Bakkalaureatsarbeit

# Automatic Generation of Graphical User Interfaces in Studierstube

ausgeführt am:
Institut für Softwaretechnik und Interaktive Systeme
der Technischen Universität Wien

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

unter Mitbetreuung von:
Dipl.-Ing. Thomas Psik

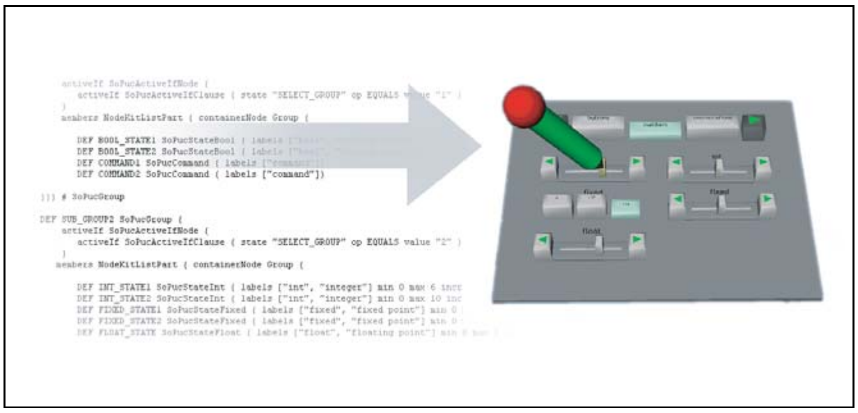
durch
Valérie Maquil
Matrikelnummer 0126539

Wien, am 14. Juli 2004

Valérie Maquil

# Automatic Generation of Graphical User Interfaces in Studierstube

Bachelor Thesis



```
    activeIf SoPucActiveIfNode {
       activeIf SoPucActiveIfClause { state "SELECT_GROUP" op EQUALS value "1" }
    }
    members NodeKitListPart { containerNode Group {

       DEF BOOL_STATE1 SoPucStateBool { labels [
       DEF BOOL_STATE2 SoPucStateBool { labels [
       DEF COMMAND1 SoPucCommand { labels ["command"]}
       DEF COMMAND2 SoPucCommand { labels ["command"]}

}}} # SoPucGroup

DEF SUB_GROUP2 SoPucGroup {
    activeIf SoPucActiveIfNode {
       activeIf SoPucActiveIfClause { state "SELECT_GROUP" op EQUALS value "2" }
    }
   members NodeKitListPart { containerNode Group {

       DEF INT_STATE1 SoPucStateInt { labels ["int", "integer"] min 0 max 6 incr
       DEF INT_STATE2 SoPucStateInt { labels ["int", "integer"] min 0 max 10 inc
       DEF FIXED_STATE1 SoPucStateFixed { labels ["fixed", "fixed point"] min 0
       DEF FIXED_STATE2 SoPucStateFixed { labels ["fixed", "fixed point"] min 0
       DEF FLOAT_STATE SoPucStateFloat { labels ["float", "floating point"] min 0
```

Performed at the

Institute for Software Technology and Interactive Systems
of the Vienna University of Technology

Supervised by

Ao. Univ. Prof. Dipl.-Ing. Dr. techn. Dieter Schmalstieg
and
Dipl.-Ing. Thomas Psik

Vienna, 2004

## Abstract

Studierstube is a system build to develop collaborative augmented reality applications. Interaction in this system is realised by means of the Personal Interaction Panel (PIP), composed of a notebook-sized handheld panel and a pen. Traditional interaction elements are projected on this panel to enable widget interaction and parameter manipulation.

To create a user interface on this panel, each interaction object needs to be placed individually on the panel with an explicit transformation. This approach however implies that creating and changing the layout of a user interface takes a lot of time.

This thesis presents a new system able to automatically generate graphical user interfaces in Studierstube. Therefore, all application's functions need to be specified by means of a tree of state variables and commands, whose structure is then analysed in order to compute a GUI for this application. The process used to generate the GUI is adopted by the PUC system presented by Nichols et al. in [4] [3].

## Kurzfassung

Studierstube ist ein System, das zur Entwicklung kollaborativer Augmented Reality Applications erstellt wurde. Die Interaktion in diesem System wird durch das Personal Interaction Panel (PIP) ermöglicht, welches aus einer leichten Platte in der Größe eines Notebooks und eines Stiftes besteht. Auf diese Platte werden dann traditionelle Interaktionskomponente projeziert um so eine Veränderung der Parameter zu ermöglichen.

Um eine Benutzerschnittstelle für diese Platte zu erstellen, muß jedes einzelne Interaktionsobjekt anhand einer expliziten Transformation plaziert werden. Durch diesen Ansatz jedoch wird das Erstellen und das Verändern graphischer Benutzerschnittstellen sehr zeitaufwendig.

Diese Arbeit beschreibt ein neues System, das ein automatisches Erstellen von graphischen Benutzerschnittstellen in der Studierstube ermöglicht. Hierfür werden alle Funktionen der betreffenden Applikation anhand eines Baumes aus Variablen und Befehlen spezifiziert. Die Struktur dieses Baumes wird dann analysiert um eine Benutzerschnittstelle für die Applikation zu erstellen. Der Prozess, der hierfür verwendet wird, basiert auf dem PUC System, beschrieben von Nichols et al. in [4] [3].

# Contents

## 1 Introduction

The concept of Augmented Reality (AR) describes the combination of the real world with a computer generated virtual environment in a real time interactive manner. Wearing see-through glasses, the users are able to perceive the real world surrounding them, with virtual objects superimposed upon it. Exact alignment of both environments in 3D creates the illusion of virtual objects coexisting with real ones. This concept thus aims, in opposition to virtual reality, an enhancement of the reality and not a replacement of the latter.

The computer graphics institute of the University of Technology in Vienna has developed the AR system Studierstube, which is an environment for the development of collaborative augmented reality applications. An important property of this system is that it allows multiple users to interact in the same virtual 3D workspace. Furthermore, the workspace is not limited to a single application and it supports multitasking so that multiple users can use simultaneously multiple applications in the same workspace. Presentation of the virtual scene is done using see-through head mounted displays or back-projection display surfaces like the virtual table (VT).

Research in the field of augmented reality is subsequently done on different system setups and potential applications. The possible range of applications includes medical visualisations, industrial design, annotations, educational and entertainment setups. Figure 1 shows some examples of such applications. *SignPost* is an augmented reality application that is able to guide a person through an unfamiliar building, the *Virtual Construction Kit* allows you to quickly build an architectural model with buildings, trees and people, and *Construct3D* is a three dimensional geometric construction tool specifically designed for mathematics and geometry education.

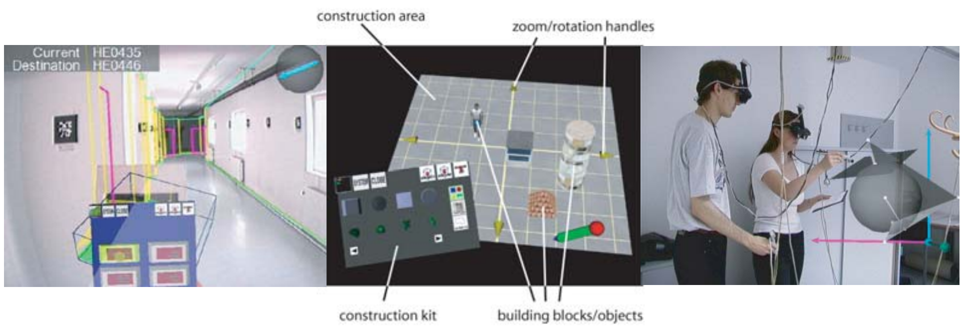


Figure 1: Recent research applications developed for Studierstube: SignPost, Virtual Construction Kit and Construct3D.

Further information about Studierstube, its properties and the current research projects are provided in [5] and on the world wide web (www.studierstube.org).

## 1.1 Interaction tools

Interaction with virtual objects in the augmented part of Studierstube is done with the Personal Interaction Panel (PIP). It is composed of a lightweight, notebook-sized handheld panel and a pen. Both of these components are tracked in position and orientation, so that their physical properties correspond to the feedback provided by the PIP elements in the augmented scene.

The pen alone can be moved in six degrees of freedom and can thus handle any 3D pointing operations or direct manipulations. The panel is virtually augmented by a projection of traditional interaction elements to enable widget interaction and parameter manipulation. These interaction elements may also be extended for 3D manipulations including 3D widgets, clipboard functionalities and drag-and-drop in 3D.

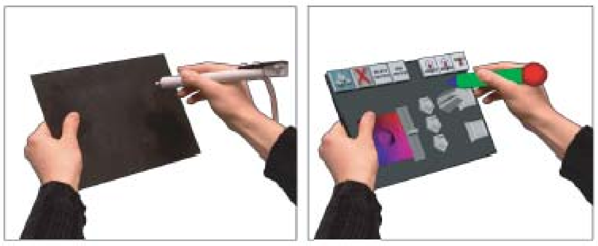


Figure 2: The personal interaction is a pen-and-pad combination that is overlaid with graphics

For use with the PIP, simple 3D widgets have been implemented: buttons, sliders and list boxes. These widgets are based on their 2D desktop counterpart: properties that were familiar have been kept, while their behaviour has been extended to make them controllable with the PIP. Figure 3 shows examples of the current widget implementations.

## 1.2 Problem statement

As the PIP is used as interaction tool in Studierstube, its user interface may be treated similarly to the one of a traditional desktop computer. Widgets are placed on a plane surface of a certain size where position and size of the different components are crucial for the quality and aesthetic look of the

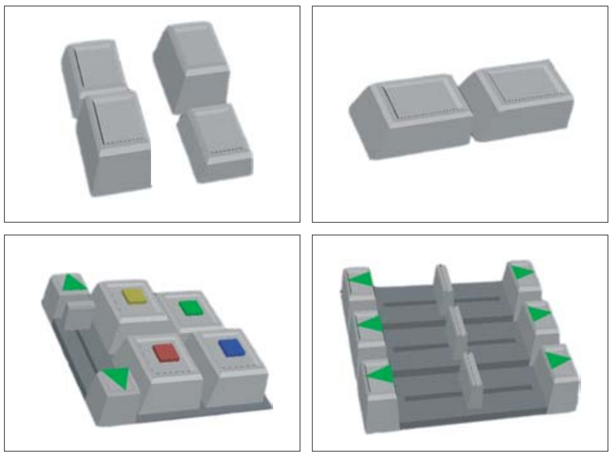


Figure 3: Simple 3D widgets have been implemented for use with the PIP: toggle buttons, push buttons, list boxes and sliders.

user interface. Visual techniques like simplicity, grouping and alignment give some hints about how to construct a graphical user interface; all decisions concerning the resulting layout however have to be taken by the designer.

As for conventional applications, the whole process of building a GUI for Studierstube consists of the following steps [1]:

1. access details of the data model
2. determine how each element of the data model is to be mapped into a control widget
3. access and apply layout rules to position and size each component

In the current system, each of these steps needs to be carried out by the programmer, what makes this procedure very time-consuming and thus difficult to expand or modify. Small changes at the level of the data model lead to a complete new generation of the user interface. By taking a closer look to the steps explained above, we notice that part of their tasks could be automatised by using layout style rules following certain visual techniques.

The aim of this research was to develop an automatised mechanism for Studierstube providing more facility in the generation of a GUI. Our system actually automatises the two last steps of the above generation process and creates and lays out the PIP interaction elements. It does this by reading a textual specification of the appliance's functions, including a high-level description of every function and a hierarchical grouping of those functions. In addition to this specification, hints concerning layout and style of the user interface are given. With all these information a GUI able to control that application is automatically created.

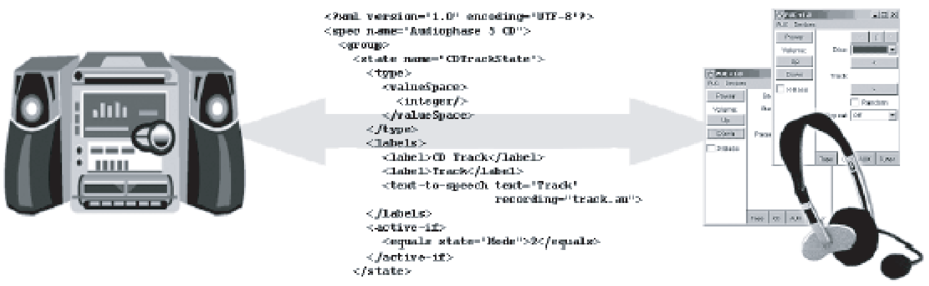
We shall first describe related systems that have been used as base for the development of our system. After that, we shall explain how parts of the presented systems have been used to develop our framework for Studierstube and give details about its implementation. Finally, we shall discuss the results and describe some possibilities to extend the system.

# 2 Related work

Quite a number of research groups are working on how to automatically generate graphical user interfaces. The personal universal controller (PUC) is a system able to control appliances from handheld devices by generating a user interface for this device from a textual specification of the appliance's functions. The specification of this PUC system contains a lot of useful indications concerning the architecture and the specification language. The generation of user interfaces for the PIP is however more complicated as for handhelds, so information provided by the PUC system may be extended with research done on visual techniques for layouts and on automatic user interface generation.

## 2.1 Personal universal controller (PUC)

Nichols et al. describe in [4] [3] a system developed as an approach for improving the interfaces to complex appliances by introducing an intermediary graphical or speech interface. This system, called personal universal controller (PUC) automatically generates a user interface with which the user interacts as a remote control for any application.



```
<?xml version='1.0' encoding='UTF-8'?>
<spec name='Audiophase 5 CD''>
  <group>
    <state name='CDTrackState'>
      <type>
        <valueSpace>
          <integer/>
        </valueSpace>
      </type>
      <labels>
        <label>CD Track</label>
        <label>Track</label>
        <text-to-speech text='Track'
                        recording=''track.au''>
      </labels>
      <active-if>
        <equals state='Mode''>2</equals>
      </active-if>
    </state>
```

Figure 4: A diagrammatic overview of the PUC. A graphical or speech interface is introduced and communicates with the appliance by means of the specification language.

### 2.1.1 Architecture

The PUC architecture consists of *appliance adaptors*, a *specification language*, a *communication protocol* and *interface generators*. The appliances allow connection to the PUC by means of the appliance adaptor which represents a translation layer to its built-in protocol. The communication between PUC devices and appliances is enabled by a two-way communication protocol and a specification language that allows each appliance to describe its functions to an interface generator. The specification language constitutes the separation of the appliance to the type of interfaces it uses. The interface generator builds then the interface for the device that is going to control it, such as a graphical interface on a Handheld or a Pocket PC or a speech interface on a mobile phone. A diagram of this architecture is shown in figure 5.

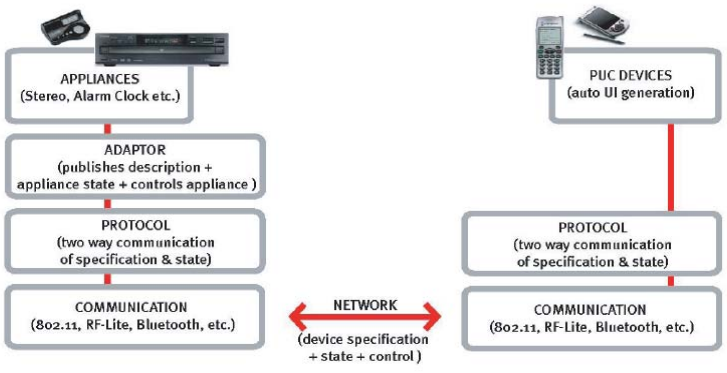

APPLIANCES
(Stereo, Alarm Clock etc.)
ADAPTOR
(publishes description + appliance state + controls appliance )
PROTOCOL
(two way communication of specification & state)
COMMUNICATION
(802.11, RF-Lite, Bluetooth, etc.)
NETWORK
(device specification + state + control )
PUC DEVICES
(auto UI generation)
PROTOCOL
(two way communication of specification & state)
COMMUNICATION
(802.11, RF-Lite, Bluetooth, etc.)

Figure 5: An architectural diagram of the PUC system showing one connection.

### 2.1.2 Specification Language

The PUC specification language is XML-based and represents all manipulable elements as *state variables* and *commands*. State variables are specified with one of seven generic types: boolean, integer, fixed point, floating point, enumerated, string or custom. The interface generator knows how to manipulate these types and can infer the functions coupled with them. The custom type allows specifying standard widget arrangements representing a familiar set of interface elements. Commands represent functions whose results cannot be described easily in the specification.

All the elements must also have information about how to label their interface components. As the *label information* depends on form factors and interface modalities, every element has a so-called label dictionary. This dictionary contains a set of labels as plain text, phonetic representations and text-to-speech.

To group similar elements close together, state variables and commands are specified as leaf nodes in an n-ary *group tree*. Each branching node is a group and each group may contain any number of state variables, commands and other groups.

Furthermore, the two-way communication feature allows it to use *dependency information*, which is information about what components are disabled depending on the values of other state variables. These are specified with three types of dependencies: equal-to, greater-than, and less-than, combined with the logical operations AND and OR. This dependency information can also be used to structure graphical interfaces or to interpret ambiguous phrases uttered to a speech interface.

#### 2.1.3 Graphical Interface Generation

The dependency information is used by the graphical interface generator to determine how to divide the screen into panels, and to assign branches of the group tree to each panel. If two variables are never available at the same time, they might be placed on separate panels. This *mutual exclusiveness* is found by checking each state variable that other commands and state variables depend upon. This dependency structure is then analysed to decide whether panels are created and how they are controlled.

Once the initial structure is defined, the generator traverses the group tree and uses a decision tree to select a component for each state variable and command. Recursively the components are then inserted in an interface tree which represents the panel structure of the generated interface and is used for translating abstract layout relationship to a concrete interface. This tree contains information on the relative position of the panels to each other and of the components of each panel to each other. These layout rules determine also where the label of each component should be placed.

The interface is made concrete by determining size and location of each panel and each component. Labels are assigned by picking the largest label that fits in the allocated space.

### 2.2 Visual Techniques for Traditional Layouts

Vanderdonckt and Gillo summarise in [6] visual techniques exported from the area of visual design that provide the designer a wide range of means for laying out interaction objects.

The following sections describe a selection of these visual techniques, which are interesting and applicable for our field of work.

### 2.2.1 Balance and Symmetry

*Balance* is a highly recommended technique that searches for equilibrium along a vertical or horizontal axis in the layout (figure 6). It places its components equally around a gravity centre located on this axis. It is justified by human perception and intense need for it in visual layout. The opposite of balance is instability and occurs when interaction objects are not distributed equally on each hand of the axis.

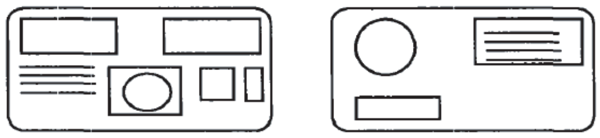


Figure 6: Balanced and unbalanced layouts

*Symmetry* consists of duplicating the visual image of interaction objects along a horizontal and/or vertical axis (figure 7). When a layout is symmetric, it is automatically balanced too. The opposite however is not justified.

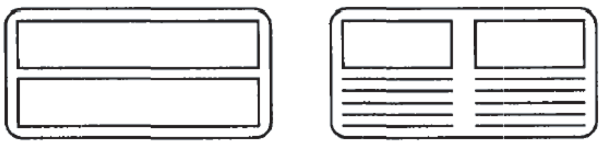


Figure 7: Horizontal and vertically symmetric layouts

### 2.2.2 Regularity

*Regularity* is a visual technique that is concerned with the horizontal and vertical uniformity and equilibrium. Interaction objects are placed uniformly according to some principle, method or convention that does not change in one particular layout (figure 8). Irregularity arises when no logical order of components is present and unexpected, unusual and unconforming layout grids are emphasised.

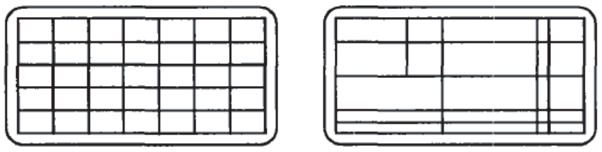


Figure 8: Regular and irregular layouts

### 2.2.3 Alignment

*Alignment* is guaranteed by reducing the number of vertical alignment points in a row and the number of horizontal alignement points in a column (figure 9). It is probably the most accessible and practical visual technique. A misaligned layout has a significantly high number of alignment points.

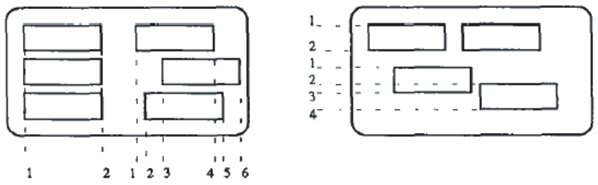


Figure 9: Vertical, horizontal alignments and misalignments

### 2.2.4 Proportion and Horizontality

*Proportion* strives for an aesthetically appealing ratio between the dimensions of an interaction object (figure 10). This can be either an aesthetic proved ratio as the *Golden Ratio* $1 : \frac{1+\sqrt{5}}{2}$ or a widely and conventionally prefered ratio (e.g. $1 : \sqrt{2}$, $1 : 2$, $1 : 1.29$, $1 : 1 : 5$, $1 : \frac{3}{4}$, $1 : 1.6$). Disproportion is implied when large differences appear between the two dimensions.

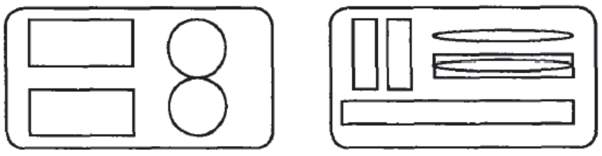


Figure 10: Proportioned and disproportioned layouts

*Horizontality* and analogous *verticality* arise when layouts with horizontal ratio (greater length than height) or vertical ratio (greater height than length) are predominant. Horizontality is highly preferred to verticality: long, narrow vertical dialog boxes are not aesthetic.

### 2.2.5 Grouping

*Grouping* is a visual technique that uses relative interaction between a set of components. The attraction of these components can be manipulated by changing the distance between these components or creating an optical similarity within that group (figure 11). With grouping, hidden connections can be set and a layout can be structured by providing an aesthetic appearance that helps remembering and accelerating a layout search. In a splitted structure, all interaction objects are placed without the ability to visually perceive an attraction or repulsion between them.

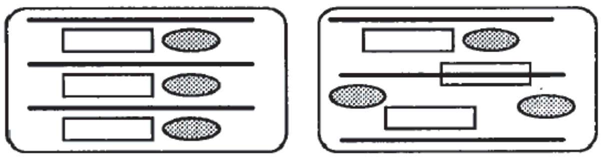


Figure 11: Grouped and splitted layouts

### 2.2.6 Sparing

*Sparing* suggests to keep the visual loading of a layout within reasonable boundaries (figure 12). Density, the opposite of sparing, takes no care about stacking interface components too tightly in the layout. It is usually measured by dividing the number of lighted pixels by the total number of available pixels.

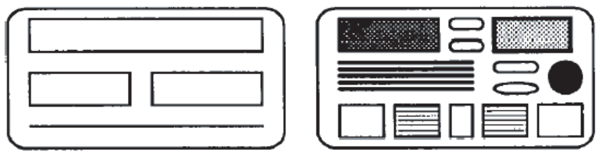


Figure 12: Uncluttered and cluttered layouts

### 2.2.7 Consistency

*Consistency* is a visual technique for developing a layout whose components are dominated by one sound, uniform, constant thematic. Consistency takes place not only in the ordering of interaction objects, but also in their small differences. Variation has no burdens for one or many themes and can be assumed by a set of widgets from which contents, shapes, colours, themes vary significantly.

## 2.3 Automatic User Interface Generation

Won Chul Kim and James D. Foley [2] have tried to develop a framework that provides high-level assistance for generating a presentation design. This tool, named DON, works in two stages, corresponding to the logical steps a designer takes.

The first step is accomplished by an *organisation manager* which uses a top-down design methodology to assist designers in organising the information, and selecting appropriate interface objects and their associated attributes. This manager embodies two main sets of rules: (1) *organisation rules* that determine how the layout should be organised, and (2) *selection rules* that select the appropriate interface object types and their attributes. To complete the presentation design, the *presentation manager* is used, which automates the actual layout of menus and dialog boxes. Therefore *layout rules* and designer-specified *layout preferences* are followed to create different design alternatives that are evaluated by means of the *evaluation metrics.*

### 2.3.1 Automatic dialog box layout

The layout process works with a tree of bounds that is generated automatically by the organisation rules. The strategy consists in organising the bounds recursively from the leaf nodes up to the roots of the dialog box tree, to systematically reduce the complex layout problem into many smaller problems.

Two bounds of a same subgroup node are then selected; their arrangement is determined by a shape and size analysis and form a new bound in the same pool. This process is repeated for each unknown group node until all the bounds in the pool are exhausted (figure 13). The designer may modify the rules for shape and size analysis to have additional fine control on the layout algorithm.

### 2.3.2 Customisation of user interface styles

DON provides an explicit method for specifying the layout constraints for each dialog box design instance. The framework makes it possible to set certain specifications global to all the dialog box instances, but also to overwrite specific constrains for exceptions. The designer has the possibility to control margins and spacing, as well as the location and orientation of 'OK' and 'Cancel' buttons.

### 2.3.3 The generate-and-evaluate strategy

The generate-and-evaluate method of design facilitates exploration by allowing a designer to view many automatically generated dialog box layouts that are evaluated by various evaluation metrics. The designer can then select the best generated layout structure by comparing the alternatives and use

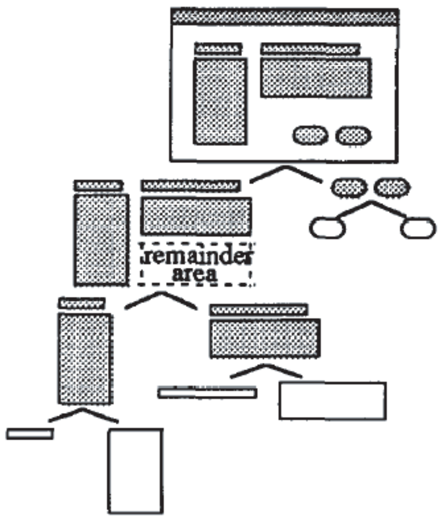


Figure 13: Illustration of the layout process, working with a tree structure of bounds.

an iterative strategy to finetune the detailed visual attributes. The evaluation criteria DON uses are based on calculation of balance, symmetry, dialog box size ratio, dialog box size range and negative space percentage.

# 3 Implementation aspects

## 3.1 Implementation details

The implementation of the framework able to generate automatically user interfaces, is based on the process suggested in the PUC system in section 2.1. The research however contains little information about techniques concerning the generation of the layout, so we have used the work of Vanderdonckt and Gillo [6], and Kim and Foley [2] to extend and adapt the structure of the resulting user interfaces for the PIP.

The following sections describe the architecture and implementation of our framework, developed for the environment of Studierstube.

### 3.1.1 High-level overview

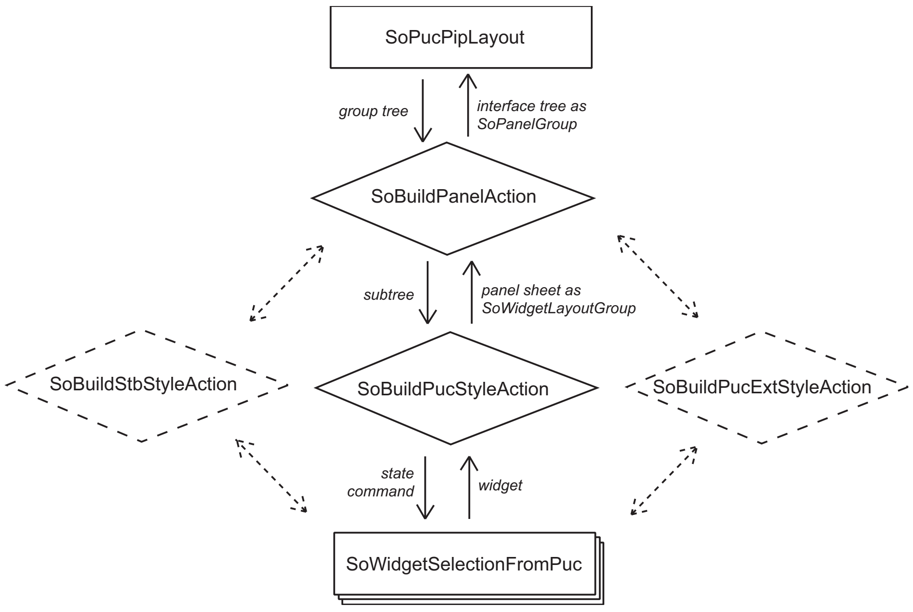


Figure 14: Diagram of the high-level structure of the software components

The main class controlling the user interface generation is the `SoPucPipLayout` class. It contains the specification of the appliance's functions and hints concerning the size and style of the layout to be generated. When created it starts the `SoBuildPanelAction` which analyses the base structure of the group tree and defines a panel structure for the interface if needed.

For each panel the `SoBuildPanelAction` now starts the action corresponding to the style chosen by the user. There are three different layout actions implemented: the SoBuildPucStyleAction, the SoBuildStbStyle-Action and the SoBuildPucExtStyleAction. These layout actions generate the whole layout for one panel sheet. They start the traversal at the group node corresponding to the root node of the content of the panel, determine for each element an interaction object and construct an interface tree consisting of SoWidgetLayoutGroup's and widgets. Based on the structure of this interface tree, size and location for each component is allocated and the corresponding values and transform nodes are added to this tree. After traversal this part of the interface tree is returned to the SoBuildPanelAction and the procedure is repeated for the remaining panel sheets.

When all the panels are constructed the SoBuildPanelAction returns the interface tree to the SoPucPipLayout class where the user interface is rendered and presented to the user.

The process may be adapted and improved by implementing new layout actions with specific behaviour. This case is handled in section 3.2.

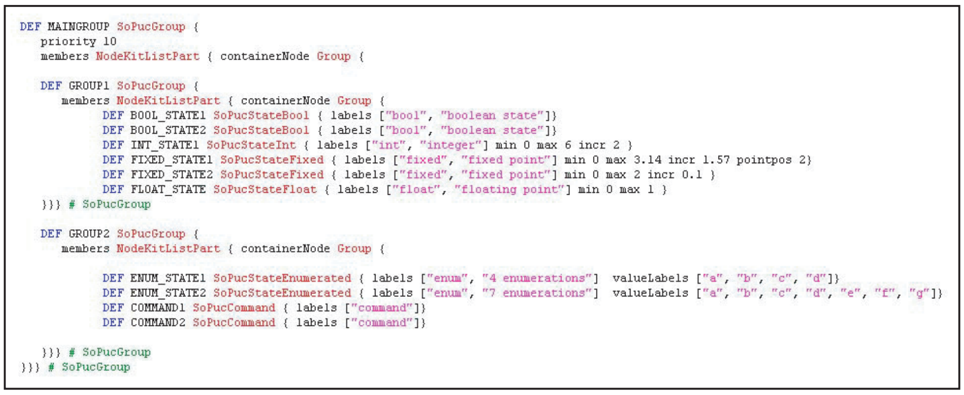
### 3.1.2 Application objects

As recommended in section 2.1.2 all appliance's functions are specified in form of states variables and commands. These are implemented in the following classes:

- the SoPucCommand represents a command
- the SoPucStateBool represents a boolean state
- the SoPucStateInt represents an integer state
- the SoPucStateFloat represents a floating point state
- the SoPucStateFixed represents a fixed point state
- the SoPucStateEnumerated represents an enumerated state
- the SoPucStateString represents a string state
- the SoPucGroup represents a group of elements

Each element contains a set of labels in *labels* where at least one label must be specified and a *priority* indicating the importance of the element. This importance can be used by the interface generator to make certain decisions. In contradiction to the SoPucCommand, all state variables own a *value* whose type depends on the state, and other type specific parameters.

Grouping is possible by means of the `SoPucGroup` which may contain any number of state variables, commands and other groups. Hierarchical use of the `SoPucGroup` structures all application objects into a group tree. Figure 15 shows an example of such a group tree.



```
DEF MAINGROUP SoPucGroup {
   priority 10
   members NodeKitListPart { containerNode Group {

   DEF GROUP1 SoPucGroup {
      members NodeKitListPart { containerNode Group {
            DEF BOOL_STATE1 SoPucStateBool { labels ["bool", "boolean state"]}
            DEF BOOL_STATE2 SoPucStateBool { labels ["bool", "boolean state"]}
            DEF INT_STATE1 SoPucStateInt { labels ["int", "integer"] min 0 max 6 incr 2 }
            DEF FIXED_STATE1 SoPucStateFixed { labels ["fixed", "fixed point"] min 0 max 3.14 incr 1.57 pointpos 2}
            DEF FIXED_STATE2 SoPucStateFixed { labels ["fixed", "fixed point"] min 0 max 2 incr 0.1 }
            DEF FLOAT_STATE SoPucStateFloat { labels ["float", "floating point"] min 0 max 1 }
   }}} # SoPucGroup

   DEF GROUP2 SoPucGroup {
      members NodeKitListPart { containerNode Group {

            DEF ENUM_STATE1 SoPucStateEnumerated { labels ["enum", "4 enumerations"]  valueLabels ["a", "b", "c", "d"]}
            DEF ENUM_STATE2 SoPucStateEnumerated { labels ["enum", "7 enumerations"]  valueLabels ["a", "b", "c", "d", "e", "f", "g"]}
            DEF COMMAND1 SoPucCommand { labels ["command"]}
            DEF COMMAND2 SoPucCommand { labels ["command"]}

   }}} # SoPucGroup
}}} # SoPucGroup
```

Figure 15: All appliance's functions are specified in a n-ary group tree with a state variable or command at each leaf.

### 3.1.3 Defining the initial structure

The first step of the generation process consists in analysing the base structure of the group tree. According to PUC system this should be done by determining mutual exclusiveness for the different states and commands and decide by means of this structure if a panel structure should be introduced. We reduce this algorithm by considering only the structure of the group tree independently of mutual exclusiveness. This simplification may provide unsatisfactory results in different cases, but we assume it acceptable for this work.

Figure 16 illustrates how our system defines this initial structure. Key component of the process is the `SoBuildPanelAction`, an action traversing only the PUC groups in the higher hierarchies. If the first child is a `SoPucStateEnumerated` and each of the remaining children a `SoPucGroup`, it creates a `SoPanelGroup` where each of the `SoPucGroup`'s forms one separate panel sheet, and the labels of these sheets are extracted from the different enumerations of the `SoPucStateEnumerated`. To allow interaction between the panel sheets, a `SoTextListBox` is constructed, which has as possible values the labels of the sheets. The content of the panel can then be switched by activating a different button of this list box.

In case where the above conditions are not provided, no `SoPucGroup`

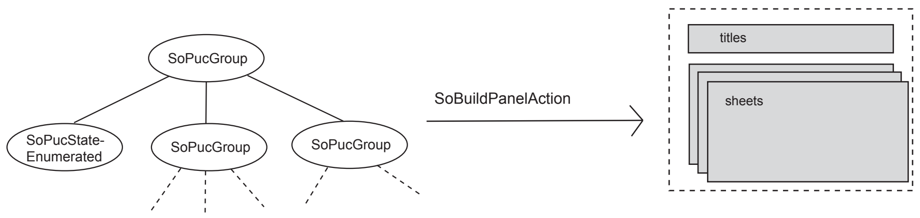


Figure 16: Diagram illustrating the generation of the initial structure of the user interface.

is created and the whole PIP-space is considered as one single panel sheet where all the components are placed on.

### 3.1.4 Apply layout rules

As soon as the initial structure is defined, the layout of the different panel sheets needs to be computed. Now decisions have to be taken concerning the number of columns or rows per page, the alignment of the widgets, the proportions of the components, ... In our system this task is fulfilled by a layout action, which commutes for each component its location and size. Figure 17 illustrates the generation of a simple layout.

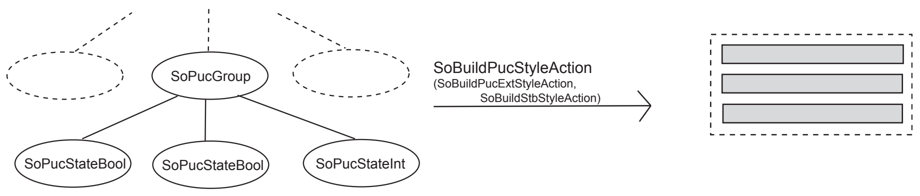


Figure 17: Diagram illustrating the generation of the interface structure of one panel sheet.

As the generation process a human designer usually follows, is very complicated and hard to imitate, we need an approximation of this process, which provides user interfaces of acceptable quality.

The PUC system places the components of one tree using a one-column layout with adjacent labels, and has some additional rules for special structures. This one-column layout is appropriate for handhelds and desktop computers where the size of the surface is of vertical ratio. The PIP however is of horizontal ratio and using a one-column layout would thus require to either limit the number of components or distort the proportions of the different components. For a high number of interaction objects this disproportion leads then to unsatisfactorily results.

A solution to this problem is to provide an additional set of generating methods using visual techniques adapted to a higher quantity of components. The current system supports three generating methods implemented in different layout actions:

1. The default layout action, `SoBuildPucStyleAction`, arranges the components in a one-column layout with adjacent labels as in the PUC system. It remains at the user's own responsibility to limit the number of components to avoid disproportion. Lower hierarchies of groups are ignored. This case corresponds to the method showed in figure 17.

2. The `SoBuildPucExtStyleAction` allows to generate an extended version of the PUC style. An additional parameter permits to specify the number of columns used for the layout. This style supports thus a higher capacity of components per sheet. To preserve balance, the widgets are placed individually in the column having the lowest level of widgets at that point. Lower hierarchies of groups are still not taken in account.

3. The `SoBuildStbStyleAction` is an approach to respect groupings in between the widgets of one panel sheet. The main structure consists of a number of rows that are filled up subsequently with widgets. The components of two different `SoPucGroup`'s are never placed in a same row. This implies that grouping is preserved, the layout may however not be balanced. The number of widgets placed in one row can be specified by the parameter *units*.

These three generation methods are illustrated in figure 18.

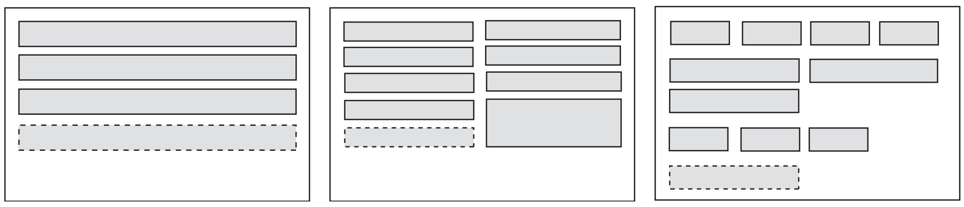


Figure 18: Illustration of the generation mode for the PUC style, PUCEXT style and STB style.

It is also at this place where the layout hints are taking effect. There are currently three layout hints that can be specified and are respected dependently on the chosen layout style:

- The *dimensions* (width, depth, height) of the desired user interface have to be indicated for each style.

- The number of *columns* can be specified for the PUCEXT style.
- The number of *units* per row can be indicated for the STB style. This parameter indicates how many widgets should be placed in one row: buttons correspond to one unit; sliders and list boxes to two units.

The layout actions determine the graphical layout with a *layout grid*. Such a grid consists of a set of parallel horizontal and vertical lines that divide the layout into regularly or irregularly sized rectangles, defining size and location of the interaction objects [6].

The implementation of this layout grid is done with the `SoWidgetLayoutGroup`, that places its elements into a given number of columns and rows with a specified relative size. For each of the elements of this group, space and location is allocated depending on its position on the grid. Further division is possible by hierarchical use of the `SoWidgetLayoutGroup`.

### 3.1.5 Transforming application data into interaction objects

In this step the generator chooses the kind of component and label to assign to each state variable and command. Therefore we use, as in the PUC system, a decision tree taking in account the type of the application object, the number of possible values and the style. Usually, a command is always represented by a push button, a boolean state by a toggle button, enumerated states by list boxes and states related to numbers (integer, fixed point, float) are represented by sliders or list boxes.

The choice of the label and its location depends only on the style: the PUC style always uses the longest label; PUCEXT and STB styles use the smallest one.

All these decisions are made in the different layout actions where for each state variable and command a static method from `SoWidgetSelectionFromPuc` is called which returnes the corresponding widget node.

Figure 19 illustrates this transformation into widgets.

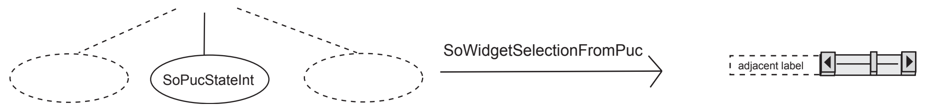


Figure 19: Diagram illustrating the generation of the different widgets.

## 3.2 Implementing a new layout action

Lets presume we want to implement a new layout action able to generate a user interface based on different rules. Hereafter we shall describe how this layout action can be implemented and made accessible.

### 3.2.1 Creating the action

The creation of our action requires some steps that are described hereafter.

**Selecting a name** The first step consists in selecting a name. To preserve consistency it should take the form of SoBuild*StyleAction. We presume that our action can be described by the word "new" and thus we call it `SoBuildNewStyleAction`.

Our class has to be inherited by `SoBuildPucStyleAction`, the default layout action. To allow external access, we add the class to the library `PUC_PIP_LAYOUT_API`

```
class PUC_PIP_LAYOUT_API SoBuildNewStyleAction :
                                 public SoBuildPucStyleAction}
```

As for each action we have to call the `SO_ACTION_HEADER()` macro for our action.

```
SO_ACTION_HEADER(SoBuildNewStyleAction);
```

**Initialising the action** Each action must be initialised before any instance of the class is created. This is usually done in the method `initClass()` where the macro `SO_ACTION_INIT_CLASS()` does the required work.

```
SO_ACTION_INIT_CLASS(SoBuildNewStyleAction, SoAction);
```

In addition, a static method for each supported node class has to be registered in this method. This includes default action behaviour for basic node classes and specific action behaviour for classes describing elements of the group tree.

```
SO_ACTION_ADD_METHOD(SoNode,                callDoAction);
SO_ACTION_ADD_METHOD(SoNodeKitListPart,     callDoAction);
SO_ACTION_ADD_METHOD(SoGroup,               callDoAction);

SO_ACTION_ADD_METHOD(SoPucStateBool,        stateBoolAction);
SO_ACTION_ADD_METHOD(SoPucStateFixed,      stateFixedAction);
SO_ACTION_ADD_METHOD(SoPucStateFloat,      stateFloatAction);
SO_ACTION_ADD_METHOD(SoPucStateInt,        stateIntAction);
SO_ACTION_ADD_METHOD(SoPucGroup,           pucGroupAction);
SO_ACTION_ADD_METHOD(SoPucCommand,         commandAction);
SO_ACTION_ADD_METHOD(SoPucStateEnumerated, stateEnumAction);
```

**Defining the constructor** The macro `SO_ACTION_CONSTRUCTOR()` does the basic work that has to be done in the constructor.

```
SO_ACTION_CONSTRUCTOR(SoBuildNewStyleAction);
```

**Traversal Behaviour** If you need to initialise the action each time it is applied, you should use the `beginTraversal()` method. Usually the base layout is build in this method, i.e. a `SoWidgetLayoutGroup` with a specified number of columns or rows is constructed. After the initialisations `traverse(node)` has to be called to guarantee the continuation of the traversal.

Operations that need to be performed after traversal should be implemented in `endTraversal()`.

```
void SoBuildPucStyleAction::beginTraversal(SoNode *node)
{
      //your initialisations

      traverse(node);
}
```

**Implementing static methods** Now we have to implement all methods that describe the specific behaviour of our action during the traversal. For basic node classes, all we need to do is calling `doAction()` where its "typical" action behaviour is implemented.

```
void SoBuildNewStyleAction::
              callDoAction(SoAction *action, SoNode *node)
{
      node->doAction(action);
}
```

The specific layout behaviour for all node classes describing elements of the group tree has to be implemented in the corresponding static methods registered in `initClass()`. You need to specify here how each state variable and command is to be transformed into an interaction object, where its label should be placed and of what length it should be. Additionally, you may want to do some checks to decide in what row or column the current state variable or command has be placed.

**Additional requirements** In addition to the standard requirements of the implementation of an action class, there are some important requirements for this kind of layout action.

If your action needs some parameters to be set, implement a public method making this work. This method will then be called in `SoBuildPanelAction` before the action will be applied.

```
void SoBuildNewStyleAction::setParameter(int parameters_)
{
      parameters = parameters_;
}
```

At the end of traversal a pointer to the resulting `SoWidgetLayoutGroup` needs to be returned to the `SoBuildPanelAction`. This is done by means of `getLayoutGroup()` where `mainLayoutGroup` is returned. You can use this variable to store your layout information, or you can overwrite the `getLayoutGroup()` method.

### 3.2.2 Making the action accessible

To make the action accessible during the generation process you need to perform some changes in `SoBuildPanelAction` and `SoPucPipLayout`.

#### SoBuildPanelAction

1. Include the action in the header file:
   `#include <SoBuildNewStyleAction.h>`
2. Initialise the action in the constructor:
   `SoBuildNewStyleAction::initClass();`
3. Add an enumeration for the action in `style`:
   `enum style PUC = 0,STB = 1,PUCEXT = 2, NEW = 3 ;`
4. Add an if-clause in `applyLayoutAction()` where in case of `actionStyle == NEW` the parameters of your action are set, your action is applied and the resulting `SoWidgetLayoutGroup` is returned.

5. In case there are parameters to be set, add a public method to set these parameters from `SoPucPipLayout`:
   `void setParameters(int parameters);`

**SoPucPipLayout**

1. If parameters for your action need to be set, add a field to store the parameter.
   `SoSFInt32 parameters;` and `SO_NODE_ADD_FIELD(parameters,(2));`

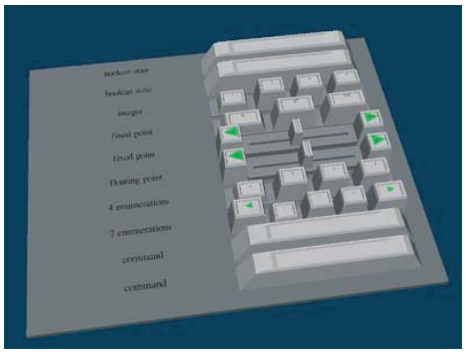
2. Add the same enumeration as above for the action in `style` and add the macro for the enumeration of the field:
   `SO_NODE_DEFINE_ENUM_VALUE(Style, NEW);`

3. Call the method setting your parameters in `doLayout()`:
   `panelAction.setParameters(parameters.getValue());`

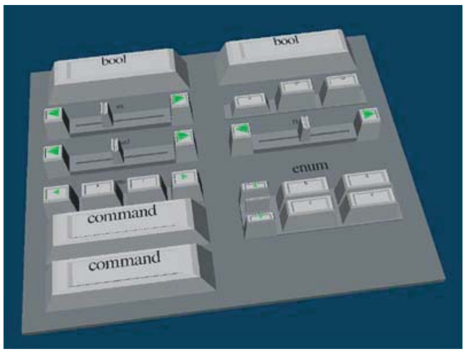
More information on how to create new actions can be found in [7].

### 3.2.3 Results

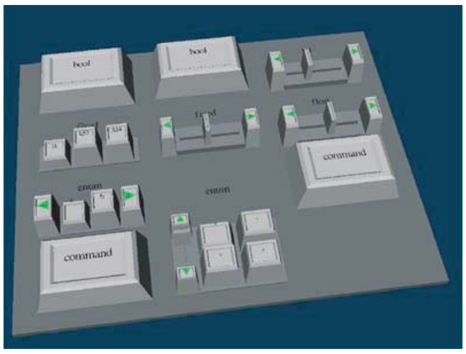
Figure 19 shows the generation of the group tree of figure 15 with different styles and layout hints.



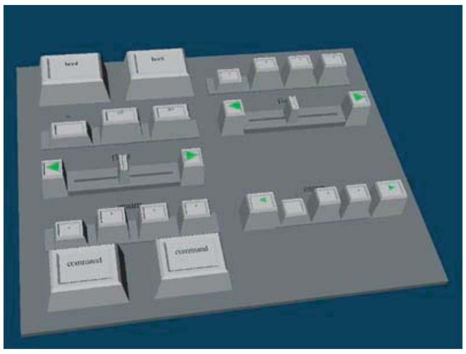
```
style PUC
```



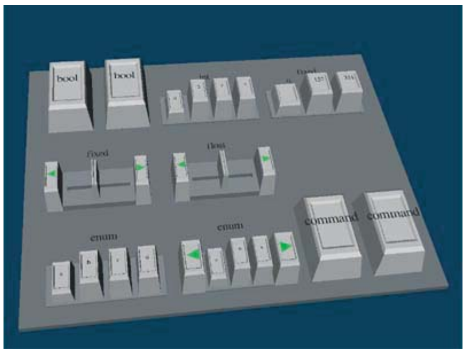
```
style PUCEXT columns 2
```



```
style PUCEXT columns 3
```



```
style STB units 4
```



```
style STB units 6
```

Figure 20: The group tree of figure 15 has been generated with the three different styles and different layout hints.

# 4 Discussion

We have now presented the results of the project aiming to develop a system that is able to automatically create graphical user interfaces for an application in Studierstube. According to Nichols et al. [4], such a system should analyse a given specification of the application's functions in form of a group tree with state variables and commands at its leaves and decide by means of this information how to construct a user interface for that application. Kim and Foley [2] describe how this generation process can imitate the usual steps a human designer follows, and Vanderdonckt and Gillo [6] show what visual techniques allow computing a user interface of good quality.

This project is justified by the fact that the generation of GUI's in Studierstube was very time-consuming, inflexible and only weakly supported by frameworks. Furthermore, by using as application objects the PUC states and PUC commands, the constructed GUI's are compatible with the PUC system and may be controlled by additional devices.

To develop this system for Studierstube we have imitated the process used by Nichols et al. [4] in the PUC system. This imitation however led us immediately to some problems: Nichols et al. have developed their system for handhelds or pocket PC's where the construction of a GUI is realised by a different manner and with different requisites. When building a user interface for these devices the designer has the possibility to use a whole set of different interaction objects and to be supported by frameworks facilitating this task. Studierstube however owns only a small set of interaction objects without features like scroll bars or panels and is also not able to define user interfaces by means of a layout grid. These circumstances forced us to firstly develop the necessary helper classes to guarantee a similar environment as the one described in the PUC system.

Moreover, as interaction with a pen and a panel is not as precise as with 2D pointing devices, we have decided not to implement a scroll bar as on conventional dialog boxes. This implied that the usable space is limited in dimension and proportion and thus we need a more performant interface generator following visual techniques to allow an optimal placing of the different components. The realisation of this interface generator turned out to be very complex. A human designer building a GUI uses its experience and intuition to decide the base structure of the layout, qualities that would afford an artificial intelligence. The implementation of such a generator would however exceed the field of work for this research, so we decided to charge the user with giving some layout hints determining the structure of the user interface. For each generation he needs to choose the style that is best adapted to the application objects. Each style works according to certain visual techniques what implies that there are always some techniques that haven't been followed.

With these adaptations and limitations we have been able to implement the considered system for Studierstube. In comparison with the PUC system there are however some aspects that haven't been satisfied. One point is that dependency information isn't supported. This information is important for a lot of decisions in the interface tree and allows reaching a better quality of the user interfaces. For example a panel group should only be created when the different groups of components are mutually exclusive. The dependency information is also useful to determine the importance of singular components, so that important widgets may be placed on locations where they are best visible.

Furthermore, due to the use of layout hints one requirement of the PUC system is not fulfilled: the specification still includes specific layout information [3]. To get best results, it is necessary to understand how the system reacts on different groupings and then decide how to choose the structure of the group tree and the layout hints. To solve this problem we suggest working on the interface generator and develop an AUTO modus using some metrics to extract the best style and the corresponding best parameters.

To find out what characteristics and visual techniques are most important to allow an easy use of the PIP, a study could be done, comparing different GUI's built according different visual techniques. These GUI's could then be tested on a certain number of test persons in the environment of Studierstube in order to evaluate their quality.

In general, the project has met its promises. GUI's may be produced in much shorter time and may also easily be modified and extended. Moreover, it is possible to remotely control the PIP by using the PUC. The helper classes describing a panel structure or a layout grid turned out to be very useful. These can be used to manually generate GUI's without the explicit specification of transformations and can therefore be used when the user would like to have more control on look and feel of the layout. In comparison to the automatic generation, the specification file is elongated and includes all layout information, but it is still easy to add or change its components.

If the user doesn't have concrete ideas about how the generated UI should look like, the automatic generation is surely the best solution. If he however would like to control the positioning of the singular components, use of the helper classes is recommended. In both of the cases the project has provided means exceeding those available so far.

# 5 Conclusions

This research aimed to facilitate the generation of GUI's in Studierstube by using state variables and commands defined for use with the PUC and develop a mechanism able to automatically generate a GUI out of them. This mechanism should be closely related to the PUC system described by Nichols et al. [3] [4].

This goal has been satisfied and it is now possible to generate user interfaces in a much shorter time and also to modify or extend them easily. In comparison to the requirements made by the PUC, there have been some limitations: the system doesn't support dependency structure and its specification language still includes specific layout information. Additionally to the automatic generation of user interfaces, means have been provided to facilitate the manual generation of user interfaces.

It is a project of high usability that offers lots of possibilities to be extended and improved.

## References

[1] Dennis J. M. J. de Baar, James D. Foley, Kevin E. Mullet, and Charles A.van der Mast. Coupling application design and user interface design. Technical Report DUT-TWI-92-03, Delft, The Netherlands, 1992.

[2] W. C. Kim and J. D. Foley. Providing high-level control and expert assistance in the user interface presentation design. In *Proc. of INTERCHI-93*, pages 430–437, Amsterdam, The Netherlands, 1993.

[3] Jeffrey Nichols and Brad A. Myers. Studying the use of handhelds to control everyday appliances.

[4] Jeffrey Nichols, Brad A. Myers, Michael Higgins, Joseph Hughes, Thomas K. Harris, Roni Rosenfeld, and Mathilde Pignol. Generating remote control interfaces for complex appliances. In *Proceedings of the 15th annual ACM symposium on User interface software and technology*, pages 161–170. ACM Press, 2002.

[5] Dieter Schmalstieg, Anton Fuhrmann, Gerd Hesina, Zsolt Szalavari, L. Miguel Encarnacao, Michael Gervautz, and Werner Purgathofer. The studierstube augmented reality project. *Presence: Teleoper. Virtual Environ.*, 11(1):33–54, 2002.

[6] Jean Vanderdonckt and Xavier Gillo. Visual techniques for traditional and multimedia layouts. In *Proceedings of the workshop on Advanced visual interfaces*, pages 95–104. ACM Press, 1994.

[7] Josie Wernecke. *The Inventor Toolmaker: Extending Open Inventor, Release 2*. Addison-Wesley Longman Publishing Co., Inc., 1994.

## 6 Appendix

Online documentation of the relevant classes.

# Generating User Interfaces in Studierstube

## All widgets

This section describes the fields which are inherited from **SoLayoutKit** or **SoBehaviorKit** and proper to all widgets.

### width, depth, height

Determine the size of the widget.

### label, labelplacing

The field 'label' allows you to indicate textual or graphical information describing the widget. It contains one **SoLabelKit** that, inherited from SoShapeKit, can have any shape. A textual label can be entered by means of the multiple field 'text'. The different entries of this multiple field will then be separated by a line break.
The field labelplacing may take as possible values NONE, TOP, LEFT, RIGHT, BOTTOM or ONWIDGET and indicates where the label should be placed.
See: **SoLabelKit**

### enable

Indicates if the widget should be enabled (TRUE) or not.

## The SoPushButton

The **SoPushButton** implements a "normal" button. When pressed, the button goes down, a command is executed and the button releases immediately.

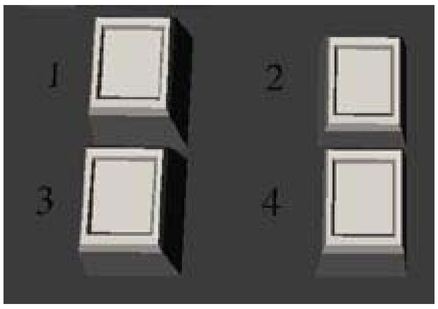
### Example

```
SoPushButton {}
```

## The SoToggleButton

The **SoToggleButton** implements a 2D check box widget. It can have two values: pushed or not pushed. When the button is pushed, it goes down and stays at that position until you release it by pressing it a second time.

### Example



```
SoToggleButton { label SoLabelKit { text "1" } labelPlacing LEFT}
SoToggleButton { label SoLabelKit { text "2" } labelPlacing LEFT}
SoToggleButton { label SoLabelKit { text "3" } labelPlacing LEFT}
SoToggleButton { label SoLabelKit { text "4" } labelPlacing LEFT}
```

## The SoSimpleSlider

The **SoSimpleSlider** class implements a typical 2D slider widget. It allows setting a range of values and selecting from that by moving the slider knob.

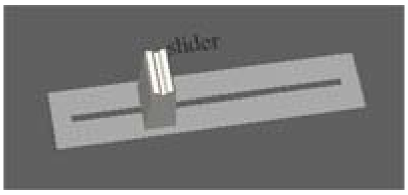
### minValue, maxValue

The minimum and the maximum of the range of values are specified by these fields. The minimum value is reached when the slider knob is moved completely to the left, the maximum at the further right.
Note: It is also possible to implement a decreasing slider by specifying a minValue bigger than the maxValue.

### value, alpha

The value of the slider is always coherent with the position of the slider knob. If one of both changes, the other is adapted too.
The alpha represents a normalized version of the value [0;1]. It also changes with the value and the slider knob, and the slider may be controlled by it.

### Example



```
SoSimpleSlider {
        width 10 depth 4 height 3
         label SoLabelKit { text "slider" }
        labelPlacing TOP
}
```

# The SoIncrementalSlider

The **SoIncrementalSlider** class implements a typical 2D slider widget. It allows setting a range of values and selecting from that by moving the slider knob or pressing the buttons on the left and the right.

### minValue, maxValue, value, alpha, widthDragger, scaleDraggerPath

Same behaviour as for the **SoSimpleSlider** (see above).

### increment, cropValueToIncrement

The increment specifies the resolution of the slider, i.e. the interval between two possible values.

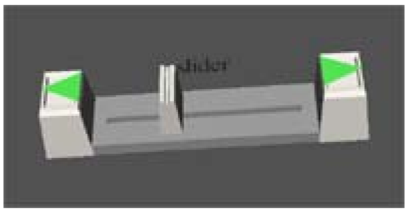
### hideSlider

If TRUE the slider and the moving knob will not be rendered. Interaction is now only possible with the increment and decrement buttons.

### buttonSpace

Specifies how much space of the slider should be used by one of the increment and decrement buttons. The portion of space is given in normalized percentage [0;1].

### Example



```
SoIncrementalSlider {
        width 10 depth 4 height 3
         increment 1.5
        label SoLabelKit { text "slider" }
        labelPlacing TOP
}
```

# The SoLabelListBox

This class provides a simple list box interface with a number of buttons showing different items and a **SoIncrementalSlider** allowing navigating through the possible items. When there are more items than buttons, the slider can be used to navigate up and down through the list. When all the items can be displayed on one page, the slider is disabled.

### multipleSelections, noneSelectionAllowed

These two fields specify how many items can be simultaneously selected. If both are FALSE (default), only one item can be selected, if multipleSelections is TRUE multiple items can be selected and if noneSelectionAllowed is TRUE, no selection may be made.

Default values: multipleSelections = FALSE, noneSelectionAllowed = FALSE

## numOfRows, numOfCols

These fields determine the structure of the list box. The visible items are placed in numOfRows rows and numOfCols columns, so these fields determine additionally the maximum number of buttons that will be visible. When the navigation slider is used, this structure remains unmodified; only the connections button-to-item will be changed.

Default values: numOfRows = 5, numOfCols = 1

## spacingWidth, spacingDepth, startEndSpacingDepth, startEndSpacingWidth

With these fields the spaces between all the components of the list box can be specified. The fields spacingWidth and spacingDepth indicate the space between two buttons, the field startEndSpacingDepth specifies the size of the top and bottom margin, and startEndSpacingWidth the size of the left and right margin. All the values are given in percent [0;1].

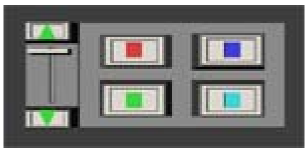
Default values: spacingWidth = spacingDepth = startEndSpacingDepth = startEndSpacingWidth = 0.05 (5 )

## navigationSize, navigationButtonToSliderRatio, navigationPlacing

These fields specify location, size and structure of the navigation slider. The ratio of the navigation buttons to the whole slider is given by the field navigationButtonToSliderRatio, the ratio of the navigation slider to the whole list box is given by the field navigationSize. Both values are given in percent [0;1]. The position of the navigation slider can be specified by the field navigationPlacing. Possible values for this enumeration field are NONE, TOP, LEFT, RIGHT, BOTTOM, INLINE_ROW and INLINE_COL. For the INLINE values no slider will be displayed. Instead the first and last visible buttons will be used as buttons to navigate through the list.

Default values: navigationSize = 0.2, navigationButtonToSliderRatio = 0.2, navigationPlacing = LEFT

## Example



```
SoLabelListBox {
    width 15 depth  6 height 3
    numOfRows 2 numOfCols 2
    spacingWidth 0.1 spacingDepth 0.1
    startEndSpacingWidth 0.1 startEndSpacingDepth 0.1
    navigationPlacing LEFT
    navigationButtonToSliderRatio 0.2

    labels NodeKitListPart { containerNode Group {
    SoLabelKit {
      appearance SoAppearanceKit {  material SoMaterial {diffuseColor .8 .2
            .2  } }
      shape SoCube {width .4 height .4 depth .4 }}

    SoLabelKit {
      appearance SoAppearanceKit {  material SoMaterial {diffuseColor .2 .8
            .2  } }
      shape SoCube {width .4 height .4 depth .4 }}

    SoLabelKit {
      appearance SoAppearanceKit {  material SoMaterial {diffuseColor .2 .2
            .8  } }
      shape SoCube {width .4 height .4 depth .4 }}

    SoLabelKit {
      appearance SoAppearanceKit {  material SoMaterial {diffuseColor .2 .8
            .8  } }
      shape SoCube {width .4 height .4 depth .4 }}

    SoLabelKit {
      appearance SoAppearanceKit {  material SoMaterial {diffuseColor .8 .2
            .2  } }
      shape SoSphere { radius .2 }}

}}} #SoLabelListBox
```

## The SoTextListBox

This class implements a list box with similar appearance and behaviour as the **SoLabelListBox**. The only difference lies in the type of the labels. In the **SoTextListBox**, labels representing the different items can only be text strings, and are specified in the field 'values'.

### Field inherited from SoLabelListBox

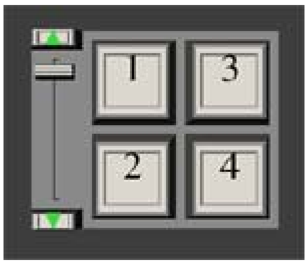
The fields multipleSelections, noneSelectionAllowed, numOfRows, numOfCols, spacingWidth, spacingDepth, startEndSpacingDepth, startEndSpacingWidth, navigationSize, navigationButtonToSliderRatio and navigationPlacing are inherited from **SoLabelListBox** and have the same characteristics as those described above.

### values

This multiple field encloses text strings for the different items. These strings are taken as textual labels for the buttons of the list box.

Default value: values = ""

### Example



```
SoTextListBox {
      width 10 depth 8 height 2
      numOfRows 2 numOfCols 2
      spacingDepth .05
      spacingWidth .05
      navigationSize .2
      multipleSelections TRUE

      navigationButtonToSliderRatio 0.1

      values ["1" "2" "3" "4" "5"]

    } #SoTextListBox
```

## The SoWidgetLayoutGroup

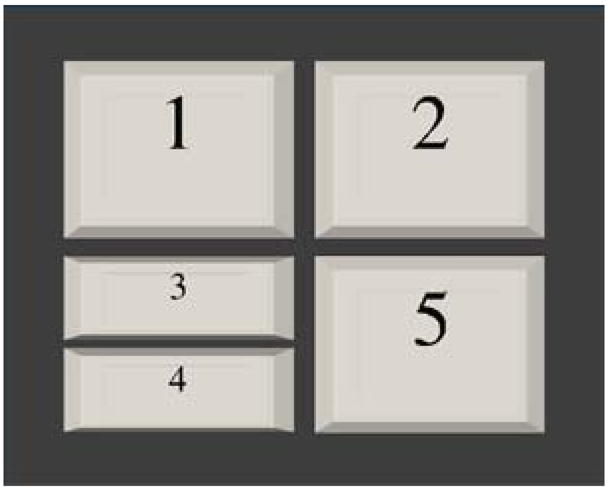
The **SoWidgetLayoutGroup** allows you to build user interfaces without explicitly specify the transformations and size of each widget. Therefore all widgets are structured onto a grid with a specified number of columns and rows. This grid can then be hierarchically subdivided furthermore in a more detailed grid.

To build the layout for your purposes you need to structure the widgets into SoWidgetLayoutGroup's. A **SoWidgetLayoutGroup** can contain widgets and SoWidgetLayoutsGroup's which can themselves contain other widgets and SoWidgetLayoutGroup's. For each **SoWidgetLayoutGroup** you may specify the number of rows and columns, their sizes, the spacings between widgets, and all the widgets and groups being element of this group. These elements are then placed on the specified grid.

### elements

In this part the different elements of this **SoWidgetLayoutGroup** are defined. Everything what is inherited from **SoLayoutKit** may be added as element.

**Example:**



```
SoWidgetLayoutGroup {
  width 22 depth 17 height 0.5
  numOfCols 2
  numOfRows 2

    elements NodeKitListPart { containerNode Group {

      SoPushButton { label SoLabelKit { text "1" } }
      SoPushButton { label SoLabelKit { text "2" } }

      SoWidgetLayoutGroup {
         numOfCols 1
         numOfRows 2

        elements NodeKitListPart { containerNode Group {
                SoPushButton { label SoLabelKit { text "3" } }
                SoPushButton { label SoLabelKit { text "4" } }

        }}} #SoWidgetLayoutGroup

       SoPushButton { label SoLabelKit { text "5" } }


   }}} #SoWidgetLayoutGroup
```

## width, depth, height

Fields inherited from **SoLayoutKit**. To start the building process, you need to set these fields of the first **SoWidgetLayoutGroup**, representing the space of the pip sheet which should be filled up by the UI. Width and depth will then be calculated individually for each child. The height will remain constantly over all sub hierarchies

Note: It isn't possible to change the sizes of the widgets in lower hierarchies by manually setting their width and depth. These ones will be overwritten. You should change the spacingWidth and spacingDepth instead.

## numOfCols, numOfRows

By setting these values you will be able to determine the structure of the layout. Default value is for both of the fields -1. There are 3 possible cases:

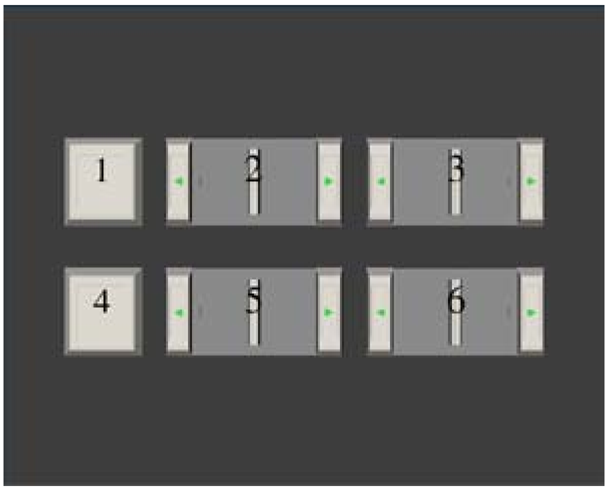
- just one of the values is set: the other one is calculated so that all the children will find a place in the grid
- both of the values are set: if there aren't enough places in the grid to include all children, the number of rows is increased
- none of the values is set: the number of rows is determined by taking the square root of the number of children to achieve an approximately regular distribution of the widgets.

You will obtain best results by setting just one of both values.

## sizeOfCols, sizeOfRows

With these fields you can vary the width of the rows and columns. The fields take as input a multiple field with a number of integers which indicate the size of each row and column. If the array is longer than the respective number of rows or columns, it will just be truncated; if it is shorter it will be filled up with 1's.

**Example:**
In this example distribution sizeOfCols [1, 2, 2] has been used. The second and third columns appear twice as big as the first.

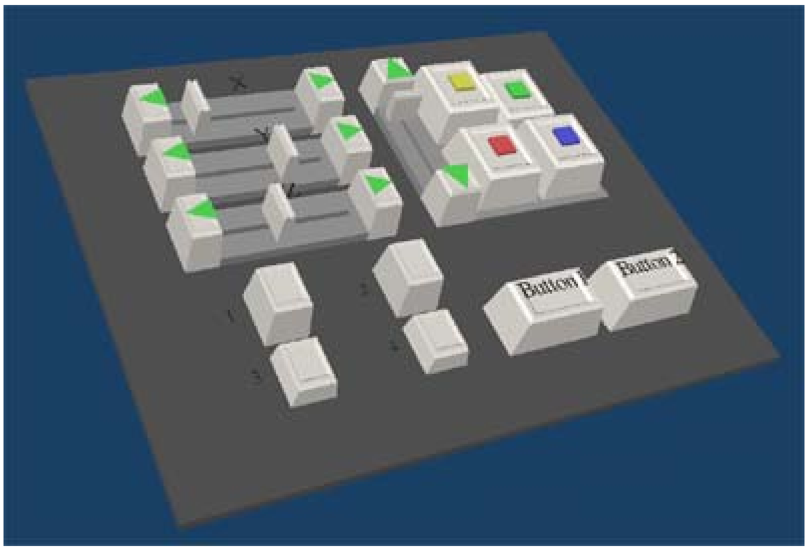


```
SoWidgetLayoutGroup {
  width 22 depth 10 height 0.5
  numOfCols 3
  numOfRows -1
  sizeOfCols [1, 2, 2]
  spacingDepth 0.2

    elements NodeKitListPart { containerNode Group {

      SoToggleButton { label SoLabelKit { text "1" } }
      SoIncrementalSlider { label SoLabelKit { text "2" } }
      SoIncrementalSlider { label SoLabelKit { text "3" } }
      SoToggleButton { label SoLabelKit { text "4" } }
      SoIncrementalSlider { label SoLabelKit { text "5" } }
      SoIncrementalSlider { label SoLabelKit { text "6" } }



   }}} #SoWidgetLayoutGroup
```

## spacingWidth, spacingDepth

These fields specify how much space should be left between each row and column. Given as a normalised number representing a percentage of the size of an average row or column.
Default value: 0.20 (20 %)

## Example



```
SoWidgetLayoutGroup {
  width 21 depth 16 height 2
  numOfCols 1 numOfRows 2
  sizeOfRows [3, 2]
  spacingDepth .2

  elements NodeKitListPart { containerNode Group {

    SoWidgetLayoutGroup {
      numOfCols 2 numOfRows -1
      spacingWidth .2

      elements NodeKitListPart { containerNode Group {

        SoWidgetLayoutGroup {
          numOfCols 1
          numOfRows -1

          elements NodeKitListPart { containerNode Group {

            SoIncrementalSlider { label SoLabelKit { text "X" } }
            SoIncrementalSlider { label SoLabelKit { text "Y" } }
            SoIncrementalSlider { label SoLabelKit { text "Z" } }

        }}} #SoWidgetLayoutGroup

        SoLabelListBox {
          numOfRows 2 numOfCols 2

          labels NodeKitListPart { containerNode Group {
            SoLabelKit {
                   appearance SoAppearanceKit {  material SoMaterial
                        {diffuseColor .8 .8 .2  } }
                    shape SoCube {width .5 height .5 depth .1 }}
            SoLabelKit {
                   appearance SoAppearanceKit {  material SoMaterial
                        {diffuseColor .8 .2 .2  } }
                    shape SoCube {width .5 height .5 depth .1 }}
            SoLabelKit {
                   appearance SoAppearanceKit {  material SoMaterial
                        {diffuseColor .2 .8 .2  } }
```

```
                shape SoCube {width .5 height .5 depth .1 }}
        SoLabelKit {
               appearance SoAppearanceKit {  material SoMaterial
                    {diffuseColor .2 .2 .8  } }
                shape SoCube {width .5 height .5 depth .1 }}
        SoLabelKit {
               appearance SoAppearanceKit {  material SoMaterial
                    {diffuseColor .2 .8 .8  } }
                shape SoCube {width .5 height .5 depth .1 }}
        SoLabelKit {
               appearance SoAppearanceKit {  material SoMaterial
                    {diffuseColor .8 .2 .8  } }
                shape SoCube {width .5 height .5 depth .1 }}

      }}} #SoLabelListBox

    }}} #SoWidgetLayoutGroup

  SoWidgetLayoutGroup {
    numOfCols 3
    numOfRows -1
    sizeOfCols [1 1 2]
    spacingWidth .25

    elements NodeKitListPart { containerNode Group {

      SoToggleButton { label SoLabelKit { text "1" } labelPlacing
         LEFT}
      SoToggleButton { label SoLabelKit { text "2" } labelPlacing
         LEFT}
      SoWidgetLayoutGroup {}

      SoToggleButton { label SoLabelKit { text "3" } labelPlacing
         LEFT}
      SoToggleButton { label SoLabelKit { text "4" } labelPlacing
         LEFT}

      SoWidgetLayoutGroup {
        numOfCols 2
        numOfRows -1

        elements NodeKitListPart { containerNode Group {
          SoPushButton { label SoLabelKit { text "Button 1" } }
          SoPushButton { label SoLabelKit { text "Button 2" } }
      }}} #SoWidgetLayoutGroup

  }}} #SoWidgetLayoutGroup

}}} #SoWidgetLayoutGroup
```

## The SoPanelGroup

The **SoPanelGroup** represents a panel structure, which consists of a **SoLabelListBox** specified in 'titles' and a certain number of SoWidgetLayoutGroup's specfied in 'sheets'. The list box is placed at the top or at the right side of the pip and the remaining space is reserved for the different sheets. Each button of the list box is coupled with a panel sheet corresponding to the order in which they have been defined.

To define a panel structure, only the **SoPanelGroup** is needed where the 'titles' and 'sheets' have to be specified. The **SoPanel** is an intermediary class needed to compute the real geometry of this structure. The sheets of the **SoPanelGroup** are then transferred to the **SoPanel**, which adds them as children of a SoSwitch node.

## titles

This part must contain a **SoLabelListBox** or **SoTextListBox** where each of the items corresponds to the designation of one of the panel sheets. Each button is then coupled to one of the panel sheets and you can change the current visible sheet by pressing one of the buttons. The navigator of the **SoLabelListBox** will be placed INLINE_COL or INLINE_ROW.

## sheets

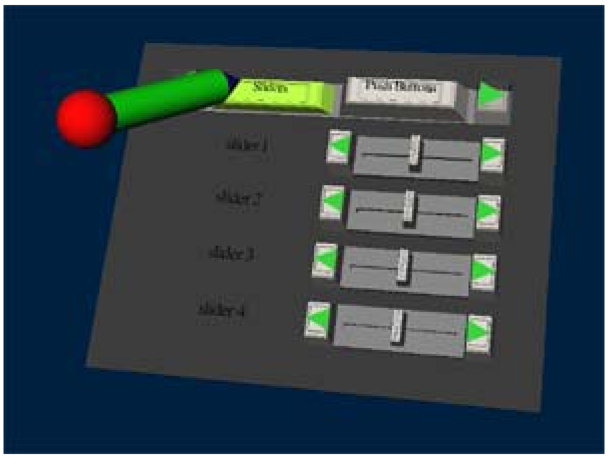
This list part contains the different panel sheets defined as **SoLayoutKit** or inheritances of it. Usually these are SoWidgetLayoutGroup's, but you can also nest multiple SoPanelGroup's or define a single widget as panel sheet.

## navigationPlacing, titlesToSheetsRatio, numOfButtons

These fields determine the structure and layout of the panel group. The field navigationPlacing specifies where the navigation list box should be placed. Possible values are LEFT and TOP. The field titlesToSheetsRatio defines the percentage of space that should be assigned to the navigation list box, and numOfButtons specifies how many buttons should be displayed.

Default values: navigationPlacing = TOP, titlesToSheetsRatio = 0.20, numOfButtons = 3

## Example



```
SoPanelGroup {
    width 22 depth 17 height 2
    navigationPlacing TOP
    numOfButtons 2
    titlesToSheetsRatio 0.20

    titles SoTextListBox {
                     values ["Sliders" "Push Buttons"]
                     }
    sheets NodeKitListPart { containerNode Group {
               SoWidgetLayoutGroup {
               numOfCols 1 numOfRows -1
               elements NodeKitListPart { containerNode Group {
                  SoIncrementalSlider { label SoLabelKit { text "slider 1"
                      } labelPlacing LEFT}
```

```
                SoIncrementalSlider { label SoLabelKit { text "slider 2"
                     } labelPlacing LEFT}
                SoIncrementalSlider { label SoLabelKit { text "slider 3"
                     } labelPlacing LEFT}
                SoIncrementalSlider { label SoLabelKit { text "slider 4"
                     } labelPlacing LEFT}

               }}} #SoWidgetLayoutGroup

               SoWidgetLayoutGroup {
               numOfCols 1 numOfRows -1
                 elements NodeKitListPart { containerNode Group {
                       SoPushButton { label SoLabelKit { text "1" }
                          labelPlacing LEFT}
                       SoPushButton { label SoLabelKit { text "2" }
                          labelPlacing LEFT}
                       SoPushButton { label SoLabelKit { text "3" }
                          labelPlacing LEFT}
                       SoPushButton { label SoLabelKit { text "4" }
                          labelPlacing LEFT}

               }}} #SoWidgetLayoutGroup

}}} #SoPanelGroup
```

## Automatic generation of GUI’s: the SoPucPipLayout

The **SoPucPipLayout** is able to automatically generate GUI's for appliances that are able to control them. All appliances functions are specified in "pucAppliances" where they should be hierarchically grouped to form a "group tree". Additionally some layout hints concerning size and style may be indicated in the corresponding fields. The generation of the user interface is then performed in three steps:

- In the first step the base structure of the group tree is analysed: if the first child of the first group tree is of type **SoPucStateEnumerated** and all the remaining ones of type **SoPucGroup**, then a **SoPanelGroup** is created where each of the SoPucGroup's corresponds to one separate panel sheet, and the titles of these sheets are extracted from the different enumerations of the **SoPucStateEnumerated**.
- In the second step the layout of each sheet is generated by following some layout rules which differ for each style.
- In the third step each state and command is transformed into a widget.

### pucAppliances

In this part all appliances functions need to be specified with the state variables and commands: **SoPucCommand**, **SoPucStateBool**, **SoPucStateInt**, **SoPucStateFixed**, **SoPucStateFloat**, **SoPucStateEnumerated** and **SoPucStateString**. The **SoPucGroup** allows you to hierarchically structure these elements into a group tree. This group tree has to be specified as part in a **SoPucDevice** belonging to a **SoPucServer**.
Example:
```
DEF LAYOUT_GROUP SoPucPipLayout {
     width 23 depth 18 height 2.0

     pucAppliances SoPucServer {
      serverName "Studierstube Demo"
      devices NodeKitListPart { containerNode Group {
        DEF RADIUSDEVICE SoPucDevice {
          deviceName "example"
          groups NodeKitListPart { containerNode Group {

             SoPucGroup { #root of the group tree
                ...

             }#SoPucGroup
         }}}#SoPucDevice
       }}}#SoPucServer
}#SoPucPipLayout
```

## width, depth, height

These fields indicate the total size of the space that the user interface should fill up. The values will be assigned to the main **SoWidgetLayoutGroup** or **SoPanelGroup** determing the initial structure of the user interface.

## style, units, columns

These fields allow you to specify some layout hints. The field 'style' may take three different values producing a layout based on different layout rules.

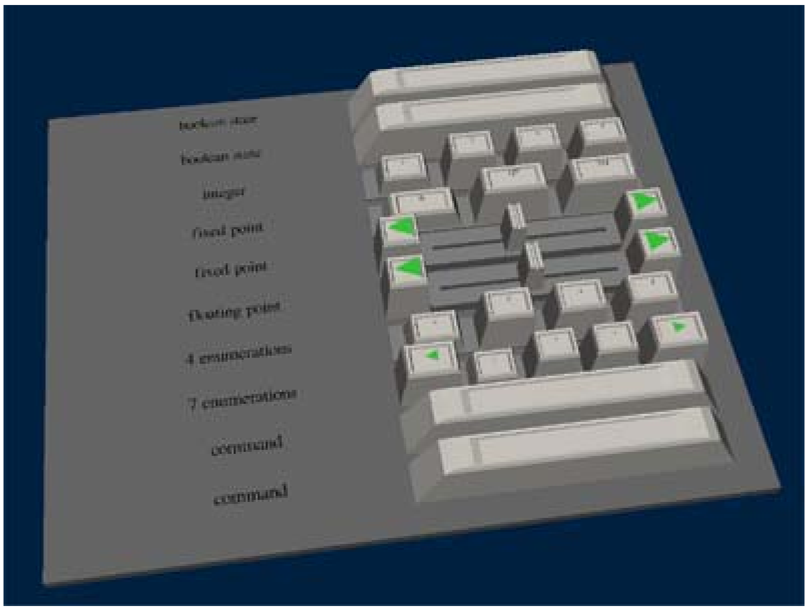
- The default style, the PUC style, arranges the components in a one-column layout with adjacent labels as in the PUC system. This style thus supports only a small number of components. Lower hierarchies of puc groups are ignored.
- The PUCEXT style allows generating an extended version of the PUC style. You can indicate the number of columns in the field 'columns'. To preserve balance, the widgets are placed individually in the column having the lowest level of widgets at that point. Lower hierarchies of puc groups are still not taken in account.
- The STB style is an approach to respect grouping in between the widgets of one panel sheet. The main structure consists of a number of rows that are filled up subsequently with widgets. The widgets of two different SoPucGroup's are never placed in a same row. This implies that grouping is preserved, the layout may however not be balanced. The number of widgets placed in one row can be specified by the field 'units', where each button corresponds to one unit, and each slider and list box to two units.

The PUC style always chooses the longest possibility as label; the PUCEXT and STB style always chooses the shortest one.

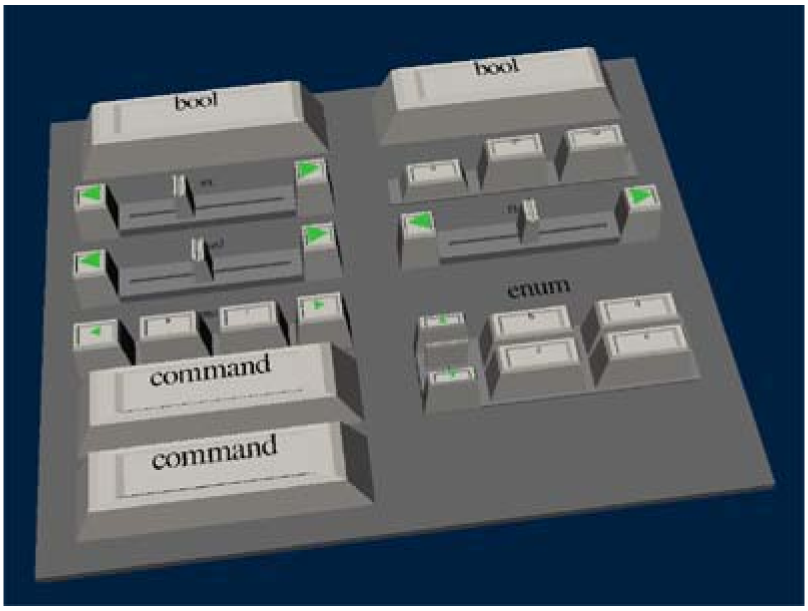
## Examples

```
    DEF LAYOUT_GROUP SoPucPipLayout {
     width 22 depth 18 height 2.0
     style PUC

     pucAppliances SoPucServer {
      serverName "Studierstube Demo"
      devices NodeKitListPart { containerNode Group {
        DEF RADIUSDEVICE SoPucDevice {
          deviceName "example"
          groups NodeKitListPart { containerNode Group {
```

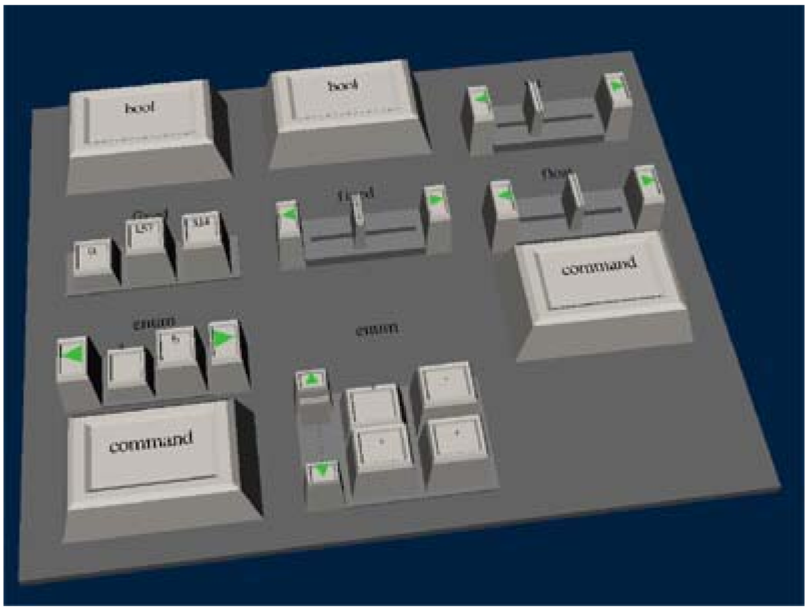
```
          SoPucGroup {
             priority 10
             members NodeKitListPart { containerNode Group {

             DEF GROUP1 SoPucGroup {
                members NodeKitListPart { containerNode Group {
                      DEF BOOL_STATE1 SoPucStateBool { labels ["bool",
                        "boolean state"]}
                      DEF BOOL_STATE2 SoPucStateBool { labels ["bool",
                        "boolean state"]}
                      DEF INT_STATE1 SoPucStateInt { labels ["int",
                        "integer"] min 0 max 6 incr 2 }
                      DEF FIXED_STATE1 SoPucStateFixed { labels
                        ["fixed", "fixed point"] min 0 max 3.14 incr
                        1.57 pointpos 2}
                      DEF FIXED_STATE2 SoPucStateFixed { labels
                        ["fixed", "fixed point"] min 0 max 2 incr 0.1 }
                      DEF FLOAT_STATE SoPucStateFloat { labels ["float",
                        "floating point"] min 0 max 1 }
             }}} # SoPucGroup

             DEF GROUP2 SoPucGroup {
                members NodeKitListPart { containerNode Group {

                     DEF ENUM_STATE1 SoPucStateEnumerated { labels
                        ["enum", "4 enumerations"]  valueLabels ["a",
                        "b", "c", "d"]}
                     DEF ENUM_STATE2 SoPucStateEnumerated { labels
                        ["enum", "7 enumerations"]  valueLabels ["a",
                        "b", "c", "d", "e", "f", "g"]}
                     DEF COMMAND1 SoPucCommand { labels ["command"]}
                     DEF COMMAND2 SoPucCommand { labels ["command"]}


             }}} # SoPucGroup
          }}} # SoPucGroup

     }}} # SoPucDevice
 }}} # SoPucServer
}#SoPucPipLayout
```



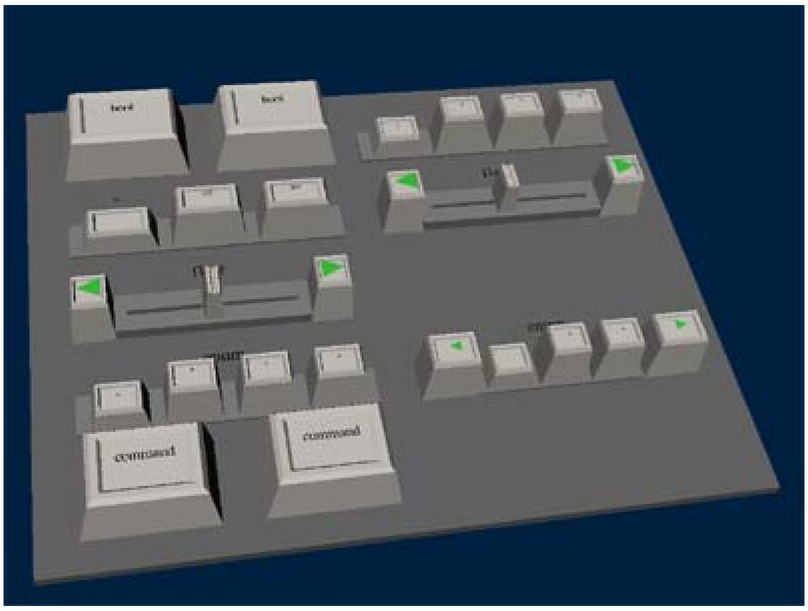
style    PUC



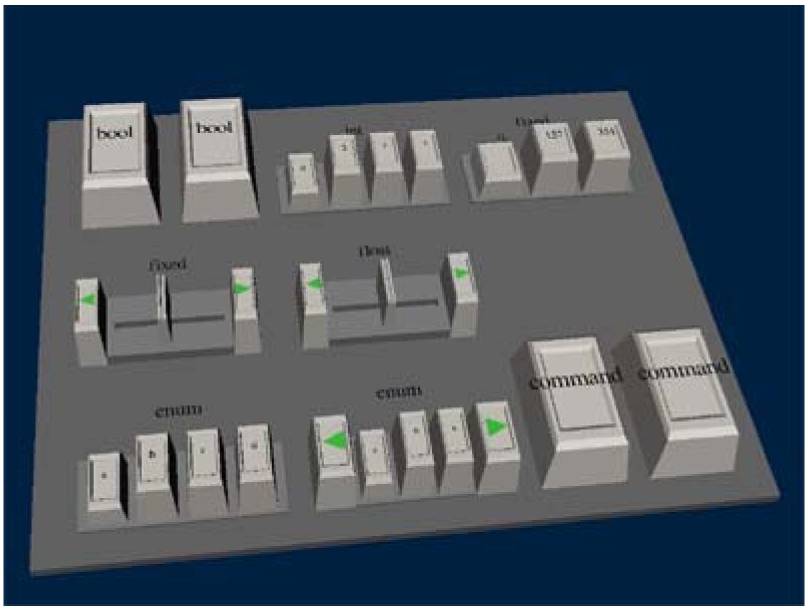
```
style   PUCEXT
columns 2
```



```
style   PUCEXT
columns 3
```



```
style   STB
units 4
```



```
style   STB
units 6
```