Encapsulating Interaction Techniques of 3D Language Editors in Visual Patterns

Jan Wolter  
University of Paderborn  
Department of Computer Science  
Fürstenallee 11  
33102 Paderborn, Germany  
jan.wolter@uni-paderborn.de

Uwe Kastens  
University of Paderborn  
Department of Computer Science  
Fürstenallee 11  
33102 Paderborn, Germany  
uwe@uni-paderborn.de

ABSTRACT

The implementation of three-dimensional visual languages requires a wide range of conceptual and technical knowledge on issues for 3D graphics and textual language processing. Our generator framework DEViL3D incorporates such knowledge and supports the design of visual 3D languages and their implementation from high-level specifications. Such 3D languages arise from different modeling domains that make use of three-dimensional representations, e.g., the “ball-and-stick” models of molecules. The front-end of a 3D language implementation is a dedicated 3D graphical structure editor, which offers interaction and navigation techniques to construct programs in their domain. These techniques allow to manipulate the 3D program directly using operations to insert, move, and restructure objects. We have developed canned solutions for all such techniques that are encapsulated in visual patterns, which are provided by our generator. The designer of a particular 3D language only has to apply visual patterns to constructs of the abstract syntax, which defines the basic structure of the language. We have complemented our development with a usability study. Participants had to solve several tasks with different interaction or navigation techniques. The results partially indicate a significant advantage of one technique over another.

Categories and Subject Descriptors

D.1.7 [Programming Techniques]: Visual Programming; D.2.1 [Software Engineering]: Requirements/Specifications—languages, tools; D.3.4 [Programming Languages]: Processors—code generation, compilers; H.5.2 [Information Interfaces and Presentation]: User Interfaces; I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

VINCI ’14, August 05–08 2014, Sydney, NSW, Australia  
Copyright is held by the owner/author(s). Publication rights licensed to ACM.  
ACM 978-1-4503-2765-7/14/08...$15.00.  
http://dx.doi.org/10.1145/2636240.2636840

Keywords

3D user interfaces; 3D interaction; 3D navigation; visual languages; visual patterns; automated generation.

1. INTRODUCTION

Visual languages are particularly beneficial for domain-specific applications. They support graphical metaphors of their domain, which enable domain experts to use their conventional way of description. Most visual languages use two-dimensional representations, but for some purposes, using the third dimension is advantageous or necessary: Inherently three-dimensional languages as architecture-like languages can be represented without loss of information only in 3D: The “ball-and-stick” models of molecules arrange the atoms according to the electron cloud repulsion, which ensures the use of a three-dimensional structure. Some 3D languages assign a semantic meaning to each dimension: The web-based 3D editor ToneCraft [5] uses boxes to compose music, whereby each box represents three different properties according to its position along the three axis. Petri Nets can be represented in 3D [15] to separate different aspects. Each aspect is represented on a different plane, which can be stacked together to one Petri Net. Furthermore, the third dimension can be used to overcome limitations in 2D arrangements: For example, some UML diagrams can be extended to 3D to reduce the problem of intersecting edges [7].

To construct instances of visual 2D languages, sometimes a pencil and a piece of paper are sufficient to sketch a draft. To construct them with editor support, two distinct editing principles exist: one takes up the sketching approach, which allows free-hand editing with subsequent parsing and the other pursues structured editing, which ensures syntactically correct programs at any time. Constructing 3D programs on a piece of paper is not a promising approach. Editor-based approaches supporting free-hand sketching for complex 3D programs are also not auspicious. However, 3D structure editors are feasible to construct 3D programs oriented on its structure. Its usability depends on the provided interaction and navigation techniques. For 3D editors—compared with 2D editors—interaction as well as navigation techniques are more complex due to an increased degree of freedom in 3D space. The navigation inside a view has to be controlled by a first-person-view camera in various ways. One particularly challenging task is the interaction with hierarchically nested language constructs being completely enclosed by their parent construct, just like Russian Matryoshka dolls. Because
of these and further aspects, the development of 3D language editors is a challenging task.

Our goal is to support language designers in the development of 3D language implementations, whose development is justified, only if the effort is appropriately small. Therefore, effective generator systems are useful. We are developing the generator framework DEViL3D (Development Environment for Visual Languages in 3D) that generates 3D language implementations from high-level specifications in a reasonable amount of time. Language designers using DEViL3D do not need to know how to implement 3D graphics because our generator completely encapsulates expertise related to 3D graphics including all interaction and navigation tasks. The interaction operations applicable to a specific language construct are always aligned to its needs.

Such issues are feasible since the language editor specification with DEViL3D is always oriented on the structure of the visual 3D language. It starts with the specification of the language’s abstract syntax that defines a concrete structure how language objects can be combined. Interaction operations, as inserting or moving objects, are directly applied to the underlying abstract syntax. Systematic solutions of such operations are realized in DEViL3D completely independent of the language’s specific abstract syntax. So called visual patterns are our vehicle to encapsulate them. Visual patterns introduce another level of abstraction on top of the specifications for the visual language generator. They provide a conceptual relation between the abstract syntax and the structure of visual representations, as well as means for canned specifications of visual language constructs including their interaction operations. Most of the editor front-end implementation can be done by applying such visual patterns.

The remainder of this paper is structured as follows. The next section gives a brief overview of DEViL3D and the process to specify 3D language editors. In Section 3, interaction mechanisms and their encapsulation in visual patterns is presented. In addition to interaction facilities, navigation tasks are particular important for three-dimensional editors and will be presented in Section 4. Afterwards, Section 5 presents a usability study which measures the effectiveness of interaction and navigation techniques from the perspective of an editor user. Then we discuss related work and show relations to our approach. Section 7 concludes the paper.

2. DEViL3D

The generator system DEViL3D [26] allows to generate 3D structure editors—supporting the direct manipulation paradigm [20]—from high-level specifications. It applies approved concepts of the predecessor system DEViL [19, 18]. On top of it, new aspects, necessary to construct three-dimensional programs, were developed. Figure 1 visualizes the specification process. The language designer has to specify the abstract structure, the visual representations, and the optional code generation. As input, DEViL3D gets such a set of specifications and generates a language processor which has a dedicated 3D graphical structure editor as its front-end. Domain experts use such editors to construct three-dimensional programs of their domain, e.g., 3D Petri Nets.

The abstract structure is the central data structure and describes the language constructs and how they are connected, without defining their concrete representations. To specify the abstract structure, a specifically tailored textual domain-specific language is used, which is strongly related to object oriented programming languages. From the abstract structure a context-free grammar is generated that is transformed into an attribute grammar. The specification of the visual representation is based on such attribute grammars, which are translated into computations of the graphical representation and arrange objects in 3D space.

On top of these specifications, DEViL3D uses visual patterns, which promote powerful abstractions, modularizations, and reuse of specifications. They are used to compose a visual language editor without writing lower level specifications, explicitly. It comprises common properties of visual language elements with respect to their abstract structure, the interaction techniques needed for them, and their visual concept. DEViL3D provides a library of visual patterns that encapsulate common representation arrangements like three-dimensional sets, lists, line connections, or cone-trees. They are defined as visual roles, which can be assigned to symbols of the grammar in a declarative way. A designer of a three-dimensional visual language may express the representation arrangement of a language construct by associating visual patterns to certain constructs of the grammar. They can be parameterized to cover a wide range of visual representations, but the actual language designer is never involved to implement interaction properties.

The concrete visual appearance of language constructs defining its shapes is based on specifications of generic depictions [27], which are referenced by visual patterns. The generic depictions play the role of building blocks from which the three-dimensional program is constructed. As a 3D graphics layer, the jMonkeyEngine [11] is used, which, in turn, uses OpenGL to render 3D scenes. To organize the 3D scene efficiently, the provided scene graph—a data structure to organize the 3D scene—is used extensively.

Similar to the visual representation, the specification of code generators is also based on attribute grammars. The language designer can use the concepts and libraries of the well-known Eli system [9] to transform the 3D program into any textual target language.

2.1 Specification Example

In this section we will give the reader a more concrete impression, how the specification of the visual representation

![Figure 1: Specification process with DEViL3D.](image-url)
Figure 2: Excerpt of the visual representation of the Petri Net editor.

As an example, we use the specification of the Petri Net editor, which goes back to an idea of Rölke [15]. Here, we will focus on the specification of the visual representation and omit code specifying the abstract structure. The abstract structure describes a 3D Petri Net consisting of a set of planes and on each of them transitions and places can be positioned. Transitions and places can be connected with each other, independent of the plane they are located on. Figure 2 shows an excerpt of the specification of the visual representation, with highlighted applications of visual patterns in bold. The entirety of plane objects (represented by the symbol Root_planes) in the abstract structure is represented as a list by associating the role VPList of the list pattern to the symbol. The aggregated plane objects play the role of list elements by using the VPListElement role. In line 4, the list pattern is parameterized to define the direction along the list grows. Following a similar principle, the place or transition nodes (Figure 2 shows only the code for places; the code for transitions is analogous) are represented according to the plane pattern. The tokens, which are located inside a place, are positioned under the terms of the 3D set pattern.

The next section presents concrete interaction techniques, language designers integrate in their editors by applying visual patterns, as shown here.

3. INTERACTION TECHNIQUES

A fundamental interaction technique is the insertion of language constructs. DEViL3D supports different operations that are triggered by so called insertion indicators to insert language constructs into the 3D scene. When an editor user clicks a button on the left-hand side-bar of the view to insert a language construct, all appropriate insertion indicators appear in the 3D canvas and highlight valid positions, where such an object can be inserted. The appearance of the insertion indicator differs depending on its context.

The insertion indicator for elements that are organized in a set—as the tokens inside a place in the Petri Net example—support locating at an arbitrary position. A cube-shaped insertion indicator including a plane to indicate a three-dimensional position supports this (see Figure 3a). The plane may be moved along the z-axis. The 3D position of the object is then determined by a mouse click on the plane: the x- and y-position on the screen projected onto the plane, combined with the z-position of the plane. Elements that are represented on a plane may adopt an arbitrary position on such a plane. Hence, the insertion indicator is plane-shaped (see Figure 3c). The editor user clicks on the plane to insert a new element and determine its position. Furthermore, insertion indicators for set and plane elements ensure that the objects do not interpenetrate each other. For elements that are organized in a list, plane-shaped insertion indicators for all valid insertion positions are available (see Figure 3e). When the editor user touches a plane, it gets highlighted. To insert a new object, the user clicks on the plane, whereon the new element is inserted and objects behind the new one move up.

The illustrated context-specific manifestations of insertion indicators and its operations to insert new objects are encapsulated in the visual patterns that describe sets, planes, and lists. The interaction tasks are automatically tailored to the needs of the representation of a visual pattern. In accordance with this approach, we have developed further visual patterns (for example matrix, cube, or connection pattern), including specialized insertion operations that fits to the needs of its representation. Language designers only have to select visual patterns to associate them to symbols of the abstract structure (see Section 2.1). As a result, they receive a high-level implementation without any further effort.

After the insertion of a language construct, the editor user may restructure or modify properties of objects. This might include manipulation operations to translate, scale, or rotate objects. For such purposes the generated editors provide so called widgets [4] that may be attached to the object under consideration. Such widgets are automatically adjusted to the requirements of the object according to the degrees of freedom the object has, defined by the associated visual pattern. The widget to translate objects of a 3D set provides three arrows and three planes between them (see Figure 3b). The user may translate it along one dimension separately (via an arrow) or along two dimensions simultaneously (via a plane). The widget that translates plane elements only consists of two arrows and one plane between them, since the elements can only be moved on the plane (see Figure 3d). The translation of list elements is limited to the dimension in which the list grows. This produces a widget consisting only of one arrow (see Figure 3f). Analogous to the translation widget there are further widgets to scale or rotate language constructs.
Interaction operations that are independent of visual patterns are applicable to all objects and in any context without assistance by the language designer. For example, for all language constructs, basic editor operations such as cut-and-paste and context menus, to access further properties of the construct, are supported directly in the 3D scene. The interaction with objects being nested in others and multiple simultaneous selection of language constructs are also applicable for all objects.

Hierarchically nested language constructs that are completely enclosed by its parent construct need specific mechanisms to interact with. Like in the real world, only the surrounding object can be picked directly. Technically, all interaction tasks supported by our 3D editors can be performed by using a classical 2D mouse. For that purpose, the ray casting metaphor is used. For every situation, where the user wants to interact with an object, a ray starting at the mouse cursor is shot into the 3D scene to determine the objects that are intersected by the ray. The default behavior is that only the first object is selected. We have developed three methods to support the direct interaction, even with objects that are deeply nested in one another. If the user clicks at a position, so that the ray intersects multiple objects, a context menu is shown that lists all such objects (see Figure 4). The user may choose the intended object from the list to apply further operations, e.g., translate it. Another approach is to let users hide the shapes of the outermost language constructs by using a function accessible from its context menu. The other way is, to select the object, which is intersected by the ray and located furthest inside directly. This object is necessarily smaller as its surrounding objects and consequently hard to pick.

In some cases it is desirable to select multiple objects simultaneously. The generated structure editors provide different methods to accomplish this task. As in many editors, whether 2D or 3D, the user may select multiple objects, one after the other by picking them, while a special control key is pressed. There are 3D scenes where this approach is not
Figure 4: Select nested objects by context menu.

applicable: Objects that are placed far away from the actual camera position appear much smaller than near located objects, which makes a precise selection difficult. To overcome this problem, we have developed a cylinder metaphor inspired by Tavanti et al. [23]. The editor user sees only the circle-shaped cylinder cover on the 2D monitor screen (see Figure 5a). The cylinder expands into the 3D scene and selects all objects that are enclosed by it. Of course, the editor user can adjust the size of the lateral surface. But again, even this method is not distinct enough, if the objects intended for selection cannot be captured by a cylinder. A lasso metaphor may be better suited: An arbitrary polygon can be created, which expands into the 3D scene as well to select the objects (see Figure 5b).

Figure 5: Two ways to select multiple objects simultaneously.

4. NAVIGATION TECHNIQUES

Navigation operations in 2D environments are less complex, since they are mostly limited to scrolling the view along two axes. The navigation in 3D environments is more than scrolling the view along three instead of two axes. It becomes clear, if we distinguish 2D and 3D environments by its degree of freedom (DOF). It describes the possibilities of object placement in space. In 2D space there are three DOFs: translation along the two axes and rotation around the neutral point. In contrast, the DOFs in three-dimensional space are twice as many: translation and rotation along all three axes. Furthermore, the higher degree of complexity is observable on objects that occlude each other. Hence, it is necessary to equip 3D editors with mechanisms to help navigating inside 3D space properly.

Like many other 3D environments, 3D structure editors generated by DEVIL3D come with a first-person-view camera. This applies to all editors for any 3D language without effort for the language designer. Such a camera can be used to address the three main tasks of navigation (according to Bowman et al. [2]): explore, search, and maneuver. The user can explore the scene without an explicit target, search a particular target, or maneuver the camera to position the viewpoint more precisely. The generated editors support these tasks in different ways. One mode to control the camera is to use mouse and keyboard: moving the mouse rotates the camera and pressing special keys lets move the camera forward, backward, upward, or downward. Such a method might be particularly useful for users who are familiar with 3D computer games, which use the same navigation control.

The other possibility to control the camera is using the navigation sphere (see Figure 6a) that is located in the upper right corner of the view. The sphere is capable to rotate the camera and has two different modes: the free mode and an axes-based mode. The free mode is inspired by the arcball metaphor [21], which allows to rotate the camera around all three axes at the same time. Whereas the axes-based mode grants the rotation around one axis separately. Our hypothesis is that the navigation via the navigation sphere, based on free rotation, fits better to the needs of experienced users. All others are served better with the axes-based sphere. Furthermore, the navigation sphere provides around its sphere different segments labeled with cardinal points to switch the viewing position rapidly and show the scene from eight distinct positions. Moreover, the user is able to save a particular camera position, which can be accessed by the H-button and leads back to the saved home position.

Users who are interested to inspect a language construct as a particular point of interest (POI) are able to access the orbiting mode. Once enabled, the camera moves immediately to the POI and orbits around it, while the user clicks a mouse button. Moving the mouse changes the position onto the orbit and shows the POI from another perspective (see Figure 6b).

The problem of occluding objects can be solved through the before mentioned techniques by reposition the camera to show the 3D scene from another perspective. But under some circumstances repeatedly repositioning is annoying. In such cases, a prospect that avoids a repositioning is preferable. Three additional lateral views showing the scene orthogonal to each of the three axes avoid repeated repositioning (compare Figure 6c). Lateral views are well known in CAD software. Users of our generated structure editors can switch to it at any time by using the button in the upper left corner.

To bring users into a position in which they are aware of its position in space, the 3D view comprises an overview widget.
Figure 6: Navigation facilities.

that can be switched on by a button in the lower left corner (Figure 6d). It shows the scene from a more distant position, which adds the global context to the user’s minds. A further orientation guide is the *axis of abscissas* in the lower right corner, which rotates according to the camera.

### 5. USABILITY STUDY

To evaluate the DEViL3D system, two distinct usability levels are observable: the *usability of the generator* and the *usability of the generated 3D editors*. The first level concerns questions, how easily a structure editor is specified for a small language and how satisfied language designers are with the generator. The latter level treats investigations regarding, how intuitive and easy to use the generated 3D editors are and how efficient users can accomplish program-related tasks. Considering that the specification level of DEViL3D is quite close to its predecessor system DEVIL, the specification-style is very similar, especially the syntax to specify abstract structures and visual representations using visual patterns are conceptually unchanged. For this reason, no new cognitions related to the generator level can be expected. Evaluation results concerning DEVIL’s usability can be found in Schmidt et al. [17].

DEViL3D has been used to generate a set of 3D languages, amongst others, editors for Petri Nets, molecules, “music in space”, generic depictions [27], 3D UML sequence diagrams, and kitchen furniture (for details of some of these languages the reader is referred to [28]). A couple of these languages were specified by students as part of their bachelor’s or master’s thesis. From our observations about these language specifications, the DEViL3D generator performs as well as its predecessor DEVIL.

In this usability study, we focus on evaluation aspects related to the generated 3D editors and answer questions, how sufficient their interaction and navigation techniques are.

#### 5.1 Research questions

The 3D editors generated by DEViL3D comprise general features to enhance the productivity like cut-and-paste, undo/redo support, export of visual representations as image files, save 3D programs as XML data files, a tree view to browse and edit programs on its structure level, and searching for language constructs by strings. Such features are useful in any editor, but the contribution of this paper are interaction techniques to construct three-dimensional programs. We are interested in usability aspects of our interaction and navigation techniques. For a lot of interaction and navigation tasks, editors generated by DEViL3D provide competing techniques solving tasks in different ways. We are particularly interested, which of these techniques the user would prefer and if there is a measurable time benefit one technique has compared to another. In detail, our usability study focuses on the following questions:

- How good perform the navigation techniques and which technique would the users prefer?
- Is there a positive impact when using the lateral views in the case of occluding objects?
• Can users easily interact with hierarchically nested objects?
• Which technique for multiple object selection (lasso or cylinder metaphor) is preferred by the users?

5.2 Participants
We observed 18 participants. All of them were either students (89%) or IT staff (11%) at the University of Paderborn. All students were studying computer science or much related subjects of study. 50% were enrolled in the bachelor’s program, 31% were enrolled in the master’s program and 19% were graduate students who are working on their PhD. The mean age of the participants was 23.9 (standard deviation=2.9). We asked the participants for previous knowledge in playing 3D games, using 3D editors, and 3D programming: 94% of them had experience in 3D games and 59% of this group even for more than ten years. 83% had used 3D editors before and 72% had practice in 3D programming.

5.3 Task Descriptions and Results
To answer the aforementioned research questions, we have designed an empirical usability test. It comprised four different parts, each related to give answers to one question. For each part, participants had to use a particular 3D editor and solve a specific task by using interaction or navigation techniques, we wanted to observe. The editors were designed to let participants solve the tasks without distraction by issues not related to the task. We have generated the editor with DEVIL3D and tailored it to the specific tasks, but the editor cannot be regarded as a front-end of an appropriate complex 3D language. For example, such editors provide dedicated 3D scenes consisting of particular objects. The participant’s task was to answer questions about the scene by using specific techniques. Before the participants started the tasks, the experimenter first introduced how the interaction or navigation techniques work and let users start with a warm-up task to gain experience.

The participants were divided into two groups and randomly assigned either to group A or group B. Each group had to solve the same problem by using different techniques. For example, counting the number of specific objects in the scene by rotating the camera with the navigation sphere either in axes-based or free mode. Such an approach avoids learning effects and lets us compare the performance of two distinct groups. The choice of the kind of the interaction technique is the independent variable. We mainly used two dependent variables counting the time, required to solve the task, and the information if the task was solved correctly. At the end of each task, a question was asked with options on a 6-step Likert-like scale to receive the participant’s individual assessment to the used technique. The whole study lasted for each participant—including introduction, warm-up, and the four tests—between 20 and 30 minutes. In the following we describe the four tasks including the determined results in detail.

5.3.1 Navigation
Within the generated editors, users can navigate by using the navigation sphere either in axes-based or free mode or the keyboard in combination with the mouse. We specified an editor, where 3D objects were placed in a maze-like 3D scene. The participants had to count language constructs with certain properties. One group used the navigation sphere in axes-based mode, the other in free mode. According to a Welch two samples t-test the time required to solve the task, was significantly faster (p-value=0.02201) in free mode (mean=20sec., standard deviation=8.746sec., median=21sec.) than in axes-based mode (mean=42.4sec., standard deviation=23.367sec., median=40sec.).

5.3.2 Lateral views
The 3D scene, we used to test the lateral views, consisted of two kinds of language constructs. They were either represented as boxes or spheres and occlude each other considered from the starting point of view. The task was to count occurrences of both language constructs. One group was able to use the lateral views and the other not. We assumed using the lateral views is helpful, since it shows the scene orthogonal to each of the axes. However, the opposite was the case: the participants using the lateral views needed on average 28.2 seconds (standard deviation=20.675sec., median=19sec.) and the group without lateral views only 12.4 seconds on average (standard deviation=5.294sec., median=13sec.)—compare Figure 8. According to a Wilcoxon test (which we used since a Shapiro-Wilk normality test indicates no normal distribution for the measured times) solving the task with help of the lateral views needed significantly more time than without lateral views. But the amount of cor-
5.3.3 Interaction with nested structures

The 3D scene to test the interaction with nested structures consisted of an outermost box including four spheres which in turn includes a set of boxes. One of these boxes was colored red, the others were blue. The participants’ task was to access the innermost red box inside two spheres and read out the value of one of its attributes. One group used the approach, which shows the list of picked language constructs in a context menu. The other group had to hide the shapes of the outermost box and the spheres to achieve direct access to the red boxes. In case of the required time, we found no significant difference between the two approaches: The first group need on average 16.2 seconds (standard deviation=5.826sec., median=17sec.) and the latter one 16.9 seconds (standard deviation=4.216sec., median=18sec.). Each participant of the two groups correctly solved the task. According to the Welch test, the cylinder metaphor was significantly faster (p-value=0.001891) than the lasso metaphor (compare Figure 10). The number of needed selection operations was greater by using the cylinder (mean=5.4, standard deviation=1.81, median=6) than the lasso (mean=2.4, standard deviation=0.527, median=2). Nevertheless, using the cylinder was faster. The reason for this is that using the lasso needs more accuracy to define the surrounding polygon whereas the cylinder needs less precision. According to the answers, we received from the Likert-scale, the participants assessed the lasso metaphor slightly better.

5.3.4 Multiple object selection

The editor used to test multiple object selection consisted of a 3D scene comprising three kinds of language constructs that were represented according to the 3D set pattern. The participants had the task to delete as fast as possible all objects of one kind. To do so, they selected multiple objects by using either the lasso or the cylinder and deleted the selection afterwards by using the keyboard. They could partition this task in as many insertion operations as they want. The group using the cylinder needed on average 11.1 seconds (standard deviation=2.472sec., median=10sec.) to solve the task; the lasso group required 17.44 seconds (standard deviation=4.216sec., median=18sec.). Each participant of the two groups correctly solved the task. According to the Welch test, the cylinder metaphor was significantly faster (p-value=0.001891) than the lasso metaphor (compare Figure 10). The number of needed selection operations was greater by using the cylinder (mean=5.4, standard deviation=1.81, median=6) than the lasso (mean=2.4, standard deviation=0.527, median=2). Nevertheless, using the cylinder was faster. The reason for this is that using the lasso needs more accuracy to define the surrounding polygon whereas the cylinder needs less precision. According to the answers, we received from the Likert-scale, the participants assessed the lasso metaphor slightly better.

6. RELATED WORK

The idea for three-dimensional languages goes back to a publication of Glinert [6]. Najork [14] developed the first 3D language *Cube*, which is semantically similar to Prolog and makes use of the data-flow paradigm. Afterwards, some other 3D languages were developed, which all differ with respect to interaction and navigation facilities that are at different levels of maturity. Systems for development of 3D languages and generating their implementation are really new. To the best of our knowledge, the only approach, except ours, related to the generation of 3D languages, is an exploration [24] in the context of Minas’ generator systems DiaGen/DiaMeta [12, 13].
No other approaches are known to us—neither in the context of visual languages nor other application domains—that generate fully usable 3D editors from a predefined set of canned solutions. For 2D editors some approaches are established: DEViL3D’s predecessor DEViL also generates editors by using visual patterns. They encapsulate interaction techniques that allow structured editing in two dimensions. The generator systems DiaGen/DiaMeta generate 2D language editors that support free-hand as well as structured editing. Technically, a graph transform calls edit operations to support structured editing. Bottoni and Grau present [1] a suite of metamodels defining syntactic characteristics of two-dimensional visual languages which have some similarities with our visual patterns.

Some ideas behind the interaction techniques our editors generated with DEViL3D use, are adapted from other 3D editors from different domains. The widgets to translate, rotate, or scale constructs as well as the lateral views are known from a variety of CAD editors. The cylinder metaphor to select multiple constructs is inspired by Tavanti et al. [23]. A navigation sphere related to ours is used in the well-known 3D modeling tool Autodesk 3ds Max and the procedure, which rotates the camera according to the mouse movements onto the sphere is taken from [21]. The orbiting approach is inspired by Tan et al. [22]. There are a lot of papers that evaluate 3D interaction techniques and the advantages of 3D representations compared with 2D: [10, 25, 8, 3].

7. CONCLUSION AND OUTLOOK

We have presented an approach to encapsulate interaction techniques of 3D language editors in visual patterns. Such patterns are part of our generator system DEViL3D and allow language designers to generate fully usable 3D language editors by applying such patterns to symbols of the before defined abstract structure. The interaction techniques fit to the needs of the visual pattern’s representation concept—for example, a list or a set—and allow to insert new objects and reorganize them afterwards. Interaction operations that are independent of a particular pattern—like cut-and-paste, multiple selection of objects, and interaction with hierarchical nested language constructs—are applicable to all language constructs. Furthermore, powerful techniques to navigate inside the 3D scene were developed. The editors generated with DEViL3D support three different ways to navigate inside the 3D world. Additionally, the editor can orbit around an object, switch on lateral views, and an overview widget to approve the user’s orientation inside the scene.

In our evaluation, we measured the effectiveness of some of the interaction techniques a DEViL3D editor provides. The basic idea was to compare competing techniques by letting two groups of participants solve the same task with different techniques. We found a significant evidence of the free mode of the navigation sphere over the axes-based mode. Furthermore, the cylinder metaphor to select multiple objects, performs faster than the lasso. The tests for the lateral views and interaction with nested structures indicate more ambiguous results, which can be explained by the fact that such a new features claim extra cognitive resources.

For the future, we plan to equip the 3D editors with the opportunity to let users gain an immersive 3D perception by using stereoscopic hardware. It would be interesting to see, if the interaction and navigation techniques perform just like in our present configuration. A further usability study can answer the question if immersive 3D perception has advantages over the usage of a classical 2D monitor. Furthermore, we intend to extend DEViL3D to generate web-based 3D editors running in a browser additionally to the local editors presented here. The support of touch interaction could be also beneficial. The touch interaction operations must also be encapsulated in the visual patterns. A re-generation of all so far specified 3D language editors would then benefit from the touch interaction.

8. ACKNOWLEDGMENTS

The authors gratefully acknowledge the support of the German Research Foundation (Deutsche Forschungsgemeinschaft – DFG); contract no. KA 537/6-1. Furthermore, the authors would like to thank Elena Rybka and Johann Rybka who developed [16] some of the interaction and navigation techniques presented in this paper.

9. REFERENCES


