DIPLOMARBEIT

Applikationsmanagement für dreidimensionale Benutzerschnittstellen

Ausgeführt am
Institut für Softwaretechnik und Interaktive Systeme
der Technischen Universität Wien

unter der Anleitung von
Ao. Univ. Prof. Dipl.-Ing. Dr. techn. Dieter Schmalstieg

durch
Andreas Zajic
Matrikelnummer 9026400
Josef-Weissenecker-Gasse 4
2380 Perchtoldsdorf

Kurzfassung

Diese Diplomarbeit präsentiert einen neuen Ansatz für das Applikationsmanagement von Studierstube, einem Augmented Reality System. Es bietet einen virtuellen Arbeitsplatz für mehrere Benutzer, Anwendungsmultitasking und die verteilte Ausführung im Netzwerk mit Migration von Anwendungen.

Die Studierstube Software verwendet einen verteilten Szenegraph zum Speichern von graphischen und applikationsbezogenen Daten. Aufbauend auf der bestehenden Studierstube Implementierung, wird dieses Konzept weiter entwickelt, indem wir die Anwendungen selbst als Knoten in diesem Szenegraph speichern. Zusätzlich verbessern wir die Struktur des bereits existierenden Teils der Software durch die Anwendung von Refactoring. Ziel ist es, den Szenegraph als zentrale Datenbank des Systems zu verwenden und andere Datenstrukturen möglichst zu vermeiden. Dabei wird spezieller Wert auf die leichte Portierbarkeit von bestehenden Applikationen gelegt, d. h., die Programmierschnittstellen werden möglichst wenig verändert.

Unsere Lösung bietet eine strukturiertere Softwarearchitektur des Studierstube Laufzeitsystems. Sie erweitert das Szenegraphkonzept in Richtung einer allgemeinen Datenbank, die sowohl graphische und anwendungsbezogene Daten als auch die Applikationen selbst in Knoten speichert. Die Applikationen profitieren von der Gestaltung als Szenegraph Knoten. Einerseits wird dadurch eine einfachere Implementierung neuer Anwendungen durch Vererbung möglich, andererseits können Applikationen wie graphische Szenegraph Knoten in Skriptdateien verwendet werden.

Andreas Zajic

Application Management for three-dimensional user interfaces

Master Thesis

Supervised by
Ao. Univ. Prof. Dipl.-Ing. Dr. techn. Dieter Schmalstieg

Abstract

This master thesis presents a new approach to the application management of Studierstube, an Augmented Reality system offering a virtual workspace for multi-user collaboration, multi-tasking of applications and network distribution with application migration.

The software framework of Studierstube manages a distributed shared scene graph storing graphical data and application data. Building on the already existing implementation of Studierstube, we pursue this concept and take it a step forward by placing applications as nodes in this scene graph structure. Additionally the existing parts of the software are improved by applying extensive Refactoring with the goal in mind, to use the scene graph as a central database and avoid additional “legacy” data structures. Special care is taken to make porting of existing applications as easy as possible, i.e. existing programming interfaces are only slightly modified.

Our proposed solution results in a more structured software architecture of the Studierstube runtime system. It extends the scene graph concept towards a general database like structure for storing graphical data, application data and applications as nodes. Applications benefit from scene graph node functionality by being easy to implement by sub classing and the ability to be used in script files like graphical scene graph data.

Contents

1 Introduction

The concept of Augmented Reality (AR) describes the combination of the real world with a computer generated virtual environment in a real time interactive manner. It allows the users to see the real world surrounding them, with virtual objects superimposed upon it. Exact alignment of both environments in 3D creates the illusion of virtual objects coexisting with real ones.

"Augmented Reality enhances a user's perception of and interaction with the real world. The virtual objects display information that the user cannot directly detect with his own senses. The information conveyed by the virtual objects helps a user perform real-world tasks." ([9])

The field of Augmented Reality is growing fast with much research being done on different system setups and potential applications. The possible range of applications includes medical visualizations, military visualizations, annotations, educational and entertainment setups. Below some pictures of such applications are shown. (Figure 1)

Figure 1: Augmented Reality applications: Signpost – a mobile AR Navigation System that is able to guide a person through an unfamiliar building (left, see [25]), Construct3D – an application for using AR in mathematics and geometry education (right, see [26])

The approach described in this work is built on the AR system Studierstube (see [4]). Studierstube is a distributed collaborative multi-user AR system, offering several unique properties:

- Collaboration:
  Studierstube allows multiple users to experience a shared virtual 3D workspace filled with multiple applications. Users can not only work simultaneously with applications offered by this workspace, but they can also work together on these applications in cooperation.
  Presentation of the virtual scene is done using see-through head mounted displays or projection systems.

- Multitasking of applications:
  Collaboration is not limited to a single application for multiple users. Multiple applications can be used concurrently by multiple users in the Studierstube environment.

- Distribution:
  Studierstube is designed as a distributed system. It allows users on multiple hosts, which are connected through a network, to participate in a virtual environment. The system manages a distributed shared scene graph to store both graphical and non graphical data (i.e. application state), and takes care of keeping multiple replicas of the scene graph on the participating hosts synchronized. The shared scene graph approach transparently hides details concerning networking functionality from the application programmer. (see [3] and [24])

- Migration:
  The distributed system of Studierstube allows application migration, enabling dynamic workgroup management with support for late joining and early exit of users, load balancing and some degree of ubiquitous computing. A user joining the system on a remote host requires the complete transportation of live applications to the remote host. Since both graphical and application state data is stored in a distributed scene

graph structure, application migration is done transparently by Studierstube's distribution system. (see [24])

1.1 Problem statement

The software of the Studierstube system has been under development for a couple of years, constantly growing in functionality and features. Many extensions were implemented and integrated into the software architecture. Over time this resulted in a very complex software structure rich in features, but difficult to extend. Partially redundant components were incorporated into the Studierstube framework, which made it confusing to understand and complicated to be used as a platform for application development.

Being a basic building block of this software, the Open Inventor toolkit (OIV; see [10]) plays a major role throughout the development of all Studierstube components. OIV is an object oriented graphics library, supplying a rich set objects for developing graphical applications with minimal programming effort. The toolkit offers database-like capabilities for managing objects representing graphical data (called nodes in OIV terminology) and their relationship to each other. This scene database is based on a very sophisticated scene graph concept, allowing easy and flexible data handling.
Additionally the OIV toolkit supports saving the scene database in its own scripting file format, making rapid prototyping of graphical scenes easy without having to modify the source code. The scene can be edited with a simple text editor and loaded into the scene database. OIV delivers certain implementation standards for developing extensions of the toolkit, which automatically inherit all the benefits the library offers. These extensions can directly use the scene graph database and scripting capabilities of the toolkit.

Although Open Inventor offers such a rich functionality, the Studierstube framework partially fails to use it for its purposes, resulting in proprietary solutions and feature limitations.
Application management in Studierstube is limited to loading an application into the runtime system. Studierstube does not offer any feature for keeping

applications persistent by saving their state to a file. An application has to implement such a feature if necessary. While OIV offers scripting capabilities for scene graph structures, it is not possible to use them for application scripting and subsequently rapid prototyping.
Since Studierstube is designed to be a multi user system, it relies on the distribution of applications to other hosts. The present implementations handles this feature in a way more complex than necessary, resulting in bad extensibility and usability. Migrating an application to another host requires to define what application to migrate at sytem startup. The distribution mechanism does not support dynamic migration.

1.2 Proposed solution

We will address the limitations of the current implementation of Studierstube by improving the system in the following ways:

The existing code base will go through a major process of Refactoring. This process will remove the complicated and redundant software components, and try to keep up the already working functionality of Studierstube. One important goal is the backward compatibility of the software interfaces used for application programming, so these will be preserved wherever possible.
Furthermore we will use more of the functionality already offered to Studierstube by the Open Inventor toolkit. The general idea of the concept is to implement applications in a way Open Inventor can handle in its database; an application will be a scene graph node. Bringing together application development and Open Inventor implementation standards will result in applications that can be handled like any other data stored in the toolkits scene database. This approach will offer new application management possibilities, including saving applications and their data to a file and scripting of application nodes.

2 Related Work

2.1 Augmented reality

This work focuses on enhancing the software framework of Studierstube, a system designed for running Augmented Reality (AR) applications. To give an understanding of AR, the following chapters offer some insight to related definitions and technologies.

2.1.1 Definition

A Virtual Environment (VE) or Virtual Reality (VR) fully immerses the user inside a completely artificial, computer generated environment; i.e. the surrounding environment is virtual.
Augmented Reality (AR) is a variation of VR. While users of Virtual Reality cannot see their surroundings, AR users are allowed to do so. They can see the real world with virtual objects superimposed in their field of view. In [8] the basic goal of an Augmented Reality system is defined as: Enhancing the user's perception of and interaction with the real world through supplementing the real world with 3D virtual objects that appear to coexist in the same space as the real world. The virtual objects display information that the user cannot directly detect with his own senses, and the information conveyed by the virtual objects helps a user to perform real-world tasks.

An AR system can be defined to have the following properties (see [8],[9]):

- Blend the real world and virtual objects, in a real environment. Ideally virtual and real object would appear to coexist in the same space.
- It is real-time interactive. Users can interact with virtual objects as they can with their real environment.

- It is registered in 3D. i.e. there is an accurate alignment of real and virtual objects. Correct registration is crucial for the illusion of virtual objects existing in the real world.

This definition of AR is neither restricted to certain display technologies nor limited to the visual sense. In fact AR can be applied to all senses.

In [11] Milgram describes the relationship between Augmented Reality and Virtual Reality by defining a Reality-Virtuality diagram as shown in Figure 2. In this continuum of real and virtual environments Augmented Reality is a part of the Mixed Reality area in the middle. AR lies on the real world end of the continuum resembling the fact that AR is mainly the real world augmented by computer generated data. Augmented Virtuality on the other end of the diagram is defined by Milgram to identify systems that offer a surrounding environment that is virtual and augmented with some real world images.

Figure 2 : Milgram's Reality-Virtuality Continuum

2.1.2 Enabling technologies

The basic technologies needed to build state of the art Augmented Reality systems are displays, tracking, registration and calibration. This chapter describes the status of and some recent advances in this area as outlined in [8].

2.1.2.1 Displays

Display technology is a major limiting factor regarding the development of AR systems. Current displays still lack sufficient brightness, resolution, field of view, and contrast to seamlessly blend real and virtual environments. On the other hand new emerging technologies are not yet small and low-cost enough for being used in a wide range of systems.

There are two major display technology approaches that are used in AR systems to combine the real and the virtual environment: see-through displays and projection displays (see [8] and [9])

See-through Displays

See-through Displays are built as head mounted displays (HMD). While using standard closed-view HMDs does not allow viewing the surrounding real environment, see-through HMDs let the user directly view the real surroundings and additionally show virtual objects superimposed over the users field of view.
The function of these displays is provided by placing optical combiners in front of the users eye. These combiners are both translucent, to allow the user to have a direct view on the real world, and reflective, which further enables the user to see images displayed on the monitors of the HMD that are reflected by the optical combiner. The same functional principle is used in military aircrafts and known by the name of Head-Up Displays (HUDs). Whereas the optical combiners are not attached to the pilots head, but are built into the aircrafts cockpit.
The setup of see-through HMDs using optical combiners has one drawback. The combiners usually reduce the amount of light the user sees from the real world.

Some recent advances in the field of see-through technology include the following as outlined in [8]:

Support for occlusion in optical see-through displays: Conventional see-through displays do not support virtual objects that can completely occlude real world objects. By positioning an LCD in front of the optical combiner, it is possible to opacify selected pixels. (see [12])

Support for varying accommodation: The process of focusing the eye on an object in the distance is called accommodation. Conventional optical see-through displays let the users eyes see the real world with correctly varying accommodation, whereas the virtual part of the scene is seen with fixed accommodation. This conflict between fixed and varying accommodation can result in unwanted eyestrain and visual artifacts. New prototype see-through displays support varying accommodation according to the (virtual) distance between the viewer and the object. It is accomplished by either moving the display screen or a lens in front of it, corresponding to vergence. One prototype is described in [13].

Eyeglass displays: Much research is done to embed displays into conventional types of glasses. The goal of this research is to produce head-worn displays for AR applications, which are not any larger than ordinary sunglasses. Displays that nearly fulfill this goal are produced by MicroOptical (see [14]) and Minolta (see [15]).

Virtual retinal display: While standard see-through displays use some kind of small screens to display the virtual scene, the virtual retinal displays the images directly on the users retina. The display uses a low power lasers to “draw” directly on the retina. Advantages of this approach are high brightness, high contrast, low power consumption, and a large field of view. Virtual Retinal Displays are produced by MicroVision and described in [16].

Projection Displays

Instead of using head mounted displays in AR systems, projection displays can be used to augment the real world objects. They can project the virtual information directly on the physical objects.

If the augmentations are coplanar with the destination surface, they can be projected by a single mounted projector without any additional equipment for the users, such as special eyeglasses. This simple setup can be extended to support large irregular surfaces by using multiple overlapping projectors as shown in [17]. In [18] this approach is enhanced to be used for 3D objects by surrounding them with projectors.

Another approach to augmented reality using projection displays uses head-worn projectors. They project images along the users line of sight at the real world objects. The surfaces of these objects are coated with a special reflective material that reflects incoming light back along the angle of incidence. Since projections can only be seen in the line of the projection, multiple users can see different images on the same object. Additionally it is possible for real objects without reflective coating to obscure virtual objects, if low output projectors are used.

2.1.2.2 Registration

Registration in Augmented Reality is the proper alignment of objects in the real world and the virtual environment. Accurate registration is needed to deliver the illusion of coexisting real and virtual world to the users eyes. Having bad registration will result in serious compromising of this illusion (see [9]).
Registration problems are not limited to Augmented Reality, they also occur in Virtual Environments. In Virtual Environments these are less problematic, because they are harder to detect since the user sees only virtual objects being properly aligned to each other per definitionem. In this case bad registration between the real and virtual world will result in conflicts between different

human senses: visual and kinesthetic senses. These conflicts of different senses can cause motion sickness of users. (see [19])
Furthermore many Augmented Reality applications rely on accurate registration to function correctly and would not be possible otherwise. For example medical applications augmenting a surgeons view with a virtual representation of the patient's inner organs would be useless if the position of the computer generated display is not correctly aligned with the patient's body. It is crucial for the application that the positions of virtual and real organs correspond exactly.

Aside from Augmented and Virtual Reality applications, registration is also an important factor in the field of movie and video production. Actors are seamlessly integrated with computer generated objects. While in Augmented Reality registration has to be done in real time, special-effects artists spend much time to work on each single frame to get a perfect registration. ([9])

2.1.2.3 Tracking

To achieve accurate registration in Augmented Reality applications, it is essential to accurately track the user's location and orientation. An overview on current tracking systems can be found in [20]. These systems commonly use a hybrid tracking method to exploit strengths and compensate weaknesses of individual techniques (e.g. magnetic and optical sensors).
In prepared indoor environments a number of systems delivers good registration. On the other hand, much research has to be done in doing accurate registration in unprepared outdoor setups. This includes research to sense the entire environment, minimizing latency, and reducing requirements concerning calibration. (see [8])

2.1.2.4 Calibration

To produce accurate registration in Augmented Reality a wide amount of calibration is needed. The measurements performed may include camera parameters, field of view, sensor offsets, object locations, distortions, and so on. (see [8]).

2.2 Open Inventor and scene graph API

When implementing an Augmented Reality system such as Studierstube, a key requirement is the capability of producing state of the art 3D computer graphics output at high frame rates.
Regarding the system of Studierstube, which our work is dealing with, this requirement is satisfied by using the Open Inventor library as a basic building block to construct the system's software framework upon it. This library does not only offer sophisticated rendering capabilities, but also supplies many helpful object structures to the software engineer, making rapid development of computer graphics applications possible.
The following chapters shows a short survey on the capabilities the Open Inventor toolkit offers to support computer graphics software development.

2.2.1 The Open Inventor Library

Open Inventor (OIV) is an object oriented 3D toolkit written in C++ that enables graphics programmers and application developers to create interactive 3D graphics applications. The library offers a rich set of objects and methods enabling the programmer to write applications that take advantage of powerful graphics hardware features with minimal programming effort. The objects provided can be used as they are, or modified and extended to meet custom needs.
Since the OIV framework is written in C++, it offers the efficiency of an object-oriented system. Additionally OIV supports the exchange of 3D scene data using a built in interchange file format. (see [1] and [2])

The 3D graphics toolkit was first introduced in [10] by P. Strauss and R. Carey as an object-oriented toolkit for developers of interactive 3D graphics applications. It is designed with the primary goal of making it easier for programmers to implement sophisticated 3D graphics applications that support direct manipulation techniques instead of or in addition to (at the time) conventional two-dimensional widgets. The toolkit supports three major areas regarding graphics application development (as defined in [10]):

- Object representation: All graphical data is stored as editable objects. The collections of drawing primitives representing these objects are encapsulated within them. So applications are able to specify what it is instead of worrying about how to draw it.
- Interactivity: To support direct interactive programming of 3D graphics applications an event model is integrated with the representation of graphical objects.
- Architecture: Although introducing object representation and interactivity, applications should not have to adapt to these policies imposed by the toolkit. Instead, the mechanisms offered by the toolkit should be used to implement the desired policies. This flexibility is directly reflected in the ability of extending the toolkit to meet custom needs.

The Open Inventor toolkit library consists of three main sections – Interaction, Node Kits and the Scene Database, as shown in Figure 3. Additionally there are two utility libraries built on top of the toolkit, providing window objects and the associated event translation to the 3D toolkit library.

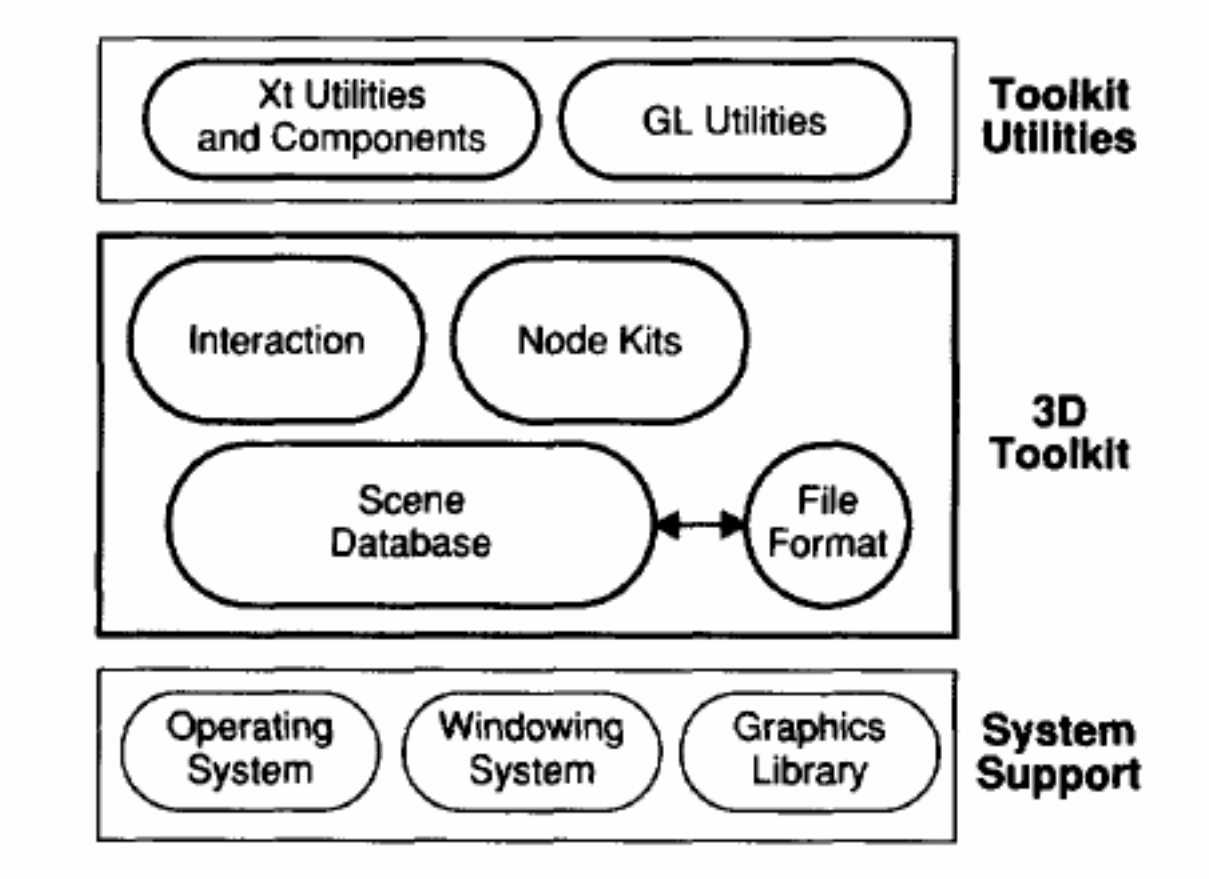


Figure 3 : Open Inventor toolkit system overview

2.2.2 Scene database

The scene database is the foundation of the OIV 3D toolkit. It stores the dynamic representation of 3D scenes as graphs (typically directed acyclic graphs) of objects called nodes, representing the basic building blocks of the database. These graphs are referred to as scene graphs. There are various classes of nodes implementing different geometries, properties, and database traversal behaviors. The database also provides a set of actions that can be applied to scenes or part of scenes; these actions include rendering, picking, computing a bounding box, handling an event, and writing the scene graph to a file. The format and methods for storing scenes in files and retrieving them are defined by the database.

2.2.3 Interaction

The interaction section of the toolkit introduces event classes and smart nodes that process these events. An example of a smart node is the Manipulator node. It responds to interaction events and edits other nodes in the database. A manipulator node typically employs a surrounding object, usually a bounding sphere or box, that represents the manipulator visually and provides a means for translating events into changes to the database. For example a trackball manipulator uses a bounding sphere around an object to modify the rotation of that object. Manipulator nodes provide an easy way for applications to incorporate direct 3D interaction.

2.2.4 Node Kits

The node kit section of the OIV toolkit defines node kit objects, which supply an easy way to create a structured, consistent scene database. Each node kit object combines some scene sub graph, attachment rules, and other policies into a single class; e.g. a Sphere Kit is a wrapper around a sphere node that adds material, geometric transformation, and other properties in the correct place when needed. Node kits allow application programmers to create higher-level objects that encapsulate application specific behavior.

2.2.5 Utility libraries

The OIV 3D toolkit library itself does not include any objects that represent windows to ensure window system independence and greater portability. Built on top of the toolkit, utility libraries tied to specific window systems provide a basic Render Area object to the programmer. This object maintains a window that handles automatic redrawing, translates events from the window system to toolkit events, and distributes events to objects rendered inside the window. Additionally a set of application-level components that implement common interactive functions is provided by the utility libraries (e.g. editors and viewers).

2.2.6 3D toolkit architecture

According to [10] the following paragraphs introduce the most important parts of the OIV architecture.

2.2.6.1 Nodes

The basic building blocks of the OIV scene database are referred to as nodes. Every node in the scene database performs a specific function. There are shape nodes representing geometric or physical objects, property nodes describing various attributes of these objects, and group nodes, which connect multiple nodes into graphs and sub graphs. Camera and light nodes are also provided.
Nodes are designed to allow sharing of common properties when possible. For example, coordinates are specified in separate property nodes that can be shared between multiple shapes. Following this scheme offers the benefit of enforcing consistency of representation.
Special group node classes exist to connect the nodes into scene graphs. The group node classes define how traversal of the group nodes children is performed. Furthermore custom behavior can be implemented.

2.2.6.2 Fields

Within nodes instance specific information is stored in sub objects referred to as fields. Every node class defines a number of fields, where each field is associated with a specific value type. These field objects provide a consistent mechanism for editing, querying, reading, writing, and monitoring instance data within nodes.

2.2.6.3 Paths

The concept of a scene graph allows a node to be a child of multiple group nodes, so that common sub graphs may be shared. This scheme results in more compact and manageable scene graphs in many cases.
On the other side sharing of nodes may result in the inability of referring to an object unambiguously. To overcome this problem, OIV supplies path objects, that refer to a chain of nodes inside the graph, including all nodes below the last node of the chain.

2.2.6.4 Actions

To perform specific operations on a scene the scene graph is traversed by dedicated objects called actions. These operations include rendering, computing a bounding box, searching for a node, or writing scene graph data to a file. An application performs an action on a scene in the database by applying it to a node in the scene graph, which is typically the root node. Additionally actions may also be applied to paths.
The OIV 3D toolkit can not only be extended by the application programmer with new classes of nodes, but also new classes of actions that perform new custom tasks while traversing the scene graph.
When an action is applied to the scene graph it usually results in a traversal of the graph. Although custom traversal behavior may be implemented by new node classes, the standard method of traversing the scene graph is left-to-right and depth first.

2.2.6.5 Sensors

Sensor objects are used to track changes to node data inside the scene graph and to implement simple animation. There are two types of sensors, which both execute a user defined callback function when triggered.
A data sensor is attached to a node and is always triggered when it detects changes of the data inside this node or any node below in the scene graph. Such a sensor can also be attached to the root of the scene graph to track

changes of the entire graph. The second type of sensor is a timer sensor, which is triggered at a specified time or at a regular interval.

2.2.6.6 Node kits

Since there are no strict rules for forming scene graphs, it is possible to create confusing and sometimes meaningless collections of nodes unless some sort of structural guidelines are imposed. Also, class specific traversal and inheritance rules make it difficult to examine a scene graph and determine exactly how sub graphs of nodes relate to “objects” in the 3D scene.
Node kits provide a way to make these tasks easier by enforcing a consistent policy for database construction, editing, and inquiry. Each node kit object effectively contains some structured sub graph of database nodes.

2.2.6.7 Open Inventor notation conventions

The key shown in Figure 4 (as introduced in [10]) is commonly used in the relevant literature and throughout this work to show examples of scene graphs.

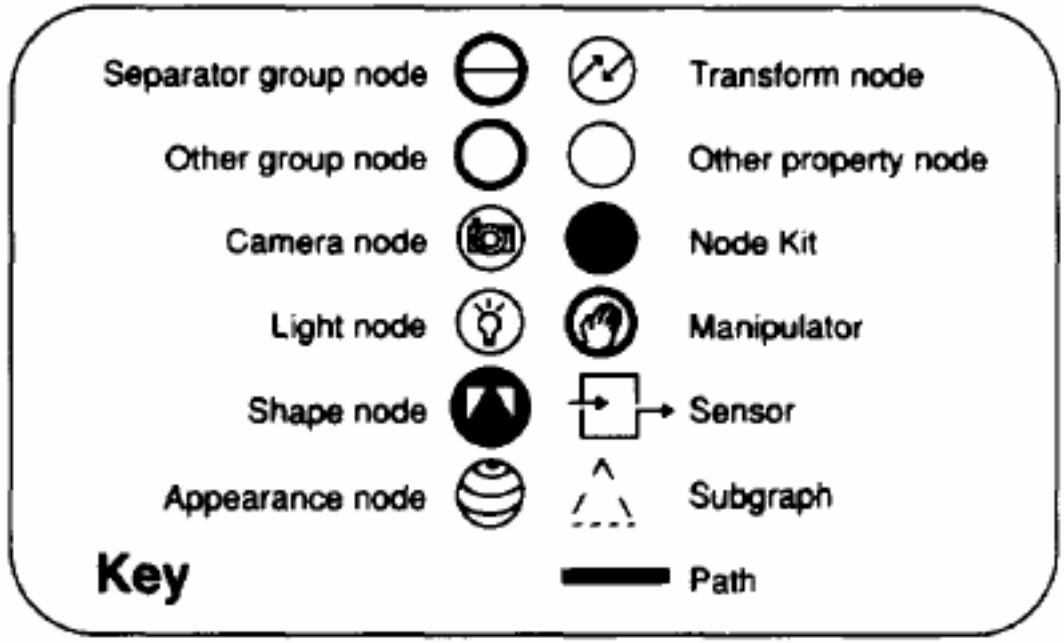


Figure 4 : Open Inventor scene graph notation

2.3 Distributed Open Inventor

Our approach on improving the application management mechanism of the Studierstube system includes the feature of application replication and migration between network connected hosts running the Studierstube software. Fitting perfectly to our solution (see Chapter 3 - Design Issues) we use the already available Distributed Open Inventor extension to accomplish this task of application migration.

2.3.1 Introduction

The Open Inventor toolkit focuses on building real time graphics applications executing on a single host. Each application started has its own scene database, where all used geometry resides. There is no built in support for transparently sharing scene graph data among different applications and furthermore sharing geometric data between different hosts connected through a network.

In [3] an extension of the Open Inventor toolkit called Distributed Open Inventor (DIV) is introduced to address this limitation and supply distribution of scene graph data over a network.

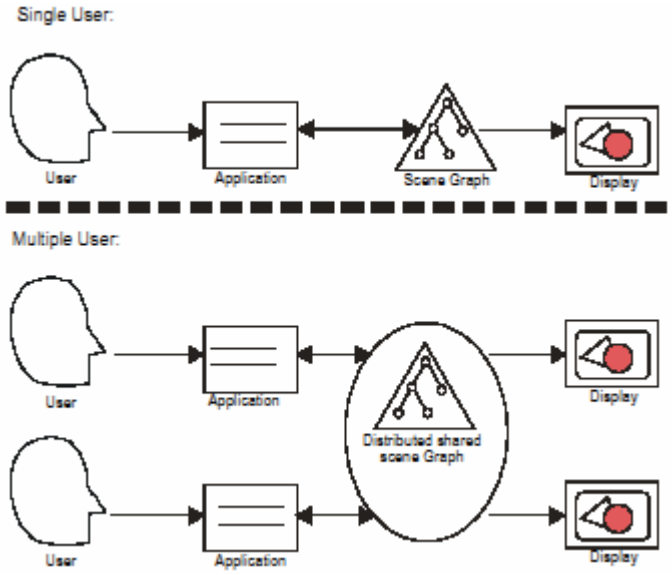


Figure 5 : A single user's view (top) of a graphical application is extended with the concept of a distributed shared scene graph (bottom) used by multiple users.

Similar to distributed shared memory, the scene graph model of OIV is extended by the concept of a distributed shared scene graph. From the application programmers point of view multiple hosts share a common scene graph. (as shown in Figure 5)

2.3.2 Distributed shared scene graph

The concept of a distributed shared scene graph introduced by DIV, has the semantics of a database held in distributed shared memory. It is possible for multiple workstations connected through a network (forming a distributed system) to make concurrent updates to the scene graph, and all these updates are reflected at each workstations view of the scene graph. The scene graph represents the shared state of the distributed system to the application and the users alike.

The DIV extension takes care that all views of the scene graph at the workstations are updated correctly to maintain consistence of the scene graph structure.

Since both OIV and the DIV extension are designed for building real time interactive graphic applications, the way DIV implements the shared scene

graph database takes the following approach: The system relies on replication of the scene graph at every workstation and keeping these replicas synchronized when an update of one replica occurs. So the rendering can be as fast as possible, since all relevant data is stored in each workstations memory.
Although it would be simpler to support a synchronous view of the scene database by storing it only once and then redirecting access to this storage, it would not fulfill the real time interactive requirements of the Open Inventor applications.

When analyzing the communication path for interactive graphics applications, we can identify a loop consisting of the following stages [3]:

- User input
- Computation performed by the application
- Scene graph representation
- Display of the scene graph to the user

Rendering and the processing of input streams are placing the most computational load on each users workstation. Application specific computation tasks can be assigned to another unit. This scenario is typically achieved by setting up a dedicated application server and several rendering clients. The application server stores a master copy of the scene graph and does the application specific computations. Each rendering client stores a replica of the scene graph and renders the image for the user. All updates made to the master scene graph are distributed to all rendering clients that hold a replica using multicasting.
Additionally a system can consist of multiple application servers storing mutually exclusive sub scene graphs. Furthermore clients are allowed to choose whether to replicate all sub scene graphs, or select only some of them.

2.3.3 Scene graph replication

The state of each node of the scene graph is determined by the values of its fields (see [10]). There are basically three operations that can be applied to the scene graphs state:

- Reading the value of a node's field.
- Writing a new value to a node's field
- Changing the structure of the scene graph by adding or removing nodes

When a value of a node's field is read by the application, the state of the scene graph is not changed. Therefore this operation does not invoke the DIV replication mechanism to distribute data to the replicas.

Most of the operations are updates of field values. These continuous updates have to be distributed to the scene graph replicas to keep them synchronized. These fields store basic data types (i.e. numerical values, flags, vectors, matrices). The information needed to perform the update at the replicas is encoded using fixed size messages and then distributed over the network.

Changing the structure of the scene graph by adding or removing nodes represents a special case during replication. There are special messages reserved in the replication protocol for adding and removing nodes. While graphical applications frequently perform field update operations on the scene graph, there are relatively rare changes in the structure of the scene graph. Furthermore it happens very often that more than one node is added, or a whole sub graph respectively. To make this process more efficient, DIV offers a mechanism to load a sub graph from a file at all replicas in parallel.

Nodes are anonymous by default unless given a name by the application programmer. To support node references inside messages, a unique node identifier is required. Therefore DIV offers special messages for naming a node.

2.3.4 Local scene graph variations

Aside from sharing a whole scene graph, DIV also offers a mechanism that allows local variations of the scene graph at each workstation. Using this feature a broad variety of graphical applications can be implemented. These local variations introduce several useful features that can be used by these applications:

- Displaying individual content per user
- Viewing the same data with different attributes by multiple users
- Individual viewpoints
- Editing operation requires local graphics variations including highlighting, selection and dragging

Implementing a partially distributed scene graph requires the client's scene graph to be a superset of the master scene graph, but nevertheless this does not interfere with usability. Implementing a distributed scene graph with local variations is a straightforward process.

2.4 The Studierstube AR System

The software framework of Studierstube is built on the Open Inventor class library and its network extension Distributed Open Inventor.

2.4.1 Introduction

Being a representative of an Augmented Reality system, the Studierstube environment was introduced in [4].
In the Studierstube system three-dimensional stereoscopic graphics are simultaneously presented to a group of users, where each one is wearing a lightweight see-through head mounted display (HMD). These displays do not limit the natural communication and interaction between users, resulting in effective cooperative working together. All Studierstube users see the same spatially aligned model of the virtually augmented environment, but they are able to control their own viewpoint independently. It is also possible to display different layers of data to individual users.

"The mixture between real and virtual visual experience, created in our system by see-through HMDs, is a key feature of our system. Thus, it is possible to move around freely without fear to bump into obstacles, as opposed to fully immersive displays, where only virtual objects can be perceived. This enables a work group to discuss the viewed object, because the participants are seeing one another and can therefore communicate in the usual way."([4])

Figure 6: A typical Studierstube setup: two users collaborate in investigating virtual objects. The objects (cone and sphere in the center) are only visible through the HMDs the users are wearing. (Construct3D application setup, see [26])

In Figure 6 a possible setup of the Studierstube is shown: two users are investigating a virtual object.

2.4.2 Properties of the Studierstube System

Throughout the following paragraphs the key properties of the Studierstube system are described as introduced in [4]. These properties characterize the attributes of the system.

Virtuality and Augmentation

In the environment of Studierstube objects that do not exist or can not be investigated in the real world because or their unique properties, can easily be viewed and thoroughly examined. These investigations can be handled as easy as working with real objects. An object's properties do not limit the investigation process as they would in the real world; e.g. size and other physical properties are not an issue in a virtual environment.

The system also allows augmentation of real objects with spatially aligned (virtual) information. This feature can be used to extend the properties of the real objects; e.g. adding new parts to such an object. On the other hand, additional information regarding the real object (such as descriptions) can be displayed to the user.

Multi-User support

In Studierstube multiple users can work together which can be generally summarized as CSCW (computer supported cooperative work). There are however no special mechanisms needed to support this working together on the side of the software, because normal human interactions such as gestures and verbal communication can be used efficiently in an Augmented Reality setup like Studierstube.

Independence

While certain AR systems rely on a guiding person and limit other users to the role of passive observers, Studierstube offers each user to individually move around and choose a viewpoint. Each user gets a spatially correct stereoscopic view of the augmented environment presented on his HMD. Aside from observation independence interaction with the augmented objects is also performed on a personal basis.

Sharing and Individuality

The objects visible in the augmented environment are shared among the users, meaning that all users see the same model of the objects. Since each user is wearing his own HMD, it is also possible to display different data to each user. This feature allows displaying of different layers of information for different users regarding personal preferences or application specific needs.

Interaction and Interactivity

Studierstube offers support for augmented tools like the Personal Interaction Panel (PIP) as described in [5]. These tools offer a convenient way to explore the visualized object interactively. The components presented on a users PIP can be kept private to this user, i.e. invisible to all other users, or public so that everyone can see them.

2.4.3 Augmented Features

In addition to the general properties outlined in the last chapter, the Studierstube system incorporates special augmented features: the concept of layers, annotations and the use of tracked mobile objects. The following paragraph briefly summarizes these features as described in [4].

Layers and Annotations

The concept of layers separates data into different sets. These sets are assembled by semantic considerations. The display of each layer can be turned on or off individually by the user. Using this feature all users see the same model of data, but everyone sees different aspects of the model according to their personal preferences and needs.
The system also offers the possibility to apply text label to objects which are referred to as annotations. These are linked to a 3D point of the objects and automatically aligned to the display. Furthermore the system takes care that none of the annotations overlap each other. Using layers enables the users to switch annotation on and off.

Tracked mobile objects

The inclusion of static objects into the Studierstube setup is quite simple: the geometric model of the object has to be imported into the system and then correctly registered in the virtual environment.
For complete inclusion of real world objects into the system, additionally to the static properties of the object, all changes in position and orientation have to be reported to the system. Therefore Studierstube supports tracked mobile objects, which can be moved around by the users. The main usage of such objects are manipulation tools such as the PIP or an associated pen.

2.4.4 Interaction tools in Studierstube

Interaction with the augmented part of Studierstube is performed with the Personal Interaction Panel (PIP). The sophisticated interaction tool is described in [5], along with a great number of possible techniques and metaphors to use the PIP for interaction and manipulation inside the Studierstube system.

The PIP consists of a panel and a pen. Panel and pen are both tracked in position and orientation, so that their augmented representations can be presented to the user.
In Studierstube the PIP acts as a multifunctional information display and interaction area for the user. Different kinds of controls for the system are placed directly on the PIP and applications can place their own control widgets on the PIP by defining their own so called PIP sheets. Being held in the users hand, it supplies the user with direct feedback of the position and orientation of the PIP elements in the augmented scene.

3 Design Issues

3.1 Refactoring of existing code

Due to its long time ongoing development, the Studierstube Augmented Reality system has become a very complex formation of many different components working together. Especially the parts of Studierstube concerning application management have evolved to be a confusing element for application programmers, because of the high number of components involved.
To overcome the current complexity of the application management mechanism and its flaws, we introduce a new approach to get the system more “light weight”, flexible and easy to use for the application programmer and user alike.

Tough the software architecture of Studierstube is very complex, it is a “ready to run” system offering many features. To preserve these features and keep the system in a working state, we use methods of Refactoring to transform the system’s code base into a more suitable structure.

“*Refactoring*” can be defined as being the process of changing a software system in such a way that it does not alter the external behavior of the code but improves its internal structure (see [7]). It is a disciplined way of cleaning up the code that minimizes the chance of introducing new bugs into the code. When Refactoring is applied to a piece of code its design is improved after it has been written.

When developing software usually the first step is designing and after that the actual code writing begins. During the ongoing development the code will be modified over and over again, resulting in a fading integrity of the code that was well designed in the first place. The state of the software system is slowly transformed from engineered to hacked.

The process of Refactoring works in the opposite way. Using Refactoring it is possible to take a bad design with chaotic lines of code and rework all this into well designed code.

3.2 Building on Open Inventor

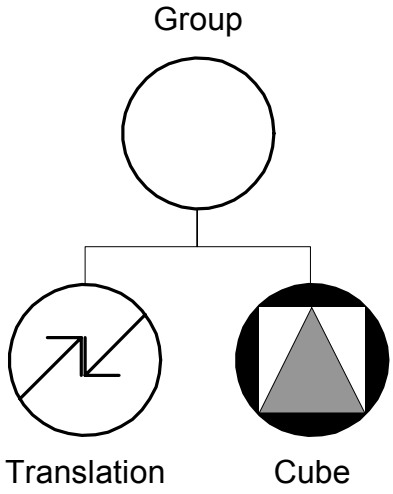
When doing an overview of the Studierstube software architecture we can find one important component being the fundamental building block of the system. It is the Open Inventor class library (see [1], [2]). The OIV 3D-toolkit supplies the system with the ability to render complex three-dimensional scenes for users in realtime, but it is much more involved in the system than just being a rendering engine. Apart from rendering, OIV functionality is already responsible for e.g. event handling, interaction and user management in Studierstube.

The cause for OIV being so highly involved in the Studierstube software design can easily be found by doing a quick survey on the mechanics of the toolkit: Open Inventor offers a database to store data objects, referred to as nodes. These nodes are arranged to each other in such a way that a so called scene graph is constructed. This scene graph structure usually represents the scene with all its geometric data. The nodes stored in a scene graph can then be accessed by traversing the graph beginning at the root node. Following the visitor pattern as outlined in [6] every node in the scene graph is accessed during the traversal and special methods inside each node are invoked. This is exactly what is done to render the scene. The same mechanism is used to distribute events through the scene graph to reach the nodes and invoke a special behavior there.

The clue of Open Inventor's design is, that the mentioned database functionality supplied by the scene graph is not limited to the purpose of storing data related to rendering. It can easily be extended to support special purpose nodes unrelated to rendering that can be stored and accessed in the same way as graphical data. Such a node can be implemented to only react to specific traversals, e.g. ignore any rendering traversal of the scene graph.

Another integral functionality of OIV is the ability to read and write scene graph data. Using this function we can generate an ASCII-file containing a linear form of the hierarchical scene graph. Generating a linear form of an OIV scene graph, works similar to Object Serialization in the Java Programming Language (see [22] and [23]). What makes this feature so powerful is, that it does not simply enable saving and loading a complete scene graph, but also enables us to manipulate the saved scene graph using a common text editor. So it is possible to construct complex scene graphs and pass the resulting files to the application without the need for writing new source code. It is obvious that using this feature considerably reduces the time needed for developing an OIV application.

In Figure 7 a sample part of a scene graph is shown together with its accompanying ASCII file format. Files containing OIV scene graph data are referred to as .iv-files in OIV terminology. When applying and OIV save-function to the scene graph shown on the left of Figure 7, the corresponding file is automatically constructed. Vice versa the correct scene graph with all its objects is constructed when the OIV reading-function is applied to the .iv-file.




```
Group {
        Translation {
                translation 1.0, 2.0, 0
        }
        Cube {
                width 2.0
                height 3.0
                depth 3.0
        }
}
```

Figure 7 : A sample scene graph shown in Open Inventor notation (left) and its corresponding file format representation (right).

Concluding the short overview of Open Inventor, we notice that OIV is very well capable of handling data objects and relations between those objects in a flexible way, using the concept of a scene graph. In addition OIV can be extended to manage data unrelated to its prime mission, rendering three-dimensional computer graphics.

3.3 Redesigning application management using Open Inventor

Considering the possibilities offered by the OIV library on one hand and the current Studierstube application management mechanism on the other hand, we introduce the following approach as being the next logical step: We move the current implementation from its legacy foundation to an OIV implementation, that offers all the previously mentioned features to the application management system:

Applications are now implemented as OIV nodes to enable management using a scene graph structure and all its advantages.

Application nodes being parts of a scene graph implicitly enable us to represent these nodes using a .iv-file. This scene graph and file dualism automatically supplies us with the following powerful possibilities:

- Loading applications from a .iv-file
- Saving the application state to a file
- Use of .iv-file scripting together with application nodes
- Migration of applications using DIV

Fitting this new design every Studierstube application is now being accompanied by an appropriate **.iv-loaderfile** containing a description of the application node plus additional data. This file is loaded by the Studierstube system to start the application using the built in OIV loading functionality.

3.4 Parts of an application node

Studierstube applications are now implemented as OIV nodes following our new design. An application node consists of the following components:

- SoApplicationKit wrapper
- SoClassloader node
- SoContextKit, containing the application code
- Pip sheet geometry
- Window geometry
- Miscellaneous application data

These parts of an application node form the appropriate loader file used to start the application in the Studierstube environment. An example of a loader file can be seen in Source Sample 1.

```
# application kit wrapper node
DEF RB SoApplicationKit {

      # read-only flag protecting the loader file from being
overwritten
      readOnly TRUE

      # class loader node
      # loads and initializes a node class from a DLL
      classLoader SoClassLoader {
                  className   "SoRedAndBlueKit"
                  fileName    "../apps/redandblue/redandblue_stb"
      }

      # context kit node containing the application code
      contextKit  DEF REDANDBLUE SoRedAndBlueKit {

            # template geometry of the application's pip sheet
            # (here the pip sheet contains 3 buttons)
            templatePipSheet  Separator {
                              RotationXYZ {axis  X angle 1.57 }
                              DEF RED_BUTTON So3DButton {
                                    width 5 depth 5 height 2
                                    buttonColor 1 0 0
                              }
                              Translation { translation 10 0 0 }
                              DEF BLUE_BUTTON So3DButton {
                                    width 5 depth 5 height 2
                                    buttonColor 0 0 1
                              }
                              Translation { translation 10 0 0 }
                              DEF SAVE_BUTTON So3DButton {
```

```
                                    width 5 depth 5 height 2
                                    buttonColor 0.5 1 0.5
                                }
                            }
            # the application's window geometry
            # (here an empty window is supplied)
            windowGroup Group {
                                    SoWindowKit {}
                                }

            # clone flag
            # (when set, the pip sheet geometry is copied for every
user,
            # otherwise all users use the same instance of the
geometry)
            clonePipSheet FALSE
      }

      # additional application geometry (application icon)
      # represents the application on the pip sheet similar to icons
      used
      # on 2-dimensional desktops.
      appGeom     Separator {
                  Texture2 { filename
                  "../apps/redandblue/redandblue.gif" }
                  }

      # info node holding special application information.
      info        Info {}
}
```

Source Sample 1 : An application loader file example

In Source Sample 1 all important components of an application are shown. To give a short overview of their function, they are described in more detail in the following paragraphs:

3.4.1 SoApplicationKit

The SoApplicationKit node kit class functions as a wrapper class that encapsulates all OIV nodes that construct the application. It is the node that is actually read from the loader file and attached to the scene graph.
The following data is stored in the SoApplicationKit class:

- readOnly

  A flag to protect the loader file from being overwritten, when the application is saved to a file. When the flag is set, a filename for the save

file is generated, to save the loaderfile for later use. Otherwise the loader file is overwritten.

- classLoader
  A special node that allows dynamic loading of libraries (dynamic link libraries or shared objects) when reading the scene graph from a file. These libraries hold the binary code of the application nodes.
- contextKit
  The application object loaded by the class loader. It contains window and pip sheet geometry.
- AppGeom
  Additional geometry associated with the application. This field is used for storing an icon specific to the application. It is used to represent the application in the system similar to icons used in the Windows operating system.
- Info
  This node may be used to store additional information for the application.

3.4.2 SoClassLoader

The SoClassLoader node enhances the loading function of OIV to be more flexible. Using this node it is possible to load a custom implementation of an OIV node from a dynamic link library. The filename of the library and the name of the node class can be supplied to the node by passing the names to the appropriate fields.

3.4.3 SoContextKit

The SoContextKit node kit class represents the actual application node containing the application's functionality. SoContextKit acts as a base class to be used by the programmer to implement a custom application for the Studierstube system by deriving a new subclass from SoContextKit.
Additionally the SoContextKit is supplied with window geometry associated with the application and pip sheet geometry. The usage of pip sheet geometry can either follow the paradigm of one shared instance used by all users of the

application, or take place as using the supplied geometry as a template for separate instances of the geometry passed to the user. When using a shared instance of the pip sheet geometry, pip interactions of one user are reflected to all the other users pips; e.g. one user moves a slider on the pip and the sliders on all the other pips instantly move in the same manner.
Both window and pip sheet geometry are usually passed to the application using the loader file, but it is also possible to hardcode the construction of the geometry into the application node, though it is not recommended.

3.5 Application management components

3.5.1 SoContextManagerKit

The node kit class of SoContextManagerKit functions as a managing class for application nodes. It manages the part of the scene graph where the application nodes reside. Furthermore it supplies methods for loading, saving, starting and stopping applications to the programmer. Together with DIV (see [3]) the SoContextManagerKit class manages network distribution of applications transparent to the user.
The design of the management class follows the paradigm of a singleton instance as described in [6].

3.5.2 SoUserManagerKit

Following a similar design as used for application management, user management is performed by a singleton instance SoUserManagerKit node kit class. All data associated with a Studierstube user is encapsulated in an OIV user node kit and stored in the scene graph.
User management components are already in use by the previous version of the Studierstube software and need only slight modifications to fit the new application management.

3.6 Approaching the new design

When planning a practical path for realizing a new implementation of the Studierstube application management, we have to start with analyzing its current structure as shown in Figure 8.

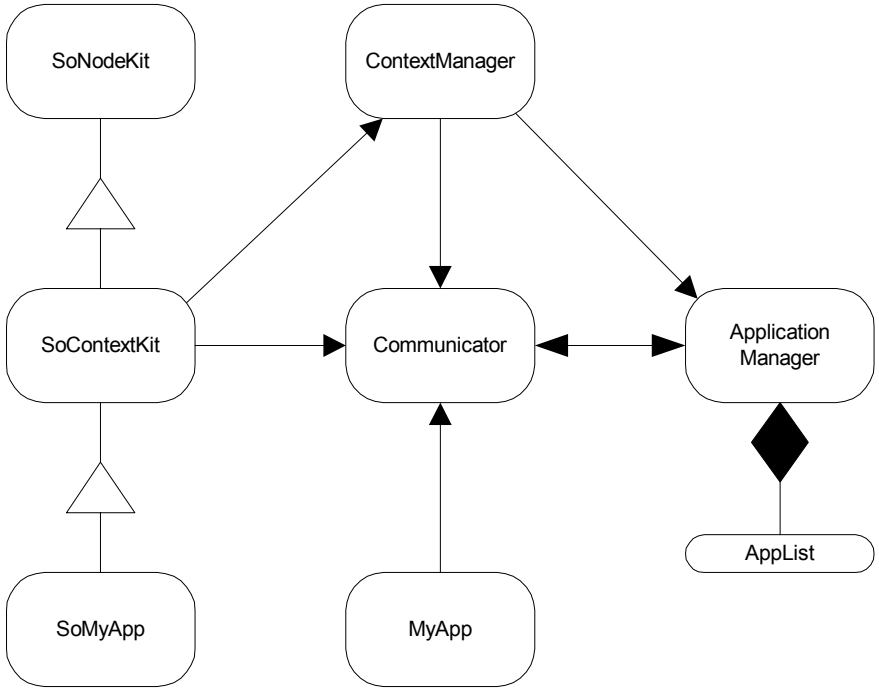


Figure 8 : Application management components and their interaction paths.

In Figure 8 we can see three groups of components: management components (Context Manager, Application Manager, AppList), legacy application support (Communicator, MyApp) and Open Inventor style application support (SoContextKit, SoMyApp). Though already based on an OIV base class in the old version of Studierstube, applications are not managed by the OIV database nor do they take advantage of their inheritance.

The system diagram shows pretty good the evolution of the Studierstube system resulting in a complex communication structure. Although very complex in its structure, it represents an already working software architecture. Taking this into account we will reuse very much of the present code by using methods of Refactoring (see [7]).

In our new design everything shown in Figure 8 will be transformed and reduced to the structure shown in Figure 9. We will no longer support legacy

application design, which was introduced in an early version of Studierstube. The new SoContextManagerKit component takes over the task of application management. Communication with the management component is handled in a direct way, eliminating the communicator class. Applications are no longer managed using the AppList structure, instead we use the scene graph structure for handling applications.

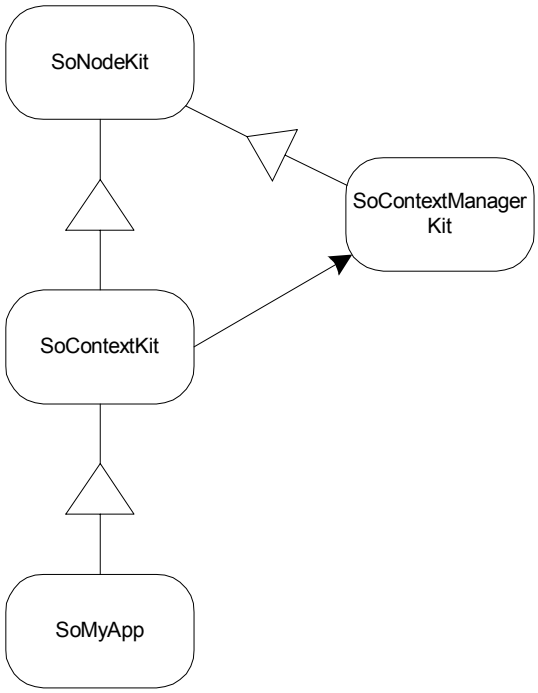


Figure 9 : Structure of the application management components in our new approach.

4 Implementation

4.1 Studierstube components involved in application management

The following chapter describes the components (C++ classes) of the Studierstube system that are involved in application management. All of them are OIV classes and therefore inherit OIV class functionality.

4.1.1 SoContextManagerKit

The SoContextManagerKit class is implemented as an Open Inventor node kit class. Following the design pattern of a singleton instance, there is only one instance of this class active at any time in the application management system (see [6]). It is responsible for loading and saving Studierstube applications, keeps track of all applications currently active in the system, and dynamically manages the applications status (master or slave status).
In Figure 10 the internal structure of SoContextManagerKit is shown. Depending on their state, applications are either attached to the masterContexts group node or slaveContexts group node.

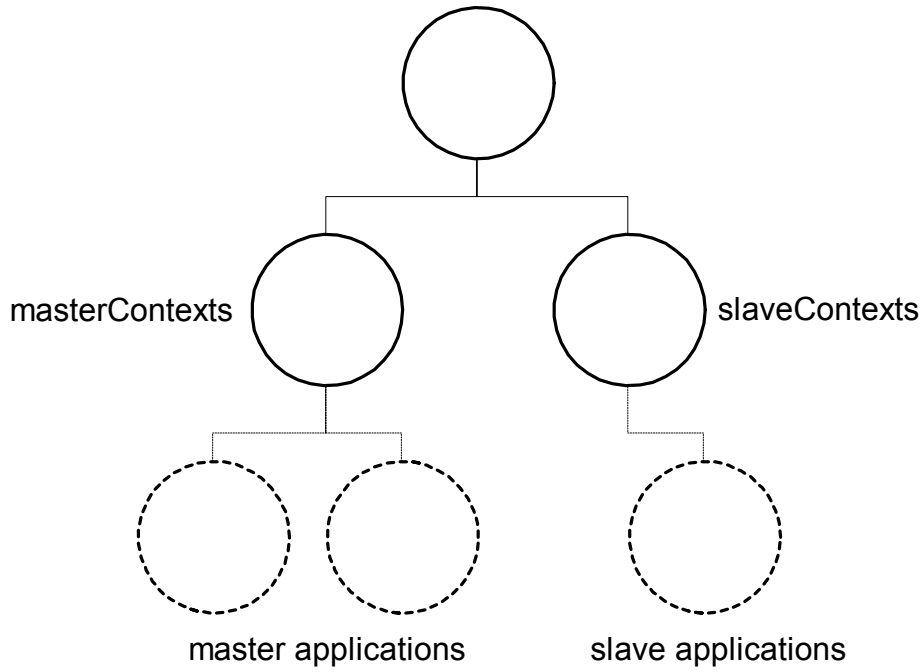


Figure 10 : Internal structure of the SoContextManagerKit node kit class, containing group nodes for master and slave applications.

A selection of SoContextManagerKit's methods:

- **getInstance**()
  Get the singleton instance of the SoContextManagerKit class. Using this method is the only way to correctly generate an instance of this class, since the constructor function is declared as a private function.
- **setMasterMode**(SoDivGroup *appGroup, SbBool isMaster)
  Set an applications mode to either master mode or slave mode. If the application is not known by SoContextManagerKit, it is added to the list of managed applications.
- **registerPipSheets**(int uid)
  When a new user is added to the system, this method is called to make the pip sheets of existing applications available for this user.
- **loadApplication**(SbString fn, SoNodeList &lst, int uid, SbName &locale)
  This method is used to load an application from a .iv loader file *fName* into the Studierstube system. After successful loading, *aList* contains a list of all the applications loaded from the file *fName*.

- **saveApplication**(SbAppID appID)
  Save the application back to its loader file. The application will be saved together with all its windows including content geometry to the file it was loaded from. If the readOnly flag of the application's SoApplicationKit is set to TRUE, the *saveApplicationAs* method is used instead.
- **saveApplicationAs**(SbAppID appID)
  Save the application to a new file. The filename is automatically generated by extending the filename of the loader file with "_xx". (with "xx" = 00, 01 … 99)
- **saveAll**()
  Save all applications currently loaded to files.
- **stopApplication**(SbAppID appID)
  Close and unload an application from the Studierstube system.
- **shutDown**()
  Close and unload all applications and shut down Studierstube.

4.1.2 SoClassLoader

To support the needs of application loading the loading mechanism of OIV has been extended. The original loading function of OIV does the following when loading a scene graph from an .iv-file:

- Load the name of the new class from the .iv-file
- If the class is already registered in the OIV database then construct an instance of this class
- If the class is not know to the OIV database, it tries to find the initClass() method of the class in order to register it to the database. During this process OIV searches the current working directory and the defined PATH for a library with a filename matching the name of the class found in the .iv-file.

It is obvious that the limitation of the loading function to the working directory and the directories that are stored in the PATH variable has to be overcome. Therefore the SoClassLoader node has been introduced. This node supports loading a user defined OIV class that is unknown to the OIV database from an .iv-file.

Fields of SoClassLoader:

ClassName

Name of the class to be initialized

FileName

Filename of the dynamic link library containing the class (including path)

When a SoClassLoader node is read from a file, it loads the DLL defined in the Field *fileName* using standard operating system functionality (e.g. WIN32API:loadlibrary()). After successfully loading the library, the initClass() method of the class defined in the *className* field is called to initialize the class and add it to the OIV databases list of known classes. From the point where the given class is registered to the OIV database in this way, it is possible to read an instance of such a class from a file without further arrangements.
Using this extended method of registering OIV classes when loading them from a file, it is also possible to compile multiple classes into one DLL, instead of compiling one DLL for every custom class and choosing the filename after the class name, as the standard OIV approach would be.
From the users point of view the SoClassLoader node has to be placed in the .iv-file or the scene graph respectively before the occurrence of the unknown new class. An example can be seen in Source Sample 2 with the accompanying scene graph shown on Figure 11.

```
Group {
     SoClassLoader {
          fields [ SFString className, SFString fileName ]
          className     "SoMyNodeKit"
          fileName"../apps/test/mynodekit"
     }
     SoMyNodeKit {
            …
     }
}
```

Source Sample 2 : File format usage of a class loader node (see Figure 11)

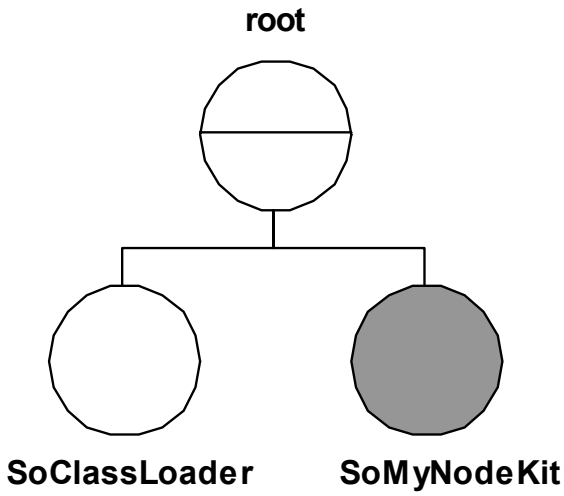


Figure 11 : Scene graph using SoClassLoader to initialize SoMyNodeKit (see Source Sample 2)

4.1.3 SoApplicationKit

The Application Kit class is a wrapper node kit that encapsulates everything that defines an application. It incorporates a SoClassLoader node to load and initialize the application node kit derived from SoContextKit. The node kit structure also stores an application geometry that is used as an icon that represents the application in the system and a SoInfo node to store additional information the program can use.

The SoApplicationKit node kit also represents the structure that is used to load and start an application from a file to the Studierstube system. Every Studierstube application is accompanied by an .iv-file containing the SoApplicationKit with the correct entries to load the application. The node kit structure of SoApplicationKit is shown on Figure 12 and the OIV file format

definition of a SoApplicationKit that can be used to load an application is shown on Source Sample 3.

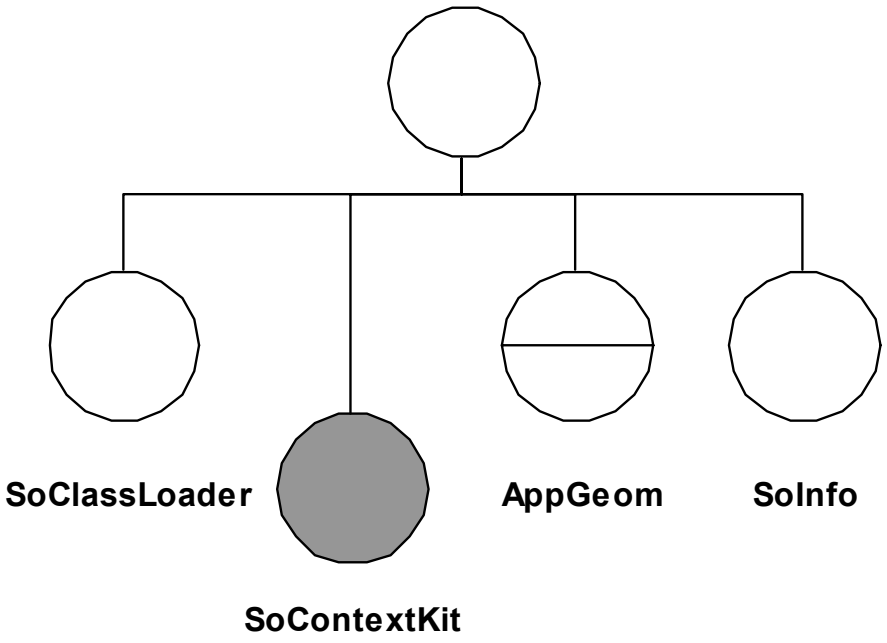


Figure 12 : SoApplicationKit internal node kit structure.

```
# SoApplicationKit
# file format representation

DEF SP SoApplicationKit {
      fields [ SFBool readOnly,SFNode classLoader, SFNode contextKit,
               SFNode appGeom, SFNode info ]
      readOnly TRUE
      classLoader SoClassLoader {
            fields [ SFString className, SFString fileName ]
            className   "SoSprayingKit"
            fileName    "../apps/spray/spray_stb"
      }

      contextKit  DEF SPRAY SoSprayingKit {
                        …
                  }
      appGeom     Separator {
                        Texture2 { filename
                  "../apps/spray/spraycan.gif" }
                  }
      info        Info {
                  }
}
```

Source Sample 3 : File format representation of a sample SoApplicationKit node kit. This representation is used as a loader file, to load an application into the Studierstube system.

Fields of SoApplicationKit:

readOnly

This Boolean field is used to mark the loader file as write protected. When it is set to TRUE the Studierstube system's saving function does not overwrite the loader file. A new file with a name extension of "_xx" is created instead.

4.1.4 SoContextKit

The SoContextKit class is the most interesting class for the programmer who wants to implement his own Studierstube application. It is the basic foundation class that is used to derive new applications from that fit the Studierstube system environment.

It incorporates a group node that holds all windows of the application (see Figure 13). These SoWindowKits can be passed as geometry from the OIV loader file, which is the preferred way since the application derived from SoContextKit is part of the loader file, or placed there by the application's functions itself.

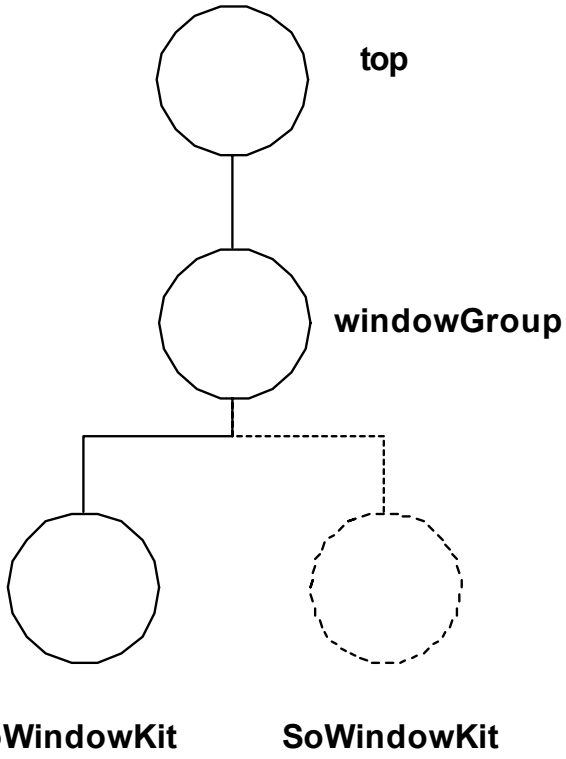


Figure 13 : SoContextKit node kit internal structure.

Fields of SoContextKit:

templatePipSheet

The field contains a geometry template of the pip sheet used by the application.

clonePipSheet

FALSE	every user accesses the same instance of the pip sheet
TRUE	the pip sheet is copied for every user separately.

The following selection describes some of the methods of SoContextKit. These methods are used to enable communication with SoContextManagerKit and are usually not used by a Studierstube application programmer:

- **initContext**(SbBool masterMode)
  This method is usually called by SoContextManagerKit to register the application with the application management. It invokes geometry checking, registration of the pip sheet geometry for each user, initializes the applications state as master, and switches the focus to the application.
- **exitContext**()
  Cleanup method called by SoContextManagerKit before the application is actually removed from the system. It closes all windows associated with the application.
- **registerPipSheet**(int uid)
  Registers the application's pip sheet geometry for a user's pip. The application is integrated into the pip taskbar, which allows the user to switch the pip geometry between different applications.
- **forceFocusChange**()
  When this method is called, the input focus is set to this application. Furthermore the pip sheet geometry displayed on the pip, is switched to this application's pip sheet.
- **setMasterMode**(SbBool masterMode)
  Handles changes of the application's state. It is usually called by SoContextManagerKit when the state of the application is changed from master to slave or vice versa.

SoContextKit incorporates some methods which are exclusively used to be overwritten by the application programmer in order to supply the application with custom behavior. These methods are listed in the following paragraph along with a short description:

- **checkWindowGeometry**()
  This method is called when the application is registered in the application management system. It is used to check the correctness of the window geometry passed to the application from the loader file in respect to the application's needs.
- **checkPipGeometry**()
  Like checkWindowGeometry() this method is used to check the pip geometry passed from the loader file upon registration of the application in the system.
- **checkPipMasterMode**(SoNode * pSheetContents, SbBool mMode)
  When the state of the application is changed, this method is invoked to let the application react in a defined way to that event; e.g. disable controls of the application when slave mode is enabled.
- **checkPipConnections**(int uid, SoNode * pipSheetContents)
  Especially when using application scripting with field connections between the pip sheet widgets and the application, cases of missing connections can occur. This is the case when the pip sheet template is copied for multiple users. Therefore this function was integrated to allow setting of the correct connections after the application was loaded.
- **focusChange**(int uid)
  To give the application a custom behavior when its focus is changed, this method can be overwritten by the programmer. It is called whenever the application's focus changes.

4.1.5 SoUserManagerKit

Similar to SoContextManagerKit, which manages applications, SoUserManagerKit is used to manage all users of the Studierstube system. It supplies several methods for managing SoUserKit node kit classes, such as adding users to and removing users from the system.

4.1.6 SoUserKit

The SoUserKit node kit defines a user of the Studierstube system and his resources, including pen, pip and display. The appropriate user kits are loaded from an .iv-file on startup. Furthermore users can be added at runtime by loading the users .iv-file using a method of SoUserManagerKit.

5 Results

5.1 Writing an application for Studierstube

The following chapter describes how to write a custom Studierstube application. To demonstrate the procedures involved, a simple spraying application is constructed step by step until ready to be loaded into the Studierstube system.

Creating a Studierstube application basically involves the following steps:

- Select a name for the new application and derive a new node kit with this name from SoContextKit as shown in [2].
- Add OIV fields to the node kit to store the applications data.
- Overwrite methods inherited from SoContextKit.
- Add your custom application's functionality.
- Compile the new application class into a DLL or SO respectively.
- Generate an appropriate .iv-file to load the application into the Studierstube system.

5.1.1 Deriving a new application node kit from SoContextKit

The first step when writing your own Studierstube application is to derive a new OIV node kit class from SoContextKit. So the new application automatically inherits all OIV node functionality to fit the scene graph database structure. Creating the application node kit follows the standard OIV procedure of node kit creation as described in [2].

We start the development of our small spraying application with the creation of a class we call SoSprayingKit. Source Sample 4 shows the header file containing the class definition with the basic methods needed to be overwritten when implementing a new application node kit. The class definition shown includes an Open Inventor specific node kit macro, constructor and destructor

functions, the newly introduced geometry checking functions, and an event callback function. At first there are no methods added that give the application its own custom functionality.

```
//
// SoSprayingKit class definition
//

class SoSprayingKit : public SoContextKit
{
      // OIV node kit macro
      SO_KIT_HEADER(SoSprayingKit);

public:
      // OIV class initialization method
      static void initClass();

      // Constructor
      SoSprayingKit();

      // virtual constructor
      SoContextKit* factory();

      // Destructor
      ~SoSprayingKit();

private:
      // Pip geometry verification method
      SbBool checkPipGeometry();

      // window geometry verification method
      SbBool checkWindowGeometry();

      // user defined function implementing the behaviour when the
      // applications (master/slave) mode changes
      virtual void checkPipMasterMode(SoNode * pipSheetContents,
      SbBool masterMode);

      // event callback function
      virtual SbBool windowEventCB(
            void* data,
            int messageId,
            SoWindowKit* win,
            int uid,
            So3DEvent* event,
            SoPath* path
            );
};
```

Source Sample 4 : SoSprayingKit class definition

5.1.2 Adding fields to store data

Since Studierstube applications are implemented as OIV node kits, there are some topics to keep in mind during application designing. To get the full advantage of being an OIV node kit, the program has to store important data concerning the state of the application in OIV fields. Loading or saving an application is based on the possibility of OIV to write a scene graph to a file or read it from there. Since only data stored in fields of a node is written to a file together with the node, it is imperative to store data necessary for saving the application state in fields. Furthermore network functionality of Studierstube also depends on this design, since distribution of an application to another host using DIV can be seen as saving the application state first and reload the saved state on the other machine.
On the other hand, it is also a very flexible way of passing data to the application through fields, since they can simply be filled in the .iv-file that is used to load the program into the Studierstube system.

Following this design guideline we add two fields to our class definition as can be seen in Source Sample 5. The *voxelField* is an integer field containing the number of voxels already sprayed in the window and *templateMesh* stores the geometry of a single voxel that is used as a template for the voxels being sprayed into the window.

```
class SoSprayingKit : public SoContextKit
{
      …
private:
      …
      // number of sprayed voxels inside the window
      SoSFInt32 voxelField;

      // geometry template of a voxel
      SoSFNode templateMesh;
};
```

Source Sample 5 : Adding fields to the application class definition

5.1.3 Overwriting inherited methods and adding functionality

Our small spraying application will allow multiple users to use their pen to either draw single voxels inside a Studierstube window or spray several voxels at once. The user will be able to adjust the color used for drawing and spraying. Also the spraying radius will be adjustable. All voxels will be stored in a SoIndexedTriangleStripSet node.

We will start filling the framework we just set up, with our custom code, beginning with adding two variables for our convenience. They are just used to keep the source code more readable, it would also be possible to get hold of the correct values by evaluating a bunch of fields every time necessary. As shown in Source Sample 6 we add the variables *mesh* and *vp* to store a pointer to the voxel mesh and its property node to finish the class definition of SoSprayingKit.

```
class SoSprayingKit : public SoContextKit
{
      …
private:
      …
      // pointer to the voxel mesh
      SoIndexedTriangleStripSet* mesh;

      // pointer to the property node of the voxel mesh
      SoVertexProperty* vp;
};
```

Source Sample 6 : Adding private variables to the application class definition

In Source Sample 7 we begin the actual implementation of SoSprayingKit. Apart from standard OIV node kit implementation we add definitions of constants to identify the current drawing mode and supply initialization for the *mesh* and the *vp* variable through the constructor. The virtual constructor has no special functionality here and just returns a new instance of SoSprayingKit.

```
// OIV node kit source macro
SO_KIT_SOURCE(SoSprayingKit);

const int DRAWING_MODE  = 0;
const int SPRAYING_MODE = 1;

// static class initialization

void
SoSprayingKit::initClass(void)
{
      SO_KIT_INIT_CLASS(SoSprayingKit, SoContextKit, "SoContextKit");
}

// constructor

SoSprayingKit::SoSprayingKit()
:
mesh(NULL),
vp(NULL)
{
      SO_KIT_CONSTRUCTOR(SoSprayingKit);
      SO_KIT_ADD_FIELD(voxelField, (0));
      SO_KIT_ADD_FIELD(templateMesh, (NULL));
      SO_KIT_INIT_INSTANCE();
}

// destructor

SoSprayingKit::~SoSprayingKit()
{
}

// virtual constructor

SoContextKit*
SoSprayingKit::factory()
{
      return new SoSprayingKit();
}
```

Source Sample 7 : Implementation of basic application node methods

When deriving a new class from SoContextKit the following methods have to be overwritten with custom code to integrate the application with the Studierstube system:

- `SoContextKit::checkWindowGeometry()`
- `SoContextKit::checkPipMasterMode(SoNode *pipSheet, SbBool masterMode)`
- `SoContextKit::checkPipConnections(int uid, SoNode *pipSheet)`
- `SoContextKit::checkPipGeometry()`

The *checkWindowGeometry()* method is called when loading the application from a .iv-file to allow checking of the supplied window geometry. Our implementation checks if there is actually a window and if none is found it creates a standard Studierstube window. Furthermore the *mesh* and *vp* pointers are set to the correct value. In case of a supplied window geometry a more sophisticated error checking could be implemented here. It is assumed that the geometry is correct for the needs of the application here for sake of simplicity. (Source Sample 8)

```
//
// window geometry verification method
//
// It enables the program to check the validity of the window geometry
// supplied to the application by the loader file
//

SbBool
SoSprayingKit::checkWindowGeometry()
{
      SoGroup *wGroup = (SoGroup*)windowGroup.getValue();
      SoWindowKit *windowKit;

      // Is a window supplied in the loader file?
      if (wGroup->getNumChildren() == 0)
      {
            // NO window supplied: create a new window

            windowKit = new SoWindowKit;

            windowKit->size.setValue(0.5, 0.5, 0.5);

            // add a property node and the voxel mesh to the new window

            SoSeparator* clientVolume = windowKit->getClientVolume();
            vp = new SoVertexProperty;
            vp->materialBinding = SoVertexProperty::PER_VERTEX_INDEXED;
            mesh = new SoIndexedTriangleStripSet;
            mesh->vertexProperty = vp;
            clientVolume->addChild(mesh);

            // initialize the voxel counter field
            voxelField.setValue(0);
      }
      else
      {
            // window geometry is supplied by the loader file

            // …
            // we assume that the supplied geometry is correct in this
            // case for sake of simplicity, but nevertheless further
```

```
            // testing could be implemented here.
            // …

            windowKit = (SoWindowKit*)wGroup->getChild(0);
            SoSeparator* clientVolume= windowKit->getClientVolume();

            // initialize the private variables
            mesh = (SoIndexedTriangleStripSet*)(clientVolume-
      >getChild(0));
            vp = (SoVertexProperty*)(mesh->vertexProperty.getValue());
      }
      return TRUE;
}
```

Source Sample 8 : Implementation of the checkWindowGeometry() method

The *checkPipGeometry*() method is similar to checkWindowGeometry() but it allows checking of the Pip geometry passed from the loader file. In our implementation of the spraying application we assume that the Pip geometry is correct and therefore this method does only return the value “true”. (Source Sample 9)

```
//
// Pip geometry verification method
//
// By implementing this method, the program is able to verify if the
// Pip geometry passed from the loader file is valid.
//

SbBool
SoContextKit::checkPipGeometry()
{
      // we assume that the Pip geometry passed to the application is
      // valid in this demo implementation.
      return TRUE;
}
```

Source Sample 9 : Implementation of the checkPipGeometry() method

Every time the mode of the application changes (master or slave mode) the *checkPipMasterMode*() method is called to set up the Pip according to the current mode. We use this method in our spraying application to activate and deactivate the clear-button by adding and removing the callback function that is attached to the button. (Source Sample 10)

```
//
// Method that implements the Pip's reactions to changes of the
// applications (master/slave) mode.
//
// It is called whenever the applications mode changes.
//

void
SoSprayingKit::checkPipMasterMode(SoNode* pipSheet, SbBool masterMode)
{

      So3DButton* clearButton;

      // find the spraying applications "clear" button on the pip
sheet
      clearButton =
(So3DButton*)findNode(pipSheet,"SPRAY_CLEAR_BUTTON");

      // Is the application currently running as master?
      if(masterMode)
      {
            // YES: master mode is active
            // we enable the button on the pip
            clearButton->addReleaseCallback(clearButtonCB,this);
      }
      else
      {
            // NO: slave mode is active
            // the button is disabled and does not handle any input
            clearButton->removeReleaseCallback(clearButtonCB);
      }
}
```

Source Sample 10 : Implementation of the checkPipMaster() method

When connecting fields of the pip sheet definition to geometry outside the pip sheet sub graph, the checkPipConnections() methods is used to set up these connection correctly. This is needed in the case the pip sheet geometry is copied for every user. The SoSprayingKit implementation does not need this method due to the structure of the pip sheet geometry, so we do not add it to the source code.

Next we add functionality to the application by supplying methods that add voxels to the scene. We add one method which adds a single voxel to mesh and another that does the spraying.

The **drawVoxel**() method (see Source Sample 11) adds a single voxel to the client volume of the application window at a specified position. The voxel design is stored in a template mesh structure, which is passed to the program through the loader file (described later in this section, see Source Sample 14). The template mesh in this example implementation consists of 8 vertices, representing a cube.

The method implements an atomic section in regards of network distribution, adds the new voxel to the mesh containing the set of already existing voxels and adjusts the voxel counter accordingly.

```
//
// voxel drawing method
//
// It adds a single voxel to the scene at a specified position
//

void
SoSprayingKit::drawVoxel(SbVec3f position, SbColor color, float size)
{
      int i, voxels;

      // start an atomic DIV section
      if(getDivObject()->getDiv())
            getDivObject()->getDiv()->atomicActionBegin();

      // increment voxel counter
      voxels=voxelField.getValue()+1;
      voxelField.setValue(voxels);

      // number of triangles of the template voxel
      int triNum =
((SoIndexedTriangleStripSet*)templateMesh.getValue())
            ->coordIndex.getNum();

      // number of already sprayed triangles.
      int triBase = (voxels-1)*triNum;

      // property node of the template voxel mesh
      SoVertexProperty* templateVP;
      templateVP=
      (SoVertexProperty*)((SoIndexedTriangleStripSet*)templateMesh.get
Value())->vertexProperty.getValue();

      // number of already sprayed vertices
      int vertexBase = (voxels-1)*8;

      // increment number of vertices in the property node
      vp->vertex.enableNotify(FALSE);
      vp->vertex.setNum(vertexBase+8);
      vp->vertex.enableNotify(TRUE);

      // add 8 vertices of a voxel to the mesh
      for(i=0; i<8; i++)
```

```
            ((SoVertexProperty*)mesh->vertexProperty.getValue())
                  ->vertex.set1Value(vertexBase+i,
            templateVP->vertex[i] * size + position);

      // set the color of the just added voxel
      int colBase = voxels;
      ((SoVertexProperty*)mesh->vertexProperty.getValue())
            ->orderedRGBA.set1Value(colBase-1,color.getPackedValue());

      // adjust the index counters of the mesh to include the new
voxel
      mesh->coordIndex.enableNotify(FALSE);
      mesh->materialIndex.enableNotify(FALSE);
      mesh->coordIndex.setNum(triBase+triNum);
      mesh->materialIndex.setNum(triBase+triNum);
      mesh->coordIndex.enableNotify(TRUE);
      mesh->materialIndex.enableNotify(TRUE);

      // construct triangles with the newly added vertices
      for(i=0; i<triNum; i++)
      {
            int idx =
      ((SoIndexedTriangleStripSet*)templateMesh.getValue())
                  ->coordIndex[i];

            mesh->coordIndex.set1Value(triBase+i, idx == -1 ? -1 :
            idx+vertexBase);
            mesh->materialIndex.set1Value(triBase+i, colBase-1);
      }

      // end the atomic section
      if(getDivObject()->getDiv())
            getDivObject()->getDiv()->atomicActionEnd();
}
```

Source Sample 11 : Voxel drawing function

The **spray**() method (see Source Sample 12) sets multiple voxels. There are two modes supported, which can be switched by a button on the applications pip sheet in this demo application. There is a mode for spraying and one mode for drawing voxels. When called in spraying mode, this method draws multiple voxels in a spherical region around the position supplied. In drawing mode, a voxel is drawn at the position supplied, if the new voxel does not overlap with a neighbor voxel.

```
//
// Spraying method
//
// adds several voxels to the scene
//

void
SoSprayingKit::spray(SbVec3f position, int uid)
{
      // get pointers to the pip sheet GUI elements
      SoNode* sheet = getPipSheet(uid);
      So3DCheckBox* modeButton;
      modeButton = (So3DCheckBox*)findNode(sheet,"SPRAY_MODE_BUTTON");

      SoMaterial* mat = (SoMaterial*)findNode(sheet,"SPRAY_COLOR");
      SbColor color = mat->diffuseColor[0];

      So3DSlider* sizeSlider;
      sizeSlider = (So3DSlider*)findNode(sheet,"SPRAY_SIZE_SLIDER");
      float size = sizeSlider->currentValue.getValue();

      // Spraying or Drawing mode enabled?

      if(modeButton->pressed.getValue() == SPRAYING_MODE)
      {
            // Spraying mode
            // A voxel will be sprayed somewhere in a spherical space
            // (the radius is set by a slider element on the Pip)
            So3DSlider* radiusSlider;
            radiusSlider =
                  (So3DSlider*)findNode(sheet,"SPRAY_RADIUS_SLIDER");
            SbVec3f randomVec(osRand()-0.5,osRand()-0.5,osRand()-0.5);
            position += randomVec*radiusSlider-
      >currentValue.getValue();
            drawVoxel(position,color,size);
      }
      else if(modeButton->pressed.getValue() == DRAWING_MODE)
      {
            // Drawing mode
            // A voxel at the cursor position if there not already
another
            // voxel in place.
            // (the last drawing position is stored in a translation
node
            // added to the Pip sheet)
            SoTranslation* lastDrawingPos;
            lastDrawingPos =
                  (SoTranslation*)findNode(sheet,"LAST_DRAWING_POS");
            SbVec3f last = lastDrawingPos->translation.getValue();
            float offset = (position-last).length();
            if(offset > size) // if far enough from last voxel
            {
                  drawVoxel(position,color,size);
                  lastDrawingPos->translation = position;
            }
      }
}
```

Source Sample 12 : Spraying function

5.1.4 Finish the application

To make the SoSprayingKit ready for action there have to be done three final steps: First we have to add the source code for handling the events that are passed to the window. When the button of the pen is pressed, we start spraying at the position of the pen. (see Source Sample 13)

```
//
// event callback function
//
// handles input events
//

SbBool
SoSprayingKit::windowEventCB(void* data, int message, SoWindowKit*
window, int uid, So3DEvent* event, SoPath* rootPath)
{
      if (message != WM_EVENT || uid == -1) return FALSE;

      // check if button or move event has occurred
      SbBool pressEv = (event->getType() ==
So3DEvent::ET_BUTTON0_EVENT)
                        && (event->getButton(So3DEvent::BUTTON0) ==
                  TRUE);
      SbBool moveEv  = (event->getType() == So3DEvent::ET_MOVE_EVENT)
                        && (event->getButton(So3DEvent::BUTTON0) ==
                  TRUE);

      // return if no button or move event was detected
      if(!pressEv && !moveEv) return FALSE;

      // calculate the position where the event has occurred
      SbVec3f position;
      window->orientation.getValue().inverse().multVec(
            event->getTranslation()-window->position.getValue(),
      position);

      // start the spraying
      spray(position, uid);
      return TRUE;
}
```

Source Sample 13 : The event callback function

The applications source code is now complete, so we can proceed to the second final step: The application must now be compiled to be a dynamic link library (on MS-Windows based systems) or a shared object (when running on UNIX based systems).

Finally we must assemble an appropriate .iv-file that can be used for loading and starting the application in the Studierstube system. The final spray.iv loader file is shown in Source Sample 14. Special care must be taken to supply the correct paths to the files that are referenced inside the loader file.
It is obvious that the definition of the pip sheet for SoSprayingKit takes up most of the entries in the loader file. We also use an entry in the pip sheet geometry to store the last drawing position of the user connected to the pip sheet.

```
#Inventor V2.1 ascii

#
# Spraying application loader file
#

DEF SP SoApplicationKit {
      fields [ SFBool readOnly,SFNode classLoader, SFNode contextKit,
             SFNode appGeom, SFNode info ]

      # Protect the loader file from being overwritten
      readOnly TRUE

      # load and initialize the application class
      classLoader     SoClassLoader {
            fields [ SFString className, SFString fileName ]
            className   "SoSprayingKit"
            fileName    "../apps/spray/spray_stb"
      }

      # the SoSprayingKit application node
      contextKit      DEF SPRAY SoSprayingKit {
      fields [ SFInt32 userID, SFNode templatePipSheet, SFBool
             clonePipSheet, SFNode templateMesh]
      userID 10

      #
      # Pip sheet geometry template
      #

      templatePipSheet  Separator {
            RotationXYZ {
                  axis  X
                  angle 1.57
            }
            SoTransform
            {
                  rotation 0 1 0 1.57
            }

      #
      # sliders
      # for red, blue, green color component selection
```

```
#
      DEF SPRAY_R_SLIDER So3DSlider {
            bodyColor 1 0 0
            startPoint      -20 0 0
            endPoint        20 0 0
            position        0.6
            currentValue    0.6
            radius      0.042
            width       0.15
            increment       0.1
            callback ON_RELEASE
            }
      Translation {
            translation     0 0 0.05
      }
      DEF SPRAY_G_SLIDER So3DSlider {
            bodyColor 0 1 0
            startPoint      -20 0 0
            endPoint        20 0 0
            position        0.6
            currentValue    0.6
            radius      0.042
            width       0.15
            increment       0.1
            callback ON_RELEASE
            }
      Translation {
            translation     0 0 0.05
      }
      DEF SPRAY_B_SLIDER So3DSlider {
            bodyColor 0 0 1
            startPoint      -20 0 0
            endPoint        20 0 0
            position        0.1
            currentValue    0.1
            radius      0.042
            width       0.15
            increment       0.1
            callback ON_RELEASE
            }
      Translation {
            translation     0 0 0.1
      }

      #
      # slider
      # for droplet size selection
      #

      DEF SPRAY_SIZE_SLIDER So3DSlider {
            startPoint      -20 0 0
            endPoint        20 0 0
            minValue    0.005
            maxValue    0.015
            position        0
            currentValue    0.01
            radius      0.042
            width       0.15
            increment       0.1
            callback ALWAYS
```

```
      }
Translation {
      translation     0 0 0.05
}

#
# slider
# selects spraying radius
#

DEF SPRAY_RADIUS_SWITCH Switch {
      whichChild 0
      DEF SPRAY_RADIUS_SLIDER So3DSlider {
            startPoint      -20 0 0
            endPoint        20 0 0
            minValue    0.03
            maxValue    0.06
            position        0.33
            currentValue    0.04
            radius      0.042
            width       0.15
            increment       0.1
            callback ALWAYS
      }
}
Translation {
      translation      -0.12 -0.008 -0.222
}

#
# pip sheet buttons
#

Separator {
      SoTransform {
            rotation 0 1 0 -1.57
      }

      #
      # mode selection button
      # selects paint/spray mode
      #

      DEF SPRAY_MODE_BUTTON So3DCheckBox {
            width  0.08 depth  0.035 height  0.02
            buttonColor 0.7 0.7 0
            textureSwitch Switch {
            whichChild 0 = USE SPRAY_MODE_BUTTON . pressed
            Group {
                  Translation { translation 0 0.01 0 }
                  Texture2 { filename
            "../apps/spray/iconpaint.gif" }
                  Cube { width 0.07 depth 0.03 height 0.001
            }
            }
            Group {
                  Texture2 { filename
            "../apps/spray/iconspray.gif" }
                  Cube { width 0.07 depth 0.03 height 0.001
            }
            }
```

```
            }
      }
      Translation {
            translation      0.1 0 0
      }

      #
      # clear button
      #

      DEF SPRAY_CLEAR_BUTTON So3DButton {
            width  0.08 depth  0.035 height  0.02
            buttonColor 0.7 0.7 0
            textureSwitch Switch {
            whichChild 0 = USE SPRAY_CLEAR_BUTTON . pressed
            Group {
                  Translation { translation 0 0.01 0 }
                  Texture2 { filename
            "../apps/spray/iconclear.gif" }
                  Cube { width 0.07 depth 0.03 height 0.001
            }
            }
            Group {
                  Texture2 { filename
            "../apps/spray/iconclear.gif" }
                  Cube { width 0.07 depth 0.03 height 0.001
            }
            }
            }
      }

      # additional slider icons
      Separator {
            Translation { translation 0.07 0 -0.03 }
            Texture2 { filename
      "../apps/spray/iconsize.gif" }
            Cube { width 0.04 depth 0.02 height 0.001 }
            Translation { translation 0.05 0 0 }
            Texture2 { filename
      "../apps/spray/iconradius.gif" }
            Cube { width 0.04 depth 0.02 height 0.001 }
      }
}

#
#  sphere size and spraying radius indicator
#

Translation {
      translation      0.12 0.008 0.223
}
Translation {
      translation     0 0 0.10
}
Material {
      diffuseColor  0.5 0.5 0.5
      transparency  0.5
}
Switch {
      whichChild 0 = USE SPRAY_RADIUS_SWITCH . whichChild
      Cylinder  {
```

```
                  height 0.01
                  radius 0.04 = USE SPRAY_RADIUS_SLIDER .
            currentValue
            }
      }
      Translation {
            translation 0 0.02 0
      }
      DEF SPRAY_COLOR Material {
                  diffuseColor    0.6 0.6 0.1 =
                  ComposeVec3f {
                        x       0.6 = USE SPRAY_R_SLIDER .
                  position
                        y       0.6 = USE SPRAY_G_SLIDER .
                  position
                        z       0.1 = USE SPRAY_B_SLIDER .
                  position
                  } . vector
                  transparency  0
      }
      Sphere  {
            radius 0.01 = USE SPRAY_SIZE_SLIDER . currentValue
      }
      Translation {
            translation     0 -0.05 -0.18
      }

      #
      # Translation node for saving the last drawing position
      #

      DEF LAST_DRAWING_POS Translation { translation 0 0 0 }
}
# pip sheet geometry template end

# enable pip sheet cloning
# every user get his own copy of the sheet
clonePipSheet       TRUE

#
# default window for the application
# containing an empty triangle strip set
#

windowGroup Group {
      SoWindowKit {
            size  0.5 0.5 0.5
            title "Spray"
            clientVolume Separator {
                  IndexedTriangleStripSet {
                        vertexProperty VertexProperty {
                              materialBinding PER_VERTEX_INDEXED
                        }
                  }
            }
      }
}

#
# template voxel
# (defining a cubic shape)
```

```
	#

	templateMesh IndexedTriangleStripSet {
		vertexProperty VertexProperty {
		vertex [
			-0.5  0.5  0.5,
			-0.5 -0.5  0.5,
			0.5  0.5  0.5,
			0.5 -0.5  0.5,
			0.5  0.5 -0.5,
			0.5 -0.5 -0.5,
			-0.5  0.5 -0.5,
			-0.5 -0.5 -0.5 ]
		}
		coordIndex
		[
			5, 3, 7, 1, -1,
			2, 4, 0, 6, -1,
			5, 4, 3, 2, -1,
			1, 0, 7, 6, -1,
			7, 6, 5, 4, -1,
			3, 2, 1, 0, -1,
		]
		}
	}

	# application icon to be displayed on the pip
	appGeom		Separator {
				Texture2 { filename
			"../apps/spray/spraycan.gif" }
			}
	# additional information may be stored here
	info		Info {
			}
}
```

Source Sample 14 : Spraying demo application loader file

5.2 Application Scripting

As application nodes are loaded using a correct loader .iv-file, it is also possible to implement an application by only composing an .iv-file. The application node in this file is SoContextKit. Using SoWindowKit node kits, SoButtonKit node kits and appropriate chaining of OIV fields, simple applications can be constructed.

Source Sample 15 shows a scripted application defined entirely in an .iv-file. The application presents a sphere in a window and a checkbox button that adjust the spheres color when activated or deactivated.

Being only a short example to show the principle of application scripting, it also shows what potential the integration of application loading and the Open Inventor scene graph has.

```
#Inventor V2.1 ascii

#
# "Red and Blue" Application scripting example
#
# A checkbox is displayed on the pip. According to the state of this
# checkbox a sphere displayed in the application window changes its
color.
#

DEF REDBLUE SoApplicationKit {
      fields [    SFNode classLoader, SFNode contextKit,
                  SFNode appGeom, SFNode info ]

      ContextKit  SoContextKit {
      fields [ SFInt32 appID, SFInt32 userID, SFNode templatePipSheet,
            SFBool clonePipSheet,  ]

      # the template pip sheet geometry supplies a checkbox widget
      TemplatePipSheet  Separator {
                              RotationXYZ {axis  X angle 1.57 }
                                    DEF CHECKBOX So3DCheckBox {
                                          width 5 depth 5 height 2
                                          buttonColor 1 0 0
                                          pressed 0
                                    }
                              }
      # a window containing a sphere
      # The key feature of this demo is using a "SoSelectOne" node to
      # select the spheres color. The color index is selected by the
      # state of the checkbox widget using an Open Inventor field
      # connection.
      windowGroup Group {  SoWindowKit {
                              size  0.3 0.3 0.3
                              title "(Default window)"
                              clientVolume Separator {
                                    Material {
                                    diffuseColor 1 0 0 =
                                    SoSelectOne {
                                          type SoMFColor
                                    # field connection to checkbox
widget
                                          index = USE CHECKBOX .
pressed
                                          input [1 0 0, 0 0 1]
                                          } . output
                                    }
                                    Sphere {
                                          radius      0.1
                                    }
                              }
                  }
```

```
      }
      clonePipSheet     FALSE
      }
      appGeom     Separator {
            Texture2 { filename "../apps/redandblue/redandblue.gif" }
                  }
      info        Info { }
}
```

Source Sample 15 : Application scripting example

5.3 Migrating an old application to the new scheme

Migrating a Studierstube application written for StbAPI 2.0 (a context application; see [21]) to our new application management and loading scheme, is a rather straight forward task. A successful migration involves the following steps with the features of the new application management in mind:

- Removing all the StbAPI 2.0 specific management paradigms. Our scheme uses Open Inventor scene graph structures for application management. There is no need to use other data structures than the ones supplied by Open Inventor.
- Removing static graphic data from the source code. Since the loader file contains all graphic data, there is no need to build up data, such as windows and pip sheets, inside the source code. All these definitions can be done in the loader file.
- Rename some of the methods of the old scheme application to the corresponding methods supplied by the new application management.
- Generate an appropriate loader file.

The following paragraphs describe the migration steps using the non scripted version of the “Red&Blue” application from the last chapter.

5.3.1 Changes of the class definition

We start by modifying the class definition of the application. In Source Sample 16 the original class definition is shown. The methods printed in bold letters are

specific to the old Studierstube API and are no longer used. These will be replaced by newly introduced methods that take over part of their functionality.

```
// "Red and Blue" Application
// old class definition according StbAPI 2.0

class SoRedAndBlueKit: public SoContextKit
{
      SO_KIT_HEADER(SoRedAndBlueKit);
public:
      static void initClass();
      SoRedAndBlueKit();
      SoContextKit* factory
      ~SoRedAndBlueKit();
      static void colorButtonCB(void* data, So3DButton* button);
private:

      // old StbAPI 2.0 method (now obsolete):
      SoWindowKit* createWindowGeometry(int index);
      SoNode* createPipSheet(int uid);
      virtual void setSheetMasterMode(SoNode* pipSheet, SbBool
masterMode);
};
```

Source Sample 16 : Old Red&Blue application class definition (StbAPI 2.0)

The *createWindowGeometry* method was originally used by the applications to build up windows and their contents for the Studierstube system. With our new application management scheme, graphical data including windows and application specific content, should be passed to the application using the loader file. Therefore this method is obsolete and replaced by the *checkWindowGeometry*. This method may perform any kind of verification of the window geometry that is defined in the loader file.

The *createPipSheet* method is also obsolete and replaced by the *checkPipGeometry* method. Similar to *checkWindowGeometry* it is used to verify the pip sheet geometry passed from the loader file.

The *setSheetMasterMode* method is replaced by a *checkPipMasterMode* method, which is evaluated when the mode of the pip changes between master and slave mode.

```
// Newly introduced methods replacing retired functions
private:
      SbBool checkWindowGeometry();
      SbBool checkPipGeometry();
      virtual void checkPipMasterMode(SoNode *pipSheet, SbBool
masterMode);
```

Source Sample 17 : New application class methods

Concluding this chapter, Source Sample 17 shows the new section of the class definition of our application.

5.3.2 Migrating the methods

Using the old Studierstube API it was necessary for the programmer to implement methods for creating window and pip sheet geometry. With our new approach, this geometry data is passed to the application by using the required loader file. To make sure that the information supplied by the loader file is correct for the applications needs, the *checkWindowGeometry* and *checkPipGeometry* methods of the SoContextKit class may be overwritten to test the geometry data of windows and pip sheet respectively.
In Source Sample 18 the old implemetation of the method that builds the window geometry is shown. This method is no longer supplied by the SoContextKit class, it is replaced by the *checkWindowGeometry* function to test the window geometry. Although the window geometry information now comes from the loader file, it is still possible to create the geometry structures inside the *checkWindowGeometry* method, e.g. when there is no window information in the loader file.

```
// old StbAPI 2.0 method (now obsolete):
//
// It creates a window including client geometry that is placed inside
// the window. (a sphere in this example)
//

SoWindowKit*
SoRedAndBlueKit::createWindowGeometry(int index)
{
      WindowCreate wc;
      SoWindowKit::defaultWindow(wc);
      SoWindowKit* windowKit = comm->createWindow(NULL, &wc, NULL,
NULL);
```

```
      windowKit->size = SbVec3f(0.3,0.3,0.3);
      SoSeparator* clientVolume = windowKit-
>getClientVolumeSeparator();
      clientVolume->addChild(new SoMaterial);
      SoSphere* sph = new SoSphere;
      sph->radius = 0.1;
      clientVolume->addChild(sph);
      return windowKit;
}
```

Source Sample 18 : Old window creation method (StbAPI 2.0)

To satisfy the needs of the new application management we completely remove the createWindowGeometry function show in Source Sample 18, and introduce the checkWindowGeometry method from Source Sample 19. The implementation shown here checks the existence of a window. If no window is defined in the loader file, the method creates one, containing a sphere.

```
// New window geometry checking method:
//
// This method allows checking of the application's window geometry
// which is supplied inside the associated loader file. Furthermore
// it can be used to create additional geometry.
// In this example a window is created, if no window geometry is found
// in the loader file.
//

SbBool
SoRedAndBlueKit::checkWindowGeometry()
{
      SoGroup *wGroup = (SoGroup*)windowGroup.getValue();
      SoWindowKit *windowKit;

      // Is a window supplied in the loader file?
      if (wGroup->getNumChildren() == 0)
      {
            // No window geometry found.
            // Create a new window
            windowKit = new SoWindowKit;
            windowKit->size.setValue(0.3, 0.3, 0.3);

            // add a sphere to the windows client volume
            SoSeparator* clientVolume = windowKit->getClientVolume();
            clientVolume->addChild(new SoMaterial);
            SoSphere* sph = new SoSphere;
            sph->radius = 0.1;
            clientVolume->addChild(sph);

            // add the window to the application
            wGroup->addChild(windowKit);
      }
      return TRUE;
}
```

Source Sample 19 : New window geometry checking method

Similar to the window geometry, pip sheet geometry is no longer built by the application, but tested to ensure its correctness. In Source Sample 20 the old API implementation is shown where the sheet geometry is read from a file and some additional initializations are handled.

```
// old StbAPI 2.0 method (now obsolete):
//
// This method creates a new pip sheet geometry. In this example
// the sheet geometry is loaded from a file.
//

SoNode*
SoRedAndBlueKit::createPipSheet(int uid)
{
      char buffer[100];
      comm->setContextAliasName(getName(),"R&B");

      // load sheet from file
      SoSeparator *newPipSheet =
            readFile("sheet.iv",comm->workingDirectory);
      newPipSheet->ref();
      So3DButton* bRed  =
(So3DButton*)findNode(newPipSheet,"RED_BUTTON");
      sprintf(buffer,"RED_BUTTON_%d_%s", uid,getName().getString());
      bRed->setName(buffer);
      So3DButton* bBlue =
(So3DButton*)findNode(newPipSheet,"BLUE_BUTTON");
      sprintf(buffer,"BLUE_BUTTON_%d_%s", uid,getName().getString());
      bBlue->setName(buffer);
      newPipSheet->unrefNoDelete();
      return newPipSheet;
}
```

Source Sample 20 : Old pip sheet geometry creation method (StbAPI 2.0)

The *createPipSheet* function is also removed from the application and a new *checkPipSheet* function is introduced. As for the checkWindowGeometry function, there are also many implementations possible to do a check on pip sheet geometry. One possible check would be to search for special buttons needed by the application and ensure their existence.

For sake of simplicity we assume that the pip sheet definition of this migration demo is correct and implement this function only in its minimal form here. (see Source Sample 21)

```
// New pip sheet geometry checking method
//
// Pip sheet geometry supplied in the loader file may be checked
// in this method. Similar to the checkWindowGeometry() method it
// is possible to create additional geometry in here.
// In this example it is assumed the geometry is correct and no
// further testing is performed.
//

SbBool
SoRedAndBlueKit::checkPipGeometry()
{
      return TRUE;
}
```

Source Sample 21 : New pip sheet geometry check function

The last method to be migrated to fit the new application management structure is the *setSheetMasterMode* structure, and it is also the easiest migration since only the name of the method is changed to *checkPipMasterMode* (see Source Sample 22). When the mode of the pip changes between master and slave mode, the method sets or releases the button callback functions in this implementation. So only the master pip can interact with the application.

```
// New master mode checking method
//
// This method is called every time the mode of the application
// changes (master/slave). It is used to set and release the
// callback function of the pip sheet widget.
//

void
SoRedAndBlueKit::checkPipMasterMode(SoNode* pipSheet, SbBool
masterMode)
{
      So3DButton* bRed  =
(So3DButton*)findNode(pipSheet,"RED_BUTTON");
      So3DButton* bBlue =
(So3DButton*)findNode(pipSheet,"BLUE_BUTTON");

      if(masterMode)
      {
            bRed->addReleaseCallback(colorButtonCB,this);
            bBlue->addReleaseCallback(colorButtonCB,this);
      }
      else
      {
            bRed->removeReleaseCallback(colorButtonCB);
            bBlue->removeReleaseCallback(colorButtonCB);
      }
}
```

Source Sample 22 : New master mode checking method

5.3.3 Methods unchanged during migration

The methods containing the actual functionality of the application are not changed during migration. Also the Open Inventor specific methods and macros are kept in place.
Source Sample 23 shows the callback function usually attached to the buttons on the pip. It implements changing the color of the window contents to match the color of the button when pressed by the user.

```
// pip sheet button callback function
// changes the color of the sphere inside the client volume
// to the color of the button
//

void
SoRedAndBlueKit::colorButtonCB(void* data, So3DButton* button)
{
      SoRedAndBlueKit* self = (SoRedAndBlueKit*)data;
      SoSeparator* winRoot = self->getWindow(0)->getClientVolume();
      SoMaterial* mat = (SoMaterial*)winRoot->getChild(0);
      mat->diffuseColor.setValue(button->buttonColor.getValue());
}
```

Source Sample 23 : Pip sheet button callback function

The Open Inventor style initClass() function and class constructor implementations also remain unchanged. Their implementation uses some Open Inventor macros to make the application a node kit. (refer to [2]). Furthermore the factory function acting as a virtual constructor is not changed during the migration process.
All these functions are shown in Source Sample 24.

```
// static Open Inventor class initialization function

void
SoRedAndBlueKit::initClass(void)
{
      SO_KIT_INIT_CLASS(SoRedAndBlueKit, SoContextKit,
"SoContextKit");
}
```

```
// Constructor

SoRedAndBlueKit::SoRedAndBlueKit()
{
      SO_KIT_CONSTRUCTOR(SoRedAndBlueKit);
      SO_KIT_INIT_INSTANCE();
}


// Virtual Constructor

SoContextKit*
SoRedAndBlueKit::factory()
{
      return new SoRedAndBlueKit;
}
```

Source Sample 24 : Unchanged application node kit class methods

5.3.4 Implementation changes

The new application management takes full advantage of Open Inventor's scene graph database functionality. Therefore the applications are implemented as fully featured Open Inventor node kits and managed in a scene graph struture.

As shown in Source Sample 25 the old Studierstube's application implementation had to call a destructor() function from inside the class destructor. This call is obsolete in our new system. An application node is simply deleted by using Open Inventor's reference counting mechanism.

```
// old StbAPI 2.0 destructor method:
//
// the call of the destructor() function is now obsolete
//

SoRedAndBlueKit::~SoRedAndBlueKit()
{
      destructor();
}
```

Source Sample 25 : Old destructor implementation (StbAPI 2.0)

The old implementation also required the lines shown in Source Sample 26 to be used in the source code. The include file and the macro are no longer used anywhere in our implementation.

```
#include <stbapi/workspace/SoContextKitSub.h>

CONTEXT_APPLICATION(SoRedAndBlueKit);
```

Source Sample 26 : Obsolete include file and macro usage (StbAPI 2.0)

5.3.5 Creating the loader file

The last migration step is the creation of an appropriate loader file for our application (see also chapter 5.1.4). The loader file shown in Source Sample 27 supplies a window with a sphere as its content and a pip sheet with a red and a blue button to the application.

```
#Inventor V2.1 ascii

# Red&Blue Application
# loader file
DEF RB SoApplicationKit {
      fields [ SFBool readOnly,SFNode classLoader, SFNode contextKit,
             SFNode appGeom, SFNode info ]
      readOnly TRUE

      # load the application class from a dll
      classLoader SoClassLoader {
            fields [ SFString className, SFString fileName ]
            className "SoRedAndBlueKit"
            filename  "../apps/redandblue/redandblue_stb"
      }

      contextKit  DEF REDANDBLUE SoRedAndBlueKit {
      fields [ SFInt32 userID, SFNode windowsGroup, SFNode
             templatePipSheet, SFBool clonePipSheet ]
      userID 10

      # Window geometry with a sphere inside the client volume
      windowGroup Group {
            SoWindowKit {
            Size  0.5 0.5 0.5
            title "Red&Blue"
            clientVolume Separator {
                  Material { }
                  Sphere {
                        radius 0.1
                  }
            }
      }
      }

      # pip sheet geometry template
      # supplies a red and a blue button on the pip
      templatePipSheet  Separator {
```

```
                RotationXYZ {axis  X angle 1.57 }
                DEF RED_BUTTON So3DButton {
                      width 5 depth 5 height 2
                      buttonColor 1 0 0
                }
                Translation { translation 10 0 0 }
                DEF BLUE_BUTTON So3DButton {
                      width 5 depth 5 height 2
                      buttonColor 0 0 1
                }
      }

      # disable pip sheet cloning
      clonePipSheet FALSE
      }

      # application icon
      appGeom Separator {
                Texture2 { filename
                "../apps/redandblue/redandblue.gif" }
      }
      info  Info {
      }
}
```

Source Sample 27 : “Red and Blue” application loader file

6 Conclusions

6.1 Summary

We presented a new application management approach for the Augmented Reality system of Studierstube. Our goal was to tightly integrate the application management with the basic building block of Studierstube's software architecture, the Open Inventor toolkit. This approach enabled us to get the full benefit of the toolkit's database and scripting functionality, and also reduce the amount of programming effort needed to write new Studierstube applications in the future.

The Open Inventor toolkit library supplies the concept of a scene graph for managing its database. We made use of this concept by implementing the applications for Studierstube as scene graph nodes and storing them inside the scene database. So we automatically benefit from scene graph scripting provided by the toolkit, which makes applications more flexible to configure and adds support for rapid prototyping to the system.

The software architecture of Studierstube has experienced an extensive process of Refactoring during the development of our application management. By simplifying the structure of the code, the whole system is now easier the be overviewed and maintained. We took special care of clearing the code of many complicated structures, which were a result of Studierstube's long time development.

We supplied the basic framework for building applications for an Augmented Reality system. Now ideas are needed to implement such applications and take advantage of the possibilities offered by virtual environments. Introducing good applications to the system, will be the key to success. Ongoing enhancements made by further development will surely be founded by the ideas born from the experience made by developing and working with these applications.

6.2 Future work

Since Studierstube is under permanent ongoing development – as is the field of Augmented Reality - many future projects can be taken into account for further improving the system.

The first important step will be the porting of existing Studierstube applications to work with the new system.

One possible future project that relies directly on this work, could be the development of a Studierstube server that is used to save the state of all applications. Clients could connect to this server over the internet and participate in the collaborative environment, similar to multi user computer games. When the user disconnects, the state of all open applications is saved to the server for later use.

7 References

[1] J. Wernecke: The Inventor Mentor: Programming Object-Oriented 3D Graphics with Open Inventor, Release 2, Addision-Wesley 1994.

[2] J. Wernecke: The Inventor Toolmaker: Extending Open Inventor, Release 2, Addison-Wesley 1994.

[3] G. Hesina, D. Schmalstieg, A. Fuhrmann, W. Purgathofer, Distributed Open Inventor: A Practical Approach to Distributed 3D Graphics, Proceedings of ACM Virtual Reality Software & Technology '99 (VRST'99), pp. 74-81, London, December 20-22,. 1999.

[4] Zs. Szalavari, D. Schmalstieg, A. Fuhrmann, M. Gervautz: „Studierstube“ An environment for collaboration in Augmented Reality, Virtual Reality-Systems, Development and Applications, 3(1), 37-49.

[5] Zs. Szalavari, M. Gervautz: The Personal Interaction Panel - A Two-handed Interface for Augmented Reality, Proceedings of Eurographics’97, volume 16-3, pp. 336-346, 1997.

[6] E. Gamma, R. Helm, R. Johnson: Design Patterns. Elements of Reuseable Object-Oriented Software, Addison-Wesley 1997.

[7] M. Fowler, K. Beck, J. Brant, W. Opdyke: Refactoring- Improving the design of existing code. Addison-Wesley 1999.

[8] R. Azuma, Y. Baillot, R. Behringer: Recent Advances in Augmented Reality. IEEE Computer Graphics and Applications, November 2001.

[9] R. Azuma: A Survey of Augmented Reality. Presence: Teleoperators and Virtual Environment 6, 4 (August 1997), 355-385.

[10] P. Strauss, R. Carey: An Object-Oriented 3D Graphics Toolkit, in Computer Graphics (Proc. ACM SIGGRAPH ’92), 341-349, August 1992.

[11] P. Milgram, F. Kishino: A Taxonomy of Mixed Reality Visual Displays, IEICE Trans. Information Systems. Vol. E77-D, no. 12, 1994, 1321-1329.

[12] K. Kiyokawa, Y. Kurata, and H.Ohno: An Optical See-Through Display for Mutual Occlusion of Real and Virtual Environments, Proc. Int’l Symp. Augmented Reality 2000 (ISAR ’00), October 2000, pp. 60-67.

[13] T. Sugihara, T. Miyasato: A lightweight 3-D HMD with Accomodative Compensation, Proc. 29th Soc. Information Display (SID '98), May 1998, pp. 927-930.

[14] M. Spitzer et al.: Eyeglass-based Systems for Wearable Computing, Proc 1st Int'l Symp. Wearable Computers (ISWC '97). October 1997, pp.48-51.

[15] I. Kasai et al.: A forgettable near eye display, Proc 4th Int'l Symp. Wearable Computers (ISWC 2000), October 2000, pp.115-118.

[16] H.L. Pryor, T.A. Furness, E. Viirre: The virtual Retinal Display: A New Display Technology Using Scanned Laser Light, Proc 42nd Human Factors Ergonomics Society, October 1998, pp. 1570-1574.

[17] R. Raskar et al.: Multi-Projector Displays Using Camera-Based Registration, Proc. IEEE Visualization '99, October 1998, pp. 161-168.

[18] R. Raskar, G. Welch, W.-C. Chen: Table-top spatially-augmented reality: Bringing physical models to life with projected imagery, Proc. 2nd Int'l Workshop Augmented Reality (IWAR '99), October 1999, pp. 64-71.

[19] R. Pausch, T. Crea, M. Conway: A Literature Survey for Virtual Environments: Military Flight Simulator Visual Systems and Simulator Sickness, Presence: Teleoperators and Virtual Environments 1, 3, Summer 1992, pp. 344-363.

[20] J. Rolland, L. Davis, Y. Baillot: A Survey of Tracking Technologies for Virtual Environments, in "Fundamentals of Wearable Computers and Augmented Reality", W. Barfield, T. Caudell, eds., Lawrence Erlbaum, Mahwah, NJ, 2001, pp. 67-112.

[21] Old Studierstube API (StbAPI Release 1.99), http://www.Studierstube.org/doc/ old/stbapi/

[22] Java Documentation: "Java Object Serialization", http://java.sun.com/j2se/1.4/docs/guide/serialization/

[23] A. Downs: Java Serialization – Adding object persistence to Java applications, MacTech Magazine Vol. 14, Issue 4, 1998, http://www.mactech.com/articles/mactech/Vol.14/14.04/JavaSerialization/

[24] D. Schmalstieg, G. Reitmayr, G. Hesina: Distributed Applications for Collaborative Three-Dimensional Workspaces, PRESENCE -

Teleoperators and Virtual Environments, Vol. 12, No. 1, pp. 52-67, MIT Press, 2003.

[25] M. Kalkusch, T. Lidy, M. Knapp, G. Reitmayr, H. Kaufmann, D. Schmalstieg: Structured Visual Markers for Indoor Pathfinding, Proceedings of the IEEE First International Workshop on ARToolKit, 2002..

[26] H. Kaufmann, D. Schmalstieg: Mathematics and Geometry education with collaborative Augmented Reality, Computers & Graphics, Vol. 27, No. 3, pp. 339-345, 2003