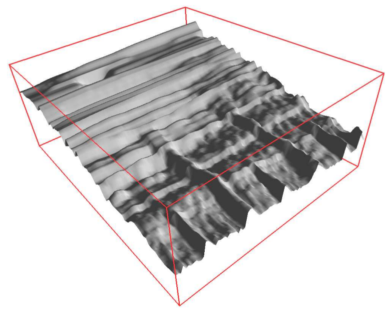
Technische Fachhochschule Berlin
Fachbereich Allgemeine Informatik

Diplomarbeit

A Real-Time 3D Visualization Toolkit for Sound and other Signals

Andreas Lücke



Durchgeführt am Center for New Music and Audio Technologies (CNMAT),
University of California Berkeley, USA

14. September 1997

Diplomarbeit

Thema:	Ein 3D-Visualisierungswerkzeug für Klänge und andere Signale
Entstehungsort:	Center for New Music and Audio Technologies, Berkeley, USA
Name:	Andreas Lücke
Fachbereich:	13
Studiengang:	Allgemeine Informatik
Matrikel-Nr.:	561682
Betreuer:	Prof. Godbersen

Hiermit versichere, daß ich meine Diplom-Arbeit selbstständig verfaßt und kleine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Berkeley, den 14. September 1997 ....

Andreas Lücke

Contents

1 Preface **6**

1.1 Abstract . . . 6

1.1.1 Motivation . . . 6

1.1.2 Goals of This Work . . . 9

1.1.3 Structure of this Document . . . 9

1.1.4 Acknowledges . . . 10

2 Overview **12**

2.1 System Engeneering . . . 12

2.2 Software Overview . . . 13

2.3 How to use tsKit . . . 15

2.4 Application Scenario . . . 16

2.4.1 Representations of Sound . . . 16

2.4.2 Generating a Data Set . . . 16

2.4.3 Rendering a Movie . . . 17

3 Background **21**

3.1 Why Visualization ? . . . 21

3.2 Requirements . . . 22

3.3 Related Work . . . 24

3.3.1 Open Inventor Applications and Extensions . . . 24

ChemKit – Molecular Visualization . . . 24

Soft Primitives – Modeling Organic Forms . . . 24

VrTool – A Virtual Reality Developers Toolkit . . . 25

3.3.2 *MET++* – A Multimedia Application Framework . . . 26

3.3.3 *MAX* – A Graphical Music-Programming Environment . . . 26

3.3.4 *VRML2.0* – Virtual Reality Meta Language 27
3.3.5 *IRIS Explorer* – A Scientific Visualization Toolkit 28
'When Timbre Comes Apart' 28
3.3.6 Thesis *A Real-Time 3D Signal Analysis/Synthesis Tool Based on the Short Time Fourier Transform* 28

4 System Design **30**
4.1 Using Libraries and Frameworks 30
4.2 The Open Inventor Library 31
4.2.1 General 32
Naming Conventions 33
Type System 33
4.2.2 Scene Graph 34
GUI for Nodes an Scenes 38
Grouping Nodes within NodeKits 39
4.2.3 Runtime System 40
Field Connection 40
Engines 41
Manipulators 41
Sensors 42
Callback Nodes 43
4.2.4 File Format 43
4.2.5 Open Inventor Summary 43
4.3 Reused Software Technology 45
4.3.1 SGI's Digital Media Library 45
4.3.2 GUI Builder *RapidApp* 45
4.3.3 Other Reused Software 45
Scrub Audio Samplerate Converter 45
4.4 Design Problems, Their Discussion and Solutions 47
4.4.1 Runtime Types 47
4.4.2 Unified Data Structure for Signals 48
One-Dimensional Signal Data 49
Two-Dimensional Signal Data 52
4.4.3 Elements - Internal Node Communication 53
4.4.4 Reading Signal Data from Files 55

4.4.5 Control of Time 57
General 57
Synchronizing Feedback 58
4.4.6 Controlling Field Values 61
Field Editors 61
Scene Graph control with `gview/tsgview` 61
Direct Control with Manipulators 62
4.4.7 Feedback for Time Signals 64
`tsLine` – Drawing a Line Strip 64
`tsAudio` – Playing Audio 64
`tsSurface` – Drawing a Surface 65
4.4.8 Time Signal Shapes with Normals 67
4.4.9 Colors and Transparency 70
4.4.10 Transforming Time Signals 71
Engine for computing Colors and Transparency Values 71
4.4.11 Control a Selection 71

5 Conclusions **73**
5.1 Conclusions 73
5.2 Limitations, known Bugs and Future Work 75
5.2.1 Limitations and known Bugs 75
5.2.2 Future Work 76

A Results **79**
A.1 Software 79

Bibliography **80**

Chapter 1

Preface

1.1 Abstract

The subject of this thesis is a software toolkit to visualize and transform audio and other signals. The presented solution for this interesting *intramedia* problem makes use of visual computing, to enhance auditive perception. I have developed a reusable software system called *tsKit* using high-end software technology for recent graphic workstations (In particular from Silicon Graphics). It is necessary to stress for my purpose that the term signal is defined as a function of time. Thus, though 'time signal' may appear redundant, I shall use this phrase selectively to stress the importance of time *qua* dimension.

The main software engineering task was to generalize a mono-media framework architecture (3D graphics library Open Inventor) into a multimedia library capable of supporting a synchronious real-time rendering of singals as auditory and visual output. Time is established as an integrated dimension for rendering in Open Inventor.

The result of this work is a collection of objects, that are integrated in the Open Inventor architecture and can used like built-in objects by scripting scene descriptions or C++ programming. The extension offers specific features for visualizing, computing and the playback of sound signals. The tools produced in this work have many potential applications in the fields of audio signal analysis and processing, speech recognition, computer music and other fields of research.

1.1.1 Motivation

The following subsections describes motivations behind this work.

Making Perception perceivable

One motivation is from the field of auditory perception; also called psychoacoustic or auditory scene analysis[1] [5]. Psychoacoustics is that part of acoustics which deals with the perception of acoustic waves; not their physics. In its most general formulation, the subject of psychoacoustics is the auditory systems of humans and animals and how these systems allow their possessors to experience the world. A central question of this field is how recieved stimuli becomes perception?

Apporaches to such issues in psychoacoustics can be physiological, psychological, musical, and technological. The same auditory phenomena can be simply experienced or become the subject of analytic study. Computer models can be used to make the process of perception transparent to conscious. This project promises to provide a metalanguage for formulating new feedback loops between different senory fields.

My personal motivation has been to make it possible to use my results to make auditory perception visually perceivable: in a way that fosters its use as a creative tool. I have built an open software tool, that can create videos which enhance, through the perception of visual images, our ability to understand auditory stimuli. I have produced a tool for research and creative endevors rather than a particular theory or application.

The Walkman Effect: Guide Video with Audio

> *I am sitting in the tram, listening to my walkman. The train starts moving with the beginning of a new song on the tape. Outside the tram, a couple of pigeons fly into the air shortly after the strings begin to play.*

It often happens that a visual and an auditory event occurs simultaniously. Visual and auditive events often coincide without a common source, sense or cause. To construct relationships between these events gives me great joy. I like to imagine they concide not randomly but for a reason; I love the moment at which apparent randomnes falls away to reveal symmetry. Each sense of the stream of time comments on the other. With the help of a computer I want to create this synaesthetic effect: one media guides the perception of another.

Shuhei Hosakawa analizes in [11] and [12] a similar phenomena of co-perception. He calls it the Walkman-Effect.

---

[1] For a short summary of the goals of Computational Auditory Scene Analysis see: Daniel D. P. W. Ellis, `http://sound.media.mit.edu/dpwe/writing/hard-probs.html`

It is not yet possible to manipulate video within this framework, but this is one of my ultimate goals. I hope that this work will lay a foundation that will eventually enable me to reach this goal.

Computer Code as a Variation Machine and a Metamedium

In [15] Gene Yongblood, Woody Vasulka and Peter Weibel discuss which role a computer program can have for the art of cinema. Cinema they generalize from the medium (film, video, holography and program code) to " [...] *the art of organizing a stream of audiovisual events in time.".*

> *Thus, the basic phenomenology of the moving image – what Vasulka calls "the performance of the image on the surface of the screen" – remains historically coninious across all media. Digital code, for example, has radically altered the epistemology and ontology of the moving image but has not fundamentally changed its phenomenology. There are no digital images that have not been prefigured in painting, film and video. With the code we can only summarize them, elaborate and unfold them or exercise modalities. Vasulka calls the code a variation machine. There are no new classes of images, there are only new variations and new epistomological and ontological conditions for generating and witnessing those variations.*

Later they write

> "[...] *aesthetic strategies invented 100 years ago in photography and cinema – scaling, perspective/negative reversals, wipes, mattes – have now become machine elements whose operations are trivially invoked through the preset button. It is a question of primitives. The code is a* **metamedium***: trough it, high-level aesthetic constructs from previous media become the primitives of the new medium. This influences which aesthetic strategies will be emphasized."* [emphasis added by author]

It is a hard work, to build a model inside a computer, that lets it 'understand' the information that it handles, even slight. So, how to make aesthetic use of a computer then and especially it's graphical output capabilities? Why not let it vary something sensible? For me the most videos generated by digital code suffer from being low in dimensionality, related to things created in the 'real world'. I want to take advantage of the fact, that a computer has to handle everything just as uniform information 'bits and bytes', but has a good access to human's sensory material; audio and video.

It can reduce audio and video into information and can turn information into audio and video. So it can become a *meta instrument*. A meta-instrument is an abstract instrument, that incorporates other instruments.

Building an Intramedia Framework

Summarizing the motivations mentioned above I looked for a suitable tool. Because I didn't found a free software offering media independent, fully extensible real-time rendering with orthogonal functionality, I decided to build it my own in this thesis.

1.1.2 Goals of This Work

My one word goal is to visualize audio. The goal for this computer science thesis is to develop a software system that efficiently supports various ways of visualizing audio using animated 3D graphics and video. The software should give real-time auditive and visual feedback, direct gestural control and allow interactive creation and modification of mappings between the two media.

In the fields exploring the perception of audio (psychoacoustic, auditory scene analysis, electroacoustic music) visualization is an undervalued opportunity to model, verify and communicate concepts of reseption and perception. This work attempts to build a easy to use framework that factors out the generic parts of this task, to make visualization a better used instrument for science and research in fields related to audio and music.

1.1.3 Structure of this Document

The thesis *A Real-Time 3D Visualization Toolkit for Sound and other Signals* comes in 2 parts.

Part 1: *Thesis An Open Inventor based Visualization-Toolkit for Sound and other Signals*

Chapter *Overview* gives an overview of basic aspects of the system's architecture, its usage and application.

Chapter *Introduction* is a short chapter, that summarizes the needs of the system and presents related work.

Chapter *System Design* deals with the software engineering and software technological aspects of this project and the basic system it is integrated into. The concepts are discussed for dealing with several problems of storing, rendering, transforming and controlling different time media in real-time and off real-time.

Chapter *Conclusions* draws conclusions and presents ideas for future work.

Part 2: *The tsKit Reference Manual*

The tsKit Reference Manual is the documentation of how the design is implemented into concrete C++ classes and applications. Here aspects are described, that are specific to the implementation of certain classes and class families. It is the detailed documentation of the C++ classes developed in this work. This part is suited for the developers who want to use the tsKit.

1.1.4 Acknowledges

Thanks to David Wessel (Director), Adrian Freed (Director of Research) and Matt Wright (Musical Applications Programmer) for inviting me to the Center for New Music and Audio Technologies [2] and the fruitful discussions and advice. Thanks to Michael Warner for catalyzing my ideas. Thanks to Gibson Guitars and Silicon Graphics for equipment support.

Prof. Dr. Godbersen was my helpful advisor at the Technische Fachhochschule Berlin, Germany.

[2] CNMAT homepage: `http://www.cnmat.berkeley.edu`

Figure 1.1: CNMAT, located at 1750 Arch St., Berkeley, USA, near the campus of the University of California, UCB. CNMAT is a satellite of the UC Berkeley Department of Music.

Chapter 2

Overview

This chapter gives an overview of the essential structures of the work and how it can be used.

2.1 System Engeneering

The largest part of the software built in this thesis is done as an extension of Open Inventor. Open Inventor is a powerfull C++ library providing classes, building blocks, and mechanisms for handling interactive 3D graphics at a high level of abstraction. For a summary of Open Inventor see [23]. TsKit is an extension of Open Inventor providing specialized classes, building blocks, and applications for visualizing sound with animated 3D graphics and synchronized audio playback.

The central data structure of Open Inventor and tsKit is the *scene graph*, a graph of *nodes*. Nodes are containers for properties and data, stored in typed *fields*. The scene graph is the description of a 3D scene that is rendered by Open Inventor using the low level graphics API OpenGL. TsKit comes as a set of additional nodes, all having the prefix '`ts`'.

See figure 2.1 for an overview of the system architecture.

See figure 2.2 for the structure of a simple scene graph and how it is rendered in a viewer window. The scene starts with a root node (`SoSeparator`) and contains standard nodes and tsKit nodes: A master clock (`tsTime`) and its graphical control user interface (`tsTimeCtrlUI`), a node containing a small two-dimensional signal (`tsData2RNFloat`), a standard node containing a color (`SoMaterial`), and finaly a node that renders the signal to a surface (`tsSurface`). Figure 2.3 in the next section shows the corresponding file in the Open Inventor file format and C++ source code

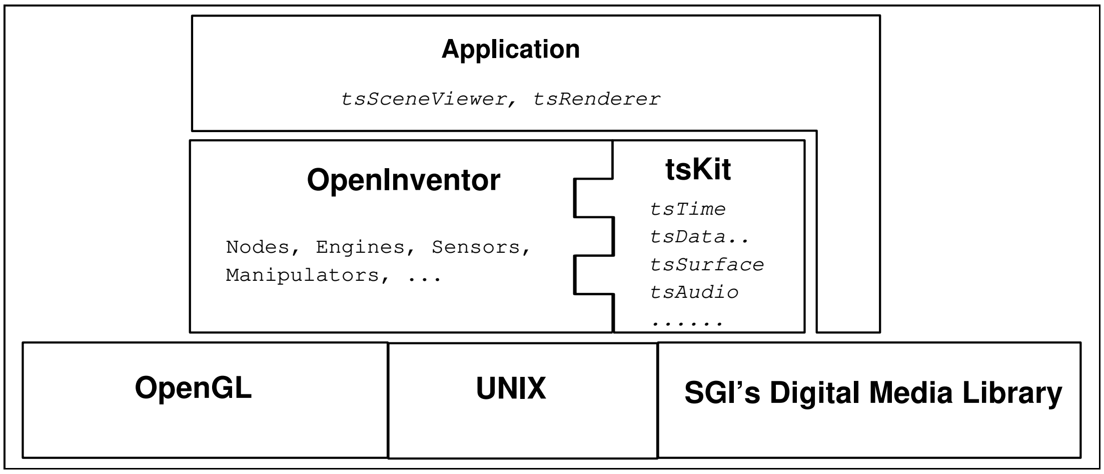


Figure 2.1: The schema shows the software architecture of tsKit.

creating this scene.

2.2 Software Overview

TsKit offers the following building blocks as nodes and applications:

Nodes – Building Blocks for the Scene Graph

`tsTime` : **Master Clock.** This node contains a automatically updating clock with fields 'time' and 'speed'.

`tsTimeCtrlUI` : **GUI to control the Master Clock.** This node offers sliders to change the fields 'time' and 'speed' of the master clock and command buttons for 'play', 'play backwards', 'stop', 'go to start', and 'go to end'.

`tsData1[R|N]Float[File]` : **Store one-dimensional signal data.** These nodes store one-dimensional signal data as float values, regularly and non-regularly sampled. Regularly sampled signals can be read directly from an audio file (AIFF, AIFC, and others) and can be subsampled to save memory space.

`tsLine` : **Rendere a one-dimensional signal.** This node draws a one-dimensional signal to a line strip connecting the sampled

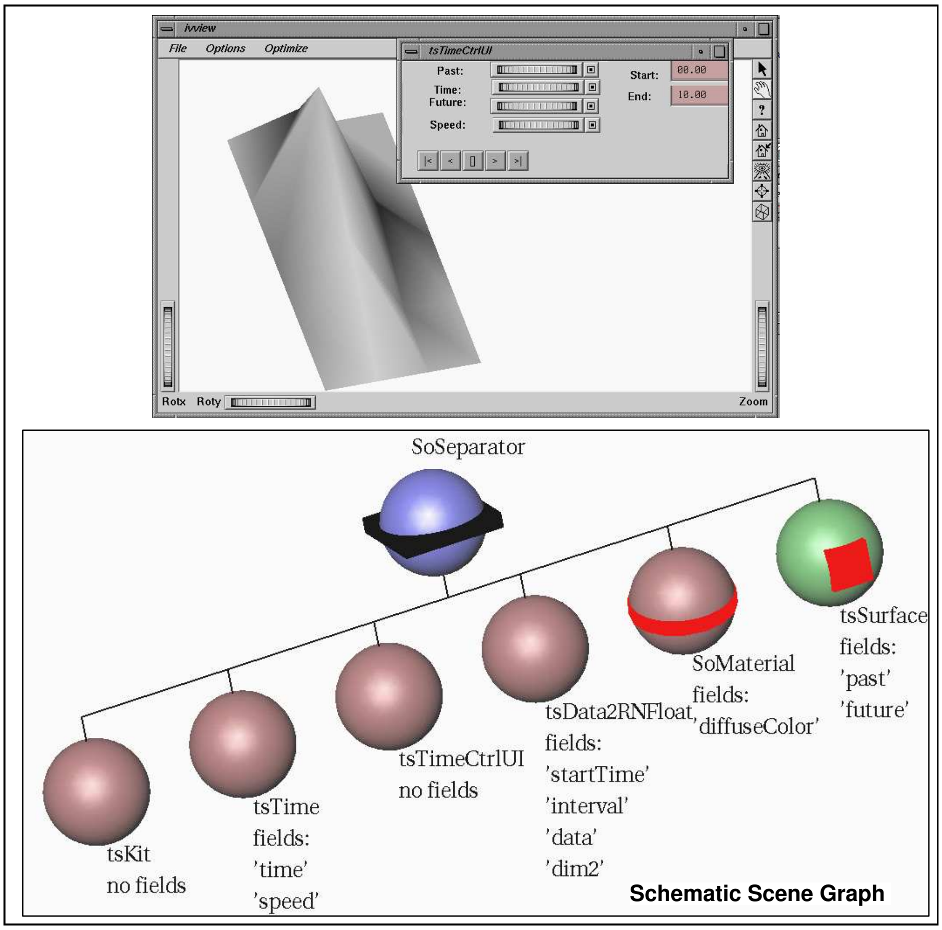


Figure 2.2: A schematic view of a simple Open Inventor/tsKit scene graph (bottom) and loaded into a standard viewer with `ivview helloTsKit.iv` (top).

values. It shows the complete signal or only an interval around the point in time dictated by the master clock (`tsTime`).

`tsData2RNFloat[File]` : **Store two-dimensional signal data.** These nodes store two-dimensional signal data as float values. The signal is sampled regularly in time and non-regularly in the second dimension. Signals can be read directly from an file containing spectral data of an audio file.

`tsSurface` : **Rendering a two-dimensional signal.** This node draws a two-dimensional signal as a surface connecting the sampled values. It shows the complete signal or only an interval around the point in time dictated by the master clock (`tsTime`).

`tsAudio` **: synchronized audio playback.** This node plays an audio file from a `tsData1RFloatFile` node, synchronized to the master clock's time and speed.

Applications

`tsSceneViewer` **: Simple scene graph editor.** This program is an extension of the `SceneViewer`, coming as part of the demonstration source code of Open Inventor. The iconic scene graph viewer `gview` is integrated for creating new nodes and editing fields of nodes.

`tsRenderer` **: Simple movie file renderer.** This program renders an Inventor scene graph file into animated frames of a movie file (JPEG compressed).

2.3 How to use tsKit

The tsKit can be used in several ways:

Scripting an ASCII file: The fastest way to use tsKit is to write an ASCII text file in the Open Inventor file format. The file describes a layout of nodes and is loaded it into a viewer program to view it as a 3D scene. You can use viewer[1] such as `ivview`, which is distributed with Open Inventor, or `tsSceneViewer`, which is part of tsKit. Figure 2.3 a) shows the ASCII text file of a simple scene. See *The tsKit Reference Manual* for detailed documentation of programs, example files and nodes to be used in a scene description. This is the most recommended way to use tsKit, because this usage was stressed in the developement and tested most.

Figure 2.3 (a) shows a simple Open Inventor ASCII file.

C++ Programming: Open Inventor and tsKit offers its full functionality and flexibility via the programming language C++. *The tsKit Reference Manual* offers a complete documentation of all classes.

Because tsKit is still a prototype, creating and arranging tsKit objects in a C++ program has not yet been extensively tested.

[1] When Open Inventor detects a unknown element of a scene graph a dynamic shared object (DSO) is loaded. This object contains the code of the new scene graph element.

Deriving new classes: The usage of tsKit as a framework by deriving new C++ classes from existing classes requires most knowledge, of the tsKit system, but offers very intersting perspectives. See subsection 5.2 for some planed extensions of tsKit.

Because tsKit is still a prototype, deriving new tsKit classes is not yet documented but quite possible and desired.

Figure 2.3 and 2.2 show a simple layout of signal data in different representations.

2.4 Application Scenario

This section shows a realistic application of how a sound is visualized with tsKit as an animated 3D graphics with synchronized audio playback.

2.4.1 Representations of Sound

One application of tsKit is to visualize sound signals. There are at least two representations of sounds that are interesting for examinations. Figure 2.4 shows a sound in both representations. First the *time domain signal.* This is the waveform of the airpressure over time and is widely used to store audio, e.g., on a CD. The second representation is the *time-frequency domain signal.* Here a time-frequency distribution is applied to the time domain signal to generate a two-dimsional signal. This signal describes the energy/amplitude of a sound in time *and* frequency and is also called spectrogram. To represent the frequency components and how they evolve over time is an essential instrument for understanding many signals and is not only used for sounds.

The time-frequency domain is more interesting for studying sounds, because it gives a better understanding in how a sound sounds[2].

2.4.2 Generating a Data Set

This section describes how to prepare a sound to be visualized with tsKit.

The time domain signal can be read directly into tsKit. The node `tsData1RFloatFile` has a field for a file name of a sound file and reads it in the first time, when the data is needed for rendering . The format of the soundfile can be

[2]Two sounds can look very similar in the time domain, but sound very different and vice versa, they look different, but sound similar.

AIFF (Audio Interchange File Format), AIFC, WAV, MPEG1 and some others. The frequency-domain signal must first be generated from a time-domain audio signal. In `tsKit/bin/`[3] there is a program `cqt3` from D.P.W. Ellis that does a constant-Q wavelet analysis of a AIFF file. See [8] and [7] for a detailed description of this analysis program. The program reads an audio file (AIFF/AIFC) and outputs a file with the extension '`.aqt`'. Here is how to analyze an example audio file :

```
cqt3  -o 8 -g .25 mySoundfile.aiff
```

These option settings are suggestions and can be changed.

Because the `.aqt` file is very big and the results can be accessed only with low performance, I wrote a converter `tsKit/bin/aqt2gridcompl` taking an `.aqt` file and outputing the relevant data in compact binary format with an extension '`.grid`'. This conversion is very slow! Here is how a `.aqt` file is converted into a .grid file containing <numberOfRows> spectra, that can be read into a node of class `tsData2RNFloatFile`:

```
aqt2gridcompl mySoundfile.aqt mySoundfile.grid <numberOfRows>
```

The last parameter is the number of rows the .grid file will contain. Here is how to calculate this value: With a suggested sample frequency of 60 spectra per second and a sound duration of <length>[4] seconds it is:
<numberOfRows> = <duration> sec * 60 rows/sec..

Now you have to write a ASCII text file in the Open Inventor file format to view your data in a viewer. Take an example file (`tsKit/data/ivfiles/*.iv`) as a template and substitute the references to the auidio files with the your file names.

2.4.3 Rendering a Movie

The animated 3D graphics can be rendered into a movie file in non-real-time. Here is how the example file from above (`myInventorFile.iv`) is rendered in a movie called `myMovie.mv` of length 4 seconds, at a speed of 0.5:

```
tsRenderer -l 4.0 -s 0.5 -o myMovie.mv myInventorFile.iv
```

After you have chosen a perspective press the 'R' button to start the rendering.

The compression scheme is SGI-JPEG, the frame rate is 30 Hz and the frame size is 640x480.

See *The tsKit Reference Manual* for detailed information.

---

[3] Make sure these executables are accessible by copying them into a directory for executables or include the path to tsKit/bin in the environment variable PATH.

[4] The duration of a sound file can be displayed using `dminfo <soundfile>`.

```
#include <Inventor/Xt/SoXt.h>
#include <Inventor/Xt/viewers/SoXtExaminerViewer.h>
#include <Inventor/nodes/SoMaterial.h>
#include <Inventor/nodes/SoSeparator.h>

#include <tsKitAll.h> // include all tsKit header files.

// set environment variable TSKIT_DIR to where you have
// put tsKit and compile with:
// CC halloTsKit.c++ -I$TSKIT_DIR/include -L$TSKIT_DIR/lib \
    -lInventor -lInventorXt -ltsKit -o halloTsKit
void
main(int , char **argv)
{
    // Initialize Inventor and tsKit.
    // This returns a main window to use. If unsuccessful, exit.
    Widget myWindow = SoXt::init(argv[0]); // pass the app name
    if (myWindow == NULL) exit(1);
    tsKit::initClasses();     // initialize all tsKit classes.

    // create the nodes and set their values:
    SoSeparator *root = new SoSeparator;
    tsData2RNFloat *mytsData = new tsData2RNFloat;
    float simpleData[] =  {\
      0.0 , 0.0 , 0.0 , 0.0 , 0.0 , \
      0.0 , 0.0 , 0.3 , 0.0 , 0.0 , \
      0.0 , 0.5 , 1.0 , 0.5 , 0.0 , \
      0.0 , 0.0 , 0.0 , 0.0 , 0.0 };
    float simpleDim2[] = { 1, 2, 3, 4, 5};
    mytsData->data.setValues(0, 20, simpleData);
    mytsData->dim2.setValues(0, 5, simpleDim2);
    mytsData->startTime.setValue(0);
    mytsData->interval.setValue(0.5);
    SoMaterial *myMaterial = new SoMaterial;
    tsSurface *mytsSutface = new tsSurface;
    mytsSutface->past.setValue(0.5);
    mytsSutface->future.setValue(0.5);
    root->ref();
    myMaterial->diffuseColor.setValue(0.3, 0.8, 0.8);

    // build the scene graph :
    root->addChild(new tsTime);
    root->addChild(new tsTimeCtrlUI);
    root->addChild(mytsData);
    root->addChild(myMaterial);
    root->addChild(mytsSutface);

    // Create a viewer in which to see our scene graph.
    // The render area will appear within the main window.

    SoXtExaminerViewer *myRenderArea = \
          new SoXtExaminerViewer(myWindow);

    // Put our scene in SoXtExaminerViewer, change the title
    myRenderArea->setSceneGraph(root);
    myRenderArea->setTitle("Hello tsKit");
    myRenderArea->show();

    SoXt::show(myWindow);  // Display main window
    SoXt::mainLoop();      // Main Inventor event loop
    }
```

C++ source code `helloTsKit.c++`

```
#Inventor V2.1 ascii
Separator {
    tsKit {   }
    tsTime { }
    tsTimeCtrlUI { }
    tsData2RNFloat
    {
  data [
      0.0 , 0.0 , 0.0 , 0.0 , 0.0 ,
      0.0 , 0.0 , 0.3 , 0.0 , 0.0 ,
      0.0 , 0.5 , 1.0 , 0.5 , 0.0 ,
      0.0 , 0.0 , 0.0 , 0.0 , 0.0 ,
      ]
  dim2 [
      1.0 , 2.0 , 3.0 , 4.0 , 5.0
      ]
  startTime 0.0
  interval .5
    }
    Material { diffuseColor .3 .3 .8}
    tsSurface {
  past      3.0
  future      3.0
    }
}
```

Open Inventor File `helloTsKit.iv`

Figure 2.3: The scene graph from fig. 2.2 described in an ASCII Open Inventor file (right) and as C++ source code that generates this scene graph and renders it in a viewer (left).

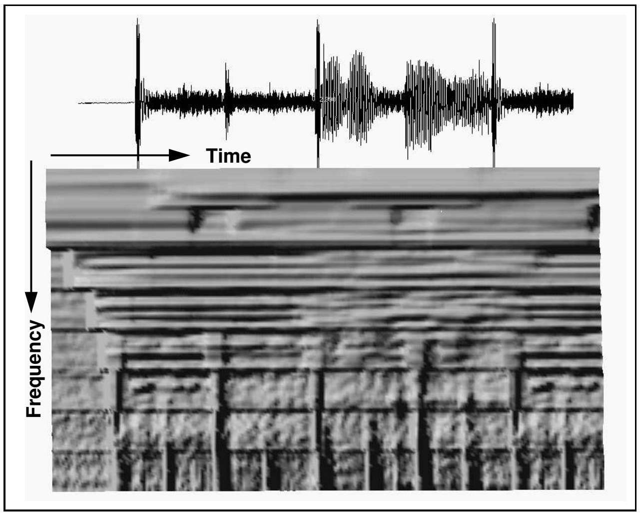


Figure 2.4: Figure showing one sound signal of 2 seconds in the time domain (top) and the time-frequency domain (bottom).

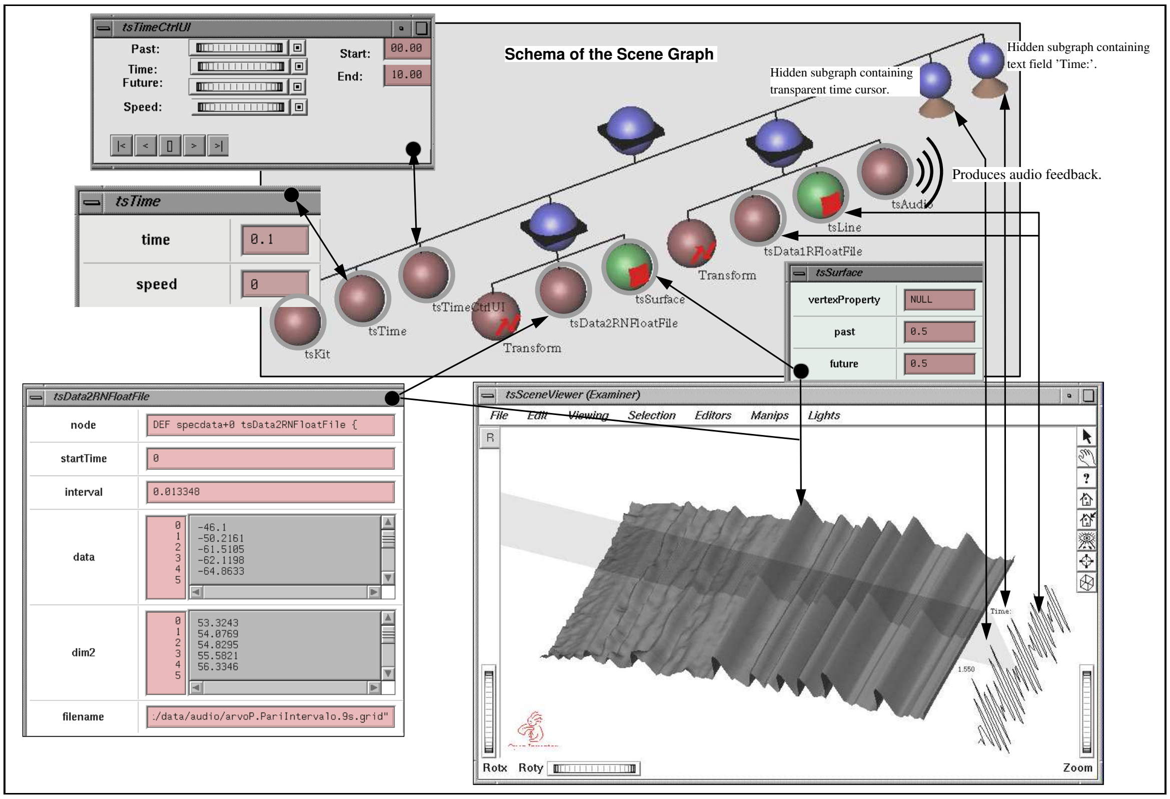


Figure 2.5: This Firgure shows a schematic application scenario of how tsKit is ment to be used. It shows the scene graph of file `Expl.Application.iv` loaded in the `tsSceneViewer` (bottom, right) and as a graph of node-icons (top). These nodes that are part of the tsKit begin 'ts' and are marked with a cricle around their icons. Arrows denote how nodes are visualized or edited in dialogs.

Chapter 3

Background

3.1 Why Visualization ?

Visualization wants to shift cognitive processing load to the perceptual system[1].

Visualization engages the primary human sensory apparatus, *vision*, *and* the processing power of human mind.

To shift to the visual system takes advantage of the special features of the visual system, especially of the powerful cognitive performance.

Visualization of data, information and concepts is getting more and more important. The fast expanding computer manufacturer Silicon Graphics Inc. has 'Visual Computing' as its main mission.

Arnheim says in [4], pp. 13: "*There is no basic difference in this respect between what happens when a person looks at the world directly and when he sits with his eyes closed and 'thinks'.*" When seeing is thinking, visualization does not need to be ending in an concrete image!

Benefiting from changes in one domain to another is often done and useful in general. Metaphors are an example for a shift to a domain that is often visual. Metaphors are everybody's visualizations from a problem domain into a pseudo-visual domain. Great use of pictures in language are an everyday example of using the visual system in its broadest terms: without getting as concrete as a real picture it is visual thinking. Often conferences are mapped to the visual domain, using its vocabulary of shapes, colors, motion and all visual perceived attributes.

[1] This is a mutation of the original cite in [10] "It seems clear that interactive animation can shift cognitive processing load to the perceptual system. And it seems plausible (but not yet proven) that 3D can be used to maximize effective use of screen space.".

The auditive perceptual system is included in this approach using computers for mapping data to sound. This area is called *Auditory Display* or *Sonification*[2]. The auditive perceptual system can be used as well. Here the ability to perceive parallel structures in time is used.

3.2 Requirements

Because the intra-media character of this framework, it shares at least requirements for multimedia frameworks.

In [1] p. 11/12, Ackermann gives a list of requirements and problems of a multimedia framework. I added my annotations in normal type face to show how these items relate to this project:

> *Software libraries that support the development of multimedia applications have to provide structures and functions that meet the following requirements:*
>
> - *integration of various media*
>
>   Within this work the term 'time signal' covers all time derived media types, including time derived data that is product of analysis or used for control. 3D graphics is native to Open Inventor. Audio (and video in future work) must be integrated. Time must be integrated as a dimension for rendering, like space is already in Inventor.
> - *user interface interactions with direct manipulation of graphical representations and high semantic feedback*
>
>   All graphical representation is done in the domain of 3D graphics. Manipulations of the properties translation, scale and rotation is a build-in feature for all Open Inventor objects. Time specific representations and their manipulation should be possible. Open Inventor's concept of manipulators can be used for this.
> - *support for media compositions to define multimedia presentations by high level synchronization specifications*
>
>   Media composition is needed only on a low level. Geometric composition is native for Open Inventor. Basic temporal composition (sequencing) should be possible with future work.

[2] see ICAD: International Community of Auditory Display, `http://www.santafe.edu/icad/`

- *real-time behavior*

  3D graphics and audio must be rendered and controlled in real-time. Media transformations can be done in off- real-time. Real-time capturing, buffering and transformation is future work

- *flexibility and portability in view of different hardware configurations and platforms*

  It is not a requirement for this work, to be platform independent, although Open Inventor and OpenGL are ported to many platforms. The intensive usage of SGI's digital media libraries makes the software SGI specific, but hides different hardware configurations.

- A requirement not mentioned above but essential is to transform media signals. Transformations cover tasks like analysis, mixing and feature extraction of media signals. Transformations are needed as building blocks for the main usage as a tool for mapping audio to video.

Additionally, there are problems that complicate the development of multimedia software systems:

- *support for the data type time is often lacking in programming languages*

  Basic data type `SbTime` is available in Open Inventor. Time must be conceptual introduced as a additional dimension for rendering and control.

- *different real-time constrains at low-level device interfaces for each media type*

  SGI's media programming API offers a device independent interface for buffering, converting and synchrinizing audio and video frames. Real-time constrains are fulfilled by playing audio (video in future work) in separate threads. 3D graphics has no real-time constrains in term of fixed frame rate. Animation is perceived as a fluent movement starting at about 15 frames per second (15 fps).

- *rapid hardware evolution*

  SGI's digital media programming API hides most changes due to hardware evolution. Although there were important changes in the API in the recent releases.

- *strong hardware dependencies because of ADCs, DACs, DSPs and other special purpose processors, e.g., for 3D graphics accelerators or video compression (JPEG and MPEG chipsets)*

  SGI's digital media programming API and Open Inventor/OpenGL hides hardware dependencies. Although knowledge about special hardware constrains is needed to achieve best performance.

- *the lack of established standard interfaces to multimedia libraries that hide hardware dependencies*

  SGI's digital media programming API is same on all SGI machines. Graphics API OpenGL is platform independent.

- *the lack of established standards for the representation of multimedia data (e.g., widely used multimedia file format standards are missing)*

  Open Inventor's file format is widely used an many converters exist to other formats. SGI's library for audio and movie files are independent in formats and compression/decompression algorithms. Standard file formats are supported for audio (AIFF/AIFC, WAVE, MPEG1) and movies (MPEG1, Quicktime).

3.3 Related Work

This section gives a short overview of related systems.

3.3.1 Open Inventor Applications and Extensions

The following projects are applications or extensions of Open Inventor:

ChemKit – Molecular Visualization

ChemKit is an extension of the standard Open Inventor library on SGI workstations. It is similar to the tsKit, but it differs in the subject of visualization. ChemKit provides specialized nodes for storing and visualizing molecular data. ChemKit is the source of many of my ideas of how to extend Open Inventor.

Soft Primitives – Modeling Organic Forms

Modeling Organic Forms Using Soft Primitives
Thesis of Scott Peterson, Cal Poly State University, San Luis Obispo, USA.

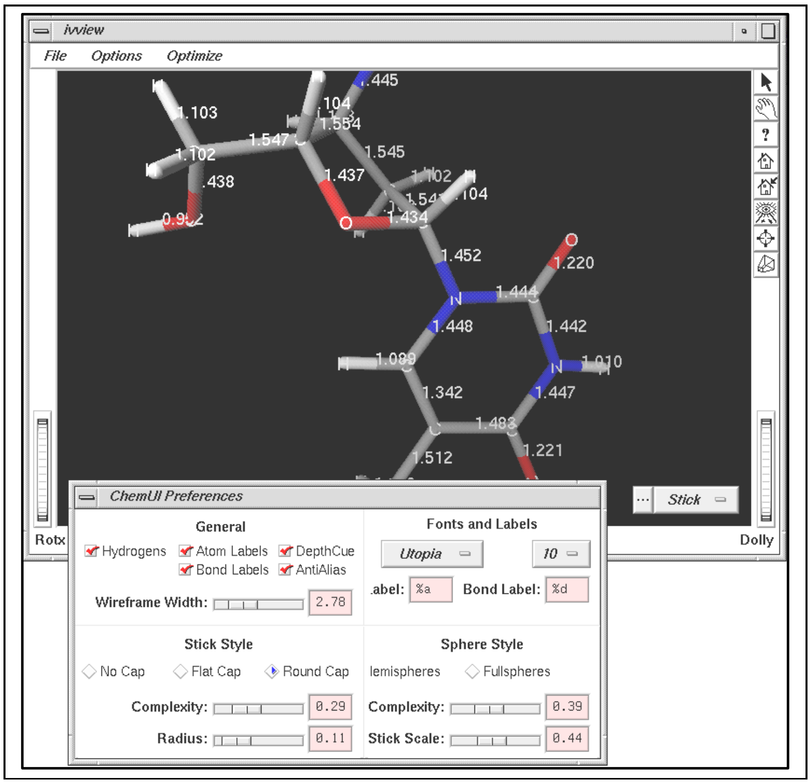


Figure 3.1: A screenshot of a standard Inventor viewer showing a scene graph with ChemKit nodes for molecular data and a dialog for editing visualization parameters.

See
`http://macabre.lib.calpoly.edu/projects/csc/Peterson_Scott_Brandon/contents.html`

From the abstract:

> "*A library of tools, dubbed Softies, has been implemented for the Open Inventor 2.1 architecture. When combined with the transformations and manipulators provided by the Open Inventor architecture, these elements become powerful organic modeling tools.*"

VrTool – **A Virtual Reality Developers Toolkit**

Virtual Environment Technology Laboratory at the The University of Houston, USA. See `http://www.vetl.uh.edu/lincom/VrTool/vrtool.html`

From the abstract:

> *"VrTool is the newest Open Inventor Virtual Reality toolkit to provide a rapid prototyping capability to enable VR users to quickly get their application running with the minimum amount of effort."*

3.3.2 *MET++* – A Multimedia Application Framework

In his works [1], [2] and [3] Philipp Ackermann describes a object-oriented multimedia application framework called MET++[3]. This framework handles the synchronization, viewing and editing of time-dependent data (audio, music, 2D and 3D graphics animation, video). This system is built on top of the application framework ET [9]. Although it is platform and window system independent (Silicon Graphics, Hewlett-Packard, Sun, Linux), it is free available and offers interesting multimedia building blocks, there are two reasons against using this framework for this thesis. First of all, as being a research project, the available version was not stable and far from being efficient, in terms of performance. Second, being a application framework, rather than a visualization framework, the complete ET and MET++ application infrastructure (including a complete GUI and window system layer) has to be reused. This implication would make my work hard to reuse and integrate in other systems. Although the novel direct manipulation interaction on temporal structures in the so-called *Time Composition View* and *Event Graph* (see fig. 3.2) is an interesting concept, to be introduced in future work.

3.3.3 *MAX* – A Graphical Music-Programming Environment

MAX is is a graphical programming environment for event and signal processing. Building blocks of a MAX application are objects with typed input and output connectors. The object incorporates the mapping from the input to the output connections. MAX offers an editor for patching these objects and their connections *while* the application is running. MAX schedules the processing so that the programmer can use it like a parallel machine.

MAX is heavily used in the are of musical signal processing and is used as a system control for live performances and installations. See [16] or WEB site from the maufactuor Opcode Systems[4]

[3] see `http://www.ifi.unizh.ch/groups/mml/projects/met++/` for a project site.

[4] `http://www.opcode.com/products/max/`

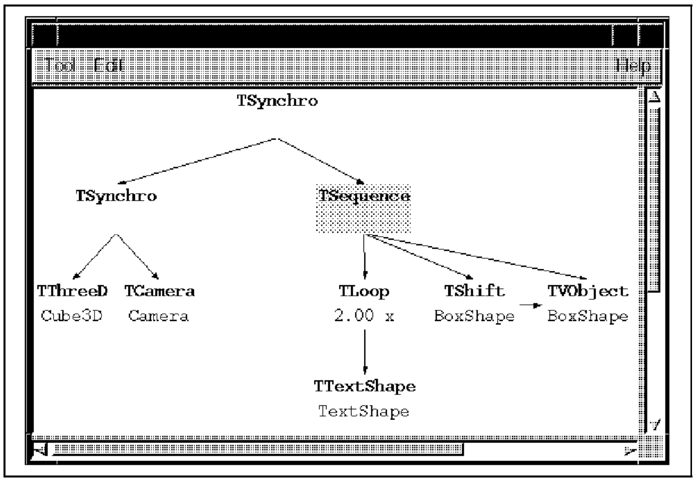


Figure 3.2: The MET++ event graph viewer.

Open Inventors concept of engines that can be created and patched on run-time is very similar to MAX, except there is no editor for patching engines (and nodes) in Open inventor.

3.3.4 *VRML2.0* – Virtual Reality Meta Language

VRML stands for *Virtual Reality Meta Language* and is a meta language for 3D worlds, like HTML (*Hyper Text Meta Language*) is a metalanguage for hypertext. VRML2.0 is the extension of VRML1.0 with an event model allowing scripting with JAVA. VRML1.0 was at least a file format for describing static and non interactive scenes in 3D. Actually it was a subset of the Open Inventor file format.

VRML2.0's architecture has great similarity with that of Open Inventor and shares the concepts of a scene graph with nodes containing the relevant data in typed members called fields that can be connected. It comes as a interpreter for VRML files optional containing JAVA scripts and is platform independent.

VRML offers extensive support for 3D graphics, a limited one for audio, video is nonsupported at all. To implement part of the VRML by extensions in native code (C/C++, called plug-ins) to circumvent limitations in performance or system integration (using other libraries) foils its advantage of being platform independent.

3.3.5 *IRIS Explorer* – A Scientific Visualization Toolkit

IRIS Explorer is an application creation system and user environment that provides visualization and analysis functionality for computational scientists, engineers, and other investigators. Internally IRIS Explorer makes extensively usage of Open Inventor.It is especially useful for those whose needs are not met by commercial software packages. Also, IRIS Explorer's Graphical User Interface (GUI) allows users to build custom applications without having to write a single line of code.

Explorer is a system for creating visualization maps, each of which comprises a series of small software tools, called modules. A map is a collection of modules that carries out a series of related operations on a dataset and produces a visual representation of the result.

See [13] and [14] or the IRIS Explorer Center[5] for detailed information.

'When Timbre Comes Apart'

This work from Jøran Rudi, Norwegian network for Technology, Acoustics and Music (NoTAM), University of Oslo used IRIS Explorer for visualizing sounds in a art work.

See `http://www.notam.uio.no/~joranru/wtca.html` for a paper and images.

3.3.6 Thesis *A Real-Time 3D Signal Analysis/Synthesis Tool Based on the Short Time Fourier Transform*

Alan Wesley Peevers, Master thesis, Department of Electrical Engineering, University of California, Berkeley.

See: `http://www.CNMAT.Berkeley.EDU/~alan/MS-html/MSv2_ToC.html`

From the Introduction:

> *This paper describes a system for audio analysis, modification, and synthesis, based on the Short Time Fourier Transform (STFT). The system is intended both as a tool for sound manipulation, and as a means to reinforce people's intuitions regarding the relationships between timbre and the harmonic structure of music and other audio signals, as conveyed via their spectrograms. This is done by creating a 3D spectrogram which shows a sound's harmonic structure in great detail as it is sampled. Similar systems in the past (for example, David Shipman's SPIRE system)*

[5] `http://www.nag.co.uk/Welcome_IEC.html`

have often sought to convey harmonic structure via two-dimensional spectrograms, sometimes in conjunction with wave form or other displays. By adding the third dimension (amplitude in dB mapped to surface height), it is hoped that a greater apprehension of the detailed structure can be achieved.

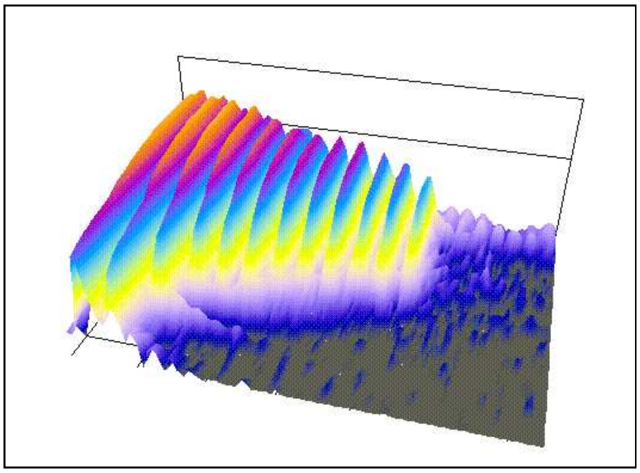


Figure 3.3: Image from [19], Copyright 1995, UC Regents, Universisty of California, Berkeley. All Rights Reserved.

Weseley's work is related to this thesis in using 3D graphics (OpenGL) to visualize spectral audio data. His work stresses performance and is not designed as an open framework.

Chapter 4

System Design

> This chapter deals with the software engineering and software technological questions of this project. After an introduction of the architecture of the reused software Open Inventor, it introduces the concepts for storing, real-time rendering, transforming and controlling different time media.

All class names and code examples are set in `typewriter` style, e.g., `SoCone`.

4.1 Using Libraries and Frameworks

To keep track with rapid hardware evolution in the sphere of application software, higher software productivity and quality is tried to be reached by effective software reuse. Object-orientation is seen as an enabling technique to create fine-grained software components which are reusable, extensible and flexible enough to (re)design software architectures of complex systems without tremendous expenditure. On the other side, object-oriented software reuse often could not keep the promises this technique has made. Objects often cannot be reused fine-grained, because single objects or mechanisms cannot be used separately, without importing the complete class-tree of objects and those, on which it relies on.

Another weakness lies in the concrete language, object-orientation is mostly realized in: C++. It is not possible, or at least not easy to build mechanisms in C++, that are generic enough to deal with customers needs and extensions. E.g. until the last revision of the ANSI C++, there was no possibility for a dynamic type checking.

Libraries are collections of function or classes. The protocol for using a library's function is kept simple and often covered by the documentation of input and output

parameters. Therefore the functionality provided by libraries can be integrated into many application domains. The user of a library still has the duty and opportunity to build his own specific 'frame' from where he calls the library.
When the functionality is encapsulated in classes of basic data types, this class library is often called *toolkit*.
Frameworks contain solutions for classes of problems and building blocks of higher levels by providing object classes *and* a general architecture of how objects cooperate. Frameworks separate the generic and the specific parts of a solution and structure the generic parts as collaborating objects. It often comes with a set of ready implemented objects as a starting point, that build a running system. To implement a specific part of a solution the user derives new classes from existing classes, be they abstract or not. To build this work a framework for 3D graphics called Open Inventor from SGI was chosen to shorten the period of development. Without using a framework this work could hardly be done in the period of a thesis. One task of this works is to verify the statement, if Open Inventor's concepts and mechanisms are kept generic enough and not limited only to render geometric data to OpenGL calls, but to render any media data to any domain.

The next section gives a introduction into Open Inventor.

4.2 The Open Inventor Library

> This section gives an introduction to Open Inventor (recent Version 2.1.3) on SGI workstations. This introduction elaborates the abstractions in the framework's architecture in respect to use of them for other media data, especially time derived media like audio (and video in future works).

Open Inventor is a powerful toolkit for programming 3D graphics providing building blocks for 3D graphics applications of different size in a fully extensible C++ library. Open Inventor is one of the main libraries available from SGI, developed over several years. To support 3D graphics programming on every level it offers *black-box* reuse as a file format[1] or a class library as well as *white-box* reuse by providing its infrastructure as a framework for custom extensions. Black-box reuse designates a reuse of components that does not require knowledge of its internal implementation but only of their interface. In Inventor this is the file format syntax and the programming interface, i.e. class structure and their public methods and members. A

[1]VRML 1.0 (Virtual Reality Meta Language) is a subset of the Open Inventor's file format.

white-box reuse, i.e. integrate own classes into the system requires the knowledge at least of a part of the internal implementation of the mother classes.

In this work Open Inventor is used as a framework because its generic parts are employed for an extended problem domain: rendering audio specific data to 3D shapes and audio-feedback in real-time and off real-time (movie file). Once Open Inventor is extended by new C++ classes, it offers these new functionality for black-box reuse on all levels.

4.2.1 General

In general Open Inventor is an interactive renderer independent in the domain it renders to. It supports hierarchical structuring of data to be rendered in an acyclic directed graph (see paragraph *Scene Graph*), interaction with this data in the render domain (see paragraph *Manipulators*), caching mechanisms, reading from and writing to a file representation (see paragraph *File Format*), generic event scheduling (see paragraph *Callback Nodes* and *Sensors*). Additionally to the data flow allong the graph-tree it allows data to flow from one object to another object including a processing in the flow (see paragraph *Field Connections* and *Engines*).

The core functionality of Open Inventor is independent from the operating system, window system, devices and render engines. The native purpose is to render 3D graphics using the low level graphics library OpenGL, which is a industry standard[2]. OpenGL is hardware accelerated on many SGI workstations and integrated in the X-window system. Open Inventor is shipped with a supplement with X-Windows components, like several viewers and editors.
Open Inventor is ported to many platforms[3].

The main task of Open Inventor is to interface between a high level usage of graphical objects in terms of description, rendering and interaction to the demands of a low level render engine, here OpenGL.

Reference Manual It is necessary for a real understanding of Open Inventor to have access at least to the reference manual of the libraries classes. There are several ways to get access to the Open Inventor's manual pages:

[2]see The Industry's Foundation for High Performance Graphics at `http://www.opengl.org`. It's recent extension, OpenGL-Optimizer has many concepts build in, that are similar to Open Inventor's and so providing an API on an abstraction level, that is between OpenGL and Inventor. See `http://www.sgi.com/Technology/OpenGL/optimizer` for more about Optimizer.

[3]see Template Graphics for availability on other platforms: `http://www.tgs.com`

On-line: Install the subsystem `inventor_dev.man.pages` of the Open Inventor development environment `inventor_dev`.

Paper: *Open Inventor Reference Manual*, Addison Wesley. See [22] and [24] for other Open Inventor documentation.

Internet: `http://mimsy.mit.edu/Transom/inventorman.html`. See [23] for a Summary of Open Inventor.

Naming Conventions

To understand the following introduction it is necessary to know the way classes are named in Inventor. See the following sections for further descriptions of these classes.

There are basic classes, for basic data structures, having the prefix '`Sb`' for 'Scene Base', e.g. `SbString` for a string, `SbVec2f` for a vector composed out of two floats. All classes of objects that can be placed in a scene graph have the prefix '`So`' for 'Scene Object'. Most of them are nodes. Nodes contain typed data as member variables, called *fields*. Fields provide a generic access protocol. All field classes have the prefix '`SoSF`' for fields containing single values ('Single Field') and '`SoMF`' for fields containing multiple values ('Multiple Field').

Classes that are specific to OpenGL, have the extended prefix '`SoGL`', e.g., `SoGLRenderAction` is an action that traverses the scene graph to render it to OpenGL calls.

The naming convention in this work *tsKit*, for 'Time Signal Toolkit', is that all classes have the prefix '`ts`'.

Type System

Like many other C++ libraries, Open Inventor has its own generic type system based on the class `SoType`. This is necessary, because C++ has no possibility for requesting the type of an object, referenced by a pointer[4]. Therefore every class derived from the very base class `SoBase` has a method for requesting the object's and class' type:
`SoType getTypeId()`
and the static version
`static SoType getClassTypeId()`. Other important services of the type system is to query the inheritance relations with

[4] Until the recent revision of the ANSI C++, there was no possibility for a dynamic type checking.

`SbBool SoNode::isDerivedFrom(SoType parent)`

and type generic object instanciation:

`void* SoType::createInstance()`

The design pattern, to create objects generic by choosing the type at runtime, is called *abstract factory* (see [9]). This mechanism allows to create objects of classes, that are designed after compile time of the program creating them! This is an essential design pattern to keep libraries and frameworks open.

4.2.2 Scene Graph

Open Inventor's concept for representing all information affecting the scene (i.e. what is rendered to an image) in an acyclic directed graph is called *scene graph*. For a better understanding of this model, the simpler model of the state machine is introduced first. Open Inventor can be seen as an interface to such a state machine.

The State Machine

The simplest state machine is nothing more than a set of functions that affect a domain. Beginning from a defined start situation, these functions are called with specific arguments and change the domain's state. The set of functions might be drawing commands of a graphic library, and the domain that is affected is a bitmap on the screen that is blank at the beginning. There are commands, that set drawing parameters, like color and affect commands executed in the future by changing the librarie's internal state. Other commands only affect the bitmap, by, e.g., drawing a line. The state, i.e., the picture that is drawn is determined by the sequence of function calls the program produces. It is Important that there is no other notation of this sequence of calls manipulating the domain and the systems state, than the program itself.

This simple way of dealing with manipulations of a domain can be implemented very efficiently in terms of performance and storage. This basic interface gives the interfacing system the biggest freedom to its internal organization. This becomes important, when dealing with tasks that are accelerated by special hardware. In general, low-level APIs are often realized as state machines.

An example is OpenGL. This is an industry standard for an platform independent API for 3D, 2D graphics and image processing. The former version IRIS GL was invented by Silicon Graphics Inc. for making usage of special graphics hardware, if this is available on a concrete workstation. The frame buffer and the set of library internal variables are the state of OpenGL. There is no particular concept for time in OpenGL.

Other examples are low level graphics libraries, e.g., Microsoft's Direct3D.

The disadvantage from the view of an application programmer is the low abstraction level that state machines offer. Especially for vision, we are used to deal with objects and their properties, e.g., we think of wheels of a car, instead of command sequences drawing a wheel. The application programmer has to make this mapping from an object idea to a sequence of drawing commands. A state machine gives neither the opportunity to summarize commands to an object, nor to deal with relations these objects have with each other! The mapping from a object description to a sequence of drawing commands is common to many applications and can be standardized, because it is redundant in many aspects. This gives the motivation to introduce the graph/traversal model described in the next subsection.

The Graph/Traversal Model

There is another concept to representing data (3-D graphics, images, audio, GUI-elements, windows ....) in conjunction with an rendering process on this data: graph/traversal model.

A very widely used concept for the given problem is to represent all data concerning the domain in a ordered directed acyclic graph (DAG) and to traverse the graph for a special purpose. A graph consists of connected nodes. Starting from a unique root-node, there are connections to its *child nodes*. For a child node (short: child) there is only one node, it is connected from, called its *parent node*. Nodes that have no children are also called *leaf nodes* (short: leaf). The connection of two nodes is used to model a relation between these two nodes, which is specific for the interpretation of the graph. A *traversal* is the software process of interpreting the graph, starting from a node and following the node's connection in a ordered way. See fig. 4.1 for a simple acyclic graph.

Acyclic means, that there is a node A connected to a child node B, but no connection that makes A a direct or indirect child of B.

In Inventor the scene graph is built out of different C++ objects, all are derived directly or indirectly from the abstract base class `SoNode`. This class contains among others the methods and member variables to be connected in a scene graph. Most nodes in Inventor are leaf nodes. All information that makes up a scene is stored in nodes, either in a shape node, that represent an actual shape or in a property

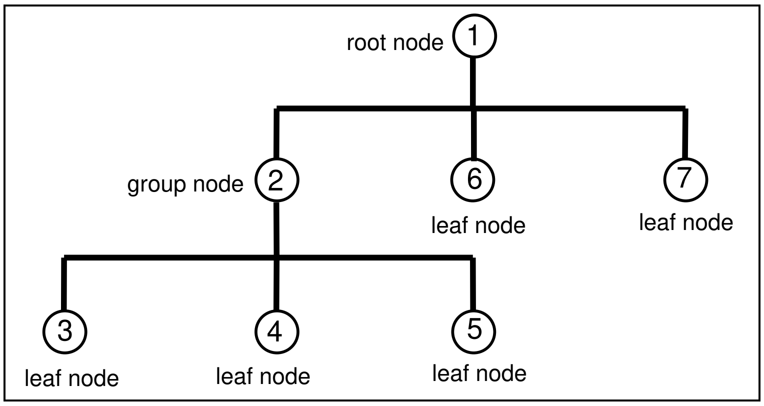


Figure 4.1: A simple acyclic directed graph (DAG) with 7 nodes, symbolized as circles. The number in the circle shows the order of traversal (from left to right) used in Open Inventor.

node, that represent a specific property for a shape[5], e.g. its color or size. Examples for nodes are `SoCube` to represent a cube, `SoMaterial` to represent color and transparency information, `SoLight` and `SoCamera` to represent lights and cameras of various types. These nodes contain relevant information to the scene, but they cannot have children. Only classes derived from class `SoGroupNode` can have children (methods among others: `addChild(SoNode* nodeToAdd)`). These group nodes do not contain relevant data, but have important influence on the interpretation of their children on traversal.

Nodes store scene graph data in objects called *fields*, derived from the abstract base class *SoField*. There are fields for one or multiple values of different types, e.g., `SoSFFloat` is a field class for one float value, `SoMFColor` is a class for multiple colors (of class `SbColor`).

The traversal is done by objects, derived from the abstract base class `SoAction`. Each action traverses the scene graph for a different purpose and with a different interpretation of each node type. The most important action is the `SoGLRenderAction`. The `SoGLRenderAction` does the core functionality of Open Inventor by traversing the scene graph and producing OpenGL calls, so that the scene is rendered into a 2 dimensional frame buffer possibly mapped to the screen[6]. There are other actions e.g., for writing the scene to a file (`SoWriteAction`), searching the scene graph for a special node (`SoSearchAction`) or computing the bounding box for parts of the

[5]A term for shape node that is abstracted from the 3D graphics domain is *feedback node*.

[6]Unless the rendering is done to a off-screen buffer.

scene graph (SoBoundingBoxAction). Finally, on traversal, the actions call node methods to perform their action specific behavior, e.g., the SoGLRenderAction calls GLRender() for every node[7].

There is one more important mechanism that is essential, before a simple scene graph scenario can be fully explained. Where does Inventor take all the properties from that can affect, e.g., a cube shape? Where is the cube positioned, which color is used, and how is it sized? It would be very unconvenient and inefficient to attach all possible properties to every shape object explicitly. It is the position in the scene graph that determines the scope of a node's influence. In general, a node that contains information, that should be shared by other nodes and so affects these, stores this information in a special object when traversed, that brings it to other nodes along the scene graph. Other nodes that follow in the scene graph read this information and use it for rendering themselves. These objects are called *elements* (derived from a abstract base class SoElement) and store information for the period of a traversal[8]. Elements function as a vehicle for the propagation of scene graph information from a property node to a shape node. There are elements for information describing, e.g. material (SoMaterialElement), geometry (SoCoordinateElement) and drawing style (SoDrawStyleElement).

The collection of all elements that are relevant for an action is called a *state* (class SoState). A state stores elements in a separate stack for each element. The top element on the stack is the valid one, that following nodes read from.

It is often neccessary to limit the effect of nodes to only a subtree of the scene graph. An important group-node class SoSeparator makes its child nodes having only a local scope by storing the complete state before its children are traversed and restores it, when this traversal is done. Fig. 4.2 shows a simple scene graph and how it is traversed by an SoGLRenderAction. Separators are used to group nodes to a bigger entity and to protect the rest of the scene graph from settings that are specific to this entity. At the beginning of a traversal elements with default values are created.

Elements affect the state in different ways. The simplest and most used policy is to replace the values in the former element. These elements are derived from the class SoReplacedElement and model an *absolute* change of properties. Translation,

[7]Nodes that do not draw a shape just set their data in the traversal state, where the data affects shapes down the scene graph. For extending the functionality by implementing new actions, e.g., for rendering new media (audio/video/image processing), it is important, that those action-methods are static. That way they can be located as a class member method or outside the class!

[8]When caching is done for an action, elements might exist longer than the period of one traversal.

rotation and scaling are the most important example of object properties that should be changed *relatively* to the former setting. This policy is practiced by all elements derived from `SoAccumulatedElement` and models a relative change of property values. The corresponding element (`SoModelMatrixElement`) does it by accumulating (i.e. adding) its translation, rotation and scale vector[9] to the one valid in the state.

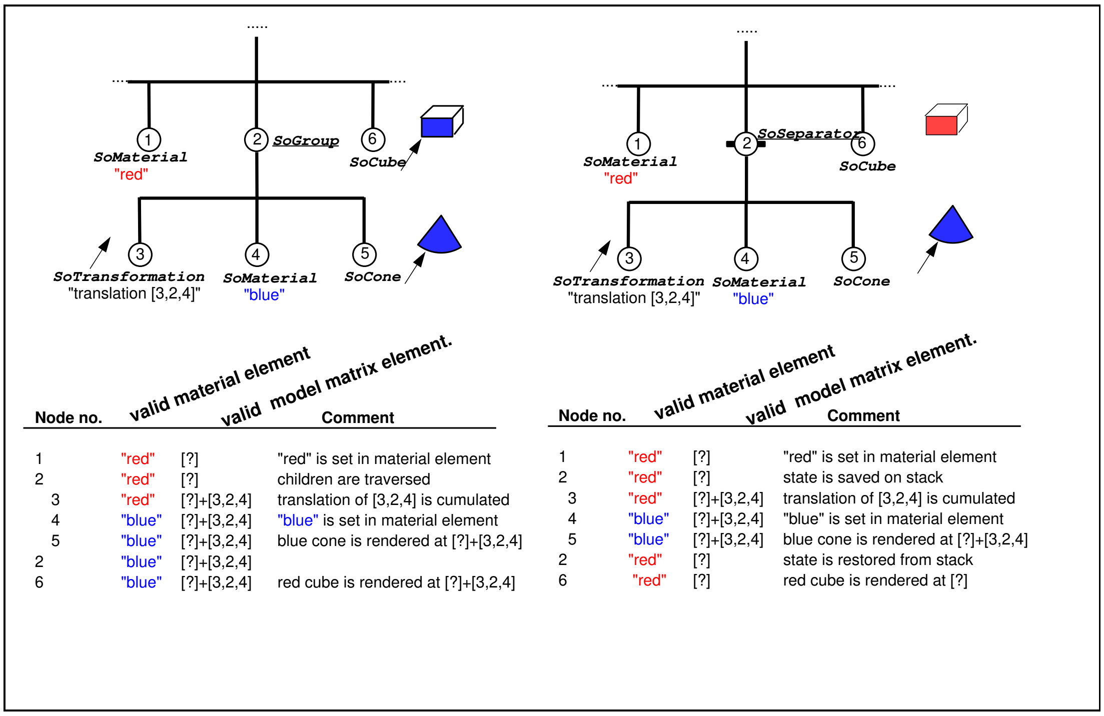


Figure 4.2: A simple scene graph and how its nodes affect the state when traversed by an `SoGLRenderAction` using a `SoGroup` node (left) and a `SoSeparator` node (right).

A well known example for the graph/traversal model is the X-Windows system: The X-Windows system models the GUI of an application in the computer system. It organizes GUI atoms (widgets) as nodes in a widget-tree. The relation that the graph represent, is a 'part of' relation. The tree is traversed by actions for drawing the widgets and handling events. Both action behaviors are coded into the widgets. Widgets draw and handle events themselves.

GUI for Nodes an Scenes

SoXtComponents are collections of X-Motif widgets wrapped in C++ objects for

[9]Actually these vectors are summarized to a matrix (`SbMatrix`).

convenience (abstract base class: `SoXtComponent`). They are attached to a single node or the whole scene graph. Their purpose is to provide a GUI for those. Examples for these classes are viewers for scene graphs (`SoXtExaminerViewer`) and editors for single nodes (`SoXtMaterialEditor` and `SoXtDirectionalLightEditor`).

The scene is normally viewed in a window on the screen by a program or widget-component called *viewer* (various classes derived from `SoXtViewer`). A viewer employs among others a render action to render a image of the scene, whenever the scene has changed, e.g., due of change of perspective. The navigation and change of perspective is realized with a manipulator that the user cannot see, that is attached to the first `SoTranslation`. See the following subsection *Manipulators* for further information about this direct control mechanism for nodes.

Grouping Nodes within NodeKits

A way to group nodes that have a common context and represent a more complex shape is a *nodekit*. A group of nodes and its scene graph structure are encapsulated within a nodekit. Its internal scene graph structure is partly hidden. A nodeKit can be equipped with new fields.

In a previous versions of the tsKit, nodekits were used to capsule the representation, e.g., of spectral audio signal with standard nodes (`SoCoordinate3/4` and `SoQuadMesh`) *and* its special behavior (animation, cursor) with new fields using a nodeKit derived from class `SoShapeKit`.

Disadvantages of the graph/traversal model

There are several disadvantages to represent media data in a hierarchical scene graph structure. They are listed in [21], in the chapter *Optimizing the Data Organization*:

> *It is common for scenes to have hierarchical definitions. Scene management techniques may rely on specific hierarchical information. However, a hierarchical organization of the data raises several performance concerns:*
>
> - *The time spent traversing pointers to different sections of a hierarchy can create a CPU bottleneck.*
>
>   *This is partly because of the number of extra instructions executed, but it is also a result of the inefficient use of cache and memory. Overhead data not needed for rendering is brought through the cache and can push out needed data, causing subsequent cache misses.*

- *Traversing hierarchical structures can cause excessive memory paging.*

  Hierarchical structures can be distributed throughout memory. It is difficult to be sure of the exact amount of data you are accessing and where it is located; traversing hierarchical structures can therefore access a costly number of pages.

- *Complex operations may need access to both the geometric data and other scene information, complicating the data structure.*

- *Caching behavior is often difficult to predict for dynamic hierarchical data structures*

4.2.3 Runtime System

Open Inventor offers several features that are specific for the runtime behavior after a scene graph is created. Runtime behavior is everything that happens dynamically and is event driven, not program driven. As it becomes more possible to express runtime functionality with built-in scene graph constructions, it becomes less necessary to write an application for a special problem domain. The generic architecture for creation and communication of objects and a system to arrange this architecture at runtime lets users express behaviour with data instead of code. In contrast to the signal processing patchwork editor MAX (see. 3.3.3), this runtime editing of object arrangement is not an native purpose of the Open Inventor system, and is supported only with programs code like `gview`, even though it is supported by the system's internal architecture (generic object creation and querying).

Here is an overview of the runtime behavior for the scene graph and how it can be used.

Field Connection

Often it is necessary to let one field having the same value than another field.

Field connections are a basic way to express this dependencies in the scene graph. Connecting fields establishs a second data flow additionally to the data flow along the scene graph. A field B that is connected from another field A is constantly updated with the value of field A. A Field connection between fields of different types is automatically routed through a converter, if one exists for the two types. Field connections are a way to construct a distribution of data from one field to one or more without a processing of the data. The mechanism of engines (see next paragraph) makes extensive use of field connections.

Engines

Engines (all derived from the base class `SoEngine`) construct a dependency of field data that include a processing of these data by an engine's type specific algorithm. Engines have input and output fields of fixed types that are connected to other fields of the scene graph. When executed, the input fields are read, transformed and written to the output fields. Engines are not nodes, but part of the scene graph by their field connections to nodes (or other engines or global fields).

An examples of a general engine is the class `SoCalculator`. It has input and output fields for single and multiple fields of floats (class `SoSFFloat`) and an input field of type `SbString` containing a formula for mapping the input values to the output values. This formula string contains the field names as variables and is evaluated when the engine is executed.

Animation of the scene graph is done by taking the global field 'realTime'[10] as an input field. Engines are evaluated only when necessary, i.e., when a) the input fields' values change, and b) the output fields' values (or any fields connected to these) are requested. That way engines are evaluated only *on demand* and their execution is only done when necessary. That way engines are never evaluated without their output needed.

Manipulators

Manipulators are property nodes, with a graphical representation of their field values as extra shapes. See fig. 4.3 for an example. When the user drags such a graphical shape, the manipulator changes its field values accordingly. Because a class of a manipulator node must be derived from the node class it manipulates, it has exactly the same functionality. For editing, nodes are replaced with their manipulator versions. The author of a manipulator class only has to represent the nodes data as a extra shape the user can manipulate with the mouse[11] and change the fields content accordingly. The shape and its ability to be dragged by the user is offered by objects called *draggers* (derived from the class `SoDragger`). Draggers allow standard

[10] The global field 'realTime' contains the time after the scene graph data base has been initialized as a field of type `SoSFTime`.

[11] or with any other input device.

or user-defined geometry to be dragged with a specific policy and a specific degree of freedom.

There are some manipulators in Open Inventor for the `SoTransformation` node. They offer various ways to change the translation, rotation and scale fields of this node, by creating boxes or circles around the objects they affect(that can be dragged by the user).

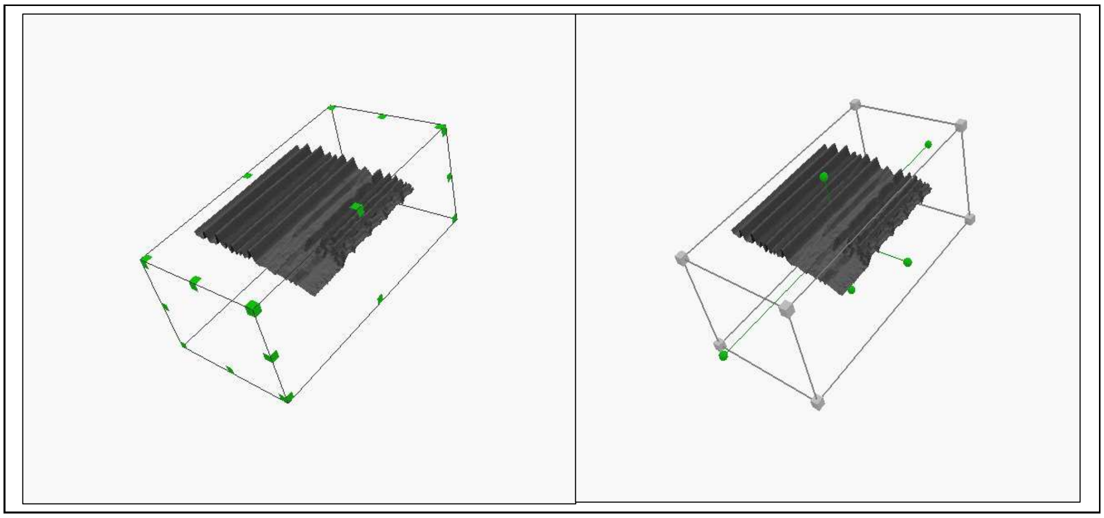


Figure 4.3: Two examples of manipulators replacing a node of class `SoTranformation`: `SoTabBoxManip`, `SoTransformerManip`. Here the node in the scene graph under the `SoTranformation` is of class `tsSurface`.

Sensors

Sensors construct reactions to events and are derived from the base class `SoSensor`. Sensors invoke a callback, whenever a sensor-specific event happens, such as a changes of the scene graph (field, node, group of nodes), the expiration of a timer, or the processor status being idle. Other dependencies can be introduced by deriving new sensors classes.

Viewers make use of sensors, by guarding the entire scene graph with a sensor of class `SoNodeSensor`. Whenever the scene graph changes its content, the viewer starts a new render action to update the scene to the render area. The scene graph might be changed by user interaction or any other reason of field changes.

Open Inventor provides automatic sensor scheduling according to a priority level.

Callback Nodes

Another way to introduce event dependencies are callback nodes `SoCallbackNode`. Open Inventor has its own event scheduling system including the window system events (keyboard, mouse). On UNIX machines these are X-Windows events. User can install callbacks to handle specific event types. If there is no callback to handle a particular event, it is handled by Open Inventor.

4.2.4 File Format

Inventor has a file format for serializing scene graphs, to write and read them into/from files. This mechanism also covers all new classes. There is an ASCII and an binary version of the file format with a default extension '`.iv`'. The ASCII format is well structured and can be used to design scene graphs with only a simple text-editor.

The file format is the simplest interface to Open Inventor.Here are the objects of a scene graph that are written to file:

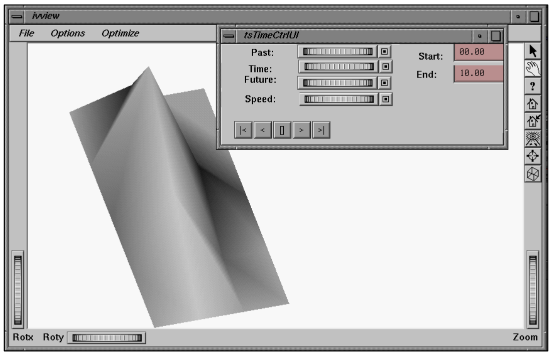
- Nodes with the values in their fields.
- Manipulatrors.
- Field connections.
- Engines and how they are connected.

4.2.5 Open Inventor Summary

Open Inventor is an object-oriented, C++ based, value-adding interface for OpenGL. It offers a hierarchical object structure for describing a scene that is translated into OpenGL calls, allowing the user to deal with abstract graphical objects instead with graphics commands of the low level state machine OpenGL.

Here is an abstract view of the scene graph concept: In the 'real world' there are independent objects occupying space with different properties. The implementation to render a computer model of this with OpenGL needs a mapping to a list of drawing commands. The scene graph model of Open Inventor organizes objects and their properties hierarchicaly to generate a serial description on traversal. Hierarchy is used for an efficient representation (data of one property node can be used by multiple feedback nodes) and control (one property can affect all following feedback

```
#Inventor V2.1 ascii
Separator {
    tsKit {    }
    tsTime { }
    tsTimeCtrlUI { }
    tsData2RNFloat
    {
     data [
         0.0 , 0.0 , 0.0 , 0.0 , 0.0 ,
         0.0 , 0.0 , 0.3 , 0.0 , 0.0 ,
         0.0 , 0.5 , 1.0 , 0.5 , 0.0 ,
         0.0 , 0.0 , 0.0 , 0.0 , 0.0 ,
         ]
     dim2 [
        1.0 , 2.0 , 3.0 , 4.0 , 5.0
         ]
     startTime 0.0
     interval .5
    }
    Material { diffuseColor .3 .3 .8}
    tsSurface {
       past      3.0
       future      3.0
    }
}
```



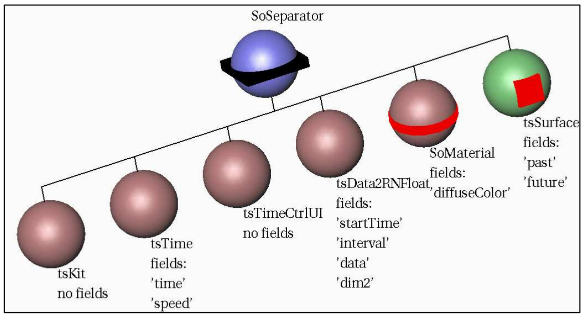


Figure 4.4: This firgure shows a simple scene graph that contains some tsKit nodes as a ASCII file (left), rendered in the standard viewer `ivview` (right, top) and a schematic view (right, bottom).

nodes) of objects. You can see Inventor's graph/traversal model as an inheritance of data *and* of behaviour. A node inherits data from nodes due to its position in the scene hierarchy and it inherits behaviour (methods and member variables) due to its position in the C++ class hierarchy.

Time is handled in Open Inventor only as data type for fields (`SoSFTime`). Time can only influence the scene graph by field connections and engines .

The basic idea of my work is to fuse Open Inventor's 3D graphics and other its general capabilities with digital media libraries to create a universal multimedia and *intra*media rendering tool. The term 'intramedia' reflects a motivation to support the inter-medial transcription from audio to video.

4.3 Reused Software Technology

4.3.1 SGI's Digital Media Library

The digital media library from SGI is in fact a collection of libraries to support media programming on SGI workstations. Until now the audio library is only used for reading and playing audio files (`tsData1RFloat` and `tsAudio` node). The image conversion library and the movie file library is used for hardware accelerated motion JPEG compression of rendered frames (`tsRendererViewer`). In future work the tsKit should be extended to be an easy interface to all digital media facilities including video and image processing.

4.3.2 GUI Builder *RapidApp*

The GUI builder *RapidApp* from SGI is used to simplify the process of building X-Motif based user interfaces. RapidApp generates C++ code for the user to insert his custom specific code. Although RapidApp makes use of the application programming framework *ViewKit*, here only the `VkComponent` class is used isolated, because this makes tsKit's reuse more easier. A `VkComponent` in RapidApp is very similar to a `SoXtComponent` in Open Inventor. See [17] and [20] for further documentation of these tools.

4.3.3 Other Reused Software

Scrub Audio Samplerate Converter

Scrub is a collection of classes written by Tobias Kunze[12] to play audio files with various speed in both directions. To prevent the playback of artifacts due to aliasing, an expensive real-time resampling algorithm is applied to do antialiasing for achieving high audio quality. It implements a signal processing algorithm from Julius O. Smith III, *Bandlimited Interpolation – Introduction and Algorithm*[13], 1994

I added the following features into scrub to integrate it into my work:

- Methods to change and request the point in time of playback.

```
class Scrubber {
```

[12] e-mail: `tkunze@ulysses.stanford.edu`

[13] See `http://ccrma-www.stanford.edu/~jos/src/src.html`

```
    ....
public:

    /// Sets the read position in audio file.
    int SetFilePos(long newpos);

    //  Sets the fiel's current read/play position in seconds.
    void SetFileTime(double newTime);

    // Returns the file's current read/play position in frames and seconds.
    long GetFilePos() ;
    double GetFileTime() ;

    ....
}
```

- Methods to change and request the ratio between the input and output sample rate.

```
class Scrubber {

    ....
public:
    // changes the output sample rate.
    int SetOutputRate(double newOutRate);

    // changes the output sample rate relatively to the source rate (in file).
    int SetIORateRatio(double newIORateRatio);

    double GetIORateRatio();

    ...
}
```

4.4 Design Problems, Their Discussion and Solutions

This section describes the design decisions made to reach the project's goal. See *The tsKit Reference Manual* for a detailed look at a specific classes.

4.4.1 Runtime Types

All classes that are introduced by an extension of Open Inventor have to be initialized as new types of Inventor objects. Being registered by the Inventor's own type system, object types can be queried and objects can be instantciated from a type (`of class SoType`). This mechanism is needed when unknown Inventor objects (nodes) are detected in a Inventor file. (See section 4.2.1 for an introduction to Open Inventor's type system.)

The type initialization of a class is done in its static method `initClass()`. For convenience and security there is a macro doing this. It takes the name of the new class and the name of the parent class[14]. Its definition for nodes is:

```
 SO_NODE_INIT_CLASS(<newClassName>, <parentClassName>,
"parentClassFileName");
```

Normally non standard Inventor classes are type initialized after their first occurrence (while reading them from a file or by explicit instantiation). A dynamic shared object (DSO) of its name is loaded and its object code is dynamically linked. During this project the class type initialization policy became a problem, because

- all parent classes of a class have to be type initialized before the class itself.
- element classes work properly only when initialized once.

A robust solution to the above problems is to initialize all new classes in an ordered way in one method. Therefore I created a new node `tsKit` whose only purpose is to summarize the class initialization. The global initialization is done by calling the static method `tsKit::initClasses()` or by putting `tsKit{ }` in the top of a file.

In future work this procedure should be done automatically and so stay hidden to the user.

[14]The last parameter is the parent's name, that is used in the file format.

4.4.2 Unified Data Structure for Signals

This subsection is about how tsKit represents its most interesting data structure, the time signals[15]. The representation should cover all kinds of time derived media signals. Audio and its related signals are subject of this first realization and therefore shaping the design.

In Inventor there are already data structures to represent arrays of basic data types like floats, integers and vectors. Scene graph nodes like `SoCoordinate3` and `SoCoordinate4` use these to represent geometric information of shapes

Reasons for introducing a data representation for signals as a new class family are:

- Memory efficient representations of signal data. Sinc esignals are often regularly sampled in one or more dimensions, memory use can be reduced enormously. These dimensions can be described more efficiently with only few values, e.g., the time of the first sample and the interval between each sample.
- A unified interface for accessing signal data makes it more comfortable for the user (simple API for developers). It also allows a more generic design of objects that need access to the signal's data: feedback nodes (objects generating a feedback of them) and engines (doing transformations of the).

All nodes storing data of a signal are derived from the abstract base class `tsDataBase` and are called *tsData nodes*. It is assumed that classes derived from this class represent a signal that has a dimension 'time' at least.

The access protocol includes methods for querying the minimum and maximum values, because these operations are fastest when directly done on the underlying data representations. If a client object knows the type of a tsData node, it can circumvent the unified access and use its type-knowledge to practice a node-type specific access. Figure 4.5 shows the methods defined in this class, realizing a unified access to the signal data.

The next subsection introduce the nodes for storing signal data that are implemented so far. See *The tsKit Reference Manual* for a detailed description of all tsKit classes.

[15]Signals are by definition functions of time.

```
class  tsDataBase {

      ...
      // alternative, because original getClassTypeId() is NOT virtual!!
      virtual  SoType  getRealClassTypeId() const
                 {return this->getClassTypeId();};
      /// returns number points sampled in time
      virtual int getNum() const;
      /// returns number points sampled in all dimensions
      virtual int getNumValues() const {return 0;};
      /// returns a value as a pointer!
      virtual const void* getValueTime(int index ) const ;
      /// returns time when data at<index> is sampled?
      virtual float getTime(int index) const;
      /// returns time of first sample point
      virtual const  float getTimeMin() const;
      /// returns time of last sample point
      virtual const float getTimeMax() const;
      /// returns index of a point in time.
      /// Starts search from startIndex (optional)
      virtual int getIndex(float time, int startIndex = 0) const;
      /// returns TRUE  if data is regular sampled
      virtual SbBool isRegularSampled() const;
      ...
}
```

Figure 4.5: Methods building a unified access to signal data, as defined in `tsDataBase.h`.

One-Dimensional Signal Data

The term 'one-dimensional signal data' means here that the signal data is sampled in one-dimension, i.e. 'time', whatever the type of the signal data is. To provide unified access for all one-dimensional data, there is the abstract base class `tsData1Base`. The numeral '1' in the class name denotes the dimensionally, for this class and its derived classes.

There are node classes to represent float signals, sampled regularly and irregularly. Their classes are `tsData1RFloat` and `tsData1NFloat`. The class name contains a 'R' for regularly sampled or a 'N'[16] for irregularly sampled and 'Float' to denote the signal's type of values.

The node class `tsData1RFloat` contains the following fields:

`SoSFNode node` (inherited from `tsDataBase`): Field storing a reference to the node it is placed in.

`SoSFFloat startTime` : Field storing time in seconds where signal begins.

`SoSFFloat interval` : Field storing the interval inseconds between two sample points in time.

`SoMFFloat data` : Field storing signal data. Values are stored starting with the value of 'startTime'.

The node class `tsData1NFloat` contains the following fields:

`SoSFNode node` (field inherited from `tsDataBase`): Field storing a reference to the node it is placed in.

`SoMFFloat time` : Field storing the points in time where signal is samples.

`SoMFFloat data` : Field storing the signal data sampled at points stored in field 'time'.

A regular float signal sampled in n points needs only n floats for the signal values plus two floats for the time axis: 'startTime' and 'interval'. The native irregular representation needs n floats for the signal values and n floats for the points in time when the data is sampled.

Comparing the use of memory for a signal of n values:

$$\frac{\text{regularly sampled}}{\text{irregularly sampled}} = \frac{n+2}{2n} \approx \frac{1}{2} \approx \text{saving of 50\%, for large } n$$

Nodes of class `tsData1RFloat` can represent audio as a time signal. See 4.4.4 for a class derived from this class, to make the signal data being read from a file.

Nodes of class `tsData1NFloat` are often used to represent signals that are derived from other signals, like the pitch contour. Fig. 4.6 shows how these nodes represent the signal.

[16] irregular stands for *N*ot regularly sampled.

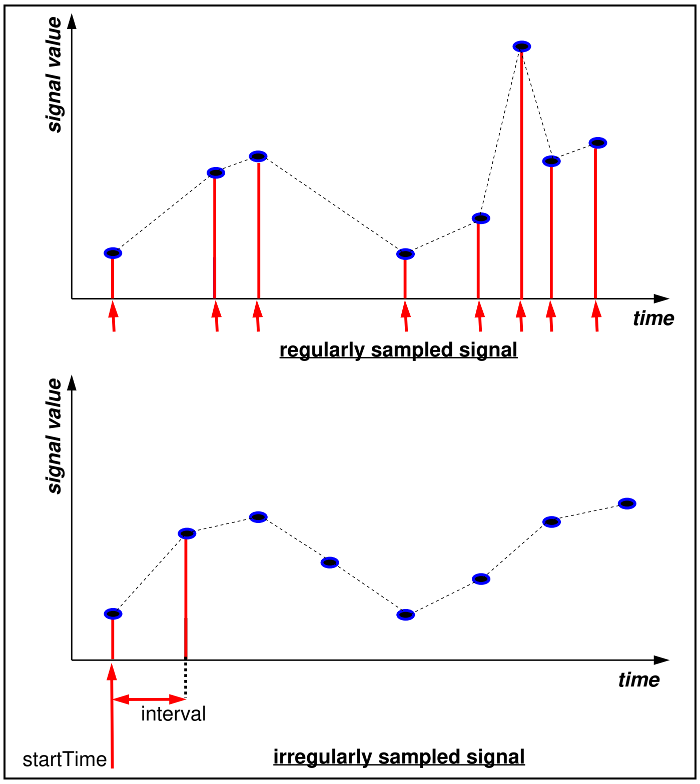


Figure 4.6: This figure shows how regular and non-regular sampled float signals are stored.

```
class tsData1Base {
    ...
    /// returns number of points sampled in time
    virtual const void*     getValue(int index ) const ;

    /// returns minimum/maximum of data values
    virtual void getValueMinMax( float& min,  float& max) const;
    ...
}
```

Figure 4.7: Methods completing the unified interface to one-dimensional signal data, as defined in `http://www.cevis.uni-bremen.de/ilab.h`.

The signal access interface is extended for methods that are specific for one-dimensional signals. The actual `getValue()` method takes a time axis index of the data and returns the value as a `void*` Pointer. Figure 4.7 shows the additional methods.

Two-Dimensional Signal Data

The term 'two-dimensional signal data' means here that the signal data is sampled in two-dimensions, 'time' and one additional one. The signal data itself can be any data type. To provide a unified access for all two dimensional data, there is the abstract base class `tsData2Base`. The numeral '2' in the class name indicates the dimensionality for this class and its derived classes.

There is a new node class `tsData2RNFloat` to store a two-dimensional float signal that is sampled regularly in time but irregularly in the second dimenion. The second dimension is irregularly sampled (called dim2 for short) for all the signal. Fig 4.8 makes this more clear. The class name contains a 'R' for the first and regular sampled dimension 'time' and a 'N'[17] for the second and irregular sampled dimension. '`Float`' indicates the signal's type of values.

The node class `tsData2RNFloat` contains the following fields:

`SoSFNode node` (field inherited from `tsDataBase`): Field storing a reference to the node it is placed in.

`SoSFFloat startTime` : Field storing time in seconds where signal begins.

[17] irregular stands for *N*ot regularly sampled.

`SoSFFloat interval` : Field storing the interval in seconds between two sampled points in time.

`SoMFFloat dim2` : Field storing points sampled in 2nd dimension (values must be monoton).

`SoMFFloat data` : Field storing signal data sampled in time and dim2. All values of one point in time are stored sequentially, starting with the values of 'startTime'.

A float signal regularly sampled in it n points along the time axis and it m points along the irregularly sampled second axis need only n*m+m+2 floats, n*m floats for the signal values, plus two floats for the time axis, and `m` floats for the second axis. The two floats are used for it stroring the start time and it interval. The native irregular representation needs n*m*3 floats. That is n*m value, each containing the actual signal value and its position at the two axes.

Comparing the use of memory for a signal of n values and m values in the 2nd dimension:

$$\frac{\text{regularly sampled}}{\text{irregularly sampled}} = \frac{n*m+m+2}{n*m*3} \approx \frac{1}{3} \approx \text{saving of 66\%, for large } n \text{ and } m$$

Nodes of class `tsData2RNFloat` are used to represent audio signal as a sequence of frequency spectra. The field 'data' stores the amplitude values sampled in frequency (values in 'dim2') and time (fields 'startTime' and 'interval').

Fig. 4.8 shows how the `tsData2RNFloat` node represent the signal.

The signal access interface is extended for methods that are specific for two-dimensional signals. The actual `getValue()` method takes a time axis index and a dim2 index and returns the value as a `void*` Pointer. Figure 4.9 shows the additional methods.

4.4.3 Elements - Internal Node Communication

Elements function as vehicles for information between nodes. They are employed by an action that traverses the scene graph for a special purpose. Each type of elements transports a special type of information. Property nodes store information, and set data in an element, providing it for nodes lower in the scene graph. Other nodes, e.g., shape-nodes use this information to draw a shape. This way geometry data stored in a `SoCoordinate3` or `tsData1RFloat` node, can be used by more than one

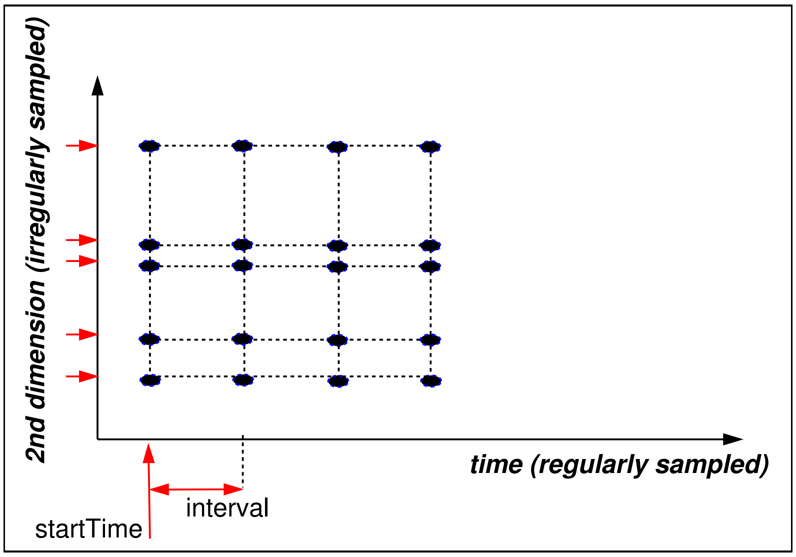


Figure 4.8: This figure shows how the two-dimensional sampled float signals of class `tsData2RNFloat` is stored.

feedback node. Usually a feedback node uses the information from more than one kind of element.

A property node models the static representation of a property and is created and arranged by the user. An element models the spreading of a information from the 'producer', a property node, to the 'consumer', a feedback node and stays invisible to the user. The granulation of property nodes and elements is not the same! The granularity of elements is driven by the needs of the destination nodes consuming them and is kept fine to making caching effective. Nodes can employ several elements to propagate their data.

Because tsKit introduces new propertys nodes (property nodes and feedback nodes for signal data) there are corresponding elements introduced as well. In this project there are several elements needed to propagate the content of one node with multiple fields (fields that store multiple values) to a corresponding element. Therefore in tsKit there is an abstract element base class `tsNodeElement` for propagating a reference of a node and that contains all neccessary methods and variables. This is done by simply storing a pointer refering this node and a type (`SoType`) storing the type of the node being referenced. This policy differs from how Open Inventor propagates multiple field data[18] and is chosen for simplicity.

[18]Inventor propagates single fields, never complete nodes. This ensures that fields can be disabled

```
class tsData2Base{
    ...
    /// returns Min/Max in dim2
    virtual void getDim2MinMax( float& min,  float& max) const;
    /// returns index in    dim2
    virtual int getIndex2(float dim2Value, int startIndex = 0);
    /// returns number points sampled in dim2
    virtual getNum2() const;
    ...
}
```

Figure 4.9: Methods completing the unified interface for two-dimensional signal data as defined in `tsData2Base.h`.

Here is an overview of the new elements and which classes set and get information to/from them:

class(es) that set data	**classes derived from tsNodeElement**	**class(es) that get data**
`tsTime`	`tsTimeElement`	all feedback nodes
all classes derived from `tsData1Base`	`tsData1Element`	all feedback nodes
all classes derived from `tsData2Base`	`tsData2Element`	all feedback nodes
`tsSurface`	`tsNormalElement`	`tsSurface`

Practice will show if this model must be extended in future work for feedback-nodes that need information from more than one signal of the same type, e.g., one two-dimensional and multiple one-dimensional signals.

4.4.4 Reading Signal Data from Files

An important requirement for this toolkit is to access signal data in file formats different from the Inventor file format. Why not let the tsData nodes read in signal data from custom file on creation time[19] ? How generic must this mechanism be?

or overridden separately. Multiple fields are propagated as pointers to an array of standard data types like floats or integers.

[19] More exactly, when traversed the first time!

Here are some possible mechanisms:

1. There is a subclass of every tsData node, for a specific signal file format. This node has an additional field (of type `SoSFString`) for the file name.

2. There is a field of type `SoSFString` for the file name in the `tsDataBase` class. By default every tsData node reads its content from a file, if the file name is not empty and the data field is empty. Each class implements the file formats which it can read.

3. Same as above but with a generic mechanism for selecting the file format. There is one set of objects for each file format. These file format objects (dynamically linkable DSO's) contain a list of tsData types they can read (and save) and maybe a list of file name extensiones. When the content of a file should be read (or saved), all file format objects fitting this file format (and extension) are tried to read the file[20].

In tsKit the second mechanism is implemented. Classes that are derived from a tsData node to implement file reading are called *tsDataFile nodes* and contain an additional field 'filename' of type `SoSFString`.

The signal data of such a node is being assumed as the default (`isDefault()` method returns true) value of the corresponding field(s) and therefore is not written when the scene is written to a file to keep the file small and simple to edit. This data is prepared for written down by changing on of its field content or simply by touching one field[21].

Here is a description of which file format is read by which tsData class:

`tsData1RFloatFile` can read all audio files, that are supported by SGI's audiofile library[22]. If the field 'interval' is not empty, the data is resampled accordingly using the audio file library's mechanism of virtual rate[23].
The node for audio playback (`tsAudio`) requires a node of this class to access the audio data to play. This limitation to this node class is due to the used audio player software (see `tsAudio` node) that only can read audio samples

[20] This mechanism would be similar to the file format independent acess of image data, realised in the Image Format Library. See man page of IFL for more

[21] Touching a field is not changing its value(s) but tonotify the runtime system, that this field is changed. This causes connected fields and engines to be updated.

[22] See manual page of `AFintro` for the currently supported audio file formats.

[23] See manual page of `afSetVirtualSampleFormat()`.

from a file, not from memory. This offers the advantage to play long audio recordings from a file but to only read a downsampled version into memory for drawing with a `tsLine` feedback node.

`tsData2RNFloatFile` can read spectral data from a file in the GRID format (extension `.grid`), which is a simple binary file written by the author. This serves as a intermediate file format to store results from a constant-Q wavelet analysis program `cqt3` written by D. P. W. Ellis in [8] ([7]).

The node maps the amplitude and frequency values to a logarithmic scale.

A more generic handling of file formats is the subject of future work.
Another subject of future work is to let the user specify a special part of the file content or a stage of preprocessing. This gets important when the tsKit will be extended to read the CNMAT's sound description file format [24]. This file format can contain different signals of several types.

4.4.5 Control of Time

In this subsection time as an important property for the rendering of media signals is introduced. Time and space are the most important dimensions in which we perceive the world around us. Hearing and seeing use these dimensions differently. In [10] Graver summarizes: "*Images exist* in *space, but* over *time. Sounds exist* in *time, but* over *space.*".

General

Visualizing audio requires rendering a time signal as a signal of time *and* space (animated 3D graphics). An essential challenge of this work is to establish time as a integrated dimension for rendering in Open Inventor. Time is handled in Open Inventor only as a basic data type `SbTime` for fields (`SoSFTime`). Time can influence the scene graph only by field connections and engines and is provided as a global field called 'realTime'[25].

The first version of tsKit used field connections for distributing the parameter time. Feedback nodes were performed according to a `SoSFTime` field they contained. This approach could not use the hierarchical structuring of the scene graph for propagating this property.

[24]SDIF stands for Sound Description Interface Format and is general file format for sound description in different domains, developed at CNMAT. See `http://www.CNMAT.Berkeley.EDU/SDIF/` for more.

[25]This field can be connected to fields of nodes or engines to create simple animations.

Time can be seen as just another property for rendering. The tsKit introduces feedback nodes that uses this new property as a built-in character.

Time itself has another special characteristic: for an ingenious realization, time must be changing continuously during the playback. Media like audio and video are very sensitive to their continuous performance in time. They are stored and performed in a constant density of frames (audio: samples, video: frames and fields) per time unit. This is called the frame rate of an audio or video stream. Uncontrolled changes in the frame rate, e.g., because of bad timing or insufficient performance causes a decrease in perceived quality. Audio is even more sensitive to changes in frame rate than video. In contrast animated graphics are least sensitive to the continious performance in time because they are not bound to frames but can be performed at any point in time.

Another view says that audio and video are best controlled by the parameter speed (or rate), expressing the continious change of time. Mathematically speaking, speed is the first derivative of time. Animated computer graphics are best controlled by absolute time. User interfaces of time media uses absolute time *and* speed (or rate) as a control parameter. Time is controlled absolutely by controls like 'go to begin' and 'go to end' and speed with 'play forward', 'play backward' and 'stop'.

One application for the control of time is the non-real-time playback. This is used for slow motion playback or to render graphics and audio to a movie file, frame by frame. Later this movie can be played back in real-time. That way visualizations can be viewed in real-time that cannot be rendered in real-time: they are too intensive in computation to be done in real-time.

Synchronizing Feedback

As a solution for the time/speed dualism I created a property node containing both parameters 'time' *and* 'speed' and that keeps these values consistent. The node class is called `tsTime` and consists of the fields 'time' of class `SoSFTime` and 'speed' of class `SoSFFloat`. The field 'time' is constantly updated[26] according to the value in field 'speed'(, as long as speed is not 0.0). This is done in a constant frequency of 30 Hz.

Here is how the `tsTime` node works as a master clock: When a `tsTime` node is part of a scene that is viewed in a viewer, the changes of the field 'time' causes a regular redraw of the scene. This way a `tsTime` node becomes a master clock for

[26] A sensor of class is employed for this. The update frequency is reduced, when the processor is not idle.

all feedback nodes lower in the scene graph (and for all nodes or engines, that are connected to the fields 'time' or 'speed').

To propagate these new properties 'time' and 'speed', a corresponding element of class `tsTimeElement` is created.

There are three kinds of latencies between a change of the clock in a `tsTime` node (the master) and the reaction of the feedback nodes (the slaves). The reason for a change of the `tsTime` node can be a user command, e.g., 'play' that sets the 'speed' field from 0.0 to 1.0 or the automatic updating of the field 'time'. Figure 4.10 shows a schema of these latencies. It shows a scene graph with a `tsTime` node that is controled by a `tsTimeCtrlUI` node and two feedback nodes (`tsSurface`, `tsAudio`).

1. After the `tsTime` node has changed the viewer program of the scene starts a new action (`GLRenderAction`) to render the scene. The first latency is the time between the change of the fields' values of a `tsTime` node and when the changed values are set in the `tsTimeElement` on traversal. This latency is the same for all feedback nodes.

2. The second latency is between the `tsTimeElement` is set and a feedback node's `GLrender()` method is called to perform its feedback. This latency depends on the distance between the feedback node and the `tsTime` node in the scene graph.

3. The third latency is the time that each feedback node needs to react to the changed time information. This latency depends on the node type and the time isthe time needed to e.g., read new media frames into a buffer and send them to a device.

Here is how the feedback node in tsKit are synchronized and how they handle the diffrent latencies of their reaction. The first latency is the same for all nodes and is not handled further. To let feedback nodes handle the second and third latency, the `tsTimeElement` contains an additional high resolution UST time stamp[27] of the time when the element's data was set. This time stamp can be used by feedback nodes to calculate the latency between the time when the `tsTime` node was traversed and when they were able to react. The UST time stamp is provided additionally to the time in the fields because it has a much higher resolution and is fast in processing.

This synchronization mechanism cannot prevent latency in the reaction of feedback nodes caused by limited processing power, but it lets the nodes handle this

[27] The SGI's specific Unadjustable System Time (UST) is used for the time stamp. See man page of `dmGetUST()` for further information.

latency explicitly. Once nodes have started their playback, they are in time. See fig. 4.10 for a schema of this mechanism.

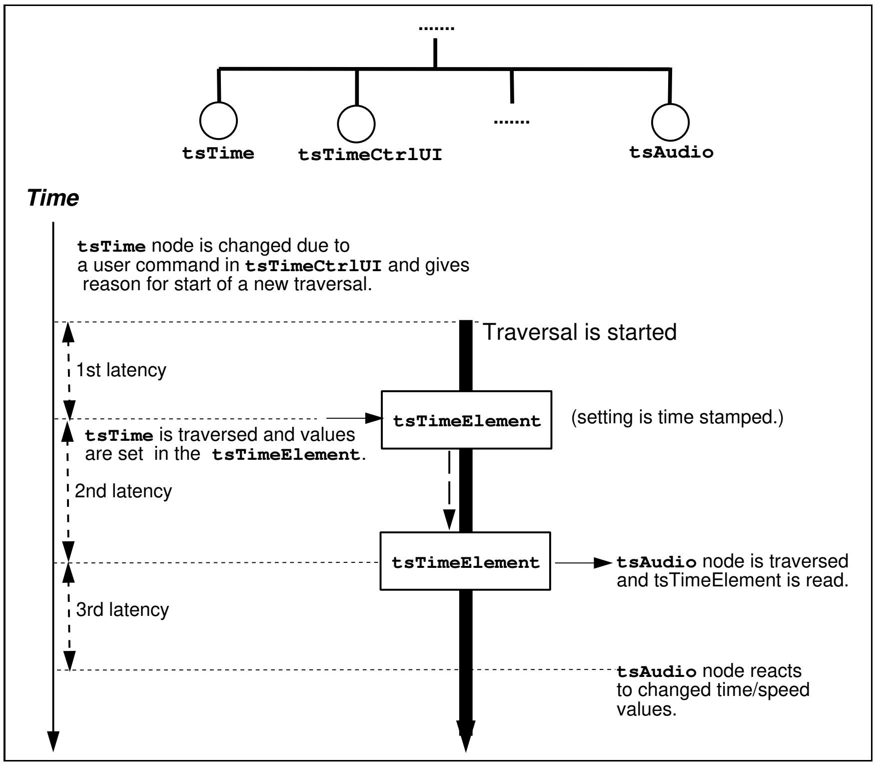


Figure 4.10: This schema shows how a feedback node (here `tsAudio`) can make use of the time stamp included in the `tsTimeElement`.

Media playback is hard, because the media data is often played back from different media buffers by different threads[28] at different rates and because of additional hardware constraints. This synchronization mechanism is chosen with respect to audio (and video) being played in separate threads.

Here is how the feedback node `tsAudio` for audio playback is implemented: the node creates a thread that keeps a frame buffer filled to a minimum level and sends the samples to an audio device. It communicates changes in speed and absolute time to that thread only when necessary. Therefore the node reads the `tsTimeElement` when traversed and checks if its thread is in time, using the element's time, speed and time stamp information. If necessary the node tells the thread to use new parameters

[28]Threads are processes with an own runtime environment, but which are associated to a UNIX process.

(absolute time position and/or speed) in order to be in time again.

4.4.6 Controlling Field Values

In Inventor and so in tsKit the scene graph is the central structure for data *and* its behavior. The scene graph contains objects *and* their relations. There are at least two aspects the user can control: The topology of the scene graph (including field and engine connections) and the values in the fields of nodes.

Fields (see section 4.2.3) are typed containers of data and are part of nodes and engines or exist as global fields.

Field Editors

Besides the trivial way to edit an Inventor ASCII file, there are several interactive ways to edit field values. The fields of a node can be edited in dialogs, which are placed in extra windows or integrated in the scene. Here are the dialogs, either general or specificly to a node type:

Special Field Editors: There is a set of standard classes in Inventor derived from `SoXtComponent` that build a X-Motif dialog for editing only nodes of a specific class. These dialogs are not nodes and therefore cannot be part of the scene graph. They only can be created by programming. Examples are `SoXtMaterialEditor` and `SoXtDirectionalLightEditor`.

Generic Field Editors: There is a class `FieldEditor` (derived from `SoXtComponent` and is part of the demo source code from SGI) that builds a X-Motif dialog for editing a field of any class type using the common field data access methods from the abstract base class `SoField`.

GUI Nodes In tsKit there is one GUI node: `tsTimeCtrlUI`. Its purpose is to create a X-Motif dialog for editing the fields of a `tsTime` node. That way the editor can be created as part of the scene graph. These node(s) are called *tsGUI nodes* because they build a user interface for tsKit nodes. The `tsTimeCtrlUI` node must be placed in the scene graph after a `tsTime` node.

Scene Graph control with `gview/tsgview`

An interactive way to control the structure and content of a scene graph is offered by the viewer `gview`. Gview shows both, a rendered scene graph and a visualization of the scene graph as a graph of icons, each representing a node. A double click on a node icon opens a dialog for editing the field's content. See Fig. 4.11 for a screenshot of gview. Gview also allows to create new nodes and delete existing ones. The list of nodes that can to be created covers all known nodes, including the classes that are added by tsKit or the user.

Until now there is no way to create engines at all. This is subject of future work.

The C++ source code of the program gview is part of the demonstration source code for Open Inventor [29] and is public domain. For the program `tsgview` I only added some minor functionality:

- These nodes for which a specialized editor exists, are edited in those. All other nodes are edited in the former generic dialog.
- The node icons in the graph view are annotated with the name of the node class (Only group nodes are left without annotations).

It is subject of future work to improve the component `SoXtFieldEditor` used by `gview`/`tsgview` for editing fields more adaptive to their type. This includes e.g. editing enumeration fields (`SoSFEnum` of class type `SbEnum`) as menu entries instead of typing its values.

Direct Control with Manipulators

The control concepts described above, all put the place of control away from the model it controls: the rendered scene. A much more ergonomic way is to integrate the control in the scene itself. When the control is part of the scene graph, control and model are not only put in the same place on the computer screen, but the semantic of the manipulation is much more realistic. Only this mechanism simulates the real world experience of manipulating objects, where the manipulation and the feedback of the manipulation is done in the same place: the object itself.

It is clear that external control is much more simple to create than control that is integrated in the scene. Therefore Open Inventor offers mechanism for direct control of nodes, called manipulators. See section 4.2.3 for an introduction. This concept is not yet used by tsKit. It is planned to build manipulators for various property nodes of the tsKit, especially for the `tsTime` node. Because time and speed cannot be manipulated in their native dimensions, they are mapped to a geometry in space.

[29] Install subsystem '`inventor_dev.src.sample`'. Look at `/usr/share/src/Inventor/demos/gview`

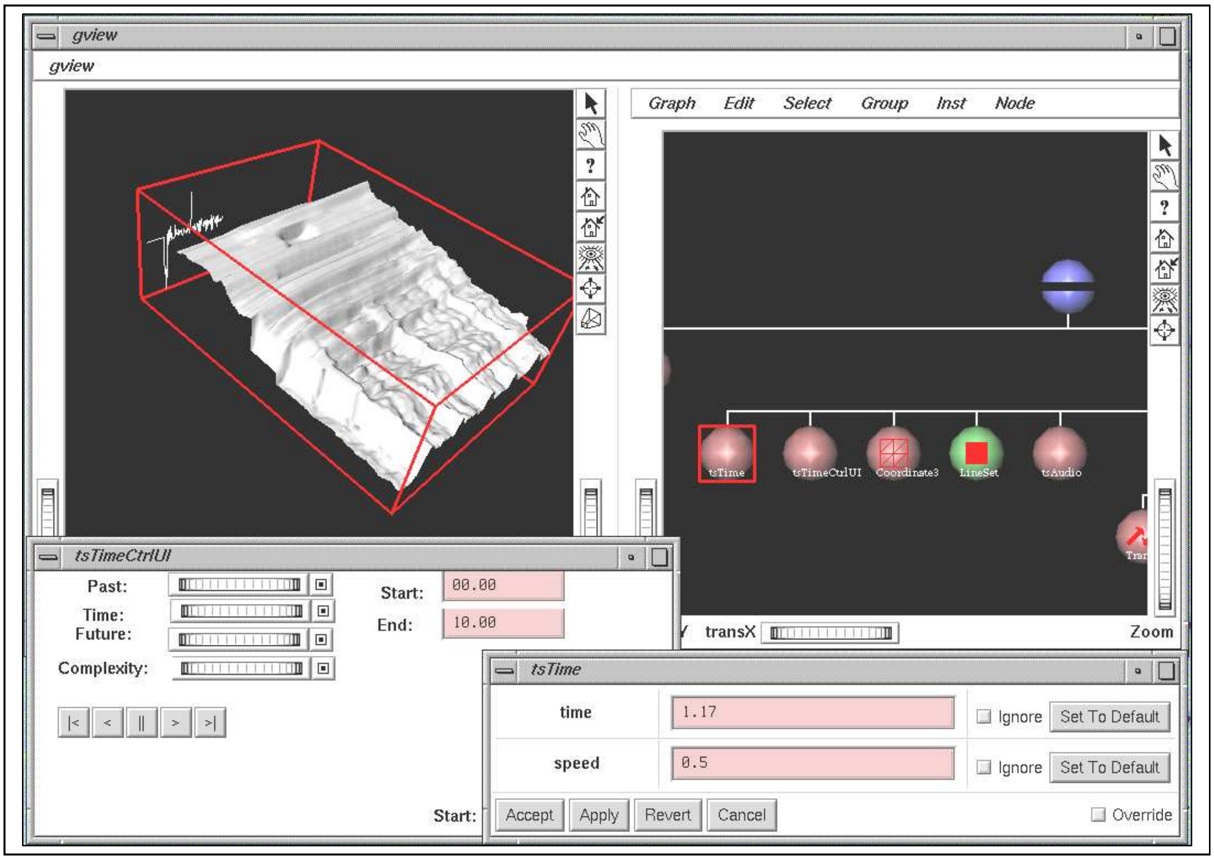


Figure 4.11: This figure shows the viewer `gview` with a scene graph loaded. The right window shows the structure of the scene graph itself, the left window its rendered version. The selected node is rendered with a bounding box. The node field's values are edited in the generic dialog window 'tsTime' (right, bottom). Additionally the scene graph contains a `tsTimeCtrlUI` node that generated the dialog window 'tsTimeCtrlUI' (left, bottom).

Time can be a frame that is dragged along the time axis of a `tsLine`/`tsSurface` node.

4.4.7 Feedback for Time Signals

The final feedback, whether it is visual or auditive, is rendered in new nodes of the tsKit, called *feedback* nodes. Their common feature is to provide feedback for time signals. All feedback nodes use data from tsData nodes, time/speed from the `tsTime` node and several other properties from other nodes to render them in their specific way. These feedback nodes contain their specific way to produce a feedback of given signals.

Until now there are nodes to render one-dimensional signal data to graphics and audio and two-dimensional signal data to graphics only.

`tsLine` – Drawing a Line Strip

This node uses the current `tsTime` and tsData1 node (`tsData1RFloat` or `tsData1NFloat`) to render the signal as a line. The signal data is directly mapped to the height of the line strip over time.

See figure 4.12 for an example.

The node class `tsLine` contains the following fields:

`SoSFFloat past` : Field containing the period of the signal that is rendered, ending at the current point in time. If value is negative, all past signal is rendered.

`SoSFFloat future` : Field containing the period of the signal that is rendered, starting at the current point in time. If the value is negative, all future signal is rendered.

`tsAudio` – Playing Audio

This node uses the current `tsTime` to render an audio stream at the current time and speed of the `tsTime` node.

The node class `tsAudio` contains the following field:

`SoSFFloat IORateRatio` : Field storing the ratio of the sample rate of the audio file and the output sample rate. It is often necessary to reduce the computation afforded by setting this field to a value of .5 or even to .25. This includes a

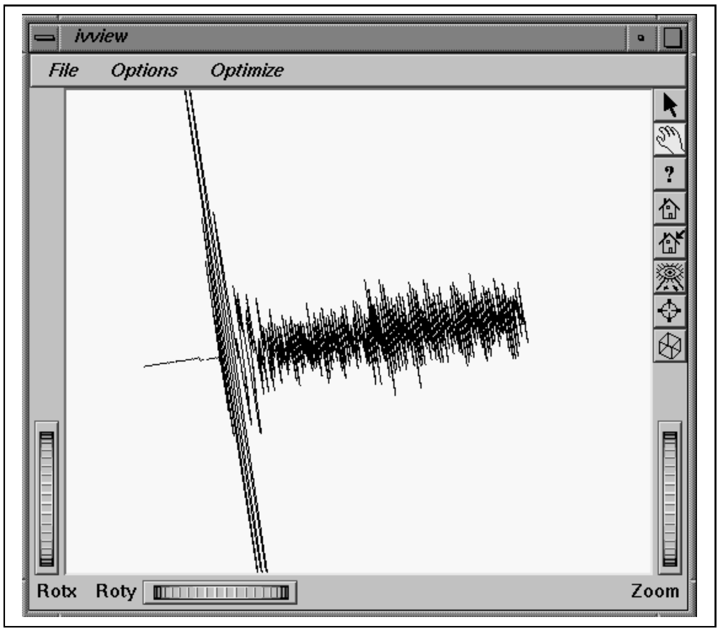


Figure 4.12: A `tsLine` node drawing data from a file. The signal data comes from a node of class `tsData1RFloatFile` and was created reading data from an AIFF file. See 4.9 for the file describing this scene.

reduction of the output's bandwidth. Otherwise a playback without drop outs is not possible.

To make the implementation easier the software *scrub* (see destion 4.3.3) is reused, it can only can read the signal data from a file. The field 'filename' from a `tsData1RFloatFile` node is used for accessing an audio file.

`tsSurface` – Drawing a Surface

This node uses the current `tsTime` and tsData2 (`tsData2RNFloat[File]`) node to render a surface representing the signal. The signal values are directly mapped to the height of the surface.

This node computes normals if necessary and uses the current materials in `SoMaterial` or `SoPackedColors`. Figure 4.15 shows how the normals are computed. The normals of the surface are computed before the first rendering. The computation is *not* done by employing an existing helper object `SoNormalGenerator` providing this service, because the normals are computed as vectors of floats. See section 4.4.8

```
#Inventor V2.1 ascii
Separator {
    PerspectiveCamera {}
    tsKit { }
    Separator {
        tsTime {
           time        1.17
           speed       .5
        }
        tsTimeCtrlUI { }
        Material {
            diffuseColor  0 0 0
        }
        LightModel {
            model       BASE_COLOR
        }
        tsData1RFloatFile {
            startTime   0
            interval    0.0001
            filename "../audio/massiveA.pianoCool.mono.5s.aifc"
        }
        Transform {
            scaleFactor 36.67 27.32 2
        }
        tsLine { }
    }
}
```

Figure 4.13: An Inventor file drawing audio data with a `tsLine` node as a line strip.

for how tsKit deals with normals.

The node class `tsSurface` contains the following fields:

`SoSFFloat past` : Field containing the period of the signal that is rendered, ending at the current point in time. If value is negative, all past signal is rendered.

`SoSFFloat future` : Field containing the period of the signal that is rendered, starting at the current point in time. If value is negative, all future signal is rendered.

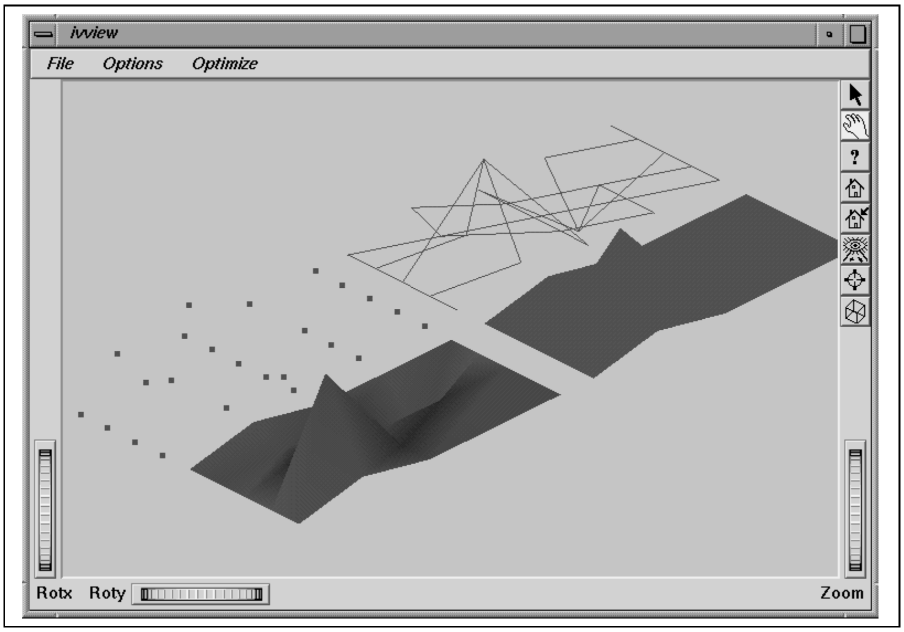


Figure 4.14: Four `tsSurface` nodes drawing the same `tsData2RNFloat` data, with different draw styles and light Models(`SoDrawStyle::POINTS`, `SoDrawStyle::LINES`, `SoLightModel::PHONG` and `SoLightModel::BASE_COLOR`).

4.4.8 Time Signal Shapes with Normals

As one way to render shapes more realistic, their surfaces are shaded. Shading is coloring a shape's surface depending on the intensity and angle of incidence it is lightened. Open Inventor offers the phong[30] lighying algorithms[31]. Like all other realistic lighting algorithm, phong shading needs to know how the surface and its polygons are oriented. This information is expressed as normal vectors (short: normals). These normals can be computed automatically from the geometry of a shape, given as polygons.

Before it is described how tsKit deals with normals, the native policy of Open Inventor is given.

[30] Phong lighting is a shading algorithm that calculates the shaded color of a surface separate for every pixel. It calculates the color values for every pixel of a polygon by interpolating the normals of the corresponding vertices.

[31] In Inventor shading model is not explicitly specified. It is dictated by the material lightning model and normal properties

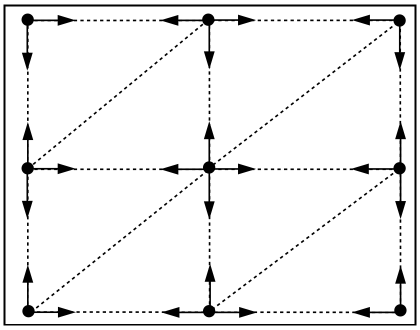


Figure 4.15: This figure shows which vectors (arrows) are used to compute the normals for the vertices (small circles) of a 3x3 `tsSurface`. The dottet lines shows the border lines of the triangle polygons that build the surface.

How Inventor deals with Normals

The drawing of shapes is very different, whether the shape is a vertex shape (also called complex shape) or a parametric shape (also called simple shape) like a cone (node class `SoCone`). The geometry of vertex shapes is completely described by coordinates of vertices. These vertices are stored in coordinate nodes (`SoCoordinate3`, `SoCoordinate4`). Vertex-shapes interprets these data in different ways to draw their geometry. Parametric shapes are specified by parameters. A parametric shape interprets these parameters, e.g., the radius for a cube (SoShere), and generates its geometry by internal calculations.

All shapes have to use normals for drawing with phong shading. When they have not fixed normals (e.g., cube), they have to be computed, before drawing. There are objects (`SoNormalBundle`, `SoNormalGenerator`) that do the computation of the normals for a vertex-shape, according to a crease angle (`SoCreaseAngle`, `SoCreaseAngleElement`). Normal vectors are stored in a `SoMFVec3f` field of a `SoNormal` or`SoVertexProperty` node. These normals are stored as 3 single precision floats using 3 * 4 Bytes = 12 Bytes.

How tsKit deals with Normals

To keep the signal representation *and* its rendering memory efficient, tsKit does not use the native concept for calculating and storaging normals described above. Because the normal information is less sensitive to inaccuracy than geometry information, it can be stored (and send to GL for rendering) only as 3 short integers instead of 3 floats.

The alternative to compute the normals while the shape is rendered is assumed to be much too expensive in computation. The computation would be very redundant, because for two different render traversals most normals to compute are the same!

A new set of a node and an element is created, that store 3 byte normals in a `SoMFInt32` field, until a field of class `SoMFVec3b` is created in future work. The class of the node storing the data is called `tsNormal` and is derived directly from `SoNode`. Because the class `tsNormal` does not contain a time signal it is not derived from `tsDataBase`.

To store the normal data of this node in the state, there is a new element called `tsNormalElement`. This class stores a reference to the `tsNormal` node.

tsKit Normals and OpenGL-Performance

It should be mentioned, that the memory efficient storage and rendering of normals as vectors of bytes involves cost in rendering performance, as documented in chapter 14 of [21]:

> *Using Geometry Operations Effectively*
>
> *If your application seems transform limited, you can improve it by considering the tips in this section. The section starts with some general points, then discusses optimizing line drawing and using triangles and polygons effectively.*
>
> *To improve performance in the geometry subsystem, follow these guidelines:*
>
> - *Use single-precision floating point parameters for vertices, normals, and colors.*
> - *Transform paths use single-precision floats is fastest to use glVertex3fv() and glVertex2fv().*
> - [....]
> - *Perspective transforms that require multiplication by 1/W or division by W are much slower.*
> - *To minimize the time used for floating point conversion, use floating point coordinates where possible, except where memory size is critical.*
> - *Don't enable normalizing of normals if the modelview matrix doesn't include scaling and if you have unit-length normals*

In spite of this, there was no measurable loss in performance.

Caching

In Open Inventor normals are cached for rendering, so that normals are only computed once and shared between shapes, that use the same geometry data. Because this mechanism is not fully documented and there is no source code of nodes using this mechanism, tsKit implements a separate and simpler way to cache normals.

Until now only the feedback node `tsSurface` needs normals to be phong shaded. On the traversal of the `tsSurface` node it is tested if there are normals stored in the `tsNormalElement` that can be used for rendering. Otherwise the normals are computed, stored in a new object of class `tsNormals` and set in the `tsNormalElement` in the state. Normals are reused for a `tsSurface` node, if the number of normals fits its demand (The number normals in the `tsNormalElement` object is equal number data values in the field 'data') and the element's type information is equal or derived from `tsSurface`! Every surface node also saves a pointer to the `tsNormals` node, storing its normals.

4.4.9 Colors and Transparency

Using color and transparency as a visual dimension is extensively used in all fields of visualization. It should be well supported in the tsKit to use color and transparency as a visual representation of signal features in terms of efficiency and easiness of use.

tsKit uses the Inventor standard nodes/elements (`SoPackedColors` and `SoMaterialBinding`) to store and map colors and transparency values to a shape. A memory efficient way to store the diffuse colors and the transparency value, is to pack them into on 32 bit integer (4 bytes) in the hexadecimal form: 0xrrggbbtt, where the rrggbb part stands for a RGB color[32], the tt part stands for a 8 bit transparency value. This is called a packed color in Inventor.

The more memory-expensive way to store colors and transparency values for every facet, vertex or part, is to use the `SoMfVec3f` fields of the `SoMaterial` node for diffuse color (diffuseColor) and transparency (transparency). This way needs 4 floats = 4 * 4 bytes = 16 bytes for each color/transparency pair!

The actual computation of color/transarency values is ment to be done with a engine of class `tsData2ToPackedColorEngine` (See section 4.4.10).

[32]This is a way to describe a color as its additive components red, green and blue.

4.4.10 Transforming Time Signals

Transforming one or more time signals into another time signal is an important need and concept of this work. Signal transformations are used to realize the central function of this system: a transformation from an audio signal into a visual signal!

Transformation of time signals is realized using the Inventor's concept of engines (see section 4.2.3 for an introduction.). The idea of an Engine is to transform one more input fields to one or more output fields, whenever the input fields have changed their values. When notified they read the input field(s), do something with this field-data and write something to their output fields. Before they can access a field (whether read from an input or write to an output field), this field must be connected to a field of another node, engine or global fields. All information an engine needs, must be accessible by the fields, the engine is connected to.

Inventor engines work on fields. In the tsKit all signal representations are stored in nodes derived from `tsDataBase`. For this reason it is be more convenient to transform tsData nodes instead of fields as entities for the operation. When the user wants to transform different types of signals, he does not need to know the representation in memory or what fields the nodes contain! Therefore the engine concept of Inventor is abstracted to deal with signals, rather then with fields. All classes derived from `tsDataBase` contain a field 'node' of type `SoSFNode` containing a reference to the node itself. This field functions as a handle for the complete node. That way tsData nodes can be connected using this field and engines can access the entire node, including all node specific methods!

Engine for computing Colors and Transparency Values

The engine `tsData2ToPackedColorEngine` is only partly implemented until now. This engine computes the colors and transparency values for a tsData2 node according to an internal algorithm. It takes a tsData2 node as input and computes entries in the connected output field of class `SoMFInt32`. It is assumed that this output is connected to the field 'orderedRGBA' of a node of class `SoPackedColor`!

Because of unsolved programming problems the engine is not running yet!

4.4.11 Control a Selection

A topic not yet covered by the tsKit is the selection of the data to be visualized. There are several reasons for selecting a subset of data for being rendered:

- Selection lets scale the demands of processing power: Selecting a subset of signal data is a way to adjust render quality to the available processing power.

  Another mechanism to scale rendering demands is to control the property node `SoComplexity`. This node contains a float value, that lets feedback nodes adapt their complexity absolute or relative to the size of screen they occupy.

  The property complexity is not yet evaluated by nodes of the tsKit!

- Realistic rendering of time: Rendering only a certain interval of the dimension time around the actual content of the property `tsTime` is reasonable choosen, because the auditory system also only has a limited period in which it can integrate auditive features. This selection is a relative selection to the render property time of node `tsTime`. It can be expressed by two values 'past' and 'future'. There should be a selection mechanism that covers various selection in future work.

  Until now this selection is done with the fields 'past' and 'future' of the feedback nodes `tsLine` and `tsSurface`, but will be put in a separate node and element.

- Efficient use of screen space: Selection can be used to focus on parts of interest and to prevent the scene of getting overloaded with unwanted information.

Chapter 5

Conclusions

5.1 Conclusions

At the end of this thesis the most goals of the project are reached. Concepts are formulated and implemented in a running prototype, proofing their quality. The work is a good foundation for further development cycles.

The following subjects are covered conceptual but have not been implemented yet:

- Engines for transforming signals and computing color/transparency maps.
- Manipulators for direct manipulation of signal rendering properties.

To summarize my experience made during the project, I observed a shift in cost of development. This work is the first time, I did intensive *white box* reuse of software *and* design of the framework Open Inventor. The time I spend less by reusing software, I might spend more figuring out, how to map the problem on the given concepts and mechanisms most efficiently and by debugging my code debugging. The more complexity one imports in a project, the more it gets difficult to handle its implications. This problem becomes important with reuse of architecture and collaboration mechanisms of a framework, rather than with using localized functionality by calling functions of a library. It takes a long time to build robust class libraries for basic data types, that are generic enough for practice[1].

I am still fascinated by the idea of Open Inventor's object oriented software technology, the way Open Inventor [1] ,,[...] *separates the generic and the specific*

[1]Examples are STL (Standard Template Library), Tools.h++.

parts of a solution and structure the generic parts as collaborating objects". But I see a demand for improvements of the user interface of frameworks. Frameworks can have very complex contracts and protocols between objects that builds its semantic. A new object class can easily violate this semantic, because the framework user can hardly know all of its constrains. I think frameworks suffers from the limited way they can verify if the user satisfies the contracts of the framework's semantic.

Although there is a debug version of the Open Inventor library (i.e. the dynamic library called DSO (dynamic shared object)), that has build in additional verifications, it showed that this mechanism is no protection of runtime errors, that could only be located or worked around with external background knowledge.

I follow Ackermann in [1], p. 44:

> *"In a framework, typical parts of a problem-domain are modeled as abstract classes. The abstract classes operate with each other, which means the framework defines the control flow. Developers who use frameworks need to know where to add their application-specific extensions, which class must derived, and which method should be overwritten so that the control flow of the framework calls them. Otherwise, they will not benefit from reuse and will implement their own solution from scratch. Unfortunately, the discussion of frameworks is complicated by the fact that no really good documentation techniques have been found for them, something that makes it difficult to learn a new framework [...]. Design Patterns are a first attempt to inform programmers about the abstractions used in a framework."*

See [9] or the 'Patterns Home Page'[2] for more about Design Patterns.

It seems that frameworks need a semantic check of their use, not only a syntactic check on the level of the programming language (C/C++).

I wished there were two ways the extension of Open Inventor, i.e. deriving of new classes, would be better supported:

1. Explicit description of the abstractions, e.g., in terms of design patterns [9]. I needed a long time to factor out the generic abstractions, to extend the problem domain of non graphic data and behavior.

2. Example code for all Open Inventor classes. Source code of standard classes would be the best demo code, because their behavior is well known and they

[2] at `http://st-www.cs.uiuc.edu/users/patterns/`

are realistic in using the complete framework's infrastructure. The examples given for Open Inventor extensions in [24] mostly deal with very simple cases only. I could find useful example code in the work mentioned in 3.3.1, a Open Inventor extension done as a masters thesis.

The source code of classes of a framework is still the most complete documentation of a framework's usage. Source code is unhandy for documentation purposes because it requires a detailed understanding of all mechanisms, wheather they are importand for application or not.

An unrenouncable help is the news-group `comp.graphics.api.inventor`. I think it is not possible to extend Inventor without this source of tips and discussion.

5.2 Limitations, known Bugs and Future Work

This section summarizes all known bugs, limitations and ideas for future work. Because all bugs are limitations, and all limitations are possible subjects for future work, this is all listed in one section.

5.2.1 Limitations and known Bugs

Normal-Field: The `tsNormal` node is implemented with the field of type SoSFUInt32 to store a vector of 4 signed bytes (although only 3 bytes are used). Future work should implement a new field class for this data, called SoSFVec4b. Because the user usually never has to deal with normals, this fact stays hidden.

`tsLine`: When audio data is subsampled, rendered period is of wrong length.

Antialiasing: `tsRenderer` does not render with Antialiasing enabled! Inventor bug?

Render-Caching of tsSurface: The `tsSurface` is never or always cached. Problem lies in the `tsTimeElement`.

Camera's near-distance: There must be a camera in the .iv file, to set the 'nearDistance' to 0.0, otherwise parts of the model are clipped (default setting is 1.0)!

Conversion to standard file format: There are several converters from the standard Open Inventor file format to many CAD and raytracing systems. To use these the tsKit-specific nodes must be converted to a standard representation on Open Inventor standard nodes.

5.2.2 Future Work

Algorithm Editing while Runtime: Algorithms of nodes or engines cannot be changed while runtime. Because they are written in C++, they must be compiled and linked between editing and usage. With the dynamic linker (`rld`) objects can be reloaded and linked at runtime. Using dynamic linking a) all ts nodes and engines could be reloaded or b) only special classes.

Automatic Layout: The temporal and spatial layout of feedback nodes can be controlled automatically for a group of nodes. New nodes derived from `SoGroupNode` can automatically control the feedback dimensions space and time by controlling the properties (`SoTranslation` and `tsTime`) of their children. There could be nodes that sequence their children or render them parallel in space or time.

In combination with editors for these layout-nodes tsKit became a general sequencer and authoring tool for time signals.

Using SoComplexity: Feedback nodes should take the property `SoComplexity` into account.

Two-dimensional Rendering: It might be useful to have graphical feedback nodes rendering only a two-dimensional shape of a signal. Signals represented in a tsData2 node can be rendered as a simple line. Combined with the next extension they could be viewed in a `SoXtPlaneViewer`. A `SoXtPlaneViewer` lets the user view a scene only orthogonal to the X-Y plane.

Viewer-Nodes: It might be useful to view a part of the scene graph in a separate viewer, parallel to viewing it with the complete scene graph. To allow this feature by scene graph layout, a new (group) node `tsViewer` could be created, that would build a viewer to view the subsequent subtree. A field can contain the kind of viewer to be used.

Using Overlay Planes: Viewers can render a second scene graph overlayed on the normal scene graph. Its background is transparencte. This scene graph is limited in use of colors and should be very simple. This overlay scene graph can be used to show data like current time and speed or simple 2 dimensional displays of signals (see item 'Two-dimesnional rendering' above). A part of the scene graph can be marked with a label node (class `SoLabel`) as the overlay scene graph.

Extending gview with engines: It should be able to create and delete engines in `gview` the same way it is possible for nodes now. They also should be visualized as icons like nodes in a schematic scene graph. Those input fields of engines that are not connected to other fields should be editable like node fields. This way engines could be configured easily.

Time Cursor: Feedback nodes normally visualize an interval of time. Therefore they have to mark where the actual point in time is visualized. The actual time in point is determined by the field 'time' of the `tsTime` node and is hearable, when a audio feedback is generated with a node of class `tsAudio`. There could be a new node/element combination storing the way the actual point in time is visualized: Node: `tsTimeCursor` with field 'mode' of type `SFEnum`, element name `tsTimeCursorElement`.

Video and Image Processing: A new node type 'tsVideo' rendering a sequence of frames could be introduced: `tsDataImage` and its file version `tsDataImageFile` storing a sequence of frames in memory. `tsDataImageVideoIn` could contain the actual frame from a video source. Becasue movie file images are very memory intensive, only a couple of frames around the actual point in time (`tsTime.time`) are held in memory. A feedback node `tsVideo` shows the image-stream in a window and/or sends it to video out.

New engines (`tsImageProcessingEngine`) could take one or more nodes of type `tsDataImage` and apply various kinds of image processing to them. The algorithm is described as a chain of ImageVision[3] or OpenGL operators. OpenGL operators can be processed in real-time. The chain is described in an input field of type string. For convenience this field is edited in a SoXtComponent `SoXtImgeProcessingChainEditor` or read from a file. The Editor uses ImageLab[4] as a comfortable environment for editing image processing chains.

[3]ImageVision is the image processing library of SGI and offers building block for writing image processing Applications on a similar level to Open Inventor. See `http://www.sgi.com/Technology/ImageVision/` for further information about ImageVision.

[4]See `http://www.cevis.uni-bremen.de/ilab`

Once the image engine is evaluated in off-real-time the result in the output field can be played back in real-time. There could be a separate action `SoImageRenderAction` for rendering all nodes related to images.

Generalized Audio Rendering: For audio there could be an extra action `AudioRenderAction` for rendering different kinds of audio represenation nodes as sounds (this would replace the `tsAudio` node!). Here are possible new nodes for storing audio that are rendered by the audio render action:

tsDataTracks: This node contains tracks describing frequencies and amplitudes of sinusoidal partials of a sound over time. The real-time additive synthesizer `softcast`[5] developed at CNMAT is used to render this handy represenation of audio. A new feedback node `tsTracks` renders this data to a graphical representation, e.g., as multiple line strips.

tsDataResonance: This node contains resonance models[6]. This data is also synthesized using the `softcast` synthesizer mentioned above.

tsDataMIDI: This node contains MIDI[7] data. This data is rendered by the SGI's build-in MIDI-synthesizer or send to connected Synthesizers.

In combination with new manipulators for these nodes, the tsKit became a 3D editor for audio.

[5] see `http://www.CNMAT.Berkeley.EDU/CAST/`

[6] See `http://wwww.CNMAT.Berkeley.EDU/Research/Resonances/`

[7] MIDI is a widely used serial protocol for communicating musical events e.g., between keyboards and synthesizers. MIDI stands for Musical Interchange Digital Interface

Appendix A

Results

A.1 Software

The software tsKit runs on SGI machines and requires IRIX 6.3 or higher.

The only tested hardware platform until now is O2.

See my homepage at CNMAT:

`http://www.CNMAT.berkeley.edu/~andreas`

for

- an online version of this thesis and '*The tsKit Reference Manual*'.
- downloading actual software and demos (images, movies, data sets).

or contact me per e-mail: `andreas@cnmat.berkeley.edu` .

There is an CD containing all documents, software,
demo data sets and animations.

Bibliography

[1] Philipp ACKERMANN,
Developing Object-Oriented Multimedia Software – Based on the MET++ Application Framework
dpunkt Verlag, 1996

[2] Philipp ACKERMANN,
Design and Implementation of an Object-oriented Media Composition Framework,
In *International Computer Msuic Conference ICMC 94.//* International Computer Music Association,1994.
ftp://ftp.ifi.unizh.ch/pub/projects/met++/papers/ICMC94.ps.Z

[3] Philipp ACKERMANN,
Direct Manipulation of Temporal Structures in an Object-oriented Multimedia Application Framework
In *ACM Multimedia 94 Conference Proc.*.
ACM, October 1994.
ftp://ftp.ifi.unizh.ch/pub/projects/met++/papers/ACMMM94.ps.Z

[4] Rudolf ARNHEIM , *Visual Thinking* .
Berkeley and Los Angeles: University of California Press,1969.

[5] Albert BREGMAN,
Auditory Scene Analysis
Cambridge, MA: MIT Press, 1990.

[6] edited by Martin COOKE , Steve BEET, and Malcolm CRAWFORD,
Visual representations of speech signals
New York, J. Wiley & Sons, 1993.

[7] Daniel P.W.ELLIS, B.L.VERCOE,
A perceptual representation of sound for auditory signal separation, Presented to the 123rd meeting of the Acoustical Society of America, Salt Lake City, May 1992.
`ftp://sound.media.mit.edu/pub/Papers/dpwe-asa92slc.ps.gz`

[8] Daniel P.W.ELLIS,
A Perceptual representation of audio, Thesis, MIT, 1993.
Paper at:
`ftp://sound.media.mit.edu/pub/Papers/dpwe-ms-thesis.ps.gz`
Software at:
`ftp://sound.media.mit.edu/pub/Code/dspB-1994may27.tar.gz`

[9] E. GAMMA, R. HELM, R. JOHNSON, and J. VLISSIDES.
Design Patterns,
Addison Weseley, 1994.

[10] W GRAVER,
The Sonic Finder: an interface that uses auditory icons,
Human Computer Interaction, vol. 4, 1989.

[11] Shuhei HOSOKAWA,
Der Walkman-Effekt,
pp. 229-251 ,*Aisthesis, Wahrnehmung heute oder Perspektiven einer anderen Ästhetik*,
Reclam Verlag Leipzig, 1990.

[12] Shuhei HOSOKAWA,
Walkman no Shujigaku (Rethoric of Walkman),
(Tokio) 1981 in *International Review of the Aesthetics and Sociology of Music*, XII:1.

[13] Silicon Graphics, Inc.,
IRIS Explorer Users Guide,
`http://techpubs.sgi.com/library/lib/makepage.cgi?007-1371-030`

[14] Silicon Graphics, Inc.,
IRIS Explorer Module Writer's Guide,
`http://techpubs.sgi.com/library/lib/makepage.cgi?007-1369-030`

[15] Gene YONGBLOOD,
Cinema and the Code,
LEONARDO, Computer Art in Context Supplemental Issue, pp.27-30, 1989.

[16] Puckette MILLER,
Combining Event and Signal Processing in the MAX Graphical Programming Environment.,
Computer Music Journal, 15(3):58-67. 1991

[17] Silicon Graphics, Inc., *ViewKit Programmer's Manual.*
`http://techpubs.sgi.com/library/lib/makepage.cgi?007-2124-003`

[18] Doug YOUNG, Young, Silicon Graphics, *ViewKit Library Technical Report.*
`http://www.sgi.com/Technology/C_Motif_paper.html`

[19] Alan WESLEY, *A Real-Time 3D Signal Analysis/Synthesis Tool Based on the Short Time Fourier Transform.*
Master thesis,
Department of Electrical Engineering, University of California, Berkeley.
`http://www.CNMAT.Berkeley.EDU/~alan/MS-html/MSv2_ToC.html`

[20] Silicon Graphics, Inc., *RapidApp Programmer's Manual.*
See at `http://techpubs.sgi.com/library/`

[21] Silicon Graphics, Inc., *OpenGL on SGI Systems*,
Addison Weseley.
See at `http://techpubs.sgi.com/library/`

[22] Silicon Graphics, Inc., *The Inventor Mentor*,
Addison Weseley, (ISBN 0-201-62495-8).

[23] Silicon Graphics, Inc., Josie Wernecke, *Summary of The Inventor Mentor*,
`http://www.sgi.com/Technology/Inventor/VRML/TIMSummary.html`

[24] Silicon Graphics, Inc., *The Inventor Toolmaker*,
Addison Weseley, (ISBN 0-201-62493-1).

[25] Silicon Graphics, Inc., *Inventor Reference Manual*,
Addison Weseley, (ISBN 0-201-62493-1).
See also at `http://mimsy.mit.edu/Transom/inventorman.html`