Studierstube Builder

Content Creation for Augmented Reality –
The *Studierstube Builder*, a Rapid Prototyping Tool

Felix Nairz

Studierstube Builder

Content Creation for Augmented Reality –
The *Studierstube Builder*, a Rapid Prototyping Tool

Master's Thesis

at

Graz University of Technology

submitted by

Felix Nairz

Institute for Computer Graphics and Vision (ICG),
Graz University of Technology
A-8010 Graz, Austria

27th October 2009

Advisor: Univ.-Prof. Dr. Dieter Schmalstieg
Co-Advisor: Dipl.-Ing. Bernhard Kainz

Studierstube Builder

Ein interaktives Tool zur raschen Erstellung von Augmented Reality Anwendungen

Diplomarbeit

an der

Technischen Universität Graz

vorgelegt von

Felix Nairz

Institut für Computergrafik und Vision (ICG),
Technische Universität Graz
A-8010 Graz

27. Oktober 2009

Diese Arbeit ist in englischer Sprache verfasst.

Begutachter:	Univ.-Prof. Dr. Dieter Schmalstieg
Mitbetreuender Assistent:	Dipl.-Ing. Bernhard Kainz

Abstract

The motivation for Studierstube Builder was the belief that augmented reality will make a breakthrough in the next couple of years. With powerful, yet mobile devices on the market and more coming out, many users will finally own suitable devices. Studierstube and the mobile phone port Studierstube ES are two powerful augmented reality frameworks developed at Graz University of Technology. Most applications that make use of computer generated graphics, such as augmented reality, use the well established concept of scene graphs to alleviate the rendering process. Creating scene graphs still proves to be a time-consuming and error-prone task as there are too few tools on the market that assist a developer. We see the lack of adequate tools as a factor that handicaps the fast creation of augmented reality applications. Therefore we started developing Studierstube Builder, an Open Inventor compatible scene graph editor which allows the rapid creation of scene graphs through a graphical user interface. The goal during the design process was to create a tool that supplies advanced users with functionality they need while keeping the user interface simple and clear enough to reduce the training period of inexperienced users. The interface is leaned on some of the best rapid application development tools on the market. A preview of the currently edited scene graph is shown all the time and instantly reflects all changes made, thus creating an instant feedback loop for the developer. Studierstube Builder, as the name suggests, has a built in support for Studierstube and thus makes a productive combination for the creation of augmented reality content. The integration can however be turned of as well. Then Studierstube Builder becomes a normal scene graph editor. Our scene graph editor is build on top of state of the art technology and is engineered around design patterns to allow flexible expansion of functionality.

Kurzfassung

Ausgangspunkt für die diese Arbeit war die Überzeugung, dass Augmented Reality (AR) Anwendungen innerhalb der nächsten Jahre ihren Durchbruch am Massenmarkt schaffen werden. Schon jetzt gibt es eine große Vielfalt an geeigneten Endgeräten, die sich in den nächsten Jahren aller Voraussicht nach noch weiter vergrößern wird. So wird es erstmals dazu kommen, dass eine größere Menge an Menschen über für Augmented Reality Anwendungen geeignete Endgeräte verfügen. An der Technischen Universität Graz werden aktuell zwei Entwicklungs-Plattformen für solche Applikationen entwickelt - Studierstube und Studierstube ES, welches speziell für mobile Endgeräte optimiert ist. Nahezu alle Anwendungen, welche computergenerierte Grafik verwenden - dazu zählen Augmented Reality-Anwendungen - basieren auf sogenannten Szenengraphen. Ein Szenengraph ist ein etabliertes Konzept um die Erstellung von interaktiven, grafischen Applikationen zu erleichtern. Die Erstellung solcher Szenengraphen ist jedoch nach wie vor ein relativ zeitaufwändiger und fehleranfälliger Prozess, weil es wenige Programme gibt, die den Entwickler bei der Arbeit unterstützen. Unserer Ansicht nach behindert dies die rasche Entwicklung von neuen Augmented Reality-Anwendungen. Dies war der Ausgangspunkt für die Entwicklung von Studierstube Builder, einem grafischen Szenengraphen-Editor. Studierstube Builder baut auf einer Open Inventor-kompatiblen Bibliothek auf und ermöglicht die schnelle Erzeugung von neuen sowie eine vereinfachte Modifizierung von vorhandenen Szenengraphen. Entwicklungsziel war die Erstellung eines Werkzeuges, das sowohl professionelle Entwickler als auch Anfänger bei ihrer Arbeit unterstützt. Um dies zu erreichen, ist es notwendig, erfahrenen Benutzern die von ihnen benötigte Funktionalität zur Verfügung zu stellen und gleichzeitig auf eine einfach zu bedienende Benutzeroberfläche zu achten, sodass sich auch Anfänger rasch zurechtfinden. Wie der Name schon andeutet, ermöglicht Studierstube Builder auch die Erstellung von Szenengraphen für das Studierstube-Projekt. Dadurch entsteht eine neue Entwicklungsplattform, die eine rasche Erstellung von Augmented Reality-Applikationen mit dem vollen Funktionsumfang von Studierstube verbindet. Die Implementierung von Studierstube Builder baut auf zeitgemäßen und etablierten Werkzeuge auf. Die Verwendung von bekannten Entwurfsmustern gewährleistet eine gute Erweiterbarkeit.

Statutory Declaration

I declare that I have authored this thesis independently, that I have not used other than the declared sources / resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

Place	Date	Signature

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommene Stellen als solche kenntlich gemacht habe.

Ort	Datum	Unterschrift

Contents

List of Figures

Listings

Acknowledgements

Sincere thanks to my parents Christa and Matias for raising me and giving me the opportunity to study. You never spared any costs or effort to support me and your trust and encouragement helped me survive the hard times during my course of studies. Cordial thanks also goes to my stepparents Christine and Reiner. You always treated me like your own child.

If there is one person I truly love, it is you Annina. You keep surprising me and there is no other person as warm-hearted as you. Thank you for all your support, encouragement, and motivation throughout the last six years. I am looking forward to our wedding :).

Special thanks go to my supervisor, Dieter Schmalstieg and my co-advisor, Bernhard Kainz who has been my mentor for a little more than a year. Thank you, Dieter, for your interesting lectures which arouse my interest in the subject of computer graphics. I enjoyed the relaxed atmosphere at the institute although you had to push me to use your first name. Bernhard, thanks to you for coming up with the initial idea for the project and the help in getting funding. Your continuous motivation and your incredible working speed is second to no one. You encouraged me to accept the challenge of such a large project and put more trust in me than I had. I am grateful for that.

I would also like to acknowledge the help and inspiration I received from Gerhard Reitmayr. You pointed me in new directions. Not to forget Keith Andrews for supplying me with literature, new inspirations, and for his thesis template. It saved me from days of suffering.

I am also indebted to my two closest study colleagues Markus Muchitsch and Martin Kandlhofer. We had an amazing time together. With you, no night was too long and no hour to early to push hard forward. You became true friends to me.

Finally, I would like to thank Aaron and Thomas for the last seven years we shared our flat. You are like a family to me and no matter how far we may spread and how forgetful I may become I will always remember the good times we had. I am grateful for these experiences.

Felix Nairz
Graz, Austria, October 2009

Credits

- This work was created using the LaTex template by Keith Andrews [Andrews, 2009].
- Thanks to Jose Miguel Espadero, Rubén López, Robert Rothfarb, and O'Reilly Media for authorization to use their images within this work.

Copyright Notice

Chapter 1

Introduction

1.1 Scene Graph Overview

Scene graphs are a widely accepted concept used to draw three-dimensional graphics scenes on a computer, but usage extends beyond these borders. A scene graph library is built on top of a low level graphics library like OpenGL and abstracts large parts of the low level commands involved in drawing graphics on a computer, thus accelerating the development of graphics applications. Usage of scene graphs spans a large number of applications. They can be used for visualization of large datasets in different research areas. Vector graphic drawing tools can make use of them [Kersting and Döllner, 2002]. Scene graphs allow the effective composition of three-dimensional scenes in the field of computer graphics. This includes computer games, virtual and augmented reality, and other applications making use of real-time graphics. By using a scene graph library, the development of the application becomes much easier compared to the use of low-level graphic libraries such as OpenGL. In addition a clear separation between application logic and visualization is drawn. As interaction facilities are usually included in scene graphs, they can not only be used to display graphic scenes but also to design graphical 2D and 3D user interfaces and implement proper reactions to user input. Other application areas outside of the classical computer graphics field have recently been unlocked. The MPEG-4 video compression standard for example uses VRML [The Web3D Consortium, 1997], a standardized scene graph file format for multimedia scene composition [Pereira and Ebrahimi, 2002; Walsh, 2002]. Finally, efforts have been made to apply the structured scene graph concept for image processing pipelines using well known image processing libraries.

Scene graphs generally store their information in a tree or a graph structure with a well defined starting point, the so-called root. Each node in the scene graph is either used to manage other (child) nodes or represents an object which modifies some properties of the scene. An image is created step by step, by visiting each node in the scene graph. Depending on the nature of the node, it can draw geometry or modify properties, like for example the current drawing color for later nodes. This visiting of nodes is called traversal of the scene graph. To achive consistent rendering result between successive frames, the order in which the nodes are visited is fixed.

Since the ”invention” of modern scene graphs by [Strauss and Carey, 1992] many different implementations appeared. Most of them however, share the same principles and differ only in their specialization and some performance related design decisions. Some scene graph libraries allow for example the usage of multiple processors or multiple threads to accelerate their rendering. Display lists can be used to cache the drawing of a certain sub-graph and thus largely accelerate rendering. Scene graphs have been written in many different languages, the most common ones being C++ and Java. Other languages are supported through so called language bindings, for example JavaScript, Python, and .NET. As most of the scene graph libraries are built on top of OpenGL, they can usually run on different operating systems as well. This makes is particulary easy for the user to create platform independent applications.

1.2 Augmented Reality

Augmented reality is a technique where the view of the world is superimposed by computer generated content. In contrast to virtual reality, where the user is completely captured in a virtual environment, augmented reality allows the user to see the real world around. Additional virtual objects are added to the users view of the world to enhance perception and display for example additional information. This section summarizes work done by [Reitmayr and Schmalstieg, 2001; Barczok et al., 2009; Azuma, 1997] to give an overview over that field of expertise.

[Azuma, 1997] defines the following characteristics of augmented reality systems:

- Combines real and virtual
- Interactive in real time
- Registered in 3-D

A movie with some computer generated special effects for example, does not fall into this category as photorealistic rendering cannot be done in real time yet [Azuma, 1997]. To allow the display of additional information, the device used first needs to sense the environment and add information to the user's perception later. While augmented reality could use multiple human senses, the most common approach is to add only visual information. This can for example be done by wearing a see-through head mounted display. The user can see through them like through glasses but additional computer generated information can be displayed as well. Another alternative which is quickly evolving is to make use of smartphones. The environment is then recorded through a builtin camera. Additional information is rendered, before the combined image is shown on the display. The user can thus use a smartphone as an augmented reality device.

Augmented reality becomes really powerful when information about the current location and the task performed by the user is available. In a laboratory environments, the location can be computed through the use of visual markers or a tracking system. In the real world GPS, coordinates, a compass, and a intertial sensor can be used to get information about the current position, although, with limited accuracy. To get a more accurate position estimation, image processing techniques can be applied to compare the current camera image with a set of registered images. However, as this depends on multiple factors like illumination conditions, quality of the built in camera, processing power of the hardware and algorithms used results may not be robust enough for wide usage. Local setups which require only the relative position to some reference point do not have to overcome this obstacles and seem to be more promising at the moment.

Although hardware performance is evolving rapidly, none of the smart phones delivers the processing power for complex augmented reality applications yet. However, recent development will likely push this border a step further. Nvidia's new Tegra system-on-a-chip for example offers OpenGL support and programmable shaders at the size of a dime while keeping power consumption below one watt [NVIDIA Corporation, 2009c]. For mobile desktop performance, one will still need to use alternative hardware like ultra-mobile PCs or even notebooks, usually in conjunction with a head mounted display.

Applications for augmented reality span a large area. Commercial TV channels for example adapt perimeter advertising of international football games in each country they broadcast. In hockey the puck is highlighted to improve cognition. Tourist could be guided through museums or other attractions, displaying relevant information where requested. Games are another growing field. Examples include "ARhrrrr!" (see [Spreen et al., 2009]), a game where the user has to shoot zombies and "The Eye of Judgement" (see [Sony Computer Entertainment Inc., 2007]), a commercial card game for Sony's Playstation 3. Interestingly, ARhrrrr! uses a prototype of the aforementioned Nvidia Tegra chip. A video showing the graphical options can be found on their website. More serious industry applications also exist. The Vidente project for example displays underground networks as extra information for field

workers to avoid damage during construction processes. See section 5.1 for more details about Vidente. Doctors could benefit from augmented reality in many ways. Data acquired through magnetic resonance imaging or computer tomography could for example be shown, so that physicians have x-ray vision inside a patients body. This can make operations less invasive or help during operation planning. Medical education could also strongly benefit from these techniques, as more training could be performed on virtual patients. Certainly, tracking has to be very accurate and reliable for medical applications. This list of examples is of course not exhaustive, many other areas can benefit from additionally displayed information. Most of the augmented reality applications make use of scene graphs as their basic graphics layer for visualization.

1.3 Studierstube as Augmented Reality Platform

Studierstube is an augmented reality framework whose creation started back in 1996 at Vienna University of Technology. The project moved to Graz University of Technology in 2005 and is available as version 4.4. Studierstube's name comes from the study room where Goethe's character Faust tries to gather knowledge. Studierstube is a project which collects and enhances many different ideas known from other projects or applications. The following paragraphs are a summary from [Schmalstieg et al., 2002].

While other augmented reality projects develop specific and specialized user interfaces for each application, Studierstube tries to solve the composition of three-dimensional user interfaces in a general way so that different applications can make use of it. The goal is to transfer the desktop metaphor into the third dimension. Support for multiple application, multi document interfaces (MDI) is standard in the 2D desktop world. Studierstube transfers this approach into the third dimension. Instead of a 2D window, applications are drawn in a 3D window with a bounding box indicating the dimensions. An application is not allowed to draw objects outside its windows boundaries. Windows can be minimized and maximized as normal desktop windows. The content of each window is stored as a separate scene graph. Multiple instances of the same application can be started, implementing the MDI for three-dimensional worlds.

In today's information society solving problems together is essential. Studierstube tackles this requirement by supplying a highly collaborative environment which supports co-workers to work together both face to face and from remote locations. Multiple users can work on one host or on many different hosts. Changes made by one user are automatically synchronized to the other hosts. Concurrent and conflicting changes are avoided by employing a master/slave principle where only changes to a master object are distributed to all slave objects. Slave objects do not react to user input, but rely on changes being delivered from the master object instead. Although master/slave roles can be changed during runtime, each application has exactly one master and an arbitrary amount of slaves at a time. In addition to the shared parts of a scene, each user can have a private space (an own note pad for example) which is invisible to the others. To facilitate collaboration and to enable a private space for each user when working on the same location, hardware support is necessary. Usually this is solved through the use of suitable display devices. This can for example be semi-transparent head mounted displays or a back-projection wall in combination with shutter glasses. Each user's position is tracked and the virtual scene is rendered separately for every user from their respective viewpoint.

When working on multiple locations or at least with multiple displays, another interesting property of Studierstube comes into play. Application windows can, but do not have to be shared between multiple displays allowing collaboration with the same set of applications running on each location. Alternatively each display can be configured individually to work on different projects or applications. In this case only applications which are shared on multiple displays are synchronized between hosts. This allows multiple users to share the same immersive experience while collaboration is possible.

To allow the usage of different display devices, a viewer supporting a number of display modes (such as field sequential or line interleaved stereo) is included in Studierstube. Tracking is solved by a separate, open-source package called Open Tracker. This way new tracking hardware can be integrated

by extending the existent architecture with new nodes.

Studierstube's software framework consists of a number of scene graph nodes like display or tracking nodes and a runtime environment that executes the Studierstube environment. Applications are handled with dynamic libraries that can be loaded during runtime. Beside the Studierstube runtime environment, all objects including applications, windows and tracking are scene graph nodes and are synchronized between different hosts using the built in synchronization mechanism.

Studierstube has also been used for mobile augmented reality applications [Reitmayr and Schmalstieg, 2001]. The system is designed around a mobile computer, a see-through head mounted display, a mobile tracking system and a camera mounted on the helmet. This way the stationary Studierstube setup is expanded by a fully mobile setup. Applications reside around the user with a body stabilized coordinate system. A user can interact with these items through a tracked pen making use of all Studierstube features described before. Meetings with users of the same mobile setup or a stationary setup can be held as well making use of Studierstube's built in synchronization features, allowing even adding or removing of users during collaboration.

Recently Studierstube has been extended to mobile phones [Schmalstieg and Wagner, 2007]. The resulting software framework, Studierstube ES, is targeted on handheld devices. The software framework runs entirely on handheld devices. To enable multi-user applications, a PC-based server application exists. Handheld devices communicate with their server through a wireless network connection. The software is specifically designed for the limited resources a handheld device offers. Tracking is for example done through a complete re-implemented, optimized tracking engine. The scene graph library used was custom written as well, borrowing many ideas from established scene graph libraries, but minimizing the functionality to the minimum necessary to perform rendering.

1.4 Motivation

Scene graphs are widely used for a variety of applications including the entire field of augmented reality discussed before. Scene graph libraries have been around for almost two decades, and the involved concepts are well understood and studied. Surprisingly however, few tools exist which support users when working with scene graphs. A scene graph tool should allow the easy and fast creation of new scene graphs or the modification of existing scene graphs through a graphical user interface. Without such tools the user has to programm, or use one of the textual file formats to define the scene graph. All of the existing tools are specifically implemented for one specific scene graph library. Thus different tools are developed for different scene graph libraries, further limiting the number of tools available for one specific scene graph library. Most existing tools are developed from companies or people other than those developing the scene graph libraries. Many tools come from within the user-community and are programmed by a small number of people. Progress depends on the number and motivation of the people involved. Therefore development for some of the tools has stopped years ago. Tools which are not under active development quickly fall behind or become even unemployable.

However, we firmly believe in the importance of great tools during the development process. Building a tool on top of a scene graph library adds another layer of abstraction for the user. This can largely accelerate the creation of scene graphs. There are some considerations in favour of it: First, writing source code is time consuming. The user requires programming knowledge, has to know the programming interface of the used libraries, and needs a programming environment. Second, most users prefer a graphical user interface when it offers approximately the same set of functionality as a command line-based application. Examples are file managers and email clients. Third, the human brain is very well suited to rapidly gather visual information while it is rather slow at gathering pure text [Tidwell, 2005, Chapter 6]. Graph or tree structures used for scene graphs are very suitable to be shown visually. Additionally, a scene graph tool has higher level knowledge than the scene graph library and can for perform additional exception management or offer new features not present in the scene graph library. Moreover,

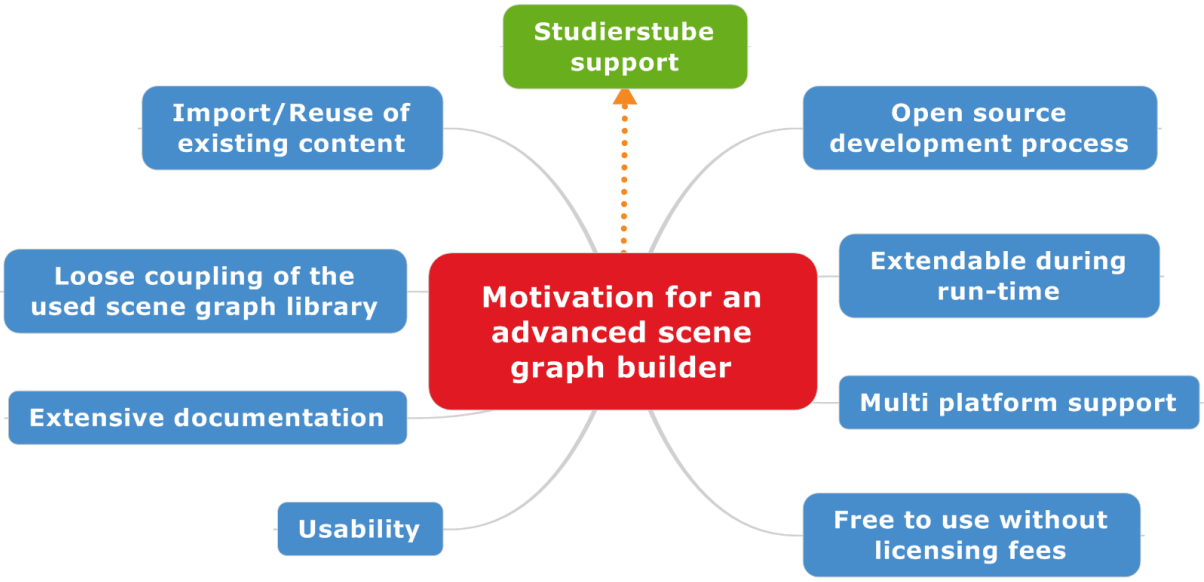


Figure 1.1: Overview of the motivation for Studierstube Builder.

the acceptance can be increased by supplying the user with a well designed user interface. Finally, a tool which allows modifications of the scene graph during run-time with an instant update of the rendering result can supply the user with a direct feedback loop. This can speed up the development process or learning of a novice user.

A tool which integrates with Studierstube is a powerful not only for general scene graph development but for the development of augmented reality applications. Much research at our institute is focused around augmented reality applications. With the recent advantages in smartphones, many applications which required expensive and bulky hardware before suddenly can run on a small device which most users carry around anyway. High-end smartphones offer all sensors required for augmented reality applications and it will not be long until some of the advanced hardware penetrates the mass market. With Nvidia's Tegra chip, a new generation of powerful, mobile hardware is about to become reality. Once these devices penetrate the mass market developers and researchers suddenly can develop augmented reality applications for a much larger audience. With Studierstube, researchers and developers have a well known and established, powerful framework at their fingertips, which performs many tasks automatically and which builds upon well tested tools. As the entire Studierstube architecture is built around custom scene graph nodes which are linked together, a well designed scene graph editor could simplify and accelerate the development of new, cutting-edge augmented reality applications. As the augmented reality market is expected to grow and evolve within the next years, tools which accelerate the development process will be beneficial.

Currently, there is no scene graph tool on the market which meets all the demands present, so development of a new graphical editor was started. This work describes all aspects of the creation of this new tool, called Studierstube Builder. Figure 1.1 gives an overview of the respective goals which will be discussed in the upcoming subsections.

1.4.1 Open Source Development Process

Once a stable structure is built and a set of basic features is implemented, Studierstube Builder should become open source allowing other developers to use and modify it. Hopefully a number of interested users will get aware of the tool and start using it. That gives the opportunity the get some practical experience with a larger number of users, concerning bugs, usability and problems concerning different system configurations. Furthermore interested programmers can implement new features as they need

them, fix existing bugs, and improve the overall software architecture. New features can be peer reviewed before being released into the tool itself.

1.4.2 Extendable During Run-time

Scene graphs are usually highly extendable software libraries. No library can ever be complete because different users will have different requirements, resulting in custom scene graph nodes. A fundamental requirement to any scene graph tool is to reflect these demands. It should be possible to add custom nodes to the scene graph editor. This is usually done trough a number of function calls in the main method of a user written software. However, this would require to change and recompile the editor's source code. This is not a desirable solution. A more flexible solution would allow custom nodes to be added to the editor during run-time. This can be done through the use of shared libraries which contain all necessary information. It should be possible to specify a number of shared libraries which are loaded during the start of the program.

1.4.3 Multi Platform Support

Building a scene graph editor means adding another layer of abstraction on top of the scene graph library. All of the scene graph libraries mentioned in section 2.7 are built on top of OpenGL. As OpenGL is a platform independent graphics library, scene graph libraries are platform independent as well. The scene graph editor should therefore offer multiple platform support too. It should not be bound to any specific hardware or application development system.

1.4.4 Free to Use

Making the tool available as open source does not necessarily mean it can be used for free. However, allowing users the free use of the tool will greatly increase the number of prospective users. It is planned to release the source code and allow free usage for the tool to everyone.

1.4.5 Usability

Some of the available tools require the user to read the documentation in order to able to use the program. Making the GUI as clear and intuitive as possible should allow the average user to utilize the tool with the least possible amount of training time. The user interface design should follow well established design patterns. To achieve this goal, a few steps have to be taken. To begin with, all actions should lead to instant visual feedback to the user, making the performed action immediately clear. This applies to the GUI used by the program and to a live-preview of the scene in work. All changes made to the scene graph should immediately be reflected in a preview being shown. Then, all icons used in the GUI should be as unambiguously as possible. Tooltips should further increase the expressiveness of all icons used. Furthermore, to avoid cluttered display for larger scene graphs, the user should be able to hide certain parts of information, allowing different level of details. Moreover, the user should be able to show or hide certain parts of the GUI such as menus or toolbars or switch between different viewing options. Finally, to allow the best possible usage throughout different screen resolutions or even multi-monitor systems the entire GUI should allow flexible resizing. Large parts of the GUI should furthermore be designed in a modular, "dockable" manner. Dockable in this context means that certain user interface elements can be treated as separated windows and placed for example on a second screen to allow more space on the primary screen. Finally, a developer should be able to change the GUI to add for example new icons or entries in the context menu. These changes should not be more complex as absolutely necessary.

1.4.6 Extensive Documentation

To enable other developers to quickly extend a program a certain level of documentation is required. Of special importance is the existence of a high level overview over the system architecture. Each class should have an overview and a more detailed section allowing other programmers to access the most relevant information with a glimpse. The more important a class, the better the documentation should be. The documentation should be made available in different formats(pdf, html, ...), suitable for each user.

1.4.7 Loose Coupling of the Used Scene Graph Library

Although different scene graph libraries offer a different set of features, many of the underlying ideas can be applied to all of them. [Bale and Chapman, 2007] came to the conclusion that 80% of the functionality is identical in the different scene graph libraries. That leads to the idea of a so called louse coupling between the front-end which manages for example the GUI and the user interaction and the back-end in the form of the scene graph library. It could therefore be possible to allow the same graphical frontend to use different scene graph libraries. A feature like that obviously needs careful testing and it has to be considered how operations that are for example only supported in one of the scene graph libraries can be integrated into the GUI without messing up the user interface for other users.

1.4.8 Import and Reuse of Existing Content

As there is much existing material used throughout different projects, this content should be utilized. It is therefore a requirement to allow the loading of existing scene graphs and a reuse of custom written nodes. Only a scene graph editor which fulfills these requirements will gain wide acceptance.

1.4.9 Studierstube Support

One special requirement is Studierstube support. It should be possible to create and modify scene graphes used by Studierstube with the scene graph editor. Studierstube requires a very specific configuration to run. Many of the prospective users will probably only want a scene graph editor without the need for an integration of the Studierstube framework. To serve both user groups the entire Studierstube integration should be optional. That means all user interface components used only for Studierstube should be hidden and performance should not be effected by Studierstube specific source code if Studierstube support is not required.

1.5 Thesis Structure

Chapter 2 introduces the most important concepts of scene graphs and other related work. This includes an overview of historical development, some definitions regarding the graph structure used, the fundamental principles all scene graphs use, and more advanced research topics on scene graphs. In addition, a number of scene graph libraries and tools are introduced. Moreover software and user interface design patterns are presented. The chapter closes with an environment analysis which has been made before the implementation started. During chapter 3, an overview of Studierstube Builder is given. The user interface is shown, functionality discussed and the user interaction is presented. Moreover, the tools for the implementation are introduced and a high level overview of the interrelationship between different program components is shown. Chapter 4 discusses the implementation into detail. Where relevant, different alternatives are compared and a reasoning for the chosen option is given. Chapter 5 shows examples where Studierstube Builder is used to quickly create scene graphs for different application areas. Finally, chapter 6 summarizes the thesis.

Chapter 2

Related Work

This chapter introduces the reader to the all aspects of scene graphs. The first sections starts by establishing the definition of a scene graph (section 2.1) followed by an overview of the development in this area over the last two decades (section 2.2). Section 2.3 introduces scene graph concepts by useing the Open Inventor library as an example. After this, section 2.4 presents different ways a scene graph library can be used to improve performance. This leads to the more advanced topics of multiprocessing and multithreading (section 2.5) and distribution of scene graphs (section 2.6). Different scene graph libraries are discussed during section 2.7. Moreover, section 2.8 introduces file formats used to store scene graphs. Section 2.9 presents a number of existing scene graph tools. In addition, a number of design patterns used during software design and user interface design are introduced in section 2.10 and 2.11. Finally, section 2.12 analyzes existing tools which influenced this work.

2.1 Scene Graph Definitions

The definition of a scene graph is somewhat vague, because it can be used for the realization of different applications. The definition used here is as follows: A scene graph is a hierarchical data structure used to organize so called nodes. There are usually three kind of nodes: a root node, group nodes and leaf nodes [Bale and Chapman, 2007]. The root node (sometimes just referred to as 'root') is the topmost node and the only node in the entire scene graph having no parents. Group nodes can have one or more children connected to them and can in turn be the child of one or more other group nodes. Group nodes are used to structure the scene graph in a certain way. It can be seen that the root node is a group node as well since it can have other nodes as children. Leaf nodes represent the actual content of the scene for example a three-dimensional object. They can have no child nodes and are connected to one or more parent nodes. The data structure created this way leads to a tree or graph structure. Most commonly used is a directed acyclic graph (DAG) structure [Walsh, 2002]. A DAG is defined as:

> A graph with no path which starts and ends at the same vertex *(Black [2004])*.

The biggest advantage of the DAG over a tree structure is that it allows nodes to be child of multiple parents. This enables so called node sharing, an important technique which will be discussed later. Figure 2.1 shows different types of nodes inside an exemplary scene graph structure. One notable difference between a mathematical graph and a scene graph is that scene graphs are heterogenous, which means they have different types of nodes as mentioned before.

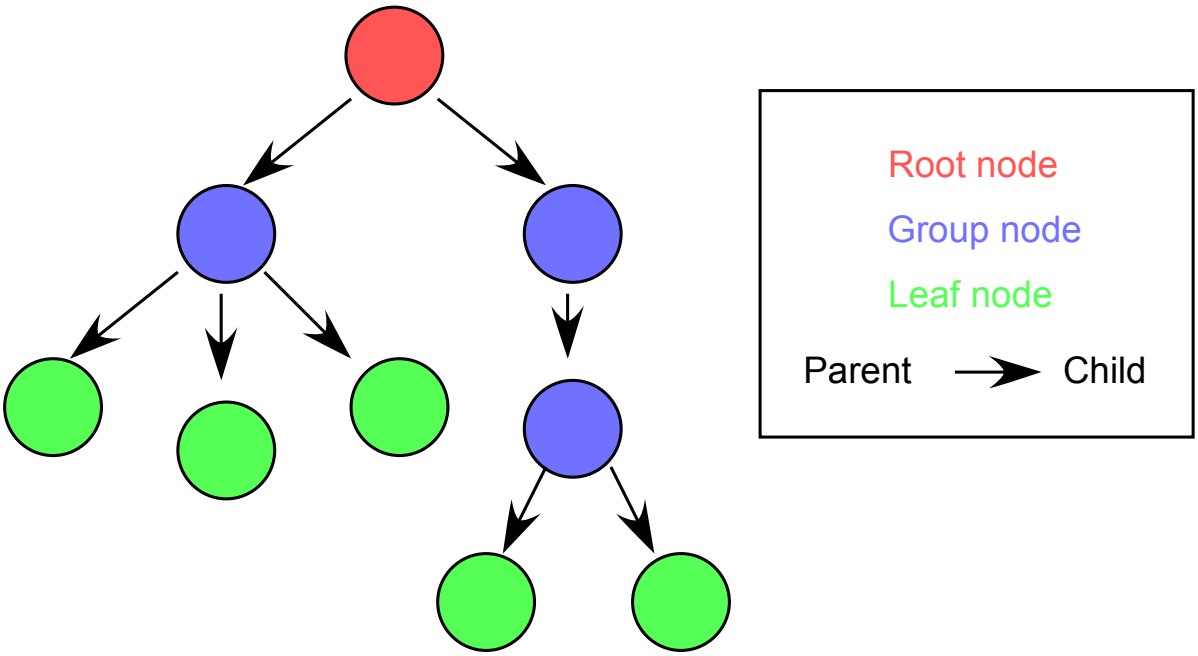


Figure 2.1: Exemplary structure of a scene graph showing the different types of nodes inside a scene graphs. Note: The above structure represents a pure tree structure

2.2 History of Scene Graph Development

Before scene graphs were introduced, programmers usually developed their applications by mixing graphics related code with functional code. The resulting code therefore tightly interweaved different aspects of their programs. Changes to the code were therefore tedious and error-prone [Walsh, 2002]. According to [Strauss and Carey, 1992], approaches used to create three-dimensional scenes at that time used one of following two technologies.

The first is a so called display list. Objects are defined as a series of drawing commands which are stored on the GPU and are executed on command. The data structure created for display lists is usually very efficient and it can be compiled if the scene does not change dynamically. However the compilation is costly and the created data structure is highly optimized for rendering. This means that for example the geometry contained in a display list cannot be used for other tasks such as collision detection. This results in a heavy memory penalty for systems using display lists, since different representations of the same data has to be kept in memory if they are required for other tasks as rendering [Rohlf and Helman, 1994]. An exemplary system which used display list is [van Dam, 1988].

The second is a so called immediate mode library. It provides an efficient low level interfaces to the graphics hardware, but leaves all other tasks to the application. During immediate mode, the host application must constantly feed the graphics hardware with primitive, vertex, normal and attribute information [Rohlf and Helman, 1994]. It must take care that the transfer of the graphical primitives is done as efficiently as possible, since inefficient transfer can strongly effect graphics performance. It should be noted that many of the immediate mode libraries do support display lists as well. The best known example of an immediate mode library probably is OpenGL, as it is for example described by [Neider and Davis, 1993]. Older examples include [Hewlett-Packard, 1991] and [Upstill, 1989].

Because 3D application development with a low level library was so time consuming, many development teams wrote their own abstraction libraries suiting their particular needs. At that time Silicon graphics, a major manufacturer of high end graphics workstations, realized that the graphic development process could be significantly accelerated by relieving developers from performing the same low-level tasks over and over again. Their main goal was to simplify this task by developing an extensible toolkit

which eliminates some of the shortcoming of the other approaches. [Strauss and Carey, 1992] had three fundamental areas they wanted to improve:

- Object representation: The programmer should focus on what he wants to create not on how this is ultimately drawn. The programmer should thus be able to change the properties of an object and the toolkit should take care how to process it.
- Interactivity: The user should be able to directly interact with the three-dimensional scene. This was done for two-dimensional user interfaces so far but not for three-dimensional scenes. An event model which supports this interaction should be integrated into the toolkit.
- Architecture: The toolkit should be designed in the most flexible way, not enforcing any unnecessary constraints on the application developer. It should furthermore be extensible, allowing developers to make inevitable modifications and extend the functionality to meet their particular needs.

Ultimately the work done by [Strauss and Carey, 1992] lead to the development of the Open Inventor library, which set the ground for all modern scene graphs used today. Although technology has rapidly evolved many of the fundamental ideas and key concepts defined in Open Inventor remain almost unchanged until today. At present, many different scene graph libraries exist, developed in a variety of programming languages. [Bale and Chapman, 2007] investigated the five most commonly used libraries and came to the conclusion that about 80% of the functionality is available in all of the libraries only 20% is what makes the difference.

The main areas of research since then have been on how to improve rendering performance and how to 'share' a scene graph by multiple computers to allow a collaborative experience for multiple users. Both topics will be discussed in more detail after introducing the basic concepts of a scene graph.

2.3 Basic Scene Graph Concepts

The purpose of this section is to introduce common scene graph concepts by using the Open Inventor library which was first introduced by [Strauss and Carey, 1992]. It should lay the fountain for some of the more advanced upcoming sections. Open Inventor is widely acknowledged as a milestone in graphics programming and is very suitable for introducing the basic scene graph concepts. Most of the ideas introduced by Open Inventor are valid for the other scene graph libraries in one way or another. An excellent introduction into Open Inventor are the two books written by [Wernecke, 1993, 1994].

2.3.1 Graph Traversal

A scene graph usually uses a DAG as it was defined in section 2.1. The underlying DAG structure can either be created by direct programming of the chosen scene graph library or by using a special file format. The scene graph structure is created during loading of the file. An example for the latter approach is the work done by [Carey and Bell, 1997]. To perform certain operations, like for example rendering, all or some of the nodes in the graph have to be visited and the necessary information must be collected. This process is called traversal of the graph. Graph traversal is usually done by recursively iterating over all nodes in a graph starting at the root node. In order to get a consistent result the order in which the iteration is done is important as it directly influences the final result. There exist different approaches to traverse a DAG, some of them can be found in [Cormen et al., 2001]. The one used most often is the so called depth first traversal. This traversal is defined by [Cormen et al., 2001, section 22.3] as follows:

Perform the following operation recursively at each node

1. Visit the node.

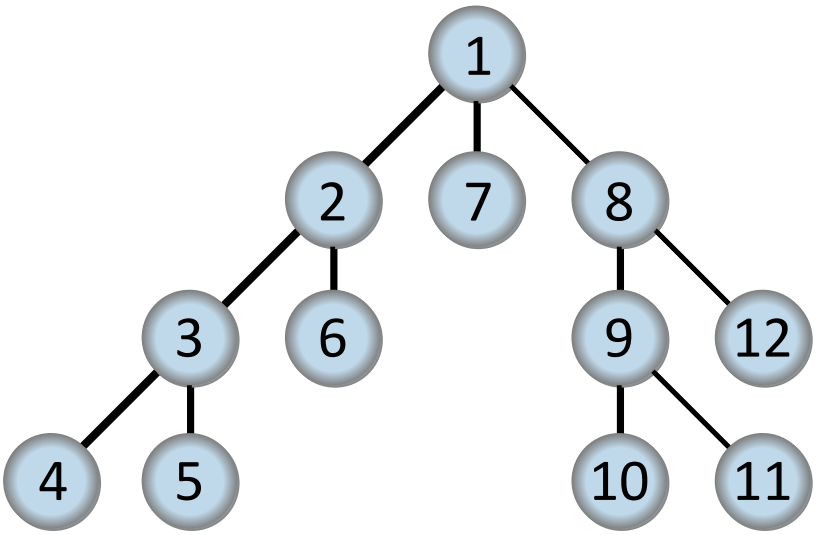


Figure 2.2: Depth first traversal of a tree structure. Starting at the root node depth first traversal visits the child nodes in a depth first, left to right order. The number inside the nodes represents the order of traversal. The first node visited is the root node, the second its left child, and so on. The last child visited in this traversal order is the right-most node. [Published by [Drichel, 2008] under GNU Free Documentation License]

2. Traverse the left subtree.

3. Traverse the right subtree.

Figure 2.2 shows a graphical demonstration of the above algorithm. During traversal nodes can update the so called traversal state. The state includes elements such as the current transformation, current material and the current drawing style. Each visited node can leave the state unchanged, replace values with their own or concatenate values, depending on the type of the node. Concatenation of values is for example used by geometrical transformations.

2.3.2 Nodes

While the group node and the root are used to structure a scene graph, its nodes represent the actual objects or properties of a three-dimensional scene. Nodes can be some geometric primitives such as cubes, spheres, cones, cylinders or other arbitrarily complex polygonal shapes [Strauss and Carey, 1992; Wernecke, 1993]. This type of nodes is sometimes also called shape nodes. There are other nodes which represent cameras, lights, materials, textures, text or nodes used to animate objects. Furthermore, to allow interaction with the user, there is a number of so called manipulator nodes which range from a simple dragger to some more advanced manipulators. A dragger is a special node which provides a user interface which can react to user input. Usually the user clicks on a dragger and keeps the mouse button pressed, while performing some action like moving an object around. Depending on the library used the number of different node objects can easily be in the order of hundreds, not included the vast number of custom-built nodes which have been developed to implement a particular special requirement. Usually nodes have some instance specific properties like for example the width, height and depth of a cube. Instead of storing its properties in a simple member variable of the respective class they are stored using so called fields. Since fields are not basic data types but classes they allow some useful behavior. If the value of a field changes for example, the library can schedule a new render pass or modify other fields depending on the current field. Fields can be used create so called field connections. After creating a

field connection between two fields, changing one field value automatically changes the second field to the new value. This functionality uses the Observer Pattern defined in [Gamma et al., 1995].

As with nodes, each library implements a number of different group nodes as well. Group nodes decide over two important properties: traversal and inheritance. While most group nodes traverse their children using the earlier described depth first approach, it can make sense to use other approaches. A group node could for example hold the same child node a number of times, each time with a different level of detail, traversing only the child node with an adequate level of detail for the current distance from the viewer. While it is sometimes desirable to inherit properties from parent to child nodes, it can be useful to isolate the rest of the scene graph from the changes made in a particular subgraph. Open Inventor uses a node called Separator to achieve the latter. It makes a copy to the entire state before traversing the Separator's children and restores the state once all children have been traversed. It thus shields changes made to the state below a Separator node from affecting other parts of the scene graph.

2.3.3 Actions

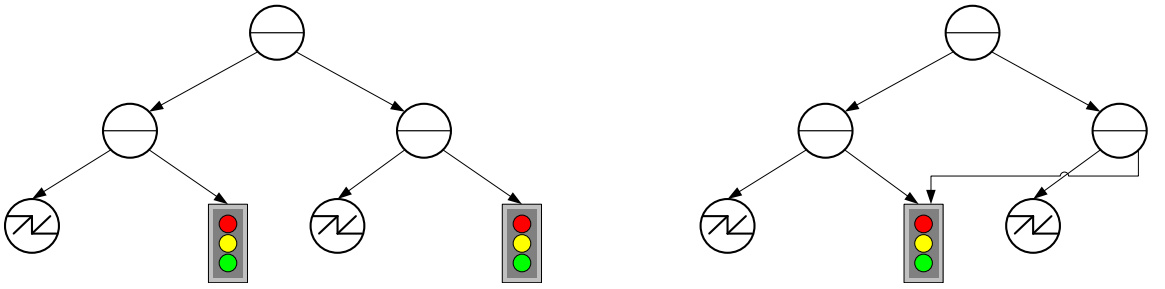
Once the structure of the scene graph has been created, the user can perform different operations, so called actions on the entire scene or on parts of the scene graph. Examples of actions include rendering, managing user events, bounding box computation, searching for a specific node or writing a scene graph to a file. While it is relatively simple for newly created nodes to implement the standard actions defined by the scene graph library, allowing for example the creation of new actions for existing nodes is more difficult. One would have to subclass the node and add the newly created behavior in the subclass. This problem called two-way extensibility problem is solved by [Strauss and Carey, 1992] using a two-dimensional table with all the nodes and all the actions on the respective axis. Each entry in the table is a function pointer pointing to a method which implements the desired action for the processed node. Applying an action to a node results in traversal of this node, as it was described in section 2.3.1 using the state to accumulate information.

2.3.4 Paths

As it was mentioned in section 2.1, it is possible for a node to have multiple parents as long as no cycles are introduced in the graph. This approach if favorable because it allows the sharing of nodes which appear more than one time in the scene. Imagine for example modeling a city. Many objects like for example traffic lights appear many times inside the scene. Allowing a node to be child of only one parent would make it necessary to have many copies of the same object in the memory. This clearly contradicts the goal of efficient real-time graphics. Sharing a node like the traffic light mentioned before allows for efficient reuse of common objects. Figure 2.3 revisits the traffic light example. The support of node sharing however implicates that it is no longer possible to unambiguously refer to an object [Strauss and Carey, 1992]. Selecting one particular traffic light and getting the underlying node will always return the same (shared) node. To deal with that, so called path objects are introduced. A path always unambiguously refers to a particular object by creating a chain of objects from a certain node (in most cases the root) down to the selected object. The path includes all nodes below the last node and all nodes that have an effect on these nodes as well thus spanning an entire subgraph. All actions can not only be applied to nodes but also to paths. Usually a path is created as a result of mouse interaction by the user. The user clicks on a particular spot in the rendering window and the library returns a path to the foremost object at that position. This process is also called picking.

2.3.5 Sensors

Sensors are objects which monitor the scene graph and call a user defined callback function when a change is detected or a certain amount of time has passed. The callback function is a function called



(a) Scene graph without sharing of the traffic light node

(b) Scene graph sharing the traffic light node

Figure 2.3: A simple example showing the use of node sharing. (a) shows a scene graph which created two instances of the same traffic light node. (b) in contrast reuses the traffic light node by using node sharing. Only one copy of the traffic light node is kept in memory. Before rendering different transformation nodes are used to make the same object appear at different places.

when the sensor is triggered. Sensors can be used to animate the scene or carry out certain actions after some data is changed. Open Inventor [Wernecke, 1993] supports two kind of sensors. Data sensors and timer sensors. Open Inventor uses different data sensors. A FieldSensor is triggered if a field changes its value. A NodeSensor is triggered if a certain node or any node below this node changes. Finally, a PathSensor is triggered whenever any node contained within a path changes. Timer sensors can trigger at regular intervals or after a specified amount of time allowing animations of the scene graph.

2.3.6 Manipulators and Draggers

Draggers and manipulators are special objects which have a user interface and that can respond to user events [Wernecke, 1993]. They work together to enable certain interactive features of the Open Inventor library. Draggers are special nodes which enable the interaction with the user. To allow this interaction a dragger adds geometry into the scene. Draggers have fields which can be connected to other parts of the scene or they can perform callback functions at the beginning, during or at the end of interaction with the dragger. A number of draggers to translate, scale or rotate nodes are built into Open Inventor.

Manipulators use one or more draggers to enable certain user interaction. While a dragger does not directly influence the scene graph, manipulators do. Manipulators usually replace a node with an editable version of that node. This is done by using one or more draggers that modify the manipulator. Manipulators make changes to the scene graph that directly affect the traversal.

2.3.7 Node Kits

A scene graph does allow any structure the user may find useful as long as the resulting graph is still a DAG. Sometimes it is however desired to enforce some kind of structure to a scene graph. Node kits are one way to create this structure [Wernecke, 1993]. They are arbitrary large collections of nodes which may contain other node kits as well. The node kit itself takes care of all the nodes it manages. Node kits provide efficient creation and management of their children. One does therefore not have direct access to its children but has to use the node kits access functions. The children are therefore sometimes called hidden children. Each node kit has a catalog which lists all available parts (=nodes). The creator of a node kit can set some of the parts to be created by default and others to be created only on request. An additional benefit of node kits is that they allow to enforce some shared semantics over node kits [Strauss and Carey, 1992]. A programmer could for example create a node kit for airplanes that can be used to create different types of airplanes. The user who ultimately uses the airplane node kit does not care how

a specific airplane for example raises its landing gear as long as there is a method to do so.

2.4 Common Ways to Improve Performance in Scene Graphs

One very common technique – node sharing – which is used to save memory and simplify the scene graph structure was already introduced in section 2.3.4. Many of the techniques are based on group and especially on Separator nodes. They are very commonly used in Open Inventor. As was mentioned before, a Separator node saves a copy of the traversal state before continuing the traversal and restore the original traversal state after all children have been traversed. Since they are so common, a lazy copy scheme is used [Strauss and Carey, 1992]. That means instead of copying the entire state each time a Separator is traversed, copying of data is postponed until any element of the state is modified. This is a common technique to avoid unnecessary work. It is for example used by operating systems [Tanenbaum and Woodhull, 2006]. Another technique used by Separator nodes is state caching [Strauss and Carey, 1992]. Once a particular subgraph has been rendered, a display list is stored which can be rendered very quickly [Rohlf and Helman, 1994]. This display list is used until a modification of any node which was used to create the display list is detected. This automatically leads to an invalidation of the cache created this way. Separators can improve picking performance as well by caching bounding boxes. A bounding box is defined by [Angel, 2008] as ”The smallest rectangle, aligned with the window, which contains the polygon”. Sometimes a sphere is used instead of a rectangular bounding box. Bounding boxes work in two and in three dimensions and are frequently used to accelerate clipping. Clipping is the process of removing geometry which can not be visible in the final, rendered image because they are outside of the frustum. By using the bounding box of a group node, the scene graph library can quickly decide if the subgraph will be visible in the final image or if it is outside of the viewing frustum [Rohlf and Helman, 1994]. Furthermore, [Rohlf and Helman, 1994] analyzes the spatial arrangement of bounding boxes in the scene graph and introduce a special data structure if the original layout is not very efficient for quick tests. Moreover, scene graphs can avoid unnecessary mode changes. They can cache the currently set mode and thus avoid setting the same mode twice. This is especially true in a multi-threaded environment where a modification of the mode basically blocks all other threads. It should be noted that many of the tasks described here (like for example culling) are nowadays performed by powerful graphics hardware.

2.5 Multiprocessing and Multithreading in Scene Graphs

When multi processor systems appeared on the market in beginning of the 1990s this was an obvious way to further improve the performance of scene graph libraries and thus rendering. [Rohlf and Helman, 1994] were the first to come up with a scene graph library which supports multiple processors. Their fundamental goal was to hide the complexity introduced by splitting the work on multiple CPUs from the application developer while allowing the greatest flexibility to develop high performance applications. They split their library into two parts, one dealing with low level, immediate mode graphics drawing, the other implementing the scene graph functionality. Different mechanism for both the low level and the scene graph library allow the multi processor support:

- Shared, reference counted memory allows different processes to share common data.
- Different data structures such as queues or multibuffered memory are support. They allow locking data and automatic synchronization mechanism between different copies of an array. A pipeline can be simulated by efficiently passing data from one process to the next. Or a producer/consumer architecture can be built between two processes. Synchronization is done by using well known concepts such as semaphores or locks (see [Tanenbaum and Woodhull, 2006] for more details).
- The traversal is split into different, independent stages which can be performed by multiple processors. First, the scene graph is checked for intersections, required for example for collision

detection. Second, culling is done rejecting invisible parts of the scene and computing the necessary level of detail for models. Third, the actual rendering is performed.

- Rendering different viewpoints can be performed on different processors.

It should be noted that by creating a pipeline of different steps one improves memory throughput on the cost of latency.

Iris Performer supports multiple processors but it does not support multi threading, something which can be done by OpenSG [Reiners et al., 2002]. To allow threads to work (read/write) with scene graph data they need to replicate the data in each thread, leading to very inefficient memory duplication and synchronization overhead. Based on the observation that the scene graph structure requires only 10% of the data while the geometry and textures make up 90% of the data only the scene graph part is replicated by default [Roth et al., 2004]. All other data is kept in fields, as described in section 2.3.2. The fields are stored in FieldContainers. Instead of replicating each field only the FieldContainers are replicated. While single values are replicated as well, so called MultiFields are not replicated and a pointer to the same instance is stored instead. Using a lazy copy scheme the data is only copied if necessary. The approach chosen by OpenSG for efficiency reasons is that change detection is not automatically done by the scene graph library, but the application has to notify the scene graph on change. Once changed the FieldContainer keeps a bitmask of the changed fields so that upon synchronization with other threads only the changed parts of the scene graph are copied [Reiners et al., 2002].

2.6 Distributed Scene Graphs

The previous chapter dealt with ways to facilitate multithreading and multiprocessing to improve the performance of a scene graph. As graphic hardware becomes ever cheaper and more powerful researchers were looking for ways to split the work not only on different processors or threads but on different computers instead. A second influencing factor was the wish to allow collaborative experiences between many persons sharing one virtual world. While a couple of distributed virtual environments (DVE) such as [Zyda et al., 1992; Park et al., 1996; Shaw and Green, 1993; Das et al., 1997] were developed, they were not generally applicable for distributed computer graphics [Hesina et al., 1999]. They also suffered from dual database problem. That is they kept the application state and the graphical state separate from each other trying to keep them synchronized. Distributed scene graphs (DSG) have to deal with some other problems as well [MacIntyre and Feiner, 1998]. First, changes made on one computer have to be synchronized to other machines as well. Second, the medium (some kind of network) used to synchronize has a limited bandwidth and some latency. This makes it necessary to reconsider the distribution of the underlying scene graph. To allow interactivity and high performance, usually some data has to be kept locally and the amount of data being transferred over the network heavily affects performance and scalability. Third, not all changes have to be reflected on all other machines. It should for example be allowed to locally select an object without necessary reflecting this selection to the other machines. These requirements make the use of traditional remote procedure (see for example [Sun Microsystems, 2009]) calls not the best option. Network topology further influence the design. While a client/server architecture might be the best option for some applications – such as many games on the internet – others may want to use a peer to peer structure.

Repo-3D [MacIntyre and Feiner, 1998] is such a DSG. It was developed with the goal in mind to make distributed applications almost as simple as local applications. It uses a object-oriented language called Modula-3. It approach is somewhat similar to to the Java RMI, all locally performed methods are atomic and serializable, which means they can't be interfered by other calls and they can be sent over the network. Each update to an object is serialized and called on all other copies of the same object in the same order. To ensure that each message is called in the correct order each function call becomes a sequence number and execution stops until all replicated objects have received and performed all function

calls with a lower sequence number. Reading access is not regarded an update and therefore does not lead to a change of the object. While this approach is certainly easier to implement, the latency involved will likely be a bottleneck. Furthermore using a uncommon language like Modula-3 will discourage many users.

Distributed Open Inventor (DOI) developed by [Hesina et al., 1999] is another approach to distribute a scene graph over multiple machines. DOI uses a local copy of the scene graph for each machine to allow interactive framerates. User interaction is divided into input streams and input events. Input streams usually do not involve complex computations but require an immediate response by the application. Examples are the user moving the camera around or interacting with a dragger. Input streams are processed only locally and sent to the other machines upon finishing the interaction. All other events which do not require an immediate response are called input events. DOI uses only a couple of different messages to distribute local changes over the network. As done by [MacIntyre and Feiner, 1998] as well, DOI does only propagate updates to a fields value to other nodes, reading is done only locally. Changes to the scene graph structure are quite simple as well. The creation of nodes only requires their type, index and parent to be distributed, deletion only requires the name of the node. Nodes are named by their path from the root. New subgraphs are created efficiently by sharing the URL or the file which is used for the subgraph instead of sending a message for each new node. The support for Local changes further increases the applicability of DOI. The distribution of the network messages is done with sequence numbers as well and distributed using the User Datagram Protocol (UDP) [Postel, 1980]. The implementation of DOI is done using Open Inventor and C++ [Stroustrup, 2000]. A node sensor, as described in section 2.3.5, is used to call a callback function which takes care of sending a change modification message to the other nodes. By facilitating a standard language and a standard scene graph library, DOI is likely to attract more users than Repo3D. DOI was successfully integrated into the Studierstube framework [Schmalstieg et al., 2002] and has been been used for example in [Reitmayr and Schmalstieg, 2001].

Scene-Graph-As-Bus (SGAB) takes a little different approach. Its design goals include the creation of distributed, heterogenous, standalone graphics applications. It is assumed that all the applications are using a scene graph, although they may use different scene graphs libraries. The different libraries are assumed to be similar enough to each other. The fundamental idea is to create a mapping layer for each different scene graph. The mapping layer maps the scene graph specific library to a so called neutral scene graph representation (NSG). The changes made to the NSG are then forwarded to the other participants in the network and again mapped from the NSG format to the scene graph specific format. By creating such a mapping layer for one scene graph library many applications using this library can make use of the SGAB design. Changes made in one scene graph are detected by using callback functions as it is used by Distributed Open Inventor as well. To allow extensibility, each mapping layer can register functions to be called by the NSG for different events. To facilitate the use for different uses both a peer to peer and a client/server approach were implemented. There are however a couple of limitations to the presented approach. First, not all scene graph libraries support change notification (by calling a defined callback function). Performer [Rohlf and Helman, 1994] and Java3D [Sowizral et al., 2000] for example do not. So, some custom mechanism has to be implemented into the respective mapping layer. Second, duplicating the scene graph, mapping it to different representations, transporting it over the network and keeping up the consistency adds quite a bit of a computational overhead to the applications. Third, although scene graphs are pretty much alike subtle differences can complicate the implementation or do not allow some parts of a scene graph library to be mapped to the NSG format.

2.7 Different Scene Graph Libraries

As was said before, there are a couple of widely used scene graph libraries. This section gives an overview over the concepts used and the differences concerning functionality. The scene graphs libraries presented here are: Open Inventor, OpenGL Performer, OpenSceneGraph, Coin3D, Java3D, and NVSG.

2.7.1 Open Inventor

As was already said before, Open Inventor was the first scene graph library which highly influenced all further development. Although technology changed significantly since then, many of the design decisions made at that time remain true until today. It was originally called IRIS Inventor [Strauss and Carey, 1992] and renamed later.

While Open Inventor does significantly simplify the creation of graphic applications that comes at a price in performance compared to pure OpenGL [Bale and Chapman, 2007]. The main reasons being the inefficient management of state changes and the way culling is done. More subtle weaknesses are the administrative overhead by making the entire interface virtual (see [Meyers, 2005, Item 34] for a discussion on that topic). Open Inventor also introduced the Open Inventor file format with the extension ".iv". Silicon Graphics Incorporated (SGI) stopped the development at some point and made Open Inventor available as Open Source license, allowing everybody to use it for free. Since the library is not under active development, Open Inventor does not support some advanced graphic technologies like for example programmable shaders. See the book by [Akenine-Möller et al., 2008] for more details on programmable shaders. While Open Inventor is widely used in the academic sector it is not used by many industry applications due to its performance problems.

2.7.2 OpenGL Performer

Iris Performer [Rohlf and Helman, 1994], later renamed to OpenGL Performer was again developed by SGI as a response to the poor performance of Open Inventor. Moving the main focus from ease of use to improving performance, Performer changed some of the design decisions Open Inventor was based on and introduced a number of new features.

Moving the current state out of the scene graph itself and storing them as separate objects Performer is able to do a much better job in sorting the graphic state which gives a performance boost [Bale and Chapman, 2007]. As was already mentioned in section 2.5, Performer supports splitting the work on multiple processors and allows the creation of a pipeline structure. That means that tasks like culling can for example be performed on one processor while rendering is performed on another processor, using the results of the first processor. Performer supports a variety of file formats for models to be used. It is still under active development, allowing recent technologies to be supported. Unlike Open Inventor, Performer is not for free and requires a commercial license.

2.7.3 OpenSceneGraph

OpenSceneGraph is another Open Source project [The OpenSceneGraph Community, 2009]. It was released 1999 and grew to a full size scene graph library since then. Developed in standard C++ making use of the standard template library (STL), it is platform independent.

As was mentioned in section 2.5, OpenSceneGraph can make use of a multi threaded architecture. The architecture is leaned against OpenGL Performer [Bale and Chapman, 2007]. One of the key feature is the strong focus on Terrain rendering, with a number of extra tools designed especially for this purpose. While it is not documented as well as for example Open Inventor, recently a book written by [Kuehne and Martz, 2007] appeared which describes the library and uses a number of examples to get the reader acquainted with the concepts used. OpenSceneGraph uses the Lesser GNU Public License (LGPL) which forbids the use for any proprietary license.

2.7.4 Coin3D

Coin3D (see [Kongsberg SIM AS., 2009a]) was originally developed by the Norwegian company Systems In Motions. Coin3D was inspired by Open Inventor design and eventually became fully compatible

with Open Inventor using the same function interfaces.

Being fully compatible, the implementation however differs in some parts [Bale and Chapman, 2007]. Written in C++ and using OpenGL for rendering, Coin3D is platform independent. In addition different tools and Viewers exist. It is for example possible to make use of volume rendering through SIM Voleon [Kongsberg SIM AS., 2009d]. Bindings for different graphical user interfaces (GUIs) such as Qt by [Nokia Corporation, 2009h], the MSWindows Win32 API (Application programming interface) by [Microsoft Developer Network, 2009] or Xt/Motif are readily available. In contrast to Open Inventor, Coin3D is still under active development, allowing support for programmable shaders, terrain rendering and other more modern techniques. Coin3D is released under a dual licensing scheme and provides excellent documentation. While open source programmers can use the library free of charge, commercial products need to pay for their license.

2.7.5 Java3D

The first version of Java3D [Sowizral et al., 2000] was released 1998. Intel, SGI, Apple and Sun joined forces and developed Java3D together. The development was stopped for some time before Java3D was released as open source using the GNU General Public License (GPL) license .

As the name suggest Java3D is developed in Java. It can use either OpenGL or Direct3D to render graphics and runs on multiple platforms. Java3D's functionality is somewhat similar to the scene graphs already discussed. It includes support for multithreading, has facilities to create sound and can make use of programmable shaders [Sowizral, 1999]. Furthermore, head mounted displays are natively supported as well.

2.7.6 NVSG

Nvidia develops its own scene graph library as well. It is called NVSG. Although information on NVSG is hard to find, [NVIDIA Corporation, 2009b] lists some details about NVSG. According to them, NVSG can be used for high-performance graphics applications. NVSG builds on top of either OpenGL or Direct3D. It can make use of multiple graphical processing units (GPUs) and extends to graphics clusters as well. NVSG uses programmable shaders through Cg (more details on Cg can for example be found in the book by [Fernando and Kilgard, 2003]). NVSG can import the scene graph from a number of file formats.

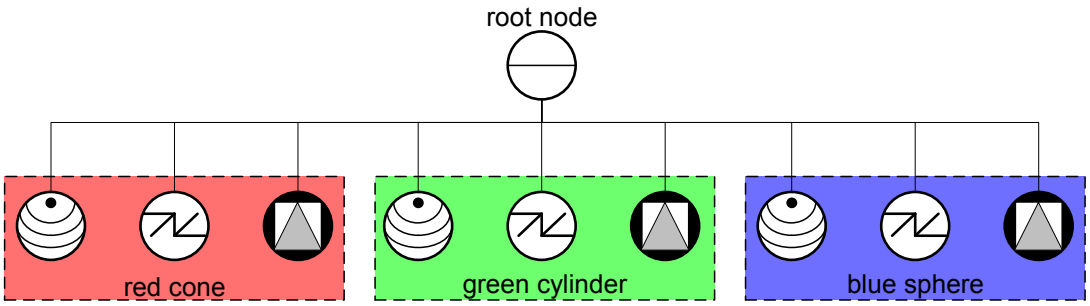
2.8 Scene Graph File Formats

Having the scene graph store its content in a file can have different purposes. One purpose of a file format is to allow a scene graph to store the current structure to a file and reload it later. Another purpose is to allow the users of scene graph libraries another interface to the library than using the libraries API. Many file formats allow the content to be stored in plain text or in a binary format. Plain text allows the user to modify or create such files by simply using a text editor and load the content of the file into the library during runtime. It also allows quick modifications to an existing scene without the need to recompile the entire program. One of the most used formats is the Virtual Reality Modeling Language (VRML) format, which has been introduced in 1994 resembling large parts of the Open Inventor file format [Thierfelder, 2004]. The intention of VRML97 was to allow three-dimensional graphics inside of web browsers. It however, never received the support of any of the major players in the web business. To view VRML, a browser plugin has to be installed first. VRML97 was standardized by [The Web3D Consortium, 1997] in 1997. The successor of VRML97 is called X3D. It extends the functionality of VRML97 but uses the Extensible Markup Language (XML, see [Bray et al., 1998] for details) to structure the file.

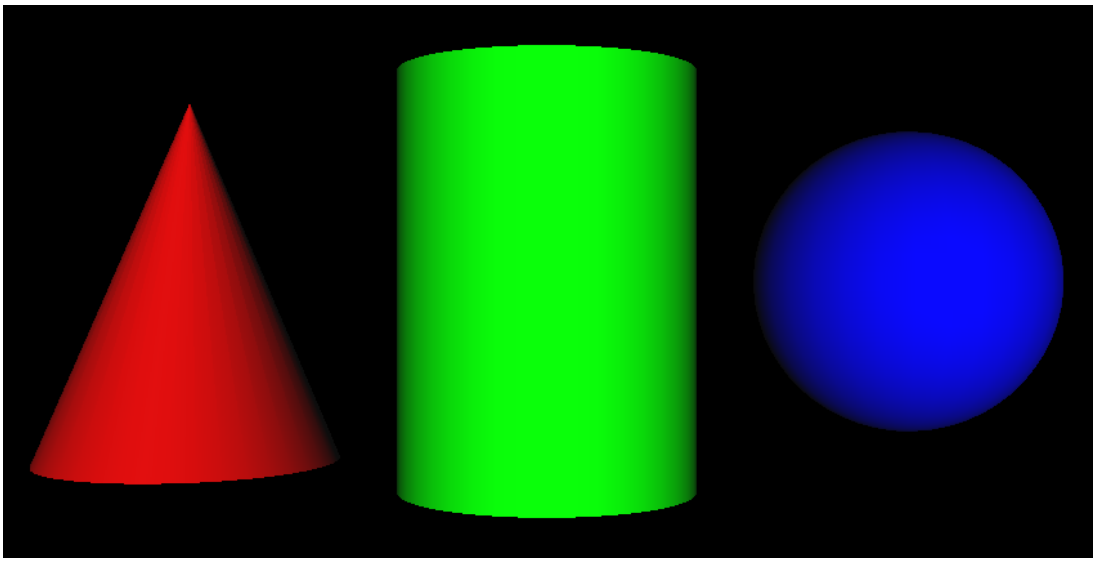
A couple of other file formats like 3DS (3D Studio Max), DXF (Drawing Interchange Format by AutoCAD, see [Autodesk, 2009]), stl (3D Systems) and obj (Alias Wavefront Object) are used by various programs. Some of them can be imported/exported into scene graphs described in section 2.7. Listing 2.1 shows an example of the Open Inventor file format in ascii mode. The scene graph and the rendered image are shown in figure 2.4.

```
1  #Inventor V2.1 ascii
2
3  Separator {
4
5    Material {
6      diffuseColor 0.9 0 0
7      specularColor 0.1 0.1 0.1
8      emissiveColor 0.01 0.01 0.01
9      shininess 0.25
10     transparency 0
11   }
12
13   Transform {
14     translation -3 0 0
15     rotation 0 0 1  0
16     scaleFactor 1 1 1
17   }
18   Cone {
19     bottomRadius 1.25
20     height 3
21   }
22   Material {
23     diffuseColor 0 1 0
24   }
25   Transform {
26     translation 3 0 0
27   }
28   Cylinder {
29     radius 1.25
30     height 3.5
31   }
32   Material {
33     diffuseColor 0 0 1
34   }
35   Transform {
36     translation 3 0 0
37   }
38   Sphere {
39     radius 1.25
40   }
41 }
```

Listing 2.1: A simple Open Inventor File, showing colored geometric primitives. The root node is a Separator. Note how to define parent/child relationships. The parents (the Separator) children are defined using curled brackets (line 3 and line 41). The order of the children is defined by their order in the text. Thus the Separator's first child is the first Material node followed by a Transform node, and so on. To show how field values are defined the first material node (line 5 to 11) is taken as an example. All fields are again surrounded by curly brackets (line5 and line 11). A fields is defined by its name and one ore more values. The diffuseColor field for example defines a red color. The value is given in RGB (red, green, blue) order with each value ranging from 0 to 1. Other fields like for example the transparency have only one value. Although the second (line 22) and third (line 32) material node share the same fields as the first one (line 5), they are not stored in the file since their values haven't changed to get a more compact representation. Note how the Transform nodes translations are accumulated.



(a) Scene graph structure



(b) Rendered image

Figure 2.4: (a) shows the structure of the scene graph and (b) shows the rendered results, both loaded from the file presented in listing 2.1

2.9 Scene Graph Related Tools

Although scene graph libraries represent a large step forward in comparison to low level, immediate mode libraries such as OpenGL, they still require a good understanding of computer graphics, spatial sense and some programming skills. Note that the later is not essential when using file formats such as VRML. These files can be loaded by a viewer to create a very simple graphic application. Depending on the file format and the scene graph library, some features might however not be accessible through the file format or can only be added during runtime. Furthermore, using the file format to create scenes is tedious, and a violation of the syntax or a simple typo will prohibit the viewing of the scene. This leads to a barrier, especially for users with no or only rudimentary programming knowledge. This section takes a closer look at different scene graph tools and how they can make it easier for the user to create scene graphs.

2.9.1 Text Editors

One of the most obvious tools to simplify the creation of scene graphs is a text editor with support for one of the file formats created by the respective scene graph library. ”Support” in this context means non standard text editor features like syntax highlighting or auto-completion. Syntax highlighting is the ability to display text which follows some defined structure with a set of different fonts and colors to enable faster recognition by a human reader. Auto completion means that the text editor can automatically make a number of predefined suggestions while the user is typing. Both features require the text editor to know the format of the file and the language syntax.

VrmlPad is one example of such a text editor [ParallelGraphics, 2009]. It supports VRML files, as the name suggests. Auto-completion is done via a drop dropdown list and it supports customizable syntax highlighting. Furthermore an automatic error detection is included which points the user to a number of problems such as syntax errors, redefinition of nodes, unknown identifiers and more. A tree style structure show a compact overview of the scene graph and can be used to connect fields. It allows the preview of the current file using some of the pre installed VRML-viewers. Usage requires a commercial license from the software producer.

2.9.2 RenderSoft VRML Editor

RenderSoft VRML Editor [Rendersoft Software, 2009] is a very simple editor which allows VRML files to be loaded and saved. A scene can be created by selecting a number of geometric primitives and perform transformations on them. Textures, material colors and lights can be added to the scene as well and simple animations can be created. There is no text mode at all, all actions are done only in a live preview window which gives the user immediate feedback to all the actions done. Although rendering is done via OpenGL the only supported platform is Microsoft Windows. The software is freeware.

2.9.3 XIP Builder

Siemens Corporate Research Inc. develops the XIP (eXtensible Imaging Platform) library and the XIP Builder. While the first is developed under a Open Source license, the latter uses a proprietary license. XIP targets at a variety of medical applications which can use the library to rapidly create new research prototypes [Tarbox et al., 2008].

The XIP library is an initiative to wrap a number of open source libraries into scene graph objects and make them interoperable [Paladini, 2007]. XIP uses Open Inventor as its scene graph library and supports other libraries such as the Insight Segmentation and Registration Toolkit (ITK) [Ibanez et al., 2003], the Visualization Toolkit (VTK) [Schroeder et al., 2007] and the DICOM ToolKit (DCMTK). The functionality of these libraries has been wrapped into scene graph objects, allowing for example image

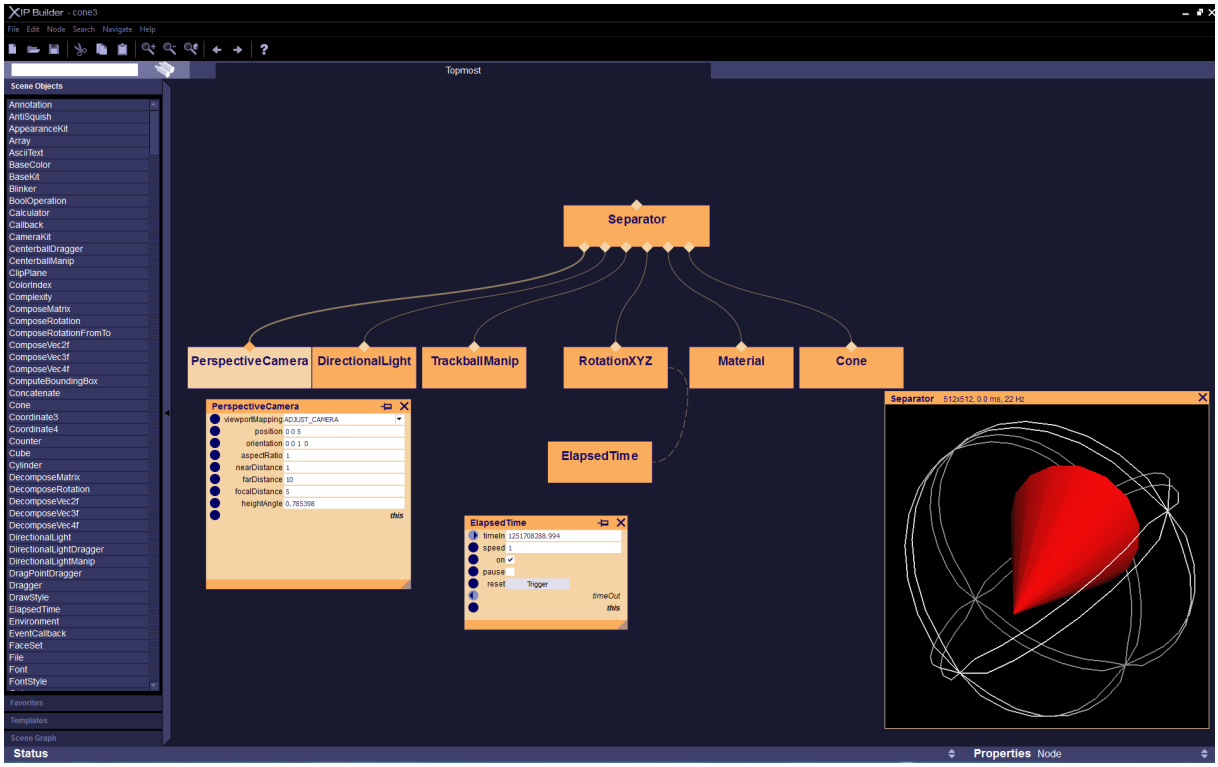


Figure 2.5: XIP Builder showing a simple scene with a rotating cone. The scene graph can be created by simple drag and drop of different nodes. Nodes can then be connected to a tree or graph structure. A live preview shows the results of the current render root.

processing pipelines to be created using the well known scene graph architecture. Furthermore there are a number of converters from one internal data representation to the other. The wrapping of libraries like ITK and VTK is mainly done by automatic scripts [Paladini, 2007]. Programmable shaders can be used via the OpenGL Shading Language (GLSL) [Kessenich et al., 2009], allowing techniques like customized volume rendering. Rendering can be done on the same machine or using a remote approach. The most recent development was the inclusion of the Compute Unified Device Architecture (CUDA), developed by Nvidia [NVIDIA Corporation, 2009a]. It enables the facilitation of the processing power of the GPU. [Gidén et al., 2008] describes the work of implementing CUDA support into the XIP libraries. It enables multiple kernels to be linked together in a pipeline. [Gidén et al., 2008] claim that their implementation of the kernel-pipeline does not allow real high-performance applications but the simple creation of the pipeline enables faster prototyping and testing of new functionality.

XIP Builder as mentioned before is not developed under open source license. It acts as a graphical user interface to the underlying XIP libraries. XIP Builder allows users to rapidly prototype scene graphs without the need to actually write code or know the API of the libraries. XIP Builder had a strong impact on the creation of this work. Figure 2.5 shows a screenshot of the XIP Builder

2.9.4 MeVisLab

Another scene graph-based tool rooted in the medical domain is MeVisLab, developed by MeVis Medical Solutions AG from Germany [MeVis Medical Solutions AG, 2009a]. It allows the user to prototype scene graphs using a graphical user interface. As the XIP Builder, it builds upon Open Inventor and OpenGL. Image processing is done either trough a custom image library or a module which uses the ITK for that task. The VTK library is supported as well. MeVisLab enables a live preview while editing and allows volume rendering to be done. It can be extended, making use of the internal interfaces or

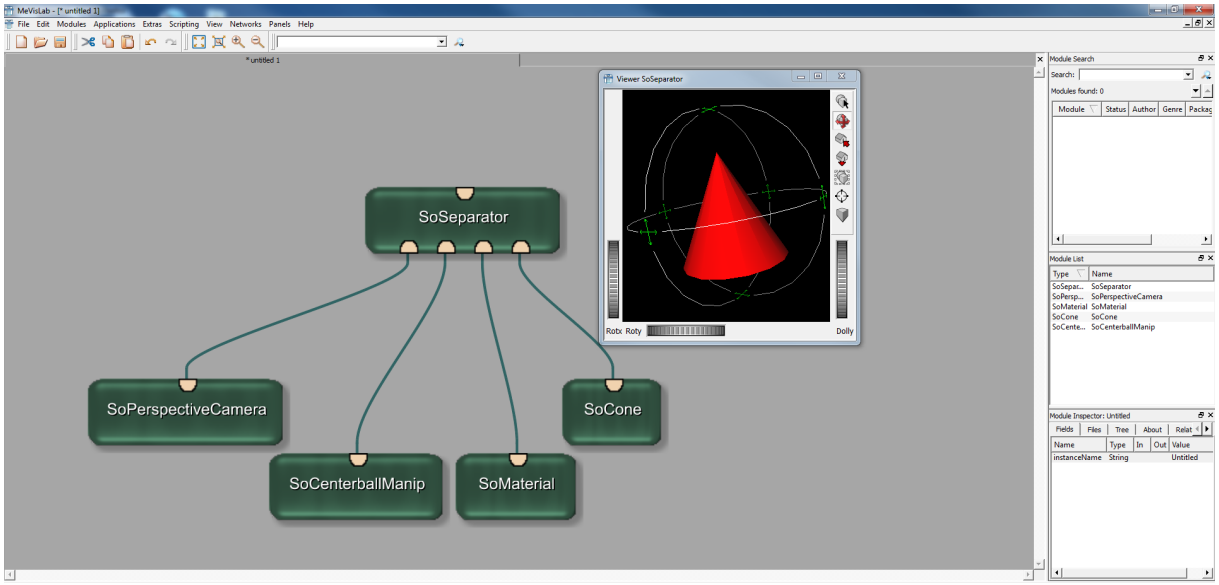


Figure 2.6: MeVisLab tool to create scene graphs. A simple scene graph with a red cone and a manipulator is shown.

using a built in macro language. As can be seen, it is pretty similar to the XIP Builder in the list of supported features. In contrast to the XIP Builder, MeVisLab includes extensive documentation (see [MeVis Medical Solutions AG, 2009b]) of the functionality and many useful tutorials. MeVisLab uses a number of different available licensing options. Free licenses are only available for non commercial applications. For free licenses a number of features is not available. Commercial licenses allow the use of all features. Prices for commercial licenses are on request only. Figure 2.6 shows a simple example with a red cone and a manipulator done in MeVisLab.

2.9.5 Coin Designer

Another tool to create Open Inventor scene graphs was developed by [Aguado and Eparado, 2005]. It is called Coin Designer. The functionality is much smaller than the one offered by MeVisLab or XIP Builder. Open Inventor nodes can be created from a list. The nodes are inserted into a tree structure instead of creating graphical representations on a kind of workspace as it is done by XIP Builder and MeVisLab. Changes to nodes are instantly reflected into a live preview. Files can be imported and exported into a number of formats. Coin Designer is open source and released under GPL license. Figure 2.7 shows a screenshot from Coin Designer.

2.9.6 Cosmo Worlds

Cosmo worlds is a VRML97 editor originally created by SGI [Rothfarb, 1998]. It allows the creation and modification of scene graphs through a GUI. Nodes can be moved, resized and rotated in one the preview windows. The scene graph is displayed in a tree view and field properties can be edited in a separate window as well. Cosmo Worlds allows the designer to create animations as well as to add sound to the virtual world. It used to be a commercial product before it was announced to become open source. Unfortunately, that never happened. Figure 2.8 shows a screenshot of Cosmo Worlds.

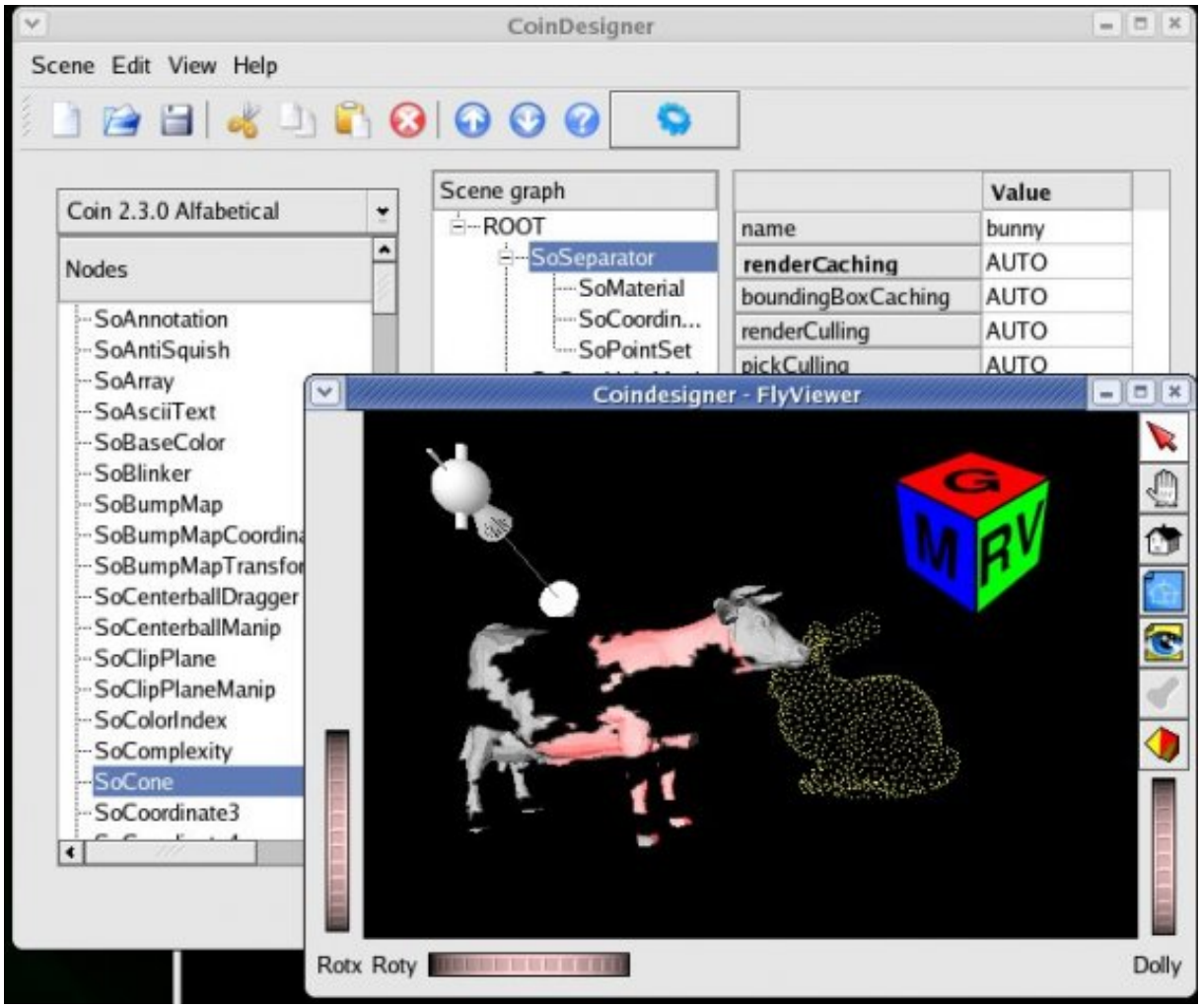


Figure 2.7: A screenshot showing Coin Designer. Scene Objects are dragged from the left list into the tree-formed scene graph and a viewer.
[Copyright 2005 Jose Miguel Espadero, used with permission.]

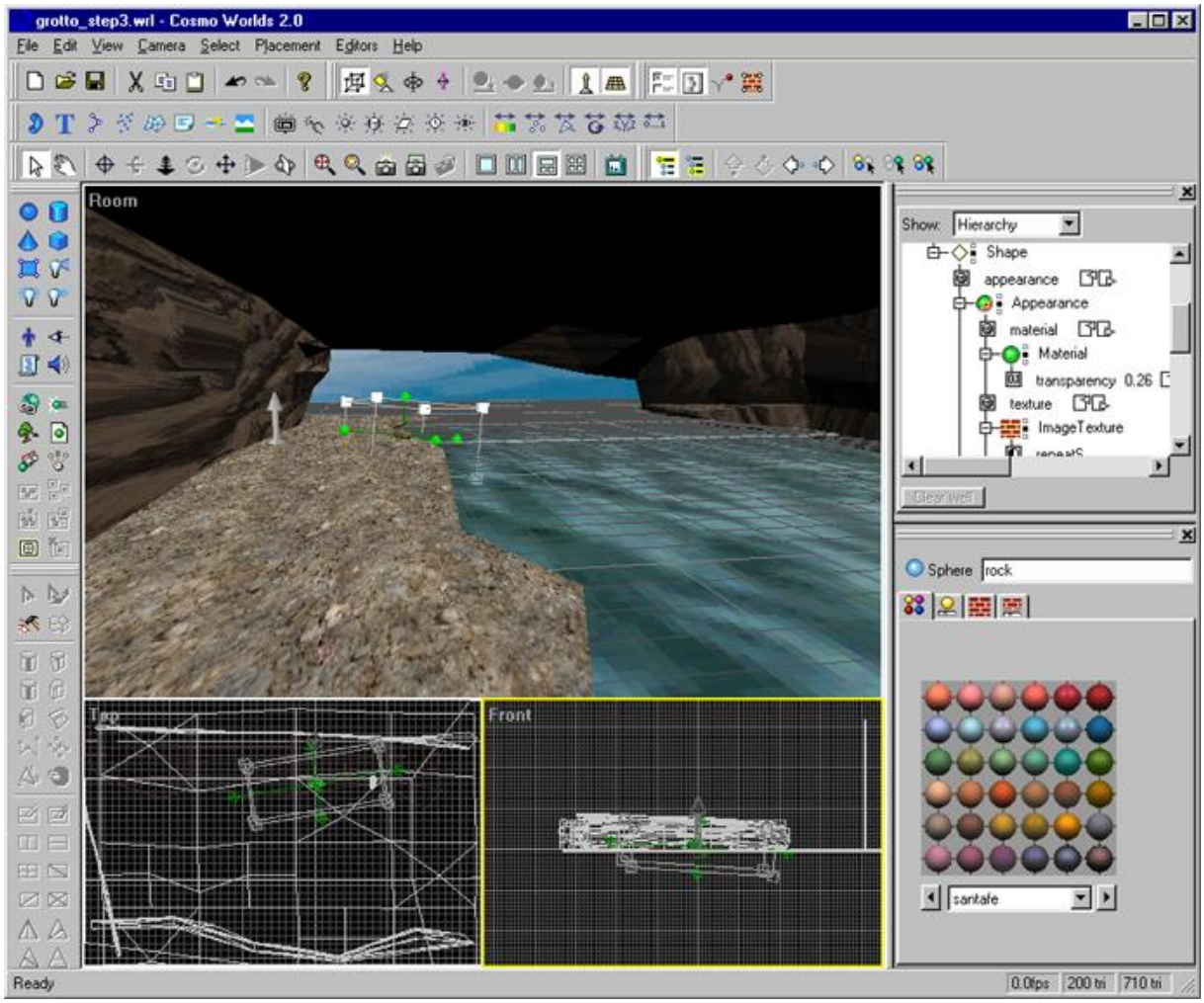


Figure 2.8: A screenshot of Cosmo Worlds. Multiple Views can be seen simultaneously(3D-View, Left, Front, Top). Many actions can be performed through the GUI of Cosmo Worlds. [Copyright 1998 Robert Rothfarb, used with permission.]

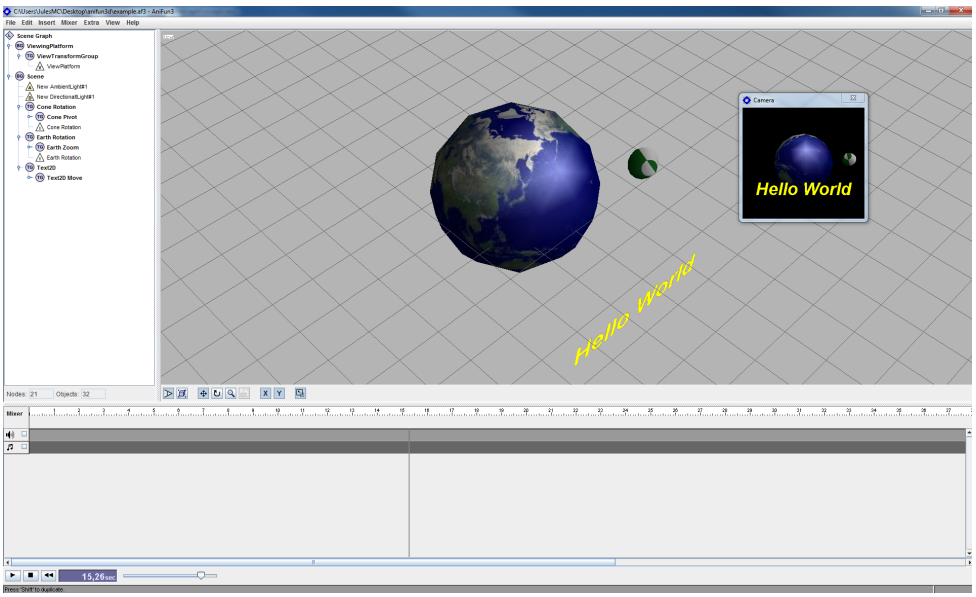


Figure 2.9: A screenshot showing AniFun3 Designer with a simple scene composed of a earth with cone rotating around the earth and some 2D text. The tree view containing the nodes is shown on the left. The main window shows a three-dimensional preview of the scene graph. The audio mixer is shown on the bottom.

2.9.7 AniFun3

AniFun3 is a scene graph tool which is built on top of Java3D (see section 2.7.5) [Faulhaber, 2006]. It allows the creation of simple scenes by using a tree list view. Node properties can be edited and changes are shown instantly in the live preview. It allows the import of several well known 3D models such as VRML, 3DS and DXF (see section 2.8). User written extensions to the Java3D library can be loaded during run-time. One feature not very common in other scene graph tools (expect Cosmo Worlds) is the integrated sound mixer which allows synchronous playback of sounds with defined animations. AniFun3 is freeware. Figure 2.9 shows a screenshot of AniFun3.

2.9.8 OSGEdit

OSGEdit is a GUI-based tool to create OpenSceneGraph (OSG) scene graphs. The scene graph structure is shown in a tree view with a live preview window reflecting the current scene graph all the time. All OSG supported file formats are enabled in OSGEdit as well. Transformations, rotations and scaling can be done visually in the preview. Fields of a node can be edited via the GUI as well. Figure 2.10 shows a screenshot of OSGEdit.

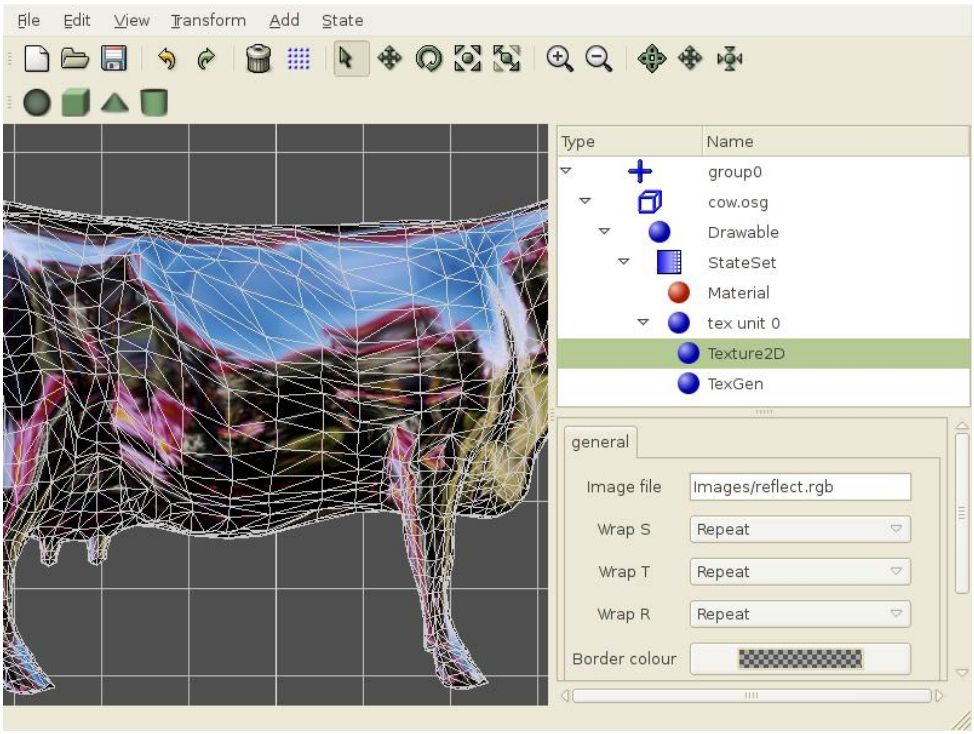


Figure 2.10: A screenshot showing OSGEdit. The scene graph structure is shown using a tree view on the right side. Fields can be edited below. The preview is shown in the main part of the window.
[Copyright Rubén López, used with permission]

2.10 Selected Software Design Patterns

This section gives a short introduction to a number of selected software design patterns. It is not the intention to give a full reference of all design patterns available, but rather to give a short summary of relevant patterns which are used in Studierstube Builder with a reference forwarding the interested reader to the relevant literature. The understanding of these patterns facilitates a fine grasp of the following chapters.

The book by Gamma et al. [1995] was the first collection of Design patterns and defined the entire field. Although other patterns have been found since that time and other books been written, it is still considered the one source for design patterns containing the most important patterns. Other books in the field are for example written by Fowler [2002] and Freeman et al. [2004].

> Design Patterns describe simple and elegant solutions to specific problems in object-oriented software design. Design patterns capture solutions which have developed and evolved over time. They reflect untold redesign and recoding as developers have struggled for greater reuse and flexibility in their software. Design patterns capture these solutions in a succinct and easily applied form. *(Gamma et al. [1995])*

2.10.1 The Observer Pattern

> Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically. *([Gamma et al., 1995])*

Whenever objects need to react to changes of other objects, the observer patterns stands by. The classic example is a spreadsheet with different charts. Whenever the data in the spreadsheet changes, this change has to be reflected by the charts [Gamma et al., 1995]. How this is done most elegantly is described in the observer pattern. The key idea is that objects that change - called subject - need to notify other interested objects - called observers of their change. The number of observers does not have to be fixed. It can be any number and vary during run-time. If a new chart is for example created, a new observer needs to be notified when the subject changes, increasing the number of observers by one. If a chart is deleted, this object does not need to be notified any more, decreasing the number of observers by one. That is why Observers need a way to announce and cancel their interest in the changes of a subject during run-time. This is usually done through a pair of public register/unregister functions in the subject. To propagate changes from the subject to the observer, usually all observers define an update function which is called by the subject if something changes.

The usual way this is implemented is to have an (abstract) interface for the subject and the observer defining register/unregister and update functions [Freeman et al., 2004]. The subject keeps all the observers in an internal data structure (for example a vector). Concrete classes just need to inherit the interface and override the functions with their own functionality. The changed data can be passed around either using push or pop, depending on the implementation. Push means the subject sends the changed information to all the observers (for example as a parameter in the update method), no matter if they are interested at the entire information or not. Pop on the other hand means that observers are just notified of change but without passing the data data has changed to all objects. Each observer can then decide which information it is interested in requesting just that information (by calling for example a getter-method on the subject). The Observer pattern is used to notify the view an item's value changes.

2.10.2 The Factory Pattern

> Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory method lets a class defer instantiation to subclasses. *([Gamma et al., 1995])*

A factory is a class which is used to instantiate objects during run-time[Freeman et al., 2004]. Often the type of the object is not known during compilation, as it may for example depend on user input or other conditions. Depending on the number of possible objects to instantiate and the conditions which have to be differentiated, the selection of the correct object to create can be pretty complex. The same decision which object to create has to be made at multiple places throughout the code. When new objects are added, the instantiation decision has to be expanded, often at different locations. The factory pattern was created to solve these problems. A factory is a class which abstracts the object creation and hides it from other parts of the code. Instead of instantiating a new object directly in the code, one just calls the factory class with some parameter (very often a string or an integer, preferable defined as enums or constants in the factory class). The factory method then decides which object to instantiate based on the parameter received. The Factory method thus shifts object creation from multiple places in the code to one central place. This increases extendibility and maintainability. The Factory pattern is often used in combination with the Singleton pattern, described in the next section. The factory pattern is used during user interface prototyping and to chose the correct input widget for a field.

2.10.3 The Singleton Pattern

> Ensure a class only has one instance, and provide a global point of access to it. *([Gamma et al., 1995])*

Sometimes it is necessary to make sure only one instance of a class can exist. The classic example given by Gamma et al. [1995] is the printer spooler. Other examples include a window manager, or a file system. The Singleton takes care of this by providing a single point of access to a class. Depending on the language, different implementations exist. Singletons make use of class operations, meaning a static function in C++ [Gamma et al., 1995]. Usually singletons have only one public method (which is static), often called Instance or getInstance which returns a pointer or a reference to the concrete object. The objects constructor is usually hidden (declared as private or protected) from the outside. Singletons can make use of lazy instantiation, to avoid object creation as long as possible. The Singleton pattern is used for the Handler classes during user interface prototyping and to make sure hover events are sent in the correct order.

2.10.4 Model View Controller

Model View Controller (MVC) is not precisely a design pattern [Freeman et al., 2004]. It is more a collection of a few different design patterns put together. MVC is one of the most powerful patterns available. It enables complex behavior between multiple objects. It is very often used to split the work between a user interface (the view) that displays some complex information and a model which implements the complex behavior. User interactions are catched in the user interface and forwarded to the model which implements the appropriate reaction. Instead of directly programming the proper reaction of a user interface event into the view, a new class is created which sits between view and model: the controller. It is the controllers task to translate user interface events into calls which mean something to the model. See figure 2.11 for an overview of MVC. By letting the view focus only on the creation of event and display of information, loose coupling is reached. The direct integration of calls to the model into the view (instead of letting the controller do the work) on the other hand leads to a tight coupling between the model and the viewer and complicates the viewers code. Whenever the model changes, interested classes are notified. One can see that this will very likely use the Observer Pattern presented in section 2.10.1. By letting the viewer focus on how to display data, it is pretty clear that multiple viewers can be implemented for the same model. There are a few more improvements possible which make use of other design patterns such as the Adapter pattern which adapts an model to the needs of existing views and controllers. See the excellent description of MVC by Freeman et al. [2004] for more details. MVC

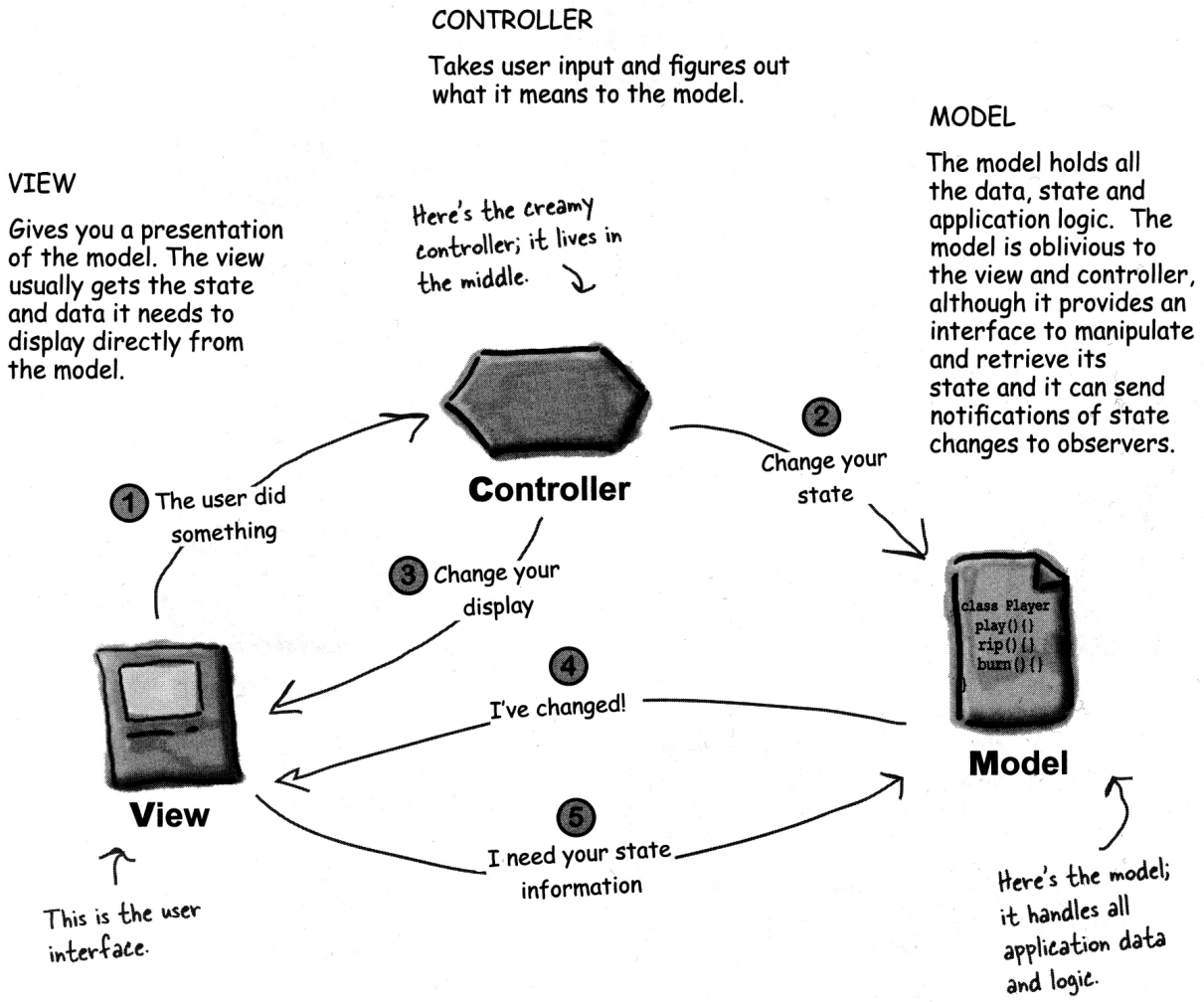


Figure 2.11: An overview of the MVC concept. The view is just responsible to display the information and forward events to the controller. The controller has to know what that information means to the model and call the respective methods of the model. The model may change the data or the state and notify all observers of the change.
[Freeman et al. [2004]. Copyright 2004 O'Reilly Media, Inc. All rights reserved. Used with permission]

is used by Qt within the model/view framework classes. They are used for the display of field values and for the scene object list.

2.11 Selected GUI Design Patterns

Software design patterns are only one field where patterns were discovered as a tool to improve the results achieved during software design and programming in general. Another field where these design patterns are useful is the design process of GUI's. Many books about GUI design have been written which offer advice and guidelines for prospective designers. Shneiderman and Plaisant [2009, page 88] for example list their rather general "Golden rules of interface design" as follows:

- Strive for consistency: Being consistent is important. This applies to the terminology, the order of actions performed by the user, color scheme used, and different menus to reach the actions.
- Cater to universal usability: Designing a user interface for different user groups at the same time often leads to better user interfaces for all users.

- Offer informative feedback: The system should return visual feedback for every user interaction performed.
- Design dialogs to yield closure: Dialogs with a user should have a beginning, middle and an end.
- Prevent errors: Design the system around the principle to catch as many user errors as possible before they occur. This can be done through input masks, hints how input should be formatted or clear steps the user can take to recover from the error.
- Permit easy reversal of actions: As many actions as possible should be reversible to allow the user to experiment with different features.
- Support internal locus of control: Give experienced users the ultimate control. Changes in the user interfaces or system behavior have to be made with the uttermost caution as this may annoy users.
- Reduce short-term memory load: The design should not force the user to remember any information from previous screens.

Tidwell [2005] take a more practical approach by building a book around established and well accepted design patterns for GUI design. Tidwell [2005] acts as the prime source for the following summary of GUI patterns used for the implementation of Studierstube Builder.

2.11.1 Center Stage

> Put the most important part of the UI into the largest subsection of the page or window; cluster secondary tools and content around it in smaller panels. *(Tidwell [2005])*

Many applications such as image editors, spreadsheets and many web sites use this pattern. As the center takes up the largest amount of space, it is the main focus of user attention on startup. Tidwell [2005] recommends a size at least twice as large as the other navigation panels, and a color different to the surrounding panels.

2.11.2 Moveable Panels

> Put different tools or sections of content onto separate panels, and let the user move them around to form a custom layout. *([Tidwell, 2005])*

Give the user control over the layout of the panels used. Each panel should have a self-evident meaning. As each user may use a different working style letting the user decide which layout to use may improve the efficiency. For some tasks a user might not need a certain panel and should for example be able to temporarily close it to save screen space.

2.11.3 Jump to Item

> When the user types the name of an item, jump straight to that item and select it .*([Tidwell, 2005])*

When using a long list of items, enable users to quickly jump to an item by typing the item's name. This prevents users from performing long searches and enables them to keep their hands on the keyboard. Rapidly typing a name selects the first matching item with this name, while repeatedly pressing the same button switches through all items starting with that button.

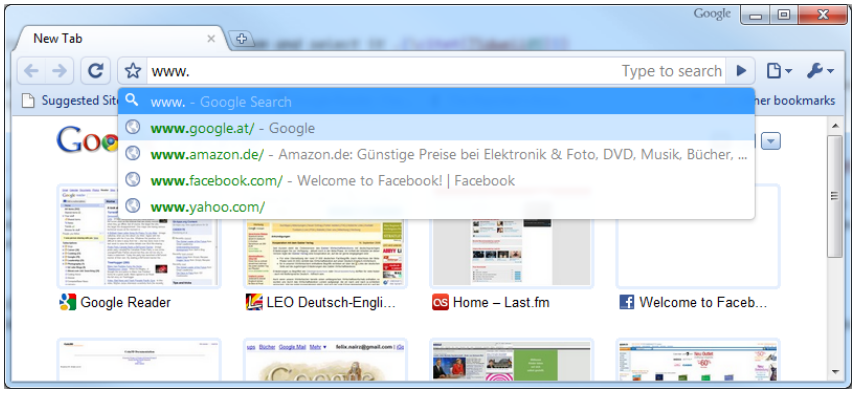


Figure 2.12: An example of the autocompletion pattern using Google Chrome. While the user types in the address bar, previously visited web pages are displayed in a pop up list

2.11.4 Autocompletion

> As the user types into a text field, anticipate the possible answers and automatically complete the entry when appropriate. *([Tidwell, 2005])*

While the users types, show an auto completion with appropriate entries. This might be a string stored previously (such as the URL address of a previously visited site) or a number of predefined options. This can speed up working speed as well as it can prevent errors for complicated or long names. Different options to display a preview exist. Usually a number of matching items are shown below the current place a user types. Alternatives are to show the preview only upon request as it is done in many console applications.

2.11.5 Property Sheet

> Use a two-column or form-style layout to show the user that an object's properties are edited on this page. *([Tidwell, 2005])*

Property sheets are well known elements in a variety or programs. Qt Designer for example uses one to modify the properties of a widget, most integrated programming environments have one. Property sheets can be used whenever the user needs to modify the properties of an object in a structured way. Property sheets often use a set of input widgets depending on the property type edited. Structuring the listed elements in a meaningful way is important as soon as the number of items increases. Alternatives include alphabetical, categorized or most common used options being order first [Tidwell, 2005]. Figure 2.13 shown an example usage of the property sheet pattern.

2.11.6 Preview

> Show users a preview or summary of what will happen when they perform an action. *([Tidwell, 2005])*

Showing the user a preview of the current action performed is desirable. This gives the user reassurance and enables the recognition of errors present so far. A good preview can accelerate the preview of the desired result.

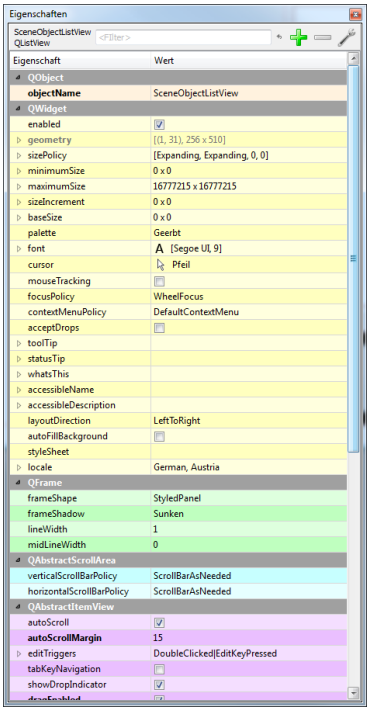


Figure 2.13: An example use of the property sheet pattern. The image shows the property editor using in Qt Designer to modify the properties of widgets. Properties are shown in the left column, while their values are shown on in the right column. Note the different input widgets used, depending on the type of the property.

2.12 Environmental Analysis

This section introduces existing programs which influenced the creation of Studierstube Builder. A strong focus is put on the supported functionality and the workflow these tools offer. The usability of these programs is discussed as well. Section 2.12.1 takes a detailed look at two aforementioned scene graph editors, namely XIP Builder and MeVisLab. Qt Designer, a tool for user interface prototyping and yEd, a graph editor are introduced in section 2.12.2.

2.12.1 Graphical Scene Graph Editors

Before the implementation started, a detailed analysis of existing scene graph tools was performed to see how others solved some of the issues faced. Special attention was put on the topic of usability and on the features offered by these tools. The analysis was limited to two tools which came closest to ideas presented in section 1.4. These tools are XIP Builder, as introduced in section 2.9.3, and MeVisLab, as introduced in section 2.9.4.

To analyze and compare the two products, the analysis has been split into different topics. The analysis has been done thoroughly, testing each of the tools for typical tasks a user will be interested in. It is not unlikely that at least some of the issues found will be improved in later versions of the tools. The test was performed with version 0.2.1 of XIP Builder, from the 22nd of May 2009. MeVisLab was tested with version 2.0, from the 9th of June 2009.

Generally speaking, both products have their advantages and disadvantages and there is no clear ”winner”. MeVisLab offers a much larger functionality and certainly is the more ”professional” product. It is obvious that the development of MeVisLab started before XIP Builder and that MeVisLab is a commercial product. XIP Builder on the other hand, feels a little more intuitive to operate. While both products allow a much faster creation of scene graphs than with coding or writing an Open Inventor file

(see section 2.8) some standard tasks like creating field connections are surprisingly counterintuitive. Both products have their weak areas. The following subsections summarize the results of the analysis while detailed results can be found in figure 2.14, 2.15, 2.16, 2.17, and 2.18.

XIP Builder Analysis Summary

In summary, the drag and drop interface of XIP Builder is intuitive to operate. Objects, connections between objects, and field connections can be created through drag and drop. However, connecting objects would be more intuitive if more visual feedback could be given to the user during creation of the connection. The creation of field connections is counterintuitive, although it can be done using drag and drop. It requires the user to open many small floating windows. Drags start and end at small icons which are hard to hit and whose purpose is initially unclear. The small floating windows unfortunately can not be moved outside of the main windows borders and quickly clutter the user interface. Packages on the other hand allow many nodes to be collected to single representation and thus help to avoid a cluttered user interface. Fields can be modified using some input widgets but the implementation is not as good as those of MeVisLab. A fields type is not displayed in the field editor and the information is only visible through a tooltip. Error control is performed but works only for parts of the types. Scene graphs can be exported to the Open Inventor file format and saved in a custom file format. The custom file format would benefit from some standardized structure (for example XML). Currently, the information is just saved as plain strings into Open Inventor comments. Custom Extensions can be loaded but not during run-time and only through editing some text files. This requires to consulting the manual as there is no integration in the GUI.

MeVisLab Analysis Summary

MeVisLab does not use as much drag and drop as XIP Builder. The user interface of MeVisLab does not feel as consistent as it could be. Some tasks can be done using drag and drop while others can only be done trough context menus. Objects on the work space, especially the docks used to connect objects, look pixelated which is a little bit irritating as the rest of the user interface does not show drawing artifacts. The framing of the GUI is very well done. A separate, dockable window is available for each task. Fields can be edited using a number of helpful widgets and especially the modification of multi fields is well done. Handling of erroneous input is a little error-prone but existend. Connecting objects is done better as it gives more visual feedback to the user. However, a major bug was found here as well. If the user cancels a current connection operation all previously made connections to and from this object are deleted. Extensions can be loaded during run-time, a feature not present in other editors so far. The creation of field connections does not really fit into the way other actions are performed. Once it was figured out how to create field connections, usage remains cumbersome but works reliable. A little sad is that no option to export the current scene graph into the Open Inventor file format is present. The user is therefore bound to MeVisLab as an editor. Writing an importer for the MeVisLab file format (MDL) is not a very attractive solution.

2.12.2 Other Relevant Tools

Although not directly related to scene graphs, there are a number of other tools worth mentioning which influenced the creation of Studierstube Builder. The first, Qt Designer, is a tool which allows quick prototyping of user interfaces. Qt Designer has a very well designed user interface. The second, yEd Graph Editor, is an editor to quickly create graph structures with a drag and drop interface. This section will briefly present these two tools, without the intention of presenting a complete feature list. The focus rather lies on how these tools support their users in doing their work as efficiently as possible.

Figure 2.14: First part of the environment analysis. Compared are two state of the art products: XIP Builder and MeVisLab. XIP Builder is shown in blue and MeVisLab is shown in yellow, while the topics discussed are shown in orange. The comparison criterion for this figure is how object creation is done and how field values can be modified

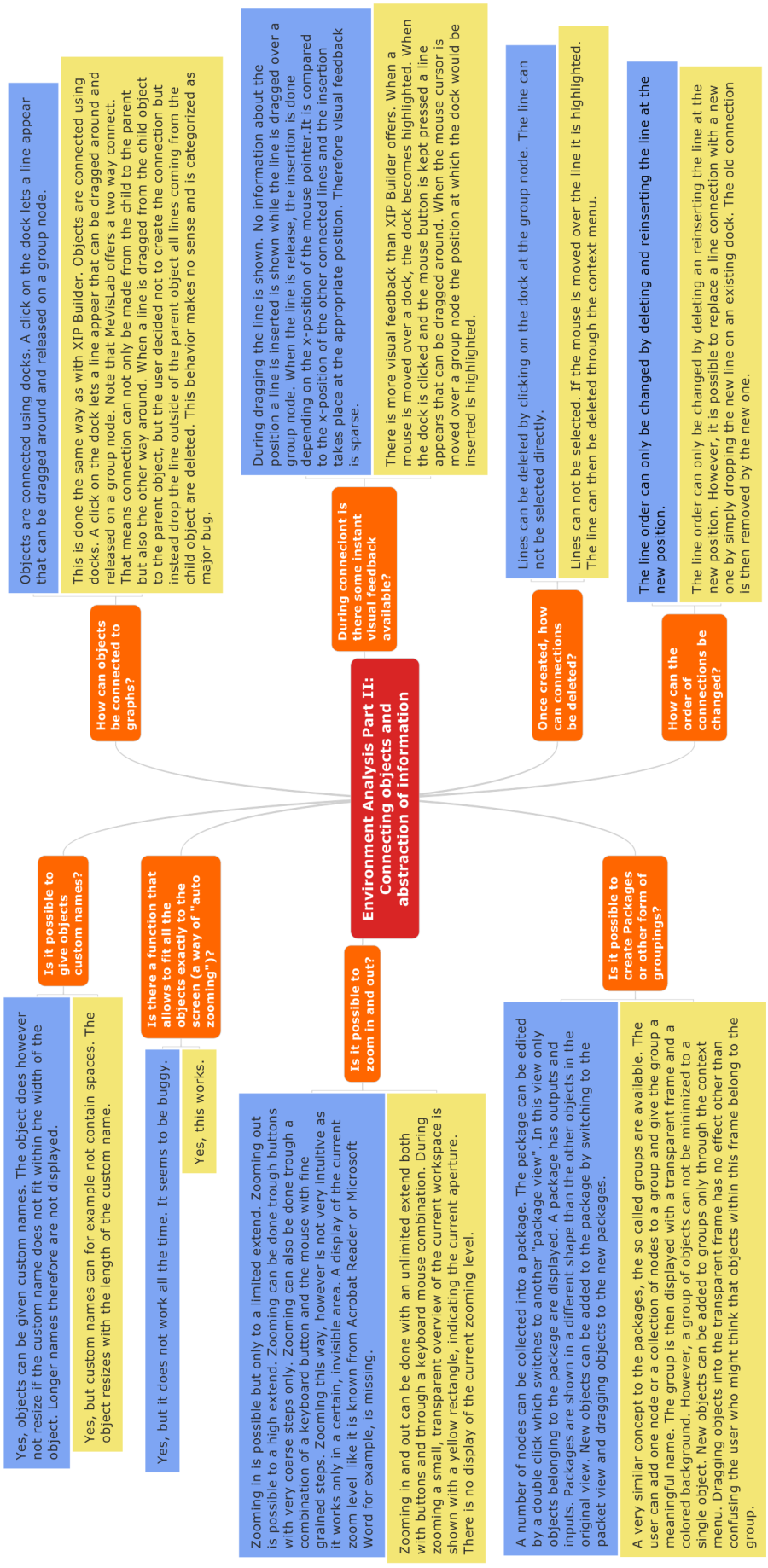


Figure 2.15: Second part of the environment analysis. Compared are two state of the art products: XIP Builder and MeVisLab. XIP Builder is shown in blue and MeVisLab is shown in yellow, while the topics discussed are shown in orange. The comparison criterion for this figure is how objects can be connected to graph structures and how information can be abstracted

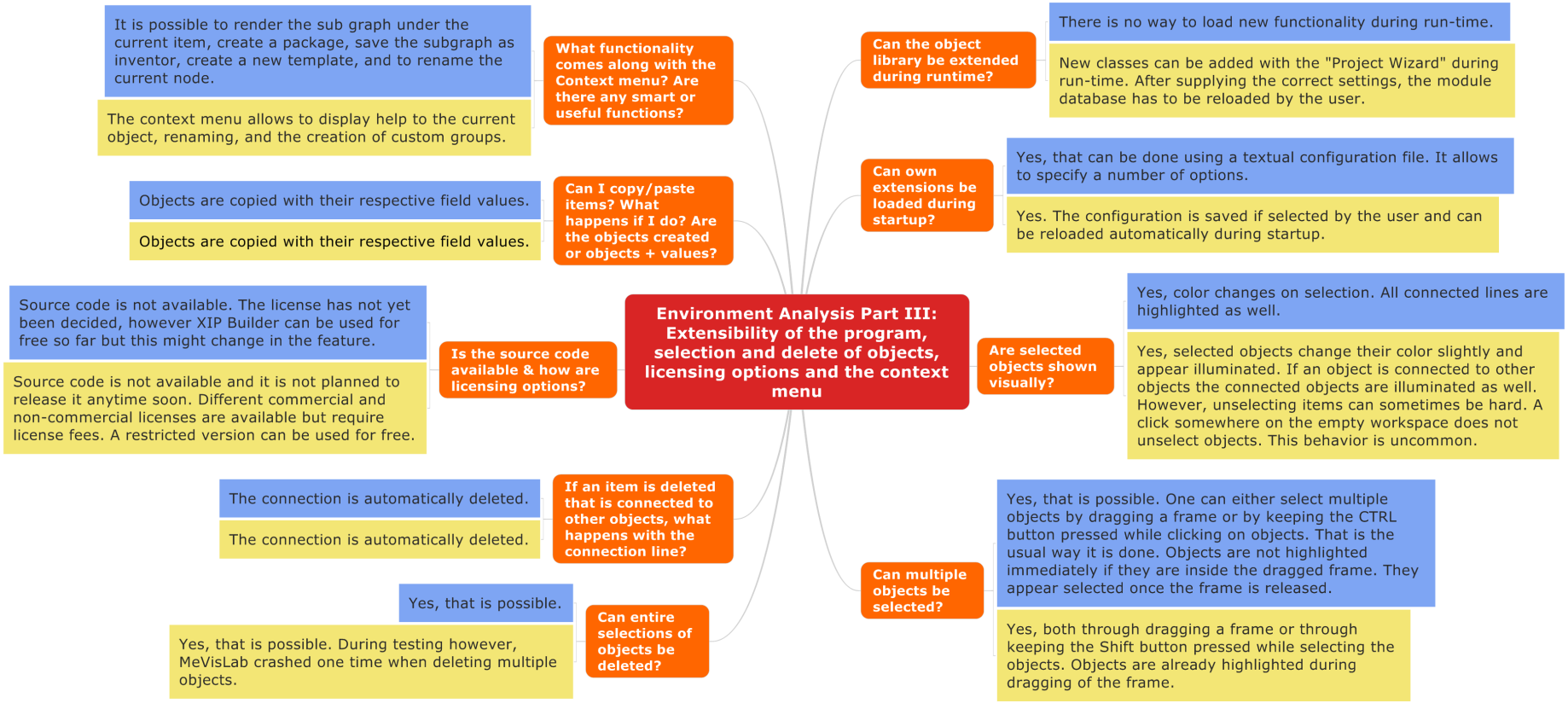


Figure 2.16: Third part of the environment analysis. Compared are two state of the art products: XIP Builder and MeVisLab. XIP Builder is shown in blue and MeVisLab is shown in yellow, while the topics discussed are shown in orange. The comparison criterion for this figure is if the scene graph library can be extended during run-time or during startup, how objects can be selected, which licenses are available, and what actions the context menu offers

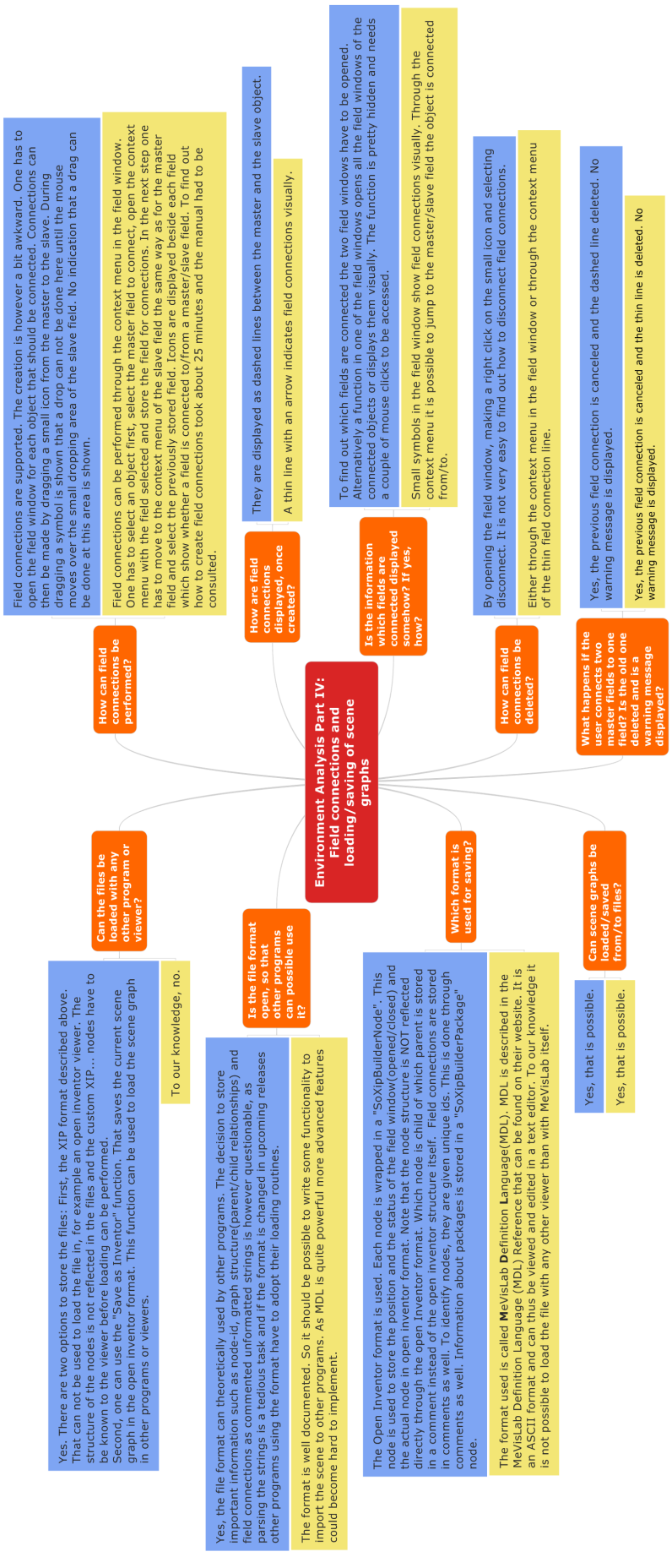


Figure 2.17: Fourth part of the environment analysis. Compared are two state of the art products: XIP Builder and MeVisLab. XIP Builder is shown in blue and MeVisLab is shown in yellow, while the topics discussed are shown in orange. The comparison criterion for this figure is how field connections are handled and how scene graphes can be stored and loaded

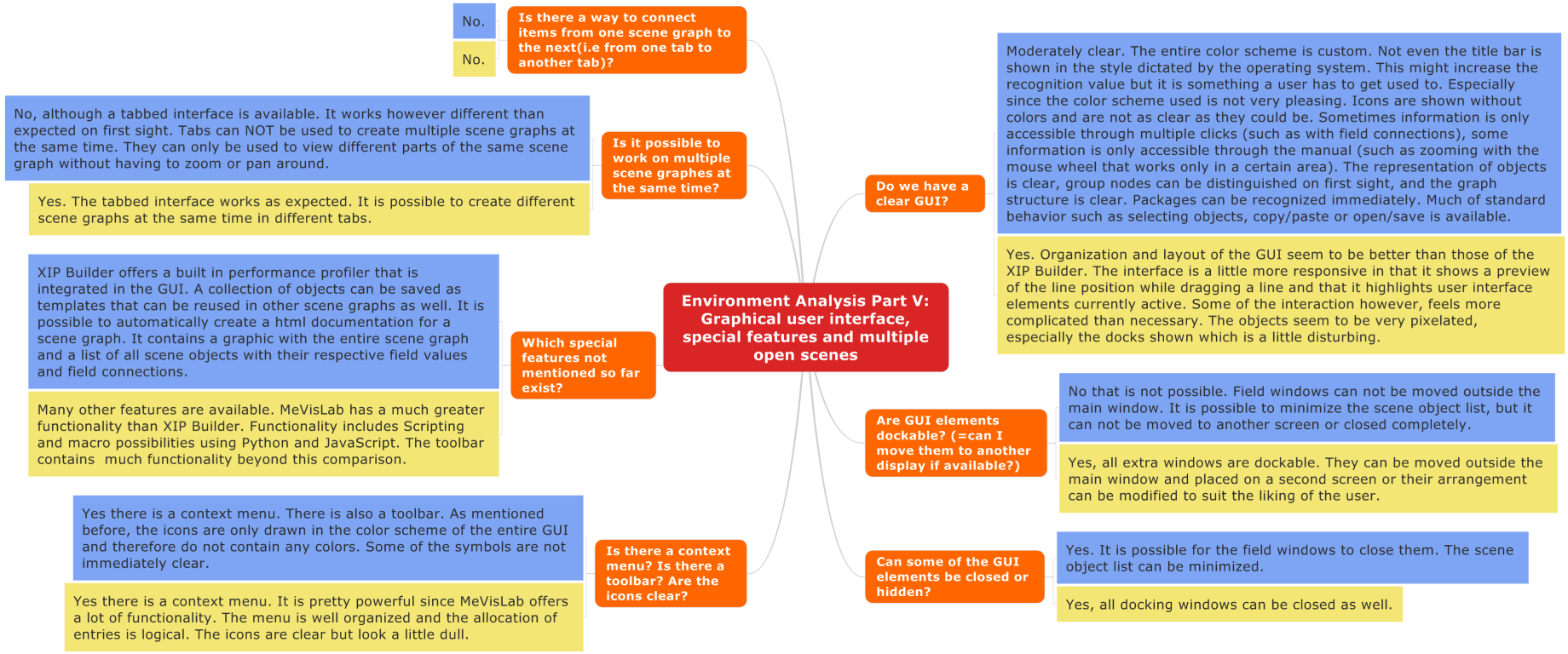


Figure 2.18: Fifth part of the environment analysis. Compared are two state of the art products: XIP Builder and MeVisLab. XIP Builder is shown in blue and MeVisLab is shown in yellow, while the topics discussed are shown in orange. The comparison criterion for this figure is how the user interface is designed, which special features exist and whether multiple scene graphs can be edited at the same time.

Qt Designer

One of the foremost products in the field of GUI design is Qt's GUI Designer [Nokia Corporation, 2009i]. It helps the user to rapidly create prototypes of new Qt-based user interfaces. The entire interface is designed to allow a what-you-see-is-what-you-get (WYSIWYG) experience. That means that during the entire user interface design process, everything looks exactly as it looks when it is released. A large part of the functionality which can be accessed trough Qt's API is available through graphical interfaces of Qt Designer as well. Many of Qt's user interface elements (called widgets) can be added to the currently processed GUI by simple drag and drop operations. The positioning of the widgets can be done manually or automatically by using one out of many available layout classes. While dragging an object, a preview of the object is shown under the mouse cursor. When dragging a widget into an area managed by a layout, this area becomes highlighted. During the dragging inside of a layout the current position at which the widget would be inserted into the layout is displayed. The preview of the position is of course updated as the user moves the widget around. The repositioning of widgets that were previously placed somewhere on the GUI can be done with a simple drag and drop operation as well, with the same kind of preview being shown while dragging as was mentioned before. The properties of each widget can be edited via the property editor in Qt Designer. A click on a widget loads all the properties into a list. Depending on the type of property (integer, boolean, decimal number, ...), different input widgets are used to change the properties values. In addition to the WYSIWYG preview of the GUI, the so called object inspector displays all the available widgets and layouts in a tree structured preview which makes it easier for the user to identify the logical object structure in a crowded GUI.

A few other functions of Qt Designer alleviate the process of GUI construction. Qt offers a so called Signal/Slot mechanism [Bögeholz, 2009] to pass changes from one object to other interested objects. That is how for example a dialog window can be popped up if a button is pressed. Qt Designer enables the creation of Signal/Slot connections through a graphical interface. When the user activates the Signal/Slot mode, the object under the mouse cursor is highlighted. The user can start a connection by dragging a line from the sender of the signal to the receiver holding a slot. While dragging, a connection line is displayed and the widget under the current mouse position (receiver) is highlighted as well if there is one.

Qt enables the creation of so called Actions. Elements in a context menu (for example File/Save) or a toolbar are examples of Actions. Qt Designer offers an easy way to create actions, allowing the user to specify text, icons, tooltips and keyboard shortcuts. Different icons for example can be defined for a number of different situations such as selecting an action, moving the mouse over an icon, disabling an icon and more. After defining an Action, they can be inserted into the GUI by a simple drag and drop operation. While dragging Qt Designer again highlights the current dropping area.

To bridge the gap between programming an application and designing the application GUI Qt allows different ways to integrate the GUI into application code [Bögeholz, 2009]. Depending on the requirements and the flexibility needed by the application programmer one can chose different approaches. First, it is possible to load a user interface during runtime. Second, Qt can automatically create C++ code which can easily be integrated (for example by inheritance) into the own project. Problems can be found during compile time. Changes to the GUI on the other hand require a recompilation of the source code.

The mentioned features above such as highlighting of widgets during many tasks, drag and drop support for almost all operations, tight or loose integration of the created GUI into the rest of the application, WYSIWYG support, access of the most properties through the GUI, and the graphical creation of Signal/Slot connections make Qt Designer an excellent and intuitive rapid application development tool. Figure 2.19 shows a screenshot of Qt Designer.

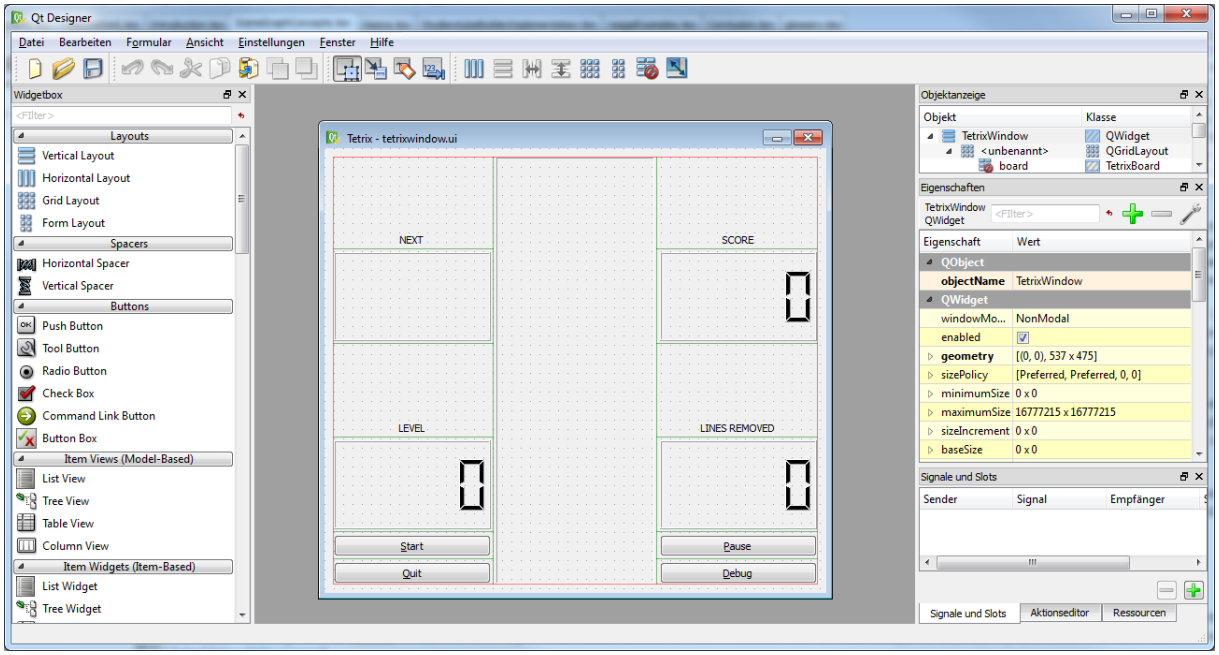


Figure 2.19: A screenshot showing Qt Designer. All available widgets are listed on the left in a sorted way. The GUI that is currently created is shown in the middle. The image shows the user interface for a Qt-based Tetris example. The right hand side houses a hierarchical view of all widgets, a property panel and a view which houses all Signal/Slot connections.

yEd Graph Editor

Another tool worth mentioning is the graph editor yEd. It can be used to quickly generate graph drawings trough a drag and drop user interface. Items can be created by dragging them onto a workspace and objects be connected by dragging lines between them. In addition to the items, a tree view shows the hierarchical structure of the graph. YEd can use different straight and bended connection lines and items can have different shapes and colors. YEd's most remarkable feature is the number of different automatic layout algorithms implemented. One can chose simple algorithms like tree or circle layout or more advanced algorithms which reduce the number of intersected lines. See figure 2.20 for a screenshot of yEd.

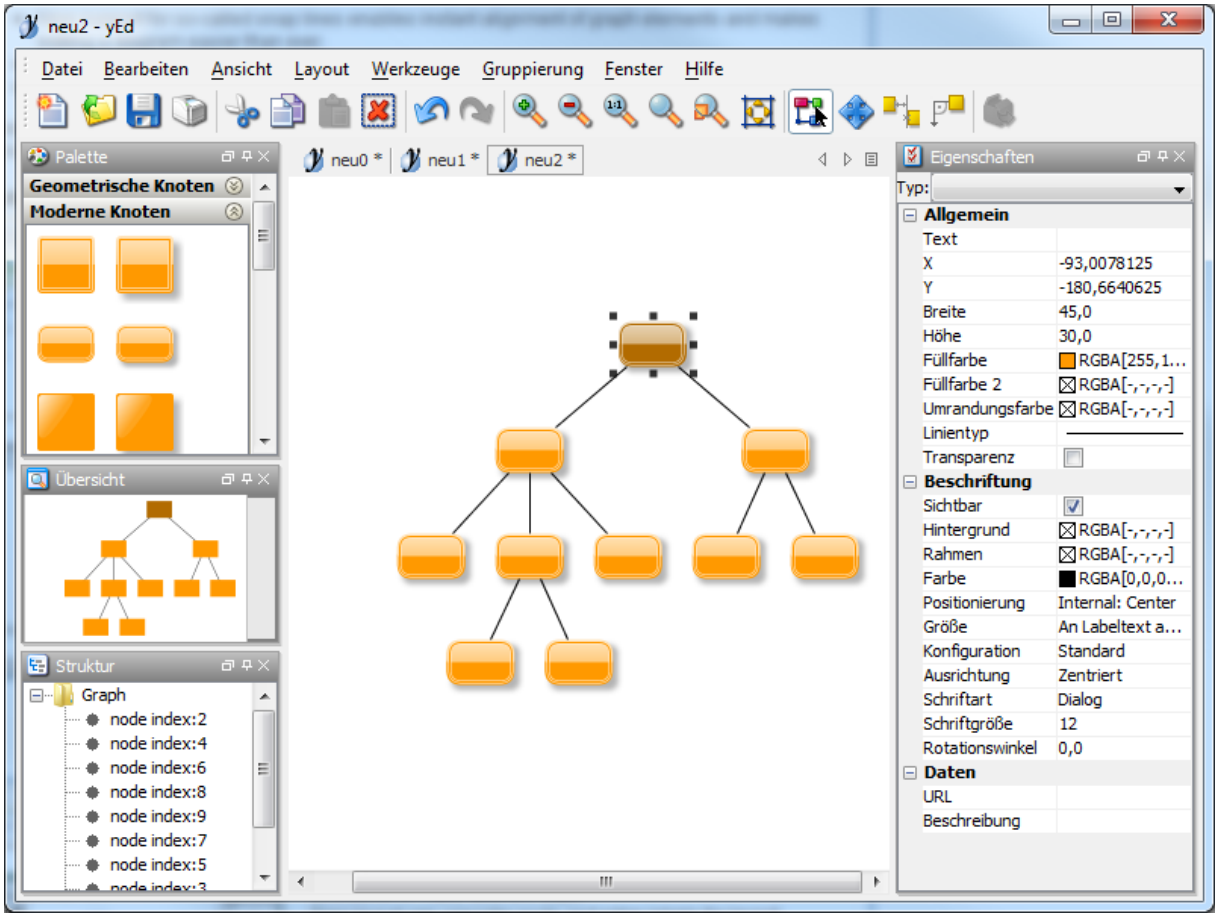


Figure 2.20: A screenshot showing yEd, a general graph editor. Objects can be created by dragging them on the workspace. The image shows a graph structure with a tree layout applied on it.

Chapter 3

Overview

The purpose of this chapter is to give an overview of the most important aspects of Studierstube Builder.

Studierstube Builder uses graphical objects to represent scene graph objects. The objects can be freely placed on a workspace and the arrangement can be easily changed during the work by just dragging the objects to a new position. This well-established approach is also used in XIP Builder, MeVisLab, and yEd (see section 2.9.3, 2.9.4, 2.12.2, and chapter 2.12). The workspace allows zooming to view different levels of detail. The graphical objects can be connected to DAG structures, creating a scene graph. Each connection between scene graph objects is represented by a line being drawn between the graphical objects. The order in which children are connected to parents is shown visually. During creation, a render root can be set which shows a preview of the current scene graph in a viewer. When selecting a scene graph object in the workspace, its field values are loaded into a field editor. The field editor uses the property sheet pattern (see section 2.11.5) to allow a modification of the values. Changes made in the field editor are immediately reflected in the preview.

The chapter is structured as follows: Section 3.1 introduces the parts of the user interface and the actions which can be performed on them. Moreover, section 3.2 gives an overview of the supported features. The tools used for the implementation are presented in section 3.3. Finally, the chapter closes with a an overview of the interrelationship between the introduced tools (section 3.4).

3.1 User Interface

One of Studierstube Builder's most important part is the GUI. Studierstube Builder should be as easy to learn and operate as possible. See figure 3.1 for an overview of the GUI and the definition of some common terms.

3.1.1 Workspace

The workspace is the center of Studierstube Builder. It is the main activity area and source of almost all interaction between user and the prototyping tool. The workspace acts as a dropping area where a user can drag items from the scene object list and drop them onto the workspace, thus, creating a new scene object with a graphical representation. The graphical representation displays the type of the object, a custom name if selected and the order of incoming connections if the type is a group node. Once the has user created a couple of scene objects, they can be connected to a DAG structure. Scene objects can be selected, moved, and deleted. During the entire interaction, instant visual feedback is given to the user, implementing the Preview patter (see section 2.11.6). A number of more advanced features, such as, exporting the current subgraph to the Open Inventor file format, deleting multiple connections at once, or giving a scene object a custom name can be chosen from the context menu of the workspace. On

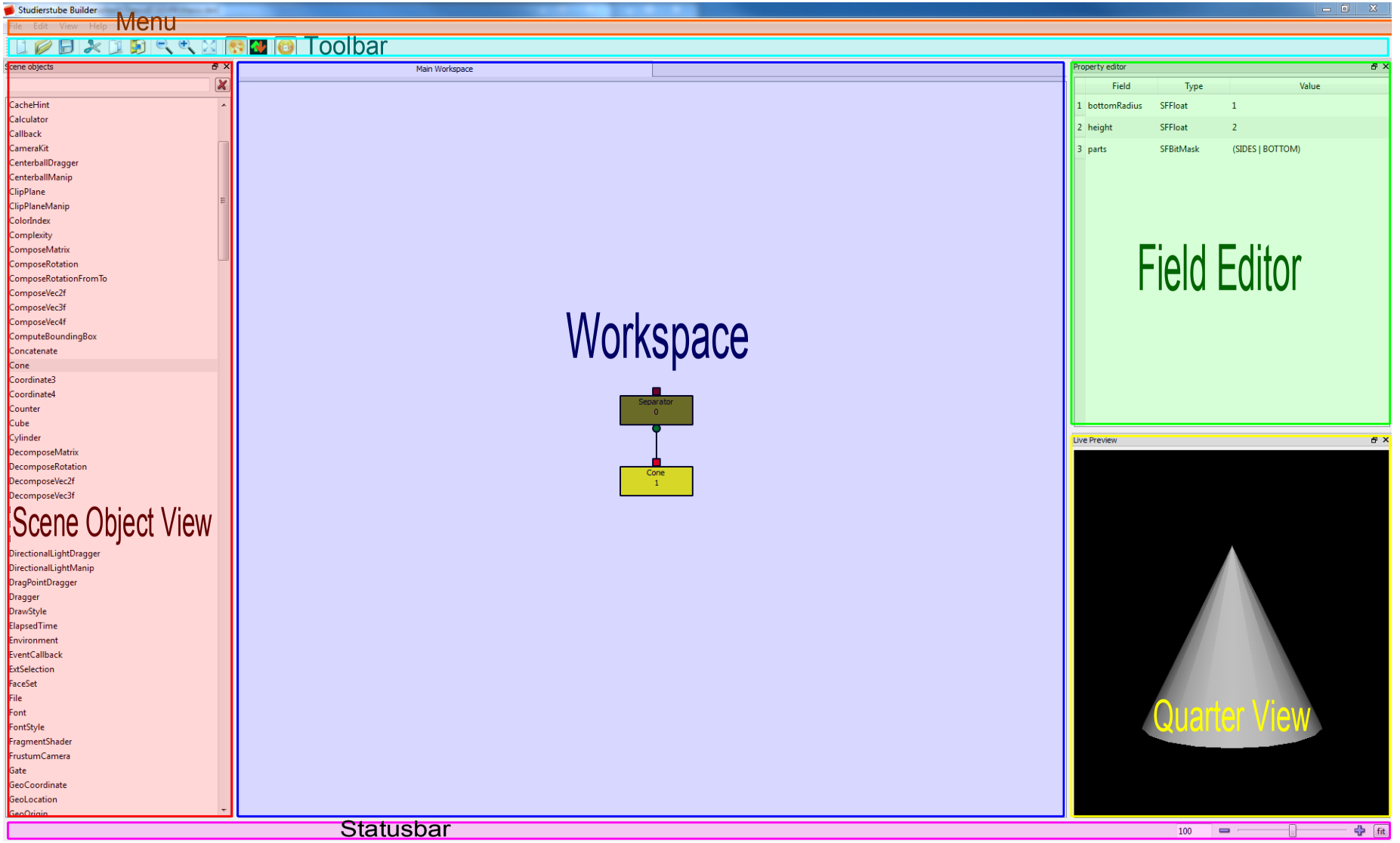


Figure 3.1: Overview over the GUI of Studierstube Builder. The scene object view can be used to select and create scene objects from a list. The workspace is the main area, where most of the interaction takes place and where the scene graph is created. The field editor can be used to modify the fields of a selected scene object. The quarter view is the place where the current scene graph is displayed or where subgraphs can be displayed. The menu, toolbar and status bar are standard GUI elements and complete the user interface. Note that all parts of the GUI except the workspace are dockable and can be closed temporary.

selection of a scene object, the objects field values are loaded into the Field Editor, where they can be modified.

3.1.2 Scene Object Viewer

The scene object viewer holds all available scene objects from the scene graph library. They are dynamically loaded during startup of the program after initializing the scene graph library. The user can select one or more of the objects from the scene object viewer and drag them onto the workspace. This leads to creation of new scene objects with the selected type. The scene object viewer itself is a dockable widget. It can be closed temporarily and the position can be changed to lie somewhere outside the main window.

3.1.3 Field Editor

After a scene object has been created, its field values can be modified inside the field editor. Once a scene object is selected by the user, it is queried to receive the name, type, and value of the fields and display them in the field editor. Depending on the type of the field, different input widgets are used. Decimal number and integers for example use a (double) spin box to edit their values. Most types however use a simple line to allow the user to change their field values. User inputs are checked before they are set as new values. If a typo or another error is found, an error message is displayed to the user. The erroneous input is stored, so that the user can correct the error instead of having to retype the entire string again. The field editor is dockable as well.

3.1.4 Menu, Toolbar, Context Menu and Status Bar

Some other standard GUI elements exist, as well. The menu can be used to select all available options from a number of pull-down menus. Standard actions such as open, save, save as, exit, and many other features can be selected through the menu. As usual, keyboard shortcuts can be used to open menus and perform actions associated with an entry.

The toolbar is located right below the menu and gives instant access to most frequently used entries. Distinct symbols are used for each action in the toolbar and a tooltip appears if the mouse is kept over an entry for a certain amount of time.

Different context menus appear, if the user right clicks a scene object (scene object context menu) or an empty space on the workspace (workspace context menu). The scene object context menu allows a number of scene object specific actions. One of the most important actions is the option to set the current object as the new root for rendering. Giving the current scene object a custom name is possible through the scene object context menu, as well. Features which allow to disconnect the current scene object from multiple other items can be selected. One can either delete all connection from a scene object, delete all connection from children (for group nodes) or delete all connections to parents. Subgraphs can be saved to the Open Inventor file format as well, using the current scene object as root for the subgraph to store. The workspace context menu allows standard file operations (”Open”, ”Save”, ”Save as”) for the current scene graph as well as adjusting the current zoom level through ”Zoom in” and ”Zoom out” buttons. Figure 3.2 shows the two context menus

The status bar contains another zoom widget which allows even more control. There are Zoom in and Zoom out buttons which perform zooming in steps. This is not done in fixed steps (for example 20% each time the zoom button is pressed). Instead a step rate depending on the current zoom level is chosen. In addition to the two zoom buttons, a slicer can be used to quickly adjust the current zooming level. The amount of zooming again depends on the current zoom level. To allow even more control, a line edit displays the exact current zooming level and supports fine grained changes. In addition to the zoom control, other information can be shown in the status bar.

(a) General context menu **(b)** Scene object context menu

Figure 3.2: A comparison of the general context menu which appears when the user clicks on an empty position on the workspace and the scene object context menu which is displayed if the user right clicks on a scene item

3.1.5 Preview

One essential part of Studierstube Builder is the live preview function. Once a render root has been set the preview shows the current scene graph. All changes made to the scene graph below the render root are immediately reflected in the preview. As the field editor and the scene object viewer, the preview uses a dockable widget which can be freely moved and resized to meet the users requirements. The viewer can use different options for rendering. Example settings include wireframe, point, bounding box, different stereo modes, and transparency settings.

3.2 Supported Functionality

Studierstube Builder offers an extensive set of features, some of which are not found elsewhere. This section gives a short overview of all the features present. More details on each feature and how the implementation is done is presented throughout chapter 4. See figure 3.3 for an overview of the supported functionality.

The most important feature is certainly the representation of scene graph objects through graphical objects. Each scene graph node is represented by a rectangular graphical object. The rectangle holds information about the type and an optional custom name of the object. All objects, which can be added to a group node, have an additional dock on top of the rectangle which can be used to connect it to other objects. Connections between objects are drawn as lines. Field connections are drawn as lines as well, but with a different drawing style, which is clearly distinguishable from normal connections. Graphical objects enable standard behavior such as selection, rearrangement, and deletion of one or more objects. The user can thus arrange all graphical objects as desired. The workspace can also be viewed under different zoom levels, allowing to swiftly switch between different levels of detail.

Most of the user actions are performed through drag and drop. The user can create scene graph objects by selecting one or more entries from the scene object view and drag them onto the workspace where a graphical object is created. Once a number of objects are created, they can be connected to DAG structures by dragging a line from the dock of the child object to the parent object. Before releasing the dragged line over the parent object, a preview of the position at which the new line will be inserted is shown. This is important, as the order affects the traversal sequence and thus the rendering result. Field connection lines can be created in an almost similar manner. A separate mode, called field connection mode is introduced. Once activated by the user, a field connection line can be dragged from source (master) to the target (slave). Once the field connection line is released, a dialog allows the user to select

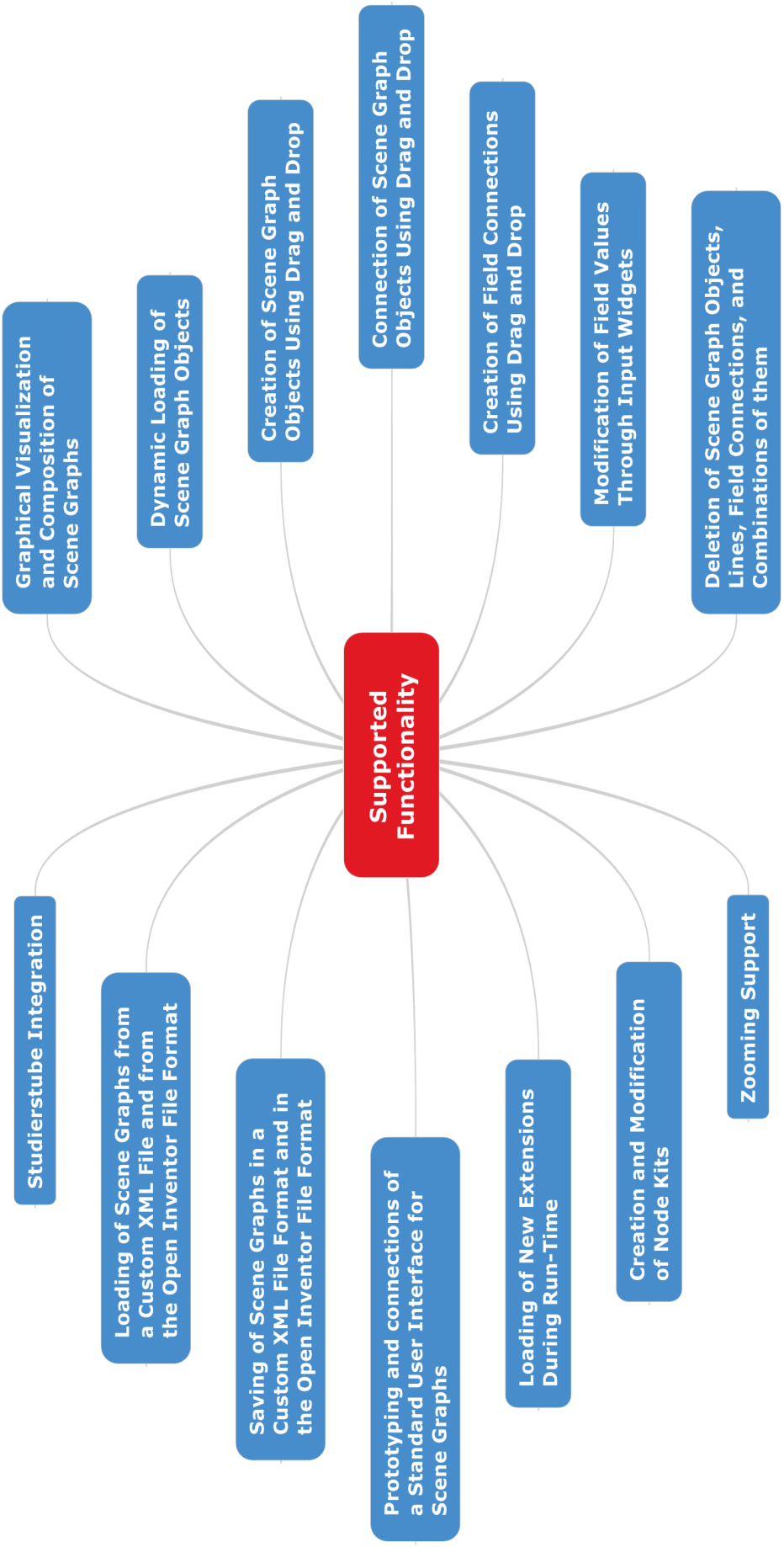


Figure 3.3: Overview of the functionality offered by Studierstube Builder.

compatible fields which should be connected.

Selecting a graphical object loads the object's fields into the field editor. The field editor displays name, type and value of each field in a tabular manner. A field can be modified by a double click on the value. This loads an input widget with the current fields value set. Depending on the type of the input, different widgets are used to enable the easiest modification of the value. The user can confirm the change by pressing the enter key. However, the user input is checked for errors before any changes are made to the scene graph. Error checking is done by making use of regular expressions. This avoids partly accepted strings, as was discussed during chapter 2.12.

Node kits can be created and modified as well. When a node kit is created, only the default elements from the catalogue are initialized (see section 2.3.7 for more details on node kits). A double click on the graphical scene object opens a new tab with an initially empty workspace. All node kit children are instantiated and a graphical scene object is created for each member of the node kit. A layout algorithm automatically arranges all node kit members. Each element can be edited as usual, through selection and modification of field values. However, new objects cannot be created, and existing object cannot be deleted in this new workspace.

As many users will want to expand their scene graph library with custom scene graph objects, Studierstube Builder is extendible. User written extensions can be loaded during run-time, once the user has created a Dynamic Link Library (DLL). All new scene graph objects are dynamically registered in the scene graph library. After registration, the scene object view is updated and the viewer is reloaded. The user can also specify to automatically load a number of DLLs during startup of Studierstube Builder. All changes made to one of the DLL files are registered within the scene graph library during startup of Studierstube Builder. The scene object view also automatically reflects the current version, as it extracts all information dynamically from the scene graph library.

A custom scene graph object allows to connect a user interface created with Qt Designer (see section 2.12.2) to a scene graph. The user interface is then loaded during run-time. During loading, all widgets are processed and a scene graph field with a compatible type is created. The newly created field is automatically updated each time the widget is modified by the user. The user can connect these custom fields to other scene graph object's fields, making use of the automatic update mechanism. User interfaces loaded this way can also be modified during run-time as changes are automatically detected. All existing field connections are then saved, the user interface is reloaded, and previously made field connections are restored. If a connected widget was deleted, the field connection is dropped.

Scene graphs can be saved in the Open Inventor file format and in a custom file format. The custom file format uses XML syntax to store the necessary information. The custom file format has the advantage that the user defined layout is preserved and that objects which are not children of the scene graph are stored as well. Thus, the custom file format allows some features not present in the Open Inventor file format. Studierstube Builder also allows reuse of existing content by supplying an Open Inventor file format importer. The importer extracts all required information directly from the imported scene graph and creates the graphical scene objects, connection lines, and field connection lines. A layout algorithm ensures that none of the graphical scene objects overlap and that the result is visually pleasing.

Finally, Studierstube Builder can directly manipulate Studierstube scene graphs. The Studierstube framework is started in a separate thread and configured as usual. The user can then create scene graphs which can be set as the new content of Studierstube.

After introducing the user interface and the functionality of Studierstube Builder, figure 3.4 gives an overview how the user can interact with the tool. Figure 3.5 and 3.6 go into more detail, showing how the functionality described in section 3.2 is handled in the user interface.

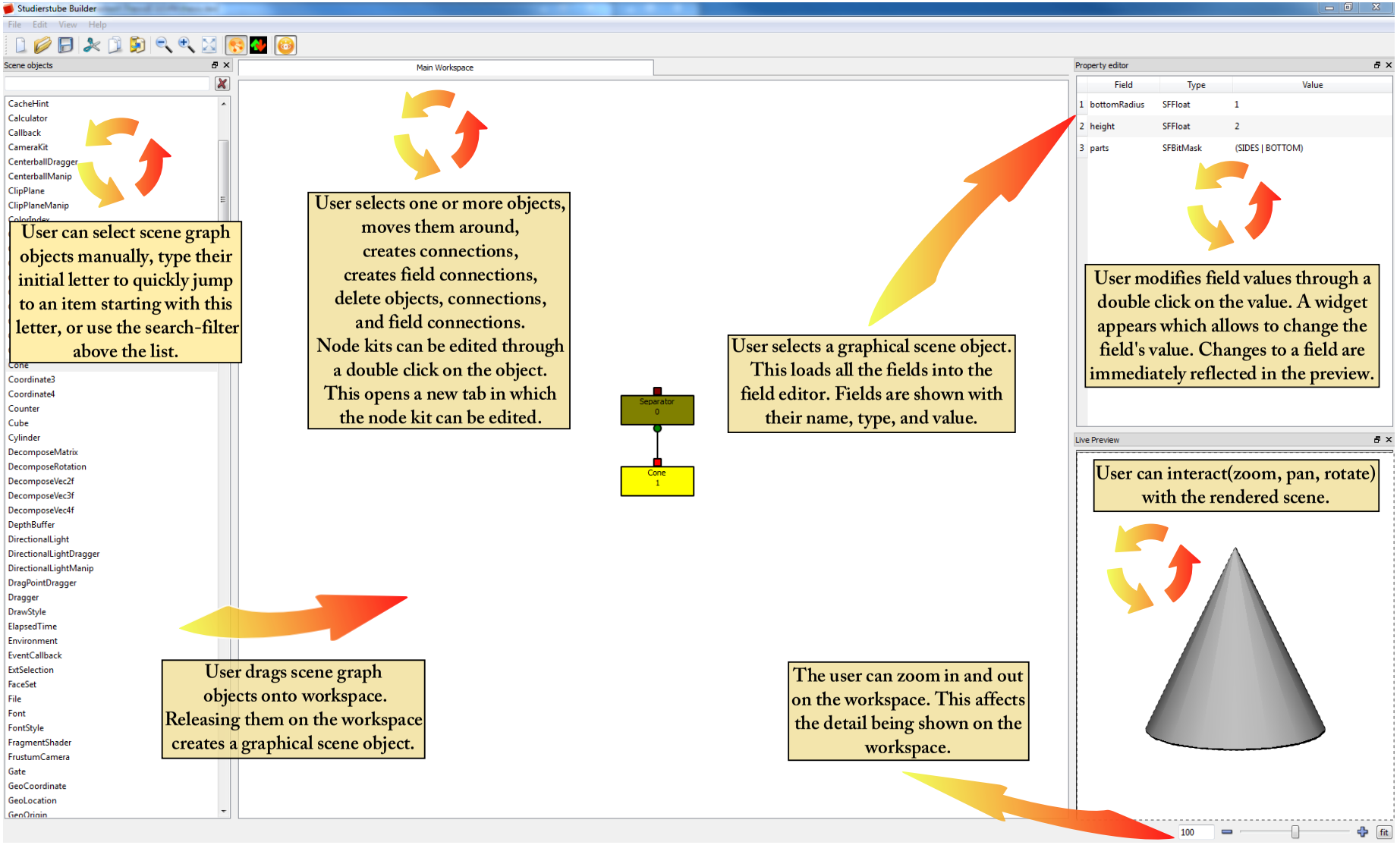


Figure 3.4: Overview of the interaction between the user and Studierstube Builder. Circular arrows indicate actions which are performed within one user interface element. Bended arrows indicate actions which are performed within one part of the interface but which affect other parts as well.

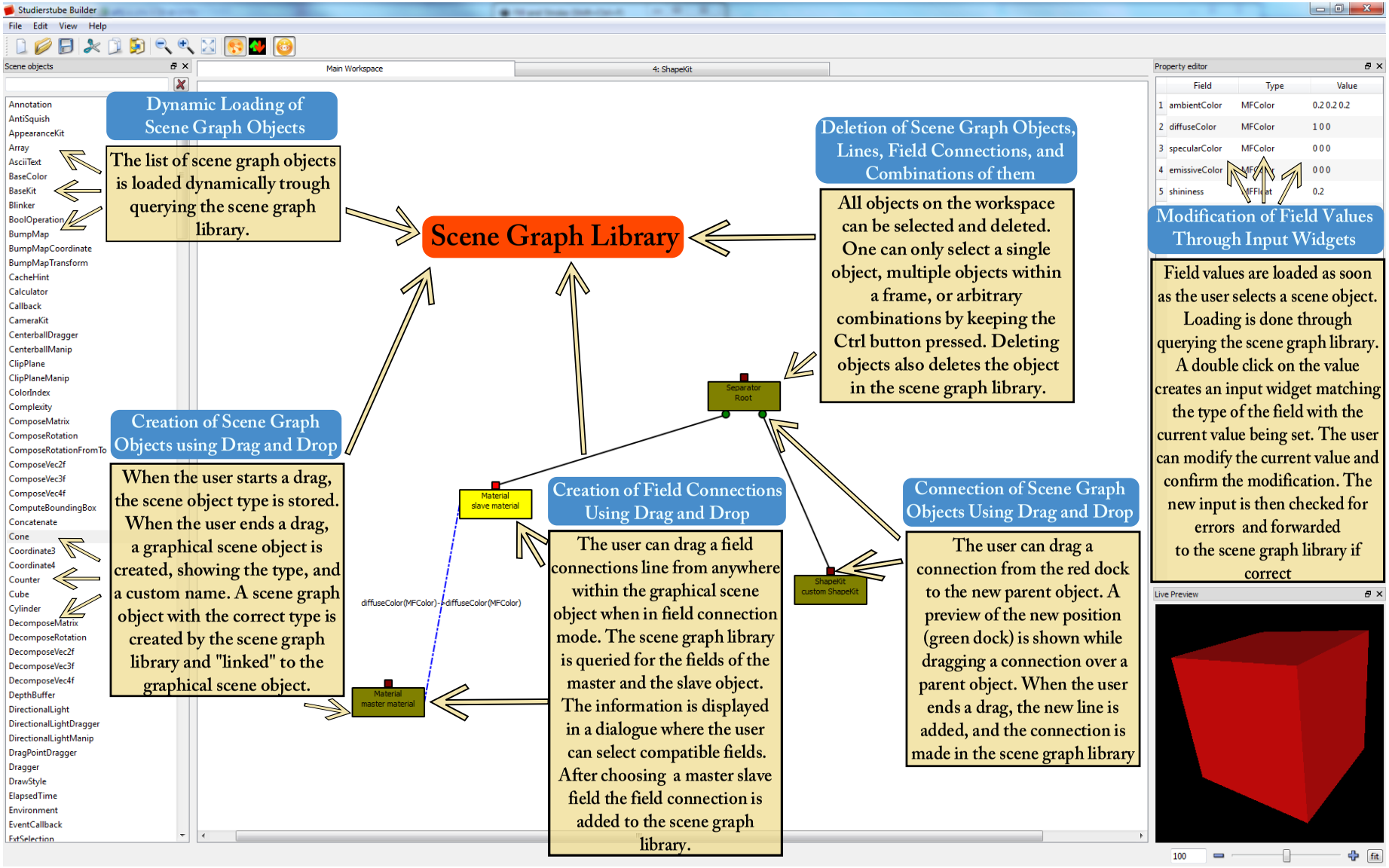


Figure 3.5: The first half of features presented in figure 3.3 into more detail and with a focus on the user interface and how the different user interface elements relate to each other.

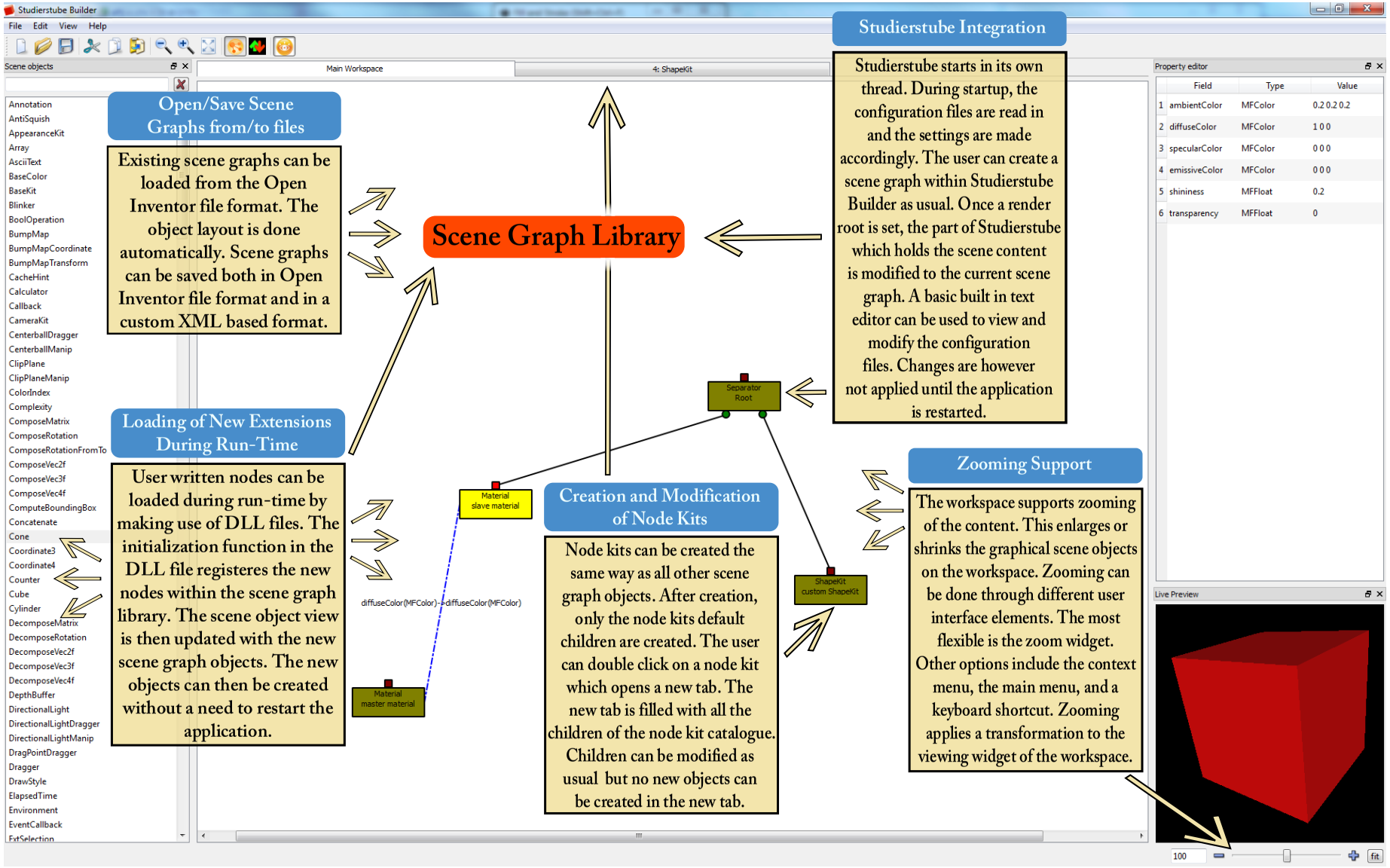


Figure 3.6: The second half of features presented in figure 3.3 into more detail and with a focus on the user interface and how the different user interface elements relate to each other.

The following list shows all available program operations the user can perform.

- Create new scene objects through drag and drop.
- Connect scene objects through drag and drop.
- Create field connections through drag and drop.
- Search for objects in the scene object list by name.
- Create a new (empty) workspace.
- Open an existing Open Inventor file.
- Open an existing Studierstube Builder file.
- Save the current scene graph to Open Inventor file format.
- Save the current workspace to Studierstube Builder file format.
- Dynamically expand the scene object list with custom scene objects from a dll file during run-time.
- Reload all scene objects from the scene graph library.
- Zoom in and out, using different widgets.
- Switch between Field Connection Mode and Scene Object Mode.
- Open/close the scene object widget.
- Open/close the field editor widget.
- Open/close the live preview widget.
- Show/hide all field connection lines.
- Set any scene object as render root.
- Give any scene object a custom name.
- Disconnect all parents/children from a group node.
- Disconnect all children from a group node.
- Disconnect all parents from a group node.
- Delete single scene objects, lines, and field connection lines or any combination of them.
- Select one or more scene objects.
- Reposition scene objects on the workspace.
- Edit node kits in separate tabs.
- Change field values using different widgets.
- View the rendered scene, rotate, pan, and zoom.

3.3 Selection of Appropriate Tools

3.3.1 Scene Graph Library

Most work at the ICG is done using the Open Inventor scene graph library. More precisely, most implementations are done using Coin3D (see section 2.7.4). A scene graph tool-based on another scene graph library than Open Inventor could therefore not be used for much work done at the ICG. Coin3D is offering a number of advantages. First of all, it is still under active development and it includes features like programmable shader support which is not part of the Open Inventor library. Second, the full compatibility to Open Inventor enables old code written for Open Inventor, to run with Coin3D as well. Third, the documentation is pretty good and licensing is flexible enough for the desired goals. Non commercial applications can use the Coin3D library for free. Furthermore Coin3D fulfills another important requirement: platform independence. Moreover, a number of additional libraries are available such as volume rendering and, more important, a number of viewers are supplied as well. Finally, by making use of Coin3D all existing extensions and examples can be used continuously. While more advanced or faster scene graph libraries, such as OpenGL Performer (section 2.7.2) or NVSG (section 2.7.6) exist, the performance and functionality offered by Coin3D is sufficient.

3.3.2 Programming Language

Coin3D is written in C++ (see for example Stroustrup [2000]) but language bindings for several other languages such as Java, Javascript and Python[Fahmy, 2006] are available as well. C++ is one of the most used programming languages, especially in graphics development. The two most used graphic libraries (OpenGL and DirectX) offer APIs for C++. Many different compilers and programming environments for all types of platforms exist and countless books have been written which cover all areas of the language.

Java offers many of the advantages of C++ and is platform independent. This independency is reached through a translation of the source code into byte code. The byte code is then executed by a the Java Virtual Machine running on the respective platform [Lindholm and Yellin, 1999]. Thus the source code only needs to be compiled once and can run on multiple platforms. C++ in comparison is directly compiled to executable source code and needs to be compiled on each platform used. Java also comes along with an extensive API. See Flanagan et al. [1999] for more details. There has been quite some discussion on the performance of Java compared to C++. While it is generally acknowledged that Java performed significantly slower when it was first introduced, the performance largely improved since then. Reinholtz [2000] for example claims that Java will outperform C++ because of the dynamic compilation process which can perform a number of optimizations not available to static linked programmes. The construction of objective benchmarks is a difficult task. Bruski [2008] for example showed that some modifications to the C++ code changed the benchmark result from Java being faster than C++ to C++ being two to three times faster than Java. In practice performance is to a much larger extend influenced by the software architecture, used data structures, algorithms employed and requirements set by the customer (for example thread safety). The knowledge and foresight of the development team thus has a much greater effect than the pure selection of a programming language.

The choice of a suitable programming language for a scene graph tool therefore is a personal decision depending on the language bindings available for the scene graph library, knowledge of the API of the respective programming language, tools used for development, the operating system used and much more. For the reasons stated, this work was done using C++ and making great use of the Qt framework which is described in the upcoming section.

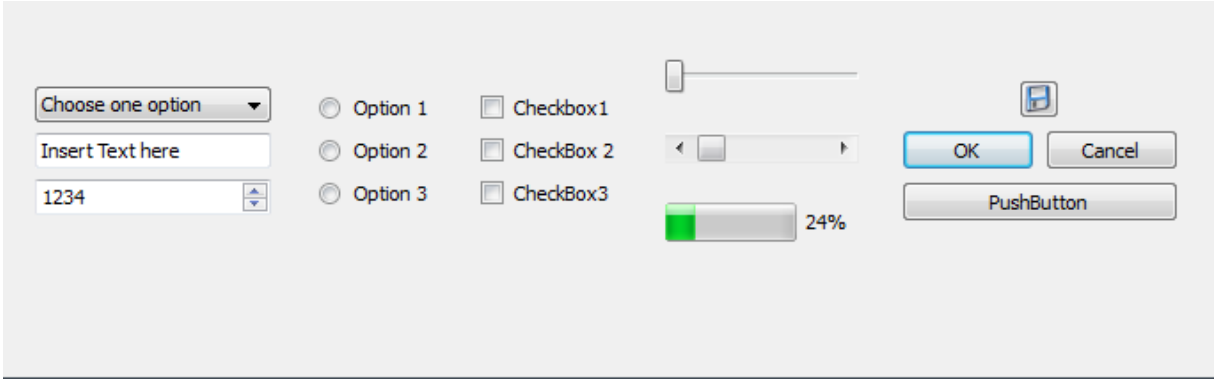


Figure 3.7: A selection of different standard widgets offered by Qt. Shown are some input widgets, dragging widgets and buttons.

3.3.3 Qt

> Qt is a cross-platform application and UI framework. It includes a cross-platform class library, integrated development tools and a cross-platform IDE. Using Qt, you can write applications once and deploy them across many desktop and embedded operating systems without rewriting the source code. *[Nokia Corporation, 2009h]*

Qt is built using C++ and can be run on many platforms such as Windows, Mac OS X, Linux, WinCE and even on smartphones such as the Nokia S60. The advantage of Qt is the vast set of tools and libraries which can be used. In contrast to Java, C++ does not offer an extensive API. It only includes a number of template classes in its Standard Template Library (STL) (see for example [Musser and Saini, 1995]). The STL consists of data structures, String handling classes and algorithms but the extend is generally pretty limited. Qt fills this gap by providing an extensive and well documented API for a wide range of applications. Qt is a cornerstone which enables the development of a platform independent prototyping tool. Qt supports the building of advanced GUIs by providing an extensive set of so called widgets. A widget is the basic building block of a user interface. Figure 3.7 shows a selection of input widgets, draggers and buttons offered by Qt. Qt has built in support for operations such as drag and drop, different types of lists, tables, multithreading, and a vast number of container classes. Furthermore, it has facilities which allow to efficiently paint a large number of custom objects on a workspace. Qt offers different licensing schemes. For software which is developed under the GPL or LGPL, Qt can be used without licensing fees [Nokia Corporation, 2009j]. Other licences require the developer to buy a commercial license.

As part of Qt comes Qt Designer as was already presented in-depth during section 2.12.2. Qt Designer is a tool to rapidly prototype user interfaces and has an excellent user interface which enables the fast creation and modification of GUIs. In summary, Qt offers an excellent platform for C++ application development. Its free availability, large included functionality, cross-platform support, extensive documentation, and the number of useful tools offered make it a very suitable basis to built an interactive prototyping tool.

3.3.4 Viewer

As was already mentioned in section 2.7.4, Coin3D currently offers a number of viewers for different platforms. To avoid having to deal with different viewers for different platforms, Coin3D offers two platform independent viewers. First of all, Coind3D's SoQt Viewer [Kongsberg SIM AS., 2009b] uses Qt for the GUI and works on all platforms. Second, Quarter [Kongsberg SIM AS., 2009c] also offers the same multi-platform support by using Qt. Quarter has a better integration into the Qt framework than

SoQt and can be used as a widget in Qt Designer during the creation of the user interface. Therefore Quarter is favoured over SoQt and the other platform specific viewers.

3.3.5 Build System

The build system should allow developers to work on Studierstube Builder without the hassle of having to deal with compilation configurations. As developers might prefer different operating systems, application development systems, and compilers, the build system has to support this variety of choices. Qt offers such a tool, called qmake [Nokia Corporation, 2009f]. Qmake uses the information in a special file to create a project specific makefile. Qmake can even be used for projects without Qt integration, but some Qt specifics are of course supported too. Another tool which allows automatic makefile creation is Automake by the GNU foundation [Vaughan and Tromey, 2000]. In addition, a tool called CMake [Martin and Hoffman, 2003] has gained wide acceptance in the community. The main difference to other build tools is described as follows:

> CMake is designed to be used in conjunction with the native build environment. Simple configuration files placed in each source directory (called CMakeLists.txt files) are used to generate standard build files (for example makefiles on Unix and projects/workspaces in Windows MSVC) which are used in the usual way. *[Kitware Incorporation, 2009]*.

That is, CMake offers a number of advantages. First, it allows the creation of project files for the desired application development system not just the makefile. Second, CMake is platform independent and has built in support for a number of libraries such as Qt. It can thus be used to control many specialities of Qt, otherwise only available to qmake. Third, it is easy to learn and is used for a number of large open source projects such as ITK, VTK and MySQL. That makes CMake the ideal candidate for a comfortable and platform independent built system. CMake is used as built system for Studierstube Builder.

3.3.6 Application Development System Overview

CMake can generate project files for a number of application development systems. Therefore the decision for the used development system is a personal one. Development took place making use of Visual Studio 2008, enhanced with the Visual Assist-X plugin [Whole Tomato Software, 2009]. Figure 3.8 shows an overview of all the tools used and how they interconnect.

3.4 Component Interrelationship

Before discussing the implementation, this section gives a high-level overview of the interrelationship between the tools presented during the previous section. The operating system on the one hand side, and OpenGL on the other hand, are the two basic components all other parts rely on.

Qt builds on top of the operating system and provides a platform independent framework. As said before, Qt acts as the cornerstone of the development. Qt is an extensive framework with hundreds of classes which are split into packages. The packages relevant for the implementation of Studierstube Builder are only a few. The workspace is implemented making use of the Graphics View Framework. The scene object viewer list, containing all the scene objects, and the field editor are implemented with Qt's Model/View framework. User interaction is handled through the event system of Qt. The entire main window, zoom widget, and context menus are designed in Qt Designer. Qt Designer is also used for the rapid prototyping of user interfaces. The custom file format uses Qt's XML Module. Studierstube is started as a new thread utilizing the Thread Module. Communications between the different program components is done using the Signal/Slot concept of Qt.

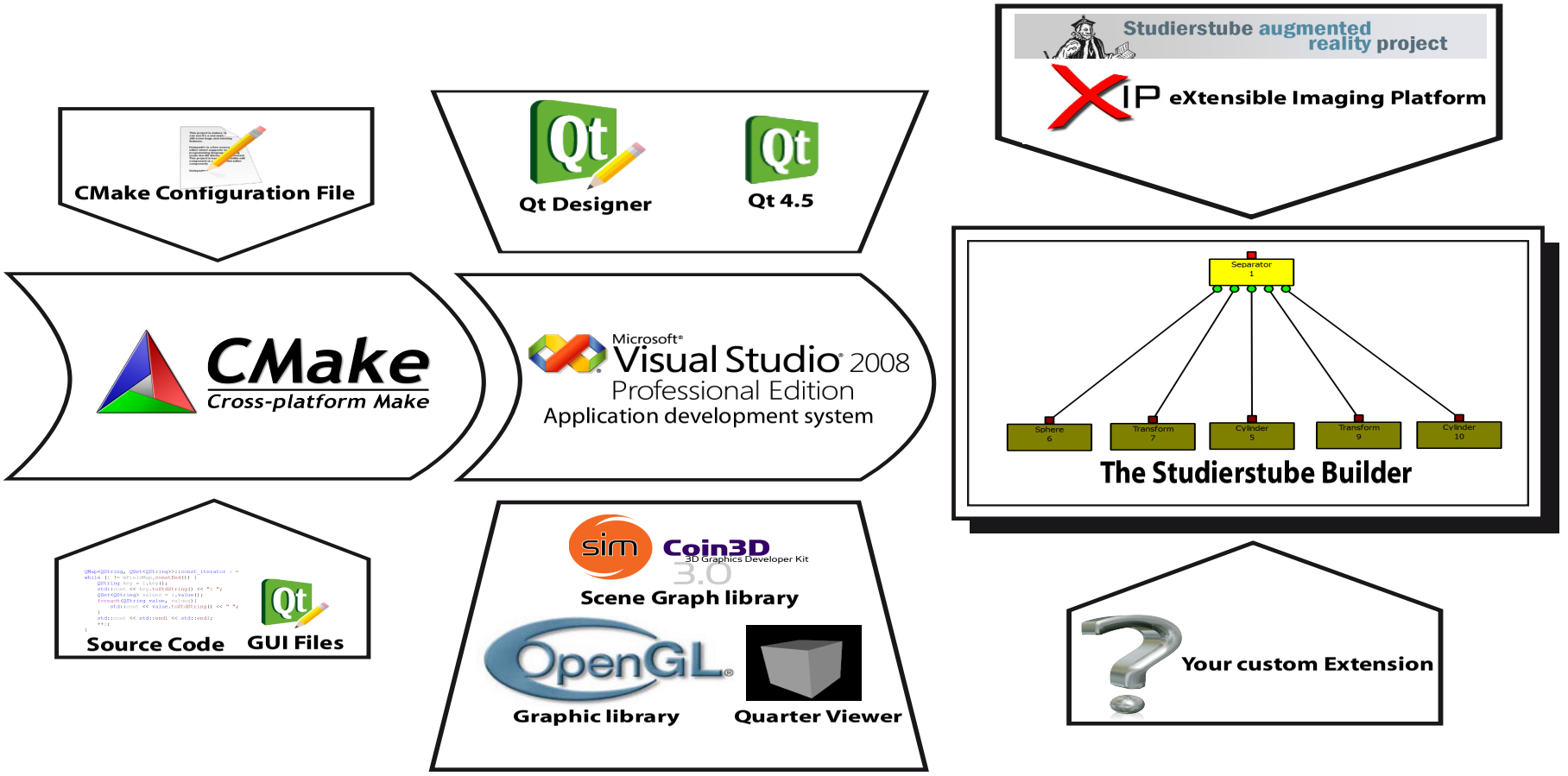


Figure 3.8: An overview of all the tools used. CMake is configured using a textual configuration file. The source code and the GUI-files to be used have to be specified in the configuration file. CMake can then create project files for your application development system, in this case Visual Studio 2008. Visual Studio uses the Qt-libraries, the GUI files created using Qt Designer and Coin3D as a scene graph library. Coin3D uses OpenGL to draw the scene in a Quarter viewer. The project can be compiled and Studierstube Builder can be started. Once started one can use Coin3D objects, the XIP libraries, Studierstube objects or custom written extensions.

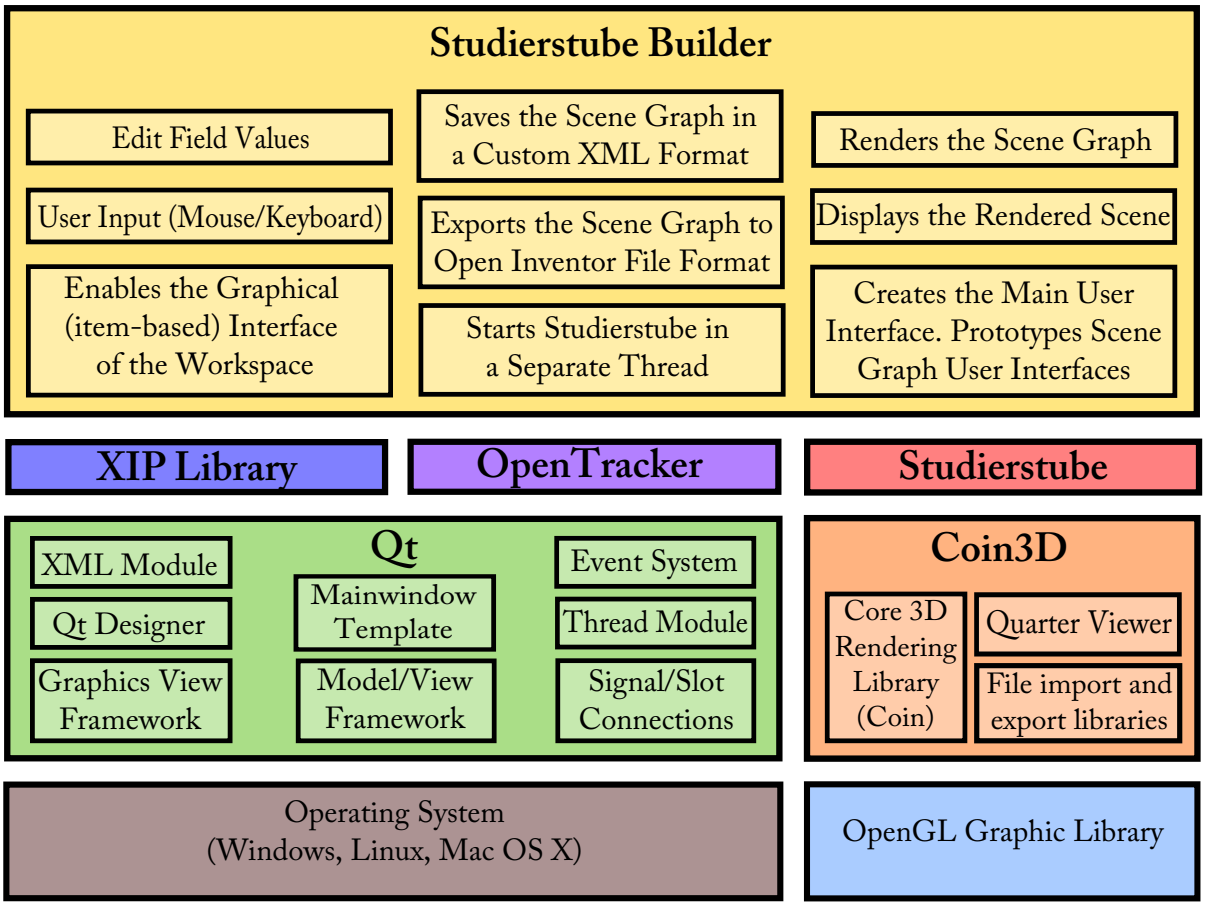


Figure 3.9: The components used for Implementation of Studierstube Builder. The blocks on the very bottom built the foundation for the other parts. Qt supplies components above with a platform independent framework. Coin3D uses OpenGL to perform rendering of the scene graph. Studierstube builds on Coin3D and uses OpenTracker. XIP library uses Coin3D as well. Studierstube Builder sits on top of these other tools and can make use of them.

Coin3D and its viewers are based upon OpenGL and standard C++. Studierstube uses Coin3D as the scene graph library and OpenTracker for tracking. The XIP library is based on Open Inventor but has been ported at the ICG to make use of Coin3D. On top of all these tools and libraries sits Studierstube Builder. Studierstube Builder uses Qt for the user interface and Coin3D as the underlying scene graph library. It can use Studierstube plus OpenTracker and the XIP library. Figure 3.9 shows the interrelationship graphically.

Chapter 4

Implementation

This chapter describes the implementation of Studierstube Builder in detail. The implementation is described per feature. The chapter starts with an introduction of how the user interface is integrated into the rest of the source code in section 4.1. This description is followed by an explanation how Studierstube Builder extracts all scene graph nodes from the Coin3D library during section 4.2. Furthermore, section 4.3 illustrates how drag and drop operations can be implemented in Qt. Section 4.4 presents Graphics View Framework, how work is split between its classes, and how graphics items are drawn. Section 4.5 explains how objects can be connected and what technical pitfalls have to be overcome. Next, the creation of field connections is discussed during section 4.6, followed by a description of how field values can be altered in section 4.7. The mechanisms to allow zooming are discussed within section 4.8. How node kits can be modified and which considerations have to be made is investigated in section 4.9. Moreover, section 4.10 demonstrates the deletion of one or more scene objects. Section 4.11 explains how new scene objects can be added during run-time. Section 4.12 investigates how a Qt user interface can be used to prototype an application's user interface by connecting the Qt interface with the scene graph during run-time. Section 4.13 introduces the file format used to store the scene graph and how Open Inventor file import is implemented. Finally, this chapter closes with section 4.14 describing the integration of Studierstube.

4.1 User Interface Integration

As already mentioned in section 2.12.2 and 3.3.3, Qt Designer was used to create large parts of the user interface. Qt Designer however, can only be used to create the GUI and some very simple behavior through use of predefined Signal/Slot connections (like for example clearing a line edit when a button is pressed). More complex behavior, as it is for example needed for Studierstube Builder can only be realized through programming. Qt Designer and the Qt library offer different ways to integrate the graphical user interface with the source code which defines the program behavior.

Depending on the approach chosen the integration is done at compile time or at run-time [Nokia Corporation, 2009m]. Run-time integration is very practical for many tasks. Run-time loading especially represents an advantage when prototyping the user interface because changes to the user interface do not require a recompilation of the entire source code. For the requirements of Studierstube Builder the compile time approach, however, has several advantages. First of all, code complexity is decreased as many checks do not have to be made. Moreover, loading the user interface during run-time means all user interface components need to be checked before accessing them. Listing 4.1 shows a simple example illustrating the checks which have to be performed.

These checks do not pose a problem for simple examples, but for a large GUI like the main window of Studierstube Builder a tighter integration into the code saves many of these checks and is the better

```
1 QPushButton *ui_findButton = NULL;
2 ui_findButton = qFindChild<QPushButton*>(this, "findButton");
3 if(ui_findButton != NULL){
4 ...
5 }
```

Listing 4.1: A simple example showing the typical checks which have to be performed when assuming certain widgets in the ui-file. The example wants to use a QPushButton with the name ”findButton” for some actions. To get an object pointer Qt offers the qFindChild template function which searches the loaded ui-file after an object with the given name. If there is no such object, a NULL-pointer is returned which has to be checked afterwards

```
1  #include "ui_Mainwindow.h"
2  class Mainwindow:public QMainWindow, Ui_MainWindow {
3    ...
4    void on_actionSave_As_triggered();
5    ...
6  }
7  Mainwindow::Mainwindow(){
8    setupUi(this);
9    ....
10   this->show();
11 }
```

Listing 4.2: The integration of a ui-file for the main window of Studierstube Builder. First of all, the ui-file has to be included. That is done in line 1. Second, the class is inherited from Ui_MainWindow to allow access to all widgets. Line 4 shows an example definition of a function which is automatically called if the ”Save As” button is clicked. In line 8, the setupUI function is called. This is the Qt function responsible to create the user interface. Line 10 finally displays the interface to the user.

option. In addition, more advanced functionality can only be accessed with a compile time integration. The most powerful integration can be reached through single or multiple inheritance from the user interface. To allow this feature Qt offers the so called user interface compiler (UIC) [Nokia Corporation, 2009l] which can automatically create a C++ header and source files from a ui-file. The class that would like to use the ui-file only needs to inherit from the class created by the UIC. Qt furthermore offers a comfortable mechanism to connect signals from widgets in the ui-file to custom slots which implement the source code suitable for the reaction. It is not necessary to create a Signal/Slot connection manually, one can simple define a function which follows the following naming scheme

```
void_<object name>_<signal name>(<signal parameters>);
```

The Signal/Slot connection is then automatically done by the UIC. A simple example showing the integration of the ui-file for the mainwindow can be seen in listing 4.2.

4.2 Dynamic Loading of Scene Graph Objects from Coin3D

To allow drag and drop operations of scene objects, the available scene objects first need to be loaded to the scene object viewer (see section 3.1.2). This is done dynamically to allow custom scene graph objects. Coin3D offers a number of useful API functions to query the library. First of all, Coin3D supports

the creation of a type from a name, for example like this:

```
SoType base_type = SoType::fromName("SoNode");
```

Second, having a type one can get all objects derived from this type through the calls to the following functions:

```
SoTypeList type_list;
SoType::getAllDerivedFrom(base_type, type_list);
```

Third, the elements in the type_list can be asked if their type is abstract or concrete by calling

```
if(type_list[i].canCreateInstance()){...}
```

with a valid index i. Thus only concrete types derived from SoNode remain this way. Note however, that this is not entirely sufficient, since SoEngines are not derived from the SoNode class. The same procedure has therefore to be done with SoEngines as well. Getting all objects derived from SoNode and SoEngine is sufficient so far but one can easily extend the functionality to other types with only a few lines of code. It is however important to only allow concrete objects to get into the scene object viewer list as an instance of the object is created if the user drags an element onto the workspace. This would not work for abstract classes.

All scene objects received through the Coin3D API this way are stored in a QStringListModel and displayed by a QListView.

4.3 Supporting Drag and Drop

Once the scene objects are loaded into the scene object viewer and the GUI is shown to the user objects can be dragged from the scene object viewer onto the workspace to create an instance of a scene object. Drag and drop is something supported very well within Qt. One only has to implement or override a few custom functions (see listing 4.3) to get things working.

Qt uses the Multipurpose Internet Mail Extensions (MIME) defined by [Levinson, 1998] to store relevant information for drag and drop operations and to pass information through the clipboard between different applications. More details about implementation of MIME in Qt is supplied by [Nokia Corporation, 2009g]. Qt uses a slightly modified version of the MVC-pattern to handle data between a view and a model [Nokia Corporation, 2009d]. The model is responsible for holding the data which the view just displays. The view forwards modifications by the user to a so called delegate (a modification of the controller) who forwards changes to the model. In the context of Studierstube Builder, the scene object viewer uses a QListView [Nokia Corporation, 2009k] to display data while a QStringListModel supplies the data which should be displayed. The data is just a string list of scene objects (retrieved from the scene graph library) which are displayed in alphabetical order. The view is updated automatically whenever the data in the model changes. Drag and drop operations have to be implemented into the model class as the view is only responsible to display the data.

To allow custom drag and drop operations, it is necessary to inherit a custom class from the model used and override a number of functions. In this case a QStringListModel is used. Listing 4.3 shows the relevant parts of the new model. To keep things simple, just the name of the scene object the user wants to drag is copied into a custom MIME type. While a number of plain text MIME types are suited to hold just a plain text object name, using them might be dangerous. Other applications supporting drag and drop might use the same MIME format to copy text and a user might end up dragging text from another application into Studierstube Builder. The scene graph library might then be confronted with

```
1  class SceneObjectListModel:public QStringListModel{
2    //Defines the supported drag actions
3    virtual Qt::DropActions supportedDragActions () const;
4
5    // The flags of the items
6    virtual Qt::ItemFlags flags(const QModelIndex &index) const;
7
8    //defines the MIME types
9    virtual QStringList mimeTypes() const;
10
11   // Creates the mime data from the selected items
12   virtual QMimeData *mimeData(const QModelIndexList &indexes) const
       ;
13
14   ...
15 };
```

Listing 4.3: The relevant part of the new model derived from QStringListModel. Line 3 defines the action performed when a drag starts. This can for example be copy or move. Since the original model should not be changed the data is only copied. Line 6 signals the item flags to the view. The flags are responsible to define whether an item in the list can for example be select or dragged. Without allowing them to be dragged, no drag and drop can be performed. The function in line 9 defines the MIME types which can be created. Qt allows to set multiple MIME types at the same time, holding different data. Finally, the function in line 12 actually creates the MIME data and decides under which of the defined MIME types the data is stored.

the desire to create a new scene object with an unknown type. Therefore, only text dragged from within Studierstube Builder should be allowed. This can be reached through the functions in line 9 and 12 of listing 4.3. The MIME format used is:

```
application/StudierstubeBuilder/SceneObjectNameList
```

With the custom model scene objects can be selected and dragged around, with the correct MIME type and data being set. However, that is only the first part of the solution. A second part which accepts the data in the MIME format introduced and performs the necessary steps to create a real scene graph object out of the data supplied, is required, as well. This second part is the workspace, which accepts drops in the required format. The workspace, whose implementation will be described in the upcoming section, basically needs to implement two functions [Nokia Corporation, 2009a]:

```
virtual void dragEnterEvent( QDragEnterEvent* event );
virtual void dropEvent( QDropEvent* event );
```

The dragEnterEvent is used to inform Qt of the MIME types accepted. It is called once, when a drag operation is detected which enters the area of the object implementing the dragEnterEvent. It just checks for a correct MIME type and accepts the drag, which modifies the mouse icon to signal the user that a drag can be performed here. The dropEvent function is called by Qt when the user drops the dragged items on the workspace. This function is responsible to extract the relevant data (the name of the scene object in this case) from the MIME data stored in the drag event and perform an appropriate action. In this case the creation of a scene object instance of the given type.

4.4 Graphical Representation of Scene Objects

This section focuses on how the scene objects are represented graphically after the user drops an item onto the workspace. Most of the implementation is done in subclasses of Qt's Graphics View Framework from which the most important concepts are introduced. Much functionality is related to these classes. To have a better separation of the different topics, not all of the features using the Graphics View Framework are described throughout this section. The goal is to give a high level overview of how objects are created using the framework and how the different parts work together, thus setting the ground for a description of the individual features throughout later parts of this chapter.

4.4.1 The Graphics View Framework

Qt offers the so called Graphics View Framework which is described as:

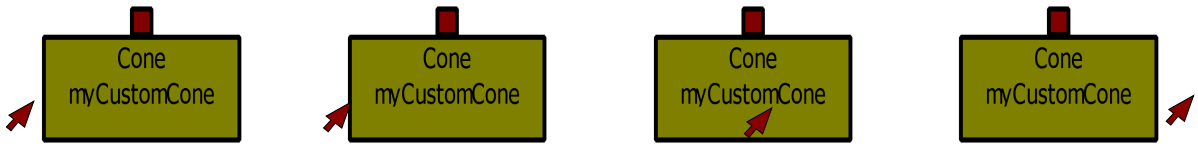
> Graphics View provides a surface for managing and interacting with a large number of custom-made 2D graphical items, and a view widget for visualizing the items, with support for zooming and rotation. The framework includes an event propagation architecture which allows precise double-precision interaction capabilities for the items on the scene. Items can handle key events, mouse press, move, release and double click events, and they can also track mouse movement. Graphics View uses a Binary Space Partitioning (BSP) tree to provide very fast item discovery, and as a result of this, it can visualize large scenes in realtime, even with millions of items. *([Nokia Corporation, 2009c])*

As a consequence, the Graphics View Framework is perfectly suitable to solve some of the tasks presented. The Graphics View Framework again makes heavy use of a slightly modified MVC pattern (see section 2.10.4). The work is split between three core classes: the Scene, the View, and the Item. The functionality of these classes is summarized from [Nokia Corporation, 2009c].

The Graphics Scene is the "model" behind the view seen by the user. It is one of the cornerstones of Studierstube Builder. The scene receives user events which are forwarded from the view and is responsible to propagate them to the correct items. Usually that is the item at the cursor position, where the user has created an event. Furthermore, the scene is responsible to manage (add, delete) all items in the scene and to keep track of the item's state. Examples for the item state are the selection of items or an item having the (keyboard) focus.

As intended in the MVC pattern, one scene (=model) can have multiple views attached showing different parts of the scene or using a different visualization. One of the views main responsibilities is to propagate user events to the scene. As the view can change the transformation matrix, it is possible to zoom or rotate the contents of the scene inside a view. User events, thus, have to be converted by the scene from the current view coordinates to the actual scene coordinates. The view offers a number of functions which allow this mapping.

The items are the actual content of the scene. Items can have a large number of functions, mainly for event management. Usually a scene forwards the events received to the item under the mouse cursor. Items can for example react to mouse events (moving the mouse, pressing or releasing of buttons) keyboard events (pressing/releasing a button) and much more. Furthermore, a mechanism called hover-events enables to display information or perform some actions, if the user moves the mouse over an item. As with the view, items have their own, local coordinate system. Items decide how they paint themselves and have to specify their bounding boxes for collision detection by the scene. A number of pre-defined Qt classes offer simple items which represent geometric primitives and text but completely own classes can be used as well.



(a) No hover event **(b)** hoverEnter **(c)** hoverMove **(d)** hoverLeave

Figure 4.1: Illustration of the different hover events. They happen exactly in the order from 4.1a to 4.1d. Only the event shown in 4.1c must not always be performed, as an item could be entered and left immediately afterwards.

4.4.2 Implementing the View

The first responsibility of the view is to handle drop operations as described in section 4.3. The drop operation has to be implemented in the view as this is the component the user interacts with. The remaining implementation work is basic. Most of the functionality, like forwarding events to the scene, is already done in the QGraphicsView class, the base class for all views in the Graphics View Framework [Nokia Corporation, 2009k, QGraphicsView]. Another important task is the management of transformations applied to the view. Rotation does not make much sense for a scene graph builder, but zooming enables the viewing of different levels of details. That is, the view has to catch some of the mouse or keyboard events used for zooming which are otherwise just forwarded to the scene. The implementation of zoom control, however, depends on some other components, as well. How these components work together is discussed in more detail during section 4.8.

4.4.3 Implementing the Scene

The scene does most of the management of scene items and has to make proper reactions to the user input. As already mentioned in, Qt distinguishes between different user events. Listing 4.4 shows an overview of the user event functions which are used in Studierstube Builder. QGraphicsScene offers standard implementations for the event functions suitable for most of the standard tasks [Nokia Corporation, 2009k, QGraphicsScene]. Based on the default implementation, one can build an interactive application and save much development time. With a few exceptions the default implementations have just been extended. The way these extensions usually work is by overriding certain virtual functions from the QGraphicsScene base class. Some examples can be seen in listing 4.4. The overridden virtual function is called instead of the base classes implementation. The own function implements some additional behavior, like remembering certain data or drawing new items before it calls the default implementation of the QGraphicsScene. This way one can use most of the work already done by Qt and still extend the behavior. A few exceptions where this does not work are, however, still present.

One such example where the default implementation is not sufficient are the so called hover events. Hover events are by default sent to items under the mouse cursor. Three type of hover events exists. First is the hoverEnter, second the hoverMove and third, the hoverLeave event. When the user moves his mouse around over the view, the mouse position is forwarded to the scene which checks for an item under the current mouse cursor. If an item is found a hover event is sent to the item. Hover events are always sent in the order hoverEnter, zero or more hoverMove and a hoverLeave. Figure 4.1 illustrates the idea. If multiple items overlap and the user moves the mouse from one item directly to the next (without ”touching” the empty workspace), Qt always guarantees that a hoverLeave event is sent to the last item before the new item receives the hoverEnter event. Getting back to the original topic of the default behavior and extensibility, hover events were a problem. Hover events are only sent if the mouse is moved around, not if the mouse is moved around while having a mouse button pressed. The straight forward approach would be to implement the custom hover events (while having a mouse button pressed)

```
1  class WorkSpaceScene : public QGraphicsScene{
2    //additional functions here
3    ...
4  protected:
5    // This function is called whenever a mousebutton is pressed
6    virtual void mousePressEvent ( QGraphicsSceneMouseEvent *
       mouseEvent );
7
8    // This function is called whenever a mousebutton is released
9    virtual void mouseReleaseEvent ( QGraphicsSceneMouseEvent *
       mouseEvent );
10
11   // This function is called whenever the mouse is moved
12   virtual void mouseMoveEvent ( QGraphicsSceneMouseEvent *
       mouseEvent );
13
14   // This function is called whenever a double click is received.
15   virtual void mouseDoubleClickEvent ( QGraphicsSceneMouseEvent *
       mouseEvent );
16
17   // This function is called whenever a keyboard key is pressed
18   virtual void keyPressEvent ( QKeyEvent * keyEvent );
19
20   // This function is called whenever a a context menu event is
       received. Usually that is when
21   // the user presses the right mouse button over an item
22   virtual void contextMenuEvent ( QGraphicsSceneContextMenuEvent*
       contextMenuEvent );
23
24   //more functions here
25   ...
26 };
```

Listing 4.4: A selection of the functions offered by Qt which are called upon certain user events. The mouse events in line 6, 9, 12 and 15 have an object of the QGraphicsSceneMouseEvent type as parameter which contains all relevant information. Most important, certainly, is the position the event occurred (in different coordinate systems) and which button was pressed. The function in line 18 shows how keyboard events are processed. The function is called and the information is passed as an instance of QKeyEvent. QKeyEvent holds the type of key pressed and modifiers (like for example ctrl or shift) pressed together with the key.

into the mouseMoveEvent which can be seen in line 12 of listing 4.4 and not calling the default behavior in this case. This however, leads to new problems as the default implementation is now called only when no mouse button is pressed and the custom event creation is called otherwise. The Qt implementation does heavy caching to improve performance. By relying only on half of the implementation this scheme is mixed up. So extending the default behavior has to happen with great care.

Another responsibility of the scene is to manage the items of itself. The QGraphicsScene offers two methods intended for this:

```
void QGraphicsScene::addItem ( QGraphicsItem * item );
void QGraphicsScene::removeItem ( QGraphicsItem * item );
```

The scene takes care of the deletion of all items upon destruction. Once the programmer removes an item from the scene, the deletion must be carried out by the user. A scene graph editor, however, needs a little more information than just the number of items in it. A number of functionality needs access to the items or other information related to the items. As such information is requested frequently throughout the implementation of the scene, fast data access has to be guaranteed. For this reason, hash-based data structures are used which allow an average lookup time of only $O(1)$ [Nokia Corporation, 2009b]. The data structures store direct pointers to the objects for the fastest possible access.

4.4.4 Implementing Items

Without implementing items the user would not see a single pixel of graphics displayed on the graphics view [Nokia Corporation, 2009k, QGraphicsItem]. Items are the actual content which is drawn. As most of the custom user events are handled by the scene, the items main responsibility is to make sure it is drawn correctly and to define some functionality which helps the scene determine whether a certain area has to be redrawn or a collision between items occurred. Of course, a certain number of event handling function is required here too, but the items base class QGraphicsItem offers suitable standard implementations as well. Listing 4.5 illustrates the most important functions of an item. The paint function is introduced first as the implementation of other functions depends on it. Painting is done by composing a number of geometric objects which use a QPainter [Nokia Corporation, 2009k] class responsible for the primitive drawing. A QPainter class can define different drawing styles which directly influence the appearance. Most important are the boundary color, pattern and thickness as well as the filling pattern and color. One can create arbitrary complex shapes, but using only primitives such as rectangles or circles is sufficient. For these primitive function Qt offers standard implementations. The caller just needs to specify the dimensions of the primitives to draw. All dimensions and coordinates are done in local coordinate system, where one unit represents one pixel drawn if the view watches the scene with the default transformation matrix.

The boundingRect function (see line 10 of listing 4.5) defines the axis-aligned bounding rectangle of the current item, again in local coordinates. No painting must occur outside the bounding box. This would lead to rendering artifacts. The bounding box is mainly used by the scene to quickly determine the correct item, for example during a mouse event and for culling of items not displayed in the current view. If the scene detects an event happening inside the bounding rectangle, it will call the shape function to determine whether or not the event is really happening inside of the item.

The shape function is finer grained than the boundingRect function, which is called if precise decisions are mandatory. It must supply the scene with exact information of the outline of the tested item. Implementing these functions correctly can be a tedious task, as all computations have to be pixel accurate and the composition of multiple geometric primitives largely increases the complexity of the implementation. Supplying efficient implementations of these functions is, however, mandatory, as they are called extremely frequently by the scene.

Figure 4.2 shows a sketch of the implemented graphic items. Note however that this is a simplified

```
1  class GraphicalSceneObject: public  QGraphicsItem
2  {
3    //additional functions here
4    ...
5  protected:
6    //! \brief paints the object
7    virtual void paint(QPainter *painter, const
        QStyleOptionGraphicsItem *option, QWidget *widget);
8
9    //! \brief returns the bounding rectangle for the object
10   virtual QRectF boundingRect() const;
11
12   //! \brief returns the exact shape of the object
13   virtual QPainterPath shape () const;
14
15   //! \brief returns the type of this object
16   virtual int type() const;
17
18   //! \brief function that handles hoverEnterEvents
19   virtual void hoverEnterEvent ( QGraphicsSceneHoverEvent *event );
20
21   //! \brief function that handles hoverMoveEvents
22   virtual void hoverMoveEvent(QGraphicsSceneHoverEvent *event);
23
24   //! \brief function that handles hoverLeaveEvents
25   virtual void hoverLeaveEvent ( QGraphicsSceneHoverEvent *event );
26
27   //more functions here
28   ...
29 };
```

Listing 4.5: Line 7 shows the functions necessary to paint content. Line 10 and 13 offer functions for the bounding box of the item and the exact outline of the item. These functions are used by the scene for collision detection and repainting of certain areas. The type function in line 16 can be used to quickly determine the type of the item. As there might be different types of items which require unequal handling, this function can be used to check for the correct type before doing an expensive casting operation. The functions in line 19, 22 and 25 implement the proper reaction for the respective item to different hover events.

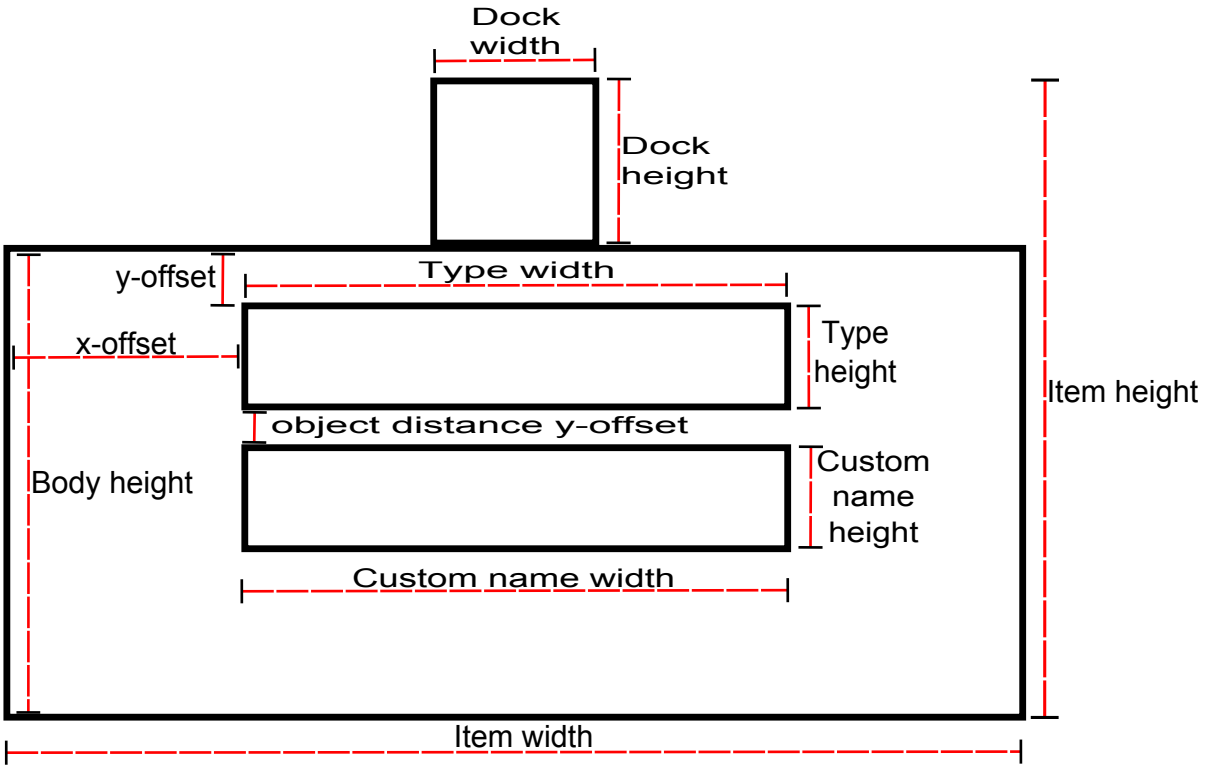


Figure 4.2: A sketch of the item which represents a normal scene nodes. SoEngines and SoGroup nodes have a different appearance. Setting up the paint function requires pixel accurate computations of the position and height/width of each geometric primitive drawn. It has proven itself useful to specify a number of variables for each height or width involved and make the computations based on these variables.

sketch. The paint function is defined to draw only half of the outline inside, and the other half outside the object. To compute, for example, the item height, one has to add $bodyheight + dockheight + 1.5 * outlinethickness$. Note that the dock is a little offset in y-direction and the outline thickness is only counted for the top edge, leading to the factor of 1.5. When using variables to define the position and the dimensions of the geometric primitives, computation of the bounding box and the shape function become easier.

4.5 Connecting Items to Graphs

Once scene graph objects have been created and are represented by graphical scene items, it is time to allow the user to connect them. As the creation of scene items is done using drag and drop, connecting them should be possible in the same way. During dragging of a connection line a position preview should be shown.

To realize the desired position preview, hover events are used. As described before (see section 4.4.3), the default Qt implementation of the mouseMoveEvent does not deliver hover events while a mouse button is pressed. To enable drag and drop of lines as described above, this feature had to be implemented manually. Creating this feature proved to be much harder than expected. The reason for this is the previously described caching of certain information. Setting of self created hover events when dragging a line and relying on the default implementation otherwise, caused incorrect results as the order in which hover events are usually sent proved no longer true. The default implementation triggered wrong hover events in between, as the caching got mixed up. Not calling the default implementation at all however, has several side effects. As various parts of the default functionality depends on each other,

it is not sufficient to just overwrite the mouseMoveEvent. One rather has to implement the entire event system and user input control. As the rest of the default behavior is desired, implementing everything from scratch was seen as a disproportional amount of work for the desired task.

The workaround chosen was to establish a dedicated class (called HoverManager) which stores information about all the hover events currently sent. The HoverManager is called right at the beginning of each of the hover functions implemented in the graphics item. See listing 4.5 for more details. This way the HoverManager is called no matter if the hover event is sent by the default implementation in the graphics scene or during the custom creation while dragging a line. The HoverManager has therefore higher level knowledge about the last hovered object and can thus enforce the strict order of hoverEnter, hoverMove and hoverLeave events by sending out custom hover events to enforce the order. Although not very elegant, this solution proved to work very well with a fraction of the effort involved in a reimplementation of the entire event system. The HoverManager is implemented making use of the Singleton pattern described in section 2.10.3

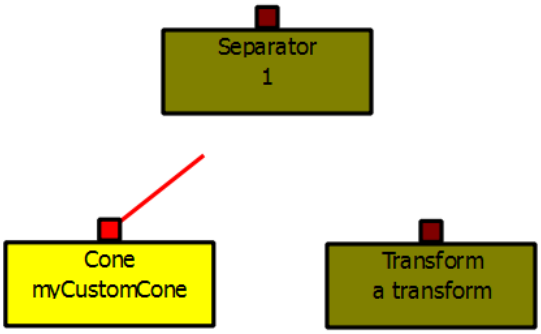
Getting the order of the hover events right and sending hover events even when dragging a line is a large step towards a position preview. The only thing left now is the implementation of the functionality into the hover event functions. A separate class called EndLineManager is used to manage the order of the incoming connections in a group node. An incoming connection from a group node's perspective means a connection going from a child to the group node. Incoming connections are displayed as green circles below the item in contrast to the red rectangles used for the docks which are placed above the item. Independent of the number of incoming connections, the docks are always evenly distributed over the items width. This is done through splitting the width in n evenly distributed sections, with n being the number of incoming connections. While dragging a line over a group item, an additional section is added and an additional green circle is shown. The position preview does not move with the cursor until the user moves the mouse cursor into the next section, in which case the preview snaps to the new position. When the user moves the mouse cursor outside of the item, the position preview is removed again. If the user decided to add the line connection at the position currently previewed, the temporary objects are inserted permanently. This architecture naturally extends to the hover events. The hoverEnter event adds the preview at the current position. The hoverMove event checks the new mouse coordinates, computes the section in which the preview shall be shown and rearranges the lines if necessary. HoverLeave finally removes the preview and restores the original position. A mouseRelease event accepts all temporary changes into the model. In case a group node has many incoming connections, the item width is adjusted so that a minimum distance between the green circles is kept. Figure 4.3 illustrates the ideas described.

4.6 Setting up Field Connections

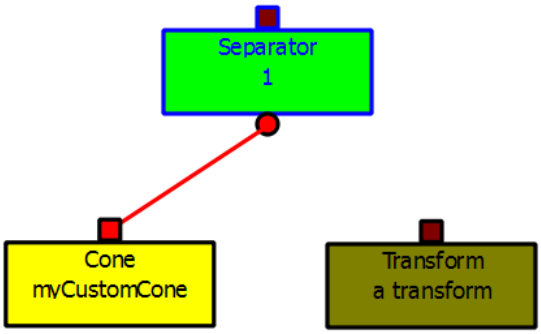
One essential feature of scene graphs is the ability to create field connections. The concept of field connections is described in section 2.3.2. Looking just at the user interface creating field connections can be done in a manner almost similar to the connection of scene objects. Field connections again make heavy use of drag and drop and give the user instant visual feedback during the connection.

To enable field connections, an additional so called Field Connection Mode is being introduced. The Field Connection Mode complements the already existing so called Scene Object Mode. The Scene Object Mode is the mode used to connect scene objects to scene graphs, move items around, delete them and so on. All possible actions discussed so far can be performed in Scene Object Mode. Scene Object Mode is thus the "standard mode" for almost all tasks. Field Connection Mode in contrast is only used to create field connections between different items. Using the toolbar or the menu, one can switch between Scene Object Mode and Field Connection mode. The modes are mutually exclusive.

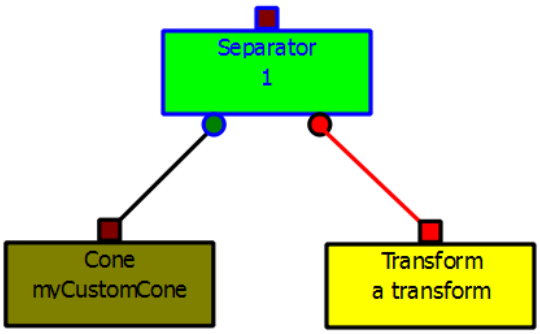
When switching to Field Connection Mode, field connections can be created by simple starting a drag operation anywhere inside an item. As with the connection of objects a line appears which can be dragged to the intended item which is highlighted as soon as the line is dragged inside its object



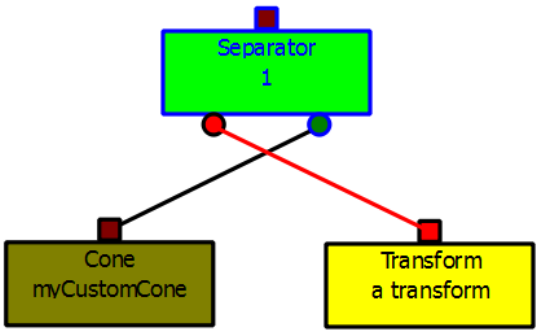
(a) Dragging a line



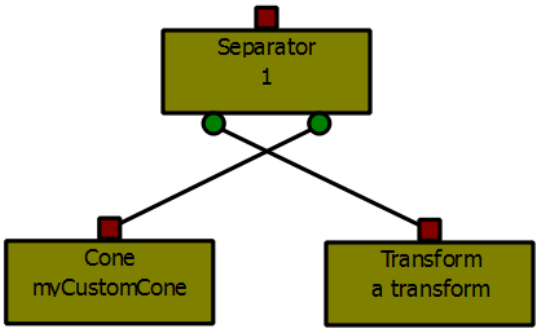
(b) Position preview shown



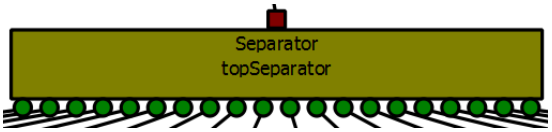
(c) Adding another connection with position preview



(d) Switching position changes the preview



(e) Releasing the dragged line adds it permanently



(f) Many incoming connections increase the items width

Figure 4.3: Illustration how the position preview works. In figure 4.3a a line is dragged from the dock of the cone object to a group node. As soon as the mouse cursor is moved over a group node, the node becomes highlighted and the position preview is shown (figure 4.3b). Since this is the only connection present at the moment the preview is positioned in the middle of the item. In figure 4.3c another position preview is shown. This time instead of releasing the line, the user moves the mouse in the left half of the item and the preview snaps into the new position on the left half of the item as can be seen in figure 4.3d. Note how the available space is split evenly into two sections, with each incoming connection being placed in the middle of its section. In figure 4.3e the drag is released and the connection is established. The item is no longer highlighted. Figure 4.3f shows an example of a group node with many incoming connections. The items width is adjusted so that a minimum distance between the docks of incoming connections is kept. If connections are deleted the items width is reduced until the standard width is reached.

boundaries. This give instant visual feedback about the target of the operation. To avoid confusion between the connection of items and creating field connections, the graphical feedback looks different. First of all, when connecting items, the line which is shown while dragging always starts at the dock while dragged field connections start on the right edge of the item. Second, the style used for the dragged line differs. Connecting items is done using a solid red line opposed to a dotted blue line used for field connections. Third, object connections end at the lower edge of the item with a green dot being shown. On the contrary field connections end at the left edge without a dock being shown.

When the user drags a field connection line from the source (master) to a target (slave) and releases the dragged line a dialog window pops up showing all the fields which can be connected. The user can then select a master field and a slave field to connect. As each scene object has multiple fields usually with different types, fading out incompatible types as soon as the user has selected one of the two fields further aids the user. This approach is implemented enabling only fields of the same types to be connected. Allowing only fields with the exact same type to be connected is however only half the truth. Coin3D supports the connection of fields with unequal types as long as they are semantically compatible. It is for example possible to convert a field holding an integer to a field holding a decimal number. This feature is supported using so called Field Converters which are placed in between the connection to perform the conversion of field values. The dialog window therefore allows the user to disable the fade out of fields and connect fields of all types. After closing the dialogue, the field connection is created if the types are compatible. If types are not even compatible with a field converter in between an error message is shown and the field connection is not created. Another special case exists where a user connects two master fields to the same slave field. Per definition this is not allowed in Open Inventor. The new field connection replaces the previous one in this case. The same approach has been implemented. A second connection to the same slave field replaces the first one and the connection line of the old field connection is removed. Figure 4.4 illustrates the presented ideas.

4.7 Modifying Field Values

Modifying field values is one of the most important scene graph operations. When creating a scene object, all fields are created using these default values. Modifying these values leads to a custom scene graph. Coin3D offers a large number of different field classes for all kind of data types.

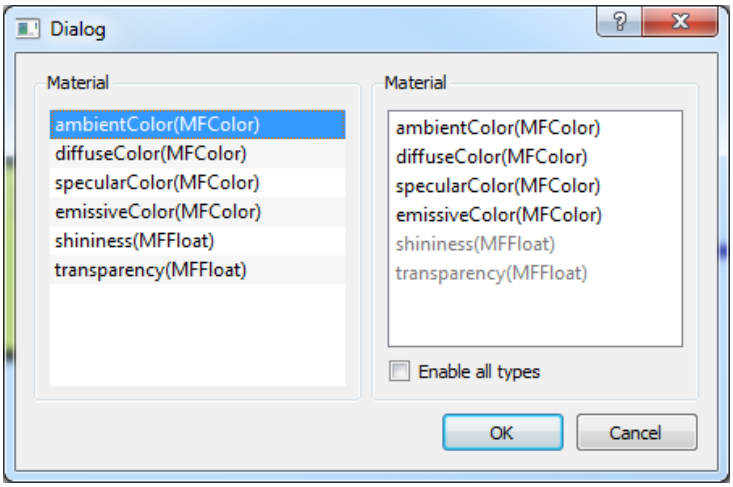
The large number of fields requires careful thoughts on which input widget to use. Coin3D offers options to get and set field values as strings. That means that theoretically a simple line edit could be used for all modifications no matter which field type is modified. However changing fields this way can be a tedious and error-prone task which decreases the usability. Input errors are another topic which has to be considered. Most Qt input widgets allow to restrict the input characters through different mechanism. For some primitive types such as integers, limiting the input to for example only numbers is sufficient.

The implementation is done in multiple steps. First, read out field values and types from the scene objects. Second, store this information in a model so that the field editor can display it. Third, if the user decides to change a field value, use an appropriate input widget (depending on the type of the field) to make the input easier. Fourth, once the user is finished, check the input for errors. Finally, if the input is correct modify the field to hold the new value.

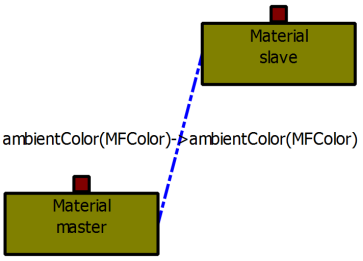
Reading out the fields is very simple, so it is not covered in detail here. The interested reader can check the Coin3D documentation [Kongsberg SIM AS, 2009] of the SoFieldContainer and SoField classes. What is more interesting is how to store the information in a model and display that information in the field editor. The underlying mechanism is the Model/View architecture Qt uses [Nokia Corporation, 2009d]. As already mentioned during section 4.3, Qt uses a modified MVC pattern to separate the model from the view. Although similar concepts are used in the graphics framework, MVC for views implementation differs in some details. All MVC classes use abstract base classes. View and Controller are combined into one class which simplifies the framework. The delegate is used when items are edited

(a) Dragging a field connection line

(b) Target item is highlighted



(c) Dialog to chose the fields which should be connected



(d) The created field connection

Figure 4.4: Illustration how field connections can be created. After switching to Field Connection Mode, one can drag a field connection between scene objects. Figure 4.4a shows how field connections look like. Note the different drawing style in comparison to normal connections. Figure 4.4b illustrates the visual feedback given as soon as the user moves the cursor over a target item. Figure 4.4c presents the dialog window which appears after dropping a field connection line over an item. The master object is displayed on the left hand side, contrary to the slave field which is displayed on the right hand side. The type is shown in brackets beside each field. Note how unequal types are disabled. The checkbox below the slave fields can be used to enable all types for selection. Figure 4.4d finally displays the result after the selection of two compatible fields. The connected fields are displayed as text attached to the line. This way the user can see the connected fields within a glimpse.

```
1  class FieldTypeDelegate: public QItemDelegate
2  {
3    // Creates the and returns the inputs widget suitable for the
       item.
4    QWidget *createEditor(QWidget *parent, const QStyleOptionViewItem
         &option, const QModelIndex &index) const;
5
6    // Sets the data from the model in the given input widget
7    void setEditorData(QWidget *editor, const QModelIndex &index)
       const;
8
9    // Sets the data from the input widget back into the model
10   void setModelData(QWidget *editor, QAbstractItemModel *model,
       const QModelIndex &index) const;
11
12 signals:
13   // Signals that an item has changed
14   void fieldValueChanged(QStandardItem*) const;
15 };
```

Listing 4.6: Line 4 shows the function called when a user double clicks on a field. It must return a QWidget suitable to edit the fields value. The function in line 7 is responsible to set the data stored in the model in the widget. Depending on the type of the widget setting these value will be different from widget to widget. The function in line 10 writes the modified data back into the model. Finally the signal used to notify the viewer from the changed data is shown on line 14

and is responsible for rendering during that time. Changes are communicated using the Signal/Slot architecture. The model in this case is a QStandardItemModel which supports the storing of multiple rows and columns which can then be displayed in a view. For the view a QTableView is used as it implements a standard table behavior which is good for displaying different types of data such as field name, type and value. The view and the model can be used without inheriting a new custom class. The standard behavior is fine. The only thing one needs to know is how to store data in the model [Nokia Corporation, 2009e]. The standard model uses a table like structure which can be accessed via row and column index. A useful feature is to specify different roles for data. The content used to display information on the view uses the Qt::DisplayRole. Other roles can be used to store additional information. Fields are stored in the model in the same order they are returned from Coin3D, displaying the field name, the field type and the field value. The user can edit the fields value by double clicking on it. The other entries (name, type) in the table can not be edited. Some fields which do not allow the user to enter meaningful information such as pointers to other scene objects that are stored in fields can not be edited by the user.

If the user decided to change a field's value, the delegate class comes into play. This is the only class in the Model/View framework which needs to be customized, although this is very simple using only a few functions. Listing 4.6 gives an overview over the most important functions. The functions are called in the order listed. As there are so many different field types to handle, the createEditor function has to create a large number of different input widgets for different field types. To reduce complexity, the instantiation of the correct widget has been moved into a class using the Factory pattern as described in section 2.10.2. Some field values have a natural matching to an input widget. Enums for example can be handled using a combo box, boolean values map perfectly to a check box and integers can make use of spin boxes. The majority of field types however would require very specific input widgets. The creation for such a large number of widgets is beyond the scope of this work. For widgets which do not map to an input widget a simple line edit is used which allows textual input.

After the user has modified a field, error control should be performed. Coin3D has some built in error control as well. Setting the value of a field will return an error if the value contains faulty input.

The problem with these built in error control, however, is that it accepts user input as long as it is correct and starts rejecting input from the moment on the first incorrect character is found. This way the user can partly modify the value of a field with only partly correct input. This behavior is undesired as it may lead to strange effects and is a little counter-intuitive. None of these problems are true for specialized input widgets which limit for example the input characters. Most fields use a line edit which enables textual input. This is where the error checking has to be done. Qt offers a Regular Expressions (RegEx) filter which can be set on a line edit to disable a number of characters, depending on the RegEx supplied.

Regular Expressions are defined as:

> A regular expression is a specific kind of text pattern which you can use with many modern applications and languages. You can use them to verify whether input fits into the text pattern, to find text which matches the pattern within a larger body of text, to replace text matching the pattern with other text or rearranged bits of the matched text, to split a block of text into a list of subtexts, and to shoot yourself in the foot. *([Goyvaerts and Levithan, 2009])*

As useful as this filtering is, the strict enforcing of the rules forbid the use of this solution. Setting a RegEx filter this way leads to perfect security if the user inputs a new string from the beginning to the end. It fails however, if the user wants to modify a passage somewhere in the middle of the string. The RegEx filter does not allow to input characters which violate the RegEx even temporarily. So modifying parts in the middle of the string does not work as this will almost always violate the RegEx condition. So this approach is not suitable. What was done instead was to allow the user to input all characters wanted and make a RegEx-based check after the input has been done but before setting the new fields value. This approach allows the highest possible flexibility to the user without running into the problems the Coin3D error handling represents. A detailed description of valid input for all field types is presented in [Wernecke, 1993, chapter 11]. More details about RegEx can for example be found in [Goyvaerts and Levithan, 2009].

Once the input is validated with a RegEx the field value is written back to the field of the scene object. In case the input is not validated, the field value is not changed. Instead an error message is displayed to the user and the flawed input is stored. The next time the field is edited, the flawed input is shown so that the user can correct the error.

4.8 Zooming In and Out

Zooming is one of the transformations supported by the view. The only challenge is to synchronize the different widgets which allow zoom control. Zooming can be done through the menu, the toolbar, the context menu, the mouse wheel (in combination with the ctrl-button) and through a so called ZoomWidget in the status bar. The ZoomWidget allows the most flexible regulation of zooming. The user can use a textual input, a plus/minus-button and a slider to quickly jump between large ranges. Zooming can be done in the range from 10% to 999%. Allowing constant zooming steps over such a large range is a bit frustrating for the user. The area below 100% is most sensitive (zooming is done in small steps), while the large range between 100% and 999% requires larger steps. To keep the advanced control in the status bar synchronized with the rest of the view, all other zooming actions just forward zooming events to the ZoomWidget. The ZoomWidget computes the new zoom level based on the current zoom level and on the current step size and updates all other zoom widgets such as the slider or the textual display with their new values. The new zoom value is finally translated into a transform which is set for the graphic view.

4.9 Editing Node Kits

Node kits are something special in scene graphs. A node kit manages the children hold by the kit using a so called catalogue. In contrast to normal group nodes, a node kit does not allow direct access to its children. One has to use access functions provided by the catalogue. Node kits can also enforce a certain structure by defining the order and the type of the managed children. The catalogue holds information about this structure and the type of the children allowed. None of the editors analyzed during chapter 2.12 enables the creation of node kits. As node kits are used frequently in scene graphs supporting their modification through a graphical user interface is seen as a strong plus.

However the more restrictive design of node kits poses some challenges, especially with the drag and drop approach used so far. As a node kit is not derived from the SoGroup node no functions for adding or removing children are defined. A more structured approach using the node kit catalogue has to be applied instead. In addition how should node kits be displayed graphically? They must be distinguishable from the other none-node-kit nodes on first sight, as they cannot be edited the same way. Node kits can also be used to hide complexity as only one scene object can be added to a scene graph instead of an entire selection of objects. This encapsulation of complexity should be reflected by the graphical representation of node kits as well.

The approach chosen is an attempt to fit the more rigid structure of node kits into the flexible designed user interface of the rest of the application. Node kits can be created as all other objects by selecting them in the scene object view and dragging them on the workspace. Node kits behave like normal nodes in the way that they have a dock which can be used to connect them into a scene graph. The catalogue used to manage the objects of a node kit holds information about which objects should be created on default and which only on request by the user. By creating an instance of a node kit, all these default elements are created automatically. The user can edit the properties of the node kit with a simple double click. This triggers several actions. First, a new tab is created which holds an initially empty workspace. Second, the newly created workspace is filled with all the scene objects from the node kits catalogue. So all objects known to the catalogue are instantiated at that time. Third, the scene objects are laid out and connected as dictated by the node kit catalogue. The layout is done in a simple manner starting from the node kit root by recursively placing all the children of a node at a certain distance below the parent object. Within the newly created node kit tab a number of user actions is not allowed. First of all, deletion of existing and creation of new scene objects is not allowed. The scene object view, the delete key and entries in the context menu are disabled for this purpose. The same applies to connections. No new connections can be made and no existing connections can be deleted as this actions are disabled as well. Thus the initial structure can not be altered.

The rest of the user interaction is not limited. One can move objects around and field values can modified. The user can change the properties and the appearance of a node kit through the modification of field values. To modify field values the concepts and techniques discussed in section 4.7 remain unchanged. As it is possible for node kits to contain other node kits as their children, this feature is supported as well. By simply double clicking on a node kit a new tab with the properties described before is created. So each node kit owns a dedicated tab. This helps to hide complexity and avoid a cluttered display as discussed before. A node kit can completely be deleted as well by switching back to the main tab and delete the object there. This deletes all objects and closes the node kit tab. Figure 4.5 illustrates the layout and structure node kits use.

4.10 Deletion of Scene Objects

After creating scene objects, the user might decide to delete scene objects later on. In contrast to XIP Builder and MeVisLab (see chapter 2.12 for more details), deletion should work the same way for all objects displayed by the view. This applies to scene objects, parent/child connections and field connections

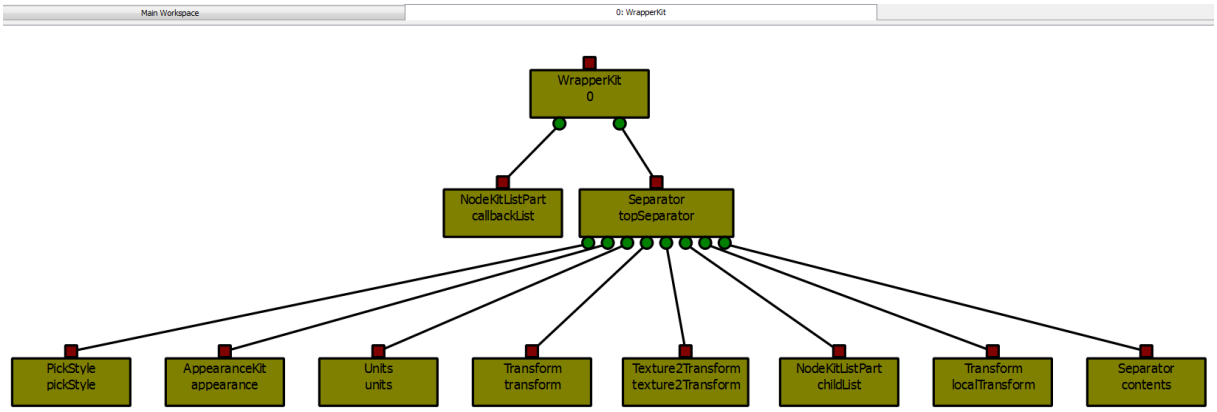


Figure 4.5: A example of a WrapperKit object. The tab bar can be seen on the very top showing the new node kit. The objects are evenly laid out. The main tab shows only one scene object instead of the entire node kit.

likewise. Furthermore, deletions should be possible to single scene objects, connection lines and field connection lines as well as arbitrary combinations of them.

Selection of arbitrary item combinations is already implemented in the default behavior of the Qt graphics framework. Items can be selected individually by simply clicking on them. Keeping the ctrl-button on the keyboard pressed allows the selection of multiple items, a functionality widely known from other programs and operating system. Multiple items can be selected by dragging a frame in which they are contained. Once the objects have been selected they can be deleted by pressing the delete button on the keyboard.

Conceptually, the implementation is not very complicated but a few things should be considered. Catching the delete key and getting the selected items can be done trough the following functions:

```
virtual void keyPressEvent ( QKeyEvent * keyEvent );
QList<QGraphicsItem *> selectedItems () const;
```

If a line is deleted, the parent-child connection has to be terminated, if a scene object is deleted all the connected lines have to be deleted and if a field connection is deleted both connected objects have to be notified, the connection has to be canceled and the item must be deleted. All items have to be removed from the scene by calling

```
void removeItem ( QGraphicsItem * item );
```

followed by a deletion of the object, as the scene hands over object deletion. The implementation however contains a few pitfalls, due to the optimized data structures which store direct pointers to objects for fastest possible access. Deleting single objects is done as described above but the deletion of selections is a bit more complicated. The objects returned by the selectedItems function are in no specific order. Performing the deletion in an arbitrary order can lead to access violations, as objects which are deleted may still be referenced by other objects. One could circumvent this problem by using one of the smart pointer classes which manage object deletion automatically through reference counting. The solution chosen is simple as well. By establishing a fixed order of deletion the problem can be solved as well. First, all lines that should be deleted are processed but not deleted yet. Notifying the objects connected through the line allows them to update their data structures. All lines processed this way are put on a deletion queue. In a second step all selected objects are deleted before the lines are finally processed.

4.11 Extension of Functionality During Run-Time

> No toolkit can ever be complete; users will always want to add something.*([Strauss and Carey, 1992])*

During section 4.2 loading of the scene objects from the scene graph library was discussed. By adding custom initialization code before the scene graph library is initialized, new scene objects can be found. This however requires the modification and recompilation of source code to work. In order to supply the user with more comfortable ways a solution which does not require a recompilation was implemented.

The fundamental idea of extensions without code modifications is to make use of DLLs. A DLL is a collection of source code which is linked dynamically during run-time. The opposite is static linking which is done statically during compile-time. Before making use of extension during run-time, the user has to compile the source code into a DLL file. To load the DLL file during run-time a user either supplies the DLL with a entry-point function or writes a custom initialization function which can be called by supplying the name of the function. When working with the same set of extensions on a frequent base it can be useful to automatically load these extensions on startup. This behavior can be configured through a dialog which appears when loading the DLL file. Internally all DLL files which should be loaded on startup are stored in a text file which contains the full path and the name of the file. It is thus also possible to manually add or delete entries by simple modifying the file with a text editor.

A problem faced during implementation was that the Quarter viewer and Coin3D do not communicate about updates of the scene graph library. A call to the

```
void SoDB::init ( void );
```

function which is used to initialize the scene graph library while rendering a scene at the same time caused crashes of the application. The workaround chosen looks as follows: First, rendering is stopped before loading extensions. Second, the Quarter viewer is closed. After that the new extension is loaded and the scene graph library is re-initialized. Finally, the Quarter viewer is reloaded and the old root is set to render the scene.

4.12 Scene Graph User Interface Prototyping

A scene graph can react to user interaction through the use of manipulators and draggers (see section 2.3.6). They can be used to directly influence the scene graph through field modification or hidden insertion of nodes which influence for example the transformation used before a node. Manipulators are part of the scene graph and interaction is performed directly in the viewer. To modify these interaction the scene graph containing the manipulators has to be modified. To take user interaction with the scene graph another step forward, a rapid user interface prototype was implemented.

The basic idea behind the implementation is the connection of two components which do not naturally fit together. The first part is the scene graph library, the second is a Qt-based user interface. Allowing to connect a user interface swiftly created with Qt Designer (see section 2.12.2) with a scene graph and to use different standard widgets to control the scene graph or modify certain values makes the creation of interactive scene graphs even faster.

To put these ideas into practice a flexible framework consisting of different components is required. The core component is a scene graph node called SoQTGuiLoader which communicates with the other components and makes sure changes in the Qt GUI influence the scene graph as specified by the user. This node is responsible to load and display one or more ui-files which contain the Qt user interface. Each widget contained in one of the ui-files is processed internally before the user interface is being shown. During initial processing one or more scene graph fields are created for each widget found in the

ui-file. This field represents the value of the widget. A QCheckBox for example is a widget which holds one binary value. A SoSFBool is a field in the Coin3D library which holds a boolean value. During the processing of a QCheckBox a SoSFBool field is created which is updated whenever the user changes the state of the QCheckBox. As there is a large number of widgets and the user can create new widgets as needed the processing of widgets needs to be very flexible and the software architecture must be extendible. All the fields created during loading of the GUI are stored in the SoQTGuiLoader class. As there might be multiple widgets of the same type, resulting in multiple fields of the same type, updating the correct field when the state of a widget changes is essential. That is why the fields are organized in associative containers. The key-value for this container is the absolute path of the ui-file followed by a "_", followed by the widgets name. Each widget within a ui-file is guaranteed to have a unique name. There must not be two files with the same filename within the same directory, making the absolute file path unique as well. This guarantees a unique key value which can be used to map a field to a widget.

After the initial processing has been done, all field values have been created and the GUI is finally shown to the user. Modifications to widgets must lead to an update of the fields value. This is done through so called Handler classes. The purpose of the Handler classes is to supply slots and some other methods which allow Qt widgets to connect. The Handler class keeps track of all the changes which happen through user interaction with the Qt-user interface. Every Handler is specifically designed for exactly one widget class and makes sure that it is only used by this class. One Handler class is used to handle all instances of the widget it was implemented for making use of the singleton pattern (see section 2.10.3). This flexible design is necessary to allow extensions with custom widgets. While the user interacts with the Qt interface and modifies widgets, all interaction is temporary stored in the respective Handler and applied to the fields during the next rendering pass. Multiple changes between two rendered frames overwrite each other and do not cause repeated field updates. The last modification thus overwrites all previous ones.

The processing and display of ui-files can be done during run-time. This enables to add multiple different Qt interfaces at the same time. In addition, even the ui-files themselves can be edited with Qt Designer during run-time. An automatic update mechanism is built in which detects if one of the shown ui-files has changed since the initial processing. If that is the case, all field connections to fields which correspond to one of the widgets in the changed ui-file are temporary saved. The GUI is closed and the file is reloaded. After the processing is finished, all previously set field connections are automatically restored. In case one of the connected widgets has been deleted during the modification the field connection is discarded.

For dynamic loading of ui-files Qt offers the QUiLoader class [Nokia Corporation, 2009k]. Finally, figure 4.6 gives an overview how the user interface prototyping tool is implemented.

4.13 File Format

Beside the Open Inventor file format, a custom file format which preserves the layout and offers more options can be used. The custom file format still contains the scene graph in Open Inventor format and allows other tools to load the scene graph. The extra information is added in commented XML format. This information allows Studierstube Builder to restore the position of items and other custom settings not supported natively in the Open Inventor file format. The additional information is added after the Open Inventor scene graph. During loading, the XML block needs to be extracted from the file and the comments need to be removed. This can be done with a couple of code lines. After that, a XML parser as it exists for many programming languages can be used to retrieve the structured information. This approach allows other programs to make use of the scene graph content stored in Open Inventor format without any modifications to the file. Other scene graph editors can easily write an importer for the rest of the information using the XML structure used. Listing 4.7 demonstrates the file format described so far by a simple example.

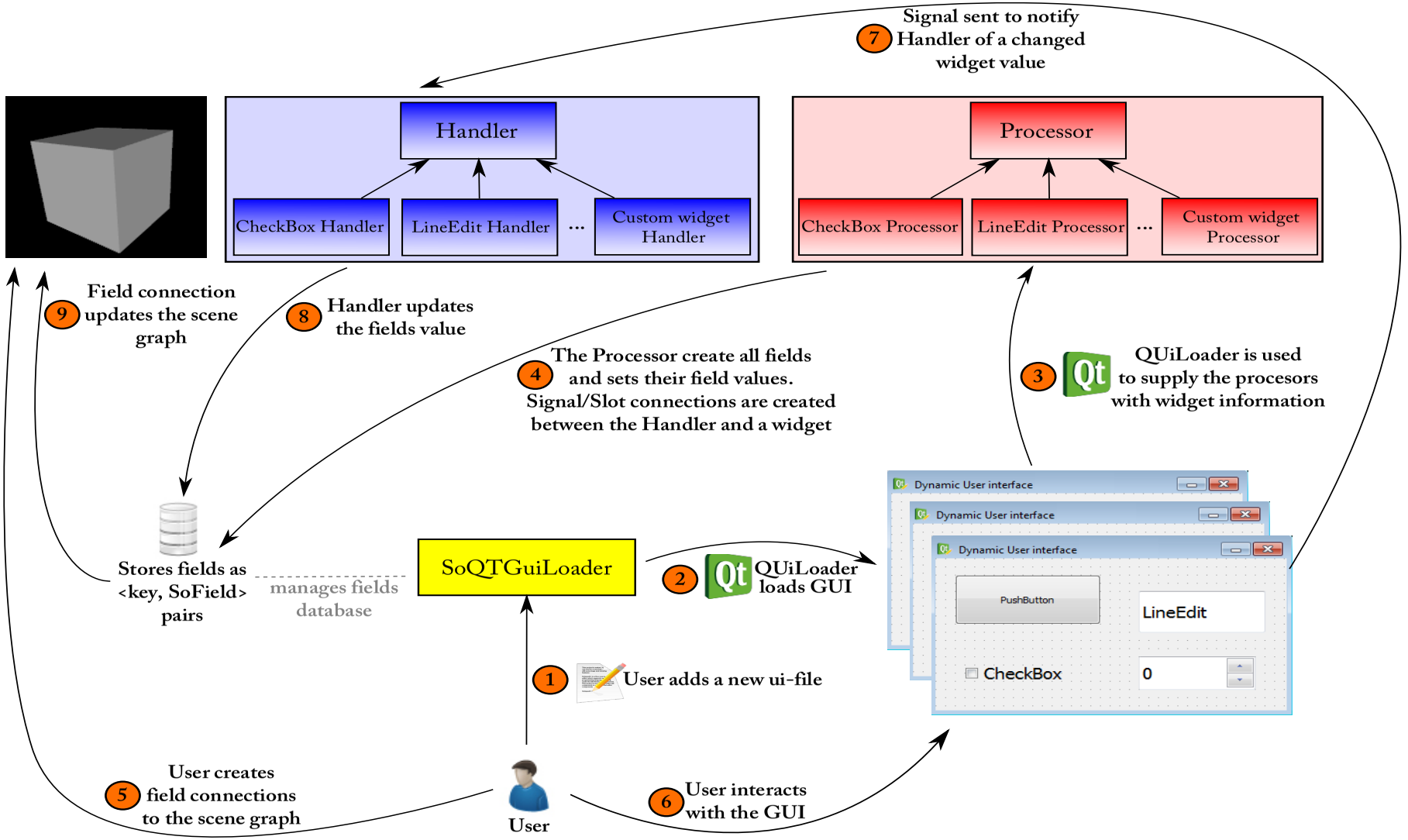


Figure 4.6: An overview how scene graph user interface prototyping is implemented. The first step (1) is triggered by the user when a new ui-file is added to the application. During the second step (2) the ui-file is loaded and shown to the user making use of the QUiLoader class Qt offers. The same class is used to supply the Processor classes with relevant information about the type and the respective value of widgets during the third step (3). Each Processor class maps the value of the widget to a field of a compatible type and stores the information together with a unique key (4). After that, the user can create field connections between one of the newly created fields and the scene graph by using the key specified (5). When the user interacts with the widgets (6), the modification of any value leads to an emitted Signal which is connected to a Slot in one of the Handler classes (7). The function called this way updates the field with the new value from the widget (8). This leads to an update of all previously connected slave fields (9).

```
<?xml version="1.0"?>
<StudierstubeBuilderScenegraphFormat>
    <GraphicalNodes>
        <Node>
            <UniqueId>0</UniqueId>
            <CustomName>A custom name</CustomName>
            <ObjectType>SoSeparator</ObjectType>
            <PositionX>100</PositionX>
            <PositionY>-100</PositionY>
        </Node>
    #...
    </GraphicalNodes>
    <Lines>
        <Line>
            <UniqueId>1</UniqueId>
            <StartObjectId>0</StartObjectId>
            <EndOjectId>1</EndOjectId>
            <EndOjectIndex>0</EndOjectIndex>
        </Line>
    #...
    </Lines>
    <FieldLines>
        <FieldLine>
            <UniqueId>2</UniqueId>
            <Text>timeOut(SFTime)->angle(MFFloat)</Text>
            <StartObjectId>0</StartObjectId>
            <EndOjectId>1</EndOjectId>
            <StartFieldOrEngineIndex>0</StartFieldOrEngineIndex>
            <EndFieldIndex>1</EndFieldIndex>
        </FieldLine>
    #...
    </FieldLines>
    <RenderRoot>
        <UniqueId>0</UniqueId>
    </RenderRoot>
</StudierstubeBuilderScenegraphFormat>
```

Listing 4.7: The XML file format used to store a scene graph. Each Node has a unique ID which is used to identify the object, a custom name which may be set by the user, an object type and a x,y position. A line which connects objects is defined by an unique ID, the unique ID where the line starts and ends as well as the position (=index) at which the line ends. A field connection line offers custom text and indices for the master and the slave field in addition to the information of a connection line. In addition the unique ID of the render root is stored.

To allow the modification of existing scene graphs an Open Inventor file format importer has been written. Importing scene graphs from Open Inventor files is something which is already supported in Coin3D, however, this only creates the scene graph. The real challenge of a custom importer is to rebuilt the graphical representation (scene objects, connection lines, field connection lines). The implementation relies on the import function of the scene graph trough a call to

```
SoSeparator * SoDB::readAll ( SoInput* in)
```

The Separator returned is the root node of the imported scene graph or NULL on error. Once imported, extraction of information is done in the following four steps:

1. Create a graphical scene object for each scene graph node with the respective type.
2. Connect the graphical scene objects as specified by the scene graph.
3. Create all graphical field connection lines by extracting information from the scene graph.
4. Lay out the graphical scene objects according to the scene graph structure and make sure no objects overlap each other.

All these steps are implemented in a very similar manner. Recursive algorithms traverse the scene graph and perform the necessary operations to Studierstube Builder. The layout algorithm is the only part whose implementation is a bit complex, as the layout has to work for a small number of objects as well as for very large and complex scene graphs. The chosen approach consists of two steps: First, recursively compute the width of the entire subgraph below the current node for each node and save the result. Second, recursively reposition all objects so that parent nodes are always centered above their children. Children are horizontally positioned at evenly distributed intervals, depending on the space requirement of each child's subgraph. The vertical distance between parent and child objects is set to a fixed distance. This algorithm works well for scene graphs tested. Note that the described procedure is simplified as node sharing (see section 2.3.4) requires a slight adaption to the recursive algorithms. Otherwise errors during loading are introduced as nodes are visited multiple times from different paths.

4.14 Studierstube Integration

After discussing large parts of the functionality which describe different part of a general scene graph editor, the focus in this section lies on how the support for the Studierstube project is done. For more details on the Studierstube project see [Schmalstieg et al., 2002]. The integration of Studierstube requires only little code making up a negligible fraction of the entire code written for the application. The first design decision was therefore to allow the user to compile the source code with and without Studierstube integration, as not all users will want Studierstube support. This is done through the use of #ifdef preprocessor macros for the fraction of code which integrates the Studierstube source code into the rest of the application. A user who is not using Studierstube will therefore not suffer from lower performance.

The Studierstube configuration of a project however is not only about the actual content of the scene but also about other parts like different tracking engines, video background and the viewer. These parts are all set as nodes in the scene graph. The scene which is edited during run-time by the user is therefore just one part of the entire scene graph which is created by Studierstube. In addition, the entire configuration of these parts is done using different textual configuration files. These configuration files are read in when Studierstube is started and are not touched until Studierstube is closed. In general almost all the configuration of Studierstube is done during startup. Run-time modifications are not foreseen in large parts of the design. Studierstube starts in its own thread showing a custom, configurable viewer with an own main loop. This architecture is kept this way. When Studierstube is chosen to be integrated, it is

started in an own thread with the configured viewer. While the rest of the configuration is untouched the scene graph can be modified from within Studierstube Builder, changing only the parts of the scene graph which holds the actual scene. Thus, this only influences a subgraph of the hidden rest of the scene graph.

Studierstube's viewer settings are loaded from a configuration file but run-time modification is implemented. The properties of the viewer are stored in fields. To detect changes made by the user a field sensor is attached to each field, calling a callback function which makes the changes to the viewer at run-time. To modify the configuration files a built in text editor can be used. It does not allow any advanced features but enables the viewing and modification of the kernel.xml and the opentracker.xml configuration. Changes to the files are however not reloaded until Studierstube is restarted.

Chapter 5

Usage Examples

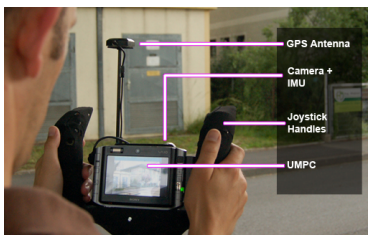
This chapter focuses on application areas where Studierstube Builder can improve the state of the art. To demonstrate the usefulness, three examples from different fields were chosen. Section 5.1 introduces the Vidente project and demonstrates how content can be added to existing data. Section 5.2 shows how XIP nodes can be imported and how the new nodes can be connected to display a medical dataset. Finally, section 5.3 presents how Studierstube Builder can be used to ease and improve education.

5.1 Vidente Online Content Creator

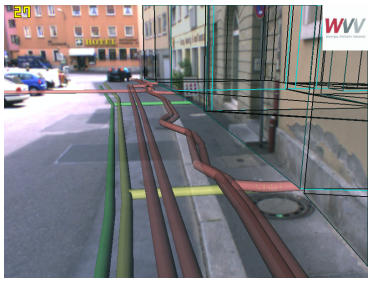
Vidente is a augmented reality project started as an collaboration between Graz University of Technology and GRINTEC GmbH[1]. Vidente uses mobile augmented reality hardware in combination with geospatial information to display underground information [Schall et al., 2009] such as telecommunication cables or gas pipes. This information is displayed by merging images taken from a video camera with computer graphics generated content (also called see-through metaphor). Field workers can use this information to alleviate their work and improve efficiency. Figure 5.1 gives an overview.

To allow field workers to use augmented reality devices on-sight, it must be reasonable small while supplying enough battery capacity and performance for the task [Schall et al., 2009]. Vidente uses a ultra mobile computer mounted on a handheld frame, called Vesp'R. The frame holds the camera and controls for interaction with the computer. Figure 5.1a shows the input device. The device obtains its position trough GPS. EGNOS [Gauthier et al., 2001] is used to further increase accuracy up to a few meters. Orientation is measured using a three degree of freedom (3DOF) sensor.

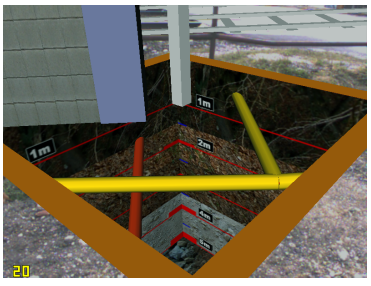
[1] http://www.grintec.com



(a) AR device



(b) AR view



(c) AR excavation

Figure 5.1: Some images from the Vidente project. Figure 5.1a shows the device used. Figure 5.1b illustrates how computer generated information is merged with a video stream. A computer generated excavation whole is shown in figure 5.1c.

Geospatial information was built using hundred of person years [Mendez et al., 2008]. This information is usually stored in different databases. The process to convert this database information into a format which can be rendered by a computer is called transcoding. This process is complex. To tackle loss of information during transcoding, additional semantic information is stored. The result of the transcoding process is a scene graph marked up with semantic attributes. The scene graph can be used for rendering together with the video background supplied by the built in camera (see figure 5.1b and 5.1c). The scene graph is automatically created from information in geospatial databases and stored in Open Inventor files from which the scene graph can be loaded.

The following example demonstrates the use of Studierstube Builder with geospatial data supplied by the Austrian railroad company ÖBB. ÖBB is frequently building sound insulating walls around newly built or existing railways. During the building of these walls holes have to be drilled in the ground to hold the newly built walls. While drilling, care has to be taken not to terminate existing existing underground cables and pipes (telecommunication, electric cable, water pipes, ...). Vidente is used to display extra information during on-sight inspection. In addition to the display of an automatically generated scene graph, new primitives such as walls and valves can be added as well. This new information can be added on-sight making use of a laptop and Studierstube Builder to quickly modify the scene graph. Changes can therefore instantly be checked on-sight using Vidente. This can create a quick feedback during on-sight planning. As the geographic position of existing scene graph objects is known one could also implement new functionality which transcodes the newly added scene graph objects back into the geospatial database. Figure 5.2 demonstrates these ideas.

5.2 Medical Dataset Investigator

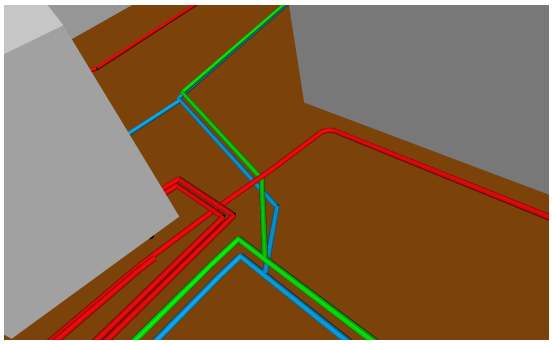
As Studierstube Builder can be extended during run-time, one can write custom extensions and connect them to scene graphs as it is done with built in types. This example demonstrates the extendibility of Studierstube Builder by using XIP nodes to create a medical dataset investigator.

As mentioned during section 2.9.3, XIP is a library developed by Siemens Corporate Research under an open source license with a strong focus on the medical domain. It wraps different other libraries such as VTK and ITK into scene graph objects. XIP compiles into a number of shared libraries which can be loaded by Studierstube Builder to make use of the XIP functionality. During loading of the shared libraries, XIP nodes are registered in the Coin3D database and can be used the same way as built-in scene graph objects.

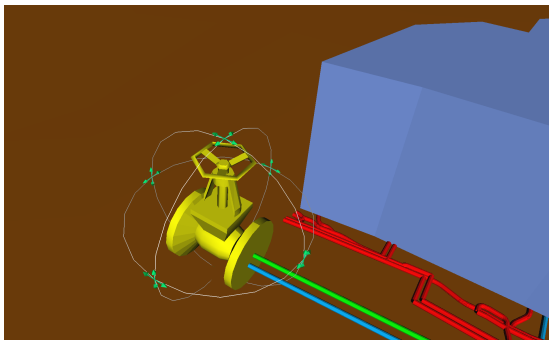
The first example shows a three-dimensional computer tomography (CT) scan of a human foot. The dataset is delivered together with XIP Builder and can be found online at [Siemens Corporate Research, 2007]. Using some of the built-in VTK nodes, one can quickly compose a scene graph which shows a three-dimensional volume of the CT dataset. A ray caster is then used to deliver a three-dimensional conception of the dataset. Different transfer function can be used to alter the visual appearance. By using VTK nodes for rendering of three-dimensional datasets the user has an additional option to render volumetric datasets. When using Coin3D, one could for example use SIM Voleon [Kongsberg SIM AS., 2009d] for volume rendering. XIP's VTK integration offers a total of 56 new nodes which can be used for different visualization tasks.

Medical data can also be altered through the use of XIP's ITK library. ITK is a library for image segmentation and processing. XIP offers almost 200 different ITK nodes. The ITK nodes can be used to load image data from different formats and perform different filter operations such as segmentation or registration on the dataset. This way a processing pipeline can be created to alter the visual appearance of a dataset.

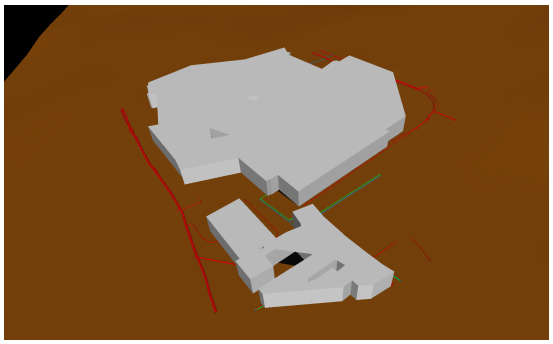
Figure 5.3 demonstrates these ideas.



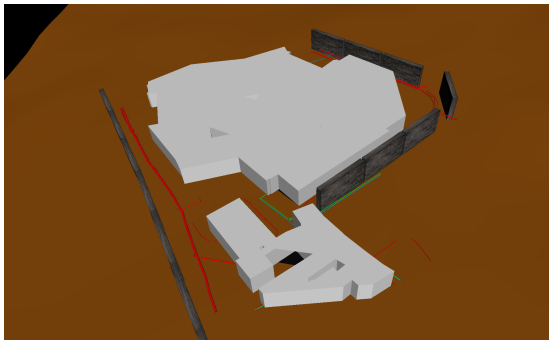
(a) Different pipes and cables



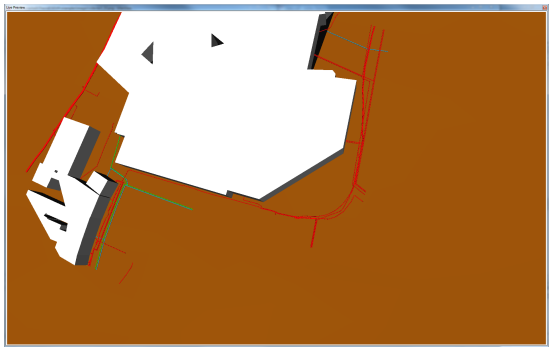
(b) Added valve



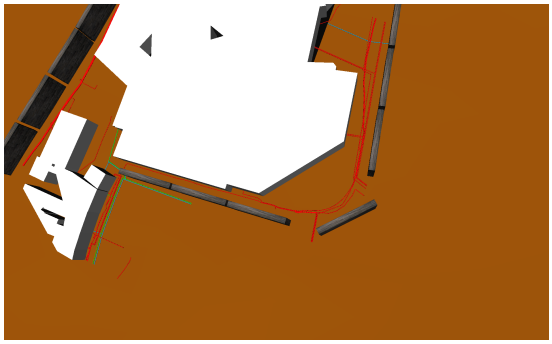
(c) Automatically extracted data 1



(d) Added noise protection walls 1



(e) Automatically extracted data 2



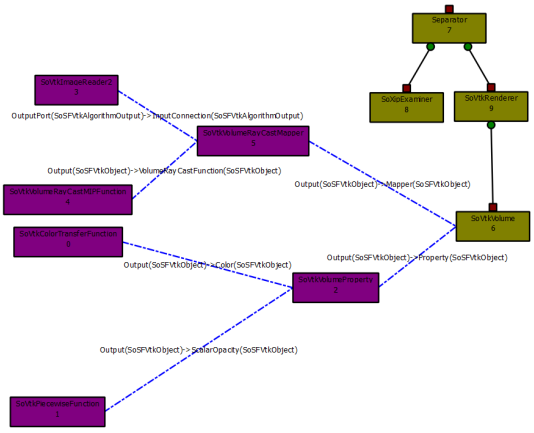
(f) Added noise protection walls 2

Figure 5.2: Different screen shots illustrating the use of Studierstube Builder for the Vidente project. Figure 5.2a shows the network of pipes which is automatically extracted from a geospatial database. The color was added later for a better distinction of electric cables (red), water pipes (blue) and gas pipes (green). Figure 5.2c shows an overview of the entire data present. Importing the automatically extracted data into Studierstube Builder one can quickly modify the scene graph. The original data does not contain colors for example. So the ground has been colored brownish and the pipes use different colors to automatically distinguish their type. For planning purposes one can for example quickly add noise protection walls around the building without interfering with the network in the ground. Figure 5.2d shows the same scene with noise blocking walls added for illustration. Figure 5.2e and 5.2f show the same scene with and without noise protection walls from different perspectives. Figure 5.2b illustrates how a valve is added belated by the user. The additional information has been added in approximately half an hour.

(a) CT scan of a human foot 1

(b) CT scan of a human foot 2

(c) CT scan of a human foot 3



(d) Scene graph for the human foot example

Figure 5.3: The CT scan of the human foot after extending Studierstube Builder with the XIP libraries during run-time is shown in figure 5.3a, 5.3b and 5.3c. Volume rendering is done using VTK nodes for importing the three-dimensional dataset and visualization using a ray caster. The resulting scene graph is very compact and consists only of a few nodes. See figure 5.3d for an overview of the resulting scene graph.

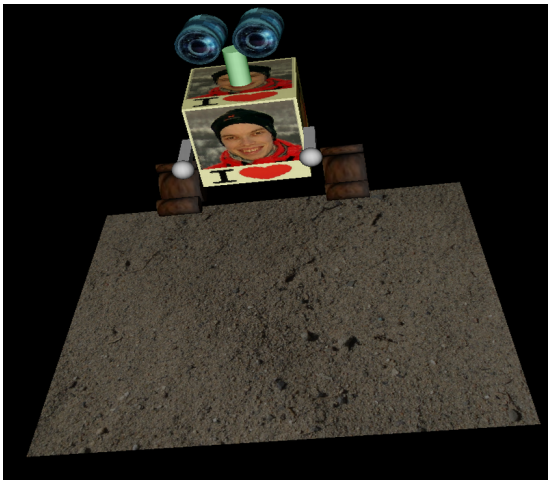
5.3 Educational Use

Studierstube Builder is also suitable for courses which require usage of scene graphs. During the year 2009 a lecture called ”Virtual Reality” was held at Graz University of Technology. During the course work, students have to hand in a number of assignments. These assignments make use of Studierstube to develop virtual reality applications. The computer graphics part of the assignments is solved using Coin3D scene graphs. The helpfulness of Studierstube Builder for educational use is shown using one of the assignments which was given out to students throughout the semester. The problem is defined as follows (translated from German):

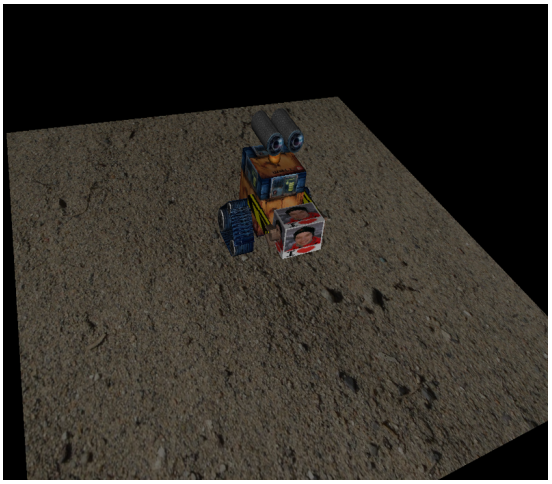
> Create a file scene.iv that defines a small scene. Create a virtual floor on top of the marker defined during task1 using either a IndexedFaceSet or QuadMesh node. Create a small creature, a robot, or an alien standing on the floor. The creature’s geometry should be defined through the use of geometric primitives supplied by Open Inventor (SoCube, SoSphere, SoCylinder, SoCone). Redundant parts of the creature should make use of instance sharing (DEF / USE of nodes). Different parts of the creature should use different colors. Use a photo of yourself and place it somewhere in the scene. *([Schmalstieg, 2009])*

The assignment was solved twice. Once manually writing the scene.iv file, loading the file with a viewer from time to time to check the contents and a second time with Studierstube Builder. For simplicity the focus lies purely on the robot created this way and not on Studierstube integration as this is done within minutes while the robot creation takes considerably longer. The two versions were created with a time lag of approximately half a year without checking the previously solution. Writing the Open Inventor file manually took approximately ten hours of whom the greatest part was used for creation and positioning of geometric primitives and only a small part was used to find and insert suitable textures.

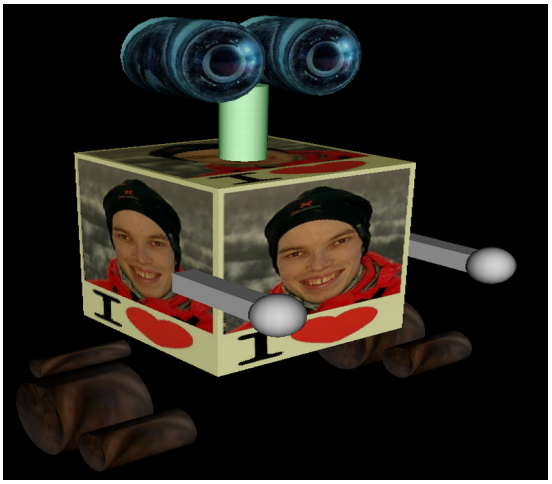
The creation of approximately the same robot with Studierstube Builder took less than two hours including a more extensive texture search. The robot created this way looks better than the original as more effort can be put into details. See figure 5.4 for a comparison between the two versions. Scene graphs can grow rapidly even for small examples. The scene graph structure for the examples shown in figure 5.4b and 5.4d can be seen in figure 5.5.



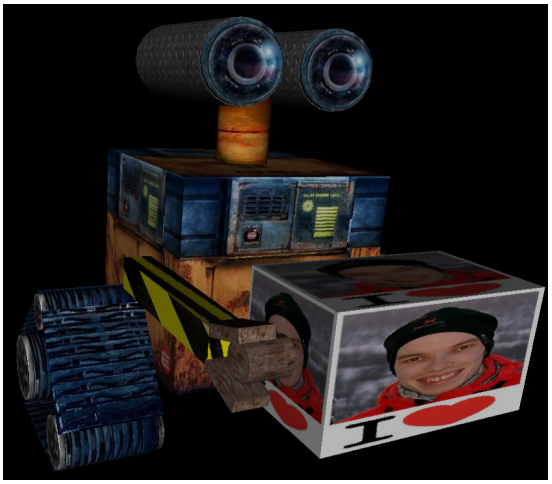
(a) Robot created through an iv-file 1



(b) Robot created with Studierstube Builder 1



(c) Robot created through an iv-file 2



(d) Robot created with Studierstube Builder 2

Figure 5.4: The Virtual Reality course assignment solved twice. The robot is heavily based on Wall-E, a film produced by Pixar. Figure 5.4a and 5.4c show the version manually created by writing an iv-file. The entire manual creation process took about ten hours. Figure 5.4b and 5.4d show the version created using Studierstube Builder within less than two hours. The creation of both robots is based on the same set of Wall-E pictures. Studierstube Builder allows much fast creation as syntax does not have to be remembered and all changes are immediately reflected in the live preview. Although only a fraction of the time has been used, results with Studierstube Builder assemble the original much closer than the manually created robot. Compare for example figure 5.4c and 5.4d

.

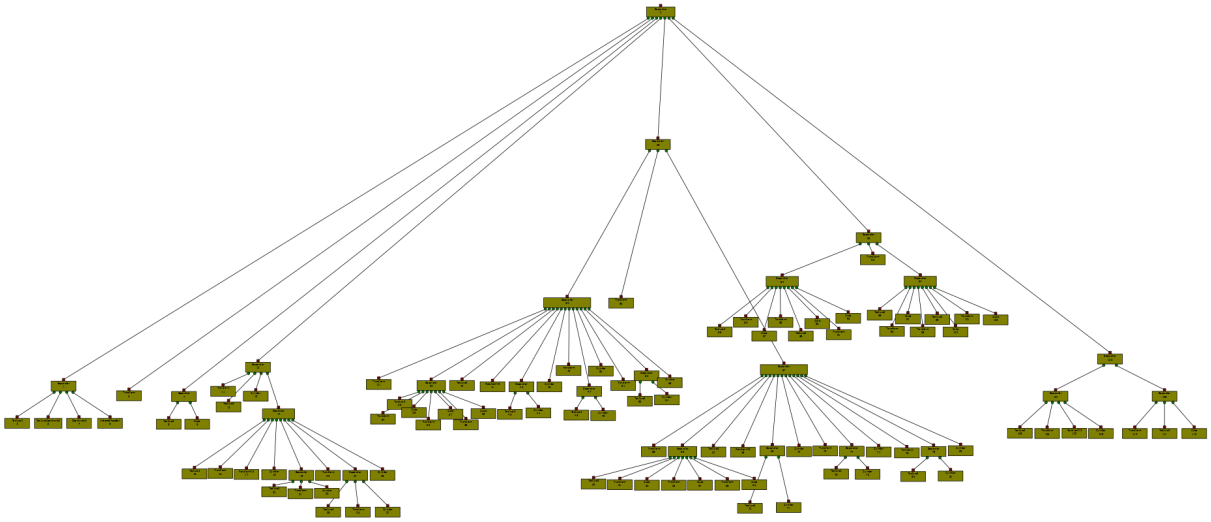


Figure 5.5: The scene graph structure for the robot example. Note how complex a scene graph can grow, even for relatively simple examples.

Chapter 6

Conclusions

In this thesis we have presented Studierstube Builder, a rapid application, rapid prototyping tool to create scene graphs. Scene graphs are a widely accepted concept to accelerate computer graphics development. A wide range of applications makes use of scene graphs among others, augmented reality applications. With the Studierstube augmented reality project being developed at our university, we have a solution for all aspects of virtual and augmented reality applications at hand. With the recent progress for mobile devices, augmented reality applications are on the rise. Studierstube ES, a complete re-development of Studierstube, was specifically designed for mobile phones as a response to this development. However, the creation of scene graphs as the underlying structure for most of these applications is still cumbersome as few tools exist to accelerate this process. None of the existing tools came close enough to the ideas we had in mind or an extensions was not possible due to the licensing terms.

Therefore, development of Studierstube Builder was started with the idea to create an extendible, free, well documented, multi-platform scene graph editor which supports the reuse of existing material and which integrates into Studierstube to accelerate application development. As most of the work done at our university uses Coin3D and existing material should be reused, we built our editor on Coin3D, which is fully compatible with Open Inventor. The development was done in C++. Platform independence is reached through the use of the well documented Qt framework. Qt is used mainly for the creation of Studierstube Builder's user interface. The software design is built on well established design patterns to allow the greatest flexibility during later extensions.

Studierstube Builder allows the swift creation and modification of scene graphs during run-time. Each scene graph node is displayed as a graphical object on a workspace with which the user can interact. Objects can be created and connected with each other by using drag and drop operations on the workspace. During operations a preview of the currently performed action is shown where appropriate. The properties of a scene object can be altered by modifying its field values through usage of convenient input widgets. All modifications are immediately reflected in a preview showing the entire scene graph or only a sub-graph. The scene objects are dynamically extracted from the scene graph library on each start, allowing user defined extensions to be loaded on startup. Field connections can be made between objects through drag and drop. A dialogue makes sure only compatible fields can be connected. Multiple objects can be selected, moved, and deleted. Studierstube Builder allows to extend the scene graph library with new nodes during run-time. This can be done by loading a dynamic library with the new content. The user can decide to automatically load a number of dynamic libraries on startup. Scene graphs can be saved in Open Inventor file format or in a custom file format using a XML syntax to store information. The user can use a zoom function on the workspace.

To our knowledge, Studierstube Builder offers a number of unique features not present in any other scene graph editor. First, a custom node has been developed which supports the connection of fields with one or more Qt widgets at run-time. This way user interfaces can quickly be prototyped and the scene graph can be modified through the use of Qt widgets. It is even possible to modify the user

interface used to control the scene graph during run-time without a need to restart. Second, node kits can be edited as well. Each node kit is shown in a separate tab. Node kit children and their field values can be edited as usual. Third, Studierstube is integrated. That means that scene graphs edited within Studierstube Builder can be modified and set as root of Studierstube, with the nodes being automatically shown. Configuration files can also be edited within Studierstube Builder, but changes are not applied until a restart is performed. Studierstube integration can be turned of during compilation if support is not required. The usefulness of Studierstube Builder is demonstrated in chapter 5 by implementing examples from different domains.

No question, there is much which could be done to improve our scene graph editor. Talking about functionality, we would like to start a beta test to get feedback about reliability, compatibility issues and bugs. Without real world usage by at least a small number of users it is hard to tackle these issues. New functionality could be added as well. Import functions from other file formats is for example desirable. We would also like to have an additional GUI element which displays the current scene graph structure in a tree view. A multi tabbed interface which allows editing multiple scene graphs at the same time would be great. The foundation for such a feature exist as node kits already use a tabbed interface but the functionality needs to be implemented. For more complex scenes, a way to further abstract or group information could improve the overview. Sub graphs could be abstracted by only one graphical object which extends on request. The field editor could benefit from more custom widgets and multiple different options to modify a fields value. Custom dialogs instead of widgets could be used as well to further improve field modifications. In addition, one could implement a function which creates C++ source code equivalent to the currently viewed scene graph. Some basic features not present so far are copy/paste and an undo option. Those feature should be contained in the next version.

The user interface could be improved in many ways. Foremost, we would like to perform a user study to get feedback about possible improvements and current problems. We would also like to have different sorting of the scene objects. The current alphabetical list is just one way but we believe that a structured representation could be beneficial. We are also not sure if the chosen icons are clear enough. Some of them were created by ourselves and can surely be improved. We could also implement more of GUI design patterns presented in Tidwell [2005]. We are also aware that the color matching between different types of graphical items is not very well. With the help of a designer the optical appearance could certainly be improved.

Bibliography

Aguado, T. and Eparado, J. (2005). Coindesigner: Un entorno de desarrollo rápido para escenas openinventor y vrml. Master's thesis, Escuela Superior de Ciencias Experimentales y Tecnología. Universidad Rey Juan Carlo. (Cited on page 24.)

Akenine-Möller, T., Haines, E., and Hoffman, N. (2008). *Real-Time Rendering 3rd Edition*. A. K. Peters, Ltd., Natick, MA, USA. (Cited on page 18.)

Andrews, K. (2009). Writing a thesis: Guidelines for writing a master's thesis in computer science. Graz University of Technology, Austria. (Cited on page ix.)

Angel, E. (2008). *Interactive Computer Graphics: A Top-Down Approach Using OpenGL (5th Edition)*. Addison Wesley, 5 edition. (Cited on page 15.)

Autodesk (2009). Dxf reference. `http://images.autodesk.com/adsk/files/acad_dxf1.pdf`. (Cited on page 20.)

Azuma, R. T. (1997). A survey of augmented reality. *Presence*, 6:355–385. (Cited on page 2.)

Bale, K. and Chapman, P. (2007). Scenegraph technologies: A review. `http://www.dcs.hull.ac.uk/people/csspmc/docs/scenegraphreview.pdf`. (Cited on pages 7, 9, 11, 18 and 19.)

Barczok, A., Himmelein, G., and König, P. (2009). Mit dem Dritten sieht man besser - Augmented Reality: Computer-unterstützter Blick in die Welt. *c't magazin für computer technik*, 20:122–129. (Cited on page 2.)

Black, P. (2004). acyclic graph. Dictionary of Algorithms and Data Structures. `http://www.itl.nist.gov/div897/sqg/dads/HTML/acyclicgraph.html`. (Cited on pages 9 and 103.)

Bögeholz, H. (2009). Norweger messer. *c't magazin*, 15:186–191. (Cited on page 41.)

Bray, T., Paoli, J., and Sperberg-McQueen, C. (1998). Extensible markup language (XML) 1.0. Technical report, W3C. (Cited on page 19.)

Bruski, P. (2008). The java (not really) faster than c++ benchmark. `http://bruscy.republika.pl/pages/przemek/java_not_really_faster_than_cpp.html`. (Cited on page 55.)

Carey, R. and Bell, G. (1997). *The annotated VRML 2.0 reference manual*. Addison-Wesley Longman Ltd., Essex, UK. (Cited on page 11.)

Cormen, T., Leiserson, C., Rivest, R., and Stein, C. (2001). *Introduction to Algorithms*. MIT Press and McGraw-Hill. (Cited on page 11.)

Das, T. K., Singh, G., Mitchell, A., Kumar, P. S., and McGee, K. (1997). Neteffect: a network architecture for large-scale multi-user virtual worlds. In *VRST '97: Proceedings of the ACM symposium on Virtual reality software and technology*, pages 157–163. (Cited on page 16.)

Drichel, A. (2008). Depth first tree. http://en.wikipedia.org/wiki/File:Depth-first-tree.svg. (Cited on page 12.)

Fahmy, T. (2006). Pivy - embedding a dynamic scripting language into a scene graph library. Master's thesis, Vienna University of Technology. (Cited on page 55.)

Faulhaber, O. (2006). Anifun3 - the scene graph editor for java 3d. http://www.anifun3.de/index.html. (Cited on page 27.)

Fernando, R. and Kilgard, M. J. (2003). *The Cg Tutorial: The Definitive Guide to Programmable Real-Time Graphics*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA. (Cited on page 19.)

Flanagan, D., Farley, J., Crawford, W., and Magnusson, K. (1999). *Java enterprise in a nutshell: a desktop quick reference*. O'Reilly & Associates, Inc., Sebastopol, CA, USA. (Cited on page 55.)

Fowler, M. (2002). *Patterns of Enterprise Application Architecture*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA. (Cited on page 29.)

Freeman, E., Freeman, E., Bates, B., and Sierra, K. (2004). *Head First Design Patterns*. O' Reilly & Associates, Inc. (Cited on pages 29, 30 and 31.)

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995). *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Professional. (Cited on pages 13, 29 and 30.)

Gauthier, L., Michel, P., Ventura-Traveset, J., and Benedicto, J. (2001). EGNOS: the first step in Europe's contribution to the Global Navigation Satellite System. *ESA Bulletin*, 105:35–42. (Cited on page 85.)

Gidén, V., Moeller, T., Ljung, P., and Paladini, G. (2008). Scene graph-based construction of cuda kernel pipelines for xip. http://www.cse.buffalo.edu/hpmiccai/pdf/HPMICCAI2008-V3.pdf. (Cited on page 23.)

Goyvaerts, J. and Levithan, S. (2009). *Regular Expressions Cookbook*. O'Reilley Media Inc. (Cited on page 76.)

Hesina, G., Schmalstieg, D., Furhmann, A., and Purgathofer, W. (1999). Distributed Open Inventor: a practical approach to distributed 3D graphics. In *VRST '99: Proceedings of the ACM symposium on Virtual reality software and technology*, pages 74–81. http://www.cg.tuwien.ac.at/research/publications/1999/Hes-1999-/TR-186-2-99-15Paper.pdf. (Cited on pages 16 and 17.)

Hewlett-Packard (1991). Sarbase graphics techniques and display list programmer's guide. (Cited on page 10.)

Ibanez, L., Schroeder, W., Ng, L., and Cates, J. (2003). *The ITK Software Guide*. Kitware, Inc. ISBN 1-930934-10-6, http://www.itk.org/ItkSoftwareGuide.pdf, first edition. (Cited on page 22.)

Kersting, O. and Döllner, J. (2002). Interactive 3D visualization of vector data in GIS. In *GIS '02: Proceedings of the 10th ACM international symposium on Advances in geographic information systems*, pages 107–112, New York, NY, USA. ACM. (Cited on page 1.)

Kessenich, J., Baldwin, D., and Rost, R. (2009). The opengl® shading language. http://www.opengl.org/registry/doc/GLSLangSpec.1.50.09.pdf. (Cited on page 23.)

Kitware Incorporation (2009). Cmake:about. http://www.cmake.org/cmake/project/about.html. (Cited on page 57.)

Kongsberg SIM AS (2009). Coin documentation. http://doc.coin3d.org/Coin/. (Cited on page 73.)

Kongsberg SIM AS. (2009a). Coin3d. http://coin3d.org/. (Cited on page 18.)

Kongsberg SIM AS. (2009b). http://doc.coin3d.org/soqt/. http://doc.coin3d.org/SoQt/. (Cited on page 56.)

Kongsberg SIM AS. (2009c). Quarter documentation. http://doc.coin3d.org/Quarter/. (Cited on page 56.)

Kongsberg SIM AS. (2009d). Simvoleon documentation. http://doc.coin3d.org/SIMVoleon/. (Cited on pages 19 and 86.)

Kuehne, B. and Martz, P. (2007). *OpenSceneGraph Reference Manual v2.2*. Blue Newt Software and Skew Matrix Software. (Cited on page 18.)

Levinson, E. (1998). The mime multipart/related content-type. (Cited on page 63.)

Lindholm, T. and Yellin, F. (1999). *The Java Virtual Machine Specification*. Addison-Wesley Longman, Amsterdam, 2 edition. (Cited on page 55.)

MacIntyre, B. and Feiner, S. (1998). A distributed 3D graphics library. In *SIGGRAPH '98: Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, pages 361–370. (Cited on pages 16 and 17.)

Martin, K. and Hoffman, B. (2003). *Mastering CMake: A Cross-Platform Build System*. Kitware Inc. (Cited on page 57.)

Mendez, E., Schall, G., Havemann, S., Junghanns, S., Fellner, D., and Schmalstieg, D. (2008). Generating semantic 3d models of underground infrastructure. *IEEE Comput. Graph. Appl.*, 28(3):48–57. (Cited on page 86.)

MeVis Medical Solutions AG (2009a). Mevislab - medical image processing and visualization. http://www.mevislab.de/. (Cited on page 23.)

MeVis Medical Solutions AG (2009b). *MeVisLab Reference Manual - Reference Guide for the MeVisLab GUI*. MeVis Medical Solutions AG, Universitätsallee 29, 28359 Bremen. (Cited on page 24.)

Meyers, S. (2005). *Effective C++: 55 Specific Ways to Improve Your Programs and Designs (3rd Edition)*. Addison-Wesley Professional. (Cited on page 18.)

Microsoft Developer Network (2009). Windows api reference. http://msdn.microsoft.com/en-us/library/aa383749(VS.85).aspx. (Cited on page 19.)

Musser, D. R. and Saini, A. (1995). *The STL Tutorial and Reference Guide: C++ Programming with the Standard Template Library*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA. (Cited on page 56.)

Neider, J. and Davis, T. (1993). *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Release 1*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA. (Cited on page 10.)

Nokia Corporation (2009a). Drag and drop. http://doc.trolltech.com/4.5/dnd.html. (Cited on page 64.)

Nokia Corporation (2009b). Generic containers. http://qt.nokia.com/doc/4.5/containers.html. (Cited on page 68.)

Nokia Corporation (2009c). The graphics view framework. http://doc.trolltech.com/4.5/graphicsview.html. (Cited on page 65.)

Nokia Corporation (2009d). An introduction to model/view programming. http://doc.trolltech.com/4.5/model-view-introduction.html. (Cited on pages 63 and 73.)

Nokia Corporation (2009e). Model classes. http://doc.trolltech.com/4.5/model-view-model.html. (Cited on page 75.)

Nokia Corporation (2009f). qmake manual. http://qt.nokia.com/doc/4.5/qmake-manual.html. (Cited on page 57.)

Nokia Corporation (2009g). Qmimedata class reference. http://doc.trolltech.com/4.5/qmimedata.html. (Cited on page 63.)

Nokia Corporation (2009h). Qt - a cross-platform application and ui framework. http://qt.nokia.com/. (Cited on pages 19 and 56.)

Nokia Corporation (2009i). Qt designer manual. http://doc.trolltech.com/4.5/designer-manual.html. (Cited on page 41.)

Nokia Corporation (2009j). Qt licensing. http://qt.nokia.com/products/licensing. (Cited on page 56.)

Nokia Corporation (2009k). Qt's classes. http://doc.trolltech.com/4.5/classes.html. (Cited on pages 63, 66, 68 and 80.)

Nokia Corporation (2009l). User interface compiler (uic). http://doc.trolltech.com/4.5/uic.html. (Cited on page 62.)

Nokia Corporation (2009m). Using a designer .ui file in your application. http://doc.trolltech.com/4.5/designer-using-a-ui-file.html. (Cited on page 61.)

NVIDIA Corporation (2009a). Nvidia cuda - reference manual. http://developer.download.nvidia.com/compute/cuda/2_3/toolkit/docs/CUDA_Reference_Manual_2.3.pdf. (Cited on page 23.)

NVIDIA Corporation (2009b). NVSG SDK Details. http://developer.nvidia.com/object/nvsg_details.html. (Cited on page 19.)

NVIDIA Corporation (2009c). Tegra 600 series. http://www.nvidia.com/object/product_tegra_600_us.html. (Cited on page 2.)

Paladini, G. (2007). Xip imaging tools. https://cabig.nci.nih.gov/workspaces/Imaging/Meetings/FacetoFace/Oct_2007/XIP_Imaging_Status_Paladini.pdf. (Cited on pages 22 and 23.)

ParallelGraphics (2009). Vrmlpad 3.0. http://www.parallelgraphics.com/products/vrmlpad/. (Cited on page 22.)

Park, D., Waters, R., Anderson, D., Barrus, J., Brogan, D., Mckeown, S., Nitta, T., Sterns, I., and Yerazunis, W. (1996). Diamond park and spline: A social virtual reality system with 3d animation, spoken interaction, and runtime modifiability. (Cited on page 16.)

Pereira, F. C. and Ebrahimi, T. (2002). *The MPEG-4 Book*. Prentice Hall PTR, Upper Saddle River, NJ, USA. (Cited on page 1.)

Postel, J. (1980). User datagram protocol. http://tools.ietf.org/pdf/rfc768.pdf. (Cited on page 17.)

Reiners, D., Voß, G., and Behr, J. (2002). Opensg: Basic concepts. In *In 1. OpenSG Symposium OpenSG*. (Cited on page 16.)

Reinholtz, K. (2000). Java will be faster than c++. *SIGPLAN Not.*, 35(2):25–28. (Cited on page 55.)

Reitmayr, G. and Schmalstieg, D. (2001). Mobile collaborative augmented reality. *Augmented Reality, International Symposium on*, 0:114. (Cited on pages 2, 4 and 17.)

Rendersoft Software (2009). Rendersoft vrml editor. http://pachome2.pacific.net.sg/~jupboo/. (Cited on page 22.)

Rohlf, J. and Helman, J. (1994). Iris performer: a high performance multiprocessing toolkit for real-time 3d graphics. In *SIGGRAPH '94: Proceedings of the 21st annual conference on Computer graphics and interactive techniques*, pages 381–394. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.105.7217&rep=rep1&type=pdf. (Cited on pages 10, 15, 17 and 18.)

Roth, M., Voss, G., and Reiners, D. (2004). Multi-threading and clustering for scene graph systems. *Computers and Graphics*, 28:63–66. (Cited on page 16.)

Rothfarb, R. (1998). Cosmo worlds - world building tutorial. http://www.hiddenline.com/wireframe/word/grotto/cosmoworlds-tutorial.html. (Cited on page 24.)

Schall, G., Mendez, E., Kruijff, E., Veas, E., Junghanns, S., Reitinger, B., and Schmalstieg, D. (2009). Handheld augmented reality for underground infrastructure visualization. *Personal Ubiquitous Comput.*, 13(4):281–291. (Cited on page 85.)

Schmalstieg, D. (2009). Aufgabe 2: Open inventor. http://www.icg.tu-graz.ac.at/courses/vr/lu/VRVU2009_aufgabe2.pdf. (Cited on page 89.)

Schmalstieg, D., Fuhrmann, A., Hesina, G., Szalavári, Z., Encarnaçäo, L. M., Gervautz, M., and Purgathofer, W. (2002). The studierstube augmented reality project. *Presence: Teleoper. Virtual Environ.*, 11(1):33–54. (Cited on pages 3, 17 and 83.)

Schmalstieg, D. and Wagner, D. (2007). Experiences with handheld augmented reality. In *ISMAR '07: Proceedings of the 2007 6th IEEE and ACM International Symposium on Mixed and Augmented Reality*, pages 1–13, Washington, DC, USA. IEEE Computer Society. (Cited on page 4.)

Schroeder, W., Martin, K., and Lorensen, B. (2007). *The Visualization Toolkit, Third Edition*. Kitware Inc. (Cited on page 22.)

Shaw, C. and Green, M. (1993). The MR Toolkit Peers Package and Experiment. In *In IEEE Virtual Reality Annual International Symposium (VRAIS 93*, pages 463–469. IEEE. (Cited on page 16.)

Shneiderman, B. and Plaisant, C. (2009). *Designing the User Interface: Strategies for Effective Human-Computer Interaction (4th Edition)*. Pearson Addison Wesley. (Cited on page 31.)

Siemens Corporate Research (2007). https://collab01a.scr.siemens.com/gf/project/xip/scmsvn/?action=browse&path=/trunk/examples/data/CT_WHOLE.img&view=log. (Cited on page 86.)

Sony Computer Entertainment Inc. (2007). The Eye of Judgment. http://www.eyeofjudgment.com/. (Cited on page 2.)

Sowizral, H. (1999). Introduction to programming with java 3d. http://www.sdsc.edu/~nadeau/Courses/Siggraph99/. (Cited on page 19.)

Sowizral, H., Rushforth, K., and Deering, M. (2000). *The Java 3d API Specification with Cdrom*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA. (Cited on pages 17 and 19.)

Spreen, K., Cochard, D., MacIntyre, B., and Tseng, T. (2009). Arhrrrr! http://www.augmentedenvironments.org/lab/research/handheld-ar/arhrrrr/. (Cited on page 2.)

Strauss, P. S. and Carey, R. (1992). An object-oriented 3d graphics toolkit. In *SIGGRAPH '92: Proceedings of the 19th annual conference on Computer graphics and interactive techniques*, pages 341–349. (Cited on pages 1, 10, 11, 12, 13, 14, 15, 18 and 79.)

Stroustrup, B. (2000). *The C++ Programming Language (Special 3rd Edition)*. Addison-Wesley Professional. (Cited on pages 17 and 55.)

Sun Microsystems (2009). Java remote method invocation - distributed computing for java. http://java.sun.com/javase/technologies/core/basic/rmi/whitepaper/index.jsp. (Cited on page 16.)

Tanenbaum, A. S. and Woodhull, A. S. (2006). *Operating Systems Design and Implementation (3rd Edition) (Prentice Hall Software Series)*. Prentice Hall. (Cited on page 15.)

Tarbox, L., Krych, J., Paladini, G., Moeller, T., Pearson, J., Smith, K., and Prior, F. (2008). Xip: the extensible imaging platform. http://erl.wustl.edu/publications/posters/amia2008xip.pdf. (Cited on page 22.)

The OpenSceneGraph Community (2009). Openscenegraph. http://www.openscenegraph.org/projects/osg. (Cited on page 18.)

The Web3D Consortium (1997). Vrml97 functional specification and vrml97 external authoring interface (eai). http://www.web3d.org/x3d/specifications/vrml/. (Cited on pages 1 and 19.)

Thierfelder, K. (2004). Dateiformate für dreidimensionale Daten. (Cited on page 19.)

Tidwell, J. (2005). *Designing Interfaces*. O'Reilly Media, Inc. (Cited on pages 4, 32, 33 and 94.)

Upstill, S. (1989). *RenderMan Companion: A Programmer's Guide to Realistic Computer Graphics*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA. (Cited on page 10.)

van Dam, A. (1988). Phigs+ functional description revision. *SIGGRAPH Comput. Graph.*, 22(3):125–220. (Cited on page 10.)

Vaughan, G. V. and Tromey, T. (2000). *GNU Autoconf, Automake and Libtool*. New Riders Publishing, Thousand Oaks, CA, USA. (Cited on page 57.)

Walsh, A. (2002). Understanding scene graphs. *Doctor Dobbs Journal*. http://www.web3d.org/x3d/content/examples/development/UnderstandingSceneGraphs.pdf. (Cited on pages 1, 9 and 10.)

Wernecke, J. (1993). *The Inventor Mentor: Programming Object-Oriented 3d Graphics with Open Inventor, Release 2*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA. (Cited on pages 11, 12, 14 and 76.)

Wernecke, J. (1994). *The Inventor Toolmaker: Extending Open Inventor, Release 2*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA. (Cited on page 11.)

Whole Tomato Software (2009). Visual assist-x for visual studio. http://www.wholetomato.com/. (Cited on page 57.)

Zyda, M. J., Pratt, D. R., Monahan, J. G., and Wilson, K. P. (1992). Npsnet: constructing a 3d virtual world. In *SI3D '92: Proceedings of the 1992 symposium on Interactive 3D graphics*, pages 147–156. (Cited on page 16.)

Glossary

Notation	Description	
API	**A**pplication **P**rogramming **I**nterface. An interface which allows a programmer to use libraries or operating systems functions	19, 23, 42, 51, 55, 60
Automake	A utility which automates the generation of makefiles.	56
CMake	CMake is a cross-platform build system which can be used to configure a project with source code once and built-in on multiple platforms using different programming environments.	56
Coin3D	A scene graph library fully compatible to Open Inventor. Used for the implementation of this work.	17, 18, 51, 55, 56, 59, 60, 70, 72, 73, 76, 77, 79, 84, 91
Context menu	A menu available in most computer programs using a GUI. It enables a number of application specific actions to be performed. In most programs a context menu is shown at the head of a GUI using textual menus.	6, 42
CUDA	**C**ompute **U**nified **D**evice **A**rchitecture. An architecture developed by NVIDIA to perform operations on the GPU. The GPU can be programmed through a C-like language. Tasks which imply many parallel computations can be greatly accelerated by the parallel hardware available on GPUs.	22
DAG	**D**irected **A**cyclic **G**raph. A graph with no path that starts and ends at the same vertex *(Black [2004])*	9, 11, 14, 45, 48
DCMTK	**DICOM T**ool**K**it. An open source library which implements large parts of the DICOM standard.	22
DICOM	**D**igital **I**maging and **Co**mmunications in **M**edicine. An image standard for handling medical image data. It defines a file format and a communication protocoll.	22

Notation	Description	
Direct3D	A low level graphic library for Microsoft Windows. It is part of the DirectX API. In contrast to OpenGL it is not available for multiple platforms but only for the Windows platform.	19
DLL	**D**ynamic-**L**ink **L**ibrary. A shared library for Microsoft Windows. Can be used to access functionality during run-time.	50, 76
DSG	**D**istributed **S**cene **G**raphs. Similar to DVE, distributed scene graphs share a scene graph between multiple computers. For performance reasons that usually means that the scene graph is stored locally on each computer and only changes - performed for example by the user - are send over a shared network to the other computers. The lower the network bandwidth and the higher the latency involved the catchier it becomes.	16
DVE	**D**istributed **V**irtual **E**nvironments. A virtual environment which is distributed among many computers who all share the same scene. Changes on one computer are immediately reflected on other computers sharing the same environment.	16
GPL	GNU **G**eneral **P**ublic **L**icense. A software license without licensing fees which requires programmes using any program or library published under GPL to use the same licensing terms as the original software.	19, 24, 55
GPS	**G**lobal **P**ositioning **S**ystem. A widely used system for navigation. GPS can be used to determine the current position using satellites.	2, 83
GPU	**G**raphics **P**rocessing **U**nit. A dedicated computer chip used to perform and accelerate graphic applications. The specialization on computer graphics leads to a dramatic performance increase, compared to integrated hardware. In recent years, GPU architecture has evolved, allowing other, non-graphic applications to facilitate the power of GPUs.	10, 19, 22
GUI	**G**raphical **U**ser **I**nterface. A graphical interface which allows users to interact with electronic devices such as computers or mobile phones.	6, 7, 19, 24, 27, 31, 32, 36, 42, 45, 47, 56, 59, 61, 77, 92
ICG	**I**nstitute for **C**omputer **G**raphics and Vision. The institute at Graz University of Technology where this thesis was written.	51
ITK	**I**nsight Segmentation and Registration **T**ool**K**it. An open source library mainly used for image segmentation and registration.	22, 23, 56, 84

Notation	Description	
LGPL	GNU **L**esser **G**eneral **P**ublic **L**icense. A software license which is a little more unconstrained than GPL. The main difference is that LGPL allows linking with libraries which are not published under the GPL. Programs or libraries published under the LGPL can be used by commercial software as well.	18, 55
makefile	A configuration file containing information how to build executable programs from source code.	56
MDL	MeVisLab file format used to store information to files and retrieve the information from files.	36
MVC	**M**odel **V**iew **C**ontroller. A software design pattern which separates the program logic from user input and display of data.	30, 61, 63, 72
Open Inventor	The mother of all modern scene graph libraries. Although outdated, many of the design decisions made remain true until today.	VII, IX, 9, 11, 12, 14, 17–20, 22–24, 35, 36, 45, 47, 50, 51, 59, 70, 79, 83, 87, 91
OpenGL	**O**pen **G**raphics **L**ibrary. A standard, low level graphic library. It is managed by a industry consortium called the Khronos Group. OpenGL stands in competition to Direct3D. It is available for multiple platforms and has a very wide support.	1, 2, 6, 10, 17–20, 22, 23, 51, 56
qmake	A part of the Qt library which automates the generation of makefiles. It is not required to use Qt to make use of qmake.	56
Quarter	A viewer for the Coin3D scene graph library using the Qt framework. For integration of Quarter into GUIs designed with Qt Designer, a widget is integrated into Qt Designer.	45, 55, 56, 76
RegEx	**Reg**ular **Ex**pressions. A flexible and powerful way to check strings if they meet certain criteria.	72, 73
SGI	**S**ilicon **G**raphics, **I**nc. was a major driver in the field of computer graphics. They were producing high end, dedicated computer graphic hardware. SGI also invented the famous Open Inventor scene graph library. They filed for chapter 11 bankruptcy twice in the year 2006 and 2009.	18, 19, 24, 56
SoQt	A viewer for the Coin3D scene graph library using the Qt framework.	55

Notation	Description	
STL	**S**tandard **T**emplate **L**ibrary. A standardized library for C++. It provides template data structures and algorithms needed in everyday programming.	18, 55
Toolbar	A widget element available in most GUI programs. It usually displays a number of icons for quick access of specific, commonly used functions inside programs. Very often the elements in a toolbar are also available in the associated context menu.	6, 42, 45, 70, 73
Tooltip	A tooltip is some information which is usually associated to a GUI element and which is used to display extra information to that element. Very often this is done through some popup window which is shown when having the mouse over a GUI element for a certain, usually short, amount of time	6, 42, 47
UDP	**U**ser **D**atagram **P**rotocol. A protocol included in the Internet Protocol. UDP can be used to send messages over IP-based networks. In contrast to TCP, it trades speed for reliability, offering no services such as handshake, ordering of the incoming packages or even data integrity. It is widely used for multimedia applications where speed matters such as for example computer games.	17
UIC	**U**ser **I**nterface **C**ompiler, included in the Qt framework. UIC can be used to create the source code required to display a user interface created by Qt Designer. The makefile can be configured to automatically generate the code.	59, 60
VRML	**V**irtual **R**eality **M**odeling **L**anguage. A standardized file format usually used to store the information of an interactive three-dimensional scene.	1, 19, 20, 22, 24
VTK	**V**isualization **T**ool**K**it. An open source library for computer graphics and visualization.	22, 23, 56, 84
Widget	A basic element of a GUI. Widgets give the user the opportunity to change or view some data through a single interaction point. Qt and other GUI libraries offer many different widgets.	29, 33, 35, 36, 42, 47, 55, 59, 60, 63, 70, 72, 73, 77, 91, 92
WYSIWYG	**W**hat **y**ou **s**ee **i**s **w**hat **y**ou **g**et - During editing, the preview shown is what you will see when loading the final result. Often used in the process of GUI design.	42

Notation	Description	
XIP	e**X**tensible **I**maging **P**latform. An open source project started by Siemens Corporate Research which has the goal to unify a number of image processing and visualization libraries into a common structure to allow rapid development of medical applications.	22–24, 35, 36, 56, 75, 83, 84
XML	E**x**tensible **M**arkup **L**anguage. A set of rules to encode information. Widely used on the Internet and for all kind of programming tasks. It enforces a certain structure by defining a number of rules which split the content from the rest of the application.	19, 35, 50, 77, 81, 91