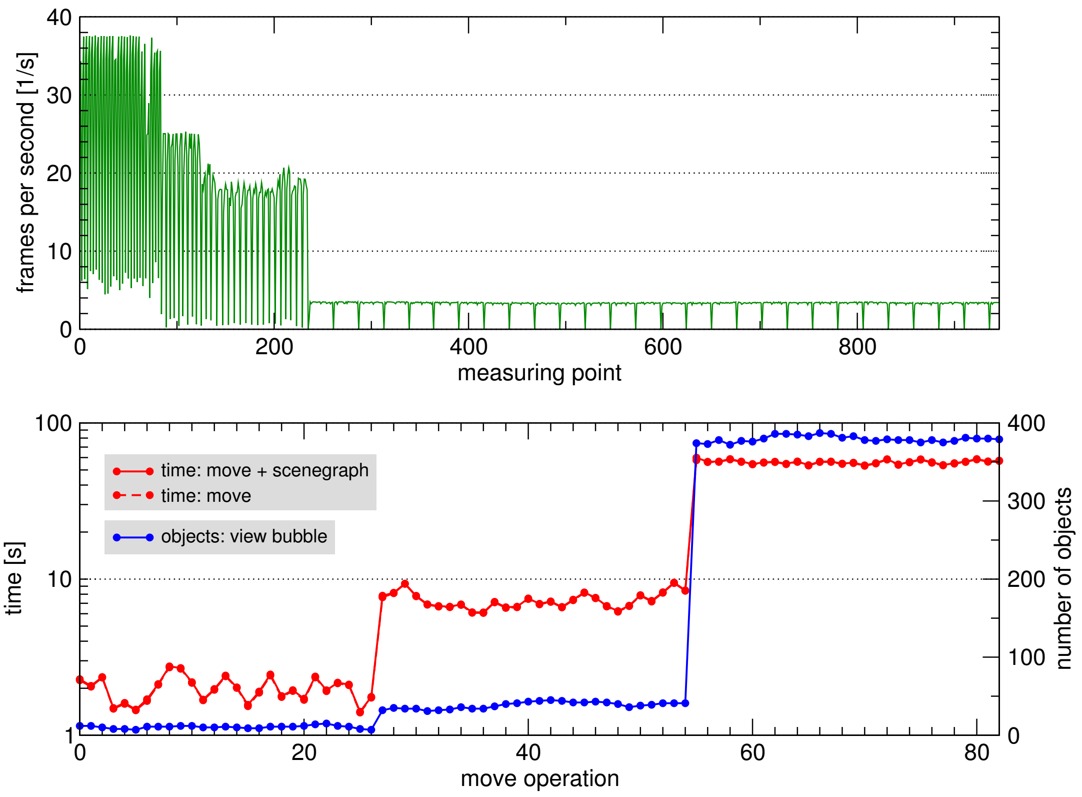


Figure 5.17: Benchmark scene 1: The scene bubble is stored on disk (storage type `TuDisk`) and the view bubble is stored in the memory (storage type `TuMemory`).

5.16 show a peak if we increase the radius from 50 to 100 and from 100 to 300. This is due to the large number of objects to be inserted into the scenegraph. Subsequent MOVE$(x, \Delta x)$ operations generate fewer objects to be inserted.

We compare the running time of the MOVE$(x, \Delta x)$ operation plus the scenegraph update for the storage types `TuDisk` and `TuMemory`. The third column of Table 5.5 shows the average values of the red curves of Fig. 5.16 and 5.17. We calculated the ratio for each measuring point and calculated the average afterwards. The ratio of a MOVE$(x, \Delta x)$ operation executed on disk to one executed in the main memory is an average of 109.07. The ratios range from 23.43 to 194.43. This means that the movement of the bubble stored on disk takes 109 times longer than movement of a scene that is stored in main memory.

Now, we investigate the frame rates that we achieve during a walk through the

	radius 50	radius 100	radius 300
average `TuMemory`	34,54	19,15	3,26
average `TuDisk`	25,59	15,95	3,27

Table 5.6: Averaged frame rates for view bubbles of different radii and storage types.

scene (see the upper diagram of Fig. 5.16 and 5.17). Like the running time for the $\textsc{Move}(x, \Delta x)$ operation, the frame rate is formed as a three-stepped staircase. Due to the larger number of objects in a view bubble with a large radius, the frame rate drops for larger bubbles. Except for the positions where the move operation is executed, the frame rates are approximately equal independent of the storage type. The frame rates of the scene stored in memory are of the same order as the one stored on the disk. Because we execute the move operations after 10 units, the system executes only the rendering of the objects without any influence on operations of the sectorgraph. The Table 5.6 shows the average frame rates for the three parts of the path. The average frame rates of the system depend on the type of storage because during the execution of the $\textsc{Move}(x, \Delta x)$ operation, the system waits resulting in a frame rate nearly zero. Due to the more expensive operation of `TuDisk`, the average is worse for a scene stored on disk.

In contrast to the curves for the running time of the $\textsc{Move}(x, \Delta x)$ operations, the number of measuring points is distinct for each of the three parts of the path. Due to the longer rendering times for larger bubbles, the system needs more time to move the bubbles although the lengths of the three parts are equal. Because the system does not skip positions, it generates more measuring points. Therefore, the number of the measuring points for the frame rate is proportional to the time used to move the bubble.

5.5.4 Movements with two Bubbles

In this section, we investigate buffered movement, i.e., we use two `TuMemory` bubbles that are slotted in the scene bubble. So far, we have used a single view bubble that is slotted in the scene bubble. The disadvantage of this configuration is that the system must frequently reload objects from the scene bubble for each movement. If the viewer explores his neighbourhood and moves forward and backward several times, the system thrashes, i.e., it loads and deletes the objects of the neighbourhood frequently. To avoid this type of thrashing, we use the buffer bubble that is slotted in the scene bubble (see Fig. 5.15). The view bubble is slotted in the buffer bubble. The buffer bubble works as a spatial cache for the view bubble. Even if in this configuration thrashing occurs between the view bubble and the buffer bubble, the problem is defused because

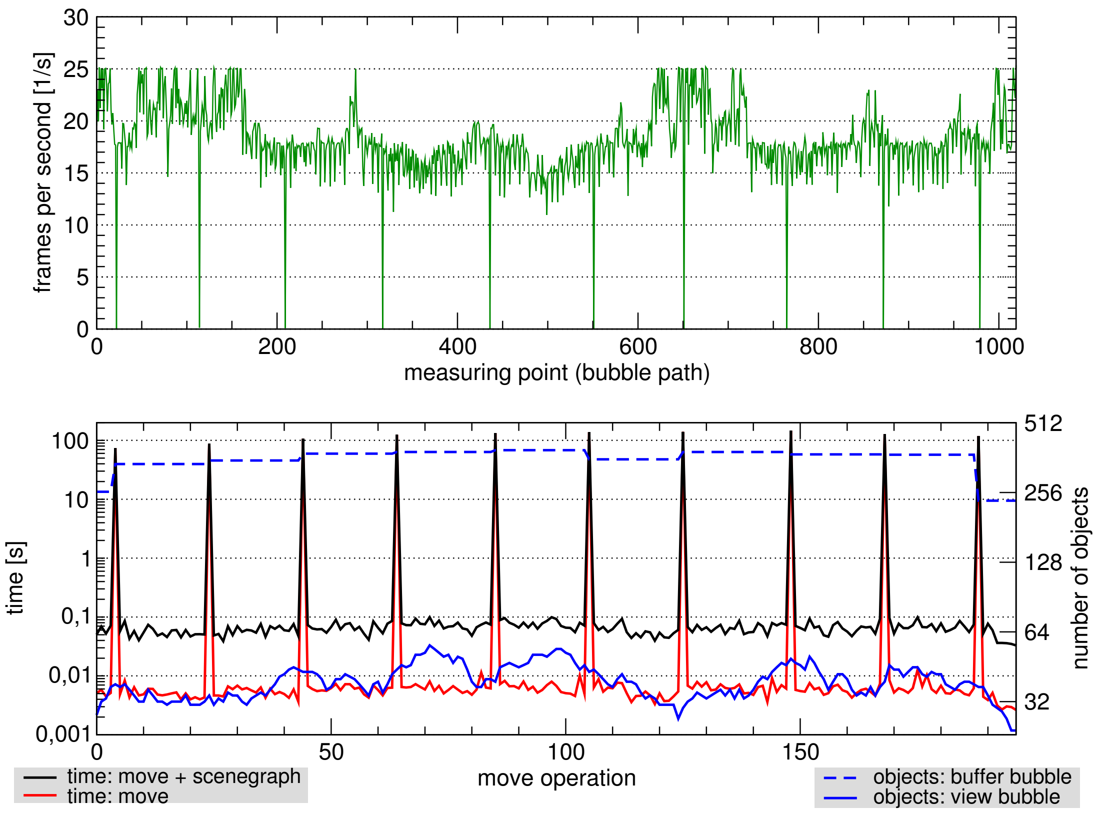


Figure 5.18: Benchmark scene 2: The scene bubble is stored on disk and the view and buffer bubbles are stored in memory. The radius of the buffer bubble is of size 300.

the buffer bubble is stored in the main memory so that fast access to the objects is possible. The system moves the buffer bubble as soon as the view bubble crosses the border of the buffer bubble.

We use benchmark scene 2 and scene 3 (see Fig. 5.15). The radius of the view bubble is of size 100 units. The radius of the buffer bubble is of size 300 in scene 2, and 200 in scene 3. For the scene bubble, we use storage types `TuMemory` and `TuDisk`. Storage type `TuNetwork` is investigated at the end of this section. We measure the frame rate of the system and the running time of the $\textsc{Move}(x, \Delta x)$ operations. The results of the four test series are shown in Fig. 5.20, Fig. 5.21, Fig. 5.18, and Fig. 5.19. We plot the frame rate in the upper diagram (green curve). In the lower diagram, we plot the number of objects of the view bubble and buffer bubble with the axis on the right side

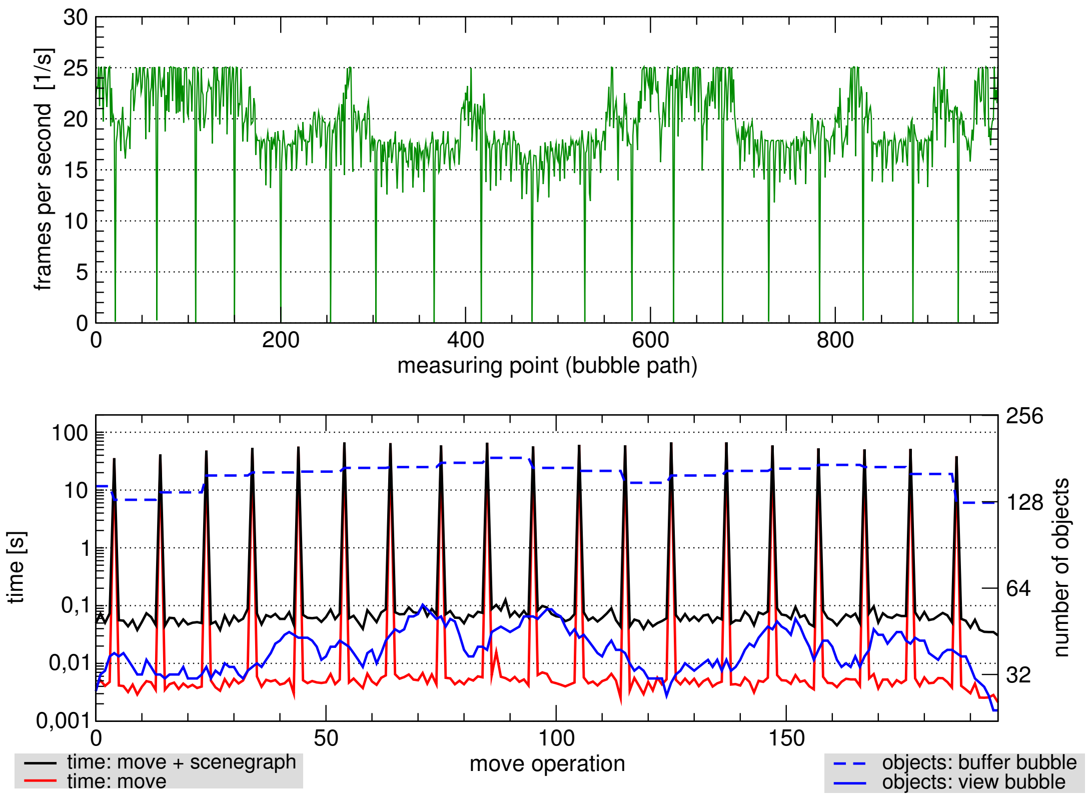


Figure 5.19: Benchmark scene 3: The scene bubble is stored on disk and the view and buffer bubbles are stored in memory. The radius of the buffer bubble is of size 200.

(blue curves). We plot the running time of the Move$(x, \Delta x)$ operation (view bubble plus buffer bubble) in the lower diagram (red curve). The black curve results from the red curve plus the running time for updating the scenegraph.

The running times of the Move$(x, \Delta x)$ operations (view bubble plus buffer bubble, red curves) show large peaks at the positions where the buffer bubble is moved. Between those peaks, only the view bubble is moved.

Frame Rate

The first question is how the configuration influences the frame rate of the system. The frame rate of the two test series for storage type `TuDisk` (buffer bubble radius 200 and 300) shows a comparable characteristic in quality and quantity (Fig. 5.18, and Fig. 5.19). This does not hold for the positions where the buffer bubble is updated. At

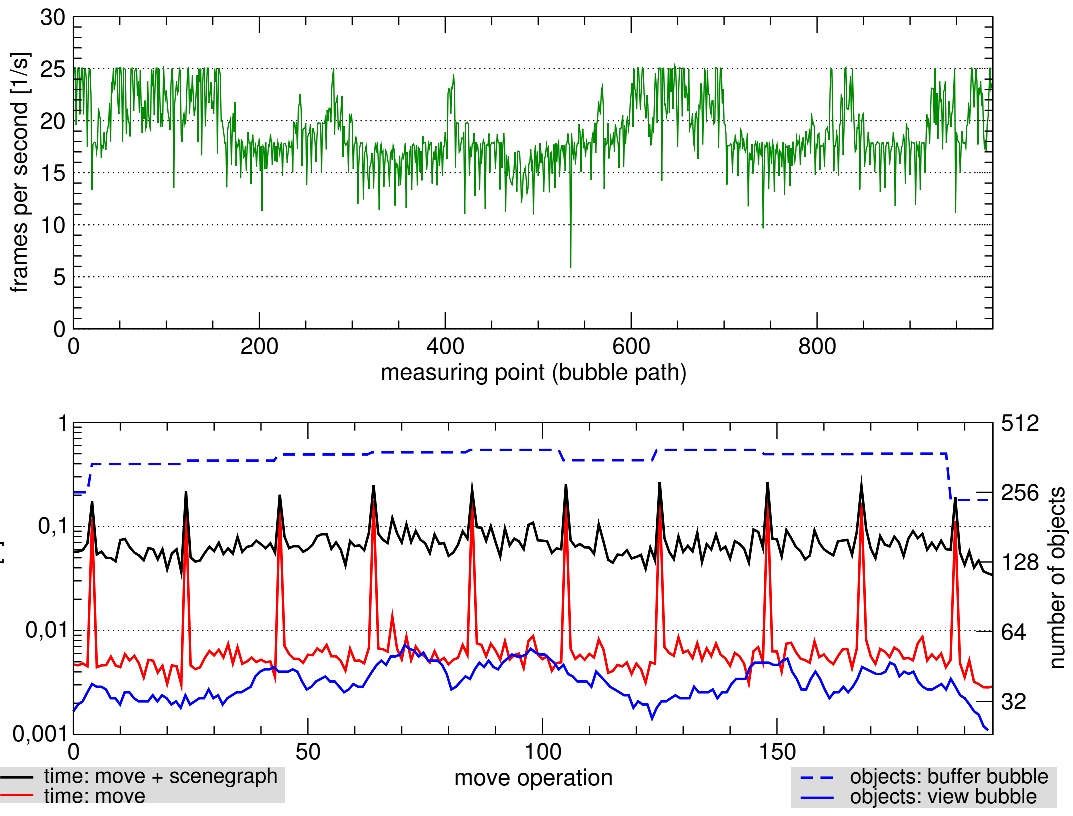


Figure 5.20: Benchmark scene 2: Scene bubble, view bubble, and buffer bubble are stored in memory. The radius of the buffer bubble is of size 300.

those positions, the frame rate drops down to zero. During the movement, the system makes 10 updates for radius 300, and 19 updates for radius 200. Except for those points of updates, the minima and maxima of the frame rate are at approximately the same measuring points on the path. The maxima and minima are mainly influenced by the number of objects to be rendered. The curve for the number of objects of the view bubble has its maxima at that point where the frame rate shows minima.

We can answer the question whether the type of storage influences the frame rate. Fig. 5.20 and Fig. 5.21 show the same two test series for the scene that is stored in the main memory. Both curves (frame rates) show a comparable characteristic in quality and quantity. Furthermore, they are similar to the two curves for the storage type `TuDisk`. The main difference is that they miss the points of updating the buffer bubble

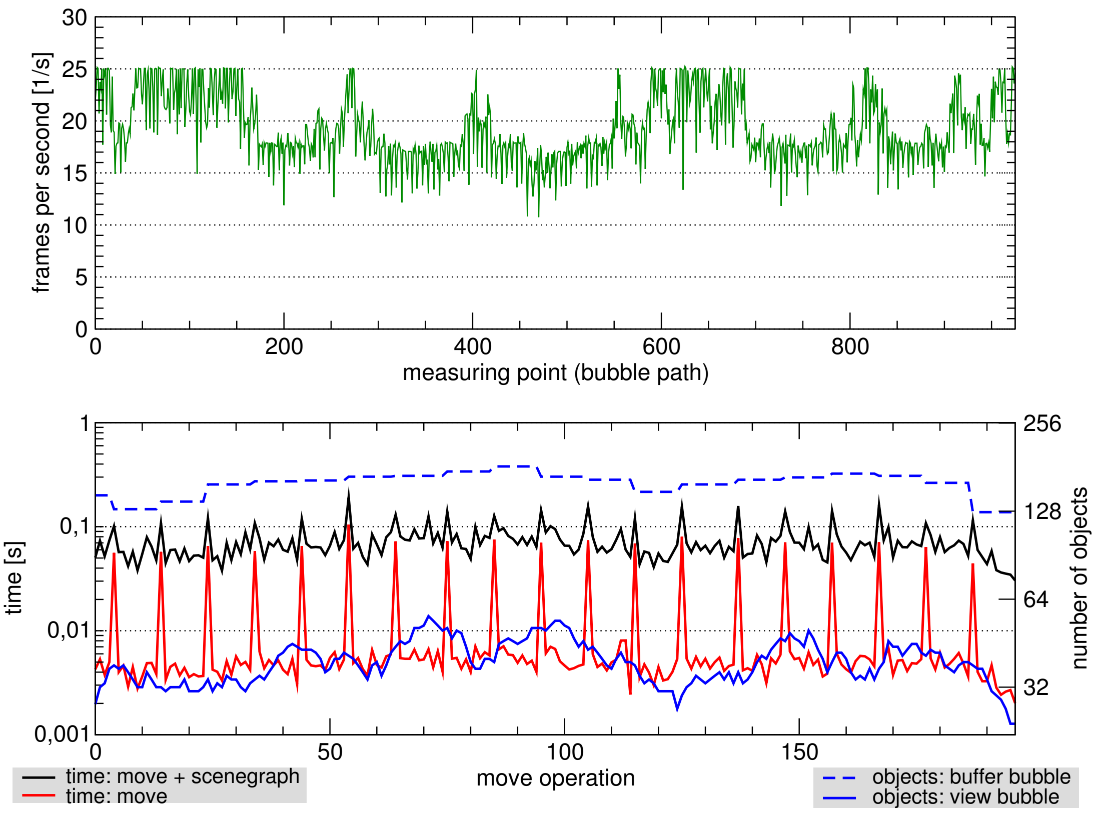


Figure 5.21: Benchmark scene 3: Scene bubble, view bubble, and buffer bubble are stored in memory. The radius of the buffer bubble is of size 200.

where the frame rate drops down to zero. Because the scene is stored in main memory, the execution of the MOVE$(x, \Delta x)$ operation is faster so that the system achieves a better frame rate.

The average frame rate for the storage types `TuMemory` and `TuDisk` and for radius 300 and 200 is shown in Table 5.7. The average frame rates are of approximately the same size. Only the minimum is nearly zero for the `TuDisk` type because of the long running times to execute the MOVE$(x, \Delta x)$ operation.

Radius of the Buffer Bubble and Speed of the Viewer

The next question is how we have to choose the radius of the buffer bubble. A large radius allows the exploration of a large neighbourhood, so that we can move arbitrarily inside the neighbourhood. This is advantageous if we are staying in the same area for a

	TuMemory		TuDisk	
	300	200	300	200
minimum	5,88	10,75	0,03	0,02
average	18,93	19,22	18,18	19,01
maximum	25,24	25,24	25,34	25,24

Table 5.7: Average frame rate dependent on storage type and radius.

scene	radius	part 1-2 [s]	part 2-3 [s]	part 3-4 [s]	total [s]
TuMemory	300	179.14	168.72	185.01	532.86
TuMemory	200	175.57	166.40	182.71	524.68
TuDisk	300	577.49	582.65	584.99	1745.12
TuDisk	200	540.56	533.72	500.21	1574.49

Table 5.8: The time for the motion of the viewer.

	TuMemory [s]	TuDisk [s]	TuNetwork [s]	$\frac{\texttt{TuDisk}}{\texttt{TuMemory}}$	$\frac{\texttt{TuNetwork}}{\texttt{TuMemory}}$
radius 200	0,0647	55,45	79,22	857	1224
radius 300	0,1526	119,58	141,25	783	925

Table 5.9: The average running time of the $\textsc{Move}(x, \Delta x)$ operation from markers 1 to 4.

long time. However, a large radius of the buffer bubble is disadvantageous if we move straight ahead to a far away position. Although we need fewer updates for larger buffer bubbles, the total number of objects to be loaded is larger. In consequence, we need more time to move from point A to B.

Table 5.8 shows the time that our viewer and his view bubble need to move from marker 1 to marker 4 of scene 2 or scene 3. The route takes more time for radius 200 than for radius 300. That holds for both storage types. Because of slow loading of the objects from disk, the viewer needs more time if the scene is stored on disk.

Bubble Movements Across the Network

The preceding test series shows that the type of storage influences the average frame rate, but has no influence on the frame rate between subsequent $\textsc{Move}(x, \Delta x)$ operations. As long as the view bubble moves inside the buffer bubble, the frame rate of the system is independent of the type of storage of the scene bubble. Therefore, the following test series investigates the running time of the $\textsc{Move}(x, \Delta x)$ operation of a `TuMemory` bubble that is slotted into a `TuNetwork` bubble.

The benchmark is the same as in the preceding tests except that we investigate the view bubble without a buffer bubble and the 3D renderer. The tests are executed on the workstations used in our tests for the construction of the scene. We move a view

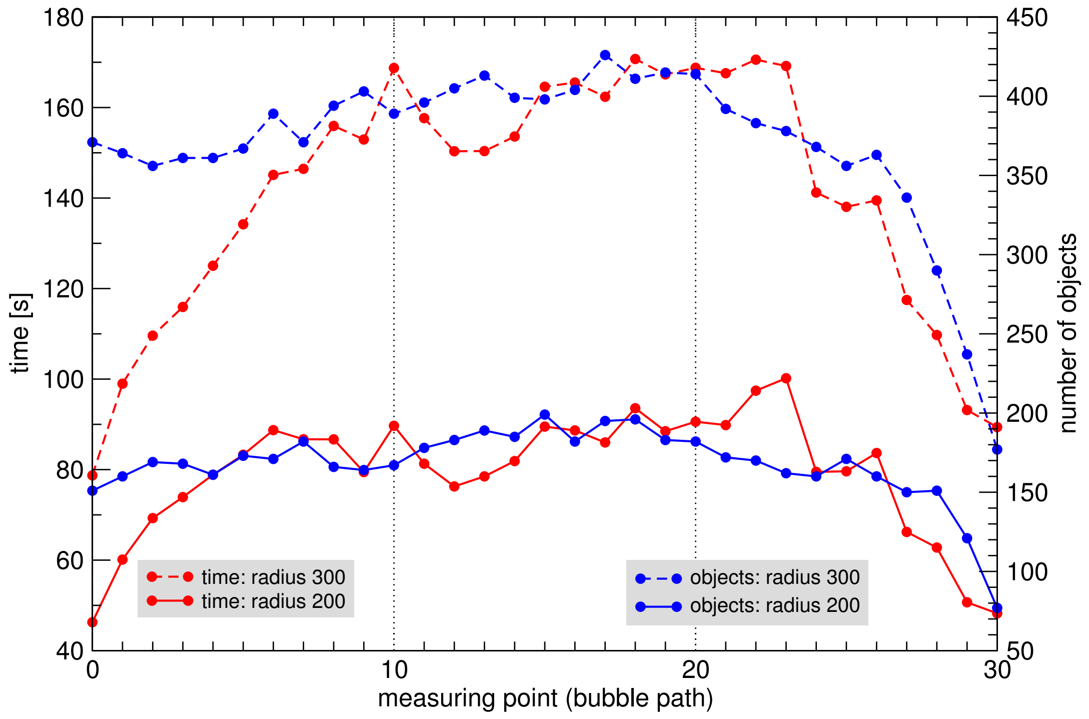


Figure 5.22: The scene is stored across the network (`TuNetwork` bubble). We slot a `TuMemory` bubble into the scene bubble. The path is the same as for scene 2 and scene 3 in Fig. 5.15.

bubble along the path defined in scene 2 and scene 3 of Fig. 5.15. We carry out two test series, one with radius 300, and the other with radius 200. The results are shown in Fig. 5.22. We plot the number of objects of the view bubble and the running time of the MOVE$(x, \Delta x)$ operation against the measuring points of the path.

From markers 1 to 3, the number of objects of the bubble varies only a little (see blue curves). At the end of the path, the curves drop down because the view bubble reaches the border of the scene at marker 4 and half of the view bubble is empty because it juts out the scene. The running time of the MOVE$(x, \Delta x)$ operations (red curves) shows the same characteristic only in the middle and at the end of the path. The running time drops down at the end of the path (marker 4) too, because our search algorithm starts its search near the border of the scene and finds fewer objects. However, the characteristic of the running time differs from the number of objects at the beginning of the path (marker 1). The running time drops although the number of objects is large. This is caused by the stretch factor of our sectorgraph. Marker 1 has the coordinates

(300, 300). Therefore, both bubbles (radius 200 and 300) are lying completely inside the scene. However, our search algorithm must search twice as wide as the radius of the bubble. Therefore, some parts of the area to be searched lie outside the scene containing no objects. These measurements show that the running time depends not only on the number of objects of the bubble, but also on the number of neighbouring objects depending on the stretch factor of the sectorgraph.

We calculated the average running time of the MOVE$(x, \Delta x)$ operation for the three storage classes (`TuMemory`, `TuDisk`, and `TuNetwork`) from the preceding test series (path of scene 2/3). Table 5.9 shows the running times for radius 300 and 200. The ratios of the different classes are somewhere in the order of the ratios that we measured for the construction of the scene. However, the running time relative to the number of objects is better than the running time of the algorithm for the graph construction because the latter is more complex than the search algorithm. The algorithm for the graph construction sorts all objects three times and makes six scan line conversions. The search algorithm of the MOVE$(x, \Delta x)$ operation carries out a single breadth first search.

5.6 Summary and Discussion

We investigate the time for the construction of the sectorgraph. The algorithm is not only for the basic construction of the sectorgraph from scratch, but also for the algorithms of the INSERT(x, o) and DELETE(x) operations.

The results show that fast construction of a scene from scratch is possible. A value of 16 minutes for construction 260,000 objects on hard disk is a usable value, because construction from scratch is executed only once for the initial definition of the scene. Later on, the sectorgraph will be modified only by the manipulation operations of the modeller.

The construction of larger scenes stored across a network is more expensive. The running time of 90 minutes for 10,000 objects is capable of improvement. The ratio of the running times for different storage types (see Table 5.1) shows that the implementation causes no problems. The ratios show typical values for accesses to memory, hard disk, and network. The problem is caused by the constant factor of the asymptotic running time. As sketched in Section 4.4.4, we can overcome the problem by a computation of parts of the graph in the main memory and the storage of the graph on remote disks afterwards. This improves the running time for the construction on remote disks, but causes other problems, e.g., partitioning of the scene.

The results show that an interactive manipulation of the scene is possible. Typically, a bubble consists of 5,000 to 10,000 objects. The running time of size 1-2 seconds is a usable value for the storage type `TuMemory`. Capable of improvement are the running times for `TuDisk` and `TuNetwork`. The running time for 5,000 to 10,000 objects ranges from 2 to 6 minutes, too slow for an interactive manipulation. We can overcome the problem by the recomputation of the sectorgraph of the bubble in the main memory and the storage of the computed graph on disk afterwards.

Our investigations of the movement through the scene show that the system makes frame rates independent of expensive accesses to the disk and network possible if the bubbles are configured as spatial caches. The movement with a view bubble inside a buffer bubble generates frame rates depending only on the number of objects of the view bubble. Not until the view bubble moves outside the buffer bubble do problems occur, then the user must wait and the frame rate drops. To prevent the long running times for loading objects from disk and network, the system could be improved by implementing prefetching and concurrent loading of objects. Because of that, the viewer must not stop. However, this causes other problems. The viewer could turn back suddenly or stand confusedly at the border of the bubble waiting for the objects.

6 Conclusion and Further Development

Conclusion

We propose a system for the management, rendering, navigation, and manipulation of a distributed virtual environment. The system uses a data structure that supports distributing scenes across a network and walkthrough and manipulation of scenes - concurrently by many users at runtime. Our theoretical analysis of the data structure proves that navigation and manipulation can be executed in a period of time that only depends on the complexity of the neighbourhood of the user, independently of the total size of the scene. The locality of the data structure makes these results possible. In the same way, the locality of the data structure permits the system to manage a conceptually unbounded extension and size of the virtual environment.

Outlook and Further Development

Although the running times of the system are independent of the total size of the scene, the experiments show that the running times are too expensive for fast interactive manipulation of the scene. The system is capable of improvements to speed-up the navigation and manipulation.

In order to speed-up navigation, we should use prefetching of the buffering bubble if we detect that the user moves the view bubble to the border of the buffering bubble. We fetch the objects of the buffering bubble concurrently with the movement of the user.

In order to speed-up the insertion and deletion of objects, we should implement these operations in two phases: in the first phase, we allow the modeller the temporary modification of the scene in his memory. Afterwards, in the second phase, we copy and store the modified data structure across the network on remote workstations. This avoids the expensive modification of the sectorgraph across the network, and reduces expensive communication.

In order to speed-up both navigation and manipulation, we should block neighbouring objects. For example, we store all neighbours of an object as a single block of data on a single remote hard disk. If we access the objects, we load all of them into the memory. A single access of the whole block is faster than access to one object after another.

Further problems that should be tackled are the balancing and the management of landmarks. Landmarks are references to fixed positions (objects) of the scene. We can jump to an object directly if we have the landmark of the object. This is especially useful for our system because the data structure does not support point location.

So far, we have distributed the objects randomly or in sequential order on remote hard disks. Alternatively, we can store all objects of a large neighbourhood onto the same hard disk. However, this kind of distribution needs sophisticated balancing algorithms to distribute equally sized parts of the scene evenly among the network.

Bibliography

[1] Howard Abrams, Kent Watsen, and Michael Zyda. Three-tiered interest management for large-scale virtual environments. In *Proc. ACM Symposium on Virtual Reality Software and Technology (VRST '98)*, pages 125–129, 1998.

[2] Howard Allan Abrams. *Extensible Interest Management for Scalable Persistent Distributed Virtual Environments.* PhD thesis, Naval Postgraduate School, Monterey, California, December 1999.

[3] Pankaj K. Agarwal. Range searching. Technical Report CS-1996-05, Duke University, Department of Computer Science, September 1996.

[4] Pankaj K. Agarwal. Range searching. In J. Goodmand and J. O'Rourke, editors, *Handbook of Computational Geometry.* CRC Press, second edition, 1997.

[5] Pankaj K. Agarwal and Jeff Erickson. Geometric range searching and its relatives. *Advances in Discrete and Computational Geometry - Contemporary Mathematics*, 223:1–56, 1999.

[6] Pankaj K. Agarwal, Edward F. Grove, T. M. Murali, and Jeffrey Scott Vitter. Binary space partitions for fat rectangles. *SIAM Journal on Computing*, 29(5):1422–1448, 2000.

[7] Timo Aila and Ville Miettinen. *SurRender Umbra^{TM} Reference Manual.* Hybrid Holding, Eteläinen Makasiinikatu 4, 6th Floor, 00130 Helsinki, Finnland, 2.3 edition, March 2001.

[8] Marc Alexa, Johannes Behr, Daniel Cohen-Or, Shachar Fleishman, David Levin, and Claudio T. Silva. Point set surfaces. In *Proc. 12th IEEE Visualization 2001 Conference (VIS '01)*, pages 21–28, 2001.

[9] Andrei Alexandrescu. *Modern C++ Design - Generic Programming and Design Pattern Applied.* Addison Wesley, 2001.

[10] Daniel Aliaga, Jon Cohen, Andrew Wilson, Eric Baker, Hansong Zhang, Carl Erikson, Kenny Hoff, Tom Hudson, Wolfgang Stuerzlinger, Rui Bastos, Mary Whitton, Fred Brooks, and Dinesh Manocha. MMR: an interactive massive model rendering system using geometric and image-based acceleration. In *Proc. 1999 Symposium on Interactive 3D Graphics*, pages 199–206. ACM Press, 1999.

[11] Daniel Aliaga, Jon Cohen, Andrew Wilson, Hansong Zhang, Carl Erikson, Kenneth Hoff, Thomas Hudson, Wolfgang Stuerzlinger, Eric Baker, Rui Bastos, Mary Whitton, Fred Brooks, and Dinesh Manocha. A framework for the real-time walkthrough of massive models. Technical Report UNC TR#98-013, Department of Computer Science, University of North Carolina at Chapel Hill, March 1998.

[12] Daniel G. Aliaga and Anselmo Lastra. Architectural walkthroughs using portal textures. In *Proc. 8th IEEE Visualization 1997 Conference (VIS '97)*, pages 355–362, 1997.

[13] Ingo Althöfer, Gautam Das, David Dobkin, and Deborah Joseph. Generating sparse spanners for weighted graphs. In *Proc. 2nd Scandinavian Workshop on Algorithm Theory (SWAT '90)*, volume 447 of *LNCS*, pages 26–37, 1990.

[14] Ingo Althöfer, Gautam Das, David Dobkin, Deborah Joseph, and José Soares. On sparse spanners of weighted graphs. *Discrete & Computational Geometry*, 9:81 – 100, 1993.

[15] Sunil Arya, Gautam Das, David M. Mount, Jeffrey S. Salowe, and Michiel Smid. Euclidean spanners: Short, thin, and lanky. In *Proc. 27th ACM Symposium on Theory of Computing (STOC '95)*, pages 489–498, 1995.

[16] Sunil Arya and David M. Mount. Approximate range searching. *Computational Geometry: Theory and Applications*, 17(3-4):135–152, 2000.

[17] Sunil Arya, David M. Mount, and Michiel Smid. Randomized and deterministic algorithms for geometric spanners of small diameter. In *Proc. 35th IEEE Symposium on Foundations of Computer Science (FOCS '94)*, pages 703–712, 1994.

[18] Sunil Arya, David M. Mount, and Michiel Smid. Dynamic algorithms for geometric spanners of small diameter: Randomized solutions. *Computational Geometry: Theory and Applications*, 13(2):91–107, 1999.

[19] Sunil Arya and Michiel Smid. Efficient construction of a bounded degree spanner with low weight. In *Proc. 2nd European Symposium on Algorithms (ESA '94)*, volume 855 of *LNCS*, pages 48–59, 1994.

[20] Sunil Arya and Michiel Smid. Efficient construction of a bounded-degree spanner with low weight. *Algorithmica*, 17(1):33–54, 1997.

[21] Inc. Avid Technology. softimage, data sheet. http://www.softimage.com, 2004.

[22] Dirk Bartz and Claudio Silva. Rendering and visualization in parallel environments. In *Tutorial of EUROGRAPHICS 2001, Tutorial 9*. Eurographics Association, 1995.

[23] Jiří Bittner. Hierarchical techniques for visibility determination. Postgraduate Study Report DC-PSR-99-05, Czech Technical University, Faculty of Electrical Engineering, Department of Computer Science and Engineering, March 1999.

[24] Jiří Bittner, Vlastimil Havran, and Pavel Slavík. Hierarchical visibility culling with occlusion trees. In *Proc. Conference on Computer Graphics International (CGI '98)*, pages 207–219, 1998.

[25] William Blanke, Chandrajit Bajaj, Donald Fussell, and Xiaoyu Zhang. The metabuffer: A scalable multiresolution multidisplay 3-D graphics system using commodity rendering engines. Technical Report TR 2000-16, University of Texas at Austin, 2000.

[26] Ted Boardman. *3ds max 6 Fundamentals*. New Riders Publishing, 2004.

[27] Sven Bormann. *Virtuelle Realität: Genese und Evaluation*. Addison-Wesley Publishing Company, 1994.

[28] Ulrik Brandes and Dagmar Handke. Np-completeness results for minimum planar spanners. *Discrete Mathematics & Theoretical Computer Science*, 3(1):1–10, 1998.

[29] Leizhen Cai and Derek G. Corneil. Tree spanners. *SIAM Journal on Discrete Mathematics*, 8(3):359–387, 1995.

[30] Paul B. Callahan. *Dealing with Higher Dimensions: The Well-Separated Pair Decomposition and Its Applications*. PhD thesis, Johns Hopkins University, Baltimore, Maryland, 1995.

[31] Paul B. Callahan and S. Rao Kosaraju. A decomposition of multi-dimensional point-sets with applications to k-nearest-neighbors and n-body potential fields. In *Proc. 24th ACM Symposium on Theory of Computing (STOC '92)*, pages 546–556, 1992.

[32] Paul B. Callahan and S. Rao Kosaraju. Faster algorithms for some geometric graph problems in higher dimensions. In *Proc. 4th ACM-SIAM Symposium on Discrete Algorithms (SODA '93)*, pages 291–300, 1993.

[33] Paul B. Callahan and S. Rao Kosaraju. A decomposition of multidimensional point sets with applications to k-nearest-neighbors and n-body potential fields. *Journal of the ACM*, 42(1):67–90, 1995.

[34] Rikk Carey and Gavin Bell. *The Annotated VRML 2.0 Reference Manual*. Addison-Wesley Professional, 1997.

[35] Edwin Catmull. *A Subdivision Algorithm for Computer Display of Curved Surfaces*. PhD thesis, University of Utah, Salt Lake City, 1974.

[36] Edwin Catmull. Computer display of curved surfaces. In *Proc. IEEE Conference on Computer Graphics, Pattern Recognition and Data Structures*, pages 11–17, 1975.

[37] Edwin Catmull. Computer display of curved surfaces. In Rosalee Wolfe, editor, *Seminal Graphics: Pioneering Efforts That Shaped The Field*, pages 35–41. ACM Press, 1998.

[38] Alan Chalmers, Tim Davis, Toshi Kato, and Erik Reinhard. Practical parallel processing for todays rendering challenges. In *Course Notes of the 28th Conference on Computer Graphics and Interactive Techniques (SIGGRAPH 2001), Course No. 40*. ACM Press, 2001.

[39] Alan Chalmers and Erik Reinhard. Parallel and distributed photo-realistic rendering. In *Course Notes of the 25th Conference on Computer Graphics and Interactive Techniques (SIGGRAPH 1998), Course No. 3*. ACM Press, 1998.

[40] Bradford Chamberlain, Tony DeRose, Dani Lischinski, David Salesin, and John Snyder. Fast rendering of complex environments using a spatial hierarchy. In *Proc. Graphics Interface '96*, pages 132–141, 1996.

[41] Barun Chandra. Constructing sparse spanners for most graphs in higher dimensions. Technical Report CS 93-05, University of Chicago, Department of Computer Science, May 1993.

[42] Barun Chandra. *Approximation and Online Algorithms for Graph Problems.* PhD thesis, University of Chicago, Department of Computer Science, 1994.

[43] Barun Chandra. Constructing sparse spanners for most graphs in higher dimensions. *Information Processing Letters*, 51(6):289–294, 1994.

[44] Barun Chandra, Gautam Das, Giri Narasimhan, and José Soares. New sparseness results on graph spanners. In *Proc. 8th ACM Symposium on Computational Geometry*, pages 192–201, 1992.

[45] Barun Chandra, Gautam Das, Giri Narasimhan, and José Soares. New sparseness results on graph spanners. Technical Report CS 92-09, University of Chicago, April 1992.

[46] Barun Chandra, Gautam Das, Giri Narasimhan, and José Soares. New sparseness results on graph spanners. *International Journal of Computational Geometry & Applications*, 5:125–144, 1995.

[47] Bernard Chazelle and Leonidas J. Guibas. Fractional cascading: I. A data structuring technique. *Algorithmica*, 1(2):133–162, 1986.

[48] Bernard Chazelle and Leonidas J. Guibas. Fractional cascading: II. applications. *Algorithmica*, 1(2):163–191, 1986.

[49] Baoquan Chen and Minh Xuan Nguyen. POP: A hybrid point and polygon rendering system for large data. In *Proc. 12th IEEE Visualization 2001 Conference (VIS '01)*, pages 45–52, 2001.

[50] Danny Z. Chen, Gautam Das, and Michiel Smid. Lower bounds for computing geometric spanners and approximate shortest paths. In *Proc. 8th Canadian Conference on Computational Geometry (CCCG '96)*, pages 155–160, 1996.

[51] Danny Z. Chen, Gautam Das, and Michiel Smid. Lower bounds for computing geometric spanners and approximate sortest paths. *Discrete Applied Mathematics*, 110(2-3):151–167, 2001.

[52] L. Paul Chew. There is a planar graph almost as good as the complete graph. In *Proc. 2nd ACM Symposium on Computational Geometry*, pages 169–177, 1986.

[53] L. Paul Chew. There are planar graphs almost as good as the complete graph. *Journal of Computer and System Science*, 39(2):205–219, 1989.

[54] P. Cignoni, C. Montani, and R. Scopigno. A comparison of mesh simplification algorithms. *Computer & Graphics*, 22(1):37–54, 1998.

[55] James H. Clark. Hierarchical geometric models for visible surface algorithms. *Communications of the ACM*, 19(10):547–554, 1976.

[56] James H. Clark. Hierarchical geometric models for visible surface algorithms. In Rosalee Wolfe, editor, *Seminal Graphics: Pioneering Efforts That Shaped The Field*, pages 43–50. ACM Press, 1998.

[57] Edith Cohen. Fast algorithms for constructing t-spanners and paths with stretch t. In *Proc. 34th IEEE Symposium on Foundations of Computer Science (FOCS '93)*, pages 648–658, 1993.

[58] Edith Cohen. Fast algorithms for constructing t-spanners and paths with stretch t. *SIAM Journal on Computing*, 28(1):210–236, 1998.

[59] Jonathan Cohen, Amitabh Varshney, Dinesh Manocha, Greg Turk, Hans Weber, Pankaj Agarwal, Frederick Brooks, and William Wright. Simplification envelopes. In *Proc. 23th Conference on Computer Graphics and Interactive Techniques (SIGGRAPH 1996)*, pages 119–128. ACM Press, 1996.

[60] Jonathan D. Cohen, Daniel G. Aliaga, and Weiqiang Zhang. Hybrid simplification: Combining multi-resolution polygon and point rendering. In *Proc. 12th IEEE Visualization 2001 Conference (VIS '01)*, pages 37–44, 2001.

[61] Michael F. Cohen and John R. Wallace. *Radiosity and Realistic Image Synthesis.* Academic Press Professional, 1993.

[62] Daniel Cohen-Or, Yiorgos L. Chrysanthou, Cláudio T. Silva, and Frédo Durand. A survey of visibility for walkthrough applications. *IEEE Transactions on Visualization and Computer Graphics*, 9(3):412–431, 2003.

[63] Daniel Cohen-Or, Yair Mann, and Shachar Fleishman. Deep compression for streaming texture intensive animations. In *Proc. 26th Conference on Computer Graphics and Interactive Techniques (SIGGRAPH 1999)*, pages 261–267, 1999.

[64] Daniel Cohen-Or and Eyal Zadicario. Visibility streaming for network-based walkthroughs. In *Proc. Graphics Interface '98*, pages 1–7, 1998.

[65] Satyan Coorg and Seth Teller. Real-time occlusion culling for models with large occluders. In *Proc. 1997 Symposium on Interactive 3D Graphics*, pages 83–90. ACM Press, April 1997.

[66] Thomas Crockett. Parallel rendering. ICASE Report No. 95-31, NASA Langley Research Center, Institute for Computer Applications in Science and Engineering, Hampton, VA, 1995.

[67] Franklin C. Crow, Gary Demos, Jim Hardy, John Mclaughlin, and Karl Sims. 3D image synthesis on the connection machine. In *Proc. International Conference on Parallel Processing for Computer Vision and Display*, pages 254–269, 1988.

[68] Ross Cunniff. Visualize fx graphics scalable architecture. In *SIGGRAPH Eurographics Graphics Hardware Workshop 2000, Hot3D Session*, pages 29–38, August 2000.

[69] Gautam Das, Paul Heffernan, and Giri Narasimhan. Optimally sparse spanners in 3-dimensional Euclidean space. In *Proc. 9th ACM Symposium on Computational Geometry*, pages 53–62, 1993.

[70] Gautam Das and Paul J. Heffernan. Constructing degree-3 spanners with other sparseness properties. In *Proc. 4th International Symposium on Algorithms and Computation (ISAAC '93)*, volume 762 of *LNCS*, pages 11–20, 1993.

[71] Gautam Das and Paul J. Heffernan. Constructing degree-3 spanners with other sparseness properties. *International Journal of Foundations of Computer Science*, 7(2):121–136, 1996.

[72] Gautam Das and Deborah Joseph. Which triangulations approximate the complete graph? In *Proc. International Symposium on Optimal Algorithms*, volume 401 of *LNCS*, pages 168–192, 1989.

[73] Gautam Das and Giri Narasimhan. A fast algorithm for constructing sparse Euclidean spanners. In *Proc. 10th ACM Symposium on Computational Geometry*, pages 132–139, 1994.

[74] Mark de Berg. Linear size binary space partitions for uncluttered scenes. *Algorithmica*, 28(3):353–366, 2000.

[75] Mark de Berg, Marc van Kreveld, Marc Overmars, and Otfried Schwarzkopf. *Computational Geometry: Algorithms and Applications*. Springer Verlag, 1997.

[76] Jauvane C. de Oliveira and Nicolas D. Georganas. VELVET: An adaptive hybrid architecture for very large virtual environments. *Presence*, 12(6):555–580, 2003.

[77] Michael Deering and David Naegle. The SAGE graphics architecture. In *Proc. 29th Conference on Computer Graphics and Interactive Techniques (SIGGRAPH 2002)*, pages 683–692, 2002.

[78] Discreet. 3ds max, Data Sheet [Data Sheet of 3D Studio Max]. Retrieved April 12, 2000 from http://www.discreet.com.

[79] David P. Dobkin, Steven J. Friedman, and Kenneth J. Supowit. Delaunay graphs are almost as good as complete graphs. In *Proc. 28th IEEE Symposium on Foundations of Computer Science (FOCS '87)*, pages 20–26, 1987.

[80] David P. Dobkin, Steven J. Friedman, and Kenneth J. Supowit. Delaunay graphs are almost as good as complete graphs. *Discrete & Computational Geometry*, 5:399–407, 1990.

[81] Frédo Durand. *3D Visibility: Analytical Study and Applications*. PhD thesis, Université Joseph Fourier, Grenoble I, July 1999.

[82] Frédo Durand. A multidisciplinary survey of visibility. In *ACM SIGGRAPH course notes: Visibility, Problems, Techniques, and Applications*, July 2000.

[83] Frédo Durand, George Drettakis, and Claude Puech. The 3D visibility complex: A new approach to the problems of accurate visibility. In *Rendering Techniques 96, Proc. 7th Eurographics Workshop on Rendering*, pages 245–256, 1996.

[84] Frédo Durand, George Drettakis, and Claude Puech. The 3D visibility complex: A unified data structure for global visibility of scenes of polygons and smooth objects. In *Proc. 9th Canadian Conference on Computational Geometry (CCCG '97)*, 1997.

[85] Matthias Eck, Tony DeRose, Tom Duchamp, Hugues Hoppe, Michael Lounsbery, and Werner Stuetzle. Multiresolution analysis of arbitrary meshes. In *Proc. 22th Conference on Computer Graphics and Interactive Techniques (SIGGRAPH 1995)*, pages 173–182. ACM Press, 1995.

[86] Daniel R. Edelson. Smart pointers: They're smart, but they're not pointers. In *Usenix C++ Conference*, pages 1–20, 1992.

[87] David Eppstein. Spanning trees and spanners. Technical Report 96-16, University of California, Dept. Information and Computer Science, May 1996.

[88] Carl Erikson. Polygonal simplification: An overview. Technical Report TR96-016, University of North Carolina at Chapel Hill, Department of Computer Science, 1996.

[89] John Eyles, John Austin, Henry Fuchs, Trey Greer, and John Paulton. Pixel-planes 4: A summary. In Alphonsus A. M. Kuijk and Wolfgang Strasser, editors, *Advances in computer graphics hardware II, Eurographics '87, 2nd Workshop on Graphics Hardware*, pages 183–207. Springer-Verlag, 1988.

[90] Raphael A. Finkel and Jon Louis Bentley. Quad trees: A data structure for retrieval on composite keys. *Acta Informatica*, 4(1-9), 1974.

[91] Matthias Fischer, Tamás Lukovszki, and Martin Ziegler. Geometric searching in walkthrough animations with weak spanners in real time. In *Proc. 6th European Symposium on Algorithms (ESA '98)*, volume 1461 of *LNCS*, pages 163–174, 1998.

[92] Matthias Fischer, Tamás Lukovszki, and Martin Ziegler. A network based approach for realtime walkthrough of massive models. In *Proc. 2nd Workshop on Algorithms Engineering (WAE '98)*, pages 133–142, 1998.

[93] Matthias Fischer, Tamás Lukovszki, and Martin Ziegler. Partitioned neighborhood spanners of minimal outdegree. In *Proc. 11th Canadian Conference on Computational Geometry (CCCG '99)*, pages 47–50, 1999.

[94] Matthias Fischer, Friedhelm Meyer auf der Heide, and Willy-B. Strothmann. Dynamic data structures for realtime management of large geometric scenes. In *Proc. 5th European Symposium on Algorithms (ESA '97)*, volume 1284 of *LNCS*, pages 157–170, 1997.

[95] James D. Foley, Andries van Dam, Steven K. Feiner, and John F. Hughes. *Computer Graphics: Principles and Practice*. Addison Wesley, 1996.

[96] Henry Fuchs, Zvi M. Kedem, and Bruce F. Naylor. On visible surface generation by a priori tree structures. In *Proc. 7th Conference on Computer Graphics and Interactive Techniques (SIGGRAPH 1980)*, pages 124–133. ACM Press, 1980.

[97] Thomas A. Funkhouser. RING: A client-server system for multi-user virtual environments. In *Proc. 1995 Symposium on Interactive 3D Graphics*, pages 85–92, 209, 1995.

[98] Thomas A. Funkhouser and Carlo H. Séquin. Adaptive display algorithm for interactive frame rates during visualization of complex virtual environments. In *Proc. 20th Conference on Computer Graphics and Interactive Techniques (SIGGRAPH 1993)*, pages 247–254. ACM Press, 1993.

[99] Thomas A. Funkhouser, Carlo H. Séquin, and Seth J. Teller. Management of large amounts of data in interactive building walkthroughs. In *Proc. 1992 Symposium on Interactive 3D Graphics*, pages 11–20. ACM Press, 1992.

[100] Michael Garland. Multiresolution modeling: Survey & future opportunities. In *STAR - State of the Art Reports, EUROGRAPHICS '99*. Eurographics Association, 1999.

[101] Michael Garland. *Quadric-Based Polygonal Surface Simplification*. PhD thesis, Carnegie Mellon University, School of Computer Science, 1999.

[102] Michael Garland and Paul S. Heckbert. Surface simplification using quadric error metrics. In *Proc. 24th Conference on Computer Graphics and Interactive Techniques (SIGGRAPH 1997)*, pages 209–216. ACM Press, 1997.

[103] Andrew S. Glassner, editor. *An Introduction to Ray Tracing*. Morgan Kaufmann Publishers, 1989.

[104] Dan Gordon and Shuhong Chen. Front-to-back display of BSP trees. *IEEE Computer Graphics and Applications*, 11(5):79–85, 1991.

[105] Henri Gouraud. Continuous shading of curved surfaces. *IEEE Transactions on Computers*, C-20(6):623–629, 1971.

[106] Henri Gouraud. Continuous shading of curved surfaces. In Rosalee Wolfe, editor, *Seminal Graphics: Pioneering Efforts That Shaped The Field*, pages 87–93. ACM Press, 1998.

[107] Naga K. Govindaraju, Avneesh Sud, Sugn-Eui Yoon, and Dinesh Manocha. Parallel occlusion culling for interactive walkthroughs using multiple gpus. In *Proc. IEEE Workshop on Commodity Based Visualization Clusters*, 2002.

[108] Ned Greene, Michael Kass, and Gavin Miller. Hierarchical Z-buffer visibility. In *Proc. 20th Conference on Computer Graphics and Interactive Techniques (SIGGRAPH 1993)*, pages 231–238. ACM Press, 1993.

[109] Chris Greenhalgh and Steve Benford. MASSIVE: a distributed virtual reality system incorporating spatial trading. In *Proc. 15th International Conference on Distributed Computing Systems (DCS'95)*, pages 27–34, 1995.

[110] Markus H. Gross, Oliver G. Staadt, and Roger Gatti. Efficient triangular surface approximations using wavelets and quadtree data structure. *IEEE Transactions on Visualization and Computer Graphics*, 2(2):130–143, 1996.

[111] J. P. Grossman. Point sample rendering. Master's thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, August 1998.

[112] J. P. Grossman and William J. Dally. Point sample rendering. In *Rendering Techniques 98, Proc. 9th Eurographics Workshop on Rendering*, pages 181–192, 1998.

[113] Digital Display Working Group. Digital visual interface 1.0 specification. Retrieved April 12, 2004 from http.//www.ddwg.org.

[114] Prosenjit Gupta, Ravi Janardan, and Michiel Smid. On intersection searching problems involving curved objects. In *Proc. 4rd Scandinavian Workshop on Algorithm Theory (SWAT '94)*, pages 183–194, 1994.

[115] Dagmar Handke. *Graphs with Distance Guarantees.* PhD thesis, Universität Konstanz, Fakultät für Mathematik und Informatik, 1999.

[116] Dagmar Handke. Independent tree spanners: Fault-tolerant spanning trees with constant distance guarantees. *Discrete Applied Mathematics*, 108(1-2):105–127, 2001.

[117] Jan C. Hardenbergh, Gavin Bell, and Mark D. Pesce. VRML: Using 3D to surf the web. In *Course Notes of the 22th Conference on Computer Graphics and Interactive Techniques (SIGGRAPH 1995) Course No. 12.* ACM Press, 1995.

[118] Taosong He, Lichan Hong, Arie Kaufman, Amitabh Varshney, and Sidney Wang. Voxel based object simplification. In *Proc. 6th IEEE Visualization 1995 Conference (VIS '95)*, pages 296–303. IEEE Computer Society, 1995.

[119] Donald Hearn and M. Pauline Baker. *Computer Graphics.* Prentice Hall, 1994.

[120] Paul S. Heckbert. Survey of texture mapping. *IEEE Computer Graphics and Applications*, pages 56–67, November 1986.

[121] Paul S. Heckbert and Michael Garland. Multiresolution modeling for fast rendering. In *Proc. Graphics Interface '94*, pages 43–50, 1994.

[122] Paul S. Heckbert and Michael Garland. Survey of polygonal surface simplification algorithms. In *Multiresolution Surface Modeling Course, SIGGRAPH '97.* ACM Press, 1997.

[123] Mark Henne, Hal Hickel, Ewan Johnson, and Sonoko Konishi. The making of toy story. In *Proc. 41st IEEE International Computer Conference COMPCON 96*, pages 463–468, 1996.

[124] Gerd Hesina, Dieter Schmalstieg, Anton Fuhrmann, and Werner Purgathofer. Distributed open inventor: A practical approach to distributed 3D graphics. In *Proc. ACM Symposium on Virtual Reality Software and Technology (VRST '99)*, pages 74–81, 1999.

[125] Rudy Hirschheim. Not just fun and games anymore.
http://disc.cba.uh.edu/r̃hirsch/fall96/doug.htm, 2004.

[126] Hugues Hoppe. Progressive meshes. In *Proc. 23th Conference on Computer Graphics and Interactive Techniques (SIGGRAPH 1996)*, pages 99–108, 1996.

[127] Hugues Hoppe. View-dependent refinement of progressive meshes. In *Proc. 24th Conference on Computer Graphics and Interactive Techniques (SIGGRAPH 1997)*, pages 189–198. ACM Press, 1997.

[128] Hugues Hoppe, Tony DeRose, Tom Duchamp, John McDonald, and Werner Stuetzle. Mesh optimization. In *Proc. 20th Conference on Computer Graphics and Interactive Techniques (SIGGRAPH 1993)*, pages 19–26. ACM Press, August 1993.

[129] Greg Humphreys, Matthew Eldridge, Ian Buck, Gordan Stoll, Matthew Everett, and Pat Hanrahan. WireGL: A scalable graphics system for clusters. In *Proc. 28th Conference on Computer Graphics and Interactive Techniques (SIGGRAPH 2001)*, pages 129–140, 2001.

[130] Greg Humphreys, Mike Houston, Ren Ng, Randall Frank, Sean Ahern, Peter D. Kirchner, and James T. Klosowski. Chromium: A stream-processing framework for interactive rendering on clusters. In *Proc. 29th Conference on Computer Graphics and Interactive Techniques (SIGGRAPH 2002)*, pages 693–702, 2002.

[131] IBM. 3D Interaction Accelerator (3DIX).
http://www.research.ibm.com/3dix,
http://domino.research.ibm.com/comm/wwwr_thinkresearch.nsf/pages
/news296.html, 2004.

[132] Aravind Kalaiah and Amitabh Varshney. Differential point rendering. In *Rendering Techniques '01, Proc. 12th Eurographics Workshop on Rendering*, pages 139–150, 2001.

[133] A. Kaufman. Volume visualization: Principles and advances. In *Course Notes of the 24th Conference on Computer Graphics and Interactive Techniques (SIGGRAPH 1997)*, 1997.

[134] J. Mark Keil. Approximating the complete Euclidean graph. In *Proc. 1st Scandinavian Workshop on Algorithm Theory (SWAT '88)*, volume 318 of *LNCS*, pages 208–213, 1988.

[135] J. Mark Keil and Carl A. Gutwin. Classes of graphs which approximate the complete Euclidean graph. In *Discrete & Computational Geometry*, volume 7, pages 13–28, 1992.

[136] David G. Kirkpatrick. A note on delaunay and optimal triangulations. *Information Processing Letters*, 10(3):127–128, 1980.

[137] Jan Klein, Jens Krokowski, Matthias Fischer, Michael Wand, Rolf Wanka, and Friedhelm Meyer auf der Heide. The randomized sample tree: A data structure for interactive walkthrough. In *Proc. ACM Symposium on Virtual Reality Software and Technology (VRST '02)*, pages 137 – 146, 2002.

[138] Guy Kortsarz and David Peleg. Generating sparse 2-spanners. In *Proc. 3rd Scandinavian Workshop on Algorithm Theory (SWAT '92)*, pages 73–82, 1992.

[139] Guy Kortsarz and David Peleg. Generating low-degree 2-spanners. In *Proc. 5th ACM-SIAM Symposium on Discrete Algorithms (SODA '94)*, pages 556–563, 1994.

[140] Guy Kortsarz and David Peleg. Generating sparse 2-spanners. *Journal of Algorithms*, 17:222–236, 1994.

[141] Guy Kortsarz and David Peleg. Generating low-degree 2-spanners. *SIAM Journal on Computing*, 27(5):1438–1456, 1998.

[142] Christos Levcopoulos and Andrzej Lingas. There are planar graphs almost as good as the complete graphs and as short as minimum spanning trees. In *Proc. International Symposium on Optimal Algorithms*, volume 401 of *LNCS*, pages 9–13, 1989.

[143] Christos Levcopoulos and Andrzej Lingas. There are planar graphs almost as good as the complete graphs and almost as cheap as minimum spanning trees. *Algorithmica*, 8:251–256, 1992.

[144] Christos Levcopoulos, Giri Narasimhan, and Michiel Smid. Efficient algorithms for constructing fault-tolerant geometric spanners. In *Proc. 30th ACM Symposium on Theory of Computing (STOC '98)*, pages 186–195, 1998.

[145] Christos Levcopoulos, Giri Narasimhan, and Michiel Smid. Improved algorithms for constructing fault-tolerant spanners. *Algorithmica*, 32(1):144–156, 2002.

[146] Marc Levoy and Turner Whitted. The use of points as a display primitive. Technical Report TR 85-022, University of North Carolina at Chapel Hill, 1985.

[147] Arthur L. Liestman and Thomas C. Shermer. Grid spanners. *Networks*, 23:123–133, 1993.

[148] Tony T. Y. Lin and Mel Slater. Stochastic ray tracing using SIMD processor arrays. *The Visual Computer*, 7(4):187–199, 1991.

[149] John Michael Lounsbery. *Multiresolution Analysis for Surfaces of Arbitrary Topological Type*. PhD thesis, University of Washington, Department of Computer Science and Engineering, 1994.

[150] Kok-Lim Low and Tiow-Seng Tan. Model simplification using vertex-clustering. In *Proc. 1997 Symposium on Interactive 3D Graphics*, pages 75–81. ACM Press, 1997.

[151] David Luebke. A survey of polygonal simplification algorithms. Technical Report TR97-045, University of North Carolina at Chapel Hill, Department of Computer Science, 1997.

[152] David Luebke and Chris Georges. Portals and mirrors: Simple, fast evaluation of potentially visible sets. In *Proc. 1995 Symposium on Interactive 3D Graphics*, pages 105–106. ACM Press, 1995.

[153] David Luebke, Martin Reddy, Jonathan D. Cohen, Amitabh Varshney, Benjamin Watson, and Robert Huebner. *Level of Detail For 3D Graphics.* Morgan Kaufmann Publishers, 2003.

[154] David P. Luebke. *View-Dependent Simplification of Arbitrary Polygonal Environments.* PhD thesis, University of North Carolina at Chapel Hill, Department of Computer Science, 1998.

[155] Michael R. Macedonia, Michael J. Zyda, David R. Pratt, Paul T. Barham, and Steven Zeswitz. NPSNET: A network software architecture for large scale virtual environments. *Presence*, 3(4):265–287, 1994.

[156] Paulo W. C. Maciel and Peter Shirley. Visual navigation of large environments using textured clusters. In *Proc. 1995 Symposium on Interactive 3D Graphics*, pages 95–102. ACM Press, 1995.

[157] Yair Mann and Daniel Cohen-Or. Selective pixel transmission for navigating in remote virtual environments. *EUROGRAPHICS '97, Computer Graphics Forum*, 16(3):201–206, 1997.

[158] Jiří Matoušek. Geometric range searching. *ACM Computing Surveys*, 26(4):422–461, 1994.

[159] Tomas Möller and Eric Haines. *Real-Time Rendering.* A K Peters, Natick, Massachusetts, 1999.

[160] Steven Molnar, Michael Cox, David Ellsworth, and Henry Fuchs. A sorting classification of parallel rendering. *IEEE Computer Graphics and Applications*, 14(4):23–32, 1994.

[161] Steven Molnar, John Eyles, and John Poulton. Pixelflow: High-speed rendering using image composition. In *Proc. 19th Conference on Computer Graphics and Interactive Techniques (SIGGRAPH 1992)*, pages 231–240, 1992.

[162] Ketan Mulmuley. *Computational Geometry: An Introduction Through Randomized Algorithms.* Prentice Hall, Englewood Cliffs, 1994.

[163] Beatrice Ng, Antonio Si, Rynson W.H. Lau, and Frederick W.B. Li. A multi-server architecture for distributed virtual walkthrough. In *Proc. ACM Symposium on Virtual Reality Software and Technology (VRST '02)*, pages 163–170, 2002.

[164] NVIDIA. GeForce 6800 - Performance Specification[Data Sheet of GeForce 6800]. Retrieved July 17, 2004 from http://www.nvidia.com.

[165] NVIDIA. GeForce4 Ti - Product Overview [Data Sheet of GeForce4 Ti]. Retrieved April 12, 2000 from http://www.nvidia.com.

[166] Mike Paterson and F. Frances Yao. Efficient binary space partitions for hidden-surface removal and solid modeling. *Discrete & Computational Geometry*, 5(5):485–503, 1990.

[167] David Peleg and A.A. Schffer. Graph spanners. *Journal of Graph Theory*, 13:99–116, 1989.

[168] David Peleg and Jeffrey D. Ullman. An optimal synchronizer for the hypercube. In *Proc. 6th ACM Symposium on Principles of Distributed Computing (PODC'87)*, pages 77–85, 1987.

[169] David Peleg and Jeffrey D. Ullman. An optimal synchronizer for the hypercube. *SIAM Journal on Computing*, 18:740–747, 1989.

[170] Hanspeter Pfister, Matthias Zwicker, Jeroen van Baar, and Markus Gross. Surfels: Surface elements as rendering primitives. In *Proc. 27th Conference on Computer Graphics and Interactive Techniques (SIGGRAPH 2000)*, pages 335–342, 2000.

[171] Bui-Tuong Phong. Illumination for computer generated pictures. *Communications of the ACM*, 18(6):311–317, 1975.

[172] Bui-Tuong Phong. Illumination for computer generated pictures. In Rosalee Wolfe, editor, *Seminal Graphics: Pioneering Efforts That Shaped The Field*, pages 95–101. ACM Press, 1998.

[173] Harry Plantinga and Charles R. Dyer. Visibility, occlusion, and the aspect graph. *International Journal of Computer Vision*, 5(2):137–160, 1990.

[174] Jovan Popović and Hugues Hoppe. Progressive simplicial complexes. In *Proc. 24th Conference on Computer Graphics and Interactive Techniques (SIGGRAPH 1997)*, pages 217–224. ACM Press, 1997.

[175] Franco P. Preparata and Michael I. Shamos. *Computational Geometry: An Introduction.* Springer-Verlag, 1985.

[176] Enrico Puppo and Roberto Scopigno. Simplification, lod and multiresolution - principles and applications. In *Eurographics '97 Tutorial Notes, PS97 TN4*. Eurographics Association, 1997.

[177] E. Reinhard, A. G. Chalmers, and F. W. Jansen. Overview of parallel photo-realistic graphics. In *STAR - State of the Art Reports, EUROGRAPHICS 1998*, pages 1–25. Eurographics Association, 1998.

[178] Amit Reisman, Craig Gotsman, and Assaf Schuster. Parallel progressive rendering of animation sequences at interactive rates on distributed-memory machines. In *Proc. IEEE Parallel Rendering Symposium (PRS '97)*, pages 39–47, 1997.

[179] Jarek Rossignac and Paul Borrel. Multi-resolution 3D approximations for rendering complex scenes. In *Proc. Geometric Modeling in Computer Graphics*, pages 455 – 465. Springer Verlag, 1993.

[180] Jim Ruppert and Raimund Seidel. Approximating the d-dimensional complete Euclidean graph. In *Proc. 3rd Canadian Conference on Computational Geometry (CCCG '91)*, pages 207 – 210, 1991.

[181] Szymon Rusinkiewicz and Marc Levoy. QSplat: A multiresolution point rendering system for large meshes. In *Proc. 27th Conference on Computer Graphics and Interactive Techniques (SIGGRAPH 2000)*, pages 343–352, 2000.

[182] Szymon Rusinkiewicz and Marc Levoy. Streaming QSplat: A viewer for networked visualization of large, dense models. In *Proc. 2001 Symposium on Interactive 3D Graphics*, pages 63–68, 2001.

[183] J. S. Salowe. Constructing multidimensional spanner graphs. *International Journal of Computational Geometry & Applications*, 1(2):99–107, 1991.

[184] Jeffrey S. Salowe:. On euclidean spanner graphs with small degree. In *Proc. 8th ACM Symposium on Computational Geometry*, pages 186–191, 1992.

[185] Jeffrey S. Salowe. Euclidean spanner graphs with degree four. *Discrete Applied Mathematics*, 54(1):55–66, 1994.

[186] Rudrajit Samanta, Thomas Funkhouser, and Kai Li. Parallel rendering with k-way replication. In *Proc. IEEE Symposium on Parallel and Large-Data Visualization and Graphics (PVG 2001)*, pages 75–84, 2001.

[187] Rudrajit Samanta, Thomas Funkhouser, Kai Li, and Jaswinder Pal Singh. Hybrid sort-first and sort-last parallel rendering with a cluster of pcs. In *Proc. SIGGRAPH/EUROGRAPHICS Workshop on Graphics Hardware*, pages 97–108, 2000.

[188] Hanan Samet. The quadtree and related hierarchical data structures. *ACM Computing Surveys*, 16(2):187–260, 1984.

[189] Hanan Samet. *Applications of Spatial Data Structures: Computer Graphics, Image Processing, and GIS.* Addison-Wesley, 1990.

[190] Hanan Samet. *The Design and Analysis of Spatial Data Structures.* Addison-Wesley, 1990.

[191] Hanan Samet and Robert E. Webber. Storing a collection of polygons using quadtrees. *ACM Transactions on Graphics*, 4(3):182–222, 1985.

[192] Hanan Samet and Robert E. Webber. Hierarchical data structures and algorithms for computer graphics. Part I. *IEEE Computer Graphics and Applications*, 8(3):48–68, 1988.

[193] Hanan Samet and Robert E. Webber. Hierarchical data structures and algorithms for computer graphics. Part II. *IEEE Computer Graphics and Applications*, 8(4):59–75, July 1988.

[194] Dieter Schmalstieg. *The Remote Rendering Pipeline - Managing Geometry and Bandwidth in Distributed Virtual Environments.* PhD thesis, Technische Universität Wien, Technisch-Naturwissenschaftliche Fakultät, December 1997.

[195] Dieter Schmalstieg and Michael Gervautz. Demand-driven geometry transmission for distributed virtual environments. *EUROGRAPHICS '96, Computer Graphics Forum*, 15(3):421–432, 1996.

[196] William J. Schroeder, Jonathan A. Zarge, and William E. Lorensen. Decimation of triangle meshes. In *Proc. 19th Conference on Computer Graphics and Interactive Techniques (SIGGRAPH 1992)*, pages 65–70. ACM Press, 1992.

[197] Otfried Schwarzkopf and Jules Vleugels. Range searching in low-density environments. *Information Processing Letters*, 60(3):121–127, 1996.

[198] Jonathan Mark Sewell. *Managing Complex Models for Computer Graphics.* PhD thesis, University of Cambridge, Queens' College, March 1996.

[199] Jonathan Shade, Dani Lischinski, David H. Salesin, Tony DeRose, and John Snyder. Hierarchical image caching for accelerated walkthroughs of complex environments. In *Proc. 23th Conference on Computer Graphics and Interactive Techniques (SIGGRAPH 1996)*, pages 75 – 82. ACM Press, 1996.

[200] Chris Shaw, Mark Green, Jiandong Liang, and Yunqi Sun. Decoupled simulation in virtual reality with the MR toolkit. *ACM Transactions on Information Systems*, 11(3):287–317, 1993.

[201] Peter Shirley and Keith Morley. *Realistic Ray Tracing.* A K Peters, Natick, Massachusetts, 2003.

[202] Heung-Yeung Shum and Sing Bing Kang. A review of image-based rendering techniques. Technical report, Microsoft Research, 1999.

[203] Silicon Graphics, Inc. Cosmo3D [Homepage of Cosmo3D]. Retrieved January 23, 2001 from http://www.sgi.com/software/cosmo.

[204] François X. Sillion and Claude Puech. *Radiosity & Global Illumination.* Morgan Kaufmann Publishers, 1994.

[205] Sandeep Singhal and Michael Zyda. *Networked Virtual Environments - Design and Implementation.* Addison Wesley, 1999.

[206] José Soares. Approximating Euclidean distances by small degree graphs. Technical Report CS 92-05, University of Chicago, Department of Computer Science, March 1992.

[207] José Soares. *Graph Spanners.* PhD thesis, University of Chicago, Department of Computer Science, 1992.

[208] José Soares. Graph spanners: A survey. Technical Report CS 92-12, University of Chicago, Department of Computer Science, July 1992.

[209] José Soares. Approximating Euclidean distances by small degree graphs. *Discrete & Computational Geometry*, 11:213–233, 1994.

[210] Id Software. DOOM [Computer Game]. Retrieved April 12, 2004 from http://www.idsoftware.com, http://www.doom3.com.

[211] Marc Stamminger and George Drettakis. Interactive sampling and rendering for complex and procedural geometry. In *Rendering Techniques '01, Proc. 12th Eurographics Workshop on Rendering*, pages 151–162, 2001.

[212] Marc Stamminger, Jörg Haber, Hartmut Schirmacher, and Hans-Peter Seidel. Walkthroughs with corrective texturing. In *Rendering Techniques '00, Proc. 11th Eurographics Workshop on Rendering*, pages 377–388, 2000.

[213] Amthony Steed and Roula Abou-Haidar. Partitioning crowded virtual environments. In *Proc. 10th ACM Symposium on Virtual Reality Software and Technology (VRST '03)*, pages 7–14, 2003.

[214] A. James Stewart and Tasso Karkanis. Computing the approximate visibility map, with applications to form factors and discontinuity meshing. In *Rendering Techniques 98, Proc. 9th Eurographics Workshop on Rendering*, pages 57–68, 1998.

[215] Gordon Stoll, Matthew Eldridge, Dan Patterson, Art Webb, Steven Berman, Richard Levy, Chris Caywood, Milton Taveira, Stephen Hunt, and Pat Hanrahan. Lightning-2: A high-performance display subsystem for PC clusters. In *Proc. 28th Conference on Computer Graphics and Interactive Techniques (SIGGRAPH 2001)*, pages 141–148, 2001.

[216] Thomas Strothotte. *Computational Visualization. Graphics, Abstraction and Interactivity.* Springer Verlag, 1998.

[217] Oded Sudarsky. *Dynamic Scene Occlusion Culling.* PhD thesis, Technion - Israel Institute of Technology, January 1998.

[218] Seth J. Teller and Carlo H. Séquin. Visibility preprocessing for interactive walkthroughs. In *Proc. 18th Conference on Computer Graphics and Interactive Techniques (SIGGRAPH 1991)*, pages 61–70. ACM Press, 1991.

[219] Enric Torres. Optimization of the binary space partition algorithm (BSP) for the visualization of dynamic scenes. In *EUROGRAPHICS '90, Computer Graphics Forum*, pages 507–518, 1990.

[220] Daniéle Tost. An algorithm of hidden surface removal based on frame-to-frame coherence. In *EUROGRAPHICS '91, Computer Graphics Forum*, pages 261–273, 1991.

[221] Greg Turk. Re-tiling polygonal surfaces. In *Proc. 19th Conference on Computer Graphics and Interactive Techniques (SIGGRAPH 1992)*, pages 55–64. ACM Press, July 1992.

[222] Pravin M. Vaidya. A sparse graph almost as good as the complete graph on points in k dimensions. *Discrete & Computational Geometry*, 6(4):369–381, 1991.

[223] Ingo Wald and Philipp Slusallek. State of the art in interactive ray tracing. In *STAR - State of the Art Reports, EUROGRAPHICS 2001*, pages 21–42. Eurographics Association, September 2001.

[224] Ingo Wald, Philipp Slusallek, and Carsten Benthin. Interactive distributed ray-tracing of highly complex models. In *Rendering Techniques '01, Proc. 12th Eurographics Workshop on Rendering*, pages 274 –285, 2001.

[225] Ingo Wald, Philipp Slusallek, Carsten Benthin, and Markus Wagner. Interactive rendering with coherent ray tracing. In *EUROGRAPHICS '01, Computer Graphics Forum*, 2001.

[226] Michael Wand, Matthias Fischer, and Friedhelm Meyer auf der Heide. Randomized point sampling for output-sensitive rendering of complex dynamic scenes. Technical Report tr-ri-00-217, University of Paderborn, November 2000.

[227] Michael Wand, Matthias Fischer, Ingmar Peter, Friedhelm Meyer auf der Heide, and Wolfgang Straßer. The randomized z-buffer algorithm: Interactive rendering of highly complex scenes. In *Proc. 28th Conference on Computer Graphics and Interactive Techniques (SIGGRAPH 2001)*, pages 361–370, 2001.

[228] Colin Ware. *Information Visualization: Perception for Design.* Morgan Kaufmann Publishers, second edition, 2004.

[229] Alan Watt. *3D Computer Graphics.* Addison-Wesley, third edition, 1999.

[230] Young C. Wee, Seth Chaiken, and Dan E. Willard. General metrics and angle restricted Voronoi diagrams. In *Proc. 1st Canadian Conference on Computational Geometry (CCCG '89)*, August 1989.

[231] Young C. Wee, Seth Chaiken, and Dan E. Willard. Computing geographic nearest neighbors using monotone matrix searching. In *Proc. ACM Conference on Cooperation*, pages 49–55. ACM Press, 1990.

[232] Young C. Wee, Seth Chaiken, and Dan E. Willard. On the angle restricted nearest neighbor problem. *Information Processing Letters*, 34(2):71–76, 1990.

[233] Josie Wernecke. *The Inventor Mentor: Programming Object-Oriented 3D Graphics with Open Inventor.* Addison-Wesley, 1994.

[234] Michael Wimmer, Peter Wonka, and François X. Sillion. Point-based impostors for real-time visualization. In *Rendering Techniques '01, Proc. 12th Eurographics Workshop on Rendering*, pages 163–176, 2001.

[235] Andrew P. Witkin and Paul S. Heckbert. Using particles to sample and control implicit surfaces. In *Proc. 21th Conference on Computer Graphics and Interactive Techniques (SIGGRAPH 1994)*, pages 269–277, 1994.

[236] Craig M. Wittenbrink. Survey of parallel volume rendering algorithms. In *Proc. International Conference on Parallel Distributed Processing Techniques and Applications (PDPTA 98)*, pages 1329–1336, 1998.

[237] Mason Woo, Jackie Neider, Tom Davis, and Dave Shreiner. *OpenGL Programming Guide.* Addison Wesley, third edition edition, 1999.

[238] L. World. Low-end system animates the depths in seaquest. *IEEE Computer Graphics and Applications*, 13(6):93, 1993.

[239] Julie C. Xia and Amitabh Varshney. Dynamic view-dependent simplification for polygonal models. In *Proc. 7th IEEE Visualization 1996 Conference (VIS '96)*, pages 327–334. IEEE Computer Society Press, 1996.

[240] R. Yagel. Volume viewing: State of the art survey. In *Course Notes of the 20th Conference on Computer Graphics and Interactive Techniques (SIGGRAPH 1993), Course No. 40*, pages 109–129. ACM Press, 1993.

[241] Andrew Chi-Chih Yao. On constructing minimum spanning trees in k-dimensional spaces and related problems. *SIAM Journal on Computing*, 11:721–736, 1982.

[242] Hansong Zhang. *Effective Occlusion Culling for the Interactive Display of Arbitrary Models.* PhD thesis, The University of North Carolina at Chapel Hill, 1998.

[243] Hansong Zhang, Dinesh Manocha, Thomas Hudson, and Kenneth E. Hoff III. Visibility culling using hierarchical occlusion maps. In *Proc. 24th Conference on Computer Graphics and Interactive Techniques (SIGGRAPH 1997)*, pages 77–88, 1997.

[244] Matthias Zwicker, Markus H. Gross, and Hanspeter Pfister. A survey and classification of real time rendering methods. Technical Report CS 332, Eidgenössische Technische Hochschule Zürich, Switzerland, December 1999.

[245] Matthias Zwicker, Markus H. Gross, and Hanspeter Pfister. A survey and classification of real time rendering methods. Technical Report 2000-09, Mitsubishi Electric Research Laboratories, Cambridge Research Center, March 2000.

[246] Matthias Zwicker, Hanspeter Pfister, Jeroen van Baar, and Markus Gross. EWA volume splatting. In *Proc. 12th IEEE Visualization 2001 Conference (VIS '01)*, pages 29–36, 2001.

[247] Michael J. Zyda, David R. Pratt, James G. Monahan, and Kalin P. Wilson. NPSNET: Constructing a 3d virtual world. In *Proc. 1992 Symposium on Interactive 3D Graphics*, pages 147–156. ACM Press, 1992.