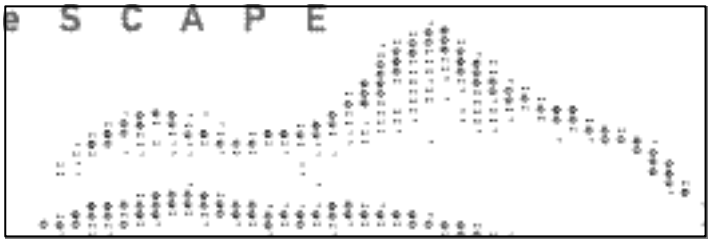


eSCAPE systems, techniques and infrastructures

.

Document ID	eSCAPE -D5.1
Status	Final
Type	Deliverable
Version	1.0
Date	September 1999
Editors	Steve Pettifer
Task	5.1

Project coordinator:

Tom Rodden
Computing Department
University of Lancaster
Lancaster LA1 4YR
United Kingdom
Phone: (+44) 524 593 823
Fax: (+44) 524 593 608
Email: tom@comp.lancs.ac.uk

The eSCAPE project comprises the following institutions:

Swedish Institute for Computer Science (SICS), Stockholm, Sweden
University of Lancaster, Lancaster, United Kingdom (Coordinating Partner)
University of Manchester, Manchester, United Kingdom
ZKM, Germany

Editors of this report:

Steve Pettifer, Manchester University

ISBN 1 86220 081 5
Lancaster University, 1999
This report is available from http://escape.lancs.ac.uk/.

Table of contents

Chapter 1
Introduction and Overview

Steve Pettifer
The University of Manchester

This volume serves as a technical annexe to the first three components of the eSCAPE Year 2 Deliverables and describes the systems, techniques and infrastructures that have been developed during this second year of research. Detail that, for the sake of maintaining a clear narrative describing the two main thematic demonstrators, has by necessity been excluded from deliverables 4.0, 4.1 and 4.2, is presented here as a series of technical reports. The document contains major pieces from all four research sites, being a compilation of work from 20 authors, and representing more broadly contributions of a large number of academics, artists and engineers.

Deliverable structure

The reports contained in this volume are presented in three sections. First, in Chapter 2 we describe in detail the technology and form of the commissioned artworks that have been the subject of the studies in the other deliverables, and which have informed the design of the thematic places described in Deliverables 4.1 and 4.2. Second, in chapters 3 and 4 we present the algorithms associated with the abstract electronic landscape and the modifications made to the DIVE Virtual Reality System to accommodate this new environment. Finally, Chapters 5 to 7 contain a collection of techniques associated with the theme of the physical electronic landscape.

The first section of this report represents what is in the most an overview of completed work. Thus it contains, where appropriate, reflections of the engineers and artists involved in the pieces with possible directions for future work. The techniques presented in the later two sections represent ongoing work in its various stages.

Section 1: The technology of the Artworks

Alongside the development of the two thematic places, this second year of the project has seen the design, implementation and ethnographic study of a number of multi-media art installations. The studies of these works, and the implications for future design obtained by observing the citizen interacting with these in the public arena have been described in detail in the companion volumes: the emphasis here is instead on the technical challenges faced in the pieces' realisation.

The chapter describes:

- **10-dencies | Sao-Paulo** in which urban designers and the inhabitants of these urban areas can interact with information describing their environment.
- The **Force Feedback table**, a novel interaction device inspired by the 10-dencies work, which combines large projected output with consistent haptic and kinaesthetic feedback.
- The **Distributed Legible City**, an evolution of a previous stand-alone installation in which members of the public can share a tour around three esoteric virtual cityscapes using immersive virtual reality technology.
- **Nuzzel Afar**, an exploration of interconnected spaces and the relationships between their inhabitants and their avatars.
- **Place – A user's manual**, in which a landscape is navigated from within a 360 degree panoramic environment.
- **The Web Planetarium & The Eve Dome**, where the initial version of the Web Planetarium developed in project Year 1 finds expression in The Extended Virtual Environment (EVE).

Section 2: The technology of the abstract electronic landscape

The abstract electronic landscape is described at an application level in Deliverable 4.1. The component technology of the abstract electronic landscape has focussed on the development of algorithms for setting out and visualising the relationships and interconnections between items of information in the abstract landscape, and for integrating the software platform with standard internet protocols. Including direct influences from the artworks, and in particular the particle-based visualisations of dynamic data from 10-dencies, the abstract electronic landscape work has developed automated placement algorithms for arranging the landscape, as well as techniques for highlighting regions containing similar information. The platform development has included the integration of the DIVE Virtual Reality platform with Java technology. Chapter 3 describes the development of the placement and region generation algorithms, and Chapter 4 includes details of the Java developments.

Section 3: The technology of the physical electronic landscape

The physical electronic landscape is described at an application level in Deliverable 4.2. The final three chapters of this volume contain a collection of techniques based around the theme of the cityscape, and which are suitable for integration into the final physical electronic landscape demonstrator. These include methods for:

- Automatically generating different styles of city-like environments based on urban theory
- Assisting way-finding within the cityscape, and
- Populating a virtual cityscape with configurable 'lightweight agents'

Section One
The Technology of the artworks

Chapter 2
Contributing Technologies

SICS, ZKM, Lancaster University, The University of Manchester

The second year of the eSCAPE project has seen the commissioning and construction of a number of multi-media installations and artworks, which in turn have inspired and informed the two thematic places. In this chapter we collect together descriptions of the various software systems that drive these installations, and the novel input and output devices developed.

10-dencies | Sao Paulo

Detlev Schwabe, ZKM

The system developed for the 10-dencies|Sao Paulo project is a specialised multi-media database in the context of urban development with an emphasis on collaborative work, planning and information retrieval. The system has first been used as a tool for urban planners, architects, artists and philosophers from Sao Paulo for collecting, organising and maintaining multi-media data about specific urban topics in the city. Every participant (an editor in the projects terminology) assigns a keyword or short phrase to each piece of data he wants to store in the database and arranges this keyword on a two-dimensional map in a way he thinks it relates to other keywords on the map and in the database.

The editors are able to view other editor‘s keywords and related content, drag them on their own keyword map and organise them spatially in relation to their own keywords. A centralised database system is used to store the participants data, keywords and keyword maps. A second component, the self-organisation, takes these maps and uses them to create a rule set for a self-organisation algorithm. The self-organisation uses these rules to iteratively calculate the motion of each keyword on a two dimensional area and stores the motion trail on the database. A second component, the force-field server, takes the actual position of each keyword, assigns a local force field pattern to each keyword and produces a two-dimensional global force field by accumulating the local force fields. This global force field is made available for retrieval over network to the visual client.

The visual client is the user’s front-end for information browsing. It retrieves a snapshot of the current global force field and guides the user by a dynamic flow visualisation towards clusters of information. Once the user has zoomed in far enough he is able to select the visible keywords to bring up a standard web browser which displays the related information along with the history map, a simple visualisation of the motion trail of that keyword over time. By clicking on certain time points on that

temporal trail, the user recalls former keyword constellations in relation to the one he has chosen.

Several technical improvements have been realised since the first inception of the work. In particular, the visual client component has obtained the most changes. Besides the implementation of a dynamic cloud visualisation of the force fields, an interactive, real-time synthesis sound system, based on a physical modelling approach, has been added to the client. Figure 1 is a sketch of the system components. The components inside the box are usually running on one machine for best performance. The arrows show the flow of data between components

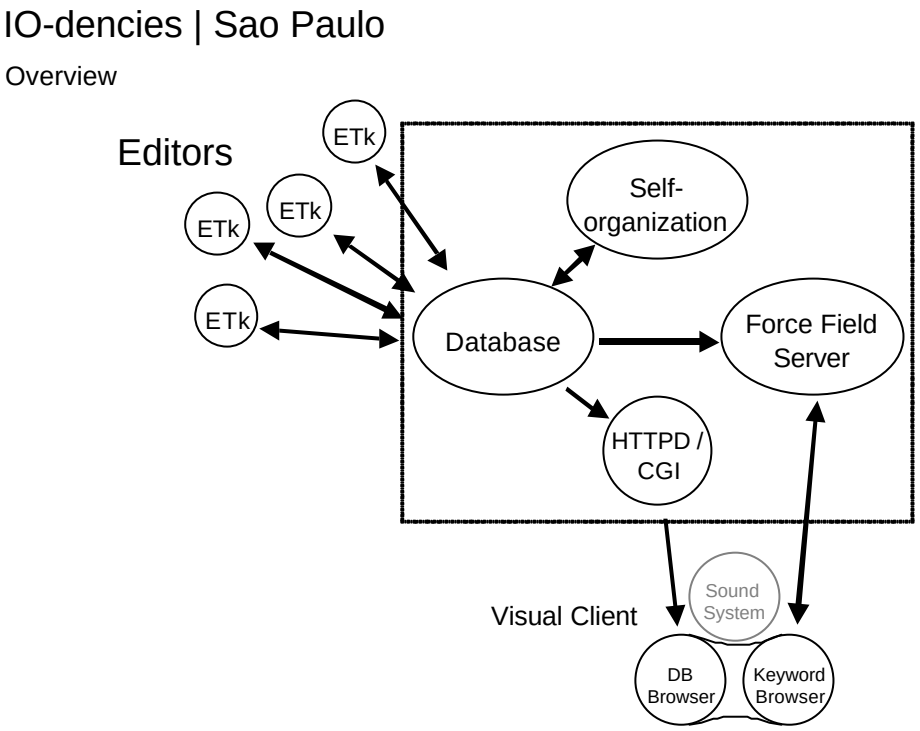


Figure 1: A sketch of the system components.

Furthermore, due to insoluble problems using the *POSTGRES* database system (http://postgresql.nextpath.com) , the decision was made to switch over to the *mySQL* (http://mysql.he.net) database system.

Editor's Toolkit ETk

The ETk is the editor's main front-end to the database and the keyword maps. It is realised as a JAVA application which utilises the JAVA/Swing API for building the graphical user interface and the MM JDBC driver for interfacing with the mySQL database. The program enables the user to upload her documents, assign keywords and annotations to them and drag keywords from a list onto her keyword map. For faster access and viewing of documents in the database, a caching mechanism is integrated which uses the local disk to cache all documents which have been retrieved

over network. The currently supported content types for documents are text (ASCII), image (JPEG, GIF), audio (AIFF, WAV) and video (Quicktime, AVI). The mime-type is determined automatically and stored in the database along with the document.

Figure 2 shows a screenshot of the main screen of ETk along with a pop-up window of the document viewer/keyword editor. The biggest area is occupied by the keyword map. The column to the right of it houses a list of all keywords which are currently in the database. Keywords are shown in blue if they were created by the editor himself and in red if they have been created by a different editor. A keyword is highlighted in grey if it is already on the keyword map. Located underneath the map is the document browser which shows all documents which have been put into the database by the editor. Each document is represented by it's name and an icon, showing it's content type. Right of this is a text window which displays the public annotation to the currently selected keyword. At the bottom of the window is a status line consisting of a one-line message field and a progress bar that usually shows the state of transfers to and from the database server.

Figure 2: A screenshot of the final version of the Editor's Toolkit, ETk.

Self-Organisation

One addition to the self-organisation component is a JAVA/Swing-based visualisation of the current spatial state of the keywords for control and demonstration purposes.

The set of parameters for the algorithm can be edited in a dialog-box and tested immediately. Figure 3 is a snapshot of the status of the self-organisation. The process can be started with a graphical user interface as shown or as a background task.

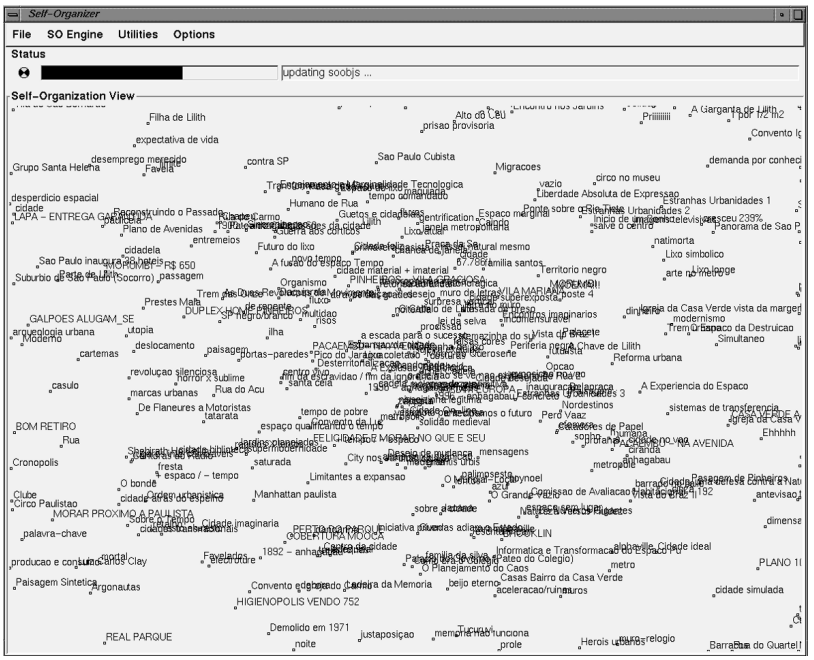


Figure 3: A snapshot of the status of the self-organisation.

Visual Client

The *Visual Client* component consists of three sub-components: the keyword browser, the database browser and a real-time sound system. One version of the visual client, which is downloadable for the public, runs on a single, standard Windows (9x, NT) platform and does not include the sound system. The second version has been developed for exhibition purposes, includes all three sub-components and needs three Linux PCs to run.

Keyword Browser

The main change to the keyword browser has been done to the visual appearance. So far, only points and lines where available to visualise the particles as they move and accelerate over the force field. In an attempt to make the intended visualisation of information clustering more obvious, a cloud-like visual style was implemented. Every particle flowing across the force field is rendered by a simple square polygon with a texture of a disc-like cloud attached to it, in which the transparency value goes gradually from zero to 100% from the centre of the disc to it's circular border. The transparency of the texture can be scaled by an additional factor to allow for a very high overall transparency. By using blending, the intensity of a cloud structure increases wherever the density of particles is very high. The result is that at areas with a total

force of approximately zero, the velocity of the particles is low but their density is high, hence the particles are blended to black clusters. At areas with high acceleration, the particles gain more velocity and hence their density is low, resulting in a very light rendering of the fast moving particles. Figure 4 is a screen-shot illustrates the start-up situation. The user is about to zoom-in on the position pointed to by the two triangular wireframe cursors. The black areas are areas of low speed and higher particle density.

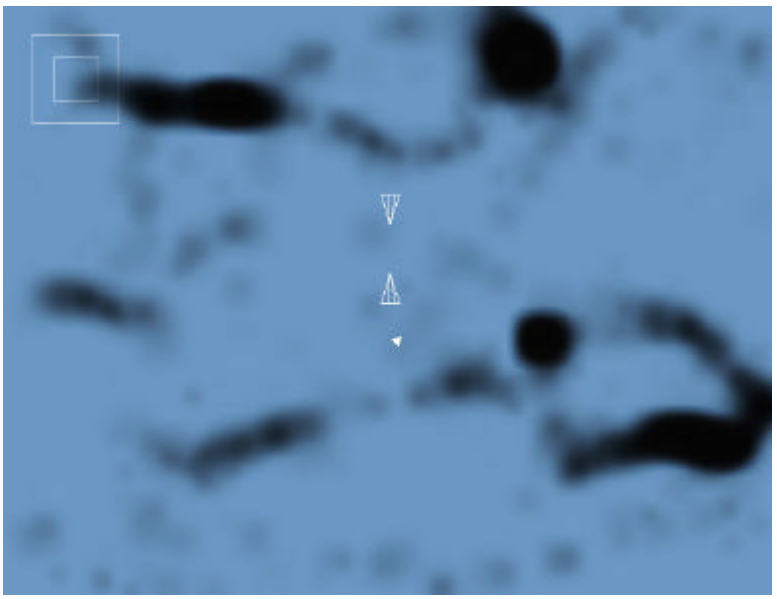


Figure 4: This screen-shot illustrates the start-up situation.

Figure 5's screen-shot shows the situation after the user has zoomed-in two levels. The keywords become visible. The current location and size of the visible area in relation to the total field is shown in the upper-left corner. In Figure 6, the user has reached the zoom level at which the individual keywords become readable. In Figure 7's screen-shot the user is about to select a keyword which would then open up the database browser. The keyword she has chosen is high-lighted and rendered at a larger size at the top of the screen. The upper-left corner now also shows a second and a third user exploring a region in the immediate proximity.

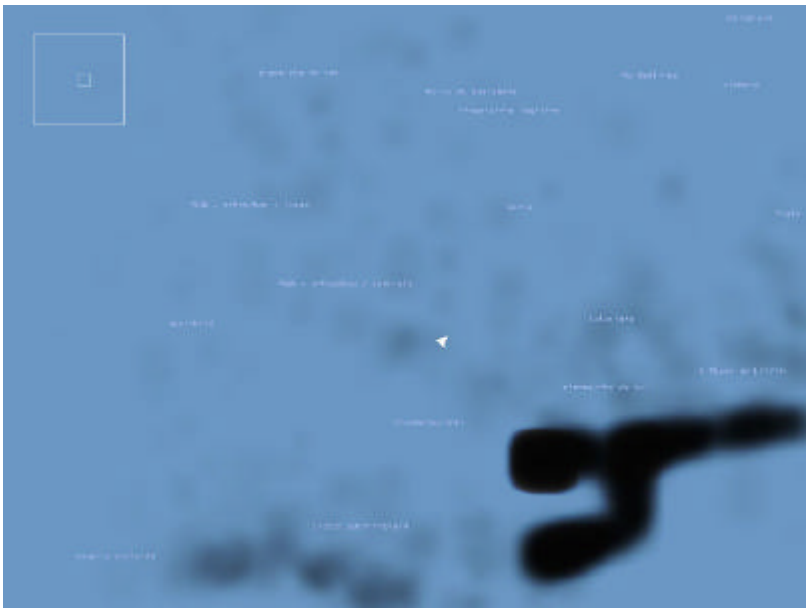


Figure 5: This screen-shot shows the situation after the user has zoomed-in two levels

Figure 6: The user reached the zoom level at which the individual keywords become readable

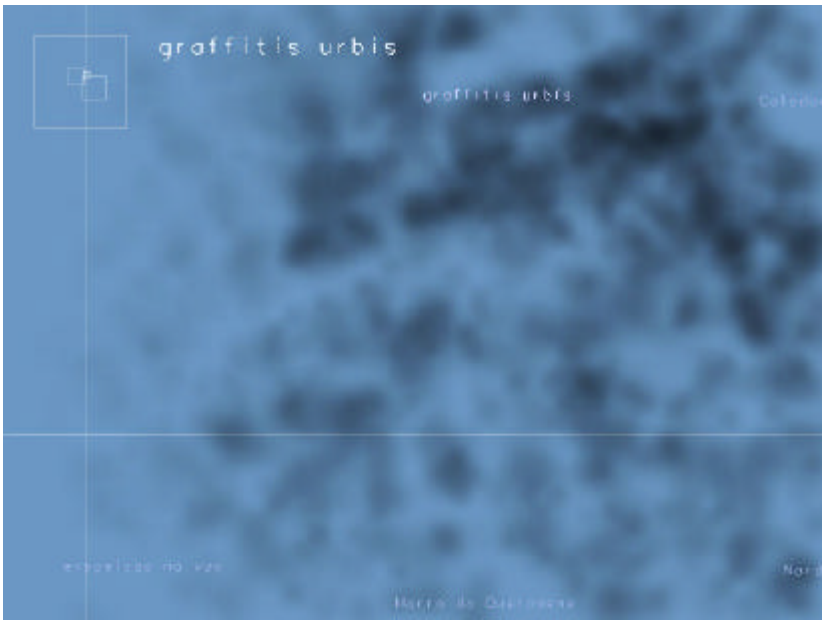


Figure 7: about to select a keyword which would then open up the database browser.

Database Browser

The decision to use the Netscape web browser as the data viewing component to the visual client, was based on the fact that the Voodoo2 card, used for the keyword browser, does not allow the use of standard graphical user interface elements for additional information display and interaction (which would have allowed for a single screen setup). With the Voodoo2 card active it is also not possible to use the standard X11 desktop for interaction in parallel. As a consequence of this, the Netscape browser has to be started remotely on a second Linux computer. Whenever the user clicks on a keyword in the keyword browser, the visual client sends a parameterised URL (depending on which keyword has been selected) to the remote Netscape window. The URL is basically the path to a PERL CGI script (residing on the database server), containing the object ID of the keyword in the database. The CGI script retrieves the data related to the specified keyword from the database, creates a frame-based HTML page displaying the name of the keyword, the name of the editor, the document (text, image, sound or video) and the history-map of the keyword. In Figure 8, we can see a standard Netscape browser showing the keyword (bilingual), the name of the editor, the related document (an image) and the historymap. The historymap shows the motion path of the keyword as well as other keywords in the immediate neighbourhood.

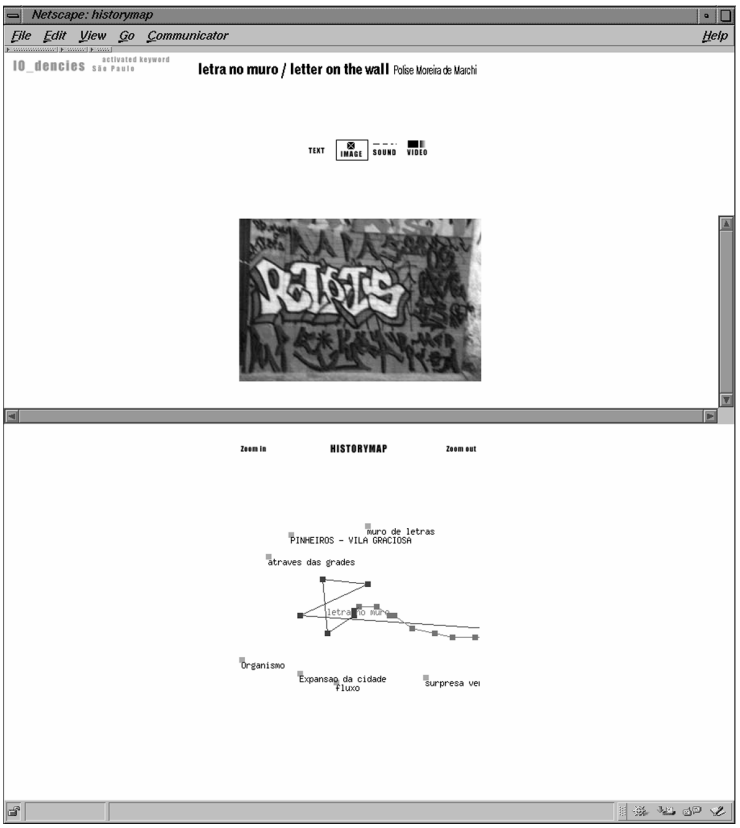


Figure 8: A standard Netscape browser showing the keyword, editor's name, the related document and the Historymap.

Sound System

The sound system is based on a real-time synthesis algorithm utilising a physical modelling approach by simulating the behaviour of a complex spring-mass system. Several mass-points are interconnected by springs and damper elements. If one or more of the mass-points are stimulated by applying a momentum, the whole system starts to oscillate. To actually produce an audio signal, the amplitudes of one or more mass-points are fed into the sound card output. With today's available processing power of a standard PC, a system with approximately 30 to 40 mass-points can be simulated with an output frequency in the audible range.

For performance reasons the physical laws are modified, so that the mass-points are limited to oscillate in only one direction and the elongation of the springs is measured by calculating the scalar difference between the positions of the two mass-points.

Arbitrary mass-spring topologies are possible by defining all masses, spring and damping constants and a list of interconnections between the masses in a simple text file, which is read by the system during start-up. Through a shared-memory interface, the sound system is able to access the underlying force field vectors of the currently visible area, viewed by the keyword browser. The x-y components of a subset of the

available force vectors are used to drive the parameters of the springs, dampers and masses of the sound system. By trimming and adjusting the values through scaling and offsetting, the result achieved is a slight but clearly audible change of the sound, whenever the user zooms or moves the keyword window, i.e. changes the force field. Additionally, depending on the cursor position over the force field, the whole system is stimulated with repeated impulses at different speeds and strengths, directly related to the magnitude of the force underneath the cursor. The graphical user interface to the physical modelling sound synthesis is shown in Figure 9. The topology of the masses and springs are defined in a text file, but all parameters can be edited with the shown tool.

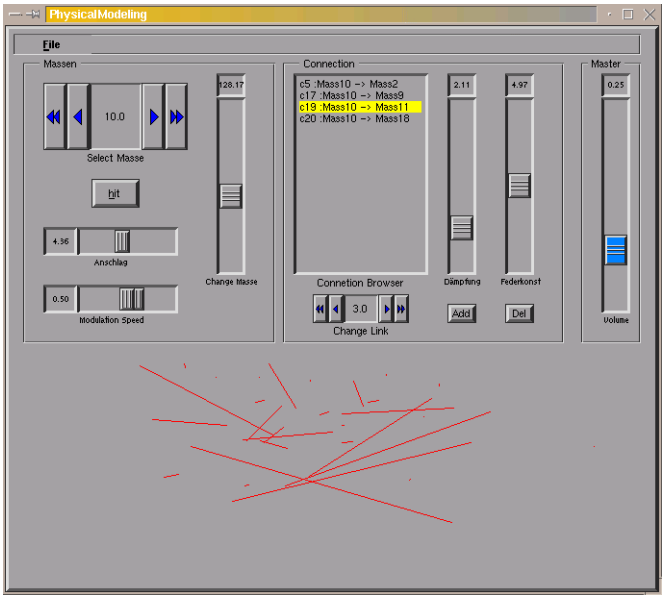


Figure 9: The graphical user interface to the physical modelling sound synthesis.

Acknowledgements

Acknowledgments go to Knowbotic Research, the authors of the IO-dencies project series, to Andreas Schiffler, who developed and implemented the force-field server and the visual client/keyword browser, to Daniel Berger who fine-tuned many of the visual parts of the keyword browser and to Andreas Weymer, who developed the Perl scripts to retrieve the database content with a standard web browser. Credits for the sound system go to Alexander Tuchaçek of Knowbotic Research.

Force Feedback Interface

Armin Steinke, ZKM

There exist so-called touch screens, where the computer screen is also an input device for a data processing unit. In these cases the user can trigger specific commands by touching specific surface areas on the screen, and thus can communicate with the data processing unit. Furthermore, there are also movable data input devices such as, for instance, computer mice, joysticks, etc. With these data input devices a cursor is dragged across the computer screen and, whenever the cursor is moved over a specific object field on the screen, pressing a button can transmit a specific command; what specific command is sent depends on what command is related to the corresponding screen position that has just been 'clicked on'. Data input devices up to now have been such that the user has complete freedom over the commands, and can position the cursor wherever he wants.

The Force Feedback Interface is a data input device for a computer that is similar to a digital tablet, by which a manual control device (mouse) can be moved across a demarcated surface area. The position of the manual control device (Force Feedback Interface) is transmitted in an absolute mode to the data processing unit (computer), so that there is a direct connection between the position of the manual control device and the current position of the computer screen (cursor). The Force Feedback Interface is spring-coupled - for example, magnetically - to a cross slide mechanism that detects the position of the manual control device by means of sensors; when the manual control device is moved the cross slide mechanism, which is driven by a motor, follows the Force Feedback Interface. The spring-coupling between the manual control device and the cross slide is designed so that when the positions coincide, no force (spring tension) is exerted upon the manual control device; but with an increasing discrepancy between the positions of the manual control device and the cross slide an increasing force is also exerted upon the manual control device. The tracking of the cross slide, which is guided by the sensors, is then combined with computer generated commands that depend on spatial and temporal coordinates; these commands cause the position of the manual control device and the cross slide to deviate from each other in a way corresponding to the spatial and temporal dependency relation, and in this way a directional force is transmitted to the manual control device.

The effect upon the user is that the manual control device on the computer screen's surface area is either drawn to specific quadrants or in specific directions, or becomes difficult to move to specific quadrants or in specific directions (command fields).

The surface area across which the manual control device is moved can be, for instance, a projected surface area, upon which the current computer screen's contents are displayed. Here there is an even stronger connection between the position of the computer screen and the effective force.

If this projected surface area is displayed as a background projection, the cross slide should be designed so that the projection is obstructed as little as possible. This can be done, for instance, by using thin steel wire or metal bands of the cross slide

mechanism, which are connected to a mechanism and - for instance - can also transmit energy and signals to the cross slide.

Thus in this case the user moves a manual control device (Force Feedback Interface) directly over a projected surface area; by moving the manual control device in the projection the user also moves cross-hairs (the shadow of the steel wire) and via the manual control device experiences a force that is spatially and temporally dependent.

Precautionary measures have been designed that will limit the translated maximal energy and insure that the transmission of force only takes place when the manual control device is being securely held. If the manual control device is released the force transmission immediately stops, in order to prevent the user from losing control of the manual control device. We present an example of one design possibility for the force feedback interface by means of the drawings below.

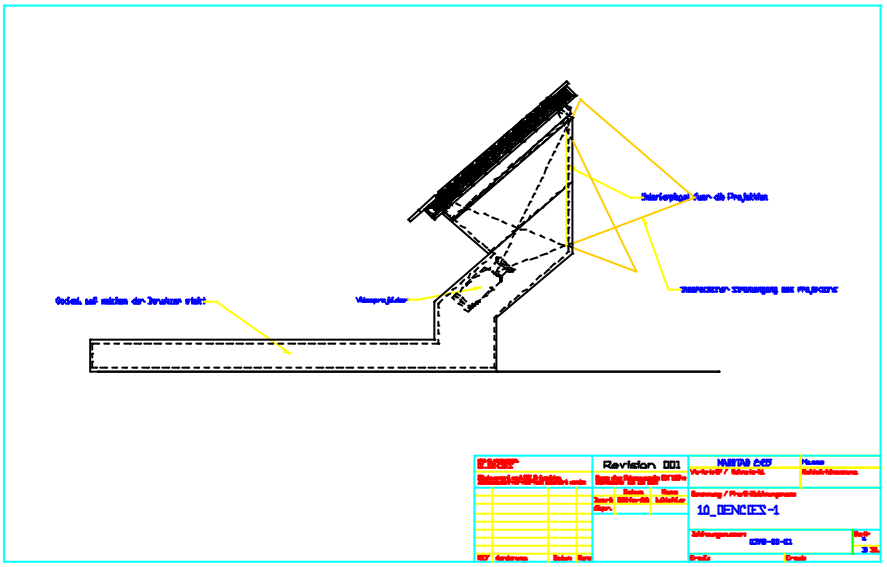


Figure 10: a cross-sectional view of the Force Feedback Interface station

Figure 10 shows in cross-sectional view a Force Feedback Interface user station with a base [1], upon which the user stands and an viewer [2] for reproducing images that are created by a video projection device. The video projection device is mounted in a relatively parallel inclination to the viewer [2] and the projection beam emitted by the video projection device is curved for the projection by means of a diverting mirror, so that the images created by the video projection device are reproduced on the viewer. Furthermore, provision is made for an additional display that can serve, for example, as an internet browser. Moreover, a cross slide wire mechanism is laid over the projection screen, which in Figure 10 is shown in its deployed position.

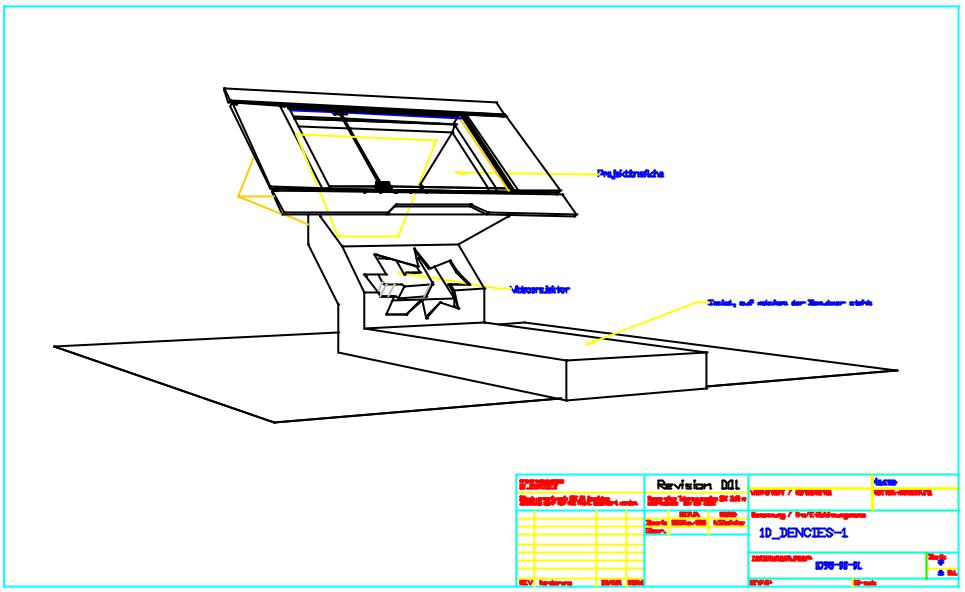


Figure 11: a view from above of the Force Feedback Interface user station

Figure 11 shows a view from above of the Force Feedback Interface user station represented in Figure 10.

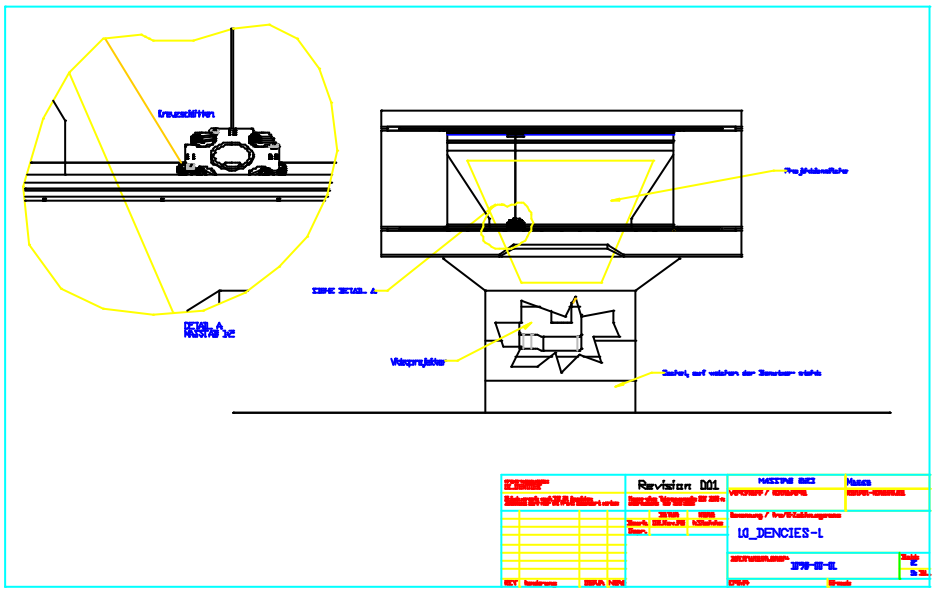


Figure 12: an enlarged drawing with detailed information concerning the Force Feedback Interface user station

Figure 12 depicts in an enlargement details of the cross slide mechanism, showing that a wire system is mounted on a cross slide, so that the cross slide can be moved in any direction along two dimensions. Every wire is connected to the opposite facing wire via deflection rollers. Furthermore, by means of a drive mechanism a force can be applied to the cross slide via the deflection rollers, whereby the given force is determined by the position of the cross slide on the projected surface area.

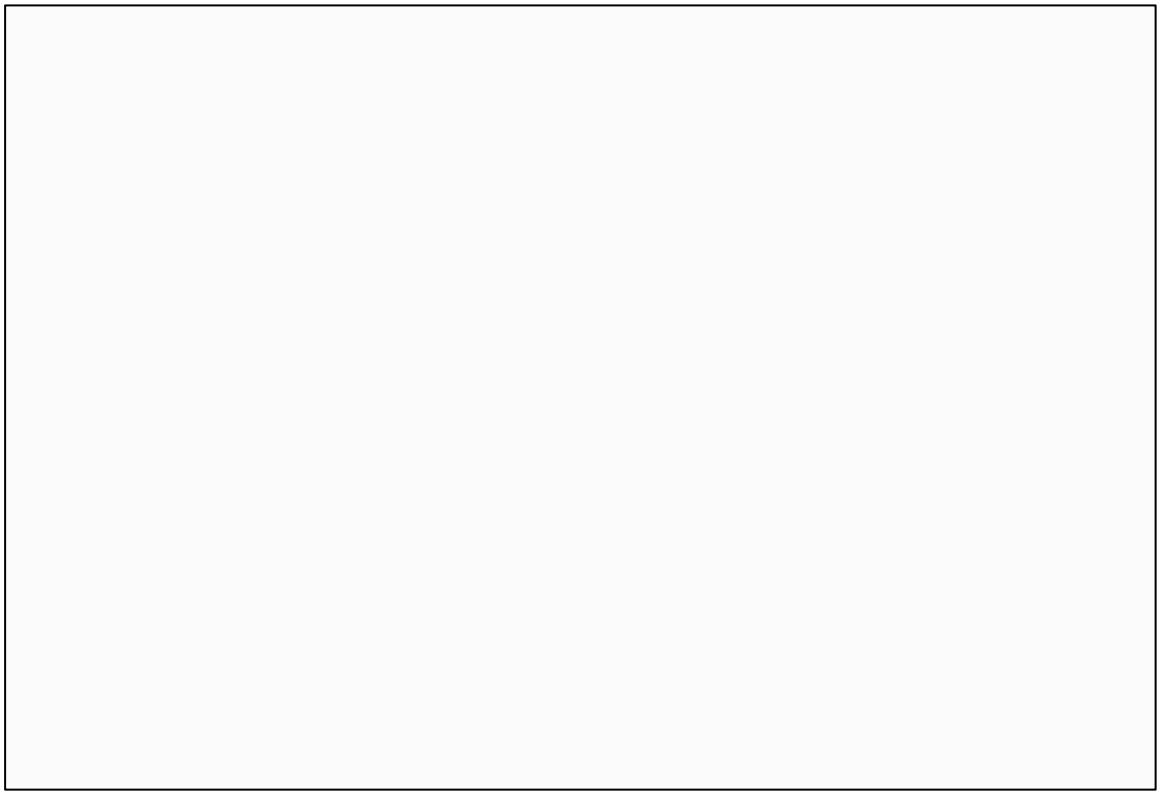


Figure 13: an example depicting the effect of the force upon the Force Feedback Interface when the computer screen is in a specific situation

In this situation the cross slide is pulled to the force field "MIN," so that the user, if he wishes to prevent the cross slide from moving there on its own, must produce a corresponding counteracting force. It is obvious that with the movement of the cross slide across the projected surface area corresponding to the current location, the user continually experiences a different magnitude and direction of force. For example, if reaching a specific location on the projected surface area is connected with a specific command, then the user might yield to the given force (if the corresponding command seems to him to be a good one) or might produce a counteracting force in order to keep the cross slide away from the location of a specific force area, thus preventing an undesirable command.

In Figure 13 several 'command locations', identified by the letters A, B, C, D and E, are depicted. These 'command locations' each indicate a command that initiates a specific auxiliary program sequence, if the user, for example, moves the manual control device onto command surface A and then, if required, presses another command key (for example, 'Enter'). If the user does this on another of the command locations, a different command might be given, so that the program initiates a specific sequence depending on the commands given by the user. It is obvious that those commands that seem particularly advantageous to the user (for example, in a game program) are

particularly difficult to reach. However, a command can also consist of clicking on a specific window, while this window overlaps with a force field or specific force lines.

Figure 13 depicts five force centres (that perhaps might not be even visible) on the projected surface area. In the situation depicted, the cross slide is located in a position indicated by the cross-hairs. In this position and situation the cross slide is pulled to the nearest force field, so that the user, if he wants to prevent the cross slide from moving there on its own, must produce a counteracting force. It is obvious that with the movement of the cross slide across the projected surface area corresponding to the current location the user always experiences a different force. For example, if reaching a specific location on the projected surface area is connected with a specific command, then the user might yield to the given force (if the corresponding command seems to him to be a good one) or might produce a counteracting force in order to keep the cross slide away from the area of a specific force surface area, thus preventing an undesirable command.

Figure 13 presents only one situation at a specific point in time. In regards to time the situation itself is continually subject to changes by a corresponding program so that there is no static situation, but rather the momentary current force is subject to constant changes and the user is thus challenged to respond to these changes. A temporal change means that not only the location of the force fields ‘MIN’, ‘MAX’ changes, but that their respective strengths also change. The spatial position as well as the strength of a force field depends on a program that is executed by the data processing unit and that connects specific commands with the spatial position of force fields.

Figure 14: a block diagram of a Force Feedback Interface

Figure 14 shows the principle construction of the Force Feedback Interface in block diagram format. Here the manual control device is arranged with the cross slide mechanism on a projected surface area; the manual control device can be moved across the projected surface area. The projected surface area is enclosed by a frame. For the cross slide mechanism y-axis drive and an x-axis drive will provide the respective force in the x and y directions. Each axial drive is coupled to a corresponding motor. Furthermore, for the x direction and the y direction an x-axis and a y-axis measuring system will be mounted on the side of the projected surface area facing the axial drive. This measuring system, like the drives, is coupled to the cross slide mechanism, which in turn is connected to force recorders, by means of which the forces acting on the manual control device in the x and y directions are recorded. The data collected by the measuring system concerning the force acting on the manual control device and the cross slide mechanism are processed in a position-force computer. This computer can be part of the data processing unit. This position-force computer controls on the one hand the video projector used to show the images on the projected surface area, while on the other hand it provides data about the current forces acting upon the manual control device in the x and y directions, which are given to a device that instantaneously adds numerical values. This computer, moreover, processes the data of the force recorders and performs an addition following the formulae of Y + B for the Y force component and a summation X + A for the X force component. The resulting data from the instantaneous adding device are processed in a regulator, whose results are given to the respective X and Y motors by means of an amplifier.

Overview of the achievements:

1. As a Force Feedback Interface of a fully developed data processing unit, which is connected to a force-generating device and which receives a directional force depending on its position.
2. As a Force Feedback Interface of a data processing unit, which receives a directional force depending on the contents of a program.
3. Data input device according to one of the previous claims, characterised by the fact that the magnitude of the force depends on a positional deviation that is determined by the position of the data processing unit and an ideal position.
4. Data input device according to one of the previous claims, characterised by the fact that the data input device is a part of a manual control device that is movable across a demarcated surface area.
5. Data input device according to one of the previous claims, characterised by the fact that data input device is coupled to a data processing unit and that this data processing unit determines several time and/or location dependent force fields on the surface area, across which the data input device is movable.
6. Data input device according to one of the previous claims, characterised by the fact that the surface area, across which the data input device may be moved, is a display for reproducing images that are created by the data processing unit.

7. Data input device according to one of the previous claims, characterised by the fact that the data input device is coupled to a cross slide wire mechanism, by means of which the respective force is applied to the data input device in a way that depends on position and program.
8. User interface with a data input device according to one of the previous claims, characterised by the fact that the surface area, across which the data input device can be moved, and the indicator surface area at least partially overlap.

The Distributed Legible City

Andreas Schiffler (ZKM) & Steve Pettifer (The University of Manchester)

In early April 1998 a design proposal was made to implement a multi-user version of J. Shaw's art installation 'The Legible City'. A number of new facilities were posited, with the primary aim of generating an environment in which there was enough 'of interest' to encourage social engagement between the environment's inhabitants. These features included:

- graphics similar to the original installation
- to use consumer technology such that the installation could feasibly be built for 'home use'
- to add avatars in form of animated bikes to the scene
- to support several players in one city (multi-user capability)
- to include possibility of scene-level interaction (for example the joining of two players to form a tandem)
- to have player interaction through a voice channel that is proximity controlled (i.e. 'you hear X when X is close').

The implementation design was based around a standard client-server structure in which the server maintains a consistent state for the virtual city, provide network services and controls the audio system, where the client provides the visual output and manages the user interface. Audio interaction on the client side will make use of a separate phone or headset. The hardware design was build around a high end PC with consumer 3D hardware and free software components. Connectivity was (originally) to be achieved with Digital-Simultaneous-Voice-Data (DSVD) modems and standard analogue phone lines.

Design changes

Two changes were included in the final work plan. One was to use a modified exercise bike as the interface just as in the original installation. The bike as an interface adds physical interaction and involvement and is a 'trademark' of the original installation. The second was to include a new level of interaction with the world via voice recognition. One would let people populate certain streets in the virtual world with words that they

speak while moving. This required the addition of a second PC that would handle the interface and recognition tasks on the client side.

The Implementation

Hardware and Software Platform

The final hardware platform has these key elements:

- Dual Pentium II PC with Dual Voodoo2 graphics adapters for client
- 21' monitor as display for each client
- Pentium PC with Force Feedback Joystick and AD/DA card for client
- Pentium PC with multi-port serial card for server
- Audio and network capabilities on all computers
- Modems on all computers
- Custom build analogue audio mixer with computer control
- Custom build exercise bikes with analogue outputs for speed and direction
- Quality head-sets and audio hardware on each client

The software platform makes use of these components:

- Linux distribution with SMP enabled
- Standard Windows 95
- Glide/Linux drivers and Mesa to run OpenGL on Voodoo2 cards
- Manchester University's MAVERIK/Deva graphics system
- SpeakFreely audio software
- IBM ViaVoice recognition software
- Many custom programs and scripts

The Graphics Subsystem

The graphics in the Distributed Legible City are generated using MAVERIK, rendered via the Free Mesa 'OpenGL workalike' libraries. Interactive frame rates were achieved using an experimental driver for the 3DFx Voodoo2 Cards and the GNU/Linux operating system.

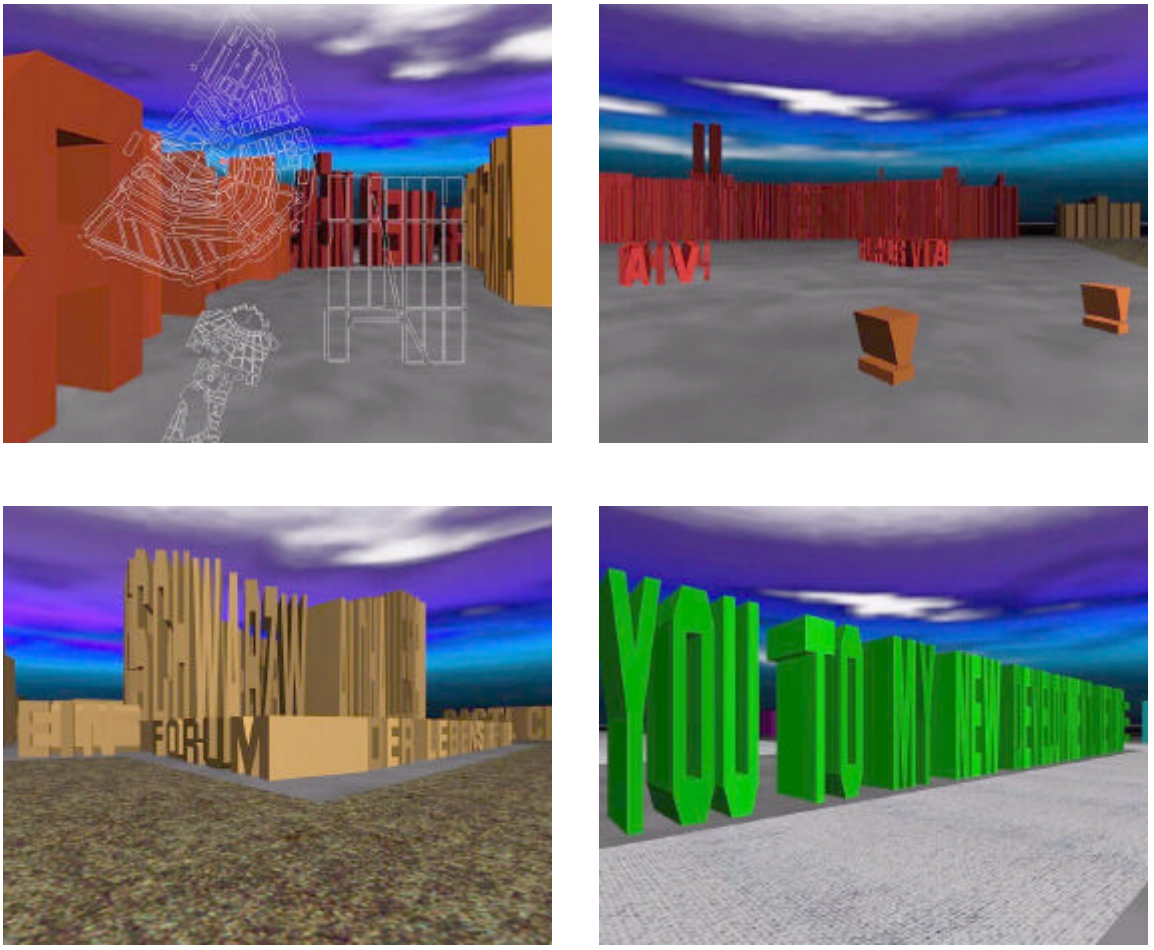


Figure 15: Scenes from the DLC

The MAVERIK (Cook, 1998) software is being used for scene management and as general graphics interface for the distribution system Deva (Pettifer, 1999). In the process, the Distributed Legible City was first implemented as a standalone version within the MAVERIK system. We could build upon an existing demo program that loaded the original Legible City databases and allowed navigation. The final program was extended to include all three cities that are selectable in the original installation in a special layout and with streets connecting them. Other graphical features and colour settings mirroring the original were included. A ground floor and a sky were introduced. The possibility of displaying a 2D map as an overlay for navigational purposes was added. This program was then integrated into the Deva system, so it could be loaded as a Deva object on request. In a similar fashion the biker avatar was added. An initial model of a biker was created and stored in a format readable by

MAVERIK. The biker consisted of 11 different objects that were placed dynamically by a program for each frame being displayed.

Figure 16: The cyclist avatar

This allowed the simulation realistic foot, pedal and arm movements depending on current speed and direction. A standalone biker was also converted into a Deva object. The distributed version of the Legible City was then made viewable by creating a city and several bike objects within the Deva system. Each client would in turn run a viewer program that attaches to this scene consisting of the cities and the three bikes. The viewer then attach its own camera-view and control to one of the bikes in the scene. This is done through external Deva commands. Simultaneous changes to all positions while the viewers are running are then propagated through the Deva server to all attached viewers. The dynamic lettering of the streets via the voice recognition would have been implemented in a similar fashion, but was left out of the exhibition version due to time constraints during the development period.

The Interface System

The user interface in the exhibition version was solely the exercise bike. It provides two degrees of freedom (speed, direction) and a button to toggle the map display. During the development period the joystick was used as a substitute input device. To drive the joystick's force feedback capabilities Windows95 is required. Force feedback with the joystick was being researched at the Manchester University on different applications, but was never implemented for the Distributed Legible City since the bike was the input device of choice. The AD/DA card that was used by the installation could only be used from a Windows platform. Because of these software constraints, a separate interface PC is employed. Two interface-server programs run on the interface PC. They are queried by a Unix client program running on the graphics PC through a local network connection in regular intervals (100Hz) and made available to the MAVERIK/Deva system through a shared memory segment. The interface PC was also running the voice recognition software, which was not used in the exhibition version as described earlier.

The joystick is connected to the sound card of the interface PC and gives direct readings of X and Y positions of the handle as well as the status of all buttons.

The exercise bike was modified to provide three voltages that could be read by the AD/DA card. The first voltage corresponds to the speed of the pedalling and was produced by a small generator that is directly attached to the fly-wheel in the bike. The second voltage corresponds to the steering direction of the handle-bar. It is generated by a linear potentiometer in the rotatable handle bar through an applied external voltage. The third voltage give the button status as two voltage extremes in a similar fashion.

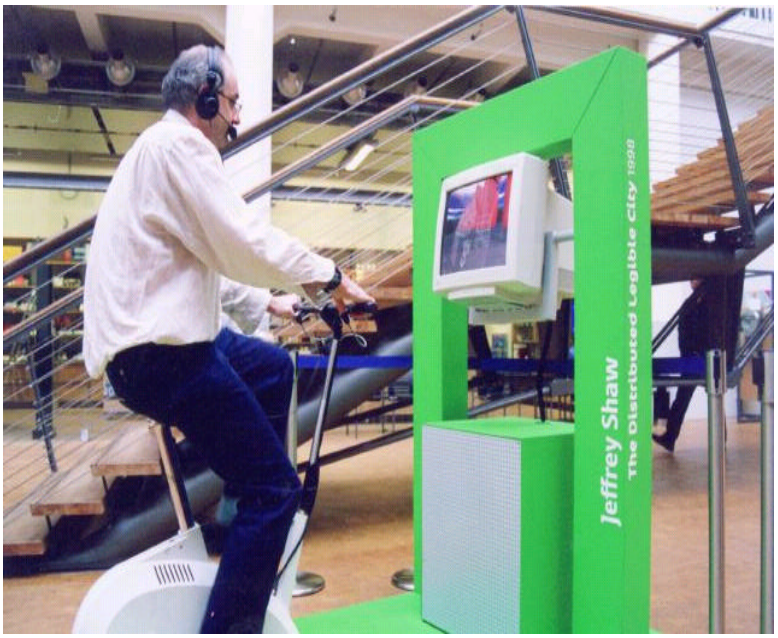


Figure 17: The DLC exercise bike and station

All software programs on the interface PC are designed to be controlled through a network connection. They will wait for a TCP/IP connection on a specific port number. Once a client is connected, they are will return the information as ASCII text on request. This allowed easy testing using the telnet program.

The Audio Subsystem

The initial specification called for a DSVD modem to transport phone-grade audio between the server and the clients while a standard PPP connection is being maintained at the same time. This mode of operation is mainly used by computer game users to be able to play and talk to an opponent at the same time. During the design process two sets of modems were tested and rejected because of the low audio quality attainable if they were working at all (the first set didn't). The final implementation made use of network audio through the sound card of the PC and a set of Unix programs for recording and playback. This would assume that there exists a sufficiently fast data connection between client and server. The headsets are connected through a multichannel-audio mixer to allow easy volume and level control. The microphone output is split and connected to the interface PC (for the voice recognition) and the

graphics PC (for audio interface). For each client-server audio connection a set of four programs from the 'speak-freely' freeware program suite: one to send the microphone input to the server, one to output this to the mixer, one the send the mixer output to the client, and one to output this to the headset. The mixer is a custom hardware located at the server site. It implements a mixing grid for three inputs and three outputs that can be controlled through the parallel port of the server PC. Since it was not possible to drive more than one sound card on the server PC reliably, two more PCs in proximity to the server are being used for audio recording and playback. During operation, a special routine within the Deva server would record the positions of the three bike objects, work out the distance between them and send volume control information to the mixing grid in one second intervals. The distance at which the volume was reduced to zero was approximately the distance used for the culling (i.e. the distance at which graphical objects disappear in the fog and are not drawn anymore).

The Networking

Since dedicated DSVD modem connections to the server were not used, the system was being connected through regular ethernet. Each of the three clients and the server were given a separate IP address and were connected to the network. The two audio computers were already part of the ZKM network and were simply tied into the installation infrastructure. The local ethernet connection between the interface PC and the graphics PC was initially done through a mini-hub and local IP numbers from the 192.168.x.x subnet. This was later changed to a separate, independent local network connected with a single short cable. The graphics PC was equipped with a second network card for this to work. To be able to place one of the installations at a remote location without requiring a connection to the Internet, an ISDN dialup into the server was installed. Through a ISDN-Terminal adapter and the PPP protocol a remote client could connect to the server for DEVA and audio data exchanges. The connection uses an Euro-ISDN connection and two channels through bundling to achieve a theoretical throughput of 128kbps (limited by the serial port of the computer to 115kbps).

The Presentations at the ZKM and IST

The visible part of the installation (client computers with bikes and monitor) was integrated into the common design of the *Surrogate 1* Exhibition at the Centre for Art and Media, Karlsruhe (ZKM). The setup consisted of a three section ground plate that was hollow so that cables could run underneath it. Attached to the plate was a rectangular three piece arch of 2m height. Underneath the arches top, a metal mount held the monitor at viewing height and angle between the arch sides. Below the monitor was a box containing the computers and other equipment. The bike was placed in front of the monitor and was loosely attached on the ground plate. All wires to the bike and computer box are hidden and detachable. On the bike is a connector for the headset that was placed on the handlebar during use. The three systems were placed at different locations within the ZKM during the exhibition time (1 Nov. - 6 Dec. 1998) and were connected via ethernet. Two systems were within visible range of each other

and placed so that users can see each other; one in the foyer of the ZKM entrance and another on the first floor above the foyer. A third system was at a more remote location on the second floor. In most cases there is no staff present at the installation and people are free to experiment; only once during a special presentation of »Surrogate« installations to invited guests, technical staff was present to answer specific questions.

During the IST '98 conference in Vienna one of the system was disassembled and setup again in the 'future technologies and interfaces' exhibition space of the IST. This time the ISDN connection was used to link the bikes in Karlsruhe and Vienna. The main differences during this exhibition were that the audience consisted of mostly professionals, the fact that the bike at IST was staffed most of the time and that during two days of the conference only two bikes were used in the virtual world, since the ZKM was closed to the public during these days.

During the exhibition we encountered two major problems with the physical parts of the installation: The pedals of the bikes broke off during the heavy public use and needed to be welded on permanently. The headset wires were severed frequently and replacement headsets had to be used.

Figure 18: The immersive DLC at Essen

Evaluation and Comparison

The installation achieved in most parts the capabilities outlined in the original proposal, although several features were changed, dropped or did not work as well as planned. The overall feel and look of the new version of the 'Legible City' was comparable to the original. This of course stems from the fact that the original database for the city layout and the fonts were used in a one-to-one fashion. Differences in the graphics were mostly due to hardware constraints and implementation shortcomings. Fogging did not work as well on the PC platform as it did on the original SGI program. This can be fixed easily by some program changes. The overall graphics frame rate was good except when the biker avatars came into view. This is likely due to the Linux/Voodoo driver implementation as well as the still very high polygon count of the model. The steering with the bike felt initially different as compared to the original - although this is mostly due to the differences in bike-interfaces physical and electrical design, software updates during the exhibitions improved this. The quality of the steering interface did deteriorate noticeably on the system that was used the most in the foyer of the ZKM.

The joystick was dropped as an input device for the exhibition version. The University of Manchester made active use though of the joystick's force-feedback capabilities in other applications to aid in navigation. Several users expressed much interest in force-feedback capabilities of the bike - especially when considering the placement of such an installation into a fitness studio. The biker avatars worked well as a representation of the connected users. People spend much time in the world locating the other users and 'chasing' each others' avatar. The avatar dynamics were constraint to just the pedal movement in the exhibition version (no arm or head movement). This was the case because of difficulties with the Deva integration of the interfaces. Users did not express this as a lack of detail. The multiplayer capabilities implemented through the use of Deva worked reasonably well. Some of the problems encountered with the Deva system were difficulties in overall system configuration, the tuning of network distribution parameters for specific network bandwidths and the addition of custom control code. This lead to erratic frame rates from time to time during the IST exhibition (where a reduced bandwith connection was used). Scene level interaction was not implemented in the exhibition version. Especially a 'tandem' function that allows one user to take a ride alone with another use is likely a big shortcoming, since it was difficult for users to stay in proximity while moving around - a requirement for continuous audio connection. The user interaction through the audio system was poor. This was partially due to the overall quality of the connection. Until the software was updated, a discussion could only proceed in a very restricted and slow manner, since there was a long delay (approximately two seconds for each server-client connection) introduced by the network-audio programs. This delay could be reduced by software changes during the exhibition to a level that made conversation possible. Overall audio quality was also a problem as distortion and noise was introduced due to the fixed level input, the software programs and its compression mechanisms. The goals set for the audio system in the proposal was therefore not entirely met and warrant a re-implementation, should such a system be used again in this or another context. Overall the system was easy to use by visitors to the ZKM and at IST. The interactive capabilities were not used by visitors as much as anticipated. This is due to the technical shortcomings as well as a matter of raising user awareness of the systems capability through presentation and interface.

Possible new Developments

New developments with the Distributed Legible City should be guided to advance the interactive capabilities of the installation. Since audio connectivity is the prime interaction between users in this installation, the audio system needs to achieve a greater quality to be useful. Since the audio system interacts with the virtual environment by proximity control related help functionality such a »tandem function« could be added to aid the user in engaging in a conversation. The map - the user interface for navigating - should be more descriptive and aid the used more in finding other people in the virtual world. Colour coding the physical set-up and using a corresponding virtual coding of colours can help as well in finding people. Since the

user interactions take place within a text, it should be made much easier to read the text. This could be achieved by employing a head-mounted display with directional tracking that allows to bike forward while looking sideways. Other functionality, such as a still image link between the systems could help users' identity the possibility of an interaction with another user - especially when systems are completely separated physically.

Nuzzle Afar

(Maski Fujihata, Annika Blunck, Keith Vincent; ZKM)

Nuzzle Afar is not a work which delivers a completed world view to its visitors. The world of this work is provided as an interactive environment constructed and transformed in real time by the presence and activity of other users. In the exhibited version Fujihata presented in November 1998 (in the Surrogate show at the ZKM), visitors were given roles as navigators through a virtual world while simultaneously participating in the creation of that space. This proved quite a challenge even for those who were familiar the participatory nature typical of simply structured interactive works. Such works do little more than provide a set reaction in response to the action of the users. Their objective is simply to make the visitor aware of an interactive environment and as such these pieces do not get beyond the level of enjoyment one can have in seeing that a light comes at the flip of a switch. Exhibits functioning on this level are concerned solely with how to get visitors to flip the switch. In Nuzzle Afar however, it was necessary to get users to discover the operating modes of cyberspace, to discover others within it without whose presence it would be incomprehensible.

The Interpretive Process

The conventional method for appreciating a work of art consists of unraveling and interpreting meanings which are assumed to exist within the work in a condensed form. Interpreting meanings requires an understanding of the cultural codes which subtend them, just as reading a book rooted in another culture requires an assortment of reference books and dictionaries. Translation is thus dependent on the translator's ability to make judgments about the level at which those cultural codes are to be converted. In the process of interpretation the reader activates these cultural codes in his brain in such a way that the relations among the various elements represented in the work gradually come into focus. This is an interactive process, but one that occurs entirely inside the mind of the reader (and it might be one of which he or she is not aware). This interactive space/time is intellectually enjoyable for the reader. However, the challenge is how to seperate this time/space interaction from the individual brain operations of the recipient and merge it with the content. The aim would be to visualize perception and interpretation through interaction. It would not be a question of conversing with the work in order to read it and then moving on to the process of

interpretation. Rather, this would be a completely new form of art in which the site of interaction with the work would itself be the space of interpretation

Intellectual Activity and Experience.

In *Nuzzle Afar* interaction takes place on two levels. On the first level, the user navigates through cyberspace via an interface connected to a computer. Interaction here allows the user to come to know a kind of space and spatial continuity which differs from that in the real world, the *Nuzzle Afar World*. Most VR works stop here, simply setting up the relation between the user and the projected virtual world. In *Nuzzle Afar*, however, the objective is to enter into interactive relationships with other people through the network. While the first stage of interaction in front of the machine is designed to function as a lead-in to the next stage of interaction with others, on the second level the *Nuzzle Afar World* proves itself as a system whose ultimate goal is to create a space where people can discover new relationships with others.

The space of the story is open to each visitor from the beginning, and dramas can be expected to unfold from the relations between the self and others. The idea that we could render transparent the interface between the world and ourselves simply by setting up an immersive environment using an HMD (Head-Mounted Display) was certainly overly optimistic. But even in the real world communication with others involves a whole array of manners and customs and there is no reason to believe that anyone can spin his or her own tale simply by changing interfaces. Even more than interfaces there are any number of environment-creating inventions, such as the postcard, the telephone, and travel, which are crying out for reevaluation.

Users can weave their own stories in complete freedom. And yet precisely this freedom may give rise to a certain melancholy. Indeed, users of this work have to be extremely proactive vis a vis the *Nuzzle Afar World*. It cannot be enjoyed in the way one reads a novel or watches a movie. Users must pay a price for the freedom to tell their own stories. Compared to the effort put in to acquiring the knowledge to interpret narrative worlds by conventional readers, the proactivity of confronting a new world may not seem like much. But what holds it back is not inherent to the experience itself, but the result of an excessive cautiousness combined with the particular intellectual style which since the advent of modernity has caused us to privilege abstract knowledge divorced from the scene of action. In our future intellectual activities we must rid ourselves of this habit of distancing ourselves from reality. We need to remind ourselves once more that knowledge is acquired solely through lived experience. The difference here is like that between looking at a chart of insects and making an insect collection. In this sense, *Nuzzle Afar* provides the user with a space in which to collect insects and to enjoy observing those insects (or other people) afterwards. During the process of interacting, the user will become aware that observing someone else involves being observed oneself as well.

When it comes to actually creating a work, designing the presence of the self in space and its embodiment vis a vis that space are extremely vital elements. Of course

these functions can always be far removed from those actually existing in the real world.

The Modeling of Communication.

The Avatar Function.

Works like this one which allow multiple users to share the same cyberspace are referred to as "shared virtual environments" or "distributed virtual reality". The field was originally pioneered through text-based software known as multi-user dungeon or dimensions (MUD) or MUD object-oriented (MOO) systems. MUDs/MOOs are novelistic worlds constructed through interactive texts. In these highly abstract worlds it is possible to carry out experiments with relative ease that would be impossible in the real world. In actuality, however, the kind of communication they make possible is beset with the same kinds of confusion and ethical problems that plague the confessional novel. Users discover the pleasure of putting on different masks through the avatars which serve as their alter egos in the virtual world. While a novelist has to bear some responsibility for the avatars he or she creates, MUD players are almost never held accountable for the actions of their avatars. (Avatar is a Sanskrit term referring to spiritual beings who manifest themselves in the real world as the incarnations of deities.)

Currently there are many three-dimensional versions of MOOs (including Sony's *Cyber City* and *Ultima On-Line*). These are designed to allow people to encounter other users while interactively manipulating a three-dimensional space in real time. Users accessing this three-dimensional space through a network are able to communicate through avatars. Most of these programs use avatars with human shapes, though the conversations take place by typing on the keyboard. As soon as the communication begins to be text-based the user's character as mediated through the avatar reverts from the visual to a textual world. The addition of a visual level distributes the communication and action into two areas. The user’s concentration must either switch between both levels or neglect one in favour of the other. Images here do little more than provide the occasion for encounters in a relatively cumbersome fashion. Little real progress has been made in terms of communication style. Ultimately we are only reminded of the superiority of verbal communication.

The design of avatars is an extremely important element in making new communication possible in this kind of spaces. Most works thus far have used avatars modeled after actual human beings. The trend now is towards avatars with the physical appearance of human beings who walk around just like people. The main engineering research laboratories are developing technology which picks up bodily communications such as movements, gestures and facial expressions and expresses them directly in the virtual space. Rendering all of the information expressed by our bodies into virtual form may be a challenging task for engineering researchers, but ultimately this work has little meaning. Even if we were able to perfect this system, it would not represent any real advance since we would only end up bringing the limitations of our bodily

communication devices into the virtual world. Using computer technologies requires that we understand their limits and work to adapt our bodies to them. In order to reduce the work of adaptation to a minimum the interface mechanism should be as simple as possible. This simplicity should make it easier for the body to react freely. For this reason it is meaningless to develop sensors and interfaces to gather information in accordance with the complex structures of our bodies. Instead it is more important to design the interface structure as simply as possible while still allowing for the most meaningful manipulations.

There is no reason to bring the bodies with which we act in the real world into virtual space. Abstract representations like text should be sufficient to constitute a virtual world. However, the mode of expression should be functional rather than formal. The outward appearance of the avatar is not all that significant. It is only necessary to be able to distinguish one avatar from another. The design of the physical appearance of the avatar has hardly anything to do with the design of communication. Functionality is most important here. Spatial relationships must be visualized, distance from others must be made clear, and the mode of the self must be clearly marked. Once these basic conditions have been established we need to concentrate how the users will get their ideas across to others. There are many important functions to be designed before we concern ourselves with controlling the facial expression of avatars; Which communication functions should be expanded? What effects would this expansion bring about? Which bodily functions are unnecessary in cyberspace? In order to get beyond the limitations of conventional communication we have to elaborate those functions which can only be established within cyberspace.

By making it possible to experience changes in the distance between oneself and others in an intentionally composed space we should be able to show changes in relations among individuals in a new way. *Nuzzle Afar* is characterized by its spatial functions intentionally composed with this in mind. Distance here does not refer to the continuous distance of geometric three-dimensional space. Rather the emphasis is placed on the sudden changes in distance resulting from mutual interactions. These represent encounters and it is here that greetings are exchanged. Human relations begin and end with greetings. Connections among people begin with greetings. Once the link has been established direct interaction begins, and depending on the quality of that interaction differences arise in the distance among people. People come closer together just as they drift apart. The quantity and speed of information exchange also effects the way we distance ourselves from others. The space of *Nuzzle Afar* is designed in a way that any contact creates a different kind of space shared only by those coming together. Contact in cyberspace is not like physical contact among people in the real world. In *Nuzzle Afar* it is conducted among avatars as new spaces arise when one enters the avatar of another person. However, the meaning of this contact only becomes clear when one realizes that entering into someone else's avatar also means having that person enter one's own.

Relationality and Distance Through Hyperlinks.

In virtual spaces following the MUD model each avatar has its own room. The contents of each avatar are expressed by its room. In *Nuzzle Afar* each contact creates a link to a different dimension, something like a hyperlink. Entering into another's avatar is like clicking on a hyperlink and moving to another level in which the contents of that avatar (or more precisely, the worlds of the avatars) are brought to life. This is not a continuous world. In the relationships of spaces brought about by these hyperlinks it is possible to create an inescapably different dimension. Here we have the two-layered spatiality of hyperspace. Extremely interesting possibilities open up between the three-dimensional world made visible by computer rendering and the n-dimensional space (which one might simple call media space) arising from hyperlinks. What we need to do is understand and effectively exploit the multi-dimensionality of cyberspace, where its real strength and power lie. *Nuzzle Afar* is ultimately concerned with the future potential of this n-dimensional space.

The Intimate Sphere.

When users encounter each other in the virtual environment of *Nuzzle Afar* a different space emerges and the users enter into a new mode. (This is what Fujihata calls the "intimate sphere.")

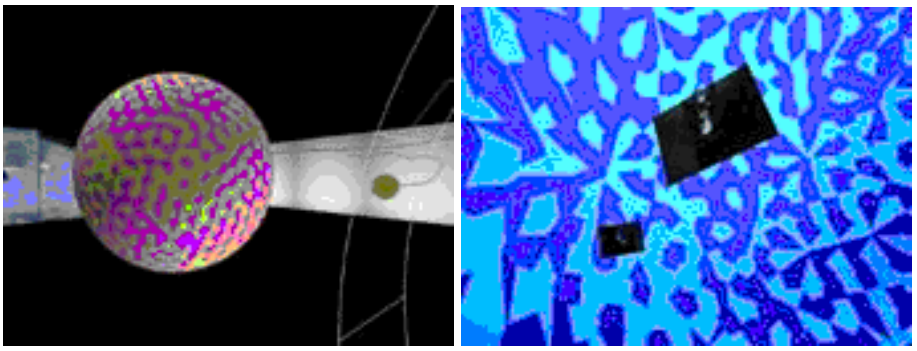


Figure 19: An “intimate sphere” seen from (a) the outside and (b) inside.

Here one has to coordinate one's actions with the other's. The system is designed so that the only possibility to return to the initial public space is by cooperating with the mutual partner. (Users can move back to the space before by causing their images to overlap.) The sense of distance in this space is different from that in the public spaces. It would be possible to make these mode-changes more diverse than they are in the present version of *Nuzzle Afar*, but in order to make it functional within an exhibition space which can accommodate an indiscriminate number of users and where the time frame for understanding how to handle the interface as well as the interaction itself is rather limited, Fujihata decided to keep that variety to a minimum. The result is that some people are left unsatisfied. It could be possible to change the mode of each world depending on the method of contact. But in practical terms, the only necessity in the

current phase is to transform the variations according to the differences in the threshold of interpretative ability among the users visiting the exhibit.

Space and Memory.

Opening up new worlds through contact with others can be compared to clicking with the cursor on an icon. In this artistic cyberspace one accesses the contents of a three-dimensional object by bumping into it. *Nuzzle Afar* in one sense makes the desktop environment of the computer or its operating system into a three-dimensional space. At the same time, if we consider urban space as a kind of media, the mode of abstracted human relations offered by this work might be considered as an example of the desktopization of urban media. There are possibilities to explore in either direction.

The avatars designed in *Nuzzle Afar* have another distinctinve characteristic--their ability to map memories on a temporal axis in spatial terms. Vis a vis the directionality of the temporal axis, computers have a special ability to remember the process of non-linear operations. (The "undo" function is a familiar example of this ability.) Applying this ability to the communication systems of the present work it is possible to record spatially the processes of movement and action of human beings in the virtual worlds. In fact the path taken by avatars is preserved by drawing a line in the 3D space. This means that even if another person's avatar itself is not visible (because the projected scene always represents the individual perspective of the user in the virtual space) it is possible to see which avatars have passed through the same space seconds before. A small sphere is attached to the end of each line and when the sphere is captured the computer automatically traces the path of the line bringing the user to the avatar's current position. By this feature memory has been inserted into the space; something which is not possible in the spaces we inhabit. It exists as a metaphor but is not something we are able to witness directly. For example, others are able to find out about or even meet the authors of published books or statements in the media by picking up their traces. But when these people no longer exist, memories scattered in space become the medium through which we can interact with them. These are traces without avatars.

There is no reason why a trace has to be a monotonous line. It would also be possible to express the amount of time the avatar spent in a certain space and what it did there by varying the thickness or color of the line. Currently, when two avatars meet a plate is left behind in the space that records their meeting, documenting the time and place of the encounter as well as the video-captured faces behind the never-changing avatars. However, it should be possible to make this a much more complex function. These memories on a temporal axis might be compared to the dramas inscribed in the individual wrinkles on an old person's face. But that is not to say that we need to create avatars with the faces of old people. Because these plates are basically the products of the connection of two times they were originally called "nodes." By designing these "nodes" as a kind of crossroad it would be possible to render time non-linear and go back to an earlier crossroad, to head off in another direction, or to retrace a path. In addition to tracing the steps of others it would be possible to retrace one's own steps as well.

In the original design *Global Interior Project # 1-3*, these nodes were rendered as spheres and any user of the application was able to go inside them. The idea was that once inside you could see and listen to the conversation that had taken place there on video. But since we were dealing with an exhibit that would accommodate any number of users Fujihata decided to limit the number of these node spheres. If he had not done so the space would soon have overflowed with nodes. Limiting their number, however, gave rise to a phenomenon that was completely impossible to understand.

If the following facts were established, ultimately the system rendered an unsolvable problem:

1. There are a limited number, perhaps ten, node spheres in the world.
2. Each time a new node sphere is created the oldest one is extinguished.
3. Say that two avatars meet in the oldest sphere and create a new one.
4. The birth of the new sphere causes the oldest one to be erased.
5. Because the new node sphere came into existence beneath the world of the oldest one, the subordinate new one is erased along with it when the old one above disappears.
6. This means that the two avatars which created the new node sphere must disappear as well. (This is where the problem arises.)

This special instance shows how objects created to record encounters erase the entities out of which they themselves arose. This is a very clear example of the way a highly common-sensical judgment can naturally create discontinuous fissures in a realistic world. Without any doubt, similar incidents actually occur in the real world we inhabit.

This particular case makes one realize that the everyday continuous "sense of existence" we take for granted is actually produced by a ceaseless effort on our part as we live our lives and go about the daily work of repairing these fissures. By modeling human relations and inserting them into cyberspace we find ourselves beginning to question the spaces we inhabit in the real world and the functions of those relations. These questions probably belong to the realm of philosophy. But the novelty produced by this kind of technology results from the fact that these questions arise not in language but in an experiential space. And finding answers to them cannot be accomplished through contemplation based on language but only by the actual attempt to programme the world. Constructing a fissure-less and continuous world view through real programming is an extremely difficult task.

Another example can be offered. The present work, as will be explained in the next section, uses multi-cast technology and lacks the central administrative device known as a server. This means that each avatar gathers information on its own and constructs a world which is then displayed visually to the user at the terminal. In fact there is no guarantee that the same world is being displayed on every terminal. The problem here arises when someone joins in late. He or she would lack the information relating to events which have been recorded in the space. This situation seems to conform best to

reality. However, for exhibitionin a museum the system had to be designed in such a way that each time this occurred the entire world would be reset. This is another example of privileging the continuity of the world as a whole.

Content and Form

The Godless World of the Serverless Network.

The world view of *Nuzzle Afar* is made possible by network-based computer technology. The information made visible by each computer terminal for users is permanently updated by collecting information distributed throughout the network transmitted by others, along with the information stored and distributed in advance. If we think of each computer as a human being, expressions on the display are renewed by the information spoken by each person. Even if the users remain motionless, a conversation is always going on among the computers on the network. Here there is no superior system that understands the whole and records it as necessary. Networks which do have such a superior system at the center are of the server-client type. In these systems each individual computer constantly reports on ist situation to its superior and obtains information about others by retrieving that information from the server. Clearly, creating this kind of system (or world) may cause the information exchanged there to double in quantity. But a "democratic" design has the advantage of keeping the information inside the network to a minimum. Indeed, in this democratic world the model whereby each individual gathers the information he or she requires is much closer to the situation in the real world. In this sense the greatest problems with the technology that underpins this work are quite different from those with older technologies. Works that thematise this kind of network are characterised by the great number of problems which have to be determined in profound relation to the problems of the forms of information distribution and the priority of values.

Delay and Accidents.

The advantage of a network is that it makes it possible to connect a number of sites which are separated geometrically from each other. And yet one always has the problem of information delays caused by the network. In November 1998 when we connected the Shonan Fujisawa campus of Keio University in Japan to the ZKM in Germany it took approximately 0.3 seconds for information to be transmitted and come back. This was a completely insurmountable temporal delay which demonstrated that no two *Nuzzle Afar Worlds* realized on different computers are ever exactly the same. This kind of delay can also sometimes cause accidents. When, for example, two avatars come together and form a node sphere it is possible that one of them might escape into another space before it receives the information about the collision. The result is that a node is formed without one of the partners appearing. The world will not collapse as long as the terminal is able to absorb the delay, but when it fails to do so a discontinuous fissure opens up. Like the ethical fissures mentioned earlier there can also be rifts on the temporal axis.

Display Mechanisms and Identity.

Nuzzle Afar was designed as an exhibit. The understanding was that an indeterminate number of people would visit the exhibition and a certain number of them would work with the terminals. Because the avatars floating in the space are expressed in each terminal as single avatars they lack a unique identity. In fact it was impossible to give them a unique identity. Also it was impossible to install complicated interfaces. For this reason Fujihata used track balls, no keyboard or buttons. In addition a microphone and a camera were installed to simplify the communication. The small red track ball was the only tool with which the user could control the world. Nonetheless it took some time for the user to learn how to navigate and explore the projected world. In order to communicate the richest content possible through an interface absolutely easy even for an unexperienced visitor to operate, different forms of reporting information in the exhibition space had to be developed. It was like moving from the 1960s when televisions were shared by the public on the street to an age in which each home had three or four television sets.)

The Fantasy of World Continuity.

It would be a mistake to view this work in the context of conventional "Electronic Art". It is not possible to survey all its dimensions from the perspective of "Man and Machine". Discontinuous worlds cannot emerge in this kind of exhibition space. Because the work is actually only a machine which operates twenty four hours a day these would be judged as bugs or break-downs. It is not the purpose of this work to manifest a mechanical world view. What it teaches us is that the world is actually ridden with fissures. "Reality" is something that is produced at every moment as one lives one's life, where damaged sites are constantly being repaired. These fissures are sewn together with great finesse and appear most characteristically in what is known in the debates over virtual reality as the "process of making the virtual real" or vice versa. The fact that we live our lives today without being cognizant of these fissures is attested to by the fact that we all believe being modern; we believe that we are able to transcend them. The vast majority of works dealing with this kind of virtual space have been created and spoken about only for the sake of this novelty and chic image. But Fujihata believes it is more important to have an accurate awareness of these fissures. And the only way to accomplish that is to move away from the site of action and take one step back. The sense of reality as something inconsistent and discontinuous lies within actual experience. It is something we are made to ignore. And yet the viewers of a work will not wait for this. *Nuzzle Afar* offers the experience of taking a step back in a different form as the experience of the viewer. Its goal is to make the fissures with reality appear in front of the user, and by this prepare him for the advent of future technologies.

Future Possibilities out of Problems : The Limitations of the Exhibition Space and towards an Internet Version

The age is over when users gratefully read the outlines of a world of value realized in cyberspace by expensive computers. Within the next ten years computers will cost the same or less than a telephone. The question is how much resistance and creativity we can bring to bear on the future which this technology will bring.

When a certain car company asked a group of elementary school students how they imagined the cars of the future one of them asked why car steering wheels were not like the joystick of a Nintendo game. This question shows that for this child the interface with the world is best exemplified by the interface of a computer game. At the same time it suggests that anything can serve as an interface for controlling the world as long as it is standardized. But the mode of manipulation will change as the interface does and of course with that there will also be changes in the types of objects which are easiest to manipulate. Ultimately this will mean a change in the way we see the world.

In this sense *Nuzzle Afar* is beset with a certain dilemma. One purpose cetainly is to familiarize the users with specific and entirely new experiences and new ways of viewing the world (and communicationg within it). It was designed out of a desire to complete those experiences and world views as extensions of the framework of conventional art. But in order to completely fulfill this desire it is clear that the environment of the user itself must be transformed as well.

Place - A User's Manual

Adolf Matthias, ZKM

The following sections provide an overview of the technical components involved in Jeffrey Shaw's *Place - A User's Manual*. It also gives a short but complete description of the compensation of the cylindrical projection distortion that was used for *Place*.

Overview

Place consists of a cylindrical projection screen of approx. 9 m in diameter that is approx. 2.6 m high. In the centre of this screen, a motor-driven rotating platform carries the graphics computer, the image projector, a modified video camera with an LCD viewer and a microphone mounted on a camera stand, and the viewer.

Graphics Rendering

The Virtual World

The virtual world of *Place* consists of an infinitely replicated Kabalah 'Tree of Life' diagram on the ground plane onto which cylindrical objects textured with panorama photographs are placed.

Triggered by sound events, three-dimensional capital letters flow into the scene. They remain within a circular environment around the viewer, and start to fade and finally vanish after a certain time. A cloudy sky spans over the whole scene.

Wide Angle Projection

The two versions of projection systems used in different versions of *Place* are described here. The location of their respective projection origin that is relevant to the distortion considerations below is discussed here.

Multiple Projectors

In order to provide a very wide angle section of the full 360°, a version of the projection system uses 3 projectors placed on a table on the rotating platform. The distance between the perspective and the projection origin stems from the size of the projectors, their centres being approx. 30 cm to 50 cm from the projection screen's centre and approx. 40 cm below the vertical centre. The two outer projector's optical axes are not radial.

Single Projector with a Mirror

A new generation of light-intensive projectors made it possible to use only one projector. The width of the projection sector is increased by using a large mirror in order to lengthen the virtual distance between projection origin and screen. The projector is placed near the front edge of the platform and projects backward onto the mirror that is behind the platform's centre. The virtual position of the projector thus is approx. 1.5 m behind the centre of the cylindrical screen, and approx. 60 cm below the centre. The projection is radial when seen from the top but is slightly inclined upward.

Distortion

The optic distortion resulting from the projection onto a cylindrical surface is compensated by a dynamically computed counter/distortion of the 3D objects prior to rendering.

The basic idea is that when viewer and projection origin are in one point, the shape of the projection surface doesn't matter at all, whereas a given distance vector between viewer and projection origin results into a noticeable distortion of the projected image.

As in all popular panoramic paintings, the origin of the viewer's perspective is assumed to be at the center of the cylindrical screen. The projection origin is the optical centre of a projector placed inside the cylindrical screen.

What the viewer expects to see of a point in space is the intersection of the line from the viewing point to that point. The projector has to illuminate that intersection point on the screen in order to render the desired point in space which it wouldn't if things were not corrected.

In the current version of *Place*, the correction is done by replacing the point $\vec{x}$ with $\vec{x}'$ according to

$$\vec{x}' = \vec{x} + l\,(\vec{p} - \vec{o})$$

with

$$l = 1 - \frac{\left\|\vec{x} - \vec{o}\right\|}{r}$$

is the viewer origin, the projector's origin, and r the radius of the cylindrical projection surface. $\| \ \|$ here denotes the orthogonal distance to the projection cylinder's axis.

It should be noted that this distortion has to be applied after linear transformations of the scene or the camera point have been carried out. In order to reduce the computational load associated with full 3D transformations, *Place* uses only rotations within the plane represented as multiplications with complex numbers of absolute value

The entire distortion process applied to vertices of the scene is a combination of this geometric transformation with the process described above.

As shown in the dataflow schematic, *Place* uses one analogue and four digital inputs that are connected to the platform controller, a computer that also provides control signals for the platform rotation motors:

- An analogue input for the rotation angle of the camera stand that is used to control the platform rotation and, coupled to it, the viewer's rotation in the scene.
- A digital input that is activated when the camera microphone is exposed to sound above a specified threshold; this input triggers the flow of text appearing in front of the viewpoint.
- Three digital inputs connected to pushbuttons. Two of these are used to control forward and backward travel within the scene, and the third one causes a jump of the viewer into the center of one of the cylindrical panoramas once it has been entered.

The platform controller transmits the state of its analogue and digital inputs to the graphics computer through a serial interface.

The Web Planetarium in the EVE dome

Detlev Schwabe (ZKM) & Mårten Stenius (SICS)

The Extended Virtual Environment EVE, is a unique implementation of a 'window-into-a-world' paradigm. An inhabitable three-quarters of a sphere projection dome contains a rotatable stereo-projection device in the center. An observer, standing inside the dome, is able to look everywhere onto the surface and the projected images will follow the motions of his head and will always be centred according to his line of sight. This paper describes the merging of EVE and The Web Planetarium. We will discuss design issues, choices and some initial observations made during set-up and use in public during exhibitions.

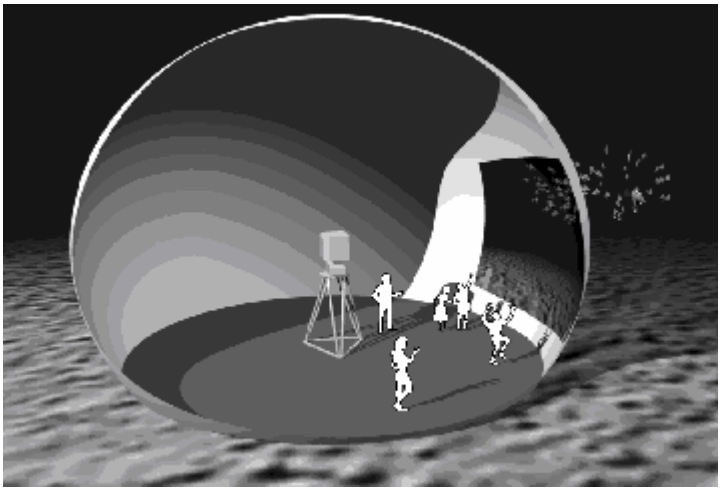


Figure 20: A schematic view of the EVE dome showing it immersed inside a virtual world

Although EVE is not meant to be a replacement for a CAVE (Cruz-Neira et al, 1993), it can be a less cost-intensive alternative, especially if larger audience groups of 40-50 people are targeted. While it currently cannot compete with the immersiveness of a four or even six-sided CAVE, the special 'window' paradigm has an appealing quality of it's own, which can be exploited by applications as will be shown with the Web Planetarium. The EVE was originally designed and built for the first time in 1993. One of the serious drawbacks of the original projection system and the related tracking system was the latency between the controlling user's head and the motion of the motor-controlled projection head. In this paper we describe the redesigned pan & tilt head as well as the new approach for the tracking of the user's head.

The next sections give a technical description of the EVE, followed by a section on the Web Planetarium including a discussion on a range of technical and conceptual design issues.

Technical Description

EVE is a sphere-like projection dome with a diameter of 12 meters and a height of 9 meters made of soft, inflatable fabric. A constant air supply is responsible for maintaining the dome's shape. The inner part of the skin is used as the projection surface. A rotating door entrance prevents the air from escaping outside.

In the original set-up, a magnetic head tracker was used to detect the current orientation of the user's head. Beside the fact that there was always a cable going from the user's head to a computer at the centre of the dome, the significant problem was the latency between the almost immediate update of the shown imagery and the much slower actual repositioning of the projectors. One would have to simulate the acceleration and deceleration of the motors in software to improve the spatial synchronisation between the position of the projection and the shown virtual scenery. To overcome this problem, a new approach to the tracking was chosen. This chapter describes the central redesigned hardware and software components.

Pan & Tilt Head

The central part of the EVE dome is a stereo-video projection apparatus which can be rotated motor-controlled by 360 degrees around the vertical axis and has a rotation range from approx. –15 degrees (pointing slightly downwards) to 90 degrees (pointing straight up) about the horizontal axis. The projection head is mounted on a tripod so that the centre of the projection coincides with the centre of the sphere. The projectors (two Synelec LightMaster (http://www.synelec.com)) utilising Texas Instruments DLP technology (http://www.ti.com/dlp)) support a 800 by 600 pixel resolution and have a wide angle lense providing a 60 degree horizontal projection angle. Linear polarised filters are mounted in front of the lenses to separate the stereo images. An audio speaker system with four mid-range speakers is also built into the head. In conjunction with a sub-woofer system at the basement of the tripod, a good sound system is available. All necessary signals for RGB video, power supply, audio, motor control and serial lines for configuring the projectors are brought into the head via a slip-ring unit.

The head automatically follows the movements of one visitor's head who carries special polarised glasses with a mounted infrared light pointer. The infrared light spot on the dome surface is tracked by an infrared camera, also mounted inside the head. The camera image is analysed by tracking software, running on a PC which is installed in the basement of the tripod. The tracking software determines the position of the light spot in relation to the centre of the camera image (which coincides with the centre of the projected images) and calculates acceleration and deceleration values which are sent to the servo amplifiers via a serial controller. As a consequence the tracking software controls the motors for the horizontal and vertical motion of the head so that the center of the projected images are coinciding with the viewing direction of the visitor.

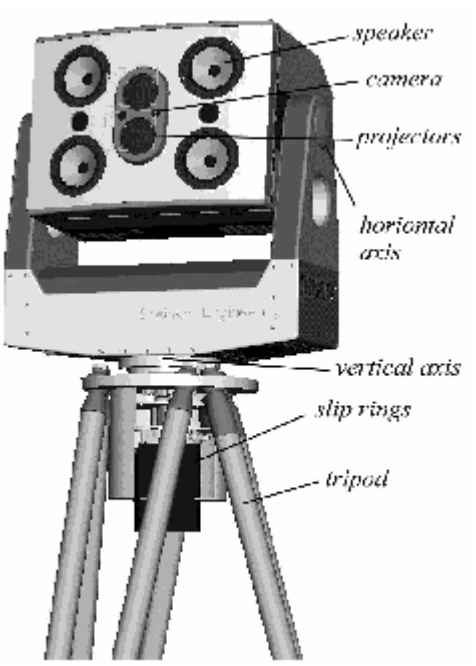


Figure 21: The pan & tilt projection head

Application Interface

Currently the application interface consists of a small circuitry board which interfaces the two angle sensors of the pan & tilt head as well as a 5-channel wireless joystick with the RS-232 serial port on the application machine. A shared-memory-based API serves as the software interface between the actual application and the state of the pan & tilt head and the joystick. A background process running on the application platform, continuously reads the current status of the angle sensors and joystick and writes these values into a shared memory buffer. An application connects to the shared memory area at program start and is then able to access the current values at any time. Additionally, the EVE API also provides a simple event queue for the 10 possible joystick events (5 buttons can be pushed or released) for the application programmers convenience.

Interface

To measure the current orientation of the pan & tilt head, two absolute angle sensors, each with a 12-bit resolution are built into the head. For mechanical reasons both sensors are installed and aligned to the vertical rotation axis. As a consequence one sensor actually measures the sum of the horizontal and the vertical angle, while the other delivers the value for only the horizontal angle. The vertical angle is determined by subtracting the latter from the first, regarding the possible overflow due to the 12-bit limit. To read out the current value of the sensor device, the interface transmits a pulsed signal to each sensor in order to receive the value bit by bit..

The wireless joystick is a standard PC game joystick, reconstructed for wireless connection, with an integrated thumb knob, one fire trigger and three generic buttons of which one is not activated. A dedicated radio receiver is able to receive five different functions from the joystick which are used as forwards and backwards on the thumb knob, the fire trigger and two of the buttons. The circuitry is responsible to read out the status of the angle sensors as well as the joystick receiver. Then it sends a complete data block to the application platform over the serial port. The refresh rate lies at approximately 40 fps, which is currently the limit at the used communication speed of 9600 baud.

Application Platform

Basically any computer hardware set-up which is capable of producing a synchronised set of two 800 by 600 pixel resolution images can be used as the application platform. The current applications are running on a two-processor 150 MHz, R4400 Silicon Graphics Onyx RealityEngine2 with a multi-channel option installed. Out of the three possible 800 by 600 pixel channels only two are used at the single available refresh rate of 60 Hz. In order to use the multi-channel option, the frame buffer must be configured to 2400 by 600 pixels (3 times 800 by 600). Since this resolution cannot be shown on a regular 19" monitor, a VT320 terminal is connected to the machine for administration and control puposes.

To be able to select and start different applications from inside the dome, a simple application chooser utility has been implemented which is fully controllable with the wireless joystick. A configuration file is used to define which applications are available and how they are started. With the thumb knob one can move through the list of applications and can start one by pressing one of the buttons. By convention all applications must be able to be terminated by pressing the fire trigger and the other two buttons simultaneously.

The Web Planetarium

The *Web Planetarium* was originally developed at SICS (http://www.sics.se) as a desktop application and was implemented using the distributed virtual environment software platform DIVE (Frécon, 1998; Hagsand, 1996). The application visualises the structure behind World Wide Web documents and hyperlinks as a 3D virtual world of planet-like abstract objects and connection beams, and elaborates on concepts introduced in WWW3D (Snowdon et al, 1996). An object is a 3D representation of a corresponding web page and, once the user is inside, displays the hyperlinks on that page as additional small objects on which the user is able to click in order to fetch new pages, and thus extend the 3D graph with new site representations.

Visual Appearance

In the Web Planetarium, the aim has been to exploit enhancements of the visual appearance to improve the navigability and overall user experience. The external

representations of a site are derived by mixing a basic artistically inspired shape with textures from the underlying web page. This gives a visually rich electronic landscape that is visually appealing, and thus suitable for a public exhibition-style display such as the EVE, but at the same times helps navigation by presenting abstracted visual cues that can used when navigating and finding a way through the structure.

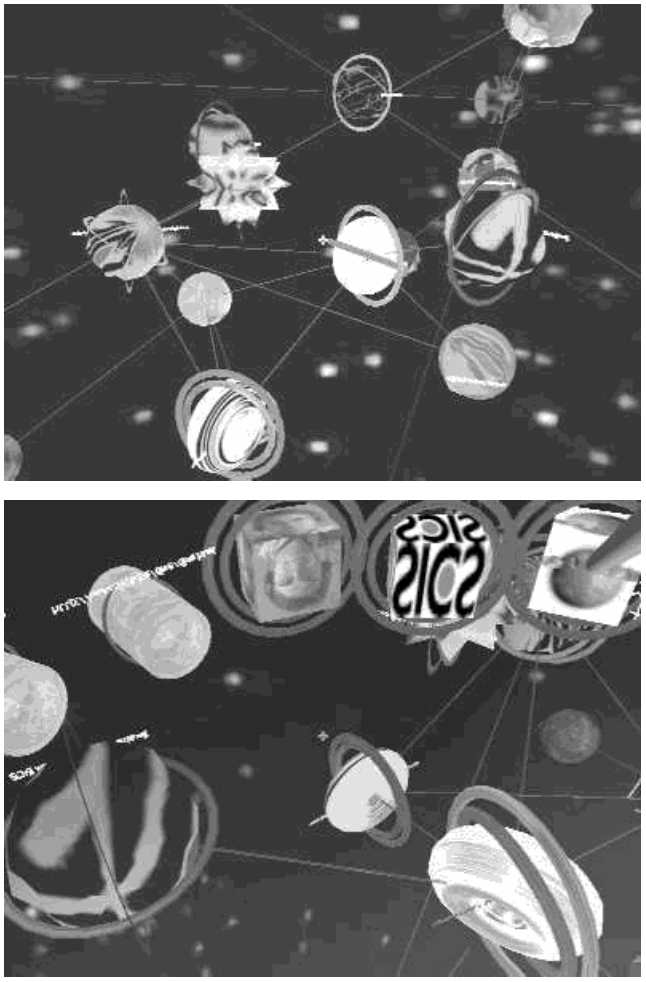


Figure 22: The left snapshot shows a view of several objects representing web pages while the right one shows a view from the inside of one of the objects: Two tubular link objects at the lower left as well as three image links can be seen. Also note the crosshair in the middle of each image, which is the point of interaction.

Sonic Experience

Given the planetary- or observatory-like nature of the EVE dome, the Web Planetarium is intended to exploit the metaphor of exploring „outer space“ while navigating the Web. To strengthen this experience, space-like sound effect has been added as feedback on different events:

Following a link or zooming in to a site results in a ‘take-off’-like sound.

- Entering a site produces an ‘opening’ or ‘trespassing’ sound.

- Clicking on icons produces a ‘tingling’ sound.

A vital function of these sound effects is to improve the aesthetic experience, and complex visual appearance of the planetarium. Thus, rather than having one simple sound effect for each of the categories listed above, the Web Planetarium has three *categories* of sounds. The sounds within each category resemble each other, but vary enough to produce an interesting soundscape. When, for instance, a link is followed, a takeoff-style sound will be played, but the exact quality of the sound will vary from time to time.

A Mix of Function and Experience

The working mix of variation and predictability that has been explored in the Web Planetarium has shown to be well suited for the public exhibition settings in which the Web Planetarium has been shown. This partly opposes traditional design guidelines for graphical user interfaces, where a very strict consistency is typically promoted. However such guidelines typically are worked out with workplace desktop settings in mind. The Web Planetarium intently breaks this strict uniformity, and this has shown to be appropriate in an public museum setting such as the exhibition in the EVE dome, where people not only expect *function* but also an *experience*.

Navigating the Landscape

The user is able to freely navigate in 3D space, enter objects either manually or automatically by clicking on the object or on a connecting beam. By clicking on a link icon within a page, the user fetches new pages and is automatically transported to the site representation of those pages once they are loaded from the Internet. A direct point-and-click interface has been proven to work well with the EVE dome, and to present an interface that is aimed at simplicity rather than sophisticated features, since the intended users of the installations are random passers by on an exhibition. The whole application is navigated and used only by looking in the desired direction, moving backward and forward, or clicking on an object of interest.

Merging two user interface metaphors

The Web Planetarium application is implemented as a DIVE process, and uses EVE as the display and interaction medium. To provide a workable merge between the two systems, some fundamental differences in the interaction methods had to be taken into account and mapped correctly to produce a natural user interface. This has been achieved by developing a device driver that provides a mapping between the EVE hardware and software setup and the interaction features of DIVE. The device driver is realised as a DIVE plugin that connects to the EVE using the EVE API described previously in this document.

In DIVE, the navigation in the environment is independent on the actual interaction point used for interaction. This is typically facilitated by combining arrow keys and mouse interaction on a desktop, or by using different buttons for different functions on a wand. In the EVE, the spatial navigation device is directly coupled to the rendering point: That is, the rotation of the input device (the laser pointer) is controlling the viewing angle of the scene display through the hardware setup. The joystick supplies no rotational information, but rather a number of buttons and digital directional information.

Furthermore, the EVE dome is unique in its combination of very large display size and spherical curvature, and completely different from previous display settings used with DIVE (desktops, flat wall-mounted panels, HMD:s, CAVE-like setups, etc). A means had to be developed to let DIVE and the Web Planetarium be presented in the EVE without the production of undesired visual artifacts and with a good stereo rendering. Given these prerequisites, the mapping between the EVE interaction and rendering devices and the DIVE interaction and event model emphasises four key subjects, which will be considered in turn.

The point of interaction

The rigid coupling between viewing angle and pointer movement of the EVE has been handled in DIVE by replacing the freely movable focus ray with a crosshair, always centred in of the rendering window. This crosshair thus functions as the point of interaction and follows the line of gaze of the user naturally, since the rendering window of the EVE is directly synchronised with the head movements of the (leading) viewer.

By using the set of buttons on the joystick more extensively, it is possible to extend this mapping to generate all kinds of DIVE interactions (grasping, clicking, and dragging objects). For the purpose of the Web Planetarium, only the clicking (selecting) signals are generated.

The rotation and position of the avatar

The current EVE interface produces backward-forward signals, controlled by joystick movement, while left-right signals currently are not transferred. By mapping the backward-forward interactions directly to a backward-forward movement in the current line of gaze, the user can navigate the DIVE space in all three dimensions – left-right and up-down rotations are taken from the viewing angle (head orientation) of the user.

Observations made during the public showings revealed that most users tend to move mostly *forward* in the line of gaze. A possible reason for this could be that there is a tendency to move towards what you see, to look closer, rather then back out to get an overview. Furthermore, the Web Planetarium also has an interface that builds much upon *zooming in* on what you see, rather then backing out. This inherent tendency of the users moving forward, however, led unexpectedly to a spiralling *upwards* in the scene. This phenomenon is attributed to the fact that a large part of the lower hemisphere of the „rendering sphere“ is inaccessible due to obvious limitations of the pan/tilt head and the floor in the dome. Even though it certainly is possible to look

upwards and „back down“, this movement is unnatural enough to rarely be used, and typically a guide had to reset the user position regularly when showing the installation to inexperienced users.

Dynamic change of the eye separation

The user of the Web Planetarium typically navigates the visualisation in two major phases: *Roaming* the ‘space’, looking at the graph of sites and links to choose an interesting site, and *visiting* a site, standing inside a site representation to look at its contents and possibly for links to new sites. When roaming, the site contents are hidden inside their planetary abstractions, and when visiting a site, the outside space serves only as a background and visual reference to the outside world.

While experimenting with the application, and tuning the stereo rendering, it turned out that while a good stereo effect was achieved in the narrow space within a site, the same eye separation[1] would produce an almost flat experience when the user roams the space. Conversely, an eye separation suited for a good stereo separation when viewing the overall scene would yield far too extreme results when viewing objects at a closer distance inside a site. This problem, of course, arises from the broad range of scale in size presented in the Web Planetarium visualisation: The scene graph is typically presented with sites separated by tens or hundreds of virtual meters, while the site contents are on the scale of meters or fractions of meters, all within the same Euclidean space. To overcome this problem, and give an acceptable stereo rendering, two solutions were considered:

1. Keep the eye separation constant, but allow for a difference in distance scale between different regions of the visualisation.
2. Keep the overall uniform distance scale, but change the eye separation according to where you are in the visualisation.

The solution chosen was the second one, since in the Web Planetarium it is easy to implement by simply modifying the state of the avatar when it crosses the border between the internal and external side of a site representation. However, a more general solution would be to apply scaling factors to regions of a virtual space, an architectural issue that was beyond the immediate scope of this experiment. However, given such schemes, the Web Planetarium could be a good example of applied use.

Support for the EVE image warping

The curved surface of the EVE puts special demands on the rendering to produce an distortion-free image (see previous discussion). The implementation for other applications in EVE to do this image warping makes use of the texture memory of the workstation. This, however, is not readily possible to do with DIVE, since DIVE

[1] Stereoscopic rendering in DIVE is fine-tuned by altering the eye separation of the avatar.

renders textured 3D worlds in real time, and thus already uses the texture memory. This could potentially be solved by letting DIVE render into some temporary buffer rather than into the frame buffer, but in the current implementation the DIVE rendering is unwarped. Given the relatively small field of view, the distortions are small enough to allow for an acceptable visual experience, but rectifying this nevertheless is a potential future extension of the interface.

Acknowledgements

Acknowledgements go to Jeffrey Shaw who envisioned the original EVE idea and the iC_inema and rePLACEd applications, to Armin Steinke who designed and built the new pan & tilt head, to Ralph Kondziella who programmed the tracking and motor-control software, to André Bernhard who designed and constructed the interface card for the angle sensors and the wireless joystick. For the Web Planetarium, acknowledgements go to Lennart Fahlén who has been a driving force behind the whole application concept, to Bino Nord who provided the graphical design of it, to Jonas Söderberg for the soundscape, and to Anders Wallberg for invaluable help with the software development

Section Two
The technology of the abstract electronic landscape

Chapter3
Q-PIT and Dataclouds: the generative algorithms

John Mariani and Andy Colebourne
Lancaster University

The abstract landscape of the Virtual Planetarium/Library demonstrator is generated by a combination of data-placement routines which organise the virtual world in meaningful groups, and 'hulling' algorithms that highlight these regions to the inhabitants. In this chapter we describe in detail the techniques that generate this virtual environment.

Generating the Q-space

There are 6 broad steps in generating a new Q-Space containing data clouds

1. Generating a Benediktine space for initial "random" positioning of points within the space.
2. Generating a similarity matrix
3. Generating the minimal spanning tree
4. Identifying regions within the tree
5. Applying a force displacement algorithm
6. Generating the data clouds

Benediktine starting point

This was the basis for the organisation of the Q-Space in the original Q-PIT system. In order to map N-dimensional data onto a 3-dimensional display space, we consider the values of a tuple as co-ordinates within the space. Three of these values are mapped directly onto (x,y,z) positional co-ordinates (extrinsic dimensions), and others are mapped onto appearance of the object's representation (intrinsic dimensions).

This is done by processing the contents of a single relation database (the Q-PIT 'relation') and extracting ordered lists of the domain values. Some of these are mapped directly onto numerical values (the index of the field within the ordered domain), and some onto appearance or behaviour (in earlier prototypes, the 'spin speed' of an object would be dictated by the underlying field value).

These mappings are specified by a user-supplied ‘mapping file’ which identifies which fields are contributing to the intrinsic and extrinsic dimensions. Where shapes are used, the user has to state which field is being employed, and a list of (value, shape) pairs.

```
extrinsic isbn x
extrinsic author y
extrinsic title z
intrinsic type shape "oversized book" cube book sphere
```

On start-up, Q-PIT processes the domain information and the mapping file to produce a set of (x,y,z, shape) co-ordinates which it now stores persistently within the so-called Q-PIT ‘relation’. These points can then be plotted within the Q-Space as a kind of ‘random’ starting position before the application of the FDP algorithm.

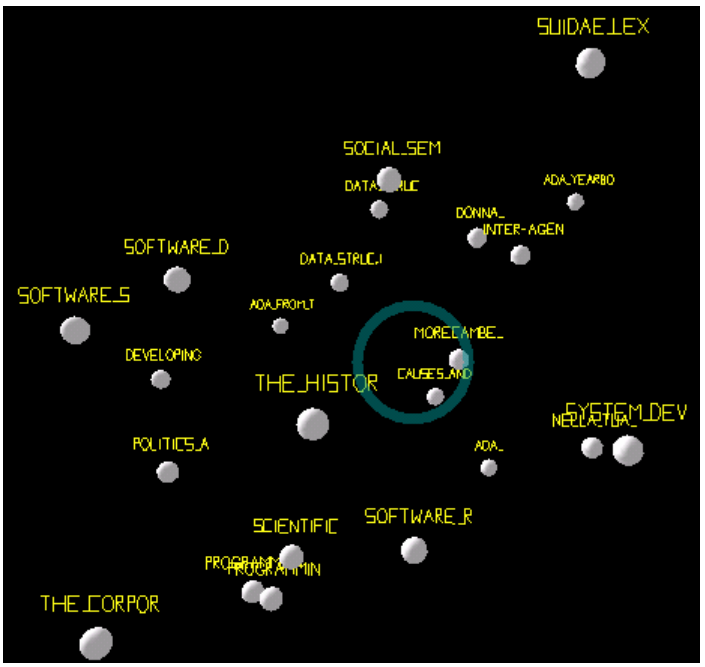


Figure 23; after the Benediktine process

Figure 23 shows a sample Q-Space consisting of 23 books found as the result of a keyword request of ‘ada’ on the library OPAC system. They are mapped out according to our Benediktine approach.

Similarity Matrix

In response to the poor performance of the original Benediktine Q-Space in terms of semantically meaningful spaces, we decided to move towards a similarity-based layout

which should generate clear and meaningful regions, visualised as data clouds. The first step in generating the cloud-based Q-Space is the production of a similarity matrix.

This is accomplished as follows : each tuple within the Q-PIT "relation" is compared with each other, in order to produce a similarity measure between 0 and 1. Each measure is then recorded in the similarity matrix. Unfortunately this is an M * M process.

The single similarity measure is generated as follows : each field of a tuple is compared with the corresponding field of another tuple. This comparison is based on the ordered domain lists generated as part of the Benediktine process. The absolute distance between two values is based on their position within the domain list.

Fm : field measure
Nvd : number of values in the domain
D : absolute distance between the two values within the two tuples
Fm = (nvd-d)/nvd

To try to speed up this process, the Q-PIT ‘relation’ now stores the mapped values for each field so that we only calculate this once and do not need to perform a ‘domain lookup’ for each field in each tuple over and over again.

The tuple similarity measure is generated by summing the individual field measures and dividing by the number of fields.

User-specified weightings

At this phase of the processing we can add some user-specified influence over the generation of the cloud Q-space. When we generate an individual field measure, we can multiply it by some user-specified factor. This means we can decide which fields are more or less important than others when it comes to generating the Q-space. For example, in the Film Finders Q-PIT, we might decide we are more interested in genre and a particular actor than we are in actress or director. In the Library Q-PIT we might decide that ISBNs have no semantic meaning whatsoever and that they shouldn't influence the similarity measure at all, so we can associate a zero factor with that field.

This is currently undertaken statically, via the Benediktine mapping file. The user can provide a list of (field, factor) pairs. If a field isn't named, the factor is 1 (one). In future developments, we intend to provide a set of slider controls ranging from (0.0 to 10) to allow more dynamic specification of these weighting. This would be associated with an animated display of the Q-space taking up its new configuration. (Clearly, by changing the similarity weightings we must begin the Q-space generation process almost from square one. However, all the points in the space would already be present within the visualisation and would be moved -- in an animated fashion -- to their new positions).

```
extrinsic isbn x
```

```
extrinsic author y
extrinsic title z
similarity author 3
similarity isbn 0.1
similarity classmark 10
```

Figure 24: revised mapping file

Minimal Spanning Tree

The next step in the process is to generate a minimal spanning tree (MST). This begins by converting the entire similarity matrix into an equivalent graph. An edge in this graph consists of a (u node, v node, weighting) triple. This set of edges is then sorted in order of descending weight.

The implementation is based on Kruskal's algorithm as described in [Weiss 93]. The algorithm considers the set of edges to be a forest of trees. This means initially each edge is a single tree. When we add an edge to the MST, we are merging two trees into one. The algorithm terminates when there is only one tree (this is the MST). The core of the algorithm is in deciding whether an edge (u, v) should be accepted. It is actually quite simple to make this determination.

Two vertices belong in the same set if they are connected in the current spanning forest. Initially, each vertex is in its own set. If u and v are in the same set, the edge is rejected, because since they are already connected, adding (u, v) would form a cycle.

Figure 25 shows the Q-Space now featuring the arcs of the MST.

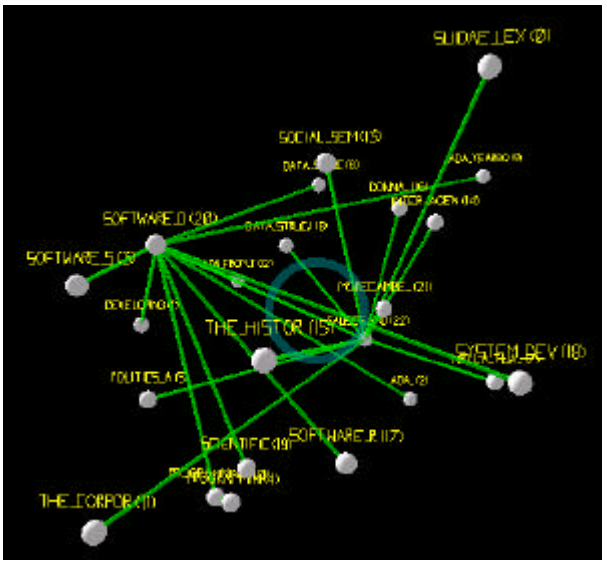


Figure 25: visualising the arcs in the MST

Regions within the Tree

The next problem is to form regions within the data. Following the work of (Ingram, 95) we have used the algorithm employed in the LEADS system, that of (Zahn, 71). Regions are represented as identified sub-graphs within the MST. Zahn's clusters are produced by identifying and eliminating ‘inconsistent’ edges, defined as edges of the spanning tree whose values are significantly greater than the nearby edge values. We can thus identify inconsistent edges by comparing each edge with its close neighbours. One method of doing this is by finding the ratio of the length (or weighting) of the current edge to the average of nearby edges. Associated with the algorithm is an adjustable threshold level; if this threshold is exceeded, then the edge is inconsistent. (Note that here is another point where the user could influence the generation process).

Once these inconsistent edges have been removed from the edge set, the nodes of the resultant isolated trees represent a cluster within the data. As Ingram points out, ‘the advantages of this algorithm are its obvious simplicity and the way that it forms clusters on the basis of the data itself, not requiring the number of clusters to be predetermined’.

To represent membership within a region, each region is considered to have a unique colour. At this stage, we colour each point in the space accordingly. Points which do not belong to a region (or rather, are the sole member of their own region) remain white. In our example, there are three non-singleton regions, and these appear as yellow, blue and green.

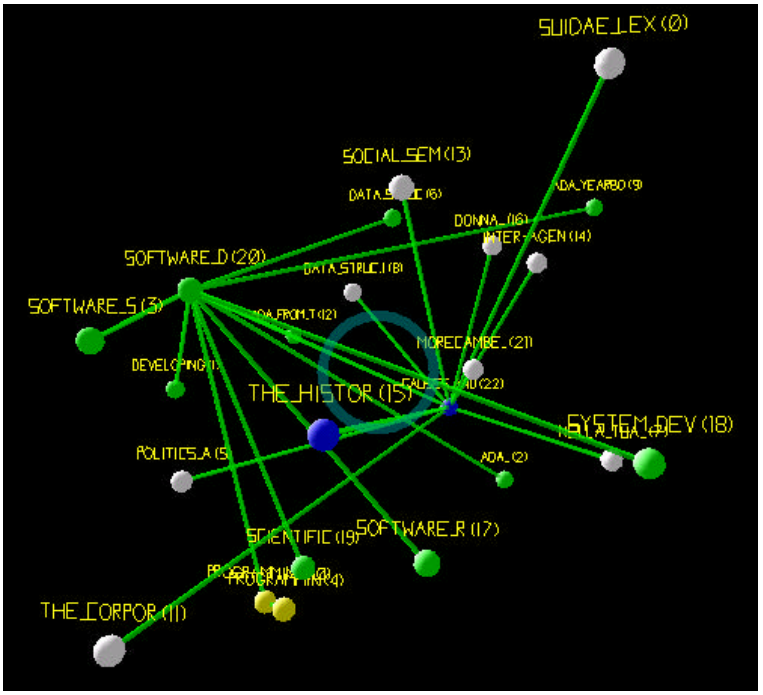


Figure 26: showing region membership via colour

Force Directed Placement

The penultimate step in producing data clouds where closeness of points in the Q-space have some underlying meaning is to apply a force directed placement algorithm to map the points in the space according to their positioning within the minimal spanning tree. In order to do so, we have applied the algorithm published by (Fruchterman & Reingold, 1991). The algorithm adapts Eades's spring-embedder model but has been developed in analogy to forces in natural system—the nodes in the graph are connected by springs but can be thought of as atoms within a gas whose motion is connected to the current temperature of the gas. The idea is that initially displacements can be quite large but as the "temperature" cools the displacements gradually become smaller until the nodes approach a stationary state.

The algorithm runs as follows: for a number of iterations, we calculate the repulsive forces, then the attractive forces, and finally limit the maximum displacement according to the current temperature and to keep the points within the display frame.

Each vertex has two vectors—pos (position) and disp (displacement). To generate the repulsive force, each vector is compared with every other, and we calculate the displacement of a vector relative to the other. These displacements are simply summed to arrive at an overall displacement for a vertex with respect to all other vertices.

To generate he attractive force, we compare each vertex with every attached vertex and again calculate the displacement. As before, these are summed into the overall displacement.

Finally, with respect to display frame and current temperature, we actually apply the displacement and change the position of the vertex.

The algorithm can be parameterised to some extend as we need to provide three functions that will directly affect the displacements :

- A cooling function that dictates how the temperature changes
- An attraction function which forms part of the attractive force calculation
- A repulsion function which forms part of the repulsive force calculation

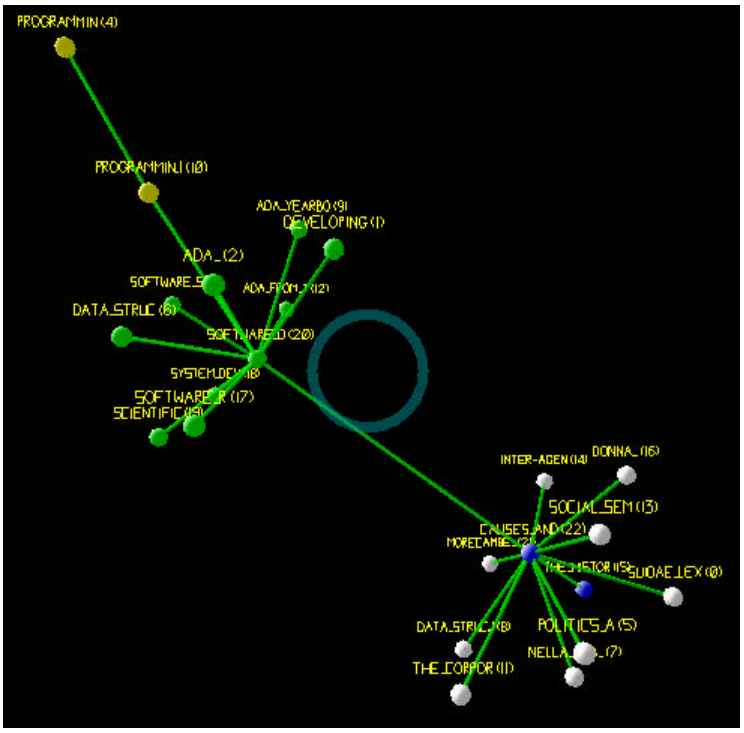


Figure 27: after FDP

Figure 27 shows the graph after FDP. It should be clear that those points which share region membership are now co-located in the space. Most of the Ada programming language books reside within the green region, with links to two specific programming books forming the small yellow region.

Conclusions

In this chapter, we have described 5 of the major processing steps which go from a Q-PIT 'relation' to generating a clouds-based Q-space. The steps alternate (to some extent) between data-based processing to display-based concerns. The generation of the similarity matrix, minimal spanning tree and region identification are all independent of the display and are based purely on the data. The FDP and cloud display processes, the two final steps, are display-based.

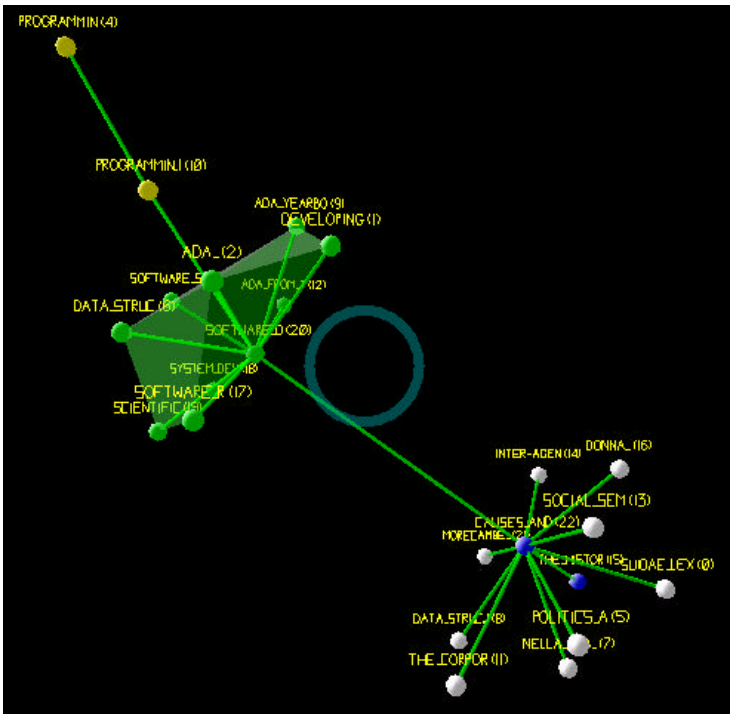


Figure 28: after hull-based clouds have been added

The initial Benediktine step however involves both data and display. The resultant display may not strictly speaking be necessary—it now serves almost as a random positioning of the points involved in the space. However, the intrinsic (appearance) mappings should still be valid. Furthermore, the generation of the ordered domains of the Q-PIT 'relation', a data-based process, is used in connection with the similarity matrix process.

Construction of Data Clouds using Convex Hulls

The construction of the convex hull of a finite point set in a low-dimensional Euclidean space is a fundamental problem in computational geometry.

The convex hull of a set of points is defined as the smallest convex polyhedron that contains a given finite set of points. Convex hulls can be calculated for an arbitrary number dimensions greater than one, but for our purposes, we are primarily interested in three dimensions.

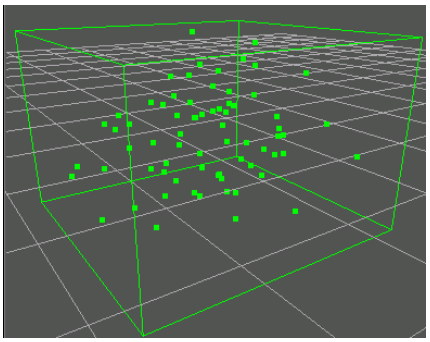


Figure 29: a set of random 3d points

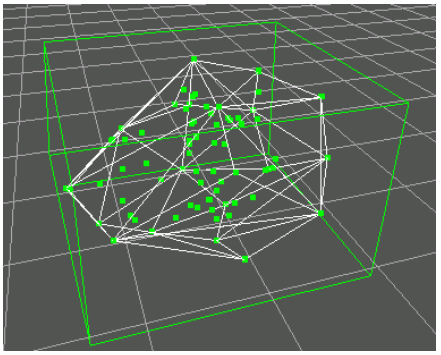


Figure 30: convex hull shown as a wireframe so that all points are visible

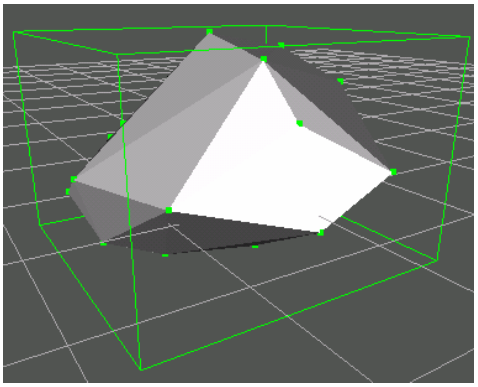


Figure 31: a polygon convex hull around the random set of points.

Figure 31 shows a convex hull as polygons. It can be seen., from this figure and the previous, that not all points are part of the hull and some are 'hidden' inside the shape. This is exactly what we need for our purpose – a simplification and grouping of a set of points or objects.

There are a number of algorithms that can be used to calculate convex hulls. For our purposes, we chose a popular and well documented approach called 'Gift Wrapping'. The algorithm, described as the Preparata and Hong algorithm in (Edselbrunner, 1997), starts with a single face and repeatedly adds a face to the edge of a previous face.

The Gift wrapping algorithm is a standard procedure for calculating convex hulls. It is one of the simplest of the many convex-hull. A basic overview of how is works is shown here:

1. Find a point that lies at an extreme of the set of points
2. Find another point close to the other which is also at some extreme. Treat this as the current 'edge' and is added to a data structure that stores edges remaining to be processed. When this list of edges is exhausted, the convex hull is done.
3. Using the current edge, reference another point (non-collinear with the edge points) and use this 'face' to calculate the plane (the same as the triangle formed by the three points).
4. Check each point in the data (excluding those currently used to calculate the plane) – if all other points lie to one side of the plane then the face being tested is valid and is added to the final shape, then the other two edges of the face are used as the current edge (goto 3) If all points do not lie to one side of the plane, this face is not valid and the other points must be checked (goto 3)

The Algorithm in more Detail

DEFINITIONS

Point/vector - data structure representing a position in 3d space (x, y, z floating point numbers)

Plane - 2d flat surface described by a point on the surface

Normal - a vector representing a direction perpendicular to a plane

Edge - a line segment connecting 2 points

Face plane - a polygon that lies on the surface of the hull. all points lie behind it.

ALGORITHM

```
Initialize: FaceList = empty, EdgeStack = empty

Find point E at some extreme (e.g. lowest y value)
Find point F e.g. second lowest y value
create edge EF from the two points

Place edge EF onto EdgeStack

while (EdgeStack is not empty)
   {
   get an edge from the EdgeStack, call this AB

   repeat
     {
     Choose a point C
```

```
    Compute cross product NM = (A - B) x (C - B)
      giving M as the plane at B with normal NM

    for every other point (apart from A, B and C)
      {
      calculate the signed distance to the plane M
        dist(P, M) = (P - A) * NM
      and find the point P with the maximum distance
      }
    }
  until (dist(P, M) <= 0) i.e. a valid face is found

  if dist(P, M) <= 0 then M is a valid face plane
    {
    add face ABC to FaceList
    add edges CA and BC to the EdgeStack
    }

  }
```

Analysis

(Borgwardt, 97) gives a probabilistic analysis of a gift wrapping algorithm on random input. It is claimed that for random input, redundant points (i.e. those which do not form part of the convex hull skin) do not need to be removed by preprocessing. It is possible that the data used to generate the cloud was not laid out in such a random fashion but e.g. in a line or a 3d cuboid shape. In these cases, it may be more efficient to represent the cloud's group with a more simple shape. Possible problems with Dive include:

- If extra data is to be added to the cloud space, the cloud must be recalculated, a cloud object remade and reloaded. Polygon objects in Dive are not dynamic i.e. vertices cannot be easily moved.
- Data removed from the cloud data only requires recalculation if that data lies on the surface of the cloud.

Chapter 4
The Java-Dive Interface

Marten Stenius and Jonathan Trevor
SICS, Lancaster University

This chapter describes the integration of the DIVE Virtual Reality system with the Java language. The Java-Dive Interface (or JDI) was the first platform designed to allow Java applications to create, change and manipulate objects maintained by the Dive system. The evolution of the JDI into JIVE the more sophisticated interface layer that underpins the Library demonstrator described in Deliverable 4.1.

The DCI

The JDI relies on the Dive Client Interface (DCI). This interface is provided by each Dive client (like the Vishnu interface) and made externally available to any application wanting to communicate with the Dive system. The DCI consists of a single socket which listens on a specified port (defined in the clients configuration file) for remote connections. Once a connection is established between the external application and Dive, plain text command strings can be sent from the remote application which are given to the Dive clients internal Tcl interpreter and executed. Any output from these commands is returned via the same connection. In effect, any external application connecting to the DCI needs to understand how to formulate Tcl commands which Dive understands (http://www.sics.se/dive/manual/tclref.html). Examples of Tcl commands which change Dive objects are shown in Figure 32.

`Dive_dir_velocity [dive_self] {0 0 0.25}`	Make the object it is written in moving forward (i.e. along its Z axis) at a speed of 0.25 m/s.
`Dive_material [dive_self] "RED_NEON_M"`	Make the object it is written in red.
`dive_move [dive_self] 0 0 1.0 LOCAL_C`	Move the object it is written in one meter forward in its local coordinate system.

Figure 32: Example Tcl commands for DIVE

Adding Java support for the DCI

The Java-Dive Interface (JDI) provides a set of Java classes that hide the socket communications to the DCI and the construction of the Dive Tcl commands from the

Java application itself. The architecture is shown in Figure 33. The JDI classes run in a separate process from the Dive client (which provides the DCI interface itself). The Java application creates a JDI connection by instantiating a special connection object in the JDI which connects to the remote Dive client.

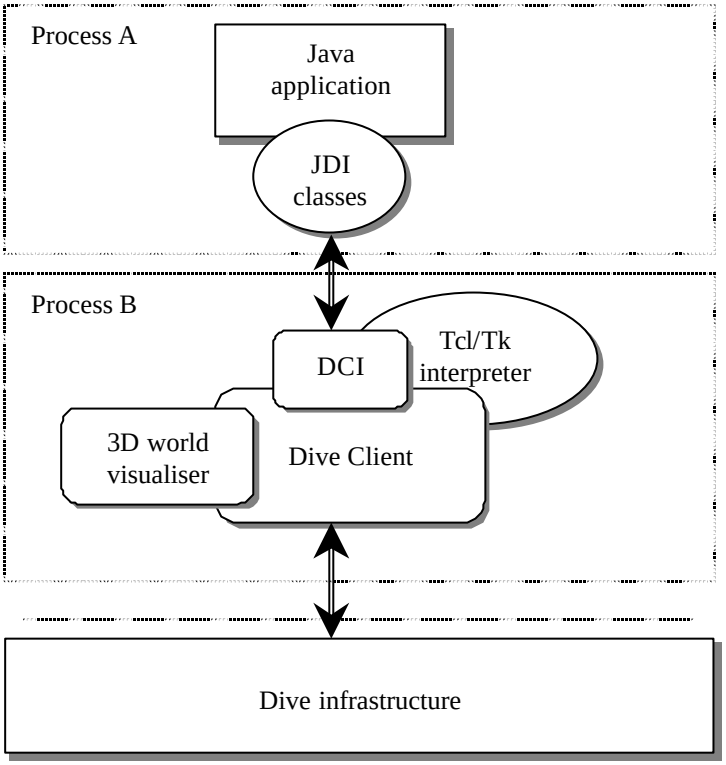


Figure 33 : The JDI-DCI architecture

The Java application performs commands on the DCI through the JDI by calling methods on instances of JDI objects. These JDI objects are organised into an object-oriented hierarchy which mimics the Dive object model, shown in Figure 34. Whenever new instances of Dive objects are detected by the JDI interface a counterpart *proxy instance* for the Dive object is created in the JDI. The Java application then affects the Dive object by invoking methods on these proxy instances. Each method is mapped down to the underlying equivalent Tcl command on the Dive object and is sent across the JDI-DCI connection and executed by the Dive client. Any return values are sent back to the JDI and re-interpreted into Java responses. For example, if a Java application asks a proxy object for its current position then the return value from Tcl is a string of three floating point values which are space separated. This string is used to construct a new 'DivePoint' Java object instance before the result is passed back to the application, which can be used in subsequent calls to other JDI objects and methods.

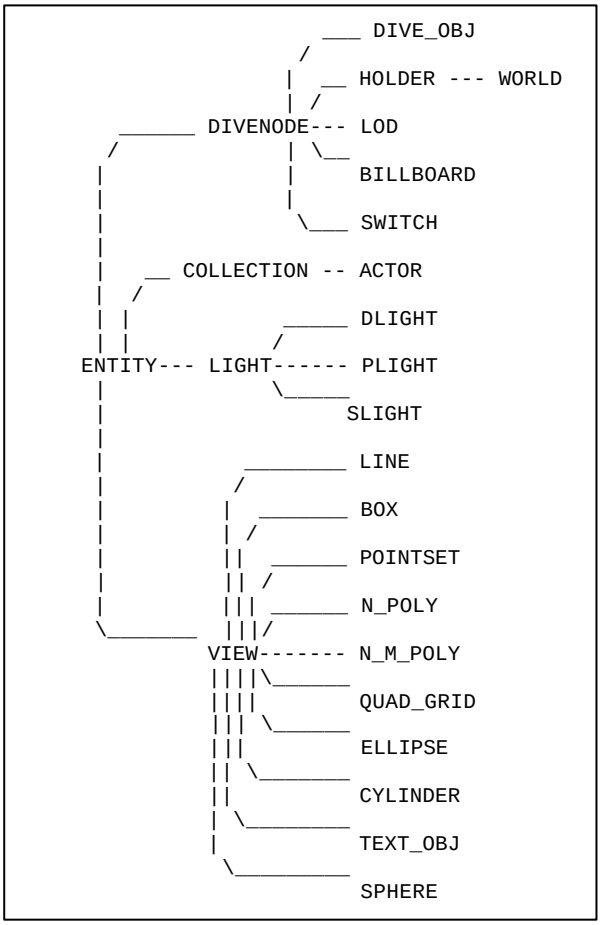


Figure 34: The Dive object hierarchy

Message and error handling

The default DCI operates by accepting a string over the DCI port, passing the string to the Tcl interpreter for execution, and returning any output from the intepreter back across the DCI. There are several consequences of this setup which need addressing for the JDI. First, any errors which occur during the processing of the message are only seen by the Tcl environment itself and are not returned by the normal DCI connection. Secondly, the DCI client has no mechanism for detecting if commands which do not normally result in any output have been executed. Finally, there is no supported form of message ordering, or even any send-reply protocol - DCI clients send messages which are executed by the Dive client and any output from these is sent to any DCI clients connected to the DCI port. The DCI client has no means of associating the incoming text (from the DCI) responses to particular previous commands that were sent.

To alleiviate these problems a simple send-reply protocol is provided by the JDI. When the JDI-DCI connection is first estiablished a special Tcl procedure

'java_execute' is sent to Dive. The procedures' code is interpreted and then can be invoked in the the same manner as any other Tcl command across the DCI. All subsequent commands sent by the JDI to the DCI are actually passed to *this* Tcl procedure for executing, rather than directly to the Tcl interpreter.

Incoming requests to the execute procedure contain a message ID, uniquely identifying the request, and the command to be executed. After executing the command, and trapping any errors, the procedure creates a response string which the DCI automatically passes back along the socket connection. Responses have three fields:

- Message ID – a simple number which identifies which request this is a response too
- Success Flag – a boolean indicating if the request executing successfully or caused a Tcl error
- Result – a text string with either the error message (if an error occurred) or the result of the request

This simple format allows the DCI client to match responses which come back from the Dive DCI port to previous requests it sent (using the Message ID) and to see any errors which occur during processing at the Dive clients side.

Executing a command through the JDI

To illustrate how the JDI and the DCI communicate consider the following example where a Java application wants to move an object in Dive forward by 3 metres. We assume that the application and the JDI has already obtained a proxy object 'obj' for the Dive object. First the move method on a Java object is invoked by the application:

```
obj.move(new DivePoint(3,0,0),"LOCAL_C");
```

The method takes the various Java objects passed as parameters (in this example an instance of DivePoint), and constructs the an equivalent Tcl command which would perform the same operation (providing its own Dive identifier for the operation):

```
dive_move 3322:122:123:232 3 0 0 LOCAL_C
```

This Tcl command is then passed to the 'DCIConnection' instance which is maintaining the link between the JDI and the remote DCI client. The connection instance allocates a new lightweight thread and a message ID for the command. A new request is constructed, containing this command, which will execute one of the special Tcl procedures which were first sent across the connection to the DCI:

```
java_execute 12123 "dive_move 3322:122:123:232 3 0 0 LOCAL_C"
```

The command is sent across the socket and is executed by the Tcl procedure. In this example no errors occur (i.e. the object exists and can be moved) and the DCI sends the output of the Tcl procedure (the formatted response message) to connected DCI clients (the JDI) (there is no value returned by the move command):

```
12123 true
```

The JDI receives and decodes the message and matches the message ID in the message against a previously sent request. It wakes the requests lightweight thread which has been waiting for the response to come back and returns the result (in this case nothing) from the original move method.

Constructing proxy objects

This is the typical sequence of actions which occur for almost all requests to Java proxy objects in the JDI. The most significant special case occurs when an object ID string is returned by the Tcl command, for example when the user creates a new object using 'readURL'. When an ID string is expected as a result from a command (such as readURL), the Java method invokes a method in the connection class to return a Java proxy instance for an object with this ID. If the object has already been 'proxied' at the JDI the connection instance will return a reference to the existing instance. If no object has been proxied yet, a new Java proxy object of the correct type is instantiated and returned.

JDI limitations

There are two main problems with the JDI. First, the performance of the JDI is only really adequate to support periodic interactions with Dive but degrades very rapidly with sustained and rapid communication - especially when the JDI has registered several frequently executed callbacks on objects in Dive (such as receiving an update whenever an object moves).

Secondly, Dive makes heavy use a C-preprocessor to analyse and interpret the text files and strings used to define Dive objects. Unfortunately neither the Tcl environment provided by the DCI nor Java itself has support for such a preprocessor which means that object definitions or commands which rely on this mechanism cannot be used over the JDI-DCI connection. As a consequence any commands or object definitions used by the JDI must be carefully checked to ensure they do not contain pre-processor directives, and where found, these directives need to be expanded by hand.

From JDI to JIVE

In this section, we will describe Jive, the Java-DIVE native interface, which has been implemented to function as a layer in the implementation of the Planetarium / Library demonstrator described in Deliverable 4.1. The motivation for implementing a Jive was shortcomings of the previously existing JDI (Java-DIVE Interface), an experimental API which was used with later versions of Q-PIT to enable Java applications to present themselves in a DIVE environment.

Having considered the JDI, and described some of its shortcomings, we continue here to discuss the prerequisites for using Java on a deeper level within the eSCAPE projects. This is followed by a short description of Java and JNI (Java Native Interface) [Liang99], and describe how this is used to realise Jive, a set of Java classes that encapsulate the core DIVE API in a package accessible by any Java programmer. Finally, some directions for the future are indicated, which includes a set of unsolved issues along with a description of some possible extensions to Jive.

The implementation of Jive

We will here give a technical discussion on the implementation of Jive, starting with an overview of possible approaches for the implementation, and a discussion on why the current method was chosen. Then an outline of the basic structure is given, with an overview of the major components, an illustration of how Jive applications interact with a DIVE world, and an introduction to how the fundamental Jive classes (the DiveNative package) are implemented. We round this section off by briefly mentioning some initial experiences of the usage and performance.

Choosing an implementation strategy

The intention when implementing Jive was to provide a means for Java processes to become full members of DIVE worlds. This means, on the network and database level, that a fully DIVE-compatible implementation is needed in the Java process. Two major approaches were possible when doing this DIVE-compatible implementation: To completely re-implement the core DIVE libraries using Java , or to use the Java Native Interface (JNI) to wrap the core DIVE native libraries in Java.

A complete reimplementation in Java would possibly be the cleanest solution, since no issues arising from collisions between different threading systems would occur, and no platform-dependent native libraries would be needed to run on a particular system configuration. However, such a solution would on the source level be decoupled from the main DIVE source tree, and any updates to the database and networking level to DIVE would have to be done twice to retain compatibility: In the C source, and in the Java source. To keep such implementations consistent in the long perspective becomes cumbersome and with it follows a high risk of platform fragmentation.

To use JNI to wrap the core DIVE libraries in Java classes introduces some issues regarding how to securely wrap the internal DIVE threading and communication mechanisms in the Java thread model, and how to do this while retaining the level of throughput in events and callbacks required by highly interactive applications. However, the fundamental requirement of full compatibility with the DIVE protocol is easily met since the underlying core libraries being wrapped are identical to those used by "standard" native DIVE applications such as the default visualiser.

Given the above, it was decided to use JNI – since the compatibility requirement along with the long-term maintenance issues was of critical importance. Hopefully, the threading and performance issues with this approach can be solved at an early stage.

Major components

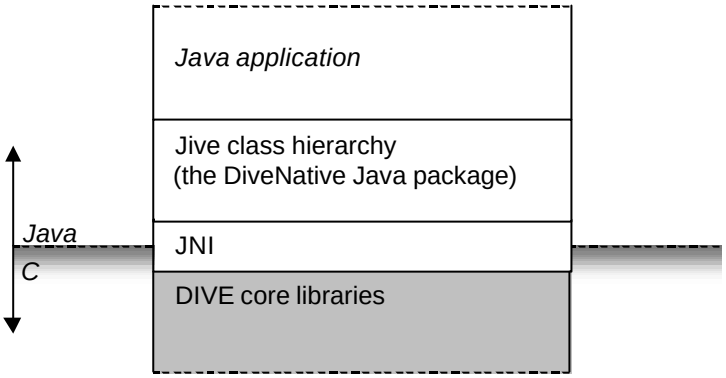


Figure 35: The fundamental layers of JIVE

Jive relies on three layers: The *DIVE core libraries* are identical to the fundamental communications and database libraries of any native DIVE process. These, implemented in C and compiled specifically for each platform, are provided with a set of stub functions defined through the *JNI* (Java Native Interface) and thus possible to call from the DiveCore Java class. Using these native Java calls, the *Jive class hierarchy* is built up, to reflect the DIVE entity class hierarchy of the native C libraries. (The DiveCore class and its relation to the other classes in the DiveNative package are discussed in a separate section below). Finally, a *Java application* can make use of the Jive classes to build shared virtual environment applications in DIVE.

Jive and the distributed database of DIVE

Jive-enabled Java processes enjoy full access to all entities present in the shared distributed database of DIVE worlds. This is achieved through a "shadowing" scheme, where the distributed DIVE object hierarchies are mirrored with proxies on the Java side, and events being translated to listener callbacks in a style reminiscent to the mechanisms supported by the Java AWT (Abstract Windowing Toolikt) classes.

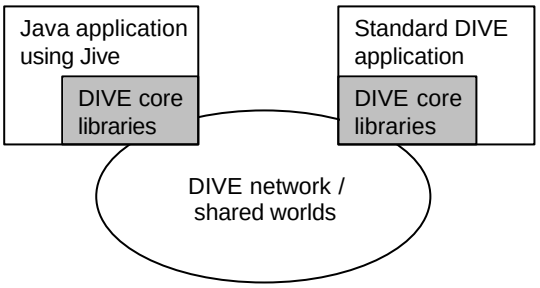


Figure 36: Jive gives a Java application full interactive access to shared DIVE worlds and any applications connecting to them.

By fully incorporating the dive database and network layers (Figure 36), a Jive process becomes a full member of any DIVE world it connects to.

This involves full support for diveserver and proxyserver connections, joining and leaving world groups and light-weight groups, and receiving and sending of state transfers and object updates and events. For a further discussions of these concepts, see for instance (Frécon, 98) or (Hagsand, 96).

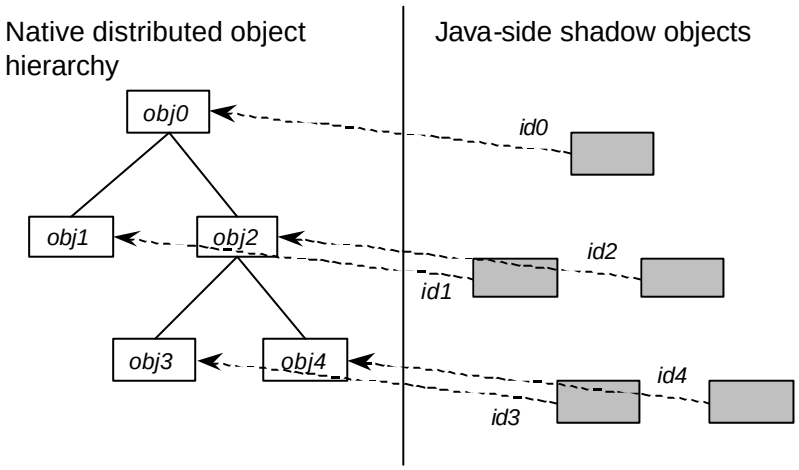


Figure 37 : The “real” objects residing in the distributed native object database are represented by “shadow” objects (or proxies) on the Java side.

Since the database objects still reside in the native layer of Jive/DIVE, and all object updates and requests over the network is handled on this level, only simple “shadow” objects exist on the Java side. To the Java/Jive user, these serve as the access point for interacting with the DIVE database, but they are in fact only “proxies” used to relay any field access or update down to the native layer (Figure 37).This means that little more than the DIVE object ID is stored in the Java shadow. Any references to the actual features of the object are directly down to native access functions through JNI.

Thus, when some feature of an object is to be modified from the Java application, the user calls a method on the Java shadow object, which is directly implemented as a native (JNI) method stub, which in turn repackages the call and furthers it to the core DIVE library. Similarly, when an update is received for an object, and an interest in such events has been registered on the Java side, the event is repackaged on the native side as a Java event object and delivered through the JNI to the Java-side registrant.

The shadow tree is continuously kept up to date with the "real" DIVE database through registering callbacks on any object additions and removals on the native level and performing the corresponding actions on the Java side. Conversely, when an object is created on the Java side, its complete DIVE structure is immediately built and distributed through the native layers. A possible modification in the future to this scheme could be to only Java-shadow those objects that actually referred to from the Java application, to reduce unnecessary object creations and overhead.

The DiveNative Java package

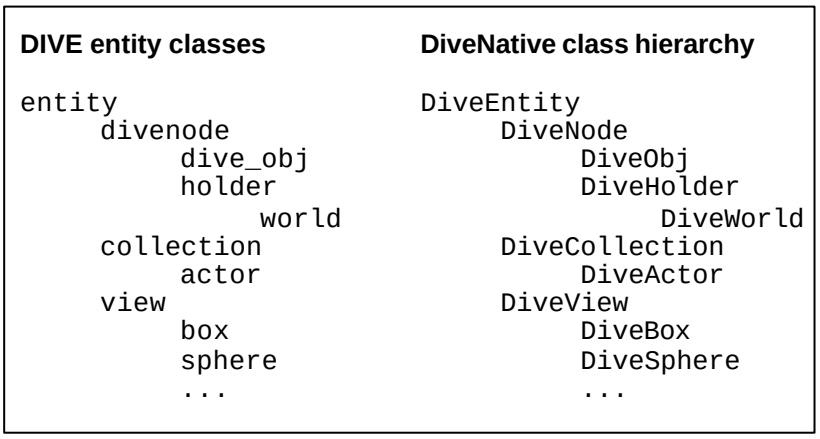


Figure 38: The DIVE entity class hierarchy has in Jive been directly mapped to a set of Java classes. (Simplified view)

The existing structure of the DIVE database is based on a "pseudo-object-oriented" approach, where object types inherit features according to the DIVE entity class hierarchy (see further the technical documentation of DIVE). This approach lends itself easily to conversion to the true object-orientation of the Java language, and this has been done in Jive. In Figure 38, we illustrate this mapping between the native DIVE entity types and corresponding Java classes.

This Java class hierarchy is mapped to underlying native DIVE calls by associating each created DiveEntity (or inheritant) to a *DiveCore* object, which should be instanciated once per session. The DiveCore class is the class that actually encapsulates all native calls, and the DiveEntity hierarchy thus has a purely semantic function, to provide a more appealing Java API than just straight mappings of the native DIVE calls.

DIVE events	DiveNative events	DiveNative listeners
ENTITY_NEW_EVENT	DiveEntityNewEvent	DiveEntityNewListener
ACTOR_MIGRATE_EVENT	DiveActorMigrateEvent	DiveActorMigrateListener
INTERACTION_EVENT	DiveInteractionEvent	DiveInteractionListener
OBJ_COORD_EVENT	DiveObjCoordEvent	DiveObjCoordListener
IMAGE_EVENT	DiveImageEvent	DiveImageListener
...	...	...

Figure 39: Mapping of the native DIVE events to Java events and listeners

The callback-based event API of the existing native DIVE has been mapped to Java by following an approach similar to the Java AWT. This means allowing the registration of *listeners* on an object, which will get called when an event occurs, which will be delivered as an event object encapsulating the details of the event (Figure 39)

As an example, an `OBJ_COORD_EVENT` is generated in the native layers for a dive_obj when it moves to a new position. This event corresponds to a `DiveObjCoordEvent` in Jive. If a Java application wants to be notified when a particular object has moved, it can thus register a `DiveObjCoordListener` with the desired `DiveObj`. The listener will then be called on a specific method and supplied with `DiveObjCoordEvent` objects as the movements occur.

To allow for registration on *all* events of a particular type, rather than just the events relating to specific objects, some events have been made available for listener registration on the "global" `DiveCore` object as well. One such events is `DiveEntityNew`, which may be interesting to receive regardless of prior knowledge of a particular object.

First experiences of use

The initial experiences from using Jive, in conjunction with interfacing the Java-based Q-PIT database application to DIVE, was that of numerous small glitches (easily fixed bugs) and some important issues that need to be solved, however not all are critical for the demonstrator to work well. Some of the non-critical issues are outlined in the next section.

A critical issue, however, was the overall performance when handling large numbers of event callbacks. This proved very slow, and the reason for this was that the native thread and communications mechanisms were protected by one single, process-wide monitor – meaning that all events lined up in the same queue with severe lags as a result. A solution for this is currently under development, it focuses on providing a per-object monitoring scheme, in combination with the ability to lock the process for critical tasks such as world connecting, state transfers, and the creation of internal DIVE threads.

Outstanding issues

While the fundamental parts of Jive has been implemented, and proved to be working – albeit with a number of performance improvements needed – some issues remain to be solved to promote a long-term general use of Jive. In this section, we outline some important issues such as how to incorporate the general set of DIVE API modules and support Tcl scripting.

Incorporation of DIVE API modules

The current implementation of Jive only incorporates the 'core' libraries of DIVE, that is, the networking and distributed database levels with the fundamental support for event handling and multi-user applications. This means the exclusion of graphics and audio rendering, and all modules.

The 'complete' DIVE package consists of a large set of modules that add higher-level functionality to the platform. Such higher-level functionality can be specific says of handling interaction, different experimental semantics for specific application settings, and so on.

This means that the current instance of Jive gives full access on the *interactive database* level to a DIVE world, which is enough to write Java applications that create different types of interactive graphical interfaces in DIVE worlds. Writing a Java-based renderer should also be possible (see below) by using different Java technologies. Doing the rendering (audio and graphics) completely on the Java side rather than through Java-wrapped DIVE classes may actually be preferred, since Java is currently gaining a large set of technologies for such presentation, in combination with generic techniques for handling input devices.

However, the lack of support for the DIVE modules results in some limitations when writing fully DIVE-capable Java applications. Of specific concern here is the method interface and the Tcl/Tk behaviour module, as well as a set of modules for import and export of various file formats for images and 3D models. Also, a rich set of interaction devices are supported through modules (even though this type of support is now moving to the use of the plugin interface instead; this will be reported in a subsequent deliverable). We will now briefly discuss some of these issues, what need to be done to incorporate such modules, and possible issues that can be foreseen to arise.

Extending the DiveNative package structure

The structure of the DiveNative package needs to be extended, and possibly reconsidered, to seamlessly and consistently support a set of dynamic modules. An obvious solution for simpler modules is to isolate them in separate classes, but many modules have more intricate relations to the core DIVE API and need other types of solutions.

The method interface module

The DIVE method interface module enables the generic attachment of scripting interpreters to DIVE objects, thus making it possible to add complex object behaviour languages without redesigning the complete DIVE core. (This is at least the intention, however the current method interface implementation is tuned to work well with the Tcl/Tk module described below, and may need some internal redesign to fully support arbitrary languages).

This module adds features to the *fundamental dive objects* themselves, and thus would require additions to the actual DiveEntity class hierarchy on the Java side. To develop some scheme that allows this type of modular additions to the DiveEntity hierarchy without the need to make module-specific changes directly to it would be desirable in this situation.

Another possibility could be to incorporate the method interface to the core DIVE libraries, since this can today be seen as a fundamental feature, and not experimental as it was when it was first implemented many years ago. Such an incorporation would then of course affect the implementation of the DiveEntity hierarchy accordingly.

The Tcl/Tk behaviour module

This module supplies Tcl/Tk behaviours for DIVE entities. Basically, each object is supplied with a Tcl interpreter, which is enriched with a set of DIVE API counterparts in Tcl. This makes it possible to quickly write complex interactive behaviours for each object. The Tcl/Tk module is designed to allow for seamless transfer of the evaluation of these scripts between the processes that take part in a DIVE world.

To allow Java processes and threads to house not only the inner workings of not only the internal DIVE threading and networking mechanisms, but also arbitrary numbers of Tcl interpreters with their own threads and widgets could, for lack of a better word, be very hairy. Nevertheless, enabling this would result in very important benefits:

1. The ability to easily control Tcl-enabled objects and processes from Java.
2. The possibility of implementing ‘strong’ DIVE processes with Java, that is, processes that can be world servers and evaluate any decentralised Tcl scripts.

To achieve this, however, the method interface must be carried over, and furthermore, any potential hurdles posed by introducing an alternative GUI mechanism into the Java process must be cleared.

Modules for interaction and devices

The Java platform currently incorporates several technologies for complex interaction devices. If a renderer is written using Java technologies, such modules may well become obsolete, or at least better implemented directly in Java as well. Such purely Java-based methods should require little modifications to the core DIVE classes,

except for making sure that the set of methods for the DiveEntity hierarchy is complete enough to allow free experimentation.

Error reporting

A fundamental addition to the interface is to transfer the error return values of the native DIVE library to the Java application. This would be done by mapping error return values of the native dive calls to the throwing of corresponding Java exceptions. A first simple DIVE exception class exists in the old JDI, and this should be further built upon when fully implementing the error reporting scheme.

DIVE configuration interface

DIVE features a configuration mechanism, which allows a set of configuration parameters to be set via a configure file or via the graphical user interface. These values can be read at startup by the module that requires a particular setting, and callbacks may also be registered to allow for run-time updating of settings.

This configuration facility is included in the Jive native DIVE classes, but only implicitly in that the configuration files are read, and the native DIVE binaries get initialised accordingly. Currently, however, no Java API is implemented in Jive to allow the reading and setting of configure variables from the Java side. This is a future extension that would be straightforward to implement, possibly in combination with a "listener-style" interface for callbacks on changing parameters.

Object ownership

Within the COVEN project, a scheme for object ownership has been implemented in DIVE: It is possible to set read and write accesses on a per-object basis, and to define a DIVE actor as an owner of a particular object. It is also possible to create groups of actors with specific permissions for an object. This scheme is not currently interfaced in Jive, all objects created with Jive get a "null" owner, and any attempted actions on objects made by a Jive process have a "null" actor as source. This default behaviour means that unless an object is explicitly protected by someone, Jive will be able to modify in and thus be able to co-exist with processes using the ownership facility. What needs to be done to fully support object ownership in Jive is basically to define a suitable API for setting and checking permissions, since the ownership functionality as such is already present in the native libraries.

Bringing DIVE to the Java world

Among the many Java technologies and APIs available, some stand out as closely related to the fundamental characteristics of the DIVE platform - collaboration in virtual environments, dynamic networked environments, highly interactive and responsive

systems. Of particular interest are Java3D and Jini, which would serve well as building blocks for new interfaces. The implications of research related to these technologies and why an integration or connection with the DIVE platform would be useful are discussed below.

A Java3D Renderer

The *Java3D* technology (http://java.sun.com/marketing/collateral/3d_api.html) is aimed at providing Java applications with a unified API for high-capacity 3D rendering. Thus, by combining Java3D and the new DiveNative package, it would be possible to build a new, completely Java-based DIVE renderer. Such a renderer would at the same time have access to the full shared VE semantics provided by the DIVE platform as it would benefit from the flexible, portable and well-structured programming Java environment. This would both benefit the DIVE platform by taking it furher and onto new grounds, as well as help leverage the Java3D standard by proving it useful and beneficial for a major, general-purpose shared virtual environment platform.

Some of the benefits for the DIVE platform, made accessible by the Java3D API, are discussed below, and include portability, extendability, and increased access to Java technologies.

Portability

Java, needless to say, is from the start intended to be platform-independent. Thus, a DIVE renderer implemented in Java would inherently be ‘as portable as the Java platform’ (given that the DiveNative classes have been ported to a given platform, of course - this goes for the Java3D API as well).

Extendability

The clean, object-oriented and modular structure of the Java language makes it simple to write applications that are easy to extend with new functionality as requirements change and new ideas need to be explored. Furthermore, specific mechanisms, such as the JavaBeans API exist to provide means for interconnecting different applications and components of applications.

Access to wide set of tools and APIs

The Java platform has gained significant ground in industry as a viable and general-purpose application development platform. Many packages and API standards have been defined and will be readily accessible to a Java3D-based DIVE renderer. Among these techologies are, just to mention a few:

- *JDBC (The Java Database Connectivity interface)* - Enables further exploration of database related research such as collaborative data visualisation and navigation in virtual environments.

- *RMI (Remote Method Invocation)* - Provides a door to new ways of distributing applications and objects.
- *Jini* - discussed in more detail in the next chapter.
- *AWT (Abstract Windowing Toolkit)* - The obvious candidate for providing the 2D GUI components of such a renderer.
- *JavaBeans* - Could be used to package DIVE features as embeddable components of other applications, or vice versa.
- *Java Sound* - MIDI, etc.
- *Java Communications API* - Enables simple control of serial and parallel interaction and data collection devices.
- *Java Advanced Imaging* - Provides tools for spohisticated image processing.
- *Java Speech* - Offers access to speech-based interaction and output.

Ad hoc CVEs using Jini

In general, *Jini* will make it easy to let arbitrary devices integrate within a CVE. With DIVE A device extended with Jini would hook into a DIVE process and through that get access of the DIVE database. These devices could range from simple mouses to complex reactive environments. We would thus see a merging of ubiquitous computational, reactive environments and CVEs.

Interaction - Presentation

During our work with CSCW-tools in general and specifically with CVE's like DIVE we have realised the importance of having a connection between physical and virtual environments. There are activities both within the physical and the virtual environment that are essential for the collaboration between people. Nothing really exists completely virtually without beeing present in the physical world. Otherwise we would never be able to perceive it. The DIVE database could be seen as a meta representation of the virtual environment and its physical proximities, i.e. those physical places where someone or something has entered the virtual space.

The connection between the physical and the virtual will be done through a number of physical devices which address different physical attributes. Some are used for visualisation, some for interaction, some for data retrieval, etc.

Visualisation and presentation

A number of different devices could be used to visualise the DIVE database. Today most of the visualisation uses 3D graphics on ordinary desktop computers but future presentation techniques can also range from sound-only to text-only.

With Jini technology the DIVE *processes* will not need to know how to render the database to present it for the user. That functionality lies within the actual *visualisation*

device rather then in the DIVE-process and is transferred to the DIVE process when the device wants to visualise the CVE.

Avatar control devices, e.g. 3D mice, joysticks and trackers

During the last decade we have seen a large number of different interaction devices for computers, especially for virtual environments. One problem related to these devices is how to communicate with them, but more important is the question on how the movements and interactions should be interpreted and mapped into to the virtual environment. With Jini, the device itself can contain the behaviours for, say, an avatar: As a minimum it could include a set of standard behaviours along with an Internet address where more extended behaviours can be downloaded.

Personal artefacts - information containers

With personal digital assistants (PDAs), people are carrying around personal information in pocket-sized computers, often fitted with different communication technologies. Today, the common ways of exchanging information with others are through connections that are explicitly set up by the users as they are needed.

With CVEs the actual virtual environment instead would act as a information container, or virtual docking station, where documents and other information can be shared or left for others to pick up. With Jini, it would be very easy for PDAs to dynamically connect to the CVE and leave and retrieve information.

A complete Java rewrite of DIVE

A possible further development of Jive could be to reimplement the classes completely in Java. This could resolve some issues arising from the combination of different threading systems (see above) as well as increase the performance, since the overhead from JNI and the complex locking mechanism could possibly be reduced.

Such a development might, however, de-couple Jini from the existing DIVE if not care is taken to ensure compatibility on the network / database level. This depends on the technology chosen for distribution.

Section Three
The technology of the physical electronic landscape

Chapter 5
Generating Virtual Cities with an Algorithmic Approximation

Eduardo Hidalgo-Parras, Steve Pettifer and Adrian West
The University of Manchester

This chapter focuses on the design of an algorithm for the generation of virtual cities. We examine in turn some urban theory as a background to the work, presenting the terms that will be used in the algorithm developed herein, the ideal form of a city generation algorithm, and consider practical variations, concluding with a presentation of an implemented algorithm and future directions for this work.

Introduction

The City Generator Project is associated with the Cityscape/Tourist Information Centre aspect of the eScape project where the intention is the creation of virtual cityscapes as social environments. Such a cityscape environment requires several components: the creation of the physical layout of the cities, the creation of avatars that live in the city, the implementation of efficient techniques for managing the environment, the insertion of various users in the environment.

Here we concentrate on the construction of the cities themselves. Assuming that e-scapes, being large scale shared virtual environments which provide connections between and integrate other shared virtual environments, will not be built 'by hand', an algorithmic approach to the design, development and maintenance is required (Bowers & Pettifer, 1998). Some of the reasons that encourage the algorithmic approximation are:

- It helps with the management of the scale. The cities are too big for the user to place all the items contained in them manually.
- It allows experimentation with multiple approximate designs. By varying the algorithm parameter values, interesting patterns can be sought in the cities.
- The efficient transmission of the virtual world. It is not necessary to transmit the entire world definition file. Only the algorithm with the parameter values that generates a certain city is necessary.

Urban Planning Theories

For a better understanding of the decisions taken in the development of the current project, it is intended to study some of the basic concepts of Urban Planning. This chapter illustrates these basic notions. Naturally, for an exhaustive understanding of the way that the cities are organised, a more complete study is necessary: however, this is not a detailed study of urbanism and we will limit ourselves to describing the only the ideas that are later used in the project.

Basic Concepts

A study of the *Cultural Origins of Settlements* is found in *Introduction to Urban Planning* (Catan, 1979). The human mind has a need to order the universe, and a manifestation of this is the ordering of the environment. All cultures have environmental ordering systems. All environments have a meaning and communicate the schema, priorities, preferences and culture of the creators. If we consider traditional cultures, there seem to be two major ordering systems. These are not mutually exclusive; in fact, they are often related. There is a geometric order, and an order related to social relationships. These two orderings systems are taken into account in the physical planning of the cities.

A good definition of *Physical Planning* is the determination of the spatial distribution of human actions and conditions to achieve predetermined goals. The key concept is the *spatial distribution*. All human actions and conditions are distributed in space: groups, cultural beliefs, buildings, vehicles and so on. Any of these variables can be defined, observed, located and translated into a map to show how they are distributed in space. But returning to the definition; what kinds of actions and conditions are spatially distributed? There are four types of variables whose spatial distribution are manipulated in physical plans: *objects*, *functions*, *activities* and *goals*.

The spatial distribution of *objects* refers to item such as buildings, parks, trees, roads, highways, sewer lines, etc… Spatially distributed objects may be as small as traffic signs and as large as airports.

The spatial distribution of *functions* is concerned with service functions provided by local government: police and fire protection, sanitation, services, utilities, transportation, education, etc… These are closely related to the distribution of objects described above.

The spatial distribution of *activities* relates to the regulatory and programming activities of urban government. Regulatory refers to those governmental activities that restrict or require specific actions, while programming refers to activities that encourage or promote specific actions. Examples of these are zoning (a city is divided into various districts) or designating an area as a historic preservation district.

The spatial distribution of goals encompasses a distribution of objects, functions and activities. Examples of this are neighbourhood improvement and economic development.

Basic Patterns

Looking at an overall view of any city, it is possible to see that these cities have common patterns that are frequently repeated. There are a lot of good books with pictures of cities where these patterns appear (Ciucci, 1979; Kostof, 1991). The four basic patterns that have been observed (Radio-concentric, rayonnant, chequered, and diamond shaped) can be seen in Figure 40.

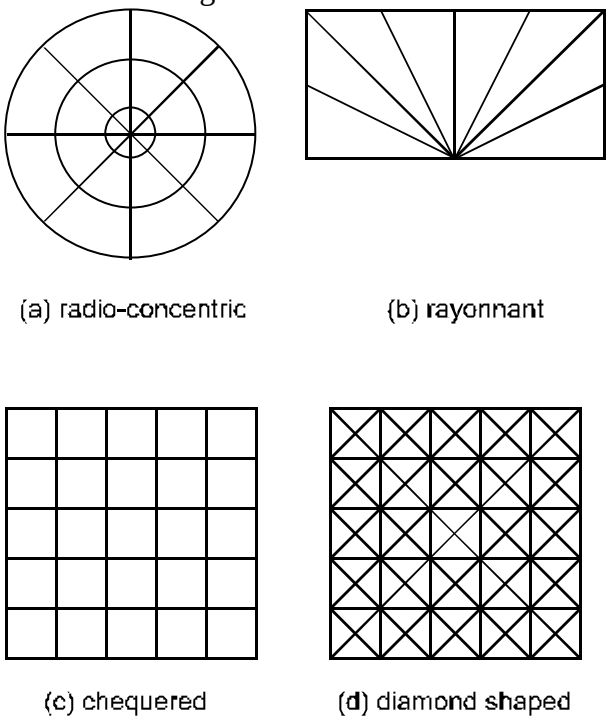


Figure 40 : Basic Urban Patterns

Central Place Theory

Large centres (cities high in population and rich in services) tend to be distant from each other in geographical space. Medium sized centres (towns with less population and range of services) tend to be more proximal to each other, with villages continuing these trends of association between size, population and range of services. In addition, services tend to be found in a centre, which would justify the cost of travel to them for the majority of their potential users. Thus, high value services tend to be found only in cities, mid-range in towns and cities, low value in villages, towns and cities. These distributions of centres and allocation of services to centres could be based on rational economic principles (Bowers & Pettifer, 1998).

Spatial Interation Models

Spatial Interaction Models in geography offer ways of quantifying the amount of interaction between centres, given information about their population and separation.

Such models can be used to optimise the spatial distribution of centres, given an expected profile of interaction between them or set of population sizes. The Spatial Interaction Models emphasise the complementarity of centres (not all centres offer exactly the same goods and services), and supply-demand relations come into existence between them. Supply-demand relations will be realised as movement and exchange in great part in relation to the friction of distance (nearest sources are preferred to more distant sources).

Towards a New Algorithm

We have introduced some concepts that will be used in the implementation of the project. The theory about urban planning that was seen previously will be useful and will guide the future decisions. Ideally, all the studied theory should be used, however as the theory and such abstract concepts are so complex have made that the implementation is only an approximation to the theory.

Some of the theory concepts have been simplified because the exact implementation of the concepts would have taken more time and the complexity would have been bigger than it is. However, the underlying theory has been taken into account in all the implemented algorithms and these algorithms generate structures that are a faithful approximation to reality.

Next, we will illustrate the creation of the generator. Firstly an ideal algorithm that includes all the theory is given. But by the very complexity of such algorithm, it has been impossible to implement it completely. Therefore, the parts of the algorithm, which have not been implemented, will be pointed out. The implemented features and the used techniques will be explained in great detail in this chapter.

The Ideal Algorithm

To develop a final algorithm, all the theory seen before will be considered. The design of the final algorithm is not an easy task because the terms introduced by the theory are too complicated for the direct application on the algorithm. Most of the terms need some kind of Artificial Intelligence techniques or the intervention of a human operator. So, remembering the theory, the goal of the city could be the economic development. However, creating a city whose distribution achieves this goal is not an easy task. It is obvious that it is impossible to solve a problem like this in a project of this duration. Nevertheless, it would be interesting to design an algorithm in such a way that future improvements can be made. The algorithm shown in Figure 41 aims to solve the problem including all the cited theory. It has been split up into five different and independent processes as this facilitates the improvement of the final result by implementing or modifying any of these processes. These processes are connected and when one of them is not implemented, the output is not taken into account by the processes that use it.

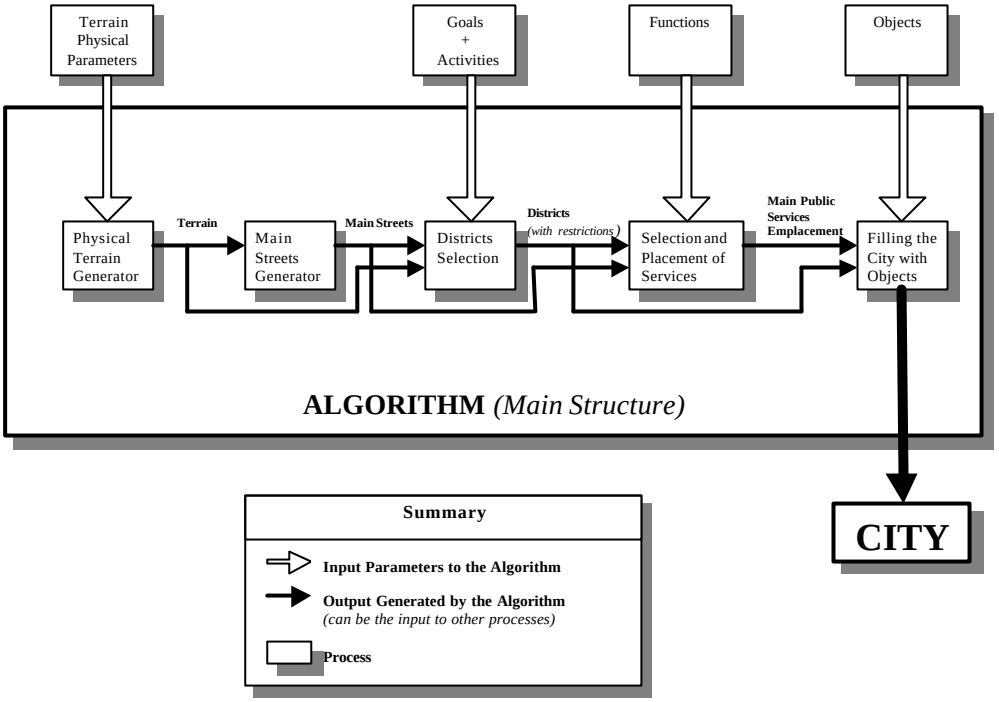


Figure 41: Structure of the City Generator

The *Physical Terrain Generator* process generates a physical environment where the city will grow up. Using a heavily parameterised process it would be possible to generate different areas with different characteristics (lakes, mountains, coasts, rivers, etc…). The generated terrain would impose restrictions and needs on the cities (buildings never can be placed over a lake, a river impose the necessity of bridges, etc…).

The *Main Streets Generator* process places the main means of communication. This is probably one of the most important processes because looking at real cities, the distribution of these means of communication are the first impression that we get of a city.

The *Districts Selection* process takes into account the goals and activities of the city. This process creates a series of districts in the cities and imposes restrictions on them. So, the selected districts can be residential, commercial or industrial districts. Possible restrictions include the maximum height of the buildings, the construction density and 'green belt' quantity.

The *Selection and Placement of Services* process locates the main public services in the city (hospitals, commercial centres, theatres, police stations, supermarkets, etc..). Therefore, it takes the services needs of the city as parameters.

The *Filling the City with Objects* process places the rest of the buildings in the city. These buildings are not important in the cities, but have aesthetic repercussions on them.

It is important to emphasise the general character of the organisation of the algorithm. With this distribution of processes, it is possible to concentrate on a specific characteristic of the algorithm without repercussions on the others. In addition,

depending on the complexity imposed on each process, it is possible to move from generating a city to trying to solve real problems in it. So, in the *Selection and Placement of Services*, if artificial intelligence techniques are used, the algorithm could seek the best placement of public services.

The Current Implementation

By the very complexity of the algorithm, it has been impossible to implement either all the features or all the processes. From the beginning, it was obvious that it would be impossible to implement all the processes. It was decided that only the most important processes would be implemented. The *Main Streets Generator* was chosen because the final appearance of the city depends on it. The *District Selection* process was selected because it gives a big variety to the distribution and appearance of the cities. Finally, the *Filling the City with Objects* was selected because its inclusion is absolutely necessary. Without it, the city would be empty, without buildings, trees, etc…

The other two processes were not implemented for obvious reasons. The *Physical Terrain Generator* was discarded because it is a very difficult process to implement and because it imposes many restrictions on the other processes. Not implementing this process, the complexity of the whole project was considerably reduced and it was possible to complete a final implementation. The *Selection and Placement of Services* process was discarded because it has no visual repercussions. It is a process which is hard to implement and it only has repercussions if the services are going to be used. Taking into account that an editor was being implementing by another student, this editor could be used to place the services manually.

The final implementation uses these three processes, but they are distributed over two programs: The Drawstreets and the Makecity programs. The first program is used to generate the means of communication and select the set of districts. The second program imposes the restrictions on the districts and generates the objects in the cities. The next sections will explain the way in which these programs have been developed

Drawing the streets

As mentioned previously, the *Main Streets Generator* is the process that has most repercussion on the final appearance of the cities. In an overview of the cities the main streets are the first recognisable features and so special care must be given to them. Unfortunately, the creation of an algorithm that generates these streets properly is quite complex. In order to generate credible structures the algorithm must have knowledge of how the main streets are distributed in real cities. This imposes the need of having some kind of knowledge on the algorithm and artificial intelligence techniques could be needed.

Nevertheless, the fact that known structures exist in the cities simplify the problem substantially. By using these structures and giving the responsibility of introducing the

streets to the user, the problem can be solved easily. If this responsibility is delegated to the user, appropriate tools must be developed to introduce the streets easily. These tools must enable the introduction of an entire city in a few minutes without much effort.

The problem has been solved using a street editor. The editor allows the creation of complex cities using primitives. These primitives are high level streets patterns such as the radio-concentric pattern, the grid pattern or the rayonnant pattern. Given that the next process of the algorithm uses the output of the editor, it is necessary to translate between the format used by the editor and the format used by the program that fills the districts. Although the editor works with complex objects, the program that fills the districts needs lists of crossroads, linear streets and districts. Moreover, when the streets are being edited, several kinds of inconsistencies can appear. Therefore, the editor needs to check the streets and correct them when necessary.

The unique streets that are needed are those that belong to a district. The rest of the streets are not useful, and so they must be deleted. The program will check and delete these streets in the simplification step. The techniques used to recognise these streets and the way used to simplify the streets will be explained later. In addition, the districts must be extracted using the street information. The algorithm developed to solve this will also be explained later.

Introducing the streets

The introduction of the streets is a vital task in the project because the final appearance of the city depends on it. So, the design and implementation of the editor has been carried out with special care. The C++ language was chosen because the object oriented methodology is very useful for this application. Several kinds of street patterns exist and it is very useful to work with all of them in the same way. The object-oriented programming facilitates the inclusion of all these patterns in a hierarchy. In addition, using these hierarchies it is very easy to add new patterns to the application. Only OOP techniques (Booch, 1996) and the features that C++ provides (Stroustrup, 1997) must be used. Adding new features to the existing patterns, it is possible to create a new pattern in just a little time.

Therefore, the classes were designed in a way that helps with the creation of new patterns. So, if someday new and more complex patterns are needed, it is possible to extend the program with little effort. The patterns and the structure of the classes can be seen in .

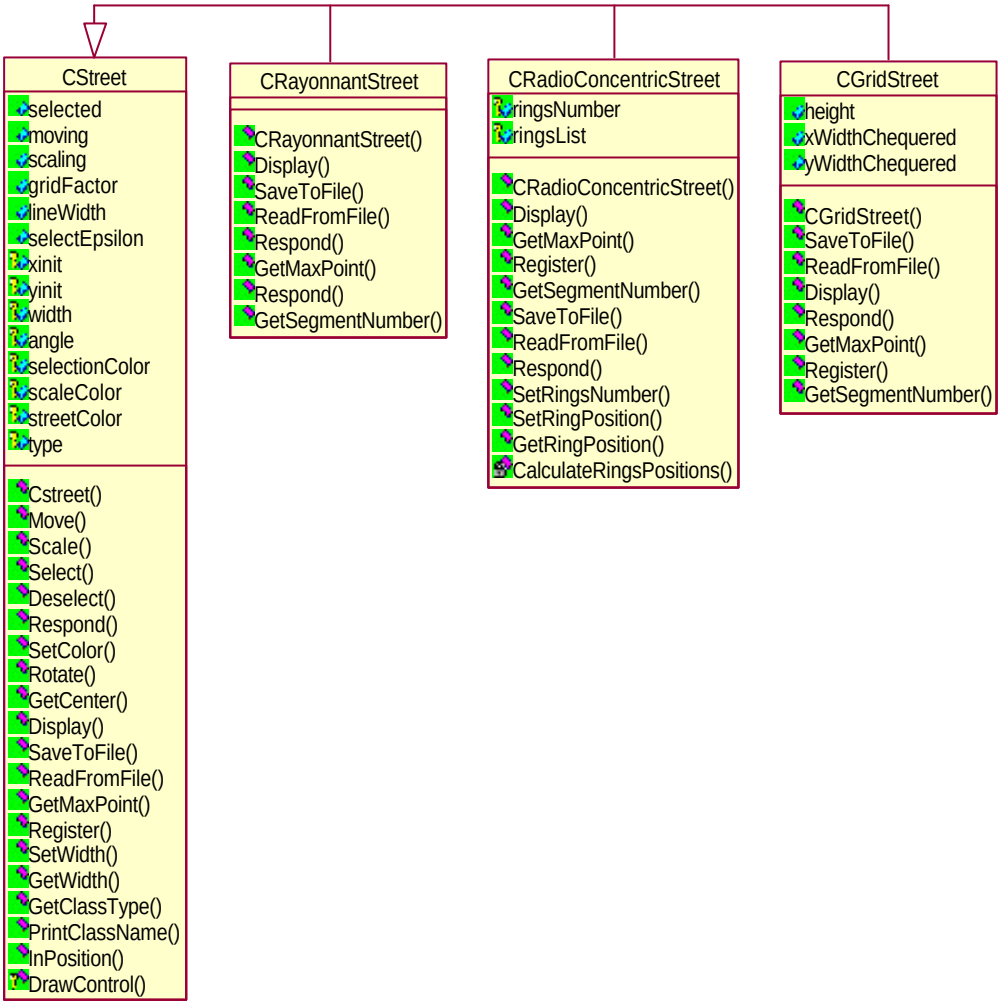


Figure 42 : UML class relationships

In the application, several patterns can overlap, leading to intersection points between them. These points will be some of the crossroads in the cities and so, it is necessary to work out these points. The way used to solve the problem avoids having to calculate whether intersections between pairs of patterns exist. This has been done using the gridcell structure. Each primitive has to register the lines that create the pattern. After that, by checking each point of the grid it is possible to know if there is a node. If two or more lines reach a given point then a node exists. But this algorithm is very expensive, $\theta(N^2)$, N being the size of the grid. The technique used to avoid this is to traverse the gridcell following the lines in a manner very similar to the way in which a graph is traversed. This algorithm is more difficult to implement but the cost is reduced to $\theta(N)$, N being the number of points in the gridcell that belong to the graph.

Simplifying the graph

To simplify the graph determined by the street patterns, two tasks are necessary. The first is the selection of the real nodes and the edges of the graph. The second is to delete the edges that do not belong to any district. These two steps in the algorithm generate a real graph that will be very useful in the future. A graph has been sought because the graphs are a very useful representation for many algorithms. There are graph theories that could be used for trying to get better algorithms.

The first step is carried out at the same time as the intersections are calculated. Choosing the final nodes is not difficult. It is only necessary to check the directions of the edges which arrive at a node. In the array that represents the grid, a byte is saved with information of the edges that arrive there. In Figure 43 it is possible to see how the edges are registered in the grid.

In this figure it is possible to see that only one bit is needed to indicate if the edge in a direction is present. As only eight directions are available, the space requirements are very small and optimised algorithms and structures can be implemented.

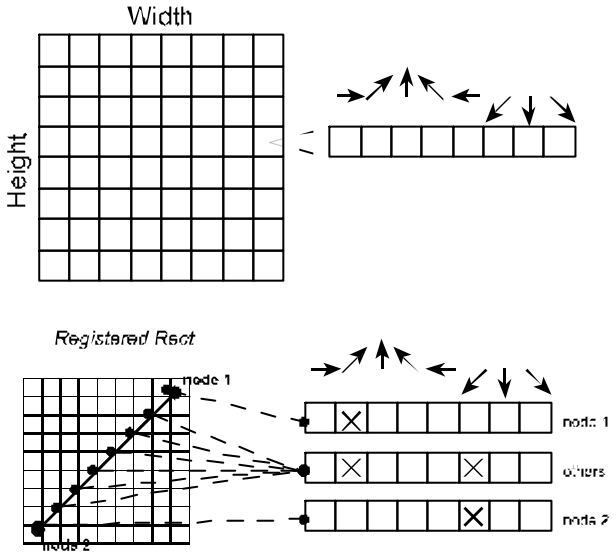


Figure 43: Way in which the streets are registered in the grid.

In the simplification, it is necessary to identify the nodes that must be deleted because they are part of an edge but are not crossroads. This is very easy to do. These nodes have grade 2 (the sum of the edges that arrive at the node plus the edges that leave the node) and have one of the following combinations of directions: North-South, East-West, Northwest-Southeast or Northeast-Southwest. Figure 44a shows some examples of nodes that must be simplified. Figure 44b shows examples of nodes that are not simplified.

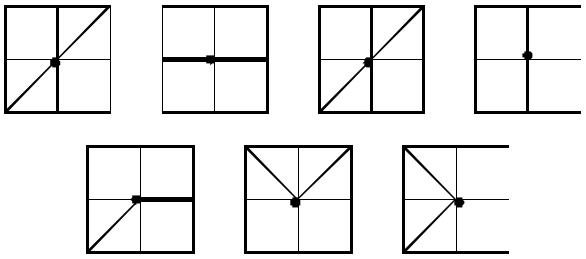


Figure 44: (a) nodes that will not be simplified, and (b) grade 2 nodes that will be simplified

After this first simplification, the final nodes are obtained. To get the edges it is only necessary to see if two nodes are connected. This is done at the same time as the graph is being simplified. Having a node that is a final crossroad, its edges are traversed following the directions. When a node that is a final crossroad is reached, the initial node and this node create a new edge. With the edges and nodes, the graph is completely determined and the next step in the simplification is carried out.

For the purpose of the project, only the edges that are part of a cycle are used. This is due to the fact that the buildings are only built inside the districts. A district must be a closed area and therefore, the areas will be represented by means of cycles. The last step in the simplification is to delete all the nodes and edges that do not belong to a cycle. This is recursively done deleting all the nodes of grade 1 until there are no nodes with grade 1. This is shown in the next figure.

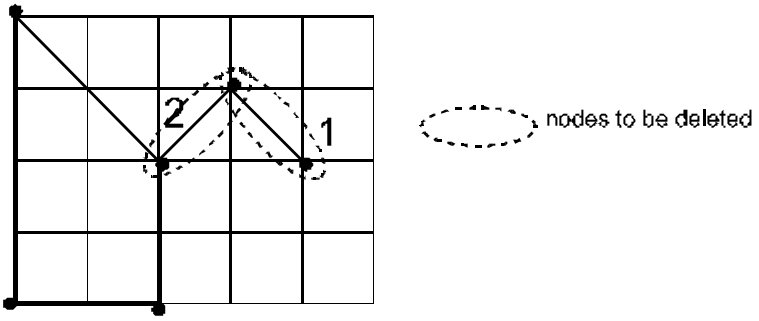


Figure 45: Nodes of grade 1 that must be deleted. The edges are deleted with the nodes

After all these steps, the graph is completely simplified and it will be used to extract useful information in it. The first task to be done is the extraction of the districts within the graph. This will be explained in the next section.

Obtaining the Districts

The task of getting the districts included within the simplified graph is the same as getting the minimum cycles in the graph. The following algorithm has been developed to obtain the information in an efficient way: the edge that joins two nodes is taken.

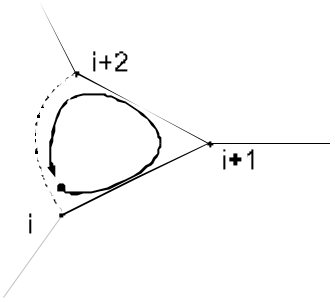


Figure 46 : Traversing a graph

Given the input edge (i, i+1), in the i+1 node the edge (i+1,i+2) is selected. The edge (i+1,i+2) is not selected in an arbitrary way. Always the edge that is the first in a given direction is always selected.

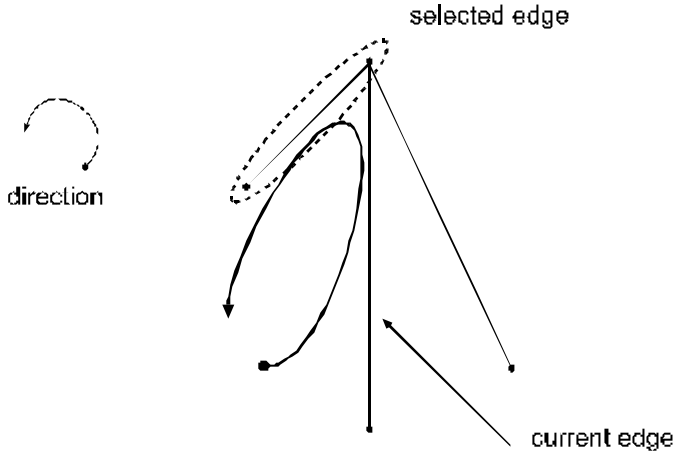


Figure 47 : an edge is selected

Using this algorithm, if a minimum cycle exists, it is always selected and it is recognised because the initial node is visited again.

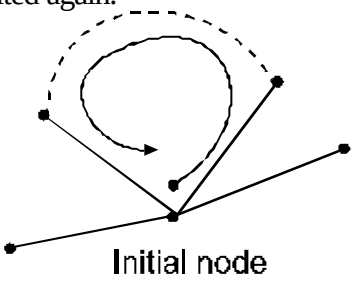


Figure 48: Recognition of a cycle by the initial node

This algorithm enables the extraction of the minimum cycle that departs from a given node d and using a given edge arrives to the initial node. If this algorithm is applied to all the edges that have the node d as the initial node, all the minimum cycles which have the node d are included.

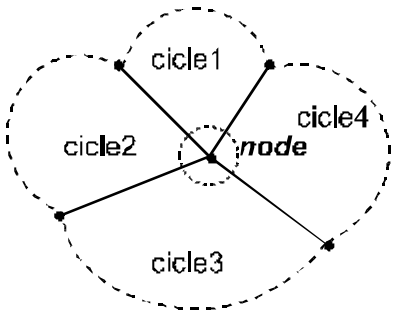


Figure 49: Extraction of all the cycles that contain a given node

Now, if a Breadth-first traversal of the graph is carried out, all the minimum cycles of the graph will be obtained. It is important to point out that repeated cycles are obtained, and these repetitions must be recognised in order to preserve the correction of the solution. In addition, using Euler's formula for the number of faces in a graph, the number of existing cycles can be found before starting the search.

$$n - m + f = 2$$

Where n= number of nodes, m= number of edges, and f= number of cycles.

This property allows the abandonment of the traversing of the graph when the number of cycles is reached. In addition, by the way in which the graph is traversed it is possible to detect if a graph is disconnected. Figure 50 shows an example of disconnected graph. These kind of graphs are not of use for the program. It is absolutely necessary that the graphs are connected and when an error situation arises, that it is detected.

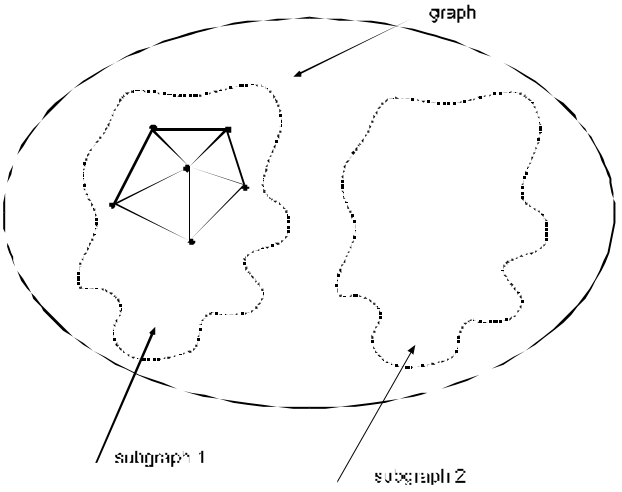


Figure 50 : A disconnected graph

When all the minimum cycles of the graph have been extracted, a new problem appears. Between all of these cycles, the external cycle is included. This cycle must be discarded because it includes to all the other cycles, and it is useless in this context. The

algorithm chosen to find this cycle consists of selecting a cycle and checking if all the nodes that do not belong to the cycle are included in the polygon represented by the cycle (see Figure 51).

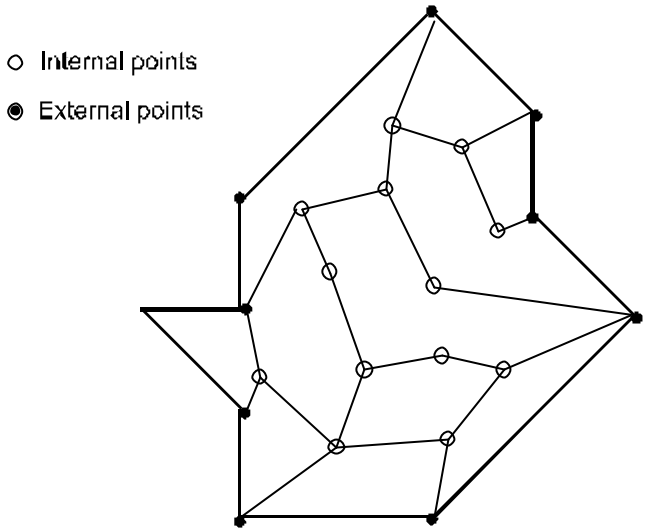


Figure 51: The external cycle

This basic algorithm will be applied to all the cycles until the external cycle is found. This is a slow process because for each cycle it is necessary to check the inclusion of the rest of nodes. The algorithm that checks if a point is included within a polygon is an expensive algorithm. So, a heuristic is used to accelerate the search. Generally, with the kind of used graphs, the cycle with the most nodes is the external cycle. But, as it is possible to see below, this is not always true.

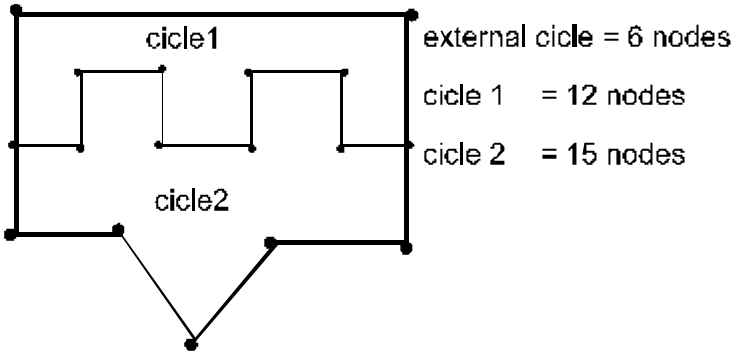


Figure 52: The external cycle not always has more nodes

In this example, the external cycle is the cycle that has fewer nodes, and is what invalidates the previous heuristic as general rule. Nevertheless, if the heuristic is used, it will guide the search of the external cycle and in the majority of the cases it will allow the desired cycle to be found as quickly as possible.

The Drawstreets program

The prototype methodology was chosen due to the little understanding of the problem. The small knowledge in techniques for developing cityscapes and the small amount of documentation available enforced the development of novel techniques. The strong probability that some of the techniques were wrong and that they invalidate the rest of the project emphasised the need to split up the implementation into independent programs. Three different programs were created for drawing the streets and obtaining the districts. A streets editor, the graph simplifier and the districts extractor. The first implementations of these programs sought to acquire some kinds of structures in order to carry out some experiments in the program which fills the district with buildings (this was the least known task and it was necessary to investigate algorithms very early). These first implementations had a lot of errors and constraints, but facilitated the developing of techniques to fill the districts with elements. With this approximation, it was possible to understand the algorithms that should be implemented. When the final ideas of the implementation were found, a final and optimised implementation was carried out. In the drawstreets program, the distribution of the programs was maintained but better algorithms were used. Finally, an optimised implementation was developed.

To make the program easy to use, all these independent programs were joined in a unique program. The final implementation is the *Drawstreets* program.

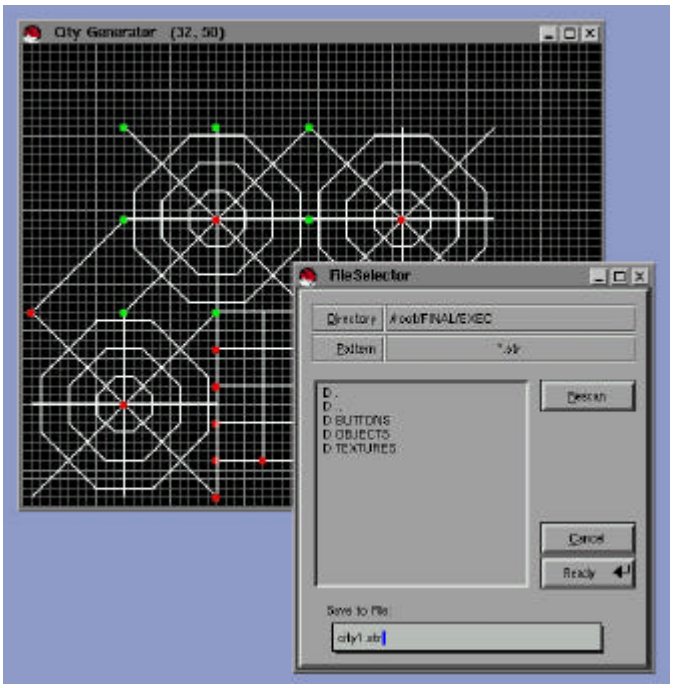


Figure 53: The Drawstreets program

This program has the common expected options: it is possible to introduce, move, scale, delete and rotate patterns. It is possible to save and load the streets. Also a

command for exporting the streets into the program that fills the districts exists. When this option is selected, the program automatically simplifies and extracts the districts included in the streets and saves them onto a file format useful for the other program. For the coding of the program the C++ language, the STL library of C++ and the OpenGL graphics library were used.

Filling the City With Objects

The next step after the insertion of the main streets and its corresponding simplification and districts extraction is the introduction of the elements of each district. The output of the program used to design the streets will be used as input for the program that implements the next step in the algorithm. This is the step that has more repercussions in the physical appearance of the city because it is the part dedicated to transforming a graph with connectivity information in a real city. This must be done by a program automatically because it is a tedious task and because the cities are too big for the user to place all the items contained in them manually. Although the computer will do the process, the user must still have control over the generated districts. So the city will be highly parameterised and the user will be able to change the parameter values to seek the preferred disposition in the cities.

Looking at the physical appearance of real cities it can be seen that different areas of the cities have different distributions of buildings, density, green areas, etc… So it is necessary that the distributions in each district are different. This characteristic increments the difficulty of the algorithm in an important manner but the results are more credible. Given this necessity of interactivity between the user and the algorithm, ease of use and an interactive interface is necessary. To write this program OpenGL, Xforms (a library for the generation of user's interfaces), C++ and STL (the standard library of C++) have been used.

The basic tasks carried out by the algorithm are the following: Firstly, it is necessary to transform the connectivity information into physical information. The edges of the input graph are the streets of the city and the nodes are crossroads. This information will be transformed into graphic information that can be used by a basic viewer. Moreover, the representation of the city must be independent of the viewer because if a change to the viewer is needed, the city file must remain unaltered. So, a high level file format that defines the city consistently but does not give graphical information will be sought. This means that graphical information such as polygons, vertices, or files with models will not be given.

At the moment the city only has the boundaries of each district, but there is nothing inside the district. So the next step is the introduction of city elements in each district. To do this buildings, roads, monuments and parks are inserted in each district. This may seem like a small number of different elements, though each of these objects can have a large number of representations, improving the appearance of the whole city.

In the following sections all these subjects are explained in more detail, and the techniques used to solve the different problems will be shown.

The Representation of the City

As it was argued in the previous section, the representation of the city must be independent of the viewer. This is necessary for various reasons, mainly because the changes to the viewer must not affect the generator. These must be independent programs and it must be possible to either introduce new algorithms to get different layouts in the districts or to change the techniques used in the generator without have to change the viewer. This was a necessity from the beginning of the project because the work process consisted of adding features to the algorithm and of using a very simple viewer to prove the results. Also the changes to the viewer must not affect the generator because the easiest way to improve the physical appearance of the cities is to create a very good viewer and leave the responsibility of giving a nice representation to the viewer. This was necessary in the project because it was impossible to create a good viewer at the beginning when the main task was the generator. It was impossible to spend too much time on the viewer. It was preferable to use this time to create a good generator and to improve the viewer at the end of the project to give a nice and convincing appearance.

Also the inclusion of the graph with the main streets is necessary in the final representation because this information can be used for quite a lot of useful tasks. So, with this information it is possible to seek short paths between districts or crossroads, study the connectivity of crossroads and any other operation in graph theory. This information is useful for navigation, to improve the generator, to establish transport lines, etc…

To achieve these objectives the information is given in three parts: the crossroads, the roads and the districts. It must be pointed out that with the roads and crossroads, the graph is implicit, and extra information is not needed.

Taking into account the underlying grid structure, the whole city is represented by tiles in a very usual way by other applications such as 2D games. The crossroads are given by their position on the grid. After that, the crossroads are represented by means of tiles. Its initial and final crossroads and the tiles between these two crossroads give the roads. Finally the districts are given by the roads and crossroads that create the district and the buildings, parks, monuments and roads inside the districts. These last four elements are given by indicating the tile where the object must be placed, the kind of the object (building, park, road or monument) and the specific height of the object if this is a building.

These are the basic ideas of the representation, but the important part is the generation of the elements in a credible way. This only can be done using appropriate techniques, as the following section will explain.

Generating the Streets

Given that the input to this program is a graph with nodes and edges, it is necessary to transform this information into a more suitable format to work out the correct tiling of the streets. The decision of only using streets with the direction being a multiple of 45º,

in addition to the simplification and facilitation of the program used to draw the streets, has facilitated the implementation of the current program. If any direction for the streets had been possible, the tiling technique would be impossible, because different kinds of tiles would have been infinite. With this approximation there are only eight possible directions (one for each cardinal direction) that can be set or not. This gives a number of different tiles of 256, still big enough to be implemented (see Figure 4.15).

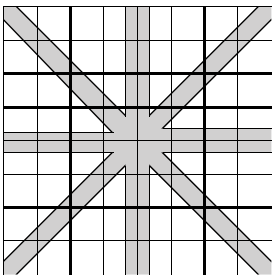


Figure 54: Format of a crossroad

The following way to solve the problem has been chosen: each crossroad has been represented by a matrix of 4x4 tiles, being a unity in the input file split in four units here. With this representation for the crossroads and a predetermined number of tiles all the combinations that can appear in a crossroad can be dealt with.

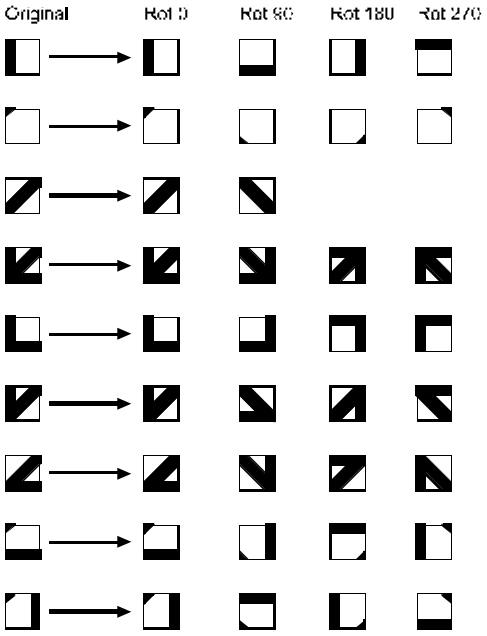


Figure 55: Tiles needed to represent the streets and crossroads

This was done because on splitting a crossroad into smaller tiles, the number of tiles needed is reduced to only 35 tiles, and by using rotations this number is finally reduced to 10 tiles (the 9 tiles shown previously and the empty tile). This is a small enough number to be used and the inclusion of files containing the 3D representation of each tile is more sensible.

So, for each crossroad the tiles that are needed are chosen and saved. Therefore, with the previous tiles, the creation of the streets is also possible. The streets can only have four different representations and the algorithm to create an optimised implementation uses this. So, the distribution of the tiles in the streets is carried out in the way shown below.

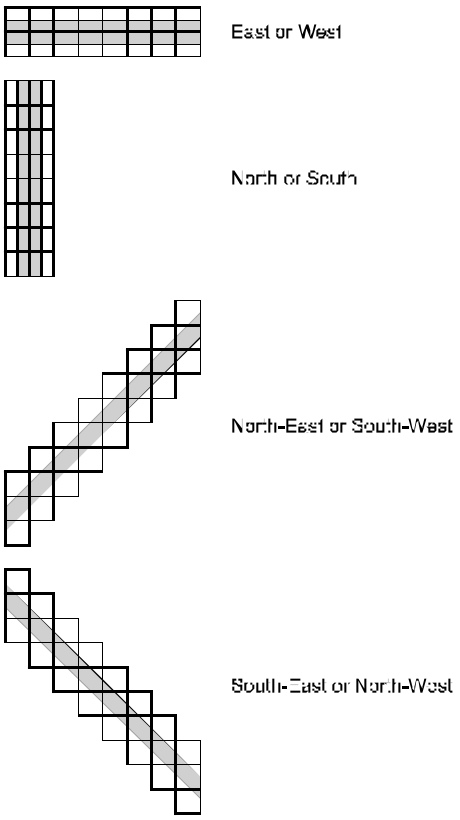


Figure 56: Different kind of distributions that can appear in the streets

Only declaring the kind of tile, but not indicating vertices, polygons or files with models, the independence between viewer and generator is achieved. The representation of the cities is not its responsibility and so the smallest description possible is given. The viewer is responsible for displaying the correct representation of each tile in the way it chooses.

Solving the Height Problem

To work out the height distributions of the districts there are different alternatives. The easier is to assign a random height to each building. This approximation is trivial to implement but the results are quite simple. A better solution must work out the height distributions taking into account *the Spatial Interaction Models* and *Central Place Theory* mentioned in a previous chapter. To do this is necessary to identify the districts that have more weight in the city. These districts will be the city centres, and the heights distribution of the city will depend of the weight of each district in the whole city. So, the districts with more weight will be taller than the others.

To do this, firstly a weight is assigned to each district and after that, depending on the weight value a height is assigned. The heights are taken from a discrete and finite set of values. This property has been taken because it is needed a finite number of buildings for the models that represent each building. This will be discussed in the viewer chapter later.

To assign the weight value, the following technique has been used. The districts with a bigger weight will be the districts that are the nearest to the rest of districts. In order to give more importance to the smallest districts, the perimeter of the districts has also been taken into account. The minimum distance between each pair of nodes of the graph that represent the city is worked out using the Floyd algorithm. Initially, the distances between each joined pair of nodes are added to an array. The maximum possible value is assigned to the nodes that are not joined (when a new distance is calculated for these nodes, it always will be smaller than the initial, overwriting it at the first opportunity). The distance between a node and itself is always zero. The recursive function that performs the calculus of the minimum distances is the following:

$$D^{k-1}(i,j) = \min\{D^{k-1}(i,j), D^{k-1}(i,k) + D^{k-1}(k,j)\}$$

Once the minimum distances is calculated between each pair of nodes, the total distance of a node ***i*** is the sum of all the elements in the row ***i***

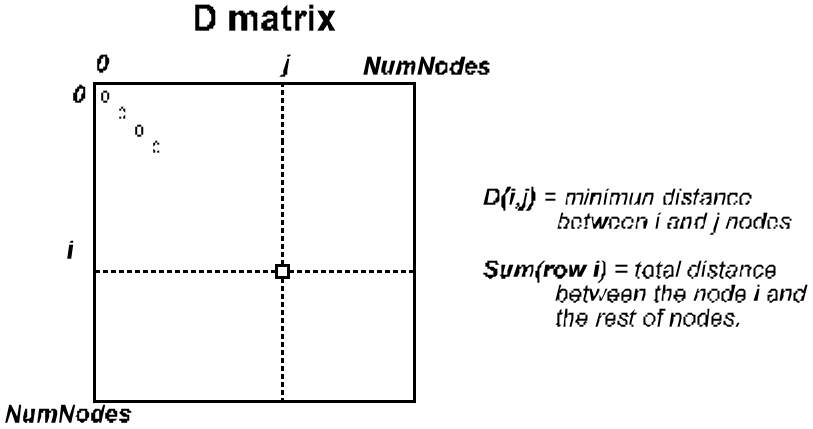


Figure 57: Representation of the *Minimum Distances Matrix*

To work out the weight value of a district, the sum of the nodes that belong to the district are divided by the perimeter of the district.

$$ValueOfNode(k) = \sum_{i=1}^{NumNodes} D(k,i)$$

Formula for calculating the value of a node

$$ValueOfDistrict(i) = \frac{\sum_{k=1}^{NumNodesInDistrict(i)} ValueOfNode(k)}{PerimeterOfDistrict}$$

Formula for calculating the value of a district

A linear mapping is applied to assign to each district a discrete and finite value given the average height of the district. To do this, a finite number of different heights are assigned, then the maximum value is given to the district width maximum weight. The value 1 is given to the district with minimum weight. Then, depending of the value of the weight, a height value is given to each district. This linear mapping can be seen in the next equation:

$$DistrictHeightValue(i) = \left\lceil \frac{(NumberOfHeights - 1) * (ValueOfDistrict(i) - MinValue)}{MaxValue - MinValue} \right\rceil + 1$$

Formula for the height of a district. The NumberOfHeights is the number of different heights that the city will have. The MaxValue and MinValue are the biggest and smallest height found in the city.

Finally, the height value is the average height value of the districts, but each district must have different values in their buildings because in other case, the results are not credible. So, a range around the height value of the district is assigned, and each district will have buildings with their height included in this range. So, being the height of the district the central value, the minimum and maximum values are calculated subtracting and adding the amplitude of the range respectively.

The Virtual City Builder (VCB)

One of the states to carry out in the algorithm is filling each of the districts in the city with buildings, parks, small roads, monuments and other stuff. In order to do this; different techniques can be used. From the previous implementation of the city generator and the study that there was made (Bowers, Murray et al, 1998), it was established that the Virtual City Builder (VCB) generated quite credible structures.

The VCB was the result of Bowers' investigations about algorithms for building virtual cityscapes (Bowers, 1995). Bowers's thinking about cityscapes was influenced by the writings of Bill Hillier and Julienne Hanson. In the *The Social Logic of Space* (Hillier, 1984), they showed, through an analysis of hamlets and small villages in the South of France, that even such simple, unplanned, 'organic' settlements manifested an 'underlying order' which subtly concentrates social encounters in some places and not others, which makes some parts of settlement more readily accessible than others, and which overall produces a configuration which aids navigability. They outlined a computer program (Hillier, 1984; p.59-61) for simulating the spatial aggregation of

buildings to make up settlements which manifest many of the properties of simple real settlements that they had identified. John Bowers in the VCB extended the ideas of this program so that it was possible to generate a greater variety of settlement forms.

VCB proceeds by repeatedly aggregating elements onto seed element. Each element consists of a closed cell joined with an open cell. VCB aggregates these elements onto a 2D surface with geometric squares as cells. Each new element added joins its open cell full facewise onto another open cell. The location of the closed cell of the element is randomly selected from those sites available which are adjacent to the element's open cell.

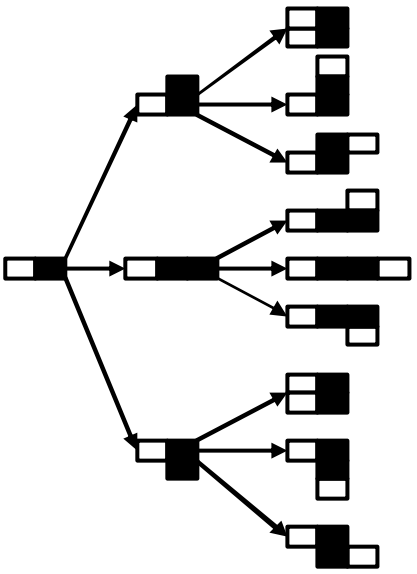


Figure 58: Possible combinations for a cell joined into an existing cell

The random selection of the closed cell location can be weighted by how many facewise neighbouring closed cells each available site has. Changing the weights from an even distribution will make VCB more or less likely to aggregate closed cells when they adjoin other closed cells. An even probability distribution tends to produce forms, which are tangled with many short winding streets and small groups of closed cells. When closed cell selection is highly weighted towards those sites which already have one closed cell neighbour, the virtual cities generated have many long streets with narrow terraces of closed cells and rarely an isolated closed cell with no neighbours.

If further constraints are introduced such as excluding sites which have only a vertex-to-vertex join to other closed cell, VCB's virtual cities tend to have a structure whose open space structure contains no rings or circuits. The open cells are connected forming a tree as it is known in graph theory.

All these features will be taken into account in the final implementation because of the variety that the cities present when they are used. Different values for the parameters of the algorithm (weights, initial number of seeds, number of cells to fill) yield different distributions. Some of them are more credible than others and it will be

interesting to search the values of the parameters that produce the best distributions. These parameters could be used as standard values for the user. In this way the user would not have to know anything about the algorithm, he just would have to indicate the kind of distribution that he wishes. Actually, the user needs to introduce the parameter values, but by simply trying different values the user can easily understand how these values must be used without understanding the underlying algorithm.

Joining the VCB with the Districts

The different shapes that the districts can have means that the inclusion of the VCB in the districts is not trivial. Now it is necessary to work with non squared shapes in a correct way but maintaining the same functionality that the VCB had before. Looking at the previous decompositions that the roads in the districts had, it's quite obvious to realise that the new algorithm must fill only the correct cells in the grid. To do this VCB fill a square that surrounds the shape that the district has. After that, only the cells that are inside the shape are preserved.

The way chosen to do this is the following. Firstly, the surrounding box of the shape is selected using its maximum and minimum values for **x** and **y** coordinates (see Figure 4.20). This surrounding box is filled with the VCB using the selected values for the algorithm. Then, a mask is created with the cells that are inside the districts. This problem is the same as that of trying to fill a polygon with pixels and so the known techniques for doing this are used in this context. The shapes that are used can be convex or nonconvex, so it is necessary that the algorithm deals with them. The problem is solved with a Filling Polygons algorithm (Foley, 1989). The crossroads and main roads of a district are given as vertex and edges for the algorithm (conceptually it is the same). The algorithm fills this shape with boolean values that indicates if the corresponding cell in a position must be used as part of the district (Figure 59). A new problem is that the roads and crossroads are continuous by nature and must be made to have discrete values. The filled shape actually has more cells than it should. To eliminate this inconsistency, the roads and crossroads are subtracted from the filled shape. After this, the shape has the correct cells and it can be used as mask for obtaining the final result.

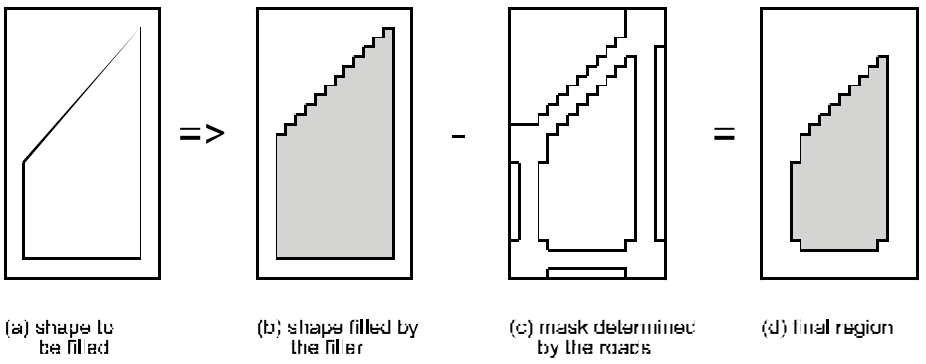


Figure 59: Joining the VCB with the districts

This mask is applied to the square worked out by the VCB and the correct cells are finally used. Now that the cells that must be used are known, it is necessary to transform these to the elements of the city. The cells returned by the VCB can be of three different kinds. They can be roads, buildings or empty cells. If a cell is an empty cell then a park is selected as a city object. If it is a building, then a random height is selected for the building in the previously calculated range. Finally, the roads are dealt with in a more tricky way. A type of road is selected for a given cell depending of the neighbouring cells. If none of the neighbouring cells is a road, then a monument is assigned to this cell. Initially, 16 different tiles are needed, but using rotations, this number is reduced to only 6 different tiles.

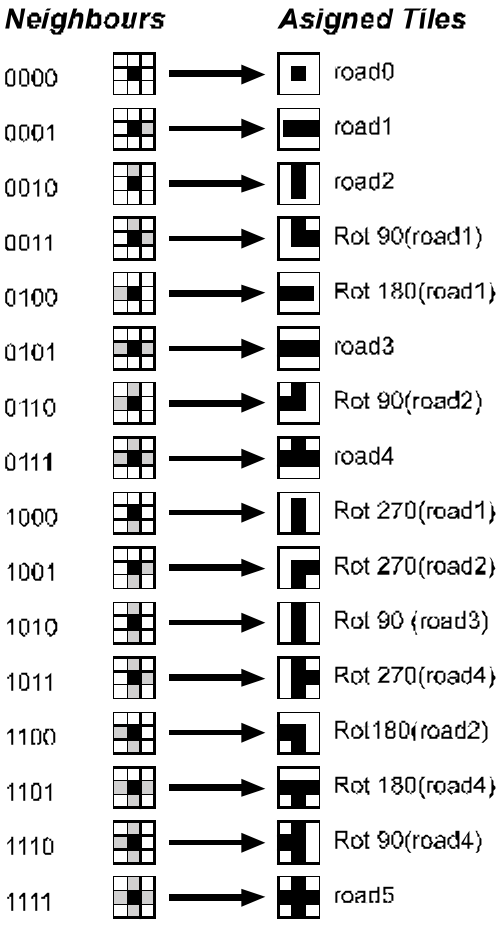


Figure 60: Calculation of tiles inside districts

All of this information is saved as elements of a district, and this process is repeated for all the districts in the city. The process of joining the VCB and the districts is quite tedious and tricky, but the quality of the obtained city makes this effort is valuable.

The Makecity Program

All the techniques discussed in the previous sections must be integrated in a unique program that carries out all of the necessary steps to generate the final cities. This program is one of the deliverables of the project and special care has been placed on it. Considering that the user needs to change the layouts of the districts in an easy way, ways to manipulate the districts easily have been implemented. A picture with the interface of the program can be seen in the following figure.

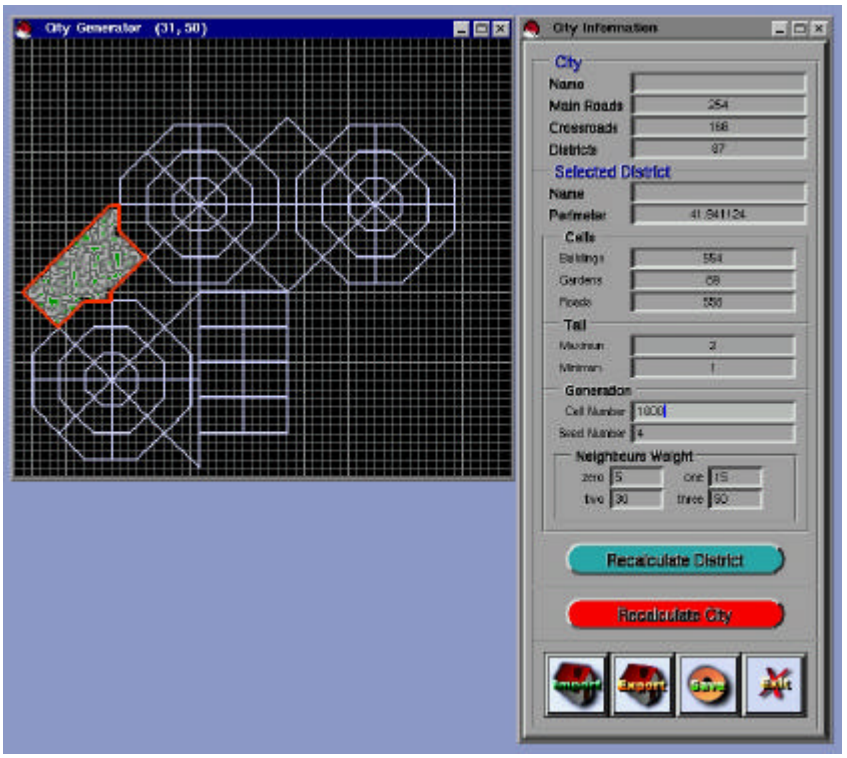


Figure 61: Interface of the Makecity program

To easily select the districts, the selection possibilities of OpenGL (Wright, 1997) have been used. By only clicking on the desired district, the district is chosen and all the available information is displayed in the panel created for such effect. In this panel the user can change all of the parameters of the selected district and see the results in real time. Also the transformation of the whole city is possible.

In order to implement the control panel, the Xforms library has been extensively used. This is a multiplatform library that allows the easy creation of complex user interfaces in short time. The library provides a graphical tool that allows the creation of the windows in a visual manner. After the windows are created the tool generates the code automatically. To associate events to the components of the windows, callback functions have been written

The program has the possibility of saving the city in a file format (this will be subsequently used by the viewer) or loading a previously saved city. The graph with the streets created in the drawstreets program can be loaded in the application using the import command. Also the option of exporting the city to anoother program is given. This is necessary because an editor for the transformation of the cities with a VR interface exists.

Discussions

Although the VCB generates quite credible structures, other algorithms could be implemented to give more variety in the different *districts*. So, it could exist districts without buildings. In these big parks some kind of *Biologically Inspired Algorithms* could be used in order to get a more chaotic distribution for the trees. The problem with the VCB and the parks is that the VCB creates quite regular structures and the green areas need more natural structures.

A problem with the present implementation can be seen in the boundaries of the districts. It seems that the main streets are disconnected from the rest of the roads inside the districts. This is because the union between these two kinds of roads has not yet been implemented. This needs new kinds of unions and the necessity of different tiles, but it is a task that must be performed.

Finally, the grid structures that exist in all the modern cities have not been implemented. This is a quite straightforward task that can be rapidly implemented. Although these structures seem unreal, they must be implemented because all of our modern cities have a lot of districts within them.

The Viewer

The main objective of the work reported here is the generation of credible physical layouts for the city. This credibility has been sought in various aspects. It is absolutely necessary that either the disposition of the elements in the city or the physical layout of the city have a real appearance. So, algorithms and tools that generated real structures were developed. It is true that a good appearance and a sense of reality is probably the main objective but this has minor utility if the contained information in the city is lost after the physical layout is worked out. Therefore, it is necessary that besides the physical disposition of the elements of the city, another kind of useful information is obtained.

By the very nature of the project the creation of a viewer is necessary. The final objective of the project eSCAPE is the creation of Virtual Reality applications where the user is able to perceive the information in a visual manner. Furthermore, for checking the correctness of the implemented algorithms, the creation of a viewer is needed. Taking into account that the viewer was a necessary task but not the most

important of the project, it was decided to implement a simple viewer that was useful for the assessment of the results.

It is necessary to note that the viewer is only useful as a tool for checking the correction of the decisions taken in the implemented algorithm. So, both the visual richness in an artistic manner and the management techniques of a virtual reality environment have been reduced to a minimum. This has allowed efforts to be concentrated on the really important parts of the project. But, as said previously, in a final application where the user will receive all the information by means of a viewer, it will be necessary to put more emphasis on it.

Architecture and Implementation of the Viewer

The previous decision to create a simple viewer must be studied in detail. This minimum comes from the fact that some of the characteristic that a viewer must have to manage the information in an efficient way will not be implemented: An example of a characteristic that a viewer should own but it has been decided not to implement is that it should have implemented some culling technique for obtaining adequate frame rates. The decision to not implement any advanced culling technique was taken because to display the layouts of the cities, it is necessary to show all the information.

For using the viewer as a checking tool, overall views of the cities were needed. This means that using culling techniques such as bounding volumes (Clar, 1976), hierarchical bounding boxes (Rubi, 1980), gridcells (Brooks, 1994) or any other technique has little use.

In a real viewer whose objective is to carry out virtual walk-throughs in the generated cities the situation would be completely different. In a viewer of this kind the user would be moving in the streets, inside the city. This means that a large quantity of the information is not visible due to occlusions, excessive distance to the camera or other reasons. In a viewer of this kind it would have been necessary to implement these techniques, as will be discussed in a subsequent section.

Another of the characteristics that has not been implemented is collision detection. Because overall views are needed, in the same way that happened with culling, collision detection is useless. This is because the images are always shown the from top where it is impossible for the camera to collide with any object. As well as not implementing these characteristics, the objects that appear in the city (buildings, trees, streets, etc…) have been simulated in the simplest possible way. This has made the usage of other techniques such as Levels of Detail (Clar, 1976) unnecessary.

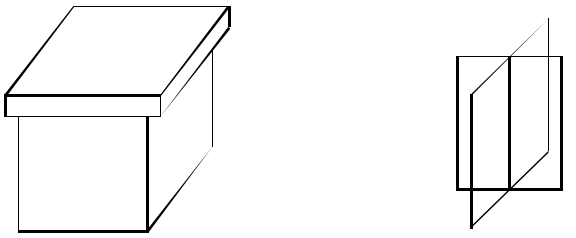
Now that the characteristics that will not be implemented have been discussed, it is necessary to see how the viewer has been designed. As mentioned in a previous chapter, there are two possibilities when the viewer and the generator are designed. One possibility is that the generator returns the basic information (streets, buildings and parks disposition) and the viewer generates the remaining information. Another possibility is that the generator returns all of the possible information and the viewer only shows this information. This last decision has been taken for different reasons: it makes the viewer simpler, because to improve the visual aspect of the city it is only

necessary to modify the viewer or because the generator has a better knowledge of the city structure for generating all the needed information.

Therefore, the viewer will only serve for showing the information created by the generator. So, with the viewer it is sought to show the generated characteristics and to demonstrate that the obtained results are realistic. Furthermore, the viewer must be useful for demonstrating that useful information is returned and not only a simple list of objects. With the viewer it will be possible to observe that the cities are composed by streets, districts, parks, etc… It always will be possible to know the buildings, parks, monuments, crossroads, streets, etc… that belong to a concrete district. This hierarchy in the information could be used for a more effective implementation of visualization algorithms, as it will be seen in a later section.

For the implementation of the viewer MAVERIK (Hubbold, 1996; Hubbold et al, 1999) has been used. MAVERIK was designed as a VR interface kernel which provides a generic framework which is easily customised for different applications (Xiao, 1997). On this occasion the viewer is programmed in C. This is because MAVERIK is programmed in C and some adapting is needed in order to use it with C++. Because the viewer is very simple, the advantages of C++ over C would hardly be noted and the difficulty of adapting MAVERIK for using with C++ large.

The viewer loads the information created by the generator and visualises it providing a basic navigation. With the information that the viewer has, it is possible to generate the visual appearance of the city. Given that it is the generator, which decides the kind of city elements, the viewer only has to show a convincing representation of these objects. This has been decided for different reasons. The first and principal is because to change the aspect of the city to have a better representation, would not need any code to be written. Actually the buildings are represented in a quite simple way (very often with a box for the walls and another one for the roof. The detail has been simulated by means of textures. With the streets and gardens the same happens. The trees are two crossed plains with texture. The monuments are the unique objects that are represented in full detail.



(a) Common building structure (b) Tree Structure

Figure 62: Common representations for buildings and trees in the viewer

These objects are modeled with AC3D objects (It is the format used by MAVERIK, but other formats such as 3Dstudio, VRML, Alias, … could be used). There exists an AC3D file for each kind of object. This is loaded by the viewer and subsequently used. This has been made because to improve the visual aspect of the city

it is only needed to change the AC3D files. This property can be used to present cities of different periods, different geographic regions, different cultures, etc…without have to change either the generator or the viewer.

The drawback of this approximation is that the viewer can not control the correct appearance of the city, because it depends of the coherence of the AC3D files. For seeing this, it is sufficient to note that when the generator works out the building height distributions, it returns the buildings with a determined type. In the type of building the height that it must have is implicit. So, if the AC3D objects that represent these objects do not represent the implicit heights in a correct way, the height distributions are lost. With the gardens (if the model represents a building instead of a park then the park is lost), monuments, streets, etc… the same thing happens. It is annoying that the viewer is not able to control the coherence of the representation, but it is supposed that the object are always going to be correct, and the advantages of this solution are bigger than the drawbacks.

Another problem with the cities is that they are large in size. So, care must be taken with the objects that form the city. A master object is used for the representation (if it is not done in this way, the memory needs would be enormous). The possibilities that the MAVERIK kernel brings have been used for this purpose. Although it has not got functions for doing this directly, it allows the access to the structures, being this a way to achieve the purposes.

The MAVERIK MAV_SMS structures have been widely used to visualise the information. As was previously pointed out, the viewer allows selecting the kind of information to visualise. Once some kind of information is selected (all the buildings of a district for example), all the information of this kind is shown. This means that special structures for searching information in an efficient way are not needed. Furthermore, because there is no collision detection, culling or object selection no special structures are needed. So, the lineal structures that MAVERIK provides are used (Cook 1998a; Cook 1998b).

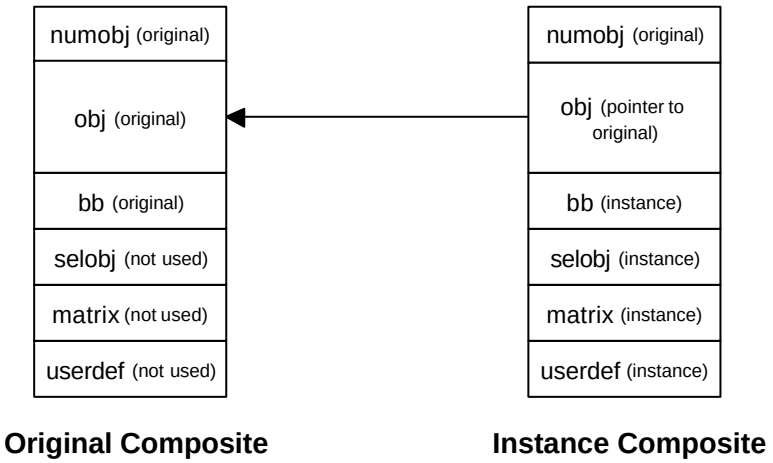


Figure 63: Saving space using MAVERIK

The information is organised in the following way; there are linear structures in the city for saving the crossroads and streets. Moreover, there is a list of districts; in each district independent linear structures for buildings, monuments, parks and ground are saved. Furthermore there are lists with pointers to the crossroads and streets that form a district.

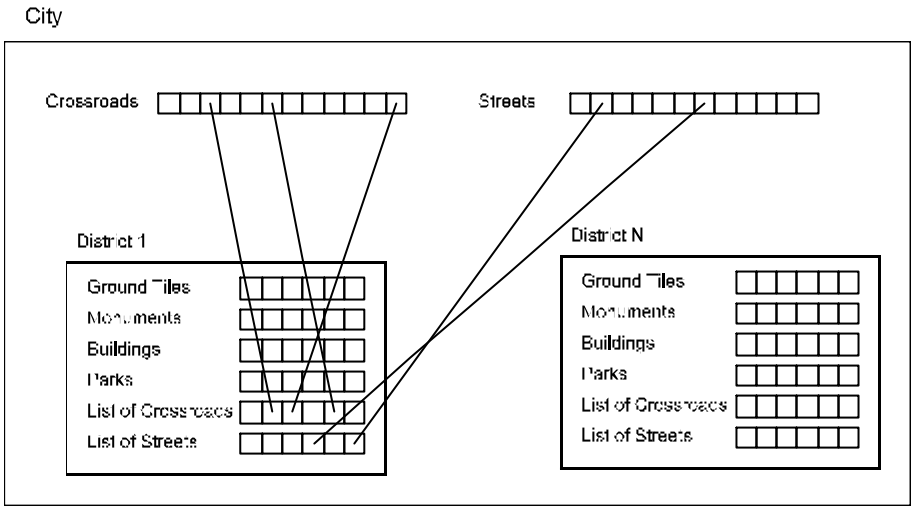


Figure 64: City Structure in the viewer (main structures)

This structuring of the information allows visualisation of the information without any search cost. If it is only needed to show the buildings of a concrete district it is sufficient with only displaying the buildings linear structure of the selected district. If all the elements would have been inserted in an unique SMS structure, to display a determined information it would be necessary to traverse all the structure checking for the current element to carry out the visualization condition. This means that the presentation of the information were slower and with lesser frame rates.

As was pointed out before, the fact that the generator creates the information and the viewer only shows it has made the viewer quite simple. Looking at the viewer source code it can be seen that an important quantity of the code has the objective of load the information from the input file created by the generator. Really, this would have been changed completely if the necessary characteristic had been implemented (culling, collision detection, levels of detail, etc…).

Due to the characteristics previously mentioned, the frame rates of the viewer when all the information is shown is quite low. For a small city, it can be about 2000 buildings and 4000 ground tiles. With this enormous quantity of information, it is impossible to get more than 4 or 5 frames per second in a machine with Pentium II and Voodoo II graphics card. Fortunately, the viewer allows the selection of the kind of information to be shown. So, if it is wanted to change the point of view of the city, it is sufficient to decrease the quantity of information, move the camera and display all the information again.

If it is sought to do a virtual walk-through by the streets of a medium size district, the frame rates in the mentioned machine are quite good. Taking into account the fact

that there is no culling technique and when the camera is inside a district the rest of the districts are normally not visible, this probably means that on applying culling techniques, the movement inside these big cities will be possible at high frame rates.

Discussions

As discussed before, the viewer is only a means to see the correction of the rest of the project. Keeping in mind this objective, the implementation has been kept as simple as possible. A better implementation must include the following characteristics:

The implementation of efficient culling techniques is needed. Looking at the kind of information displayed and the hierarchy in the information, it is possible to take advantage of this. So, the districts could be used as a quick way to select the information to display. It would be useful to check the intersection of a concrete district with the camera fustrum. So, if the district is inside the camera fustrum or intersects with it, the district should be displayed and then chosen for a refinement afterwards. In a later state, techniques such as occlusion and levels of detail could be used. The building representations should be more detailed. This would imply the necessity of using occlusions to only send to the graphics pipeline the objects that are visible. In this application, normally the buildings occlude quite a lot of information, meaning that the use of occlusion is very useful. Moreover, the use of levels of detail could increment the frame rate. Only displaying the objects in the first plane of the camera with full detail, and depending on the distance to the viewer reducing the detail, it is possible to decrease the number of polygons submitted to the render with the subsequent performance improvement.

The underlying grid structure of the cities could be used as a culling technique (see Figure 65). It could be used to select only the cells that are inside the view fustrum and apply the occlusion and level of detail techniques to these cells. Probably this could imply a better performance but it has not been tested to see how or whether the approximation is correct.

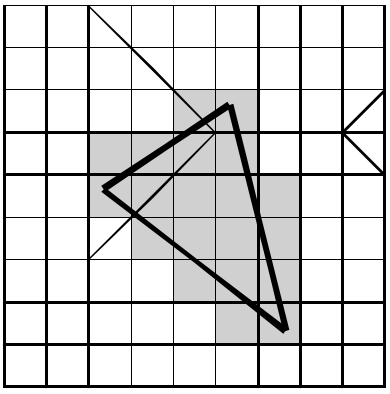
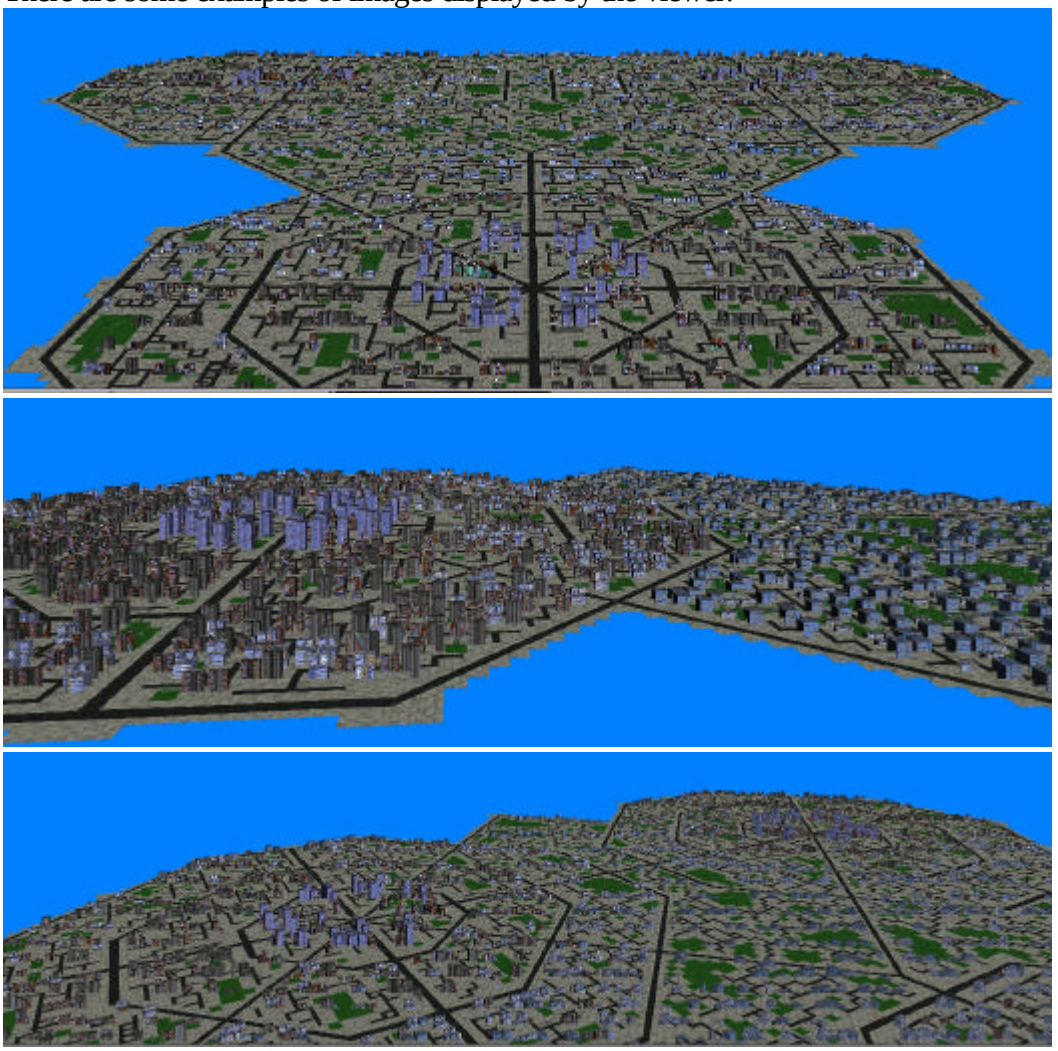


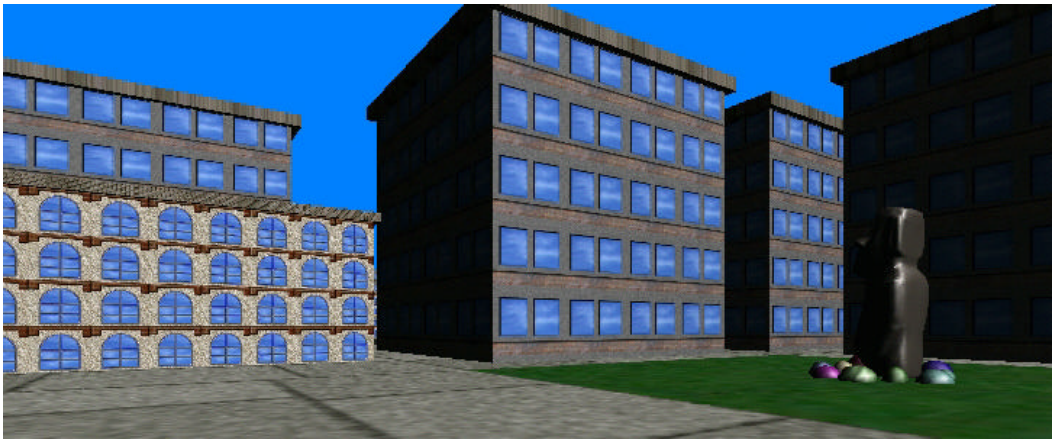
Figure 65: Culling technique. The grey cells represent the cells that must be displayed. The thin lines are the boundaries of the districts. The triangle represents the camera fustrum.

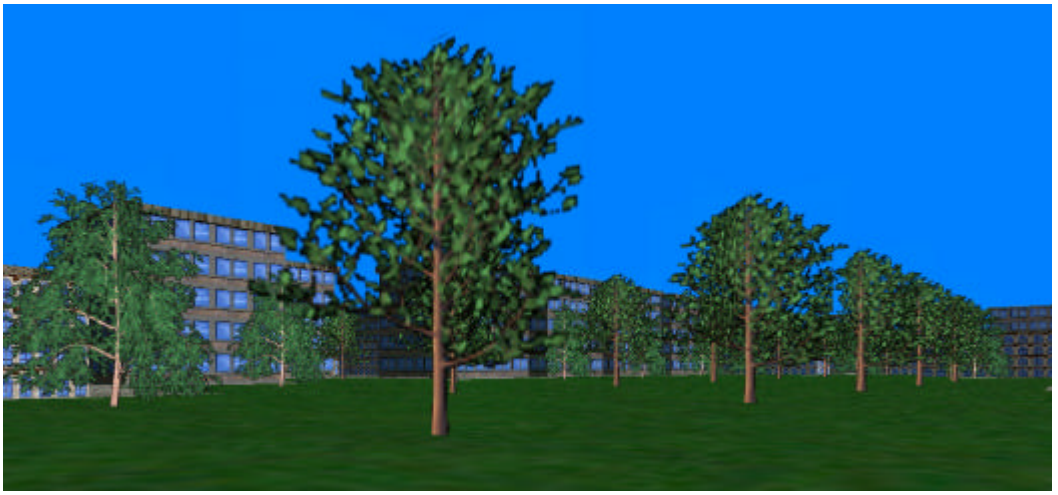
Finally, the implementation of aid techniques to the navigation is compulsory. The possibility of passing across the buildings or other objects is quite disturbing and it must be avoided. The inclusion of collision detection is needed to improve the navigation. Furthermore, novel techniques for navigation such as Forced Field Guided Walk-Through (Xiao, 1997) could be used. In it, the user's movements are guided by a force field, which assists the user to avoid obstacles during navigation.

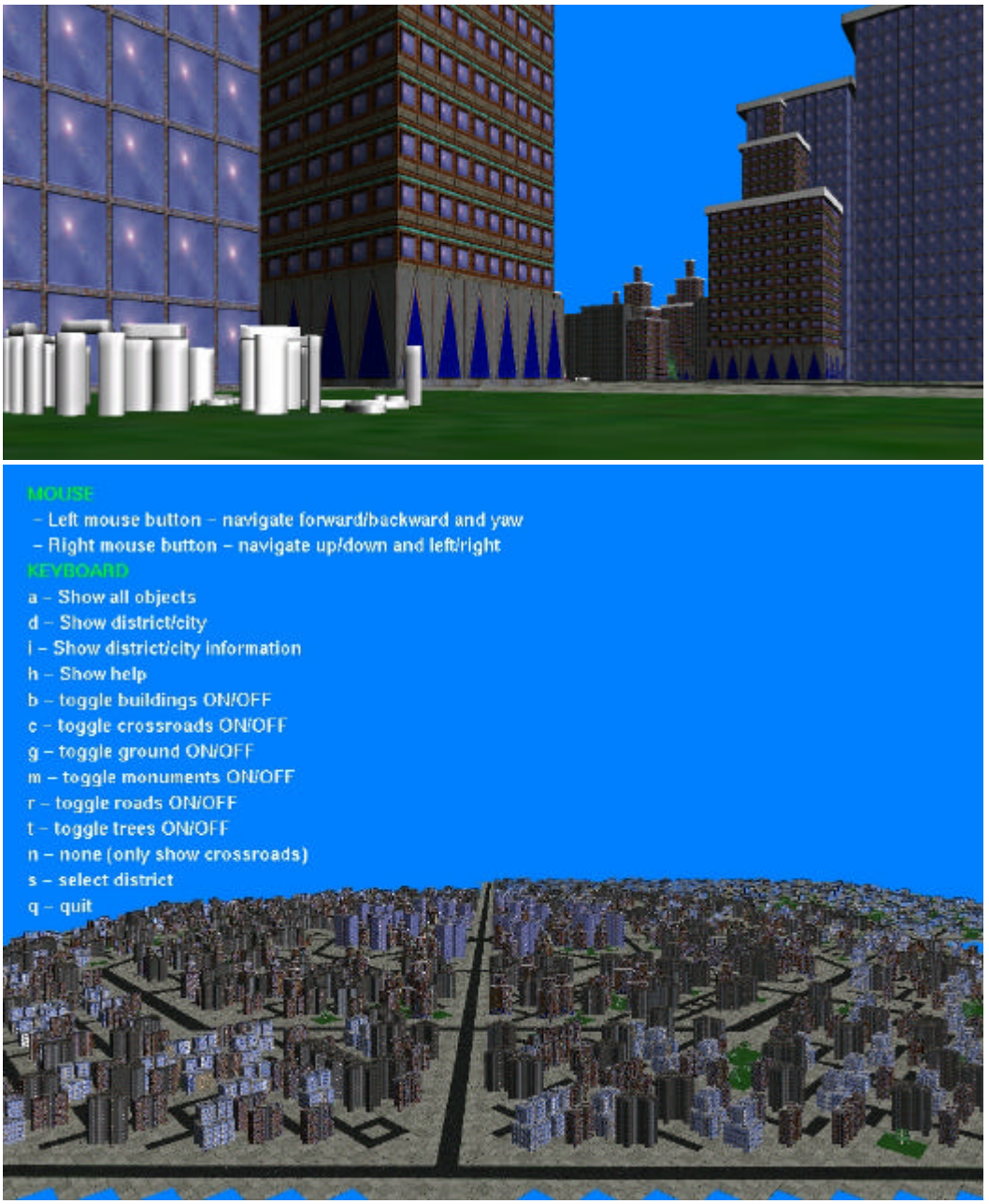
Images

There are some examples of images displayed by the viewer:









Future Work

As discussed throughout this report, the given solution needs further work to give a better approximation to reality. The actual implementation obtains quite good results, and the quality of the generated cities encourages the development of the algorithm. Summarising, the areas that need more work are the following:

- It is necessary to include a *Terrain Generator* because the inclusion of geographic properties (rivers, mountains, etc…) will give to the cities greater variety.
- Other algorithms must be included in the Filling the City with Objects process.
- The placement of services must be implemented.

- The viewer needs the use of culling techniques in order to deal with all the generated information.

.

Chapter 6
Wayfinding in the virtual cityscape : Professor Dijkstra goes walkabout

Ahmed Rahali and Roger Hubbold
The University of Manchester

This chapter describes work done to provide an efficient way of navigation and transportation around virtual cityscapes. In a number of applications, the problem of determining the, in some sense, optimal path occurs. This could be anything ranging from finding the fastest path in a network to determining the safest path for a robotic craft wandering upon the surface of Mars.In the same context, this project deals in particular with finding optimal routes in a virtual cityscape. The metaphor of a city provides its visitors with the ability to navigate around, walking along streets and across open spaces. The need for an *efficient* way of navigation and transportation around the city was evident. Our virtual tour-guide is named Professor Dijkstra after the originator of the driving algorithms.

Problem Definition

The aim here is to design and implement a solution for Dijkstra, the city guide, which he can use as an efficient way to autonomously determine the fastest route to a given destination in the city. The immediate practical problem instance that this project sets out to solve is the following:

Given the virtual cityscape as developed in, find the best way to travel between two points or even a series of points. Obstacles along the way should of course be avoided in an efficient manner whenever possible.

Project Goals and Requirements

The initial aim of this work was the research into the development of a virtual city guide in order to move around Dijkstra's city as efficiently as possible. The idea behind this was to use some sort of algorithm to determine the shortest (least-cost) path between two different nodes in a graph structure. A study into graph theory was initiated, and different algorithms that address this problem were analysed. Different ways of representing the city as a graph were studied.

Dijkstra's city is a complex virtual environment. The need to transform this complexity into a smaller, simpler and manageable structure was apparent. So, as a first step, an investigation into means of annotating the city was crucial in order to build the needed information. This information was then represented as a virtual map that is

simpler in structure, but semantically more informative to the user as it describes the city as a whole in a less obscure way. A map being also viewed as a graph could, then, easily be manipulated using different traditional graph algorithms.

Dijkstra's city and the virtual map describing it, being two separate environments called for the need to establish a link between the two environments to provide a way of communicating actions and/or changes in either. To make the application user interactive, the virtual map forms a platform that provides a means of conducting some user operations. It also serves as a way of animating all actions performed in response to user requests or alterations.

This project as proposed had a large scope for imagination that allowed many ideas to evolve during the course of work. First of all, it was clear that a city visitor may want to wander around the city independently. When following a particular route, one must not be restricted to the pre-planned destination and should be allowed to change their plans as desired. Consequently, the solution should give the user full control and freedom. On top of that, some city visitors might want to drive around, whereas others might prefer to walk. So, the system should cater for the user's choice by providing a driver's map as well as a pedestrian's map each computing different paths and their associated costs. Moreover, a more interesting solution would permit the user to interactively introduce obstructions and one way streets. These effectively cut off links between certain points in the city, making some optimal paths temporarily unavailable. As a result, new methods to find optimal alternatives incrementally needed to be developed, in order to avoid an extremely costly re-evaluation of the shortest paths in the altered graph structure.

Such an algorithmic problem are bounded by a number of factors. Firstly, it should be time and memory space efficient. Secondly, It must exhibit accuracy and reliability so that the optimal path will always be found. Last but not least, it should be flexible and general purpose, so that it can be adjusted and applied to different instances that fall in the problem scope as defined earlier.

All-pairs Shortest Path Problem

A graph $G=(V, E)$ comprises a set V of N vertices, and a set $E \acute{\imath} V \acute{} V$ of edges connecting vertices in V. Each of the edges is associated with a weight that represents the cost of getting from source to destination. In a directed graph, each edge also has a direction. A graph can be represented as an adjacency matrix A in which each element $A[i, j]$ represents the weight of the edge from node i to node j.

A path is a sequence of edges from E in which no vertex appears more than once. The shortest path between two vertices in a graph is the path that has the least cost. The single-source shortest-path problem requires that we find the shortest path from a single vertex to all other vertices in a graph. The all-pairs shortest path problem requires that we find all shortest paths for all possible pairs of vertices in a graph. The following sections describe two different approaches that address this problem.

Dijkstra's Algorithm

Dijkstra's single-source shortest path greedy algorithm computes all shortest paths to travel from a given vertex in a graph to every other vertex. The algorithm maintains a set *T* of vertices not yet visited and a list *D* of shortest distances using only nodes already visited as intermediates. At each stage a vertex *v* from *T* which has the shortest value in *D* is chosen, and *D* gets updated using:

D [W]= min (D [w], D [v] + A [v, w]). For each w in T.

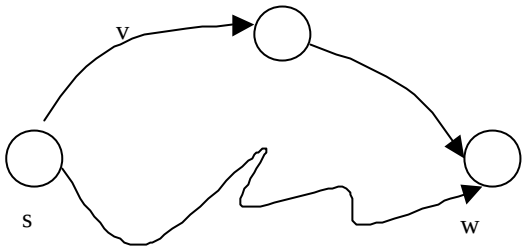


Figure 66: The comparison operation performed in Dijkstra's single-source shortest-path algorithm. The best-known path from the source vertex s to vertex w is compared with the path that leads from s to m and v then to w.

Floyd-Warshall's Algorithm

Floyd-Warshall's all-pairs shortest path dynamic algorithm computes all shortest paths to travel from any given vertex on a graph to every other vertex.

Define the function *D (k, i, j)* as the shortest distance from *i* to *j* using only nodes from 1 to *k* as intermediate points. If the given weights of edges are all *A [i, j]* entries in the graph's adjacency matrix then *D (0, i, j) = A [i, j]* describes direct paths with no intermediate nodes.

The basic idea here is that if we want to find the shortest path from *i* to *j*, using only nodes 1 to k as intermediates, two possibilities can be distinguished:

- The path does not actually use the node k, in which case we only consider the use of nodes 1 to (*k*-1) as intermediates and the cost of the path is just:

  D (k-1, i, j).

- The path does indeed use the node k, in which case the path from i to j passes through node k and therefore has the following cost:

D (k-1,i , k) + d(k-1, k, j).

- So the lease cost path is then given by the formula

D (k, i, j) = minimum (D (k-1, i, j), D (k-1, i, k) + D (k-1, k, j))

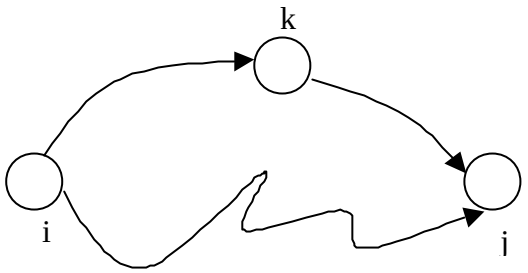


Figure 67: The fundamental operation in Floyd's sequential shortest-path algorithm; Determine whether a path going from i to j via k is shorter than the best-known path from i to j

Floyd-Warshall's All-pairs Shortest-Path Algorithm derives a matrix *S* containing the best-known shortest distance between each pair of nodes, in *N* steps, constructing at each step *k* an intermediate matrix *L (k)*. Initially, each *S (i, j)* is set to the length of the edge from *i* to *j* if the edge exists, and to *INF* otherwise. The k^{th} step of the algorithm considers each *k* in turn and determines whether the best-known path from *i* to *j* is longer than the combined lengths of the best-known paths from *i* to *k* and from *k* to *j*. If so, the entry *S (i, j)* is updated to reflect the shorter path.

Performance and Complexity Analysis

Suppose we apply both algorithms to a given graph with *N* nodes.

With Dijkstra's approach, choosing *v* from *T* requires all the elements in *T* to be examined, so we look at *N-1, N-2,..., 2* values of *D* on successive iterations, giving a total time of $O(N^2)$. The inner loop for updating *D* for each *w* in *T* does *N-2, N-3,..., 1* iterations for a total also in $O(N^2)$. The time required by a single-source version of this algorithm is therefore in $O(N^2)$. An all-pairs algorithm executes Dijkstra's greedy algorithm *N* times, once for each vertex. This involves $O(N^3)$ comparisons.

Using Floyd's approach, we perform a single comparison for every destination *j*, for every given source *i*, and every possible intermediate node *k*. From a set of *N* nodes, there are *N* possibilities for choosing *j*, *N-1* possibilities for choosing *i* and *N-2* possibilities of picking an intermediate vertex *k*. So, it is evident from the three nested loops that the number of comparisons needed is in $O(N^3)$.

It seems like both algorithms have similar complexity $O(N^3)$. However previous studies show that Dijkstra's algorithm is slightly more expensive than Floyd's dynamic

technique. In fact, if the cost of a single Floyd comparison is t, Floyd's Algorithm performs a total of $t\ N^3$ comparisons, whereas Dijksrta's Algorithm involves $Ft\ N^3$ comparisons, F being a constant. Empirical studies show that $F@1.6$; that is Dijkstra's Algorithm is slightly more expensive than Floyd's Algorithm.

From the analysis above, Floyd's Algorithm seems a good choice to effectively solve the all-pairs shortest path problem.

The City Representation

This chapter analyses techniques of annotating Dijkstra's City in order to build the information needed by Floyd's Algorithm. We first consider means by which semantics are added to some parts of the city and then move on to examine closely ways of identifying all connected points in the city in order to represent it as a graph structure.

City Annotation

Dijkstra's City could only be viewed as a collection of virtual objects, organised according to the layout generated by the Virtual City Builder Algorithm, introduced previously. These objects in the virtual scene are all constructed, using MAVERIK, from a small number of primitive graphical items such as lines and polygons. They are then mapped to different textures in order to make the world look more realistic.

Various buildings in the city are unnamed. To make city feel real besides looking real, buildings should be classified and named to enable the user to feel their existence. They are also located independent of each other, each within a single grid cell and they should, therefore, be located using a global co-ordinate system to make their location relative to each other apparent to the user.

One way of classifying and naming buildings is to examine their randomly set sizes together with their textures. Since they are rendered as MAVERIK objects their geometrical properties and texture mappings could easily be retrieved. First of all, we need to locate building cells within the city 2D grid, using a simple sequential scan operation. Once the position of a particular building is known, we look for the object that represents it in the virtual environment.

Building objects are tested for size and texture and are classified accordingly using different boundary values as thresholds. At the other end we keep a database of some possible building names that exist in a real city. The database is split into a number of classes, one for every building category. Each of these contains a collection of relatively related names that are then randomly assigned to various buildings from the relevant class.

For instance, a large building could be a university, a shopping centre, etc. One of a moderate size could be a coach station, a club, etc. One of a small size could be a corner shop, a post office, etc. One of height equals to zero must be a green space.

City Graph Construction

Buildings in the city have now been identified, located and their semantic information has been stored. However, we still do not know how different parts of the city are linked to each other. Since buildings are defined independently each within single grid blocks, the information we have so far is still not adequate to detect links between different points. Streets, however, can be used to keep track of those links, because a street cell depends on its neighbouring cells. It represents a continuation of at least one of its four neighbours. Hence, the street skeleton does indeed show how different parts are connected to one another. In this section we shall examine how we use this idea to extract the relevant information in order to build a graph of all connected points in Dijkstra's city.

Node and Edge Identification

One efficient approach to carry out this task is top down. In other words, we make use of the grid-like structure of the city platform. Firstly, we consider each street block independently and identify all points (nodes) within it, and links between them (edges). Then, we merge the collection of nodes and edges together into one single graph.

A simple sequential scan over the grid can determine all the blocks where a piece of street resides. A typical street block is described as a collection of special points and lines linking some of them to some others, as shown on figure 4.2 below.

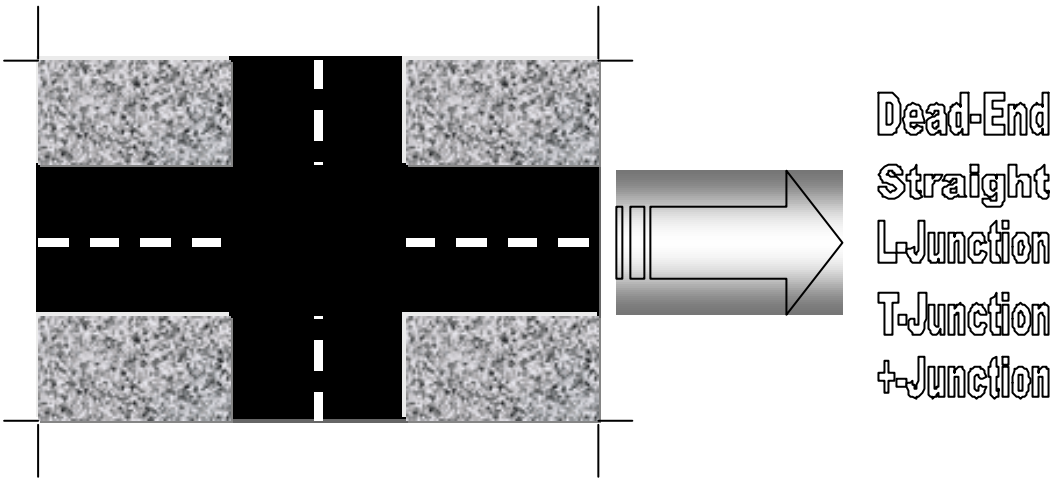


Figure 68: A street grid cell and its possible classes.

As can be seen from the figure above, the piece of street that lies within a single grid cell can be any of the listed 5 classes (each class has its own subclasses). The class can be determined by examining the neighbourhood of the current street cell. For instance if only one of the 4 neighbour cells describes a building or a wasteland, then it must be a T-junction that we are dealing with, and so on. The subclass is then determined by looking at the position of neighbouring building cells relative to the street cells: north,

south, east and west. For example, an L-junction would have the following subclasses: ⌊, ⌈, ⌉ and ⌋.

The geometry of each chunk of street within a particular cell is described in a maximum of 12 points as figure 5 shows. The idea is to make use of these already calculated points and the street class. So, the class of the street tells us how many points to consider; 4 in a dead end case and all 12 in the case of +-junction. The subclass then, determines the exact points to be chosen. A node is created for each of those picked points, and the suitable edges are also added, as Figure 69 shows.

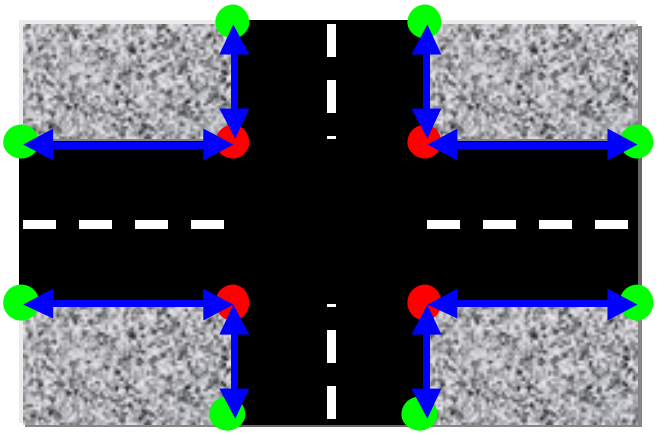


Figure 69: Idea I on street cell node and edge generation.

The nodes in red are corner nodes, and they are not taken into account unless they describe a change in direction (a corner edge). The nodes in green are intermediate nodes. These are of a greater importance because they provide a means of linking one street cell to its adjacent street cells. Each of the edges, in blue, is associated with a cost determined as being the distance between the two points it links.

It should be noted that whilst it would be possible to build a graph of all connected points using this technique, the graph's nodes and edges neither lie along streets nor along pavements, but on the separating edges. This cannot be considered as a solution for building a pavement graph and a road graph at the same time, because simply it is not giving the required level of precision for each one.

Pavement Graph

To build a pavement graph a similar approach is used, with two enhancements made. For each of the 12 points we define a counterpart that lies on the pavement. The co-ordinates of these are obtained by simple shifting and scaling operations as shown on figure 6. On top of that, the previous method generated many more nodes and edges than sufficiently needed.

In fact with a slight optimisation, we could achieve a less dense graph with fewer nodes and edges. It should be noted that some of the corner nodes are only there to trace the link between two intermediate nodes, and therefore can be omitted if we provide a direct link between those two intermediate nodes. This helps to cut down on the number of nodes as well as edges, resulting in a rather simpler structure.

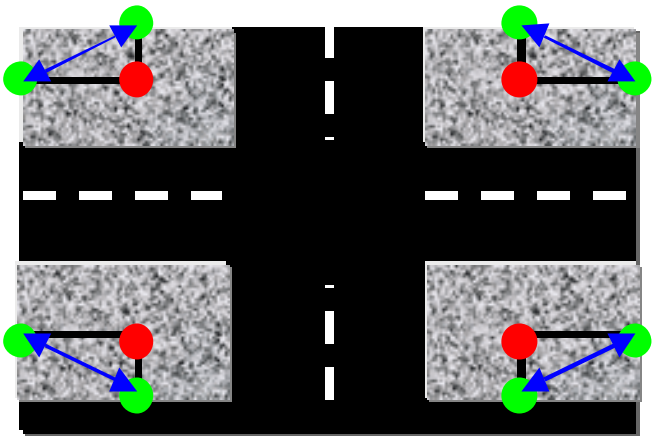


Figure 70: Idea II on pavement node and edge generation.

Road Graph

In this case a maximum of five points is needed for each street cell. These points are calculated along the middle of roads and depending on the class and subclass of the street cell the right ones are chosen together with the relevant set of edges. This is shown in Figure 71 below.

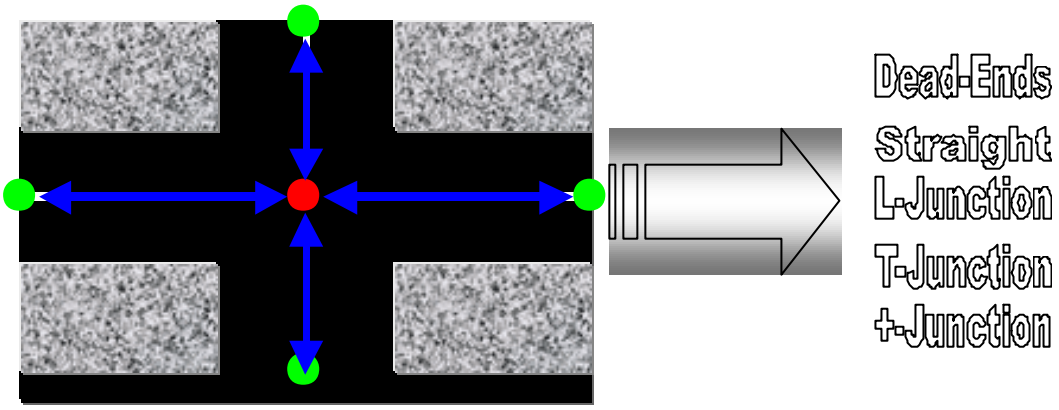


Figure 71: Idea III, Road node and edge generation.

Revision Implementation

Design Conclusion

In searching into ways of annotating the city and providing a simpler representation of it, a fundamental tenet of the design was to try focusing on the structure of the problem and how it can be broken down into its smaller, similar and simpler constituent parts. The city has a grid-like structure that forces itself to be considered, so that a task to be carried on the whole platform can be resolved within each grid cell separately. Results are then grouped into one unique solution.

Implementation and Results

The process of building the city graph (map) involved four major tasks: analysing the underlying structure of the city, extracting the relevant information, adding basic semantics where needed and summarising this information into a new database represented as a new simpler 3D environment using MAVERIK.

The city grid is scanned horizontally and vertically to locate and classify street and building cells. It is quite simple to retrieve the geometrical description of the city parts generated within each cell as they are MAVERIK objects. For instance, the x, y and z spans of a building, which determine its size, could be retrieved and could therefore be used to classify and name it. In addition, the object's matrix, which eventually describes the object's co-ordinates within its cell, determines the object's position in the whole world once translated into the world co-ordinates system.

The relevant Information is stored as two separate sections: one for 'pavement' graph and another for 'road' graph. Each section contains an Edge-database and a Node-database. The Edge-database holds data about each edge which includes the co-ordinates of its source, the co-ordinates of its destination and the cost represented as a distance between the two. The Node-database is subdivided into two classes, building nodes and street nodes. Each node entry in either keeps data about the co-ordinates of the node. If it is a street node the type of street cell it was created in is also stored. If it is a building node its name is held on top.

Using MAVERIK's primitive objects such as boxes and polylines, it is possible to represent each of the two graphs (maps) as separate environments from the city. However, it should be emphasised that although they are separate environments, the graph is a tightly linked to the city. i.e. every point in the graph has its counterpart in the city. The pavement graph and the road graph of a city of a moderate size are presented in

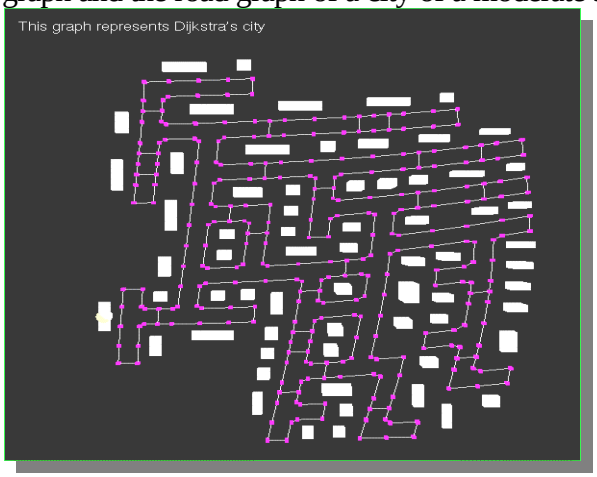


Figure 72 and Figure 73 respectively.

The pavement graph and the road graph provide the necessary information for Floyd's Algorithm to look for optimal paths. In addition, they serve as maps that describe the overall city layout to the user. On top of that, they provide the user with the ability to query what different buildings represent and to get their position relative to the global structure of the city. They also detect all connected points in the city and

make these links globally visible to the user. Finally, they provide platforms for interaction between the user and the cityscape.

Figure 72: A pavement map (graph) of Dijkstra's city

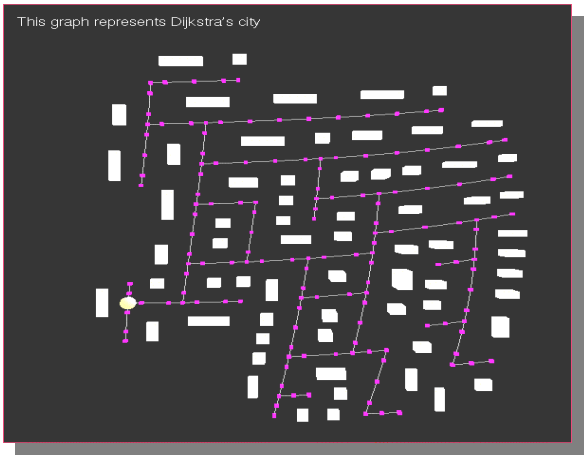


Figure 73: A road map (graph) of Dijkstra's city.

Virtual City Guide 1

Basic Operations

The city virtual map built in previous not only does it give an overall description of Dijkstra's city and all its connected parts that Floyd's Algorithm needs, but also it constitutes the basic platform for any user interactive operation. In the following sections we look specifically at finding optimal paths in the city by applying Floyd's Algorithm to the city graph. We also discuss ways to interact with the virtual map in order to input user requests and animate any output.

Finding the Optimal Path

Floyd's Algorithm provides the shortest distance between all of the nodes in terms of a table (new adjacency matrix). However, using that procedure provides no information on the specific pathway that gives rise to these values. In order to construct the shortest pathway a predecessor matrix (P) is required to be calculated at each major cycle (corresponding to k in section 3). This method can be implemented in $O(N^3)$ time so not to alter the complexity order of the algorithm. The algorithm in section 3 can be extended by the following function:

Initially:

P (0, i, j) = NIL, if there is no direct edge from i to j.

P (0, i, j) = i, if there is a direct edge from i to j.

Then we keep track of the last node visited (predecessor) using the following recursive definitions:

P (k, i, j) = P (k-1, i, j) if D (k-1, i, j) < D (k-1, i, k) + D (k-1, k, j)

P (k, i, j) = P (k-1, k, j) if D (k-1, i, j) > D (k-1, i, k) + D (k-1, k, j)

The database of nodes and edges with their associated costs is the source of information that Floyd's Algorithm accesses and manipulates to find for each pair of nodes both the distance corresponding to the shortest path as well as the path itself. The main issue here is rather performance related. Depending on the city size, the number of nodes and edges generated could be very large, and so could be the time required to run Floyd's Algorithm on them. Once the graph is constructed for any particular city, as Floyd's Algorithm solves the problem for every possible pair of nodes it only needs to be called one time. The results obtained can then be stored in a new database that can be consulted in a constant access time every time a path between two points is requested. This is performed for both road and pavement graph options.

Moving Between Two Different Points

Once the source and the destination are known, the shortest path as well as its corresponding cost can be retrieved from the stored path database. Each path is specified in terms of a series of nodes to be visited in the order given and a cost that corresponds to the distance to be travelled from source to destination. The next step is to visualise the results obtained on the virtual map and identify where the pathway lies in the cityscape.

In the virtual map Nodes are represented as MAVERIK objects and can therefore be queried for any information associated with them. Source and destination nodes can be selected interactively from the virtual map and their locations are subsequently reported and stored in their order of selection. Those locations are then tested against the (x, y, z) co-ordinates of every single node in the node database, to enable retrieve the identity index of each node. A sequential look up in the path database, for the pair of nodes that match the selected ones provides the fastest way of getting from source to destination.

To animate the results on the virtual map, each of the nodes in the graph is tested to see if it belongs to the path just found. Those that pass the test are represented in a different colour. The same is done to edges by successively taking two adjacent nodes from the path at a time and comparing them with the two end points of each edge in the graph.

Buildings in the city are not part of the connected graph. They are nodes of their own type and they are not linked to each other in anyway. Therefore, at this stage, looking for a path between two buildings will result in no answer. To solve this problem, we map each building with the nearest graph node to it (pavement node or road node, depending on the graph we choose to use). This image node represents the building front side or entrance, and looking for a path between two buildings is reduced to finding the path between their entrances.

Figure 74below shows an animation of the optimal path found between two different buildings in the city using a pavement graph.

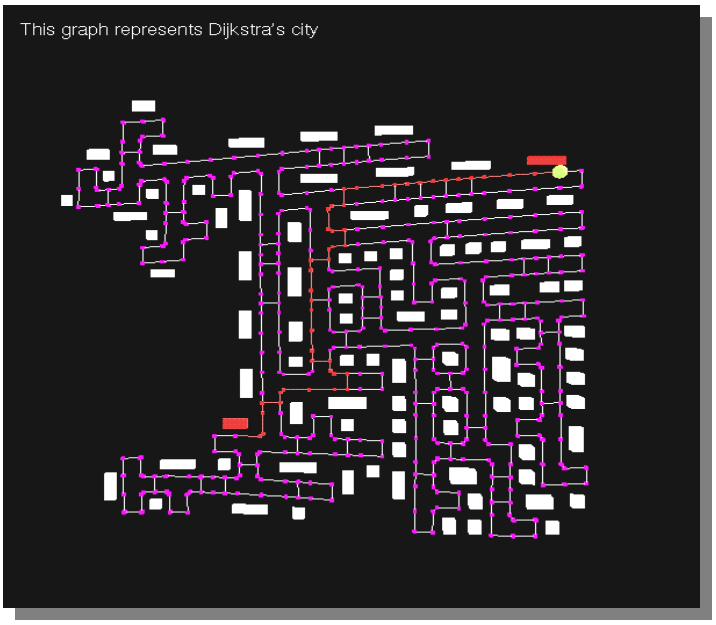


Figure 74: Finding the optimal path between two different pavement points.

Visiting a Series of Places

To visit a number of different places, different selected points and all relative information (including names for buildings) are stored in a queue structure so that the order of selection is preserved. Each pair of selected nodes is then considered in turn and the sub-problem is solved as in the previous section.

The figure below shows the optimal path found to visit five different buildings in the city using the pavement graph.

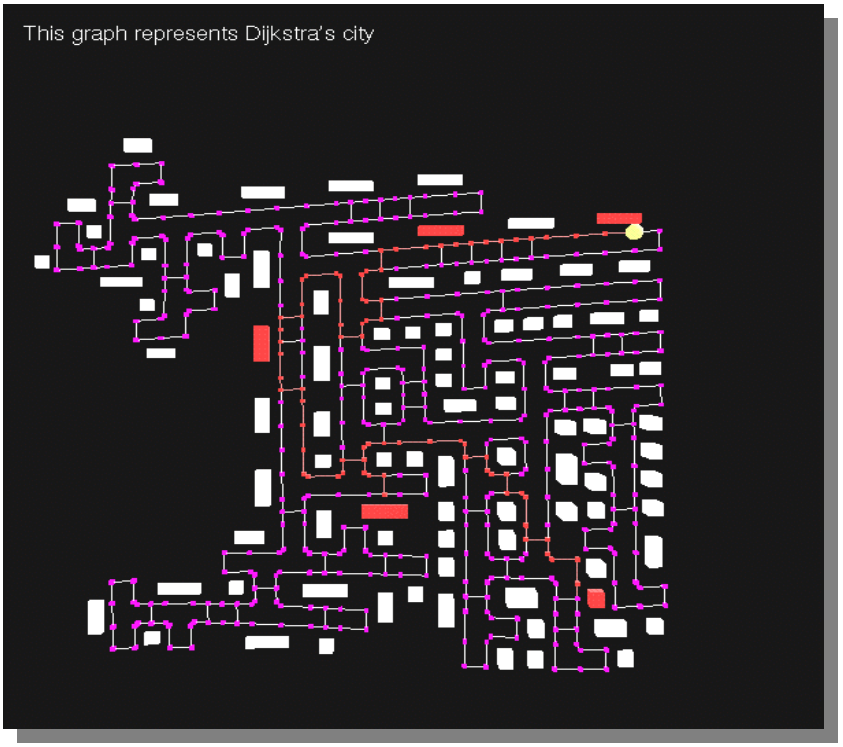


Figure 75: Visiting a number of different places using a pavement map.

Moving to the Nearest Place

A visitor, being at some point in the city, might want to visit the *nearest* pub for instance. To find a solution to this problem all pubs in the city must firstly be identified. The paths and their costs to get to them must then be calculated and the one with the least cost is then chosen.

Buildings in the graph are represented as simple MAVERIK objects whose semantics are stored in the building database. The building database maps ‘names’ to building objects and can therefore be used as intermediate means to locate all buildings that hold the requested ‘name’ in the city.

Given the name of the building the user wants to visit, a simple search through the building database identifies different indices at which buildings of that name exist. These indices can then be used to retrieve the position vectors of the corresponding objects

on the map, which are then compared with all building MAVERIK object matrices in the virtual map. The results can then be visualised on the map, by changing the colour of buildings object found to match. Figure 76 shows an example of this.

Figure 76: Locating all clubs in the city map.

Knowing the current position of the user, the path of each of the identified candidates can be found. The path with the least cost (distance) is then chosen, and the destination of the path is defined to be the nearest requested building. Figure 77 follows the results obtained from Figure 76 above.

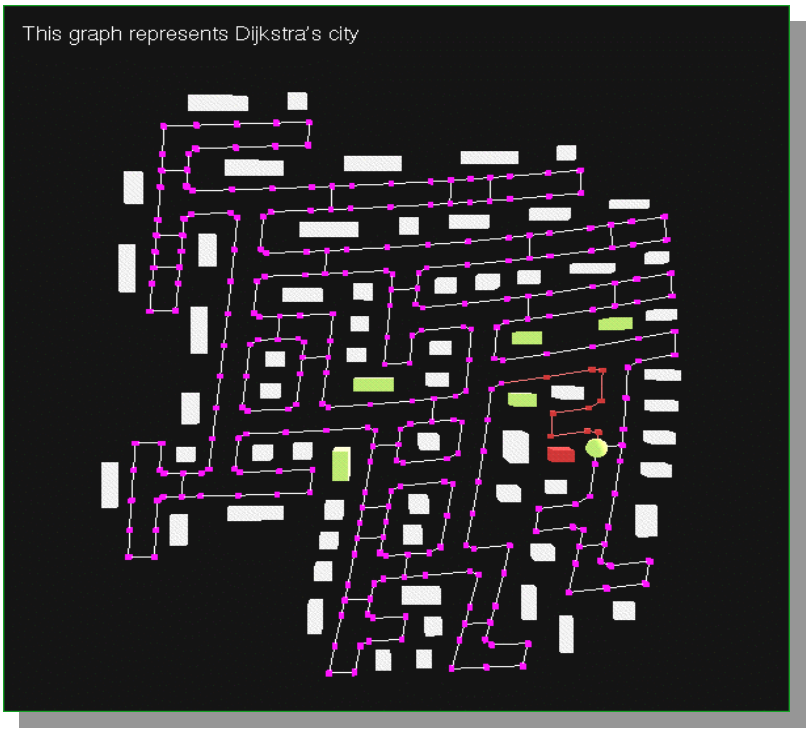


Figure 77: Choosing the nearest club. The club to which the path has the least cost.

Travelling the Path

In the previous sections the virtual map has been our centre of attention. We discussed ways of finding and visualising optimal paths around the city using the graph it represents. In this section we shall see how the results obtained so far can be used to navigate the cityscape effectively.

Each graph (map) node has its invisible counterpart in Dijkstra's city whose exact position can be found. This makes restricting moving along a predefined path relatively easy to realise, since the path is given in terms of a list of nodes.

Dijkstra is supposed to walk or drive the city visitors to their chosen destination. He is an avatar and his walking movements depend on two parameters, speed and direction. The speed can be set to any initial constant value. Obviously, if Dijkstra is driving he goes faster than when he is walking and so does the user navigation.

Both the cityscape and its associated virtual map are rendered using a MAVERIK infinite rendering loop that generates a new frame each time around. Dijkstra is rendered as part of the city and at anytime he exists at some position in the frame being rendered. In each frame, rendered at some time value T, Dijkstra is time-stamped and his location is marked. When the next frame is being rendered at time value S, Dijkstra's new location is calculated. Knowing the difference in time between the two frames (S-T) and Dijkstra's speed, it is easy to determine the distance by which Dijkstra should have moved from his old location. All is needed now is to set the direction that directs the move.

To restrict Dijkstra to follow the chosen path from source to destination, starting at the source node, the direction should be set according to the edge determined by two

adjacent nodes in the path. This direction together with the distance by which Dijkstra should move indicate the exact place where he should be placed in the new frame. Moving along the list of nodes (the path) and until the destination node is hit, the direction vector is calculated as the vector subtraction of the last node visited from the node about to be visited. This is guaranteed to work because all edges in the path are straight lines and can therefore be specified as vectors.

However, if the frame rate is slow or Dijkstra's speed is very high, there is a chance for Dijkstra to miss his route. This happens wherever there is a change of direction such as corner nodes where the direction changes after the new position is calculated which results in loosing track of the route. To overcome this problem a look-ahead test for corner nodes should be performed and when the new position is found to go past a corner node, it is set to be the position of that corner node.

The city visitor's location is represented by the eye point and the direction he/she is looking is given by the view point. To enable the user to have automatic navigation of the optimal path, the eye point and the view point vectors in the new frame should be set respectively to Dijkstra's position and his direction in the old frame. This allows Dijkstra to always be in the user's sight and give him/her the impression that he/she is following Dijkstra.

Change of Route

At any stage once the path navigation starts the user may wish to change his destination or may decide to stop at some place in the city. The user should not be tied with the planned trip and should be allowed to change his/her plans as desired.

One way of satisfying these requirements is to keep track of the user's position at any time and test if any new destinations have been selected. The new source node is given as the nearest node to the current user's location. The path from the new source node to the new destination is then found and animated in the same way as explained above.

Results and Conclusion

'Virtual city Guide 1' was intended to solve the basic navigation problem effectively and it certainly did so by enabling the user to travel between different places in the city following the best possible routes.

All user actions are initiated from either the pavement map or the road map. Starting at some point in the city the user interactively specifies a destination or a list of successive destinations. 'City Virtual Guide 1' enables the user to be shown the path he/she is travelling on the map as well as his/her position relative to the whole city and the path being travelled.

The figures below demonstrate a walkthrough of the sequence of different actions taken when navigating the path found using a pavement map.

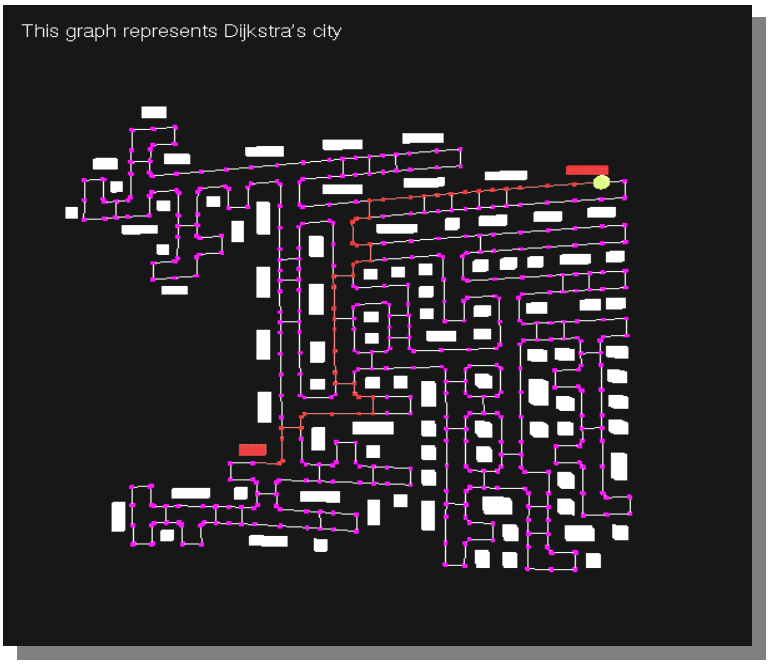


Figure 78: Finding the best path between 2 different buildings.

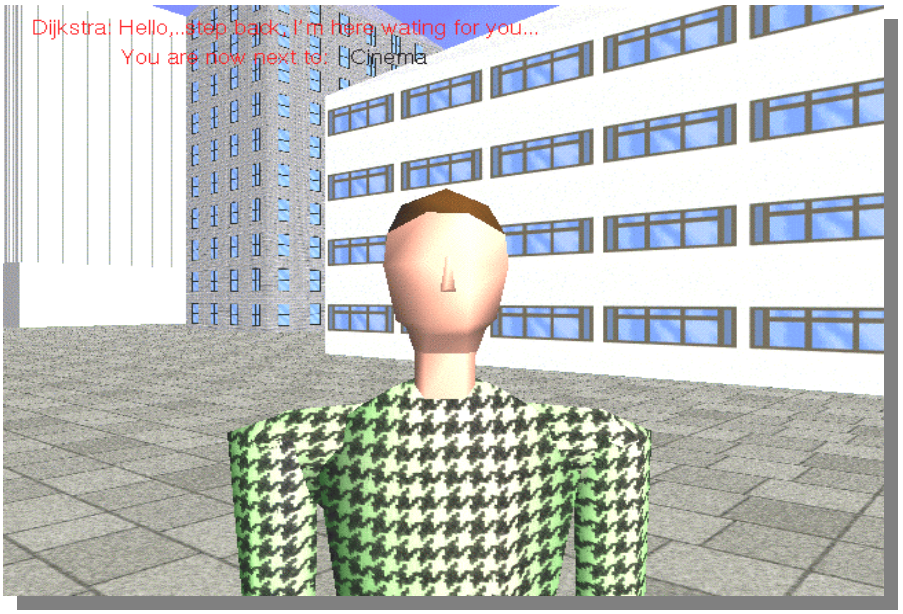


Figure 79: Dijkstra standing facing the user (eye point) near the source node.

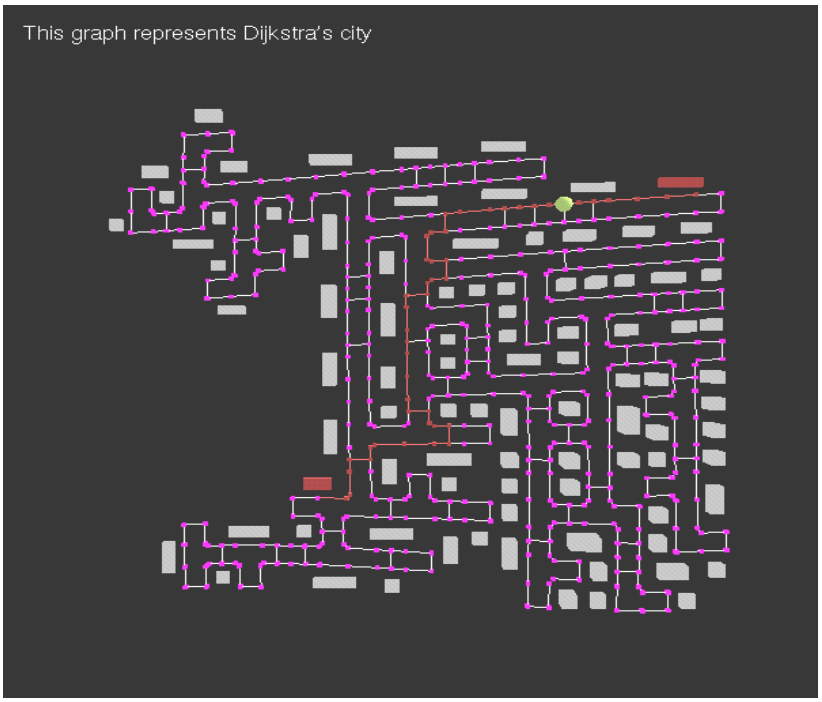


Figure 80: Animation of path travelling in the city graph.

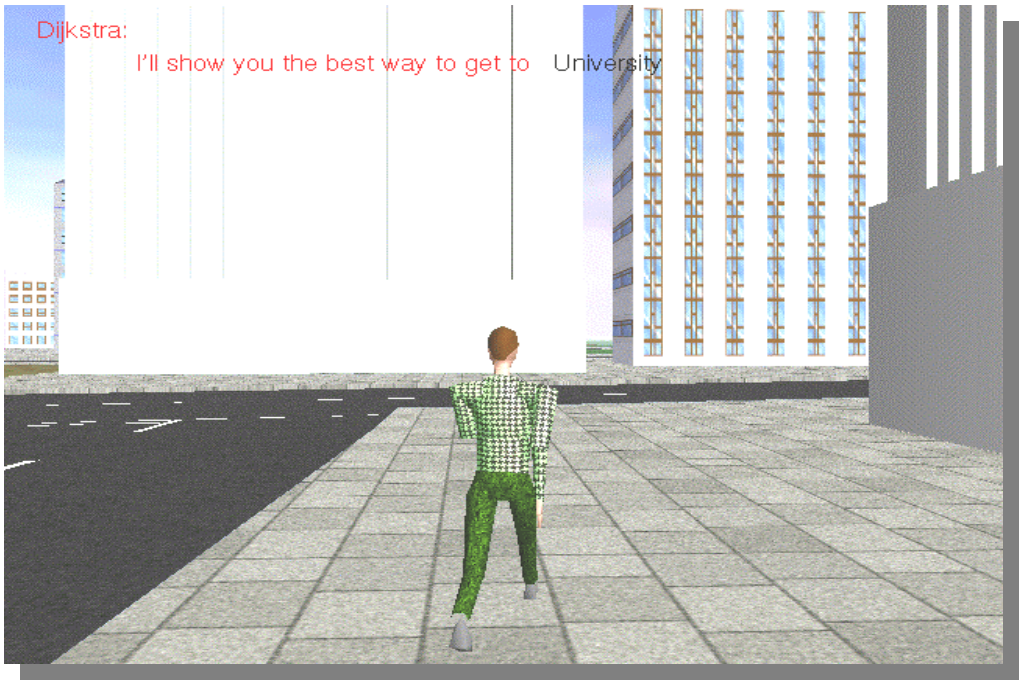


Figure 81: Dijkstra and the user travelling the path found.

Virtual City Guide 2

The virtual city guide designed and implemented as discussed in the previous sections provides a tool to effectively navigate Dijkstra's city. The initial problem was translated into a least cost path graph task, and the traditional Floyd Algorithm was used to find optimal paths around the city. However, this algorithm operates only on static graphs. The graph that describes the city is constructed once the city is generated and remains static. Assuming the city does not change, the graph will always be reliable. Altering the city environment may introduce new constraints that make the graph no longer reliable unless the changes are also reflected on it. In this chapter we discuss different ways of introducing such obstructions, and methods of detecting them and dealing with them efficiently.

Obstruction Introduction

Introducing obstructions is one way of modifying the existing city and graph environments. Adding an obstacle effectively cuts links between certain parts of the city and certainly affects a number of paths stored in the shortest path database. This should be performed by the user interactively by placing an obstacle any time (barrier) anywhere in the city. A simple way of doing this is to convert any of the nodes to an obstacle. This has the advantage of ensuring that obstacles lie on one of the paths as well as making it simple to locate a particular obstacle. We keep track of the obstacles by flagging the affected node as suspended. This is shown in Figure 82 below.

Add obstacle and mark the node as Suspended.

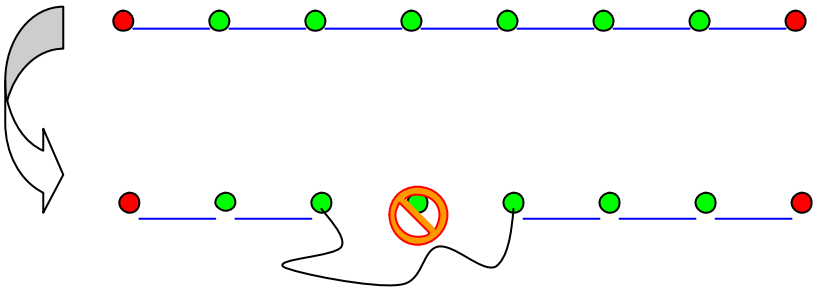


Figure 82: Adding an Obstruction.

A path between two nodes is determined initially by consulting the shortest path database. Each of the path's constituent nodes is checked to see if it belongs to the set of suspended nodes. In case the path has at least one suspended node, it is ruled out and an alternative needs to be found.

One-way Street Addition

As described previously the graph construction method results in an undirected graph which can also be viewed as a graph in which all edges are bi-directional. The advantage of this is that one is able to deduce the optimal path from A to B knowing the optimal path from B to A. One being the reverse of the other. However, this simplicity does not reflect a realistic environment, particularly in the road graph version where some streets could go only one-way but not the other. Allowing one way streets effectively means that some edges are dropped from the original graph structure. Consequently, the optimal route followed from A to B is not necessarily the reverse of the optimal route from B to A.

One way of setting two-way streets to be one-way is to do so, on a random basis, when the graph information is being extracted from the city. The graph database generated then contains within it one-way street information and the shortest path database takes that into consideration as well. The main advantage of this besides simplicity is that everything is set beforehand and the graph remains static and unchanged once formed. However, this limits the user's interaction and control. On top of that, the street random setting may result in deadlocks in situations where an intersection point has a number of in coming edges but no outgoing ones.

A better option is to allow the user to modify the graph. In the same way the user can interactively block paths, he/she can also set streets to be one-way or the other. This broadens the user interaction, freedom and control. The graph is initially generated in the same way described in chapter 4 where all streets are available both ways, by default. The user can then, at any time, set any street of his/her choice to be one-way in the direction preferred. The main issue here is that the database of all shortest paths that stores the shortest paths in the initial graph become no longer consistent with the modified graph version. Therefore, the shortest path between two points, as stored in the path database, may not be available any more and an alternative needs to be found.

A street in the road graph version is a series of nodes that begins and ends at a corner node inclusive. To identify the street that needs to be set one-way its two end points need to be detected. It is possible for the user to explicitly state the two end points. However, if the user can state only one of the intermediate points that lie in the street, it is possible to find the two end-points that define it. This is a better technique as it minimises the amount of work required by the user. Once the two end-points are found, all the edges that match the direction chosen are kept and all others in the opposite direction are marked unavailable.

The path between any pair of nodes is initially found exactly in the same way as before by consulting the path database. When the path is found, each of its constituent edges is checked whether it is flagged unavailable or not. If the path has at least one unavailable edge, it is ruled out and an alternative needs to be found. This is described in the next section.

A graph can be modified even after the path is found and the travelling begins. An obstacle may be added or a street may be converted to one-way after the path travelling has started. Therefore each of the above checking operations is performed

when the path is retrieved for the first time as well as when each time around the rendering loop and a path is being travelled. In the latter, not the whole path is checked but only the part not yet travelled. In other words, only obstacles (or one way streets) that lie on the path and ahead of the navigator should be reported. Changes that do not affect the remaining route to be travelled should therefore be neglected.

Finding an Alternative Path

Allowing the user to introduce local changes to the original graph makes the graph dynamic and evolving. This results in some of the paths stored in the shortest path database being no longer available. If a path is detected to be ruled out for any of the reasons explained above, the best possible alternative needs to be found to enable the navigation process.

Unfortunately, traditional graph algorithms such as Floyd's only operate on static fixed graphs. There is no incremental version of Floyd's Algorithm that enables us to successively retrieve optimal alternatives to a given path, having done the computation once.

Clearly one straightforward option is to re-compute the shortest paths from scratch whenever the graph is modified. This involves a modifying the graph database, passing it to Floyd's Algorithm and storing the new obtained paths in the path database. Although this solves the problem, it is time consuming and therefore not effective. For a large graph with a large number of nodes and edges, the re-computation process is a matter of minutes. Clearly we do not want the application to stall every time the graph gets altered.

By examining the basic idea of Floyd's Algorithm, one can notice that the optimal path P from A to B and through any k is constructed of the optimal path from A to k and the optimal path from k to B. One can deduce that if P is blocked at some point, there could be some other intermediate node m such that the combination of the optimal path from A to m and the optimal path from m to B, where both sub-paths are not blocked, produces an optimal alternative to P.

All the graph nodes are candidates to be m. Floyd's early computation stored in the path database can be used to find the optimal paths from A to each m, and from each m to B. The two sub-paths, for each possible m, are tested for availability and if they are clear of any obstacles they are merged together into a single path. This process could lead to a number of alternative paths because the large number of possible intermediate nodes. To choose the best alternative out of the ones found, the cost of each is calculated and the one with the least cost is chosen.

A study of this proposed solution shows that it always finds the best alternative possible if one exists. On top of that, it solves the problem only for the affected path and therefore avoids a costly re-evaluation of all-pairs shortest paths in the altered graph.

Results and Conclusion

'Virtual City Guide 2' extended 'Virtual City Guide 1' by upgrading the city and graph from static environments into non-fixed dynamic environments. This was achieved by enabling the user to interactively edit new constraints into Dijkstar's city via the graph that represents it. The user could close roads of his/her choice by placing an obstacle in the way. Additionally, the user can set roads, which are by default two-way, to be one-way.

'Virtual City Guide 2' also provided ways of detecting inconveniences and reporting them to the user. Besides a new algorithm was developed to instantly calculate optimal alternative paths in order to avoid re-evaluation of all-pairs shortest paths that could be extremely costly.

Figure 6.3 below shows how an already calculated path between two points has been interactively closed and this is interpreted by placing a barrier at the corresponding place in the city and that stops the travelling process as figure 6.4 shows. Figure 6.5 shows how the best alternative path to the first one was calculated and animated. This new alternative path is the one to be travelled and this is done in the same way as illustrated previously.

Figure 83: Placing an obstacle in the way makes the path no longer available.

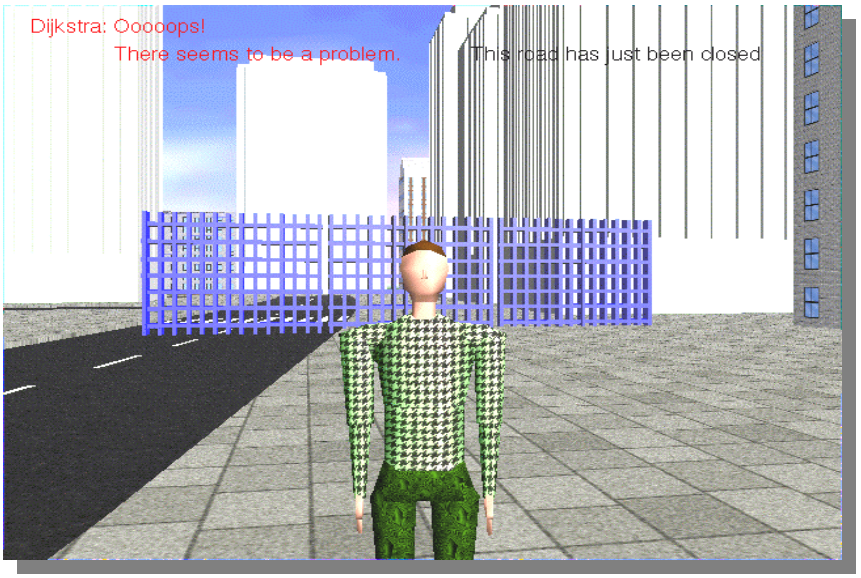


Figure 84: Placing a barrier stops the travelling process.

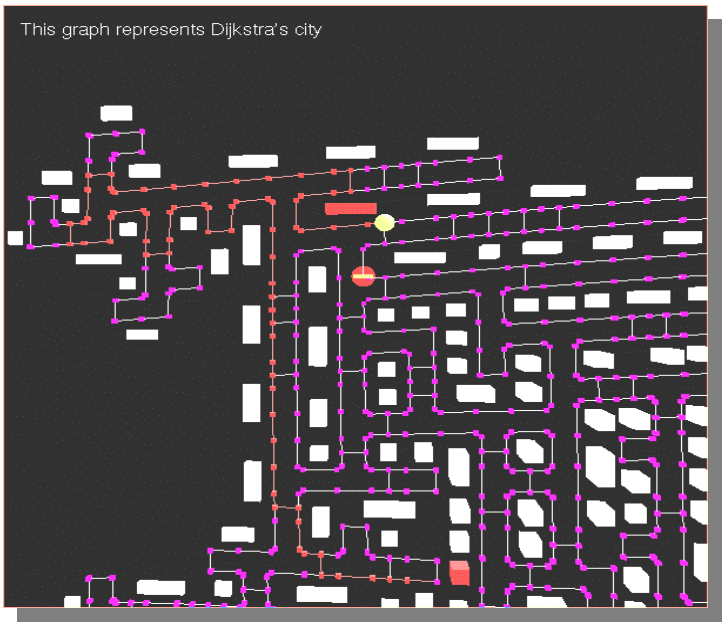


Figure 85: Finding best alternative possible to the same destination.

The User Interface

Early planning identified that an important principle of the user interface design is that the user should always feel in control of the application, rather than feeling controlled by the application. It also revealed that providing a sense of stability and consistency makes the interface familiar and predictable. In this chapter I shall throw some light on

the development of a graphical user interface and a voice recognition interface to complete the application. I shall address the main design and implementation issues.

The Graphical User Interface

The Graphical User Interface should be simple (not simplistic), easy to learn, and easy to use. It must also provide access to all functionality provided by the virtual guide application. Maximising functionality and maintaining simplicity work against each other in the interface. Hence, balancing these objectives is needed.

In order to help users manage complexity the interface uses progressive disclosure. Progressive disclosure involves careful organisation of information so that it is shown only at the appropriate time. By "hiding" information presented to the user, you reduce the amount of information to process. For example, clicking a menu displays its choices; the use of dialog boxes can reduce the number of menu options.

XForms introduced could be combined with MAVERIK to address the above issues and moderately meet their requirements. XForms provides a library whose main notion is that of a form. A form is a window on which different objects are placed. Such a form is displayed and the user can interact with different objects on the form to indicate his/her wishes.

Implementation and results

XForms library provides many different classes of objects, like buttons that the user can push with the mouse, sliders with which the user can indicate a particular setting, input fields in which the user can provide textual input, menus from which the user can make choices, etc. Whenever the user changes the state of a particular object on the form displayed the application program is notified and can take action accordingly.

However, using MAVERIK it is not very straightforward to check and report Form Events. The reason is that MAVERIK rendering of the virtual city and the corresponding map goes around an infinite loop generating a new frame each time around. Standard checking as well is a continuous process that is done in the same way and it is not possible to have two independent infinite loops. This can be solved by writing a special function to check for events and let it be called inside the main MAVERIK loop. The form objects' states are queried while the new frame is being created and any changes are reported and queued. The application program interacts with the form using a number of callback routines that are called whenever an event is picked up. Therefore, actions are triggered using the form and the results of any actions carried out are visualised using MAVERIK.

Description

The figures below show the interface form and the different objects contained within it. Different actions associated with the form objects are described in the table below.

XFORMS INTERFACE PANEL		
Object Label	Object Type	Call-back Action
Choose Graph	**menu**	**Enable the user to choose to be shown either a road graph or a pavement graph.**
Alter Graph	**menu**	**Enable the user to modify the displayed graph. The choice sets the functionality of the middle mouse button.**
Clear all	**Button**	**Returns to the original unmodified graph**
Find shortest way	**Button**	**Animates the best path between the pair of nodes selected or a series of them.**
Take me there now	**Button**	**Start travelling the path.**
Next place	**Button**	**Visit the next place in the series of places selected.**
Find an alternative	**Button**	**Find an alternative path between source and destination.**
Where am I?	**Active Button**	**Once activated, it keeps track of where in the city the navigator (user) is and the nearest building to him/her.**
Take me to the nearest	**Type-in box**	**Takes a building name as input.**
Speed	**Slider**	**Set Djkstra's speed (as well as user's navigation speed)**
Exit	**Button**	**Exit application.**
MAVERIK Virtual Environments (city & map)		
Right Mouse Button		**Up and down navigation**
Left Mouse Button		**Left and right navigation**
Middle Mouse Button		**Object selection: for source and destination selection as well as graph modification.**

Figure 86 Interface objects and their associated callback actions.

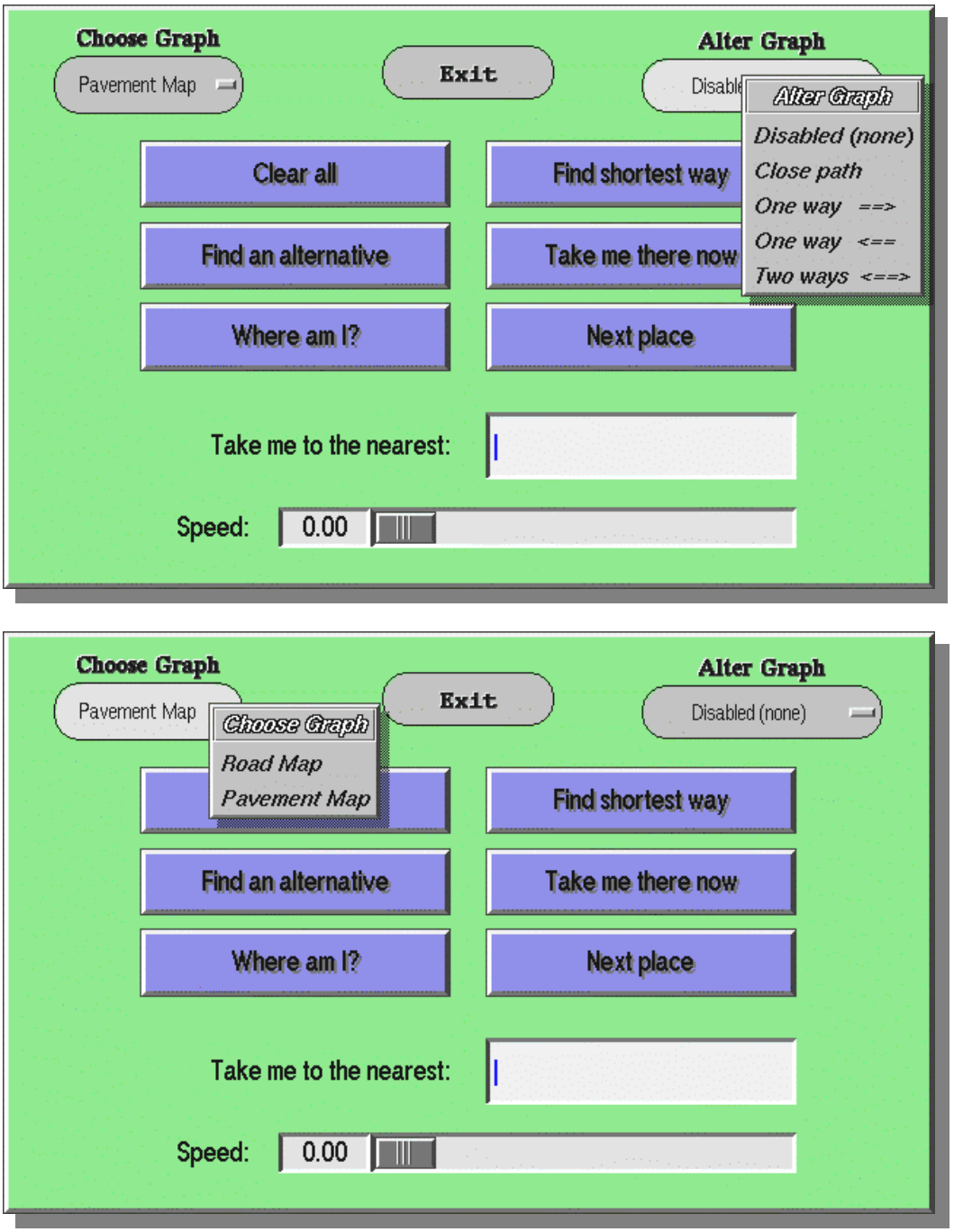


Figure 87: *The XForms Interface Panel.*

Speech Control

The Marconi Macrospeak must be integrated with the existing system to add voice-input facility. Macrospeak is programmed to allow voice activated control of the city guide implemented. The Macrospeak program defines the list of spoken words which may be used during recognition, the syntactic rules which govern their concatenation and the output from Macrospeak in the event of their use.

The word list consists of the set of mandatory vocabulary used to perform any of the actions allowed using the XForms interface panel introduces in the previous section. The table below shows the vocabulary needed for this purpose.

It should be noted that most desired actions are matched with only one command word in the list above. This has an advantage in that in most cases only one possible choice is available to the recogniser at any time, thus improving recognition accuracy for a co-operative speaker. It also has the advantage of simplifying the application syntax rules since not many words are required to be followed by others. Consequently, it prevents invalid word combinations from generating erroneous responses. The syntax rules are presented in the diagram below.

	Word	Syntax Class
1	**Exit**	**Single command**
2	**Road_Graph**	
3	**Pavement_Graph**	
4	**Obstruct**	
5	**Two_Way**	
6	**One_Way_**	**One_way**
7	**Left**	**Direction**
8	**Right**	
9	**Clear**	**Single command**
10	**Next**	
11	**Take_Me_There**	
12	**Slow_Down**	
13	**Speed_Up**	
14	**Stop**	
15	**Find_**	**Find**
16	**Nearest_**	**Nearest**
17	**Shortest_Path**	**Request**
18	**An_Alternative**	
19	**Post_Office, Coach_Station, Club, Park, ...etc.**	**Building name**

Figure 88: The list of possible voice-input vocabulary

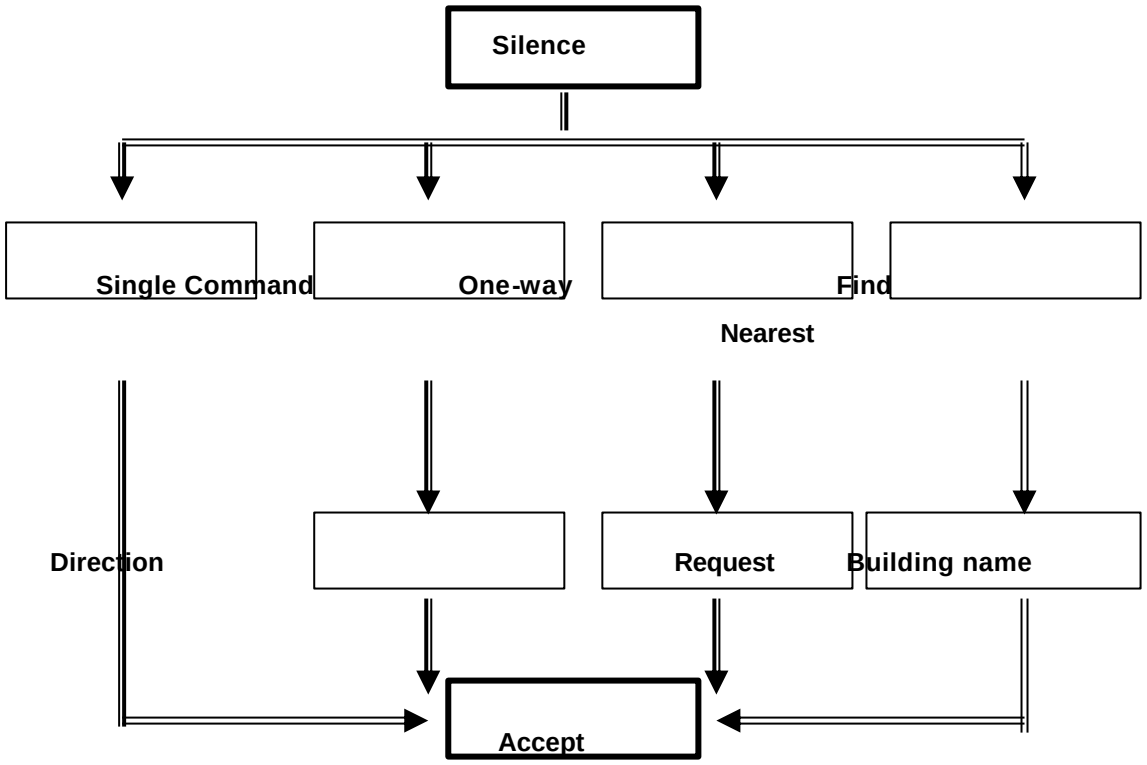


Figure 89: The Speech Syntax Diagram.

Implementation

Before the recognition proceeds, Macrospeak needs to be trained. Training in the process which provides Macrospeak with a spoken sample of each word in the word list. Each of the words is uttered in a clear positive manner. These templates are saved in a recognisable format by Macrospeak. They are used as reference patterns and are definitive version of each word.

Once programming and training are complete, Macrospeak can be connected to the application to enable receiving voice input from the microphone and intelligent interaction with the virtual guide system. Call-back function for each command input are defined exactly in the same way they are defined using XForms to perform the desired actions.

Feedback to the User

At any stage while the user is exploring the city, he/she expects to be given enough feedback on actions taken and changes occurring in the environment he/she is exploring. For instance, the city navigator expects to be given an estimate of the cost of his/her journey before he/she starts to travel as well as after the travel begins. Besides it will be an advantage for the user to know where in the city he/she is.

Effective feedback is timely, and should be presented as close to the point of the user's interaction as possible. It should also communicate details that distinguish the nature of the action. Nothing is more disconcerting than a dead screen or an unresponsive interface.

The above can be satisfied by providing messages for the user about his current location in the city, the path he/she is travelling and the cost updates. The cost is given in terms of distance and time left for the navigator to get to destination. Clearly, while the distance is fixed, the time is estimated considering the current navigation speed and varies as the speed (of Dijkstra and therefore the user) changes.

This could best be done using MAVERIK strings to display messages, because the states and/or values of most of the above parameters change with every new frame rendered. Messages could be displayed on a separate frame that gets updated each time around the main rendering loop to reflect any changes. Figure 7.5 shows an example of that.

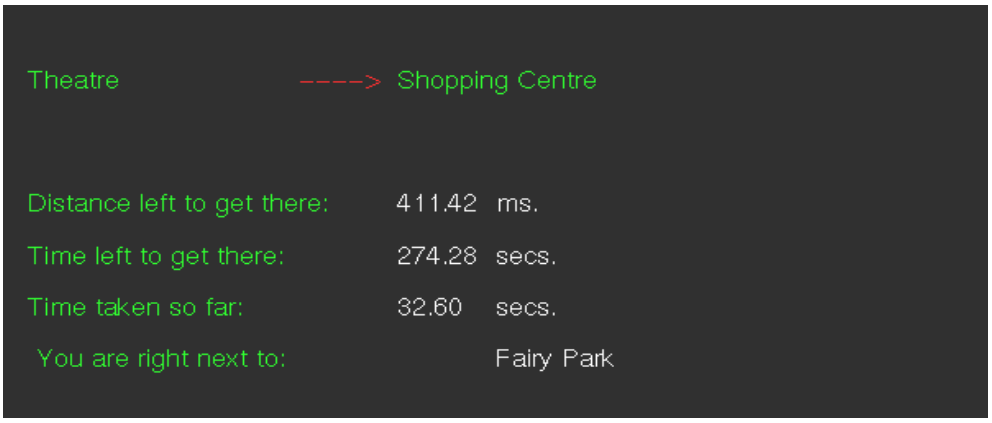


Figure 90: MAVERIK Frame to give feedback to the user. This includes the path he/she is travelling, its cost and the user's current location

Chapter 7
Crowd Control: populating the virtual cityscape

David Smith, Adrian West and Steve Pettifer
The University of Manchester

Navigating around virtual cities can feel like a lonely experience, as usually there is only one individual, or maybe a handful of people, inhabiting it at any one time. As the technology for networking such cities improves, it may be possible to fill cities with hundreds, maybe thousands, of real people, but until such time another solution needs to be found. The most obvious solution is to create simulated people to wander around the city and get on with their daily lives, and this is the main aim of this project, going by the name of 'Crowd Control'.

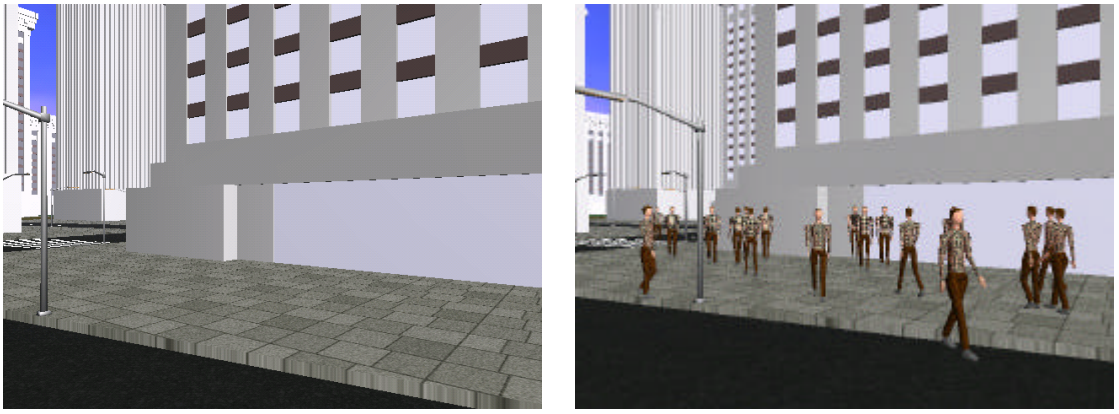


Figure 91: The cityscape, without and with crowds

One of the main parts of the cityscape project, as has been mentioned above, is to try to find natural ways of integrating information into virtual worlds. For instance, one thing that might be required is to draw the user's attention to various objects of interest within the world. There are several ways that this could be done. Possibly a big arrow could be put on top of any such objects, or they could be marked in bright colours, made to produce various noises, or maybe the user could be given a virtual map with such objects marked on it. However these solutions can look out of place in an otherwise realistic world model. A more subtle way of drawing attention to these items could make use of the crowd simulation – simulated people passing nearby could stop and look at the objects, with crowds of people congregating around the most interesting items.

There is a third 'real-world' application for the crowd simulation and that is to attempt to predict the movement of crowds in as yet unbuilt cities. Assuming that the

simulation is accurate enough this could be a valuable tool for town planners, allowing them to spot potential problem areas before construction work is started.

Interactive Frame Rates

One important feature of the Maverik system, as with most VR systems, is that it attempts to create worlds that run at sufficiently high frame rates for real-time interaction. It is generally reckoned that a rate of at least 10 frames per second is the minimum for smooth movement. As a result of this, an important feature of this project is to find fast and efficient ways of simulating crowds, thereby leaving as much time as possible for drawing them on the screen. The project also has to be scalable, so that increasing the number of people in the simulation, or the size of the city, does not exponentially decrease the frame rate. Throughout this document there are mentioned several different ways in which this is achieved.

On-line references

There has not been very much research, to date, in the field of real-time crowd simulation, but there are a few papers and other resources available on the Internet. Some of these, such as the Legion project (http://ourworld.compuserve.com/homepages/G_KEITH_STILL), are designed to accurately simulate the movement of crowds, but not in real time. This is useful for designing or redesigning areas through which crowds have to move, but is not really relevant to this project.

There are also commercial packages available, such as the Rampage system (http://www.anisci.com/RAMPAGE/rampage1.htm), which are designed to model large groups of animals. This could be customised to create animations of crowds of people, and does have a ‘goals’ system, very similar to that used in this project. However, like the Legion project, this is not designed for real-time simulation. In this case it is designed to simplify the creation of ‘herd’ animations (such as the stampede seen in the film ‘The Lion King’).

Of more relevance to this project are papers by Prof. D. Thalmann (Thalmann, 1998) and by D. Thalmann and R. Musse (Thalmann & Musse, 1997). Both of these look at the inter-relationships between different crowd members, something that has been only lightly touched on in this project (which concentrates more on the movement of the individual). The second paper also looks at ways of detecting potential collisions and avoiding them.

On Screen

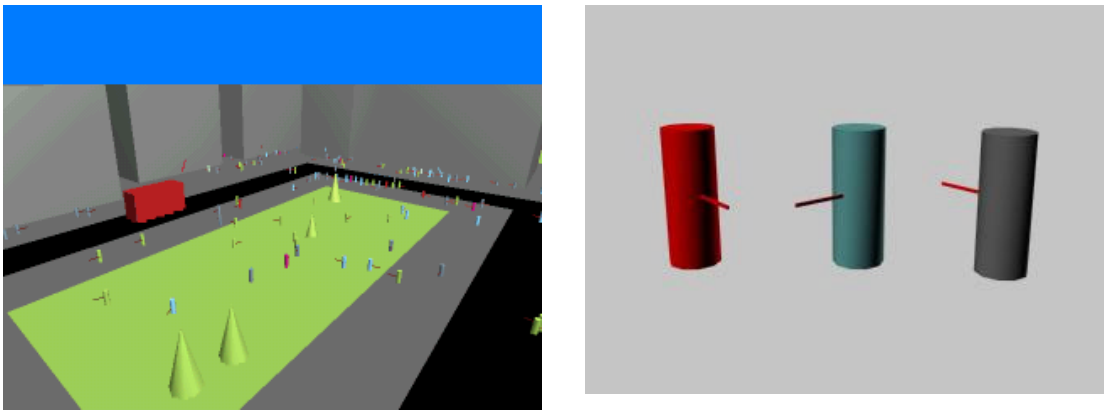


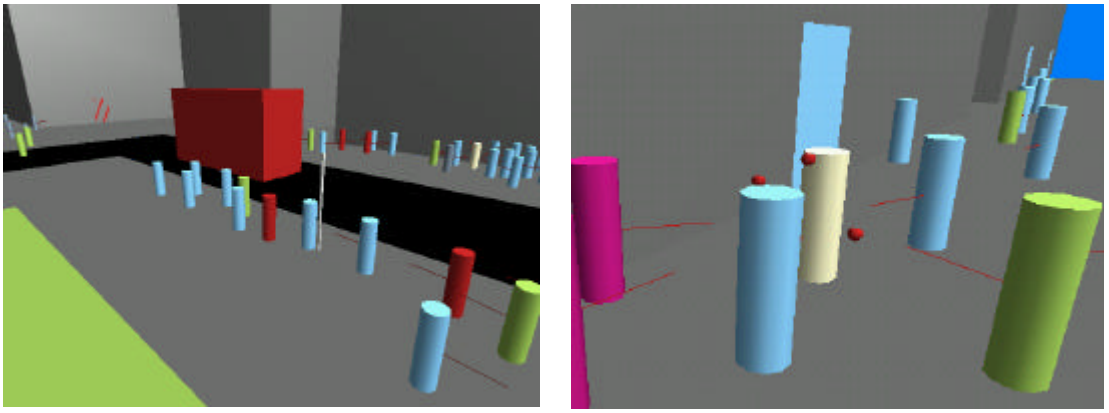
Figure 92 : **The** simplified cityscape and its inhabitants

This section describes what appears on screen once the program has loaded. It also describes how the city appears to operate and gives details about the interaction that is possible with the world.

Objects in the City

A fairly simple representation for the city was chosen, to allow a reasonable frame rate, and also to leave extra time to spend on the more important simulation aspects of the project. The city used in writing the program is based on a 25 by 25 grid. On screen each grid square is either a flat square - which represents either grass, pavement or road – or a box, representing buildings. Different colours are used to differentiate between ground types, and there are also a number of randomly placed trees on the grass squares.

The people inhabiting the city are simply represented as cylinders with a single red line, which show the direction they are facing (see fig 4 above). Again this is mainly due to the problems of getting a good frame rate with complex models such as shown in fig 2, but also it simplifies that part of the code.



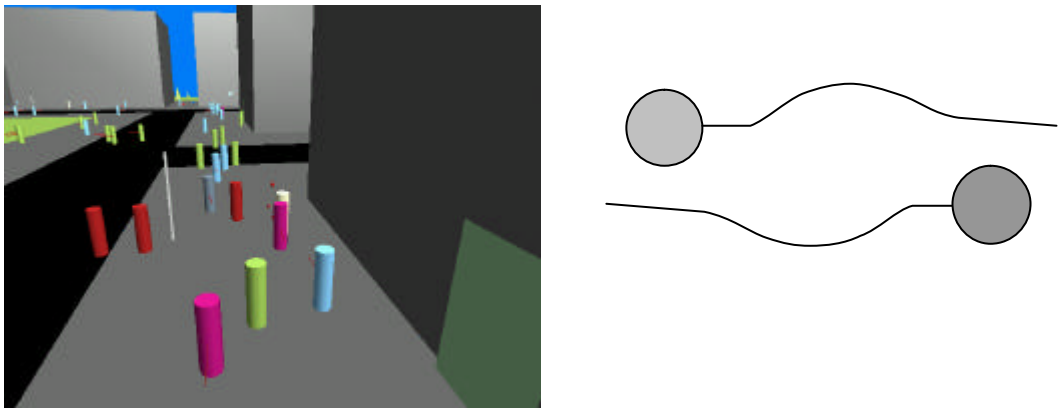
(a) (b)

Figure 93: A bus arrives to pick up a queue of people, and a 'juggler' attracts attention outside a doorway

Other objects around the city include buses (red boxes), doorways (coloured rectangles), bus-stops (tall, thin boxes) and jugglers (cylinders with animated spheres).

People Movement

As time goes by people appear in the doorways of the houses, denoted by the green coloured rectangles. These people then proceed to walk along the pavements, changing direction when they reach corners, and crossing roads at pre-defined places. As the people walk along they move out of each other's way to avoid colliding. They also avoid walking through buildings.



(a) (b)

Figure 94 : (a) Walking along a street, and (b) avoiding collisions

When they first set out, all of the people have at least one destination in mind, and this is where they initially head for. The different types of goals can be easily identified as the people have been colour coded to indicate their destination – red for bus-stops, green

for parks, grey for offices, blue for shops and magenta for a friend's house. As the people wander about the city, other objects may distract them, in which case they will go and look at these new objects, before continuing on to their original destination.

Whether it was their original goal, or something that distracted them on the way, the person will eventually arrive at an object of some kind. When they arrive at an object, their next action depends on what type of object it is. For doorways, people disappear into them, possibly re-appearing later. Bus-stops cause people to queue up until a bus arrives for them to get onto. Jugglers cause people to stand around and watch them for a bit. Finally, at parks people wander around and admire the scenery.

Interacting with the world

Although most of the simulation runs on its own accord, there are a few ways of interacting with it as it runs. Most obviously it is possible to move around the city and view it from any position. To get closer to the action it is possible to select an individual, and the viewpoint will then follow them until they leave the map (e.g. when they enter a building or a bus) or they are deselected. It is also possible to turn both the collision avoidance and distractions on or off. The former causes people to pass through each other and the latter makes sure that they only go to their initial destinations, and are not distracted by anything else on the way. The movement speed of the people is also adjustable. This does not affect any time spent waiting at a goal, but merely reduces the time it takes to get between different destinations.

An on screen display can be brought up, giving various bits of information about the current state of the simulation, such as the number of people in the map, the frame rendering time and the destinations for the current avatar (if one is selected).

One final, less serious, way of interacting with the world is the ability to 'shoot' people, by pointing at them and pressing 's'. The person falls over and nearby people move over and cluster around them, in much the same way they do with the jugglers.

Behind the scenes

External files

There are three external files that are needed to make the program work. The first of these is the city grid file, and this contains the definition of the different areas in the city (either grass, pavement, road or building). The second file type is the junction file which lists the positions of all the invisible navigation junctions in the city, and the connections between them. The final external file is the objects file, which describes all of the different objects to be found in the city (such as the doorways and buses).

Detailed information about all the file types can be found in appendix A at the end of this document.

Main Data Structures

The city is represented by a 25 by 25 array, with each entry in the array containing an integer that describes the type of ground in that square. Another array of the same size stores lists of people who are in each square at any one time. This is used to optimise collision detection, and is described in detail later in this document. There is also an array that contains all the junction points, with their position and a list of adjoining junctions, and a separate structure which stores the quickest route from one junction to another. The quickest route is stored in an n by n array (where n is the number of junctions in the city), which lists the next junction to which the person should travel to reach their destination junction. This array is generated using Dijkstra's algorithm (see previous chapter).

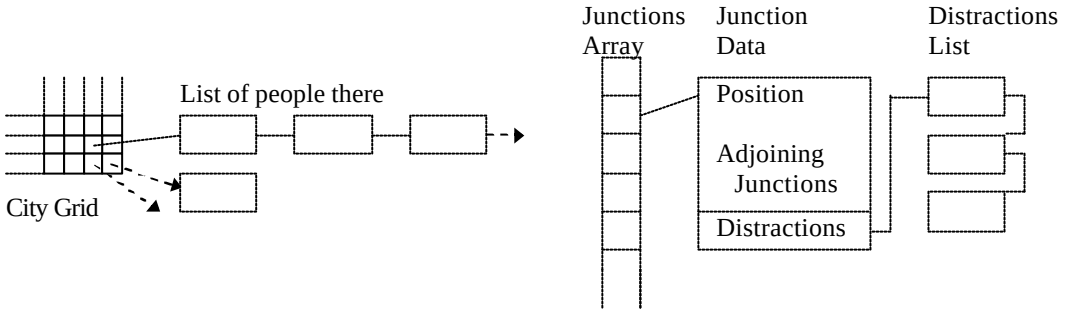


Figure 95: (a) grid square inhabitation lines, and (b) junctions data structure

All of the objects in the city (except for decorations like the trees) are stored in an objects data structure. This contains information like its position, distraction value and a pointer to the Maverik object that represents it. There is also an optional sub-structure off this that contains information relevant to people objects, such as a list of goals and a movement speed. This allows people to be stored in the same data structure as all of the other objects in the city, which, amongst other benefits, avoids having a separate data structure, and allows people to be used as distractions. The goal list sub-structure includes information such as type, position, object, and wait time.

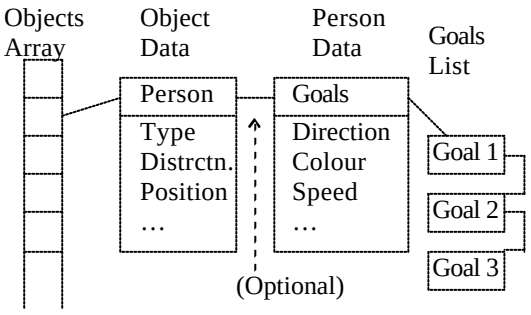


Figure 96: Object data structure'

There is a limited number of people in the city at any one time (about 200 people seem to fill the city and still giving a reasonable frame rate, see section 3.4 for more details). This prevents the frame rate from getting too low, and also allows the people to be stored in an array, which is easier to index and move through than a linked list. Any extra people wanting to get into the city are added to a waiting list from which they are removed when a space is found.

Full details about these main data structure can be found in appendix B at the end of this document.

Updating Objects

Before each frame is drawn every object in the city is updated. The actual update is dependent upon the type of object.

Busses

Buses each have a list of positions to travel to. They start out at their initial position and then head for the first position on their list. As they travel along their orientation is calculated from their current position and their goal position. The list of positions wraps around, with the last entry pointing back to the first entry, but, to distinguish it, the last entry is marked with a negative y-value (as the bus moves along flat ground, the y-value is not otherwise used). The bus has a status value that is used for two different purposes. If the bus is travelling from the last point, back to the first point, this status value is set to '-1', to indicate that the bus is not to be drawn. When the bus arrives back at the first point, the status is reset to '0' and the bus is drawn again. The second use for the status value is when a bus arrives at a bus-stop. If there are people waiting to get on the bus, then this value is set to the number of people waiting. The bus will not move until this value has dropped back to '0' again.

Bus-stops

Bus-stops look for passing buses for people to get on to (this is a more efficient solution than trying to add a new 'waiting for bus' state to the people at the bus-stop). When a bus is spotted, the bus-stop looks through the list of people waiting there and randomly sends about 2/3 of them to the bus (by adding the bus to the front of their goal list). The bus is also told how many people to wait for, and will not move off until that many people have arrived at it. The rest of the people then have their positions updated, to make sure that the queue moves up to the bus-stop again.

Doorways

Each house doorway has a wait time associated with it. If that time has elapsed then a new person is generated, and the wait time is set to:

$$\text{current time} + \frac{\text{random number } (0.0 < x < 1.0)}{10000}$$

```
        maximum number of people * average avatar speed * creation
rate
```

This attempts to match the number of people being generated, with the number of people being removed and the number of 'person-slots' available. The 'creation rate' decreases as the queue of people waiting to get into the city increases, to prevent the queue from getting too large.

When a person is generated their type is chosen at random (either going shopping, going to work, getting a bus, visiting a friend or visiting a park), and their colour, distractibility and goal list are set to relevant values (see table below). The different types all have the same probability of being chosen (except for 'getting a bus', which is less likely, due to limited space in the bus-stop queues), but a statistical bias could easily be introduced. The newly generated person is then added into the waiting list, which is ordered according to creation time (this allows people to be generated with a delay, for example, when a person goes into a shop there is a delay before they are 're-generated' by the shop doorway).

Goal Type	Colour	Distractibility	Goal(s)
Shopping	Cyan	0.5…1.0	shop doorways
Work	Grey	0.0…0.2	office doorway
Bus	Red	0.0…1.0	bus-stop
Friend	Magenta	0.0…1.0	house doorway
Park	Green	0.4…0.6	park

Jugglers

The positions of jugglers' juggling balls are updated. The jugglers themselves do not move, due to problems with moving distractions.

Parks

Parks are not updated at all, as they do not change during the time scale that this project works on.

Updating People

When the program first starts, all of the 'people-slots' in the city are initialised to 'not existing'. In this state they do nothing, except wait for a person to appear on the waiting list. When a person appears on this list, their details are copied across into the empty slot, and they are removed from the waiting list. The object representing the person will then appear in the city and start to move through the list of goals that they have been given.

Moving from goal to goal

When a goal for a person is first set the nearest junction to the goal is worked out (by simply comparing the distance to all of the junctions and taking the smallest), and stored along with the rest of the goal details. When the person first 'uncovers' this goal they perform a few checks. Firstly they check to see if they have a direct line of travel (LOT) to the goal. This means that they can go to the goal directly without passing through any buildings, or arbitrarily crossing any roads. If they have a LOT then they will go directly to the goal; otherwise they will navigate via one or more junctions.

The next check they make (assuming no direct LOT) is to find the nearest two junctions to their current position, and then choose whichever of those is closest to their goal as the first junction to travel to. Initially the nearest junction to the current position was always used, but this resulted in at least half of the people going the wrong way out of their front door, and then turning around and walking back the way they came.

Once the initial junction has been chosen the person will add this to the front of their goal list and then move towards it. To prevent the people from all unnaturally travelling down the centre of the pavements, each person actually heads for a point a random distance away from the actual junction point (although still well within the same grid square). Upon reaching the junction they once again check the LOT. If they can travel directly to their goal, then they do so, otherwise they use the 'quickest route' list to work out which junction to go to next. If, for some reason, the person arrives at the nearest junction to their goal and still does not have a LOT they will go directly to the goal anyway. This either indicates that the goal is inside a building/on a road, or that the nearest junction is on the other side of a building/road. In this case the junction list is not extensive enough and should be manually updated.

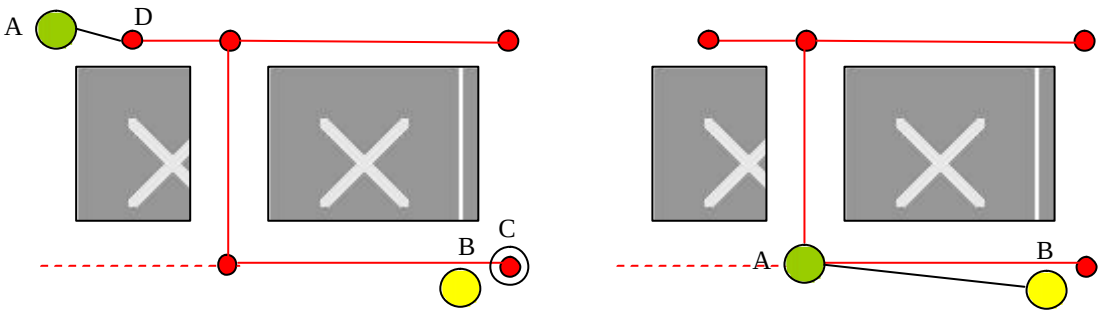


Figure 97: (a) Person A, wishes to get to object B. They first find the junction nearest the goal – C – and the first junction to travel to – D. (b)

Arriving at a goal

When a person arrives at a goal, their next actions depend upon the type of goal that they have arrived at. The simplest goal is the junction, which has already been discussed above and the rest of them are discussed below.

Doorways

When a person arrives at a doorway a new goal is added to the top of their goal list that removes the person from the map and resets the slot to the initial 'waiting for a new person' state. In the case of a shop doorway the details of the person are first copied onto the waiting list, with a suitable delay before they are re-created. This allows the person to reappear from the shop and then go on to other shops, before returning home.

Jugglers

When travelling to a juggler the person stops a specified distance away from the juggler (the distance is specified in the objects file, and is stored in the objects data structure). They then stand and watch the juggler, for a time which is calculated from their own distractibility value (the higher it is, the quicker they will move on) and the juggler's distraction value (the higher it is, the longer they will stay). Further people arriving at the juggler can often push earlier people away, so each person regularly checks how far away they are. If the distance is too great they will head back towards the juggler again and then continue to wait for the rest of the time (they do not calculate a new waiting time).

Parks

When a person arrives at a park they are given a random number of new goals, consisting of random positions to visit in the park and random times to wait at each of these locations. This could possibly be improved on, but it gives a rough impression of someone exploring a park.

Bus-stops

Assuming the queue at the bus-stop is not already too big (in which case people simply move on to their next goal), people arriving at a bus-stop are given two new goals. The first is to move to a position specified by the position of the bus-stop and the number of people at the stop; this allows a fairly evenly spaced queue of people to form. The second is to stop and wait. The people will then wait until they receive further instructions (see earlier notes about updating bus-stops).

Buses

People only head towards buses when told to do so by a bus-stop. When they arrive at the bus, they tell the bus that they have arrived (which then decrements its 'number of people to wait for' counter) and then have a 'remove from map' goal added to the front of their goal list. There is currently no way for people to be generated by buses, so once a person has got onto a bus, they are permanently removed from the simulation.

Distractions

All objects in the city can have a distraction value. This is used to work out how likely a person is to (temporarily) put aside their current goals and head towards them. To prevent each person from checking for being distracted every frame (which would be computationally very expensive), people only check for being distracted when they arrive at a junction. When an object with a distraction value greater than zero is loaded into the city a line of sight (LOS) check is performed against all junctions within a certain radius. This radius is equal to 60 plus 100 times the object's distraction value, so the more distracting the object, the bigger the radius searched. If the object can be seen from the junction, then it is added to a list of 'local distractions' stored by the junction. This means that when a person arrives at a junction, they do not have to check against all objects in the city to find the nearby ones that they can see. Instead they simply go through the list at the current junction, checking for being distracted by any of those objects, safe in the knowledge that all of those objects can be seen from their current position.

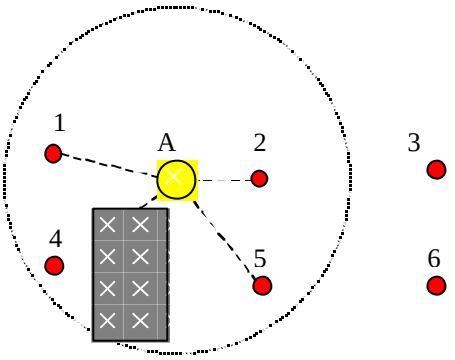


Figure 98: Object A's distraction value means that it affects all of the junctions in the given radius. Junctions 3 and 6 are too far away and ignored. Junction 4 cannot be seen, so is ignored. However, junctions 1, 2 and 5 add distraction A to their distractions lists.

The actual check for being distracted is simply done by comparing the distraction value of the object, the distractibility value of the person and a random number. If the person is distracted, then the object is added to the front of their goal list. The person then travels to the object with the normal LOT checks and junction navigation. To prevent a person from continually leaving an object and returning to it (having been re-distracted at the first junction they reach), the last object visited is stored and that object ignored when checking for being distracted. The person's distractibility is also reduced each time they are distracted, to increase their chances of actually reaching their final destination.

Updating a person's position

Once their current destination has been chosen the person has to travel to get there. This update occurs every frame, and is simpler to compute than the full route decisions which occur at junction and when arriving at other destinations. Firstly the current

direction the person is facing is calculated (simple vector algebra using their current position and their destination direction) and stored. The person's speed is multiplied by the time since they were last updated. If this distance is greater than the distance to their destination, the person is moved directly to their destination, otherwise it is multiplied by their direction vector and added to their current position.

The next thing to be calculated is any collision avoidance with other people in the city (assuming the user has not turned this option off). When a person is created their position is used to calculate which grid square they are currently in. The number of that person is then added to the list stored by each grid square. When the person's movement takes them out of the grid square and into another, the lists in both grid squares are updated accordingly. When a person comes to check for collisions with other people they simply go through the list of people in their own grid square (and any adjoining squares, if they are close to the boundary) calculating the distance between themselves and each other person. If this distance is too small (less than 5 units, ¼ of the width of a grid square), then the person attempts to move away by 1/(distance between the people)^2 in a direction perpendicular to their current heading (see fig 15 and 16). If this will take them closer to the person, then the direction is reversed.

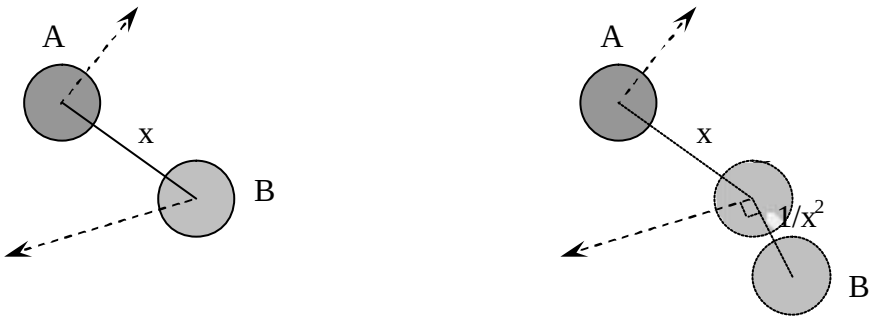


Figure 99: (a) B has moved too close to A… (b) … so B moves away from A

Before the person's position is updated their new position is checked to see if it is inside a building. If it is inside a building then the new position is rejected and the old position is kept (this prevents people from being pushed into buildings). The use of 1/distance^2 results in a much smoother avoidance of other people than a linear movement, but the value has to be capped (at two units in this program) to prevent people getting moved too far.

Originally the collision detection in this project did not use the grid squares to find nearby people, before checking the distance. Instead, each person checked their position against every other person in the city. This was very inefficient and significantly increased the amount of time spent updating the people (see section 3.5 for frame time details).

When people are just standing around (for example waiting for a bus, or watching a juggler) they still need to avoid other people (allowing people to pass through the bus-stop queue or reach the jugglers). This is achieved by calling the update person algorithm, but with a flag to prevent the person from walking closer to their goal.

Frame Rates

As mentioned in the introduction to this document, getting a reasonable frame rate is of importance to this project. The chart below gives some idea of the amount of time it takes to draw a single frame, depending on the number of people in the city at any one time.

These figures were generated on a 200MHz Pentium MMX, with 96Mb of RAM and a Voodoo2 3D accelerator, running Red Hat Linux 5.0. The simulation was given a minute so that the number of people in the city could build up, then the average frame rendering time (including processing the people) and the average people processing time measured over the following minute.

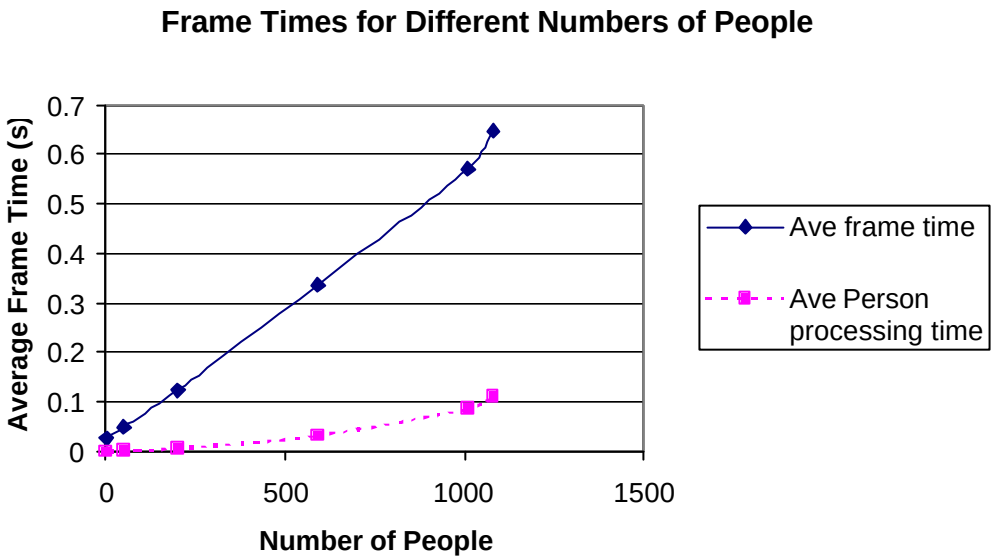


Figure 100 : Frame times for different city populations using gridcell collision

The number of people along the bottom is taken from the average number of people actually in the city during the time, and not the maximum allowed number of people. Due to the limited number of people that are generated each frame (at most one person per doorway per frame) - as the frame rate lowers, the number of people being generated, and the number of people being removed tends to balance out and prevent the maximum capacity being reached. This means that when the maximum number of people was set to 600, 1500 and 2000, the actual number of people in the city were 590, 1010 and 1080, respectively.

The graph shows a couple of important facts. Firstly the amount of time spent processing the actions of the people is fairly small compared to the time spent drawing them on screen (the distance between the two lines). This is true even with such a simple representation for the city and its inhabitants – a more complicated city would make the person calculations look even more favourable. Secondly the amount of time

spent rendering each frame increases linearly with the number of people in the city. If, however, the original collision detection is used instead of the improved version, the results are very different:

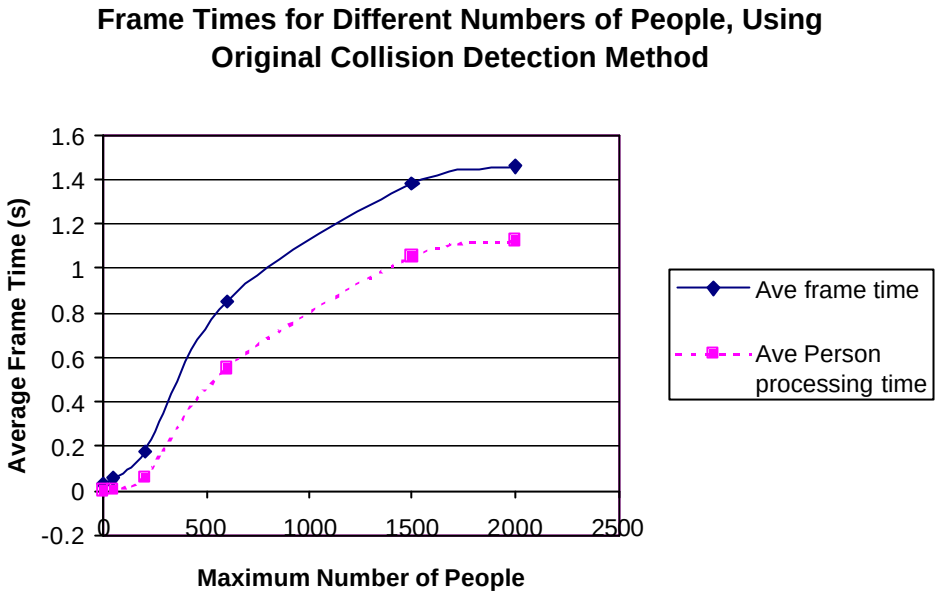


Figure 101: frame times using global collision detection

Unfortunately it is difficult to directly compare the two graphs. The second case is much more dependent on the maximum number of people allowed in the city (since every person checks against every other person for collision detection), so adjusting the values for the actual number of people in the city does not work. For reference, when the maximum number of people was set to 600, 1500 and 2000, the actual numbers of people in the city were 580, 690 and 650.

What can be seen from the graph is that the people processing is now a significant part of the rendering time, and that (until the limited number of people in the city counteracts the effect) the frame time increases exponentially with the number of people.

Rejected ideas

During the course of designing the project there have been a number of ideas that were looked at and then rejected. In this section there is a summary of some of the important ones.

Blocks of people

One of the first ideas that was considered was to simulate crowds of people as rectangular blocks, with the people moving around randomly within the blocks. The

decisions about which route to take would have been made for the block as a whole, so reducing the number of such decisions needed. The block as a whole could also be used as a simplified way of rendering the people from a distance, to increase the frame rate in large, well-populated cities. Unfortunately there were several problems with this, such as the difficulty of trying to move around corners and the difficulty of trying to get two such groups to pass each other. The other major problem is that this would lead to unnatural 'chunks' of people moving around the city.

Flocking Algorithm

Another early idea that was considered was the use of a flocking algorithm to define the movement of most of the people in the city. With this algorithm only a few people need to be fully simulated and the rest of the people merely tend to follow the nearest 'proper' person. This solution was never actually tried out, however in the final solution most of the time people are simply moving towards a point, with major updates only occurring when they arrive at a junction or an object. With the flocking algorithm they would need to continually work out the nearest 'leader', before updating their direction of heading. It appears that this would actually add to the computational overhead, rather than reducing the problem.

Navigation Styles

At one point it was hoped that several different styles of navigation would be included. These would range from the perfect routing algorithm (the one implemented in the final program), to wandering randomly in the hope of reaching the destination. The main problem with imperfect routing algorithms (such as always choosing the adjacent junction that was nearest to the destination) was how to deal with dead ends. If the person reached a dead end, they could simply backtrack and try another route, but if they found another dead end, they could easily backtrack and end up back at the original dead end. This would lead to an infinite loop and the only real solution to this would be for each person to store their entire route so far and then try to cross-off any routes that did not work. As the project was supposed to be about simulating the movement of crowds, and not to be a complex maze solving algorithm, this idea was dropped, and the perfect Dijkstra's algorithm used instead.

Line of Sight Checking

In an early version of the program LOS checking was carried out using the built in Maverik 'trace line' function, which traces a line through a 3D space and reports the first (if any) object that the line hits. Apart from this possibly being over complicated for simply spotting buildings in the way, it was also very difficult to adapt for use with the similar 'line of travel' algorithm, which needed to spot any roads, as well as any buildings, in the way. Eventually custom made functions, based on the underlying grid structure of the city, were used. These works quite well for the moment, but would

cause problems if an attempt were made to separate the simulation of the people from the particular city representation that is currently in use.

Possible improvements

There are quite a few improvements and changes that could have been made to the program. A lot of these changes are to do with making the program more portable and expandable. As the program is now it has just about reached the limit of what can be done without fairly major re-writes, but with more time it could hopefully be made into a library of functions which could be imported into other programs.

Implementation Improvements

One change that would need doing is to try to separate the 'physical' representation of the people from the control code more. For instance the LOS and LOT checking currently relies heavily on the underlying grid structure, as does the collision detection. This grid also forces every building or road to completely fill one or more grid squares, making them all multiples of 20 units in size. Although the actual layout of the city is loaded at run time, the data is stored in an array, so its maximum size is pre-set in the code (smaller cities simply leave areas empty). It would be good to find a data structure that could be more easily expanded to any required city size.

One data structure that would probably be a lot easier to change would be the objects data structure. Although the current method of using an array simplified the code whilst developing various ideas – allowing easy traversal of the objects and the ability to index a particular object without resorting to pointers – a linked list structure would be more flexible, allowing objects to be created and destroyed on-the-fly. This would also allow the number of people in the city to be adjusted, based on frame rate.

Another idea that there was not time to implement was to allow different object types to have functions registered with them. Instead of the current method of choosing the relevant reaction from a list of different object types whenever a person arrives at an object, or the object is updated, each object could have a pointer to a function which should be call in those events. This would have made it a lot easier to expand the program to include different types of objects, and would also make it a lot more useful as a library. This would hopefully result in an overall more object-orientated design for the code.

Distractions

The program currently is not very good at handling moving distractions. It is possible to operate them by regularly re-registering them with each of the junctions as they pass by (the 'register distraction' function also 'de-registers' any distractions that cannot be seen by a particular junction), but this is not perfect. Aside from being a fairly inefficient algorithm, it still means that a lot of people will miss certain objects (if they do not

happen to be at a junction when it passes by). Also if an object stops being distracting or its distraction value changes, it can take a while for the change to be noticed, as there is no way of updating the people already on their way to the object (or those standing watching it). Instead of completely rewriting the distraction-handling algorithm, it might be possible to add a secondary 'moving distraction' algorithm, and keep the first (more efficient) one for stationary objects.

On-screen Changes

There are also some features which could have been added, which would have had more of an on-screen effect, rather than just being behind the scenes. One such improvement would be to include groups of people in the city who are travelling around together. This would probably be best done using the flocking algorithm that was earlier rejected, but there would also be several other issues to address. How would the groups form? Would they be generated as a group, or would they randomly form in the map, or maybe members would seek each other out and band together? How would the groups split up? Would they all go into the same doorway and disappear, or would individual members go off on their own? The waiting of people could also be improved. At the moment if a person is told to stop and wait, they stop exactly where they are, and perhaps it would look better if the person moved towards one side of the pavement, out of the way of other people.

Another small problem that has not been adequately solved is the way that people walk across roads straight in front of buses. The only possible solution found, so far, is to put a check in the low-level 'position update' function. As bus avoidance is a fairly high level function, this code did not really fit there and so has been left out.

References and Bibliography

Black, P E, Algorithms, Data Structures, and Problems: terms and definitions, Version February 4th, 1999.

Booch, G., Analisis y Diseño Orientado a Objetos con Aplicaciones, Addison-Wesley, 1996

Borgwardt K H, Average complexity for determining the convex hull of randomly given points. Discrete and Computational Geometry, 17:79--109, 1997.

Bowers, J., Pettifer, S., Algorithms for Electronic Landscapes, in J. Bowers, S. Pettifer and M.Stenius (eds.), Understanding Connection, Transportation and Participation, Lancaster University Press, 47-77, 1998

Bowers, J., The Social Logic of Cyberspace or The Interactional Affordances of Virtual Brutalism, in A. Bullock and J.Mariani (eds), COMIC project deliverable 4.3 (deliverable to ESPRIT Basic Research Action 6225), Lancaster University, 1995

Brassard G, Bratley P, Fundamental of Algorithms, Prentice-Hall, Inc. A Simon & Schuster Company, 1996.

Brooks, M.R. Virtual Reality Interfaces for Complex Environments, Master's Thesis, Department of Computer Science, University of Manchester, 1994.

Catanese, A.J., Snyder, J.C., Introduction to Urban Planning, McGraw-Hill, 1979.

Ciucci, G., Del Co, F., Manieri-Elia, M., Tafuri, M., The American City: From the Civil War to the New Deal, The MIT Press, Cambridge, Massachusetts, 1979.

Clark, J.M., Hierarchical Geometric Models for Visible Surface Algorithms, Comm. ACM, 19(10), 547-54, Oct 1976.

Cook, J., Howard, T., Hubbold, R., Keates, M., Gibson, S., Murta, A., Pettifer, S. and West, A. MAVERIK Programmer's Guide. Advanced Interfaces Group, Department of Computer Science, University of Manchester, October 1998.

Cook, J., Howard, T., Hubbold, R., Keates, M., Gibson, S., Murta, A., Pettifer, S. and West, A. MAVERIK Functional Specification. Advanced Interfaces Group, Department of Computer Science, University of Manchester, October 1998.

Cruz-Neira C, Sandin D J, DeFanti T A, 'Surround-screen projection-based virtual reality: The Design and Implementation of the CAVE', Computer Graphics Proceedings, Annual Conference Series, volume 27, August 1993, pp 135-142

Foley, J.D., van Dam, A., Feiner, S.K. and Hughes, J.F., Computer Graphics-Principles and Practice, Addison-Wesley, Reading, Massachusetts, 1990.

Frécon E, Stenius M, "DIVE: A Scaleable network architecture for distributed virtual environments", Distributed Systems Engineering Journal, Vol. 5, No. 3, Sept. 1998, pp. 91-100, http://www.sics.se/~emmanuel/publications/dsej/

Frécon E., Stenius M., DIVE: A scaleable network architecture for distributed virtual environments, Distributed Systems Engineering Journal (DSEJ), 5 (1998), pp 91-100, Special Issue on Distributed Virtual Environments

Fructerman, T.M.J. and Reingold, E.M., "Graph Drawing by Force-Directed Placement", Software -- Practice and Experience, Vol. 21(11), pp 1129 - 1164, November 1991

H Edelsbrunner, “Algorithms in Computational Geometry”, “EATCS Monographs or Theoretical Computer Science”, v. 10, Springer-Verlag, Heidelberg, West Germany.

Hagsand O, "Interactive MultiUser VEs in the DIVE System", IEEE Multimedia Magazine, Vol 3, Number 1, 1996, http://www.computer.org/multimedia/mu1996/u1030abs.htm

Hagsand O., Interactive MultiUser VEs in the DIVE System, IEEE Multimedia Magazine, Vol 3, Number 1, 1996

Hubbold, R., Xiao, D. and Gibson, S. MAVERIK – The Manchester Virtual Environment Interface Kernel, in M. Göbel and J. David and P. Slavik and J.J. van Wijk (ed.), Virtual Environments and Scientific Visualization ’96, Springer-Verlag/Wien, 11-20, 1996.

Ingram, R., "Legibility Enhancement for Information Visualisation", PhD Thesis, Nottingham University, 1995

Kostof, S., The City Shaped: Urban Patterns and Meanings Through History, Thames and Hudson, London, 1991.

Lønningdal J C, Smart Unit Navigation, 1996. Available from http://home.sol.no/~johncl/shorpath.htm

M. Hoch, D. Schwabe, Group Interaction in a Surround Screen Environment, Computer Animation ´99, IEEE Computer Conference Series Geneva, May 26th-29th, 1999

Musse R, Thalmann D, ‘A Model of Human Crowd Behaviour: Group Inter-Relationship and Collision Detection Analysis’

Pettifer S, ‘An operating environment for Large Scale virtual reality application”, PhD thesis, The University of Manchester 1999.

Rubin, S.M. and Whitted, T., A Three-dimensional Representation for Fast Rendering of Complex Scenes, Computer Graphics, 14(3), 110-6, (Proc SIGGRAPH’80).

Schiffler A, Schwabe D, Knowbotic Research, The I0-dencies System, Design and Visualisation Techniques in Visualisation of Structure and Population within Electronic Landscapes, eSCAPE Deliverable 3.1, Mariani J. et al, editors, Lancaster University 1998

Sheng Liang, “The Java Native Interface – Programmer’s Guide and Specification”, Addison-Wesley, 1999

SICS, Dive Tcl/Tk Manual Reference http://www.sics.se/dive/manual/tclref.html

Snowdon D, Benford B, Greenhalgh C, Ingram R, Brown C, Fahlén L, Stenius M, A 3D Collaborative Virtual Environment for Web Browsing, Virtual Reality WorldWide'97, Santa Clara, CA, April 1997

Snowdon D., Fahlén L., Stenius M., WWW3D: A 3D multi-user web browser, WebNet'96, San Francisco, California, October 1996

Stroustrup, B., The C++ Programming Language (third edition), Addison-Wesley, 1997.

Sun Microsystems, "Jini Technology Architectural Overview", White paper, Sun Microsystems Inc., http://www.sun.com/jini/whitepapers/architecture.html

Sun Microsystems, "The Java3D API", White paper, Sun Microsystems Inc.

T. C. Zhao (University of Wisconsin-Milwaukee, USA) and Mark Overmars (Uretch University, the Netherlands), XFORMS Library, A Graphical User Interface Toolkit for X version 0.81, July 1996.

Thalmann D, ‘Simulation of the Human Crowd Based on the Group Sociological and Psychological Behaviours’, Ecole Polytechnique Federale de Lausanne,

The Marconi Company Limited, An Introduction to Macrospeak, January 1987.

Weiss, M.A., "Data Structures and Algorithm Analysis in Ada", Benjamin/Cummings, 1993, ISBN 0-8053-9055-3

Wright, R.S., Sweet, M., Programación en OpenGL, ANAYA – multimedia, 1997

Xiao, D., Interactive Display and Intelligent Walk-through in a Virtual Environment, PhD’s Thesis, Department of Computer Science, University of Manchester, 1997.

Yoder L., The Digital Display Technology of the Future, INFOCOMM ´97, June 1997, Los Angeles

Zahn, C.T., "Graph-theoretical Methods for Detecting and Describing Gestalt Clusters", IEEE Transactions on Computers, C-20, 1, January 1971, pp. 68-86