〔公　開〕

ＴＲ－Ｍ－０００７

An Interactive Visualization and Simulation Tool for Archaeological and Geographical Data

エドゥアルド　ネーテル
Eduardo NEETER

門林　理恵子
Rieko KADOBAYASHI

間瀬　健二
Kenji　MASE

１９９６　７．２９

ＡＴＲ知能映像通信研究所

An Interactive Visualization and Simulation Tool for Archaeological and Geographical Data

考古学データの対話的可視化システム

Eduardo Neeter （エドゥアルド ネーテル）

Rieko Kadobayashi （門林 理恵子）

Kenji Mase （間瀬 健二）

ATR Media Integration & Communications
Research Laboratories
（株）エイ・ティ・アール知能映像通信研究所

July 31, 1996

Contents

List of Figures

Abstract

We have developed VISTA system which is an interactive visualization and simulation tool for archaelogical and geographical data. The goal of this system is to support the archaeologist work in the formation and discovering of knowledge about the evolution of ancient japanese villages formation. Specifically, the system visualizes an ancient village using 3D Computer Graphics and simulates the changes over time, letting users set the term of each house and evaluate the setting. VISTA also allows users to walk through the village, which greatly helps archaeologist to study the village spatially and geographically.

Chapter 1

Introduction

1.1 The Goal

The purpose of this project is to support the archaeologist work in the formation and discovering of knowledge about the evolution of ancient Japanese villages formation. This project is embodied in the whole Meta-Museum project, carried out by Communication Support department of the Media Integration & Communications Research Laboratories.

Meta-Museum is a newly coined concept that seeks to enhance people's knowledge exploration experience in museums. The Meta-Museum project blends virtual reality and artificial intelligence technologies with conventional museums to maximize the utilization of the museum's knowledge base and provide an interactive, exciting and educational experience for visitors. The Meta-Museum offers rich and effective interaction with a museum's archive and the people behind them. Meta-Museum incorporates: (i) real, sensitivity-rich objects augmented by virtual objects with hyperlinks for accessing abstract knowledge spaces, (ii) visitor oriented and personal access, experience, and learning of exhibitions, (iii) facilitation of communication between experts (people behind the exhibitions) and non-experts (visitors), and (iv) personal artificial guide agents for the visitor.

In the Meta-Museum is pursued the creation of spaces where knowledge can be stored, evaluated, altered and personalized by different kind of users. This support the communication between experts and visitors, in the sense that the same space is shared by both of them, and the visualization and understanding of the knowledge is enhanced by novel tools.

Specifically, the VisuArch project consists of developing a system for Interactive Simulated Visualization of Archaeological and Geographical Data. The system incorporates the concepts of Knowledge Discovery in Database (KDD) and Data Mining. This process is performed as follows: (i) the real excavations data of an ancient Japanese village is provided as 2D blueprints, (ii) the data is classified and transformed as a set of constructive related pits, (iii) a Data Mining process is applied on this transformed data to get a feasible vestige marks set. Having obtained these previous results, the goal of the system is to provide a tool to test the hypothesis about the evolutionary sequence of the villages. This tool should adapt to different types of users (experts and non-experts). This is achieved carrying an iterative process over the hut's life terms data, allowing the user to visualize in an immersive mode the specified evolution, and interactively set and adjust this data to satisfy a valid pattern.

We have developed the VisTA system as a part of an interactive knowledge discovery system for archaeological data called VisuArch. VisTA assist visualization of the evolution of an ancient village letting users interactively set and modify the lifespan of each house in the village and allowing them to walk through the village.

The visualization uses 3D Computer Graphics techniques and a very simple and intuitive graphical interface. Through the interface the users can create new huts and set the life term and characteristics of each existing hut, in accordance with the valid vestige marks pattern discovered in the raw real excavations data.

Using the VisTA system, the user can interactively visualize how the the village formation process occurred, interpret the related evolution pattern and, if s/he deems necessary, return to any of the previous steps. At this point, the user can consolidate the discovered knowledge. The user can also check for and resolve potential conflicts with previously believed (or extracted) knowledge.

As experimental data, the excavation site of Otsuka [7] is selected. This site belongs to the Yayoi era. This era provides an interesting case of study because at this time the social and spatial organization of the Japanese ancient villages start to become cities. To visualize the evolution of the village in this era seems to be very helpful in the archaeologists work.

Other big advantage of a system as this one consist on the extension of the means that the archaeologist posses to study the sites after the excavation process is finished. Instead working with limited freedom and short range space visualization blueprints, the archaeologists can immerse in a virtual reconstruction of the site and take in account several factors (i.e. landscape) that are currently skipped at the moment of formulate hypothesis.

Summarizing this project consist of i) Creating a system to let the users test theirs hypothesis of the village evolution (knowledge discovering). ii) Using 3D Computer Graphics for easy visualization. iii) Providing immersive mode to be used in the Meta-Museum.

1.2 Overview

The rest of this report is organized as follows. In Chapter 2, making use of OpenInventor 2.1.1 as a 3D Graphic Programming Toolkit, an application structure is introduced. A brief review over OpenInventor and through the application requirements provides the ideal structure to be adopted. In Chapter 3 a Event Driven Simulation Engine is developed to manage the visualization process of the village evolution. In Chapter 4 is explained the Graphic User Interface used in the application, as well as how to make use of it. A section is devoted to explain how the Interface and 3D Graphics environments are merged. In Chapter 5 is dedicated to the 3D models created to assembly the virtual world, and the way as they are included in the OpenInventor environment. Chapter 6 explains the process of the Site reconstruction, and how the the initial data for the iterative process is obtained. Chapter 7 cites the future work that can be done in this context. Chapter 8 contains the conclusion of the project.

Chapter 2

OpenInventor Application Structure

This application was developed using OpenInventor 2.1.1, a 3D Graphic Programming Object-Oriented toolkit. The most of the application structure make use of OpenInventor classes and some extended sub-classes. This section is dedicated to explain this structure.

In this section, a brief description of the most important and keystone classes used in this application is given. Two new sub-classes have been created in order to satisfy some particular application's requirement. These two sub-classes are explained in detail. After this, the whole application structure used is discussed. In this section, first a review over the application functionality suggest an architecture to be adopted. Second, a review over the structure explain how the functionality required is accomplished.

2.1 Short description of the OpenInventor classes used

2.1.1 SoAlarmSensor:

This type of sensor can be used to schedule a one-time callback for some time in the future. The sensor is not guaranteed to be called at exactly that time, but will be called sometime after the specified time.

2.1.2 SoBoxHighlightRenderAction:

SoBoxHighlightRenderAction is a render action which renders the specified scene graph, then renders wire-frame boxes surrounding each selected object. Selected objects are specified by the first SoSelection node in the scene to which this action is applied. If an SoGetBoundingBoxAction applied to a selected object produces an empty bounding box, no highlight is rendered for that object. A highlight render action can be passed to the setGLRenderAction() method of SoXtRenderArea to have an effect on scene graphs.

2.1.3 SoCallback:

This node provides a general mechanism for inserting callback functions into a scene graph. The callback function registered with the node is called each time the node is traversed while performing any scene graph action. The callback function is passed a pointer to the action being performed and a user data pointer registered with the callback function.

2.1.4 SoCamera:

This is the abstract base class for all camera nodes. It defines the common methods and fields that all cameras have. Cameras are used to view a scene. When a camera is encountered during rendering, it sets the projection and viewing matrices and viewport appropriately; it does not draw geometry. Cameras should be placed before any shape nodes or light nodes in a scene graph; otherwise, those shapes or lights cannot be rendered properly. Cameras are affected by the current transformation, so you can position a camera by placing a transformation node before it in the scene graph . The default position and orientation of a camera is at (0,0,1) looking along the negative z-axis.

2.1.5 SoDB:

The SoDB class holds all scene graphs, each representing a 3D scene used by an application. A scene graph is a collection of SoNode objects which come in several varieties (see SoNode). Application programs must initialize the database by calling SoDB::init() before calling any other database routines and before constructing any nodes, paths, functions, or actions. Note that SoDB::init() is called by SoInteraction::init(), SoNodeKit::init(), and SoXt::init(), so if you are calling any of these methods, you do not need to call SoDB::init() directly. All methods on this class are static.

2.1.6 SoFieldSensor:

Field sensors detect changes to fields, calling a callback function whenever the field changes. The field may be part of a node, an input of an engine, or a global field.

2.1.7 SoGroup:

This node defines the base class for all group nodes. SoGroup is a node that contains an ordered list of child nodes. The ordering of the child nodes represents the traversal order for all operations (for example, rendering, picking, and so on). This node is simply a container for the child nodes and does not alter the traversal state in any way. During traversal, state accumulated for a child is passed on to each successive child and then to the parents of the group (SoGroup does not push or pop traversal state as SoSeparator does).

2.1.8 SoInput:

This class is used by the SoDB reading routines when reading Inventor data files. It supports both ASCII (default) and binary Inventor formats. Users can also register additional valid file headers. When reading, SoInput skips over Inventor comments (from '#' to end of line) and can stack input files. When EOF is reached, the stack is popped. This class can also be used to read from a buffer in memory.

2.1.9 SoNode:

This is the abstract base class from which all scene graph node classes are derived.

2.1.10 SoPath:

A path represents a scene graph or subgraph. It contains a list of pointers to nodes forming a chain from some root to some descendent. Each node in the chain is a child of the previous node. Paths are used to refer to some object in a scene graph precisely and unambiguously, even if there are many instances of the object. Therefore, paths are returned by both the SoRayPickAction and SoSearchAction.

When an action is applied to a path, only the nodes in the subgraph defined by the path are traversed. These include: the nodes in the path chain, all nodes (if any) below the last node in the path, and all nodes whose effects are inherited by any of these nodes.

SoPath attempts to maintain consistency of paths even when the structure of the scene graph changes. For example, removing a child from its parent when both are in a path chain cuts the path chain at that point, leaving the top part intact. Removing the node to the left of a node in a path adjusts the index for that node. Replacing a child of a node when both the parent and the child are in the chain replaces the child in the chain with the new child, truncating the path below the new child.

2.1.11 SoPerspectiveCamera:

A perspective camera defines a perspective projection from a viewpoint. The viewing volume for a perspective camera is a truncated right pyramid.

By default, the camera is located at (0,0,1) and looks along the negative z-axis; the position and orientation fields can be used to change these values. The heightAngle field defines the total vertical angle of the viewing volume; this and the aspectRatio field determine the horizontal angle.

2.1.12 SoPickAction:

This is an abstract base class for all picking actions. Currently, the only supported subclass is the SoRayPickAction.

2.1.13 SoPickStyle:

This node determines how subsequent geometry nodes in the scene graph are to be picked, as indicated by the style field.

Note that this is the only way to change the pick behavior of shapes; drawing style, complexity, and other rendering-related properties have no effect on picking.

2.1.14 SoRayPickAction:

This class performs picking by casting a ray into a scene and performing intersection tests with each object. The ray is extended to be a cone or cylinder, depending on the camera type, for intersection with points and lines. Each intersection is returned as an SoPickedPoint instance.

The picking ray can be specified as either a ray from the camera location through a particular viewport pixel, or as a world-space ray. In the former case, a valid camera must be encountered during traversal of the graph to determine the location of the ray in world space.

2.1.15 SoRotationXYZ:

This node defines a 3D rotation about one of the three principal axes. The rotation is accumulated into the current transformation, which is applied to subsequent shapes.

2.1.16 SoScale:

This node defines a 3D scaling about the origin. If the components of the scaling vector are not all the same, this produces a non-uniform scale.

2.1.17 SoSelection:

SoSelection defines a node which can be inserted into a scene graph and will generate and manage a selection list from picks on any node in the subgraph below it. Nodes are selected based on a current selection policy. Callback functions report back to the application when a path has been selected or deselected. The selection list can also be managed programmatically.

When handling events, SoSelection makes sure that the mouse release event was over the same object as the mouse press event before changing the list of selected objects. This allows users to mouse down on an object, change their mind and move the cursor off the object, then release the mouse button without altering the selection.

2.1.18 SoSeparator:

This group node performs a push (save) of the traversal state before traversing its children and a pop (restore) after traversing them. This isolates the separator's children from the rest of the scene graph. A separator can include lights, cameras, coordinates, normals, bindings, and all other properties. Separators are relatively inexpensive, so they can be used freely within scenes.

The SoSeparator node provides caching of state during rendering and bounding box computation. This feature can be enabled by setting the renderCaching and boundingBoxCaching fields. By default, these are set to AUTO, which means that Inventor decides whether to build a cache based on internal heuristics.

Separators can also perform culling during rendering and picking. Culling skips over traversal of the separator's children if they are not going to be rendered or picked, based on the comparison of the separator's bounding box with the current view volume. Culling is controlled by the renderCulling and pickCulling fields. These are also set to AUTO by default; however, render culling can be expensive (and can interfere with render caching), so the AUTO heuristics leave it disabled unless specified otherwise.

2.1.19 SoTransformation:

This is the abstract base class for all nodes that perform geometric transformations. It exists only to make it easy for applications to test whether a particular node is a transformation node (that is, is derived ¿from this class).

2.1.20 SoTranslation:

This node defines a translation by a 3D vector.

2.1.21 SoWriteAction:

This class is used for writing scene graphs to files. It contains an SoOutput instance that by default writes to the standard output. Methods on the SoOutput can be called to specify what file or memory buffer to write to.

2.1.22 SoXtViewer:

This is the lowest base class for viewer components. This class adds the notion of a camera to the SoXtRenderArea class. Whenever a new scene is specified with setSceneGraph(), the first camera encountered will be by default used as the edited camera. If no camera is found in the scene, the viewer will automatically create one. If the viewer type is SoXtViewer::BROWSER then the camera is told to view the supplied scene graph but is not added beneath that scene graph root. If the viewer type is SoXtViewer::EDITOR then the camera is added beneath the supplied scene graph root.

In addition to automatically creating a camera if needed, this base class also creates a headlight (directional light which is made to follow the camera), enables the user to change drawing styles (like wireframe or move wireframe), and buffering types. This base class also provides a convenient way to have the camera near and far clipping planes be automatically adjusted to minimize the clipping of objects in the scene.

Viewers allow the application to shadow event processing. When the application registers an event processing callback by calling setEventCallback() the viewer will invoke this callback for every X event it receives. However, unlike the render area, the viewer ignores the return value of this callback, and processes the event as usual. This allows the application to expand viewing capabilities without breaking the viewing paradigm. It is an easy way to hook up other devices, like the spaceball, to an existing viewer.

2.1.23 SoXtWalkViewer:

The paradigm for this viewer is a walkthrough of an architectural model. Its primary behavior is forward, backward, and left/right turning motion while maintaining a constant "eye level". It is also possible to stop and look around at the scene. The eye level plane can be disabled, allowing the viewer to proceed in the "look at" direction, as if on an escalator. The eye level plane can also be translated up and down - similar to an elevator.

2.2 Description of the new sub-classes

2.2.1 DataBuilding:

This class is created to store and handle the data related with each building. Objects of this class serve as a root of some of the scene graph branches. In this sense, how is needed that children nodes can be added, the class SoGroup is used as a parent class to inherit from it.

The new features added to this class are the fields needed to store the data associated to the building. Among these fields are the "Life Term" dates (build and decay years), the kind of building, and the vestige the building is related with. This is the information needed to accomplish the job of simulate the evolution of the village in terms of buildings.

The name and type of the fields are shown in Table 2.1.

Table 2.1: DataBuilding Fields

SoSFFloat	built;	(building year)
SoSFFloat	decay;	(decaying year)
SoSFString	kind;	(kind of building)
SoSFString	vestige;	(vestige identifier)

2.2.2 DataAnimation:

This class is created to store the data related with the whole simulation process. This class can be described as a leaf on the OpenInventor graph. In this sense, how these objects just have to be added to the graph at the end of a branch, na any kind of special feature is needed. The class SoNode is used as a parent class to inherit from it.

The new features added to this class are the fields needed to store the data associated to the simulation process. Among these fields are the "Simulation Term" (beginning and ending years), "Visualization Term" (beginning and ending years), "Visualization Duration", "Visualization Duration Limits" (lowest and highest values). This provide the parameters used to fix the context in which the simulation and real-time visualization process will be performed.

The name and type of the fields are shown in Table 2.2.

Table 2.2: DataAnimation Fields

SoSFFloat	periodBegin;	(Simulation beginning year)
SoSFFloat	periodEnd;	(Simulation ending year)
SoSFFloat	animationBegin;	(Visualization beginning year)
SoSFFloat	animationEnd;	(Visualization ending year)
SoSFFloat	duration;	(Visualization duration)
SoSFFloat	highest;	(Visualization highest limit)
SoSFFloat	lowest;	(Visualization lowest limit)

2.3 OpenInventor Application Structure

The structure consists of a OpenInventor's nodes graph, where a hierarchical correspondence between the nodes is established in order to achieve the application goals in the most effective manner.

First is needed to identify which objects will play a main role in the visualization, in order to provide a special treatment. Among these objects are the BUILDINGs, which are the protagonists of the visualization. Each building require of its own VESTIGE mark on the land, which serve as interface as well as a reference point. As interface because through it the user define the existence of a building, and a reference point because over the vestige mark is where the building is situated while its existence term. The third object that play an important role in the visualization is the LAND, or the "world" where all the development of the village occurs.

Is important to distinguish between these objects that land and vestiges are statics over the time This is, they don't change at all while the visualization of the simulation is played. They are always the same for the user view. Also is needed to notice that many kinds of buildings can be incorporated and coexist in a same village formation. Of course, each kind of building must posses its related king of vestige.

The buildings are the objects that change in the scene over the time. For this reason is need to handle time values, as well as location for each building. To handle this information a subclass is created. As an additional features, this class provides the fields to store and handle the information of each building. Another sub_class created is the one which contains the information related with the visualization process by itself. For more details on those two classes look at the section 2.2.

There are some special OpenInventor classes that provide functionality to the application, as the SoSelection class. This class enable the application to handle mouse pick actions in the sense of let the user select objects on the scene. What the user can do with a selected object is not discussed at this point, but the fact that some special treatment can be performed over some objects is important. This class allows to handle a selection list, and execute callback actions when a selection is done. One callback action that can be quoted here is the rendering of a bounding box around an object. It is used to identify clearly which object has been picked and selected. In the same way, the use of nodes that specify which kind of picking policy belongs to any node in the subgraph below provide more control over this structure.

Cameras play a very important role in this context, they provide the "eyes" to the user. Cameras are used to view the scene. As two different points of view are provided to the user, two cameras are needed. Each camera belongs to a different version of the scene. By this reason, two different SoSeparator nodes are included as scenes roots, where each one possess its own particular camera, but both of them looks at the same world.

As mentioned above, to assemble the world where the evolution's simulation is performed, the used objects are the "land" and the "vestige marks". In this sense, only a land exists, but many vestige marks can appear in one scene at the same time. The buildings are no fixed on the world, the presence on the scene change over the time. This provide the sense that they have to be handled separately inside the scope of the visible world. A SoSeparator to handle the land is introduced, this is just for the static objects that conform the physical world. Another SoSeparator is introduced to handle the buildings on the scene. This node is which receive the inclusion of buildings into the scope of the viewers at the time it is desired they appear on the user view. Both of them should be under the SoSelection node, because the user should be allowed to select either vestiges or buildings in accordance to the actions s/he wants to apply. The land should be particularly treated, because to select the land do not make sense, only as a deselection method.

To avoid having many copies of the same 3D vestige model, instances from the same model are created. Each instance handle its own properties about scale, translation and rotations. Each of the instances has to be identified as a unique vestige. Also, each vestige has one kind of vestige that it is associated to. By this, multiple transformation nodes will exist on the graph, providing particular information to each vestige mark, but only one node containing the 3D model of each kind of vestige is needed. Similar situation happens with the buildings. Is not needed to have many copies of the 3D models as buildings belongs two a simulation process. What is needed is a particular DataBuilding node for each building. This node must be unique and identify each building. The Building's 3D model can be instantiated from a single shared node, and the transformation on the space can be adopted from the transformation that the related vestige posses.

Just integrating and sharing some nodes, the structure of the application can be highly optimized. Multiple copies of the same nodes are avoided, an a strong correspondence between related nodes can be achieved. For example, the position where a building is going to be placed is obtained from the values of the vestige, which the building was related to.

Another particular object is the representation of the user in the world. This object belongs to the world, and must be visible for every pont of view. For this reason, this object is under the scope of the highest common node from the two scenes.

This provided and extensive overview of the application, and how the required functionality suggest an appropriated structure. As a second part, a quick review over the graph created, following each node as it is incorporated to the tree can help to clarify any weak point on this OpenInventor structure subject. In order to help this explanation, the Figure 2.1 and the Figure 2.2 are showed.

Every node has a method to specify a name. This name consist of one string of characters that identify the node. Having each node a name, it can be locate in the graph using this keyword. In the used graph (Figure 2.1 and Figure 2.2), almost every node has a name, which is unique. There are some exceptions in this rule. Some nodes are not provided a name, because following the structure formation it is really not needed.

Below, performing a travel over the graph, each node found is explained. When a node name is written between [ ] means that the name is application dependent. In fact, some of those names are identifiers obtained from the interaction with the user. Is important to mention that should be avoided to name two different nodes with the same name, in order to maintain the integrity of the tree.

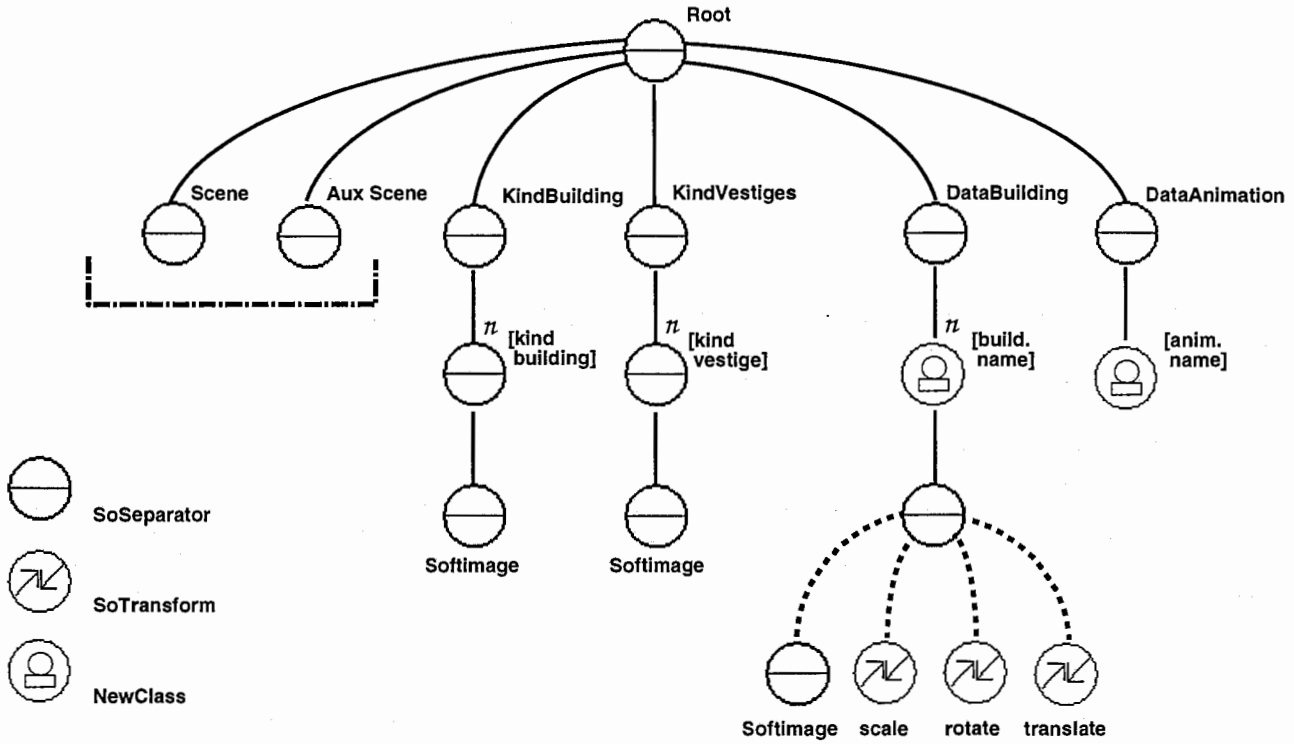

Figure 2.1: OpenInventor Application Graph

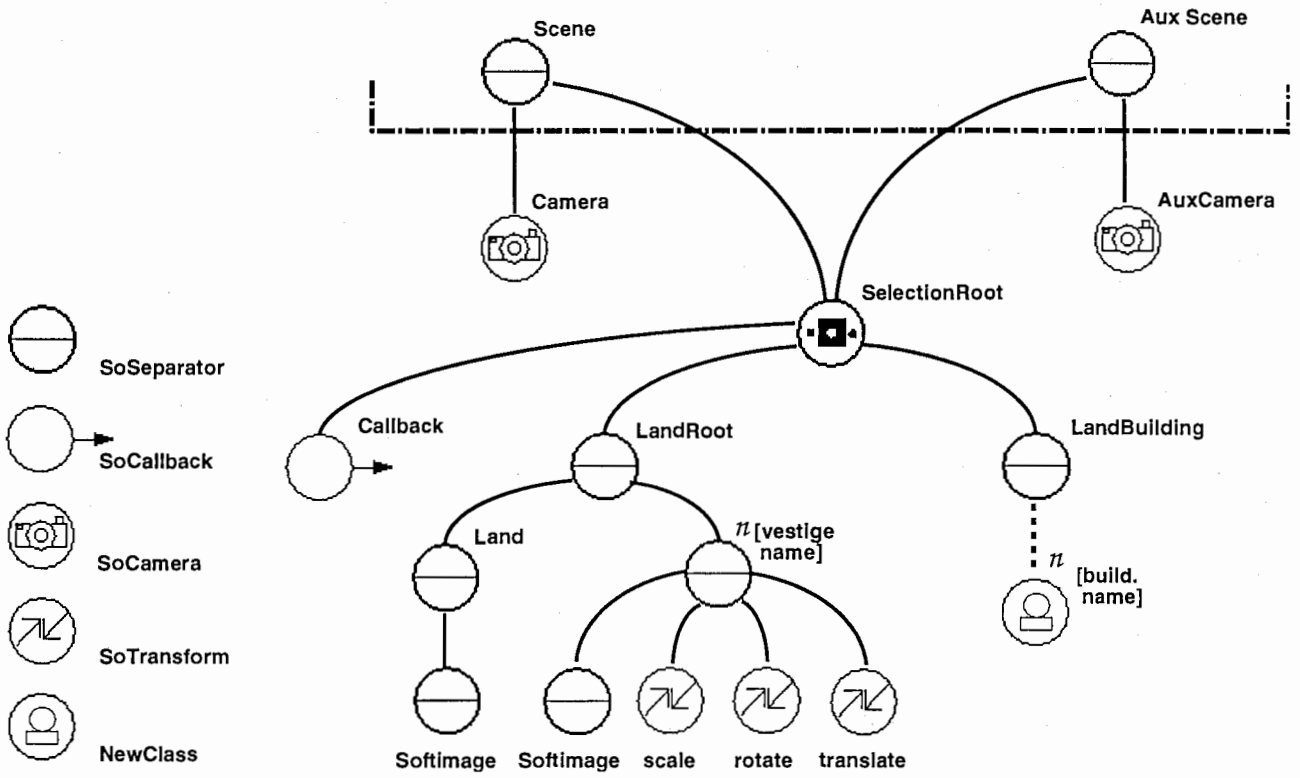

Figure 2.2: OpenInventor Scene Sub-Graph

2.3.1 Root:

Is a SoSeparator node that acts as the graph root. Its only job is join the different parts of the structure as a same graph, in order to handle it as a whole SoDB object when actions with files are required.

2.3.2 DataAnimation:

Is a SoSeparator which goal is cover the "DataAnimation" objects. The existence of this node is just to maintain the same hierarchy in all the graph.

2.3.3 [AnimationName]:

Is the "DataAnimation" object that contains all the information about the visualization process. The name provided to the node is the same the user assign as filename at the moment of save the file the first time. Also is updated when the user save the file with other name.

2.3.4 DataBuilding:

Is the SoSeparator node used to group all the "buildings". Under this node are located all the nodes related with the buildings that play a role on the simulation process. The number of children that this node handles grow in order to satisfy the user's requirements. This number is not fixed and is limited only by the memory available.

2.3.5 [BuildingName]:

This is the "DataBuilding" node that characterize each building. It contains the data related with each building. There are one different node for each building. For this reason, the number of this kind of node is undefined and change in accordance to the user actions. The building name provided by the user at the creation time, or changed when a building is edited, is used as identifier for each node. This way, each building data is easy to locate using the building name as a keyword on the search over the graph.

Under this node a new SoSeparator node is added as children. To this new node a name is not provided due it is not needed. This SoSeparator is used to group as a one object the components of the building graphic representation. This is done linking existing nodes that assemble the particular building. Those nodes are the 3D model correspondent to the kind of building and the transformations related with the associated vestige mark. The last nodes are only added as children, a logic link. No copy of those nodes is done.

2.3.6 KindVestiges:

This SoSeparator node is used to group and handle the 3D models of the different kind of vestige marks that can exist in a certain context. Different kind of vestige can be added to the application at the moment of the original database creation. (For an explanation of how the vestiges are incorporated, see Chapter 6)

2.3.7 [KindVestigeName]:

This is a SoSeparator node which is used to receive and handle the 3D model of the vestige mark. Also this node is given the kind of vestige's name. In this way, each kind of vestige has his own unique name that can be used as keyword to search this node in the graph. This name is obtained adding the string "_Vestige" to the kind of building's name that the vestige is related to. Under this node is contained the 3D model, this as obtained from an additional file. (For a explanation of how this model is incorporated and how the node name is fixed, see the Chapter 6).

2.3.8 KindBuildings:

This SoSeparator node is used to group and handle the 3D models of the different kind of buildings that can exist in a certain context. Different kind of buildings can be added to the application at the moment of the original database creation. (For an explanation of how the buildings are incorporated, see Chapter 6)

2.3.9 [KindBuildingName]:

This is a SoSeparator node which is used to receive and handle the 3D model of the buildings. Also this node is given the kid of building's name. In this way, each kind of building has his own unique name that can be used as keyword to search this node in the graph. This name is provided at the time the original database is created. Under this node is contained the 3D model, this as obtained from an additional file. (For a explanation of how this model is incorporated and how the node name is fixed, see Chapter 6)

2.4 SCENES

From this pont, the goal is to construct the scene graph instead how to store and handle data inside the application.

From here the world that the user can see is assembled. Two different views over the same world are provided to the user. First each point of view is explained and after that, the world structure by itself is also explained.

2.4.1 AuxScene:

This SoSeparator node is used to define the scope of the scene that is desired to show to the user as an auxiliary view. The "world" that this scene dominate is shared with the other Scene root explained below. The order of the explanation is broken and this world is explained later, due the fact that it is common child of two to different parent nodes, but the particular child to this node is explained right now.

2.4.2 AuxCamera:

This SoPerspectiveCamera node is added to get the rendered image of the world from an auxiliary point of view. The user can handle two views at the same time. This AuxCamera node provides the way to look from this second pont of view.

2.4.3 Scene:

This SoSeparator node is used to define the scope of the scene is desired to show to the user as the main view. As mention above, the "world" that this scene cover is shared with AuxScene node. this is made this way because there are only one world, even if there are two different views. Also this scene has its own particular child that is not shared between scenes.

2.4.4 Camera:

This SoPerspectiveCamera node is added to get the rendered image of the world from the main point of view. The user can handle two views at the same time. This Camera node provides the way to look from this main pont of view.

2.4.5 SelectionRoot:

This SoSelection node is incorporated in order to define the scope of the world, and provide the application the ability to pick and select almost all the objects that intervene in the world formation and simulation process. Every node which path go through this node can be selected. This is that every descendent node belonging to this node subgraph is already able to be picked and selected.

2.4.6 BuildRoot:

This SoSeparator node groups and handles all the buildings that are desired to appear on the user's views. All the building declared previously by the user exist in the graph, but not are visible to the camera because the buildings are not under the camera scope. This node provide the path to those buildings that should be visible in a specific time. Those building that want to be showed have to be added as a children to this node. The buildings are incorporated to the scene adding the related "DataBuilding" node as child to the BuildRoot node, and are disincorporated from the scene removing this relation. The mentioned representation of the building is just logically linked, no copy of those nodes is made.

2.4.7 LandRoot:

This SoSeparator node groups and handles all the nodes that conform the environment model, this is land and sky, and all the vestige marks. All the vestige marks are in the same hierarchy level that the land. The vestige marks are selectable, while the land is the default selection to clear any previous selection.

2.4.8 [VestigeName]:

This SoSeparator node groups all the related node for each particular vestige mark. It contains all the transformation related with the vestige as a link to the 3D model of the related kind of vestige. This is assembled at the database creation time (See Capter 6).

2.4.9 Land:

This SoSeparator node handles the 3D model of the land and the sky. It is the default world when no any vestige exist.

2.4.10 Human:

This SoSeparator node handles the 3D model used to represent the user in the world. It posses a SoTransformation node that is adjusted each time that the "Camera" of the "Scene" (main view) is moved (each time that the user moves in the world). This way there is a exact concordance between the point of view of the main view and the location of the representation of the user in the world. Also a SoPickStyle is added to this SoSeparator. This SoPickStyle is set to be "UNPICKABLE". Doing this, the human cannot be selected for the user to perform any action over it.

Chapter 3

Simulation Engine

At the beginning, the goal of the engine consisted on executing different kinds of actions in accordance with the events involved in the simulation (build and decay buildings). In this context is needed to handle the event's time information related with each building. The effect of each possible event consists in the way they affect the scene the user sees. In this sense, when an event "build" occurs, the graphical representation of the related building should appear in scene, providing the user the idea of existence of this building in a given time. In the same way, when a building decays, its graphical representation should disappear from the scene, providing the user the sense the building is not existing anymore.

3.1 1st. Stage: Sequential execution of actions in accordance with time values

The method used to execute actions related with each event make use of OpenInventor "Alarm Sensors. Alarm Sensors are objects that trigger a specific callback routine at a specific time. For each one of the buildings defined two Alarm Sensor objects are created, one for handling the "build" event and the other one for handling the "decay" event. At the creation time of each Alarm Sensor object, the kind of event has to be taken in account to decide which callback function assign. In this way, when an Alarm Sensor is created to "build" a building, the callback function "builtBuildingCB" is assigned as action to be invoked, providing the related building as a parameter for the callback function. The same procedure is done for decaying a building, but at this time the "decayBuildingCB" is assigned instead, also with the related building as a parameter. The description of these callback functions is provided in section 3.4.

The process to assemble the simulation consists on reviewing all the buildings already declared under the "DataBuildings" node (all the buildings involved in the process). For each one of them, the two Alarms Sensors are created in the way described above. These Alarms Sensors are registered in a dynamic list, in which each single record have a pointer to each Alarm Sensor. Also in these record is stored the time value for the occurrence of each related event. This time is obtained from the time specified in the "DataBuilding" subclass node. The structure of the "DataBuilding" subclass is explained in section 2.2. When all the "Alarm Sensors" are created, a quick travel over the list is performed, scheduling each "Alarm Sensor" at its respective time value.

Once completed this process, the Alarm Sensors do the rest of the work. They trigger the correspondent callback functions that add and remove buildings from the scene at the specific time these actions should occur. Performing this, the buildings appear and disappear from the user's view through the time, in accordance with the life-term value previously specified.

3.2 2nd. Stage: Providing control over the simulation clock

In this stage, the goal is to improve the previous engine in order to get control over the simulation "clock". Having control over this value, it is easy to pause the simulation and start it again from any point, jump between events instantaneously in either backward style or forward style. Also is possible make changes in the building's data and visualize them (add new buildings, change building's life-term values, and even delete some of them) and being known the time value when the simulation was paused, the current state of the village can be reconstructed and updated instantaneously.

In order to achieve this goal, several changes have to be made in the previous simulation engine architecture. First, the list must be sorted in accordance to the occurrence time of each event. Also is needed to include the concept of "relative time" for each event, that is no more than the difference between each absolute event's times. Moreover, the time proportion respect to the "Animation Term" is included. The list must be double-linked to allow going forward and backward.

Following the premises cited above, the routine to add records to the list is improved to maintain the list sorted all time. Each new record is included in the list following the criteria of going forward in the list until finding a record which time value is bigger than the new one, and then insert the new record just before it. Furthermore, a new field is added to the record. This field contains the "relative time" for each event, and is filled once that the whole list is assembled. The procedure to fill this new field consists on travel over the list and calculate the difference between each event's time value and the time value of the prior event. The "relative time" for the first event is set to 0. This is explained in more detail in section 3.3. Also another pointer field is added to the record structure in order to allow the navigation through the list in both directions, forward and backward.

Using this scheme, the method of schedule all the events at the beginning using the absolutes values is rejected. A new method is used. This method is based on having a pointer to one record of the list, which is used as a "clock-tracker" that indicate which event is being processed. This "clock-tracker" is initialized pointing to the first record in the list, and then have it traveling over the list, sequentially scheduling events. The "time from now" used to schedule each Alarm Sensor is the previously "relative time" calculated for the related event. This way, each event is triggered in real time in accordance to the last event occured, keeping the time proportion in the execution of each event's action. Once that each event occurs, the "clock-tracker" is updated to handle the next event, scheduling its related Alarm Sensor, and so on until it reaches the end of the list.

This scheme needs of only one Alarm Sensor scheduled at time. For this reason a pause action can be accomplished. To pause the simulation, it is only needed to suspend the future event (cued event). This is made using a pointer to the actual scheduled "Alarm Sensor" (using the "clock-tracker" and the list) and unscheduling it. This paralyze the execution of the simulation. Simply, no more actions are executed and the village state is untouched, it stay as it was at that time.

When the animation is paused, the "clock-tracker" can we altered manually, going forward and backward in the simulation time. The "jump" in time allowed are restricted just to reach the following or previous event that alter the state of the village. In this sense, the user can move in time between "build" and "decay" events. The method to perform this action consist in travel over the list in the required direction until find the next event of interest (build or decay). Once that this "time point" is reached, the related action with the event is performed. For example, if the simulation is pause at certain time and a "Go Forward" action is required, two different actions can be performed, it depends on the next event found. If the next event is a "build" event, then the related building is added to the scene. If the next event found is "decay" event, then the related building is removed from the scene. In both cases the "clock-tracker" is leaved in the time just after the event occurs. Contrary, if the "Go Backward" action is required, the two different possible actions are the opposite to the described above. If going backward the first event found is "build" event, the related building is removed from the scene. If the event found is "decay" event, the related building is added to the scene. In this case, the "clock-tracker" is leaved in time just before the event occurs. This is illustrated in the Figure 3.1.

3.3 Getting the times

This section is dedicated to explain the way as the event times is obtained. For this purpose is needed to define the concept of "Simulation Term", "Visualization Term", "Visualization Duration". The "Simulation Term" is the whole period in which the archaeologist wants to test his/her hypothesis. Consist of the two dates that serve as constrains for every "build" and "decay" events' time values. The "Visualization Term" is the sub-period of "Simulation Term" which the user wants to visualize. This define the starting and ending point of the animated simulation. The village state at the starting time is also simulated and obtained, but the evolution process until that time is not visualized. It is obtained instantaneously when the user set the respective starting value. The "Visualization Duration", as the name by itself explains, is the duration that the user expect the visualization of the simulation last. It is expressed in number of seconds.

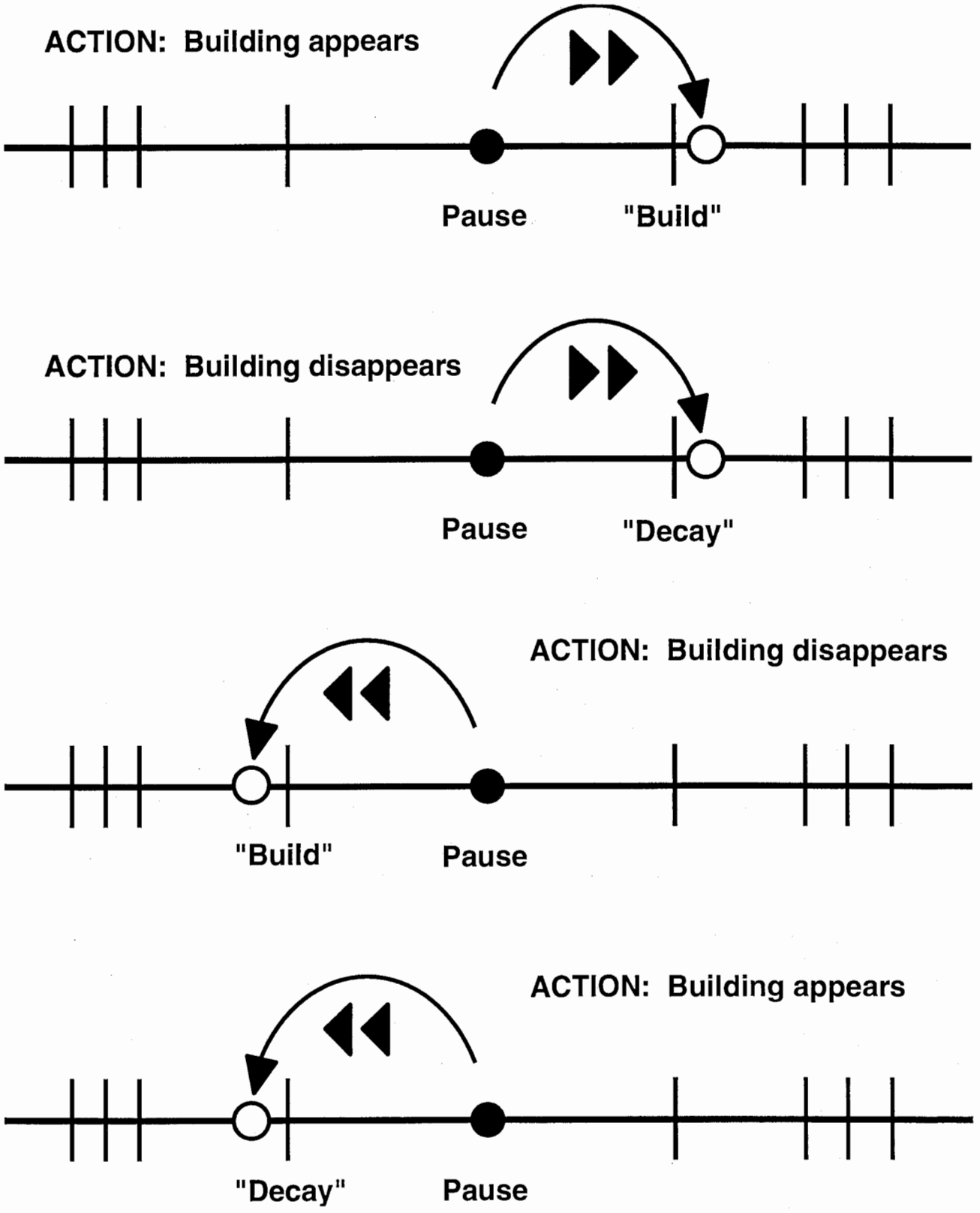

Figure 3.1: Time Tracker "jumps" in time

$$\text{Factor} = \frac{\text{"Animation Duration"}}{\text{"Visualization Duration". end} - \text{"Visualization Duration". begin}}$$

Figure 3.2: Time scale converter factor.

$$i > 0:$$

$$\text{Real\_time(i)} = [(\text{Absolute\_time(i)} - \text{Absolute\_time (i-1)}) * \text{Factor}] - \text{Real\_time (i-1)}$$

$$i = 0:$$

$$\text{Real\_time(i)} = 0$$

Figure 3.3: Real time difference among events

These three key terms are used to calculate the real time values the events are going to occur. As mentioned before, the key to reconstruct the sequence, and visualize the occurrence of the events in real time, is to have the real time value that elapse between each event in a correct proportion in the "Visualization Duration".

In this sense, a transformation is applied over the event's absolute times in order to get their respective values in term of the "Visualization Duration". This transformation consist in getting the factor that converts values from the "Animation Term" into values in "Visualization Duration" (convert absolute years values into real-time seconds). This "factor" is illustrated in Figure 3.2.

Once this factor is obtained, the interesting value is the time difference between the event's times. Having this difference in absolutes values, and multiplying it by the transformation factor, the real time intervals of time between the occurrence of each event is obtained (Figure 3.3). The work is done. At this point is very easy to reconstruct in real time the evolutive sequence of the village.

3.4 Buildings in the scene

In this section it is discussed the way as the buildings are incorporated and disincorporated from the scene as response from the occurrence of the simulation events. Here the OpenInventor application structure is cited frequently, and all the explanation assumes that this is a topic already understood. The OpenInventor application structure is explained in Chapter 2.

The way as each building is incorporated to the scene consist on add the SoSeparator object containing the graphical representation of each building, with all the required transformations (translation, rotation and resizing). The "builtBuildingCB" is in charge of add the related building to the SoSeparator "BuildRoot" node. When the recently built building is added as child of "BuildRoot", it enters into the scope of the scenes nodes, making it visible to the scene's cameras. When it occurs, the building is rendered on the user view, and the sense that the building existence starts is achieved. The building is virtually constructed.

Similarly, but the opposite way is how each building is disincorporated from the scene. It consists in removing the building from the "BuildRoot" node. Once executed this action, the building is no more under the scene's

scope and the representation of the building disappear from the user's view. The sense of the building's existence is over. This last action is performed by the "decayBuildingCB". The building is virtually destroyed.

These two callback functions are triggered by its respective "Alarm Sensor", following the relative real time values previously calculated for each event. These four points describe how the real-time visualized simulation is done.

Chapter 4

Graphic User Interface

The application's graphic user interface is composed by four different windows. Two windows used to interact and explore within the environment and visualize the simulation, and other two windows to control the application's functionality. In that sense, the interface can be classified as viewer windows and control windows. This section explains both kinds of windows, at the same time that indicate how to make use of them in order to manage the application. Also provides an explanation of how is the graphic interface environment joined with the 3D graphics environment. A view of all the windows is shown in Figure 4.1

4.1 Control Windows

As mentioned above, there are two windows used to control the functionality of the application. To understand why the control interface is dived in two windows, is needed to realize that there are two main kinds of commands that the user can executed, based on the actions that those commands trigger on the application. These two kinds of actions are: the actions to handle the files, as well as the looking of the application and special options, which controls are contained on the "Console", and the actions that affect the simulation process by itself, which controls are contained on the "Simulation Control Panel" window.

4.1.1 The Console Window

In the "Console" window can be found four sections and one button. The first section is the one related with the managing of the "File". The second one controls the display of the "Auxiliary Viewer", the third one controls the "Track Human" functionality, and the fourth one controls the "Stereo Vision". The last command available is the "quit" button. This window is showed in Figure 4.2

The "File" Section

It contains three buttons, each for the fundamental actions the application can perform with the files: "NEW" creates a new file, "OPEN" opens an existing file, "SAVE" save the actual file on disk.

When the "NEW" button is pressed, the process for create a new file (personalized database) is initiated. At this time, the user is asked to select one database to use as a starting point of his/her work. Using the "Open Database" window (Figure 4.3) the user select one of the available database. This window is a standard "open file"-like window, where a filter, directories, files and selection is shown. Also provide the buttons to accept the selection, change the filter and cancel the current operation. After a database is selected, the user is asked to introduce the term s/he wants to work with in the simulation. The "Term of the Simulation" window (Figure 4.4) is displayed. This window contains two different fields. One for the beginning of the term (beginning year) and other for the ending of the term (ending year). Also contains two buttons, one to accept the values already introduced, and other to cancel the process. The last one aborts the creation of a new personalized database, and return the application to the state how it was before.

After this process is finished, a new copy of the village database is created, and the user can start to set its own personalized data on it.

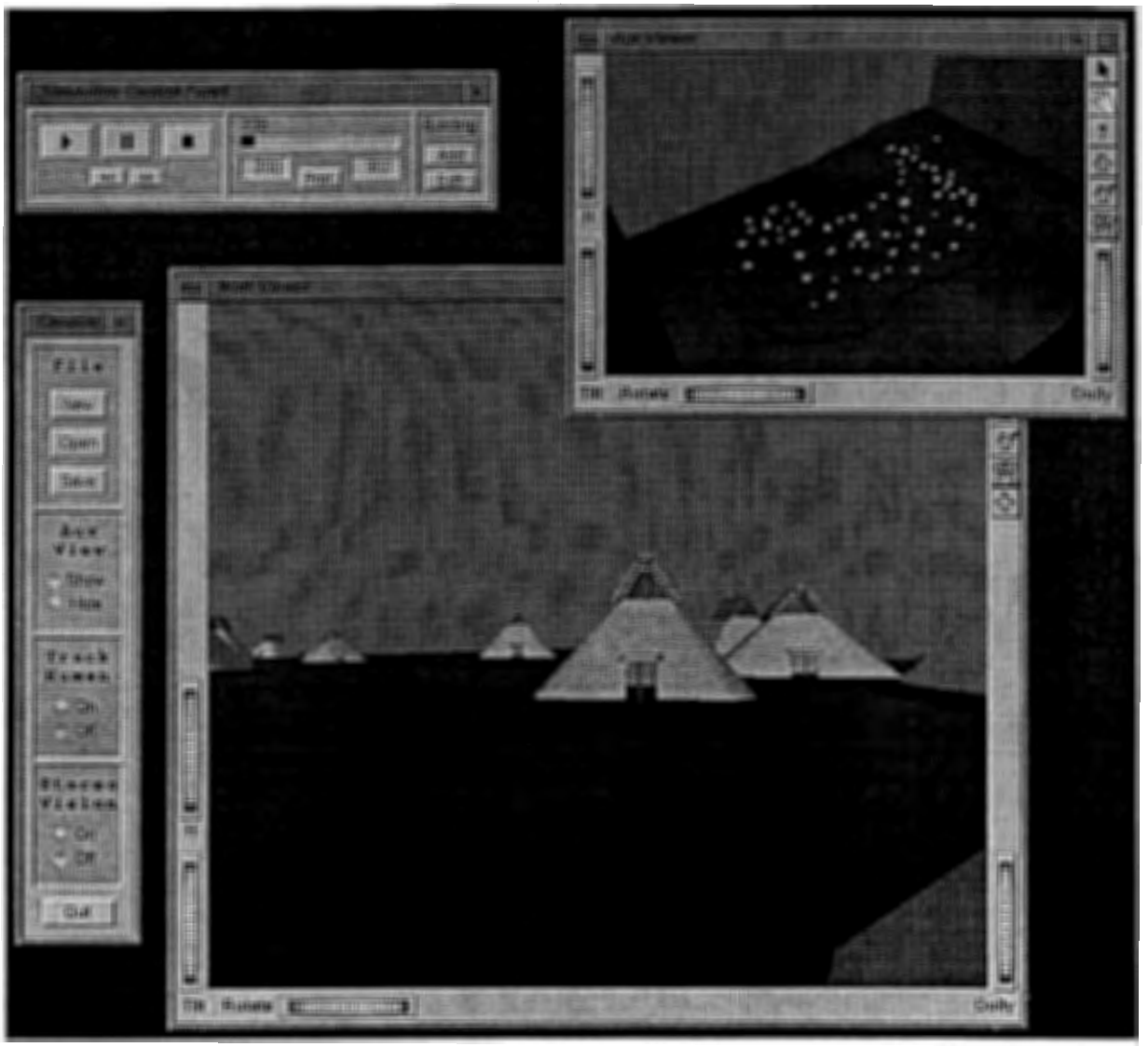

Figure 4.1: A view of all the application's windows

Figure 4.2: Console Window

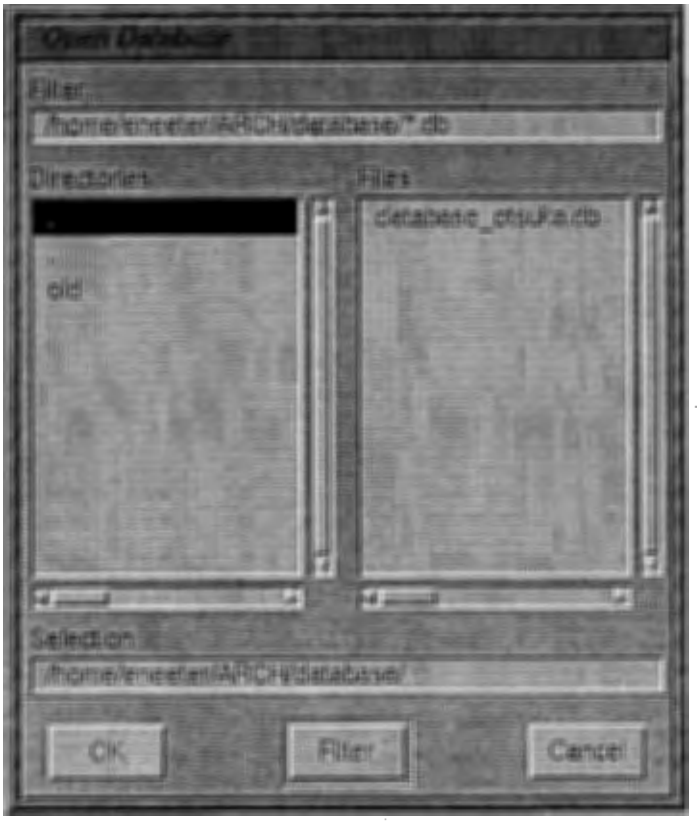

Figure 4.3: Open Database Window

Figure 4.4: Term of Simulation Window

Figure 4.5: Open File Window

When the "OPEN" button is pressed, the process for open an existing file (personalized database) is initiated. At this time, the user is asked to select one file. Using the "Open File" window (Figure 4.5) the user select one of the available files. This window is a standard "open file"-like window, where a filter, directories, files and selection is shown. Also provide the buttons to accept the selection, change the filter and cancel the operation. The last one aborts loading the personalized database, and return the application to the state how it was before.

When the "SAVE" button is pressed, the process for save the actual file (personalized database) is initiated. At this time, the user is asked to set the file's name. Using the "Save File" window (Figure 4.6) the user can either select one of the available files and overwrite it or assign a new filename. This window is a standard "save as"-like window, where a filter, directories, files and selection is shown. Also provide the buttons to accept the selection, change the filter and cancel the operation. The last one aborts saving the personalized database, and return the application to the state how it was before.

The "Aux View" Section

It contains two radio buttons. These buttons allow the user to select between "Show" or "Hide" the "Auxiliar Viewer". The "Auxiliar Viewer" is discussed in section 4.2. By default, the auxiliary viewer is hidden (not showed). When the user press the "Show" button, the viewer is opened (showed). When the user press the "Hide" button, the viewer is closed (hidden).

The "Track Human" Section

It contains two radio buttons. These buttons allow the user to manage the "Track Human" functionality, selecting between turn it "On" or turn it "Off". This functionality consist of point the camera use in the "Aux Viewer" at the position of the "human" in the virtual world. By default, this option is "Off". When the user press the button "On", the camera point at the "human" and keep pointing at it. If the user moves on the virtual world, the camera related to the "Aux Viewer" will adjust automatically to track the "human" position. When the user press the "Off" button, this options is disabled, and the "Aux Viewer"'s camera is leaved pointing to the current direction.

Figure 4.6: Save File Window

The "Stereo Vision" Section

It contains two radio buttons. These buttons allow the user to manage the "Stereo Vision" functionality, selecting between turn it "On" or turn it "Off". This functionality consist of setting the "Walk Viewer" (section 4.2) in "Stereo Vision". This functionality makes use of the Silicon Graphics "Crystal Eyes" system. By default, this option is "Off". When the user press the button "On", the actions and commands needed to set the "Stereo Vision" are invoked. When the user press the "off" button, this options is disabled, and the application is set to work normally again.

The "Quit" button

Pressing this button the user quits the application.

4.1.2 Simulation Control Panel Window

In the "Simulation Control Panel" window can be found three sections. The first section is the one which manages the "playing" of the simulation process. The second one displays the actual time state of the simulation process, as well as provides the button to open the preferences window. The third one lets the user to act over the buildings data (add and edit) calling the respective windows. This window is shown in Figure 4.7.

The first section provides the the means to control the execution of the simulation process. It consists of five buttons, "Play", "Pause", "Stop", "Forward" and "Backward". The actions related with each of these buildings are very intuitive.

Pressing the **"Play"** button, the simulation process is started. This process invoke the related routines to perform the visualization. In this way, the simulation time advances and buildings appear and disappear in the world. If the simulation was previously paused, pressing this button, the simulation is restarted from the current time.

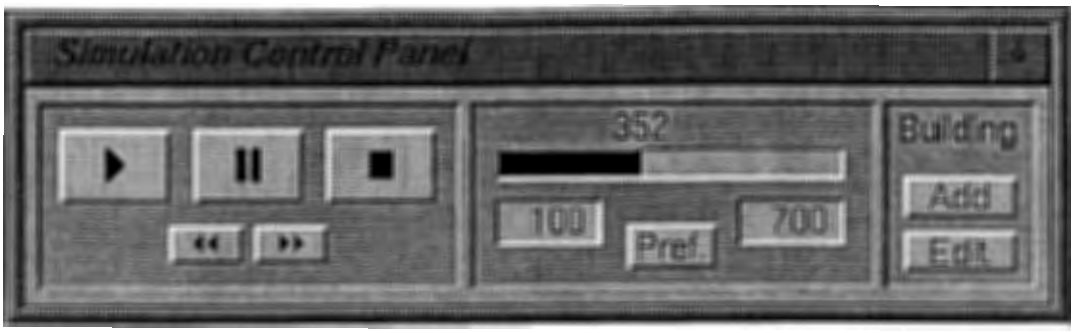

Figure 4.7: Simulation Control Panel

Pressing the **"Pause"** button, the simulation process is paused. This process unschedules the next event to be executed, and leaves the world untouched. The user can restart the simulation from this point, as well as navigate in time to next or previous events.

Pressing the **"Stop"** button, the simulation process is stopped. This process unschedules the next event to be executed, remove every building from the world, and return it to the starting point of the simulation.

Pressing the **"Forward"** button when the simulation process have been previously paused, the clock of the simulation is advanced. This "jumps" in time, and updates the village state to the correspondent time value.

Pressing the **"Backward"** button when the simulation process have been previously paused, the clock of the simulation is moved backward. This "jumps" in time, and updates the village state to the correspondent time value.

The second section of the "Simulation Control Panel" contents a slider that indicate the actual time value of the simulation clock. This provide the information of when in time it the village visualized. Looking at this slider the user can situates itself in time, as well as keep track of the time sequence. This slider has as limits the dates that were assigned to the personalized copy of the database as its simulation term. Also in this section is contained the button to invoke the "Preferences" window.

The simulation "Preferences" window (Figure 4.8) consists in two different sets of values. One set is the "Duration" of the visualization process, it is the real time period that the user wants the simulation is performed in. In this sense, the user can regulate the time the visualization is going to last, as well as how fast this process will be performed. The longer the duration, the slower the simulation. Also this section provides two fields to set the duration slider limits. This slider is used to make the process easier and convenient. The other value that can be adjusted is the "Visualization Term". This is, the two years between the user wants to visualize the evolution process. In this sense, the user do not require to simulate the whole process if it is not needed. The user can focus on one subset of the whole simulation term, and perform the simulation only between those limits.

The third section of the "Simulation Control Panel" contents two buttons which invokes the routines to handle the data of the buildings. These two buttons are "Add" and "Edit".

When the "Add" button is pressed, it triggers the process to include a new building in the village evolution process. Is is made through the "New Building" window. When the "Add" building button is pressed, it triggers the process to include a new building in the village evolution process. Before press this button, the user must choose one vestige mark on the land over the building is going to be built. This selection is made in the viewer windows (section 4.2). The specification of values related with the building is made through the "New Building" window.

The "New Building" window (Figure 4.9 consist of three parts. The first one is related with the building's "Name". The name is the value used to identify each building. This part provides a text field where the user can type the new building's name. The second part is related with the building's "Life Term". It provides two text fields and two sliders. In this part the user can input the dates of building and decaying for the

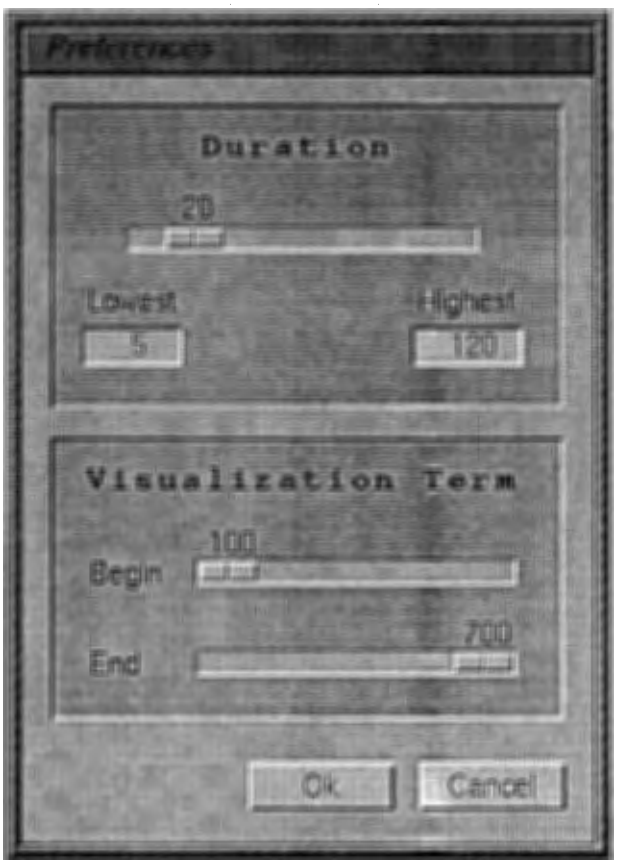

Figure 4.8: Preferences Window

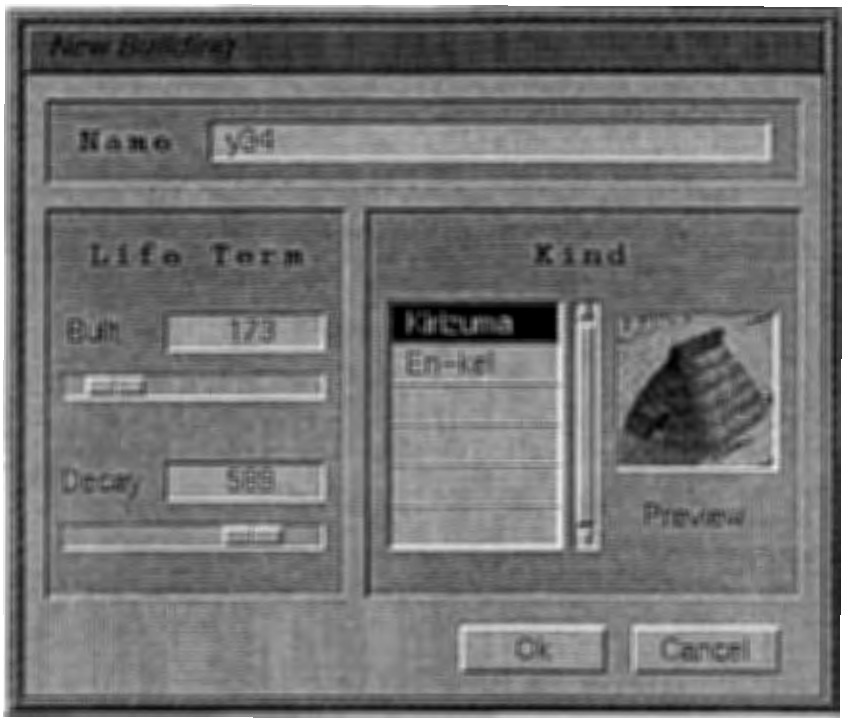

Figure 4.9: New Building Window

new building. These dates can be typed directly on the text field or can be introduced making use of the slider related whit the value. The sliders are constrained between the simulation term assigned to the current database personalized file. The third section is the building's "Kind". It consist of a list and a small preview display area. The list allows the user to choose one of the available kind of buildings. Even when the new building should be a assigned in accordance to the vestige, the user can set this value arbitrarily. The preview area display an image of the kind of building currently selected. This window contains two buttons, one to accept the values and include the new building in the personalized database, and other to cancel the process and abort the new building creation.

When the "Edit" button is pressed, it triggers the process to edit an existing building. Is is made through the "Edit Building" window. When the "Edit" building button is pressed, it triggers the process to edit an existing building.

There are three different ways to select which building is going to be edited. Two of them make use of the "List of Buildings" window (Figure 4.10), where the user select the name of the building that s/he wants to edit. The other one requires that the user previously select a building from the scene (Figure 4.11). This last way can be done only when a simulation process is paused.

The list can contains all the existing buildings, or can be restricted to show only the buildings related with a specific vestige. Which list is going to be displayed depends on the current selection the user have made on the virtual world. This selection is made through the viewer windows (section 4.2). If the user have already selected one vestige mark before press the "Edit" button, the constrained list having that vestige as a key for a query is shown. If no selection is done at that moment, the whole building list is displayed. A building can be deleted from the building list directly.

Once that which building is desired to edit, the "Edit Building" window (Figure 4.12) is displayed. This window is very similar to the "New Building" window due the user can manipulate the same values in the same form. The additional feature is the "delete" button, that the user can use to delete a building from the personalized database.

Figure 4.10: Building List Window

4.2 Viewers

The interface between the user and the virtual world is made using two different windows. One of them provide a view as the user were walking in the terrain, and the other, which is optional, provides an arbitrary view that the user can control.

The first viewer "Walk Viewer" (Figure 4.13 and Figure 4.14), is constrained to move in a land level, it resembles the walking of a human being, obtaining the view from this perspective. The user can walk in the village and look at some points and details that s/he consider appropriated. This viewer situate the user in the world, and the movements the user execute using this viewer update instantaneously the "human" model in the world. The user can choose one of every object that are actually visible through this view.

The second viewer "Aux Viewer" (Figure 4.15), which is optional, and can be suppressed when the user decide to do it, provide an auxiliary view over the world. This viewer is not constrained to keep in contact with the land, due no human representation is attached to it. The movement freedom of this viewer is larger than the "Walk Viewer". It can me manipulated to resemble a "bird" view, a "mouse" view, or as another "human" view. This is illustrated in the Figure 4.16. This viewer can be used to help the walking navigation in the village (Figure 4.17) due the human model is visible from this view, as well as provide a more comfortable way to set the building's data values over the village. The "bird view" allows the user to perceive the whole village, as if s/he were flying over it, and interact choosing one of every vestige mark that exist on the terrain.

The viewers windows used in this application are objects obtained from the OpenInventor class "SoXtWalkViewer". The paradigm for this viewer is a walkthrough of an architectural model. Its primary behavior is forward, backward, and left/right turning motion while maintaining a constant "eye level". It is also possible to stop and look around at the scene. The eye level plane can be disabled, allowing the viewer to proceed in the "look at" direction, as if on an escalator. The eye level plane can also be translated up and down - similar to an elevator.

This class inherits from the "SoXtRenderArea" among others. The "SoXtRenderArea" provides Inventor rendering and event handling inside a GLX Motif widget. X events that occur in the render area can be handled

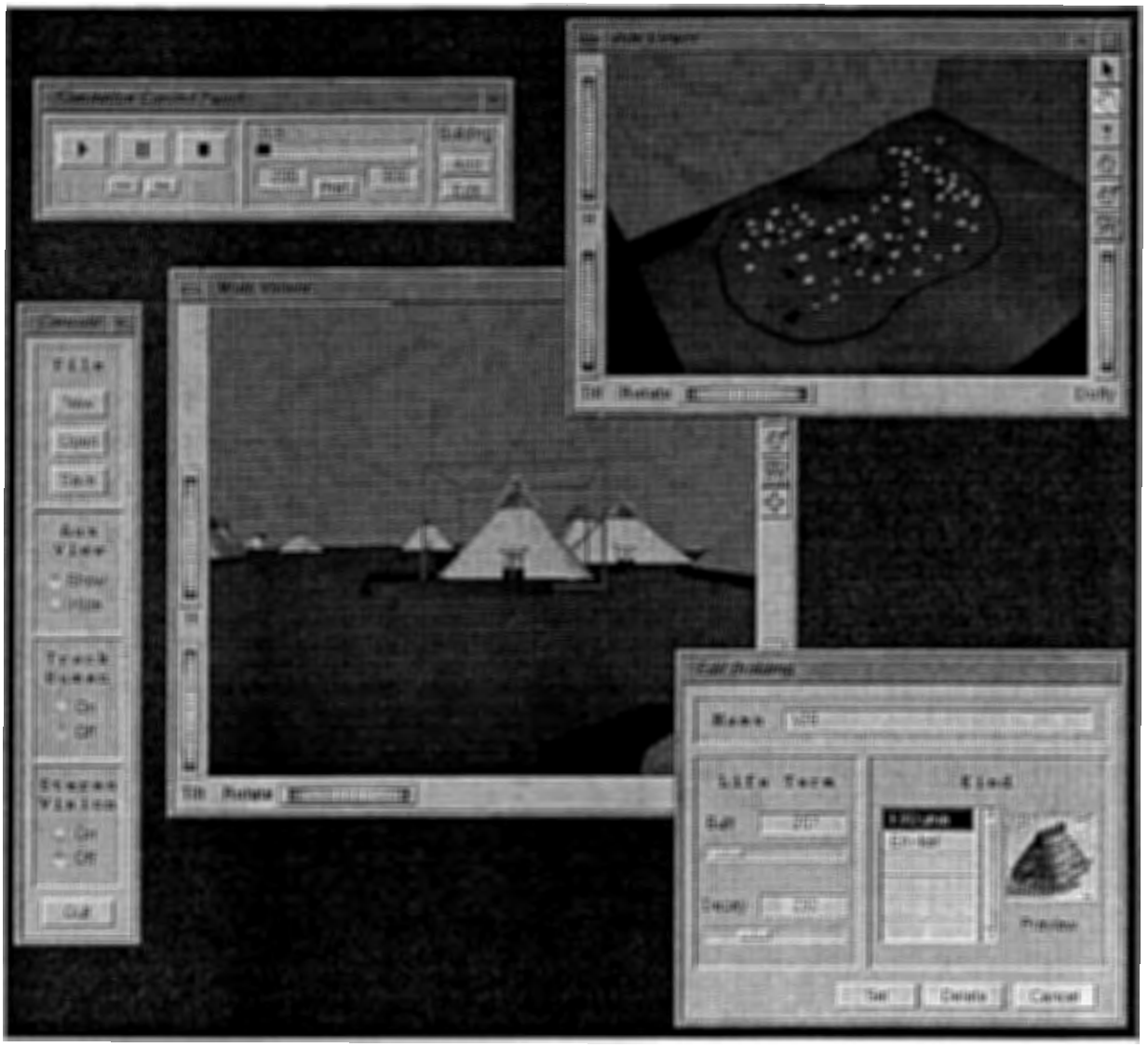

Figure 4.11: Editing a building

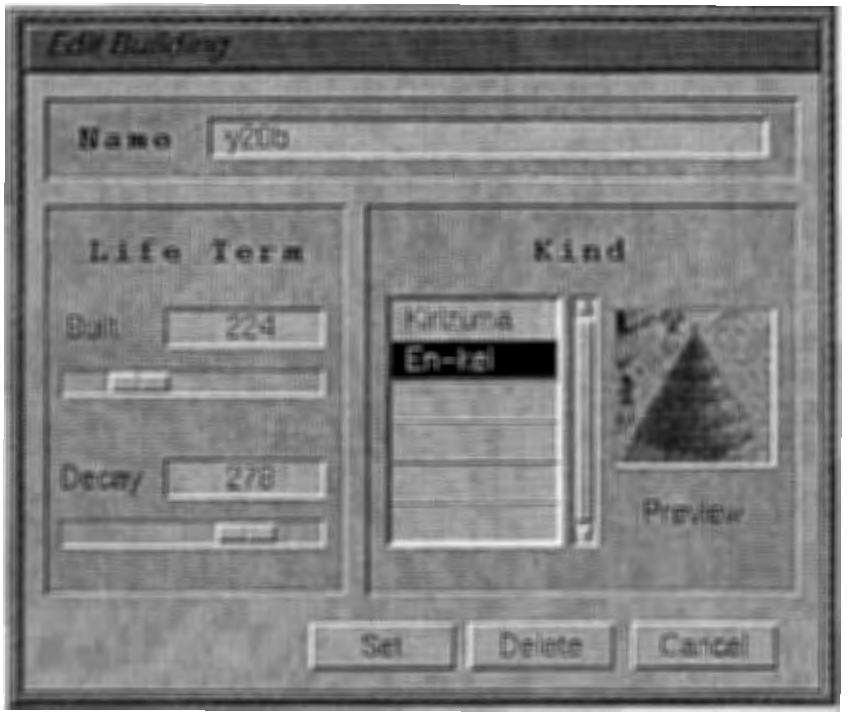

Figure 4.12: Edit Building Window

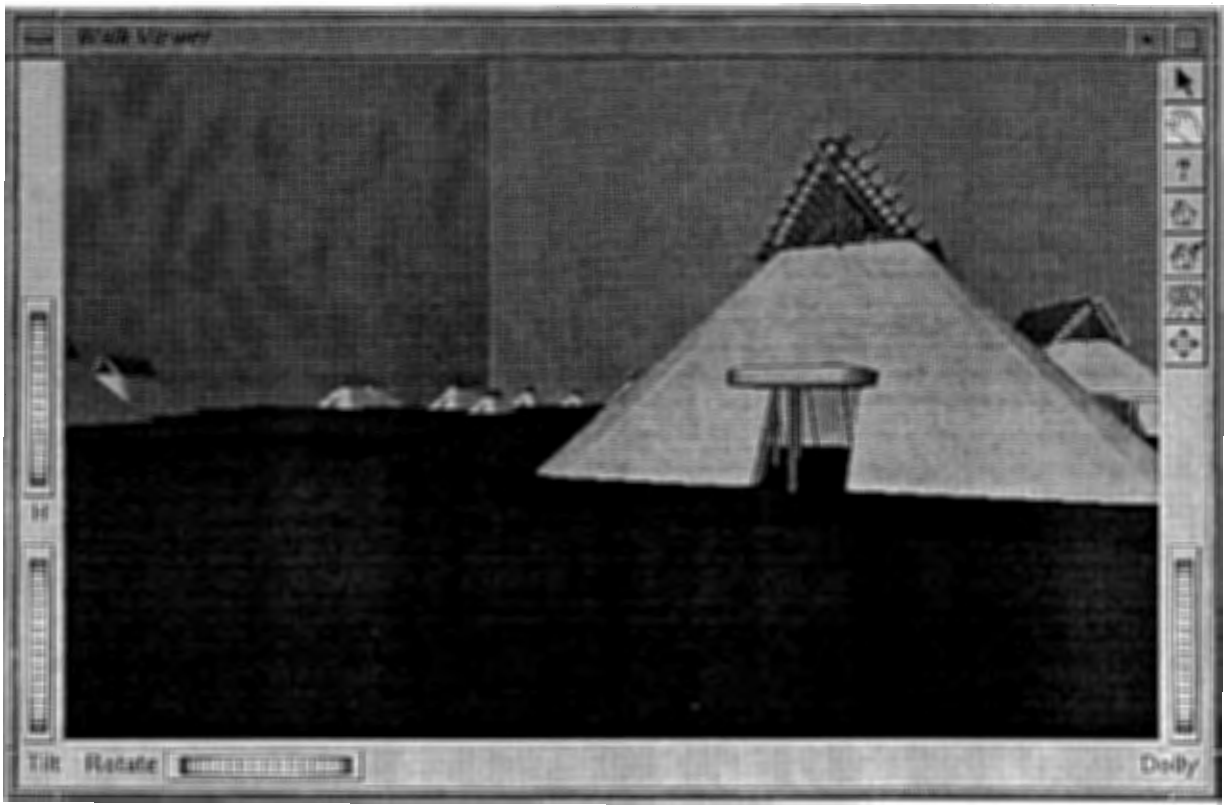

Figure 4.13: Walk Viewer

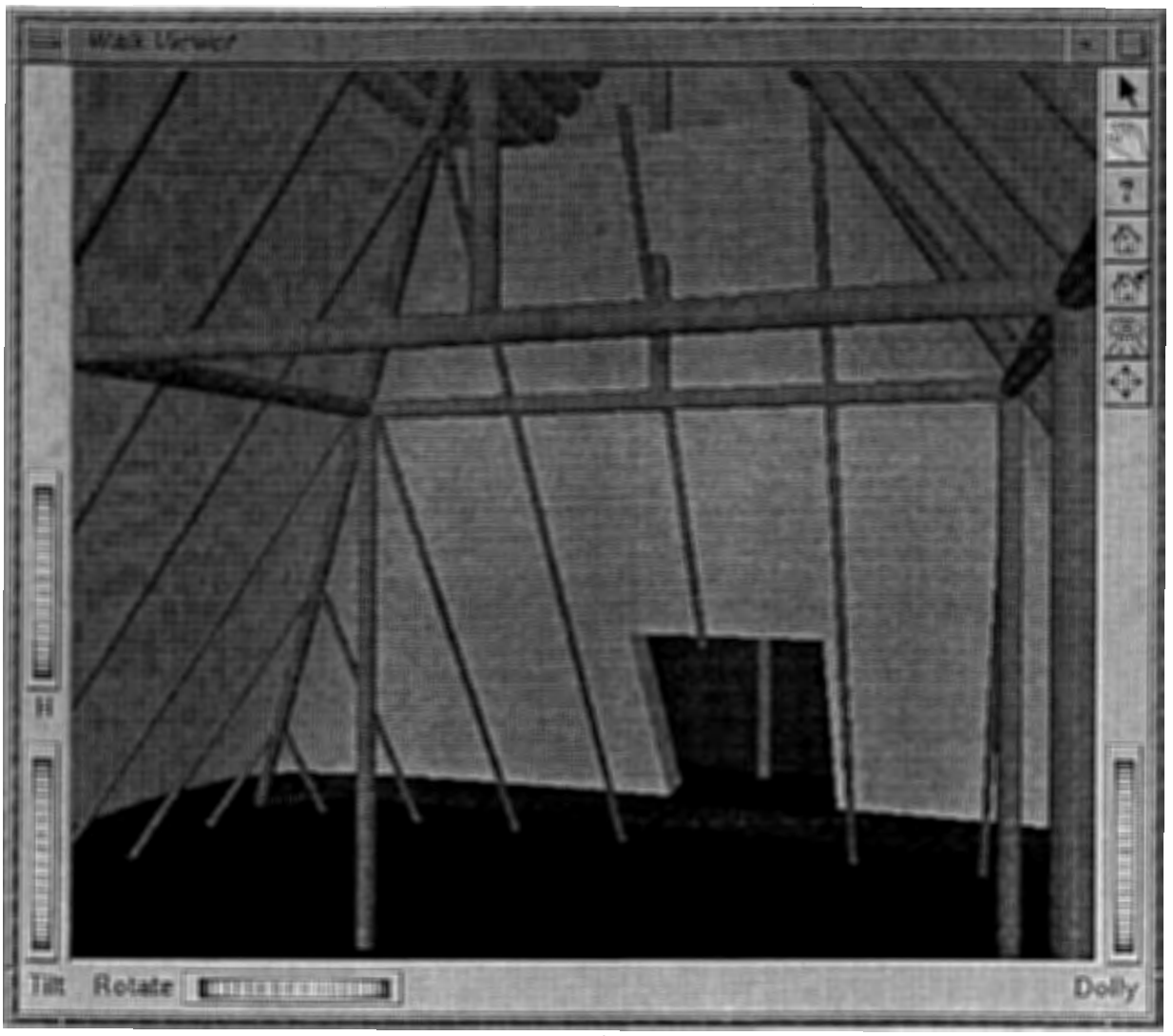

Figure 4.14: Walk Viewer (Inside one building)

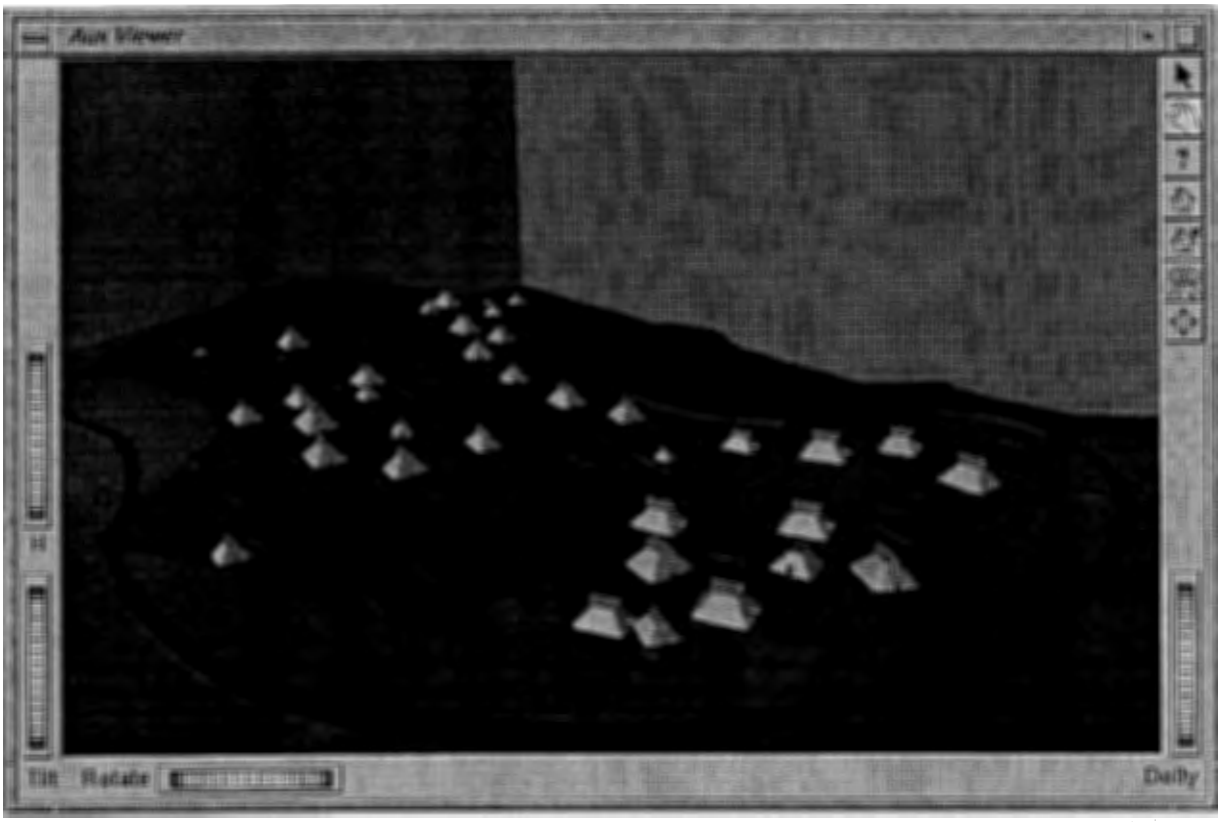

Figure 4.15: Auxiliary Viewer

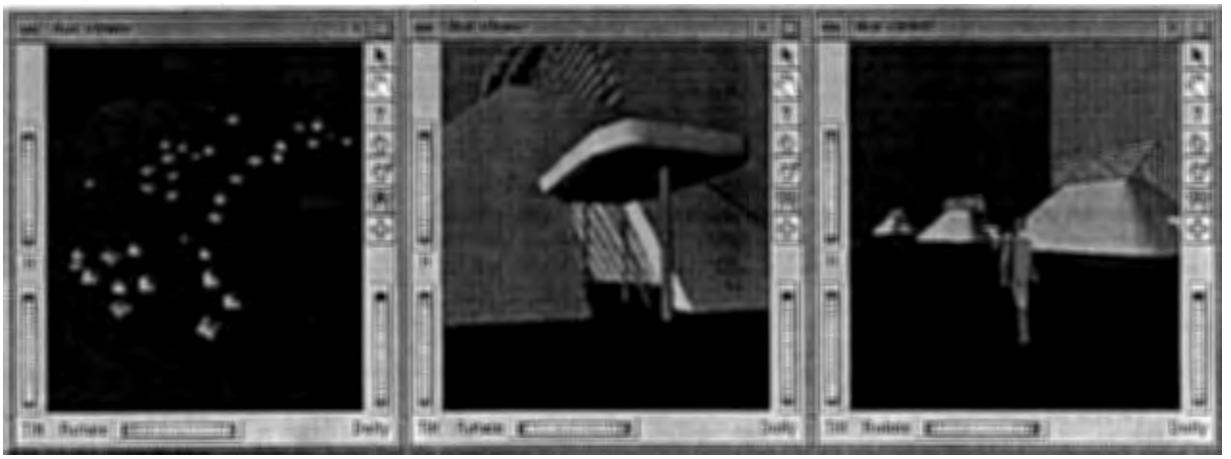

Figure 4.16: Different Point of View using the Auxiliary Viewer

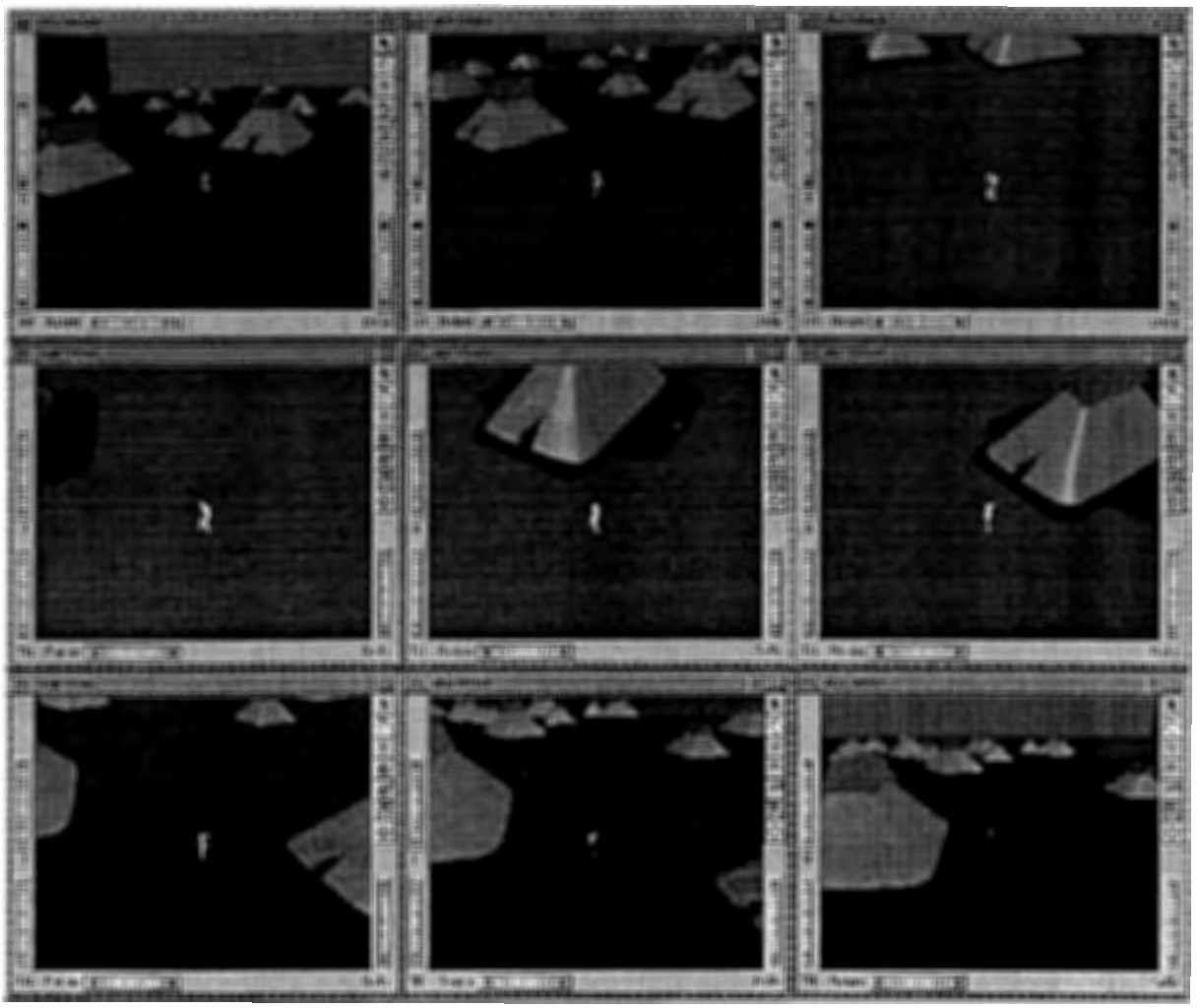

Figure 4.17: A sequence showing "Tracking the Humna" from the Aux viewer

by the application, by the viewer or by the nodes in the scene graph. When an event occurs (i.e. the user pick an object on the scene that is been displayed), it is first passed to the application event callback function registered with the setEventCallback() method on SoXtRenderArea. If this function does not exist or returns FALSE, the X event is either used directly by the viewer or translated to an SoEvent for further scene graph processing. If the viewer does not handle the event, and an overlay scene graph exists, the SoEvent is sent to that scene graph by way of an SoHandleEventAction. If no node in the overlay scene graph handles the event (i.e. calls setHandled() on the SoHandleEventAction), the SoEvent is passed to the normal scene graph in the same manner.

In the inheriting chain, the following class is "SoXtViewer". This is the lowest base class for viewer components. This class adds the notion of a camera to the SoXtRenderArea class. Also enables the user to change drawing styles (like wireframe or move wireframe), and buffering types. This base class also provides a convenient way to have the camera near and far clipping planes be automatically adjusted to minimize the clipping of objects in the scene.

Next class is "SoXtFullViewer". This is a base class used by all viewer components. The class adds a decoration around the rendering area which includes thumb wheels, a zoom slider and push buttons. This base class also includes a viewer popup menu and a preference sheet with generic viewing functions.

The last class before reach "SoXtWalkViewer"class is the "SoXtConstrainedViewer" class. This class adds methods and convenience routines available to subclasses to constrain the camera given a world up direction. This prevents the camera from looking upside down. By default the +Y direction is used.

The "SoXtWalkViewer" usage consist of:

-Left Mouse: walk mode. Click down and move up/down for forwards/backwards motion. Move right and left for turning. Speed increases exponentially with the distance from the mouse-down origin.

-Middle Mouse - or

-Ctrl + Left Mouse: Translate up, down, left and right.

-Ctrl + Middle Mouse: tilt the camera up/down and right/left. This allows you to look around while stopped.

-(s) + click: Alternative to the Seek button. Press (but do not hold down) the (s) key, then click on a target object.

-(u) + click: Press (but do not hold down) the (u) key, then click on a target object to set the "up" direction to the surface normal. By default +y is the "up" direction.

-Right Mouse: Open the popup menu.

4.3 Joining OpenInterface and OpenInventor

This application was built making use of two different programming environments, OpenInventor to handle all the 3D computer graphics funtionality, and OpenInterface to handle all the graphic interface funtionality. Both programming environments provide their own "mainLoop" routine which is in charge of checking the events that occur in their context, and trigger the related action as a response to the such event. These both routines can be illustarted by Figure 4.18.

To join the two environments, in the sense of receive user commands through OpenInterface and execute actions in OpenInventor, and/or display some OpenInventor model internal information through OpenInterface, the point where the attention must be focused is in each environment's mainloop. The way as each environment should interact with the other one becomes a communiction using shared "boards", where the sender environment writes the message, and the receiver environment reads it end execute the associated action.

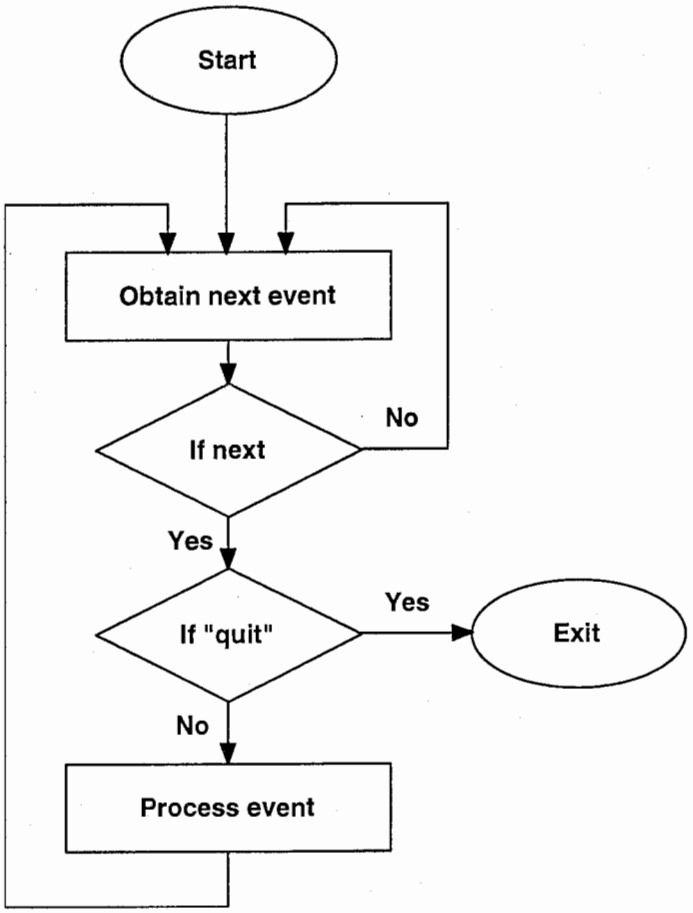

Figure 4.18: Toolkit Main Loop

The mainLoop created to be used by the application is composed by the content of each environment main loop, and a rouitine that check if there are any message in the board and trigger the related action. This mainLoop is ilustarted in the Figure 4.19.

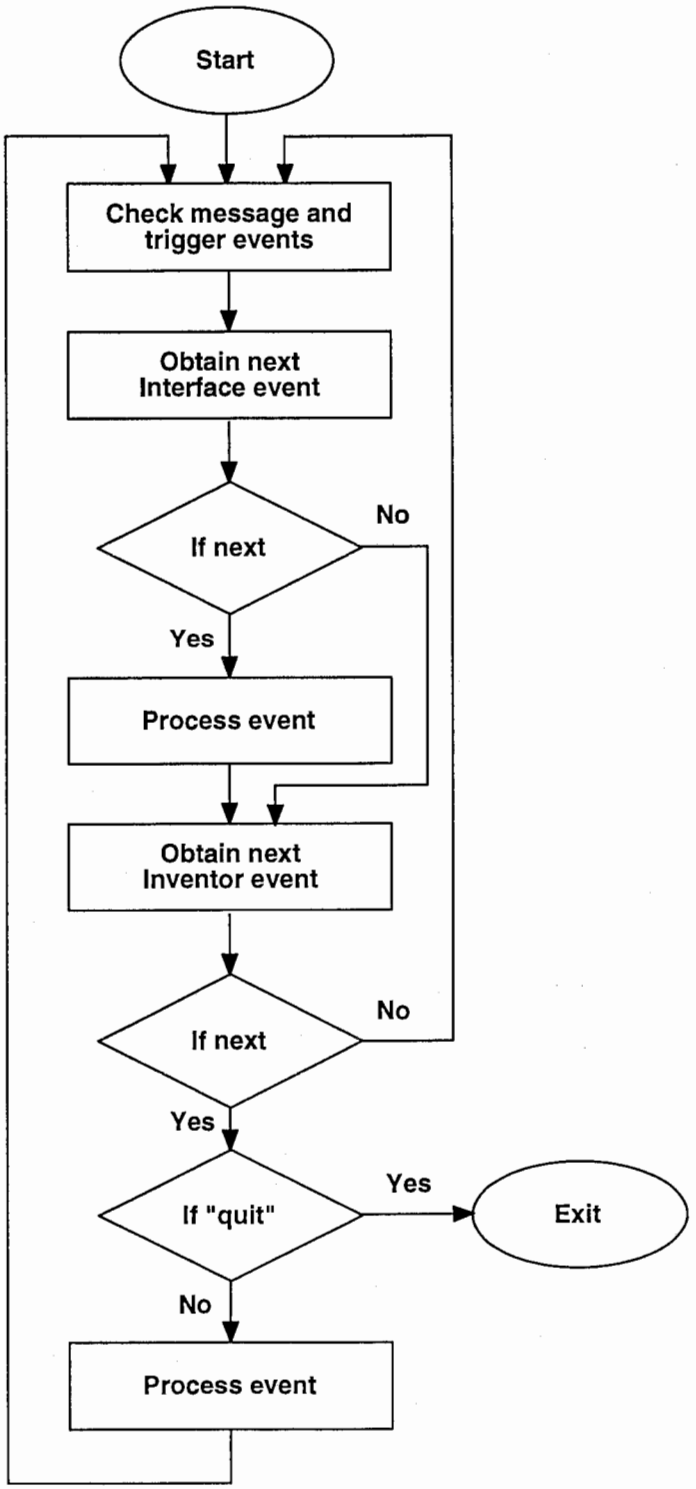

Figure 4.19: Application Main Loop

Chapter 5

3D Models

For the creation of the 3D models is used SoftImage 3D as a modeling tool. Several models are incorporated in the application. As mentioned in the Chapter 2, there are 4 groups of models. The models related with the buildings, the models related with the vestiges marks, the model of the land and the model of the human. In this section how the models were created and how were incorporated in the application is explained.

5.1 Building's model

In this version, two different kind of buildings are included. These models correspond to the archaeologist hypothesis on how the architecture of the building was. In both models the basic same construction sequence was used. First the structure of the building is modeled, creating the columns and beams, and the roof base. They are created using cylinders that work like wood stick. After the structure is created, it is covered with a surface that resembles the "walls" of the vestige, in accordance with the shape and building style. The last step is the making of the roof.

The first model created is the "En-kei" model, a rounded base, very simple structure building. The structure consist of six main columns set making a circle. Over the columns, a circle of beans is used to support all the sticks that form the "wall-roof" base. These sticks are spread over the circular beans structure, making a sort of conic shape. To this elements, a wood like material color is assigned. (Figure 5.1)

Over this structure, a cone like surface is created using two con-centered circles with different radius. This surface is provided adequate thickness that resemble the thickness that those building's wall should have. This surface is detailed modified in order to provide an entrance. A square hole in one side is made, and a small roof for the entrance is added. To this surface, a brush-wood like material color is applied. (Figure 5.2)

The other model created is the "Kirizuma" model, a square base, little bit more complicated structure building. The structure consist of six main columns that describe a rectangle. Over this columns, a rectangle of beans is incorporated to support all the sticks that form the "wall-roof" base and also support the second layer that support the roof. This second structure layer is composed by three small columns and a main bean. The sticks used as base for the roof are rest upon this main bean and the rectangle of beans of the first structure layer. (Figure 5.3)

Over this structure, a frustum like surface is created using two con-centered squares with different scale. This surface is provided adequate thickness that resemble the thickness that those building's wall should have. This surface is detailed modified in order to provide an entrance. A square hole in one side is made, and a small roof for the entrance is added. To this surface, a brush-wood like material color is applied. (Figure 5.4)

This kind of building have independent roof from the wall. This roof is modeled as a conjunct of sticks grouped horizontally that rest upon the roof base. To this element, a wood like material color is applied. (Figure 5.5)

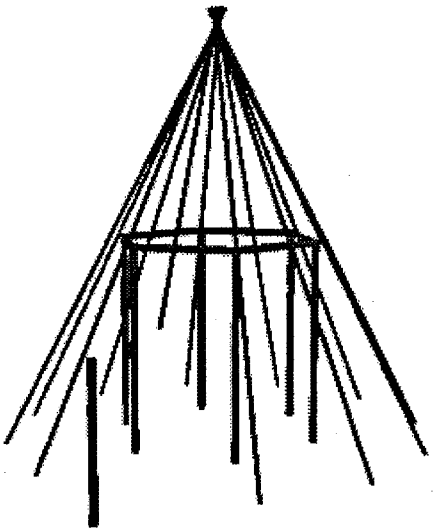

Figure 5.1: En-kei Structure

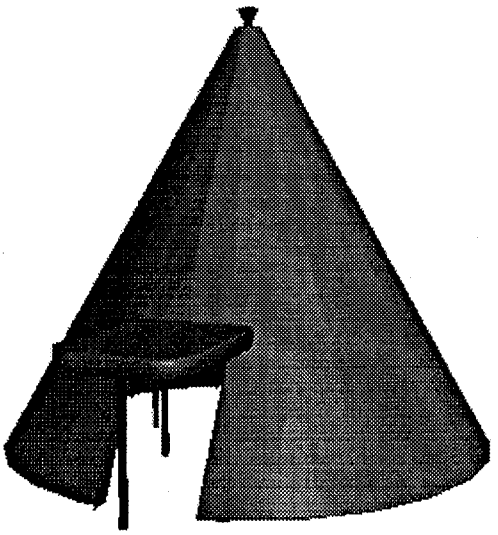

Figure 5.2: En-Kei Building

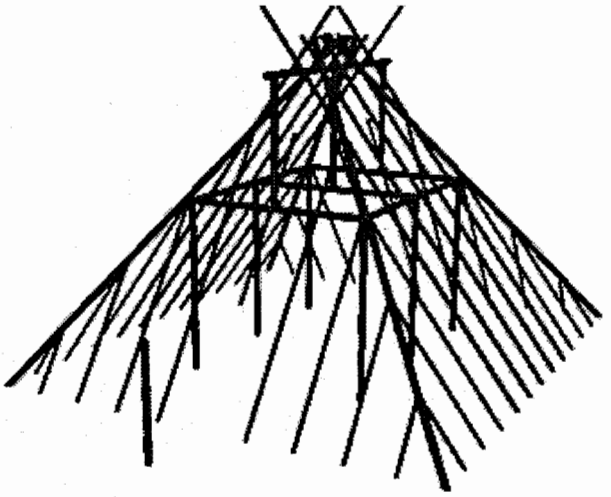

Figure 5.3: Kirizuma Structure

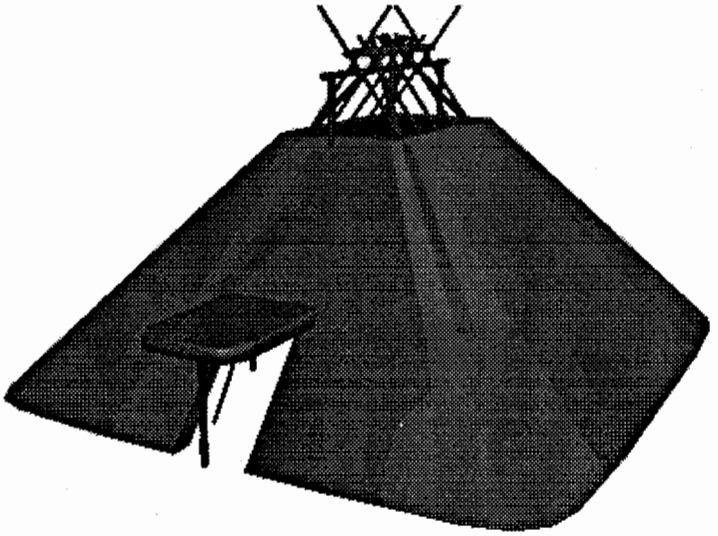

Figure 5.4: Kirizuma Surface

Figure 5.5: Kirizuma Building

5.2 Vestige's model

Each kind of building needs of its own kind of vestige. For this reason, two different vestige models are incorporated. The "En-kei Vestige" (Figure 5.6) and the "Kirizuma Vestige"(Figure 5.7) . These models consist on a small protrude of the land, in the same shape as the building base. It follows the same contour that the building floor describes. Both models have the same height, but the size and shape is obtained from the building's models. To this element, a earth like material color is applied.

5.3 Land model

The land model consist of the terrain where the vestiges marks are found and the sky and bottom land that enclose the virtual world. The terrain model is made following the details provide by the archaeologist in accordance to the site to be reconstructed. The sky and bottom land are square like objects that resemble having the world in a big but constrained room. The space is not infinite.

In order to make the terrain model the most similar to the real terrain (Figure 5.8), the contour lines are used as patterns. Over this contour lines, a set of points is choose to represent and handle the curves in SoftImage. Having the curves already in SoftImage, is needed to position each one at the level that it represents in a 3D space. A translation on the Y axe is performed, in accordance to level described by the contour, following the same scale provided in the blueprint. Once that the contours related curves are in the right "height" (Figure 5.9), a "skin surface" operation is applied over them in order to get a surface that resembles the real terrain (Figure 5.10).

It is important to realize that to be able to perform this action over the curves, each curve must be composed by the same number of points, and related in the same order. For this reason is needed first to select carefully the set of points that are going to be used in the model creation. In this version, a set of points that allow the reconstruction of the curves, without loosing too much details and avoiding the fact of increasing too much the number of triangles in the surface were selected. This way a good representation of the terrain can be obtained, whit out sacrifice significant real-time response, and the user can situate him/herself in the context of the village terrain, and interpret this information at the moment of setting the evolution sequence.

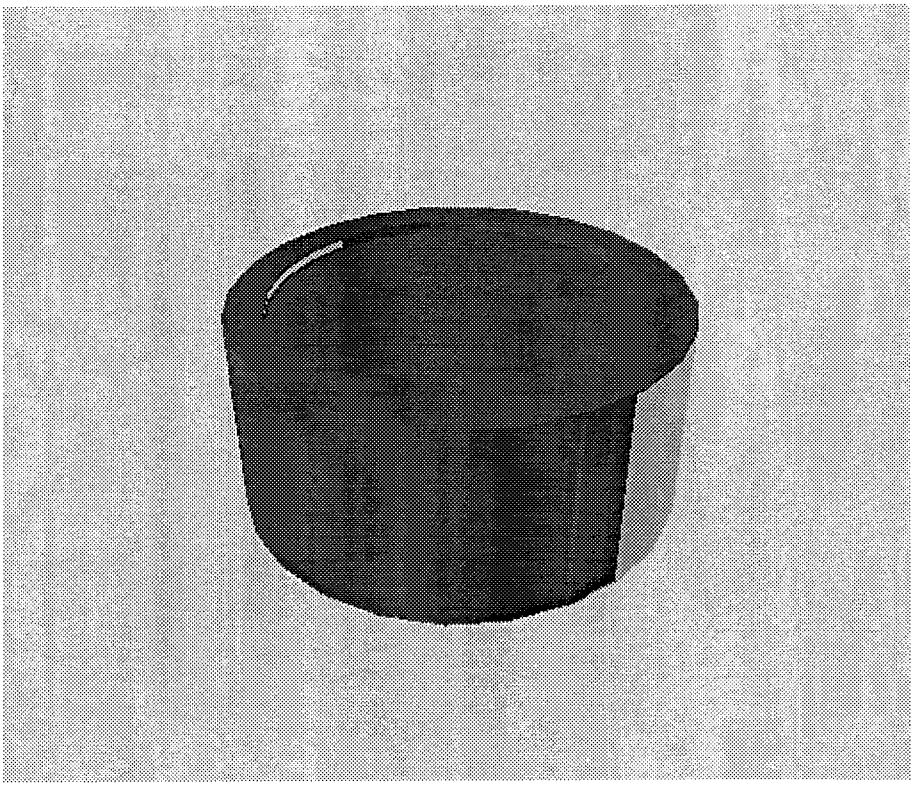

Figure 5.6: En-kei Vestige

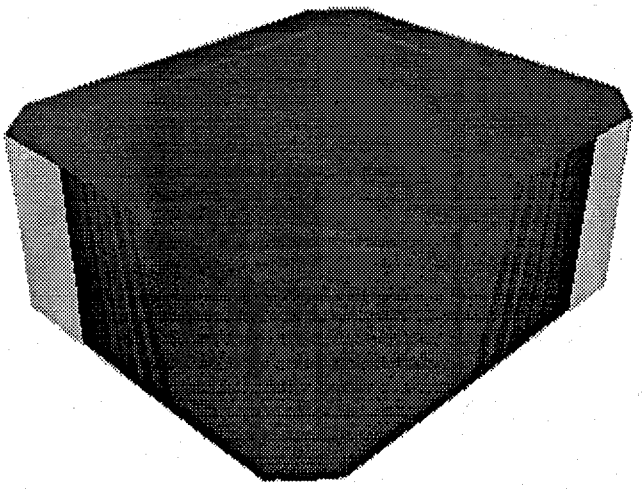

Figure 5.7: Kirizuma Vestige

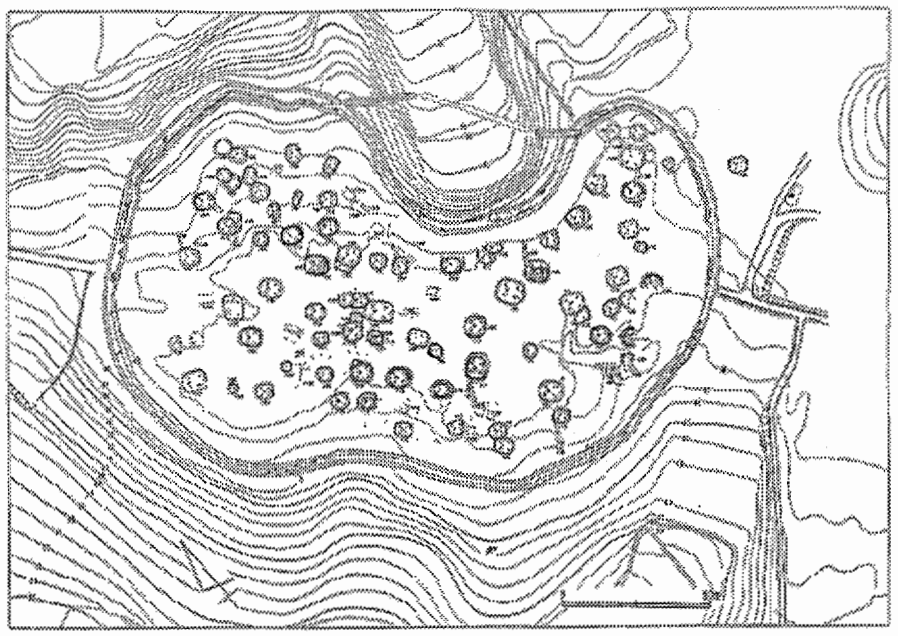

Figure 5.8: Blueprint of the Site

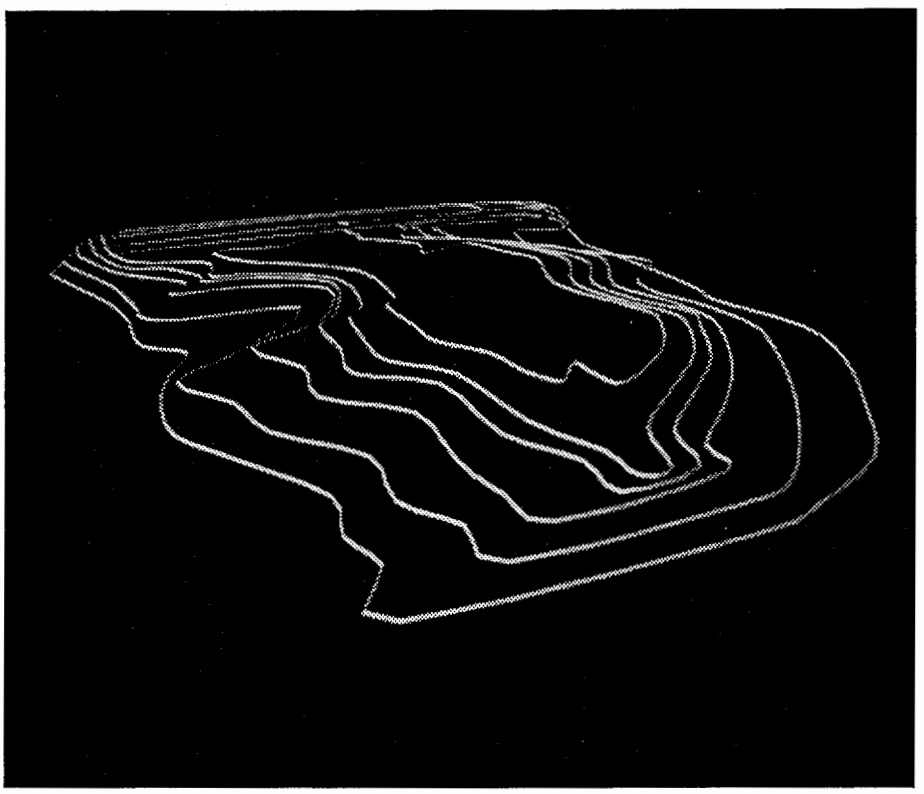

Figure 5.9: Terrain Contours

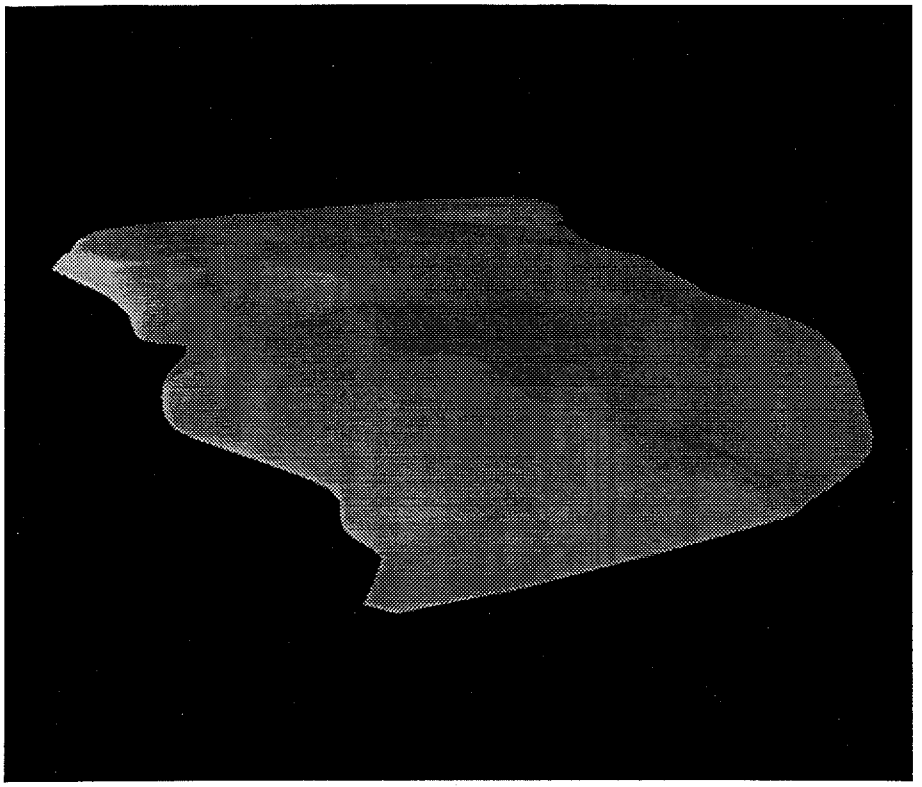

Figure 5.10: Terrain Surface

5.4 Human model

The model of the human is made very simple. Just with few cubes resized and translated, and some deferent materials are applied to some of the cubes in order to provide the idea that the human is dressed. This model also is provided hair an a human like walking posture in order that the user can recognize easily what is the front of the human, and where it is facing to. (Figure 5.10)

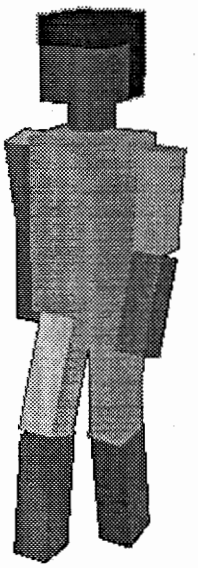

Figure 5.11: Human

Chapter 6

Reconstructing the Site and Making the database

As a first step in the whole process, is needed to create a database of the site were the evolution process is going to be simulated. This is the starting point of the actual Datamining process. In this sense, any user will make his/her personalized copy of the whole database, and using it, adjust the building's data values iteratively in order to get a valid evolution pattern, which become then the new knowledge discovered.

6.1 Obtaining the models

In order to create the database file that will be used in the application, several components are needed. This database file satisfy the OpenInventor application structure discussed on Chapter 2. In this section were cited which kind of objects compose the virtual world. They can be summarized as buildings, vestiges marks, land and human. Following these categories, is important to realize that for each kind of building that can exist in the site, is needed to provide the model for the building and the model for the vestige mark. In this version there are used two kind of buildings, the "En-kei" building and the "Kirizuma" building. For this reason the two different building's models and the related two vestiges models are included.

Those components must be provided as OpenInventor tractable objects. This is because they will be incorporated in the OpenInventor structure, and will be managed in this environment. Each element must be provided as a separated object, in a separated OpenInventor file.

As mentioned previously, the 3D Modeling tool used in the developing of this application was SoftImage 3D. All the models were created using this tool and after that, translated into OpenInventor valid format. The way that each model was translated consist of being in the SoftImage environment, joining all the components of each model under the scope of a father node, creating a tree. This tree is exported from SoftImage and saved as a Wavefront object. After that, the Wavefront object is translated into an OpenInventor object. The translator used is a freeware available from the Silicon Graphics home page (http://www.sgi.com/Works/translators.html). This translation is made this way, doing this two steps, because the translator provided to convert SoftImage scenes files directly to OpenInventor (SoftimageToIv) does not work correctly, and it was impossible to contact the translator's developers in order to get a solution to this problem. The other routine (ObjToIv) works correctly, and SoftImage by itself provides the tool to export 3D models as Wavefront Object files.

Having translated the SoftImage models into OpenInventor objects, they are already available to be included on the site database, within the OpenInventor structure. How the models are included is explained in section 6.3.

6.2 Obtaining the vestiges values

The creation of the site database by itself is a dense process that is done performing several activities. Among this activities can be found the identifying of the building's vestiges given a set of pits in a land, the retrieving of the vestige marks values of size and space localization. This pre-processing of the data is already done manually. The results obtained by these previous steps can be summarized as a list, where each vestige mark

$$\text{Factor(i)} = \frac{\text{Real\_size(i)}}{\text{Unit\_size(i)}}$$

Figure 6.1: Scale Factor

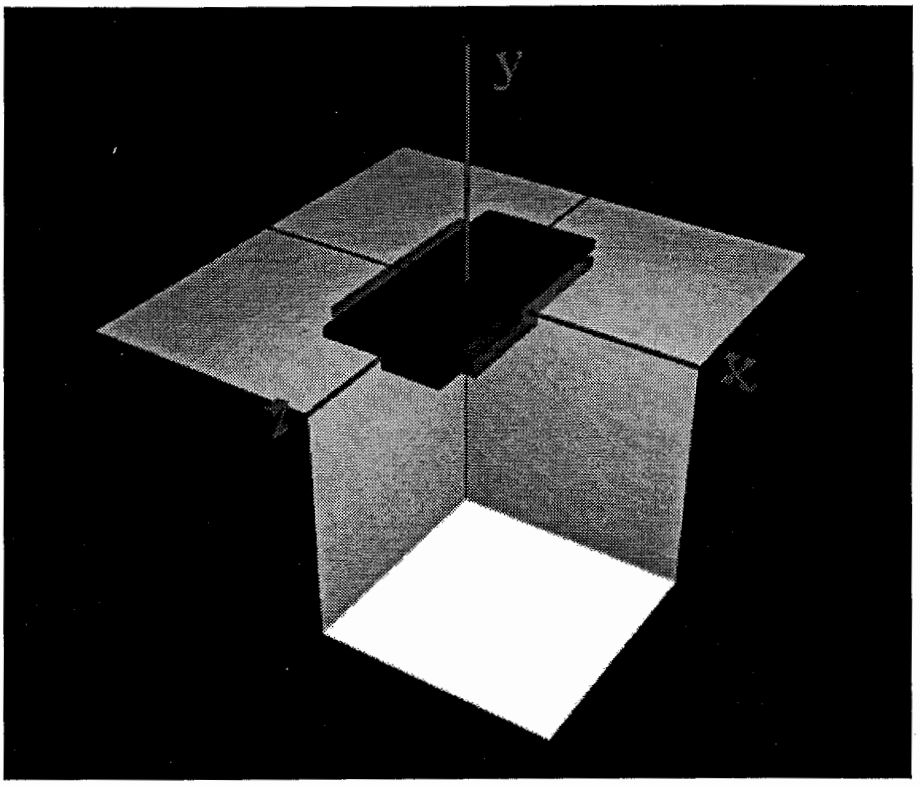

Figure 6.2: Scale Action

has its own size, its own 3D position related to a common point and a rotation on the Y axe that provide the orientation of the vestige mark.

Having this data, the place can be reconstructed in accordance to the real site. In the virtual world, the vestige marks are obtained creating instances from a original unitary vestige 3D model. This instance are scaled, translated and rotated in order to satisfy the real values. The pre-processed data of the vestiges must be provided in terms that the original model can be correctly transformed.

6.2.1 Scaling

Using the size of the unitary model, the real size values must represent a scale factor to which the model must be submitted in each one of the three axes. This scale factor is obtained dividing the real size of the vestige by the unitary size of the model. As the scale in the Y axe is unknown from the real data, an average of the X and Z scale factor is adopted in order to maintain the proportion of the models. The scale factor is shown in Figure 6.1 and the scale action is illustarted in Figure 6.2

6.2.2 Rotation

This value correspond to the angle that each vestige is rotated on the Y axe. This is the direction that each vestige posses, taking as a base null rotation (or 0 degrees of rotation) if the vestige is pointing in the Z axe direction. This value is contrary to the clockwise direction. The rotation action is illustarted in Figure 6.3.

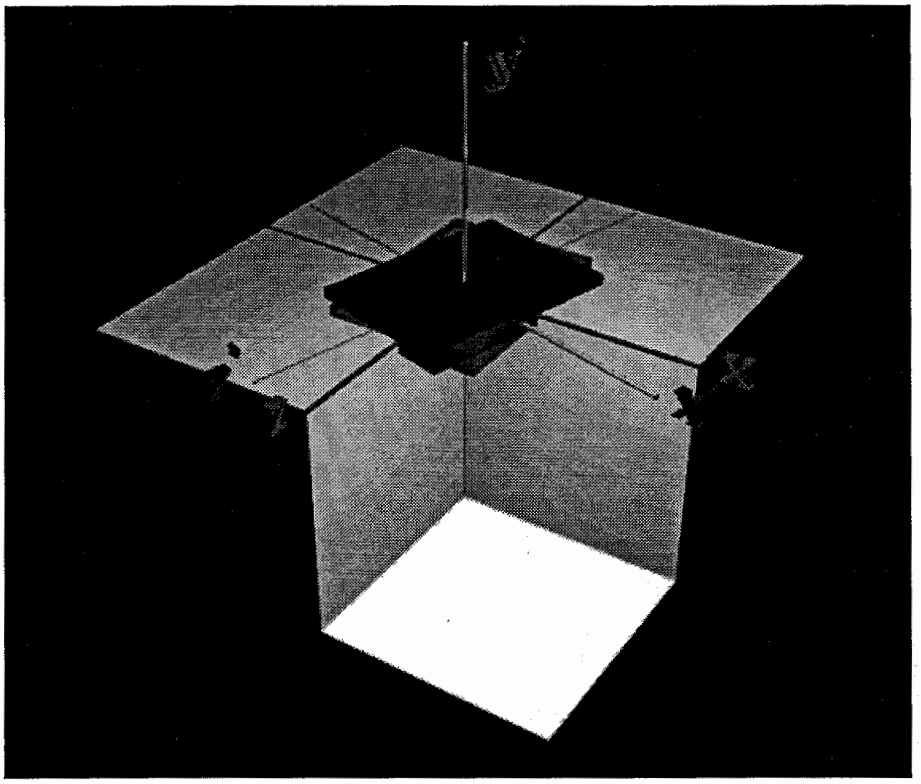

Figure 6.3: Rotation Action

6.2.3 Translating

Setting a relative origin point in the 3D space, each object can be positioned in the space correctly using this distance. In this sense, each objet is thought as being in the virtual origin and translated in each of the 3 axes. Applying the correspondent translations to every objects that exist in the world, its representation take the right place in the whole virtual world. The translation action is illustarted in Figure 6.4

Having obtained the vestiges values, they are already available to be included on the site database. How this data is used in the database creation process is explained in the next section.

6.3 Obtaining the database

This process is performed by an external routine called "Mapmaker", which get all the needed inputs and create the database file used by the application.

Among the inputs needed in this process are the 3D models and the vestige values. These two kind of inputs are processed to construct the OpenInventor structure required in the database file. The "Mapmaker" routine create all the nodes required to establish the hierarchy in the database. This structure is explained in Chapter 2.

Once that the whole graph is constructed, the "Mapmaker" reads from each OpenInventor files that contain the 3D model description, and incorporate those objects to the structure, in the respective place that the models should be located. Those objects can be already managed by the application in the OpenInventor environment.

Being the models incorporated in the structure, the next step is create the instance of the vestiges marks that appear in the virtual world. This process is performed obeying the vestige data provided. In this sense, each instance of the unitary vestige model is transformed in accordance to the specifications. This is a cycle process performed for each vestige mark existing in the site. After that, the virtual world is reconstructed, resembling the real data of the site (Figure 6.5).

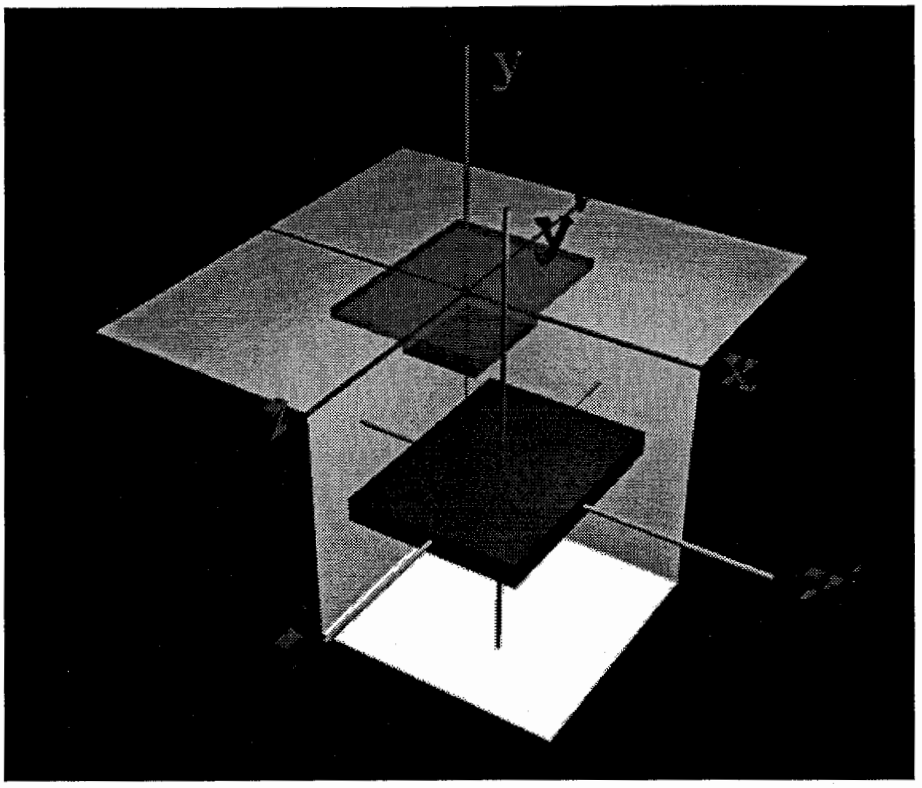

Figure 6.4: Translation Action

At this time, the virtual world is shown making use of two viewer windows. Each one related to the two viewers available in the application. Through these two windows, the user can check out the created virtual world, and set the camera home position for each viewer. The camera position is stored in the database, and when a new personalized copy is made, it will have this values as initial values.

The output of this process is a file called "database.db", this file should be renamed in order to satisfy several sites and user preferences, and moved to the default database directory. The extension of the database files should be ".db". This helps at the moment of classify the files.

The "Mapmaker" process is illustrated in the Figure 6.6.

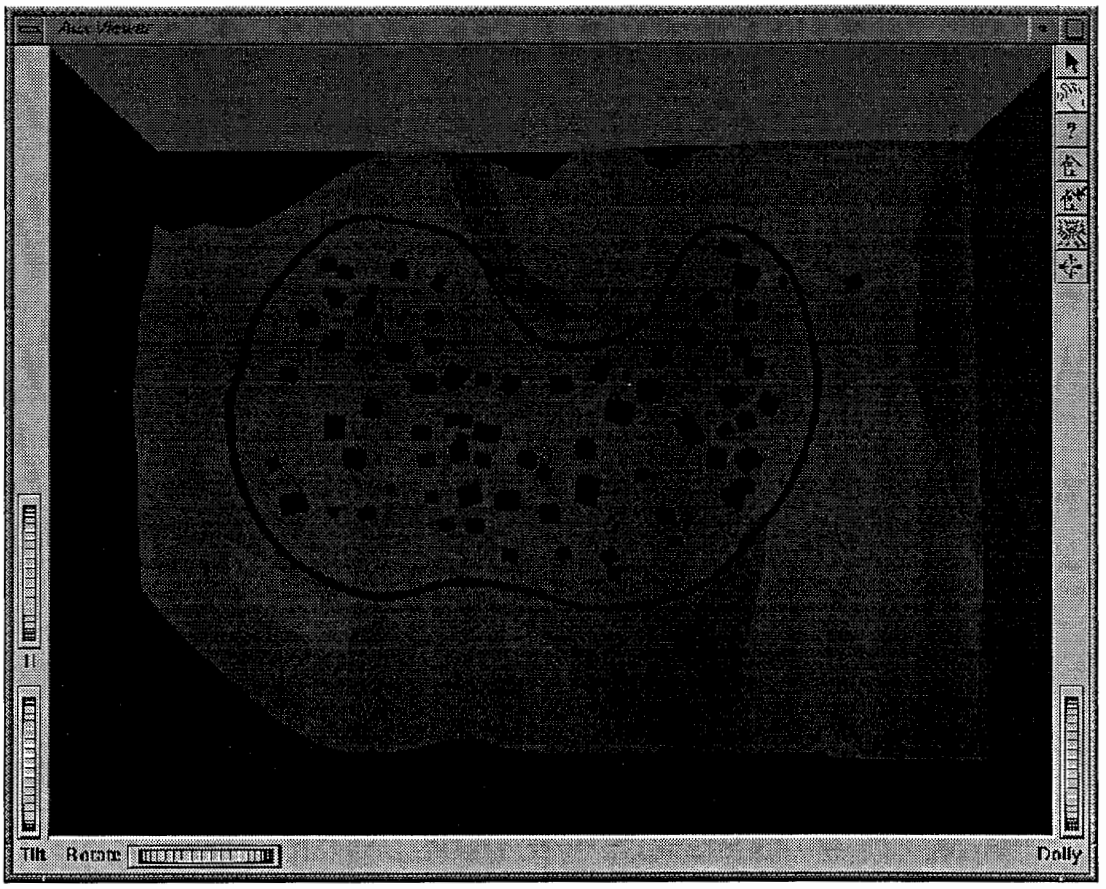

Figure 6.5: Top View of the Reconstucted Site

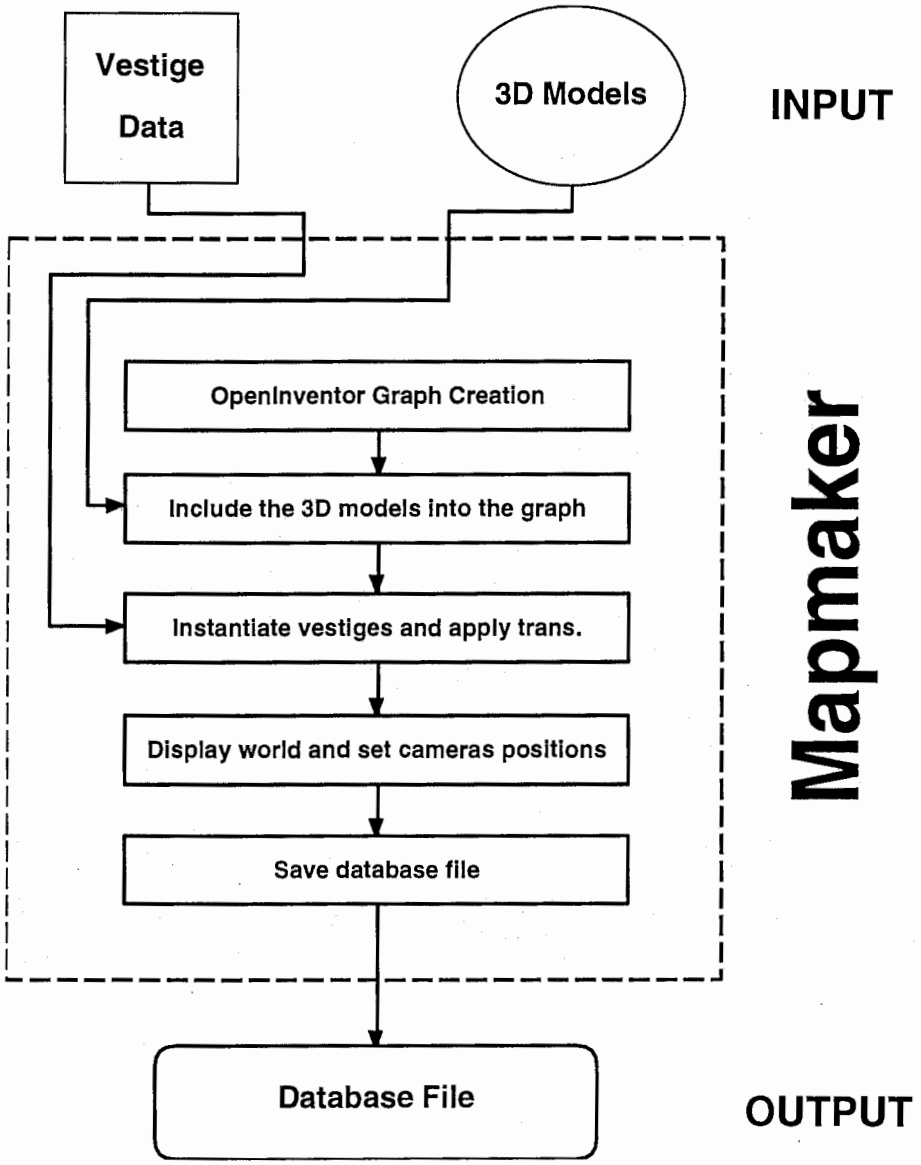

Figure 6.6: Mapmaker process

Chapter 7

Future Work

7.1 Supporting the previous steps of the KDD process

Creating a tool to help the archaeologists to determine the spatial position of the buildings based on the pits found at the excavations. This tool can be conceived as an algorithm that given a set of points in a plane, looks for those that compose a certainly pattern (vestige pattern). Having sorted and grouped the pits that can belong to a same building, the process of determining the buildings position becomes easier.

Also can be provided an environment such as ISAAC [Mine, 95], where the archaeologist can work directly in the virtual site (reconstructed from the land and pit's data). This reconstruction can be performed using Geographic Information Systems databases as a source. Once in the virtual site, the user can incorporate vestiges marks and alter them positioning, orienting and scaling using several kinds of manipulators, in accordance to the pit's data.

7.2 Improving the 3D navigation interface

The way as the user must move in the virtual environment is a hot topic. This is a very difficult goal in the Human Computer Interactions area. In the context of the Meta-museum project, a simpler and more intuitive methods have to be developed carefully, in order to support the visitor interaction in the space exploration. A good starting point is provided in [Ware, 90], where three different metaphors for the exploration of virtual environments using virtual cameras is provided.

7.3 Automatic evolution sequence generation

Another aspect that could be interesting to consider is the inclusion of some kind of automatic process that generate an evolution sequence providing some high level rules. This sequence can be suggested as a starting point of the Datamining process. This process can be thought in the context of Artificial Live principles, and can be developed easily using the Swarm software package [http://www.santafe.edu/projects/swarm/].

Chapter 8

Conclusion

We have briefly described the VisuArch project and the VisTA system created as one of the sub-projects and testbed of the Meta-Museum. The VisTA systems provides the means needed to interactively visualize the simulation of development of ancient villages, based on the archaeological and geographical data obtained from the excavations, supporting archaeologists in the formulation and evaluation of hypotheses about the formation process of ancient villages.

In the Meta-Museum context, the VisTA system supports the archaeologist's work in the formation and discovering of knowledge about the evolution of ancient villages, and allows visitors to visualize the discovered knowledge of the village formation process. The users can immerse in the archaeologists' knowledge space and communicate with them trough this space.

This system will be tested by experts (archaeologists) and non-expert (visitors) users. As a results we hope to observed how the communication process between experts and non-experts is going to be favored using a shared knowledge space. Also, incorporating factors as a comprehensive environment and immersive detailed terrain representation in the archaeologist's tool box, new frontiers of research are opened allowing to discover new knowledge.

Acknowledgements

I would like to thank Dr. R. NAKATSU, President of ATR Media Integration & Communications Research Laboratories for letting me work in this project at ATR for six months.

I would like to express my sincere gratitude to Dr. Kenji MASE, Head of the Communication Support Department, who welcomed me into his department, and gave me the opportunity to attend to several conferences, symposiums and workshops during my stay.

I would like to thank my supervisor Ms. Rieko KADOBAYASHI, Researcher, for her guidance and support of my work.

I would like to thank Dr. Sidney FELS and Dr. Armin BRUDERLIN for their very valuable comments and aid during the execution of all the related work.

Finally, I would like to thank the members of the Planning Division and Planning Section of ATR Media Integration & Communications Research Laboratories for having organized such a wonderful stay in Japan for me.

Bibliography

[1] Fayyad, U., Piatetsky-Shapiro, G., Smyth,P., Uthurusamy, R.: Advances in Knowledge Discovery and Data Mining. AAAI/MIT Press, 1996.

[2] Kadobayashi, R., Neeter, E., and Mase, K.: Interactive Knowledge Discovery from Archaeological Databases Proc. of the 1996 Information and System Society Conference of IEICE (to appear, in Japanese)

[3] Mase, K., Kadobayashi, R. and Nakatsu, R.: Meta-Museum: A Supportive Augmented-Reality Environment for Knowledge Sharing, Proc. of International Conference on Virtual Systems and Multimedia (to appear)

[4] Mine, M.: A Virtual Environment Tool for the Interactive Construction of Virtual World. Departament of Computer Science. University of North Carolina. TR95-020 1995.

[5] Ware, C., Osborne, S.: Exploration and Virtual Camera Control in Virtual Three Dimensional Environments. Faculty of Computer Science, University of New Brunswick, 1990.

[6] Langton, C., Minar, N., Burkhart, R., Askenazi, M.: The SWARM Simulation System. HTTP://www.santafe.edu/projects/swarm/

[7] Yokohama-shi treasure-trove research center: Otsuka Iseki: Kouhoku New-Town Excavation Report XII, 1991.