Abstract

WEB3D is a family of interactive online three-dimensional (3D) graphics standards that includes such powerful technologies as Extensible 3D (X3D) modeling language. The X3D language has been widely adopted by professional organizations, researchers, and 3D designers worldwide. Despite the rich functionality, the language does not currently provide tools for rapid development of conventional graphical user interfaces (GUIs). An X3D author is responsible for building—from primitives—a purpose-specific set of required interface components, often for a single use.

In this thesis we address the challenge of creating consistent, efficient, interactive, and visually appealing GUIs by proposing the X3D User Interface (X3DUI) library. This library includes a wide range of cross-compatible X3D widgets, equipped with configurable appearance and behavior. With X3DUI, we attempt to standardize the GUI construction across various X3D-driven projects, and improve the reusability, compatibility, adaptability, readability, and flexibility of many existing applications.
Extending the Web3D: Design of Conventional GUI Libraries in X3D

by

Ivan Sopin

A Thesis submitted to the Faculty of the School of Graduate Studies at Armstrong Atlantic State University in Partial Fulfillment of the Requirements of the Degree Master of Science in Computer Science

Savannah, Georgia
2010
Extending the Web3D: Design of
Conventional GUI Libraries in X3D

by

Ivan Sopin

Felix Hamza-Lup, Ph.D., Chair

Ray Hashemi, Ph.D.

Raymond Greenlaw, Ph.D.

Hong Zhang, Ph.D.

Approved:

Michael E. Price, Ed.D.                        Date
Assistant Vice President for Graduate Studies and Sponsored Programs
Acknowledgements

This work by no means has been a one-person project. I am grateful to everyone who helped me—knowingly or not—at various stages of the development and writing. These people include my research coordinator Dr. Felix Hamza-Lup, who mentored me and kept me motivated throughout my academic career; Dr. Raymond Greenlaw, who helped me improve my technical writing skills; Brian Hay, who inspired me with his ideas; Peter Schickel and his team, who provided me with a BitManagement Software Development Toolkit; authors of countless X3D scenes, who graciously shared their knowledge and findings; and of course my fiancée, who would let me sleep until noon after working all night.
## Contents

**LIST OF FIGURES** ........................................................................................................... viii  
**LIST OF TABLES** ........................................................................................................... x  
**LIST OF LISTINGS** ....................................................................................................... xi  

1 INTRODUCTION .................................................................................................................. 1  
   1.1 Motivation ..................................................................................................................... 1  
   1.2 Outline of This Work .................................................................................................... 2  
   1.3 What is Web3D? ......................................................................................................... 3  
      1.3.1 Virtual Reality Modeling Language ................................................................. 4  
      1.3.2 Adobe Flash ........................................................................................................ 5  
      1.3.3 Java 3D ............................................................................................................... 6  
      1.3.4 Extensible 3D ..................................................................................................... 7  
      1.3.5 Other Technologies ............................................................................................ 8
2 RELATED WORK .......................................................................................................................... 9
  2.1 INTERFACE CLASSIFICATION ......................................................................................... 9
  2.2 X3D-BASED INTERFACES ............................................................................................... 9
  2.3 HTML-BASED INTERFACES ........................................................................................... 12
  2.4 OTHER APPROACHES ........................................................................................................ 15
3 DESIGN ................................................................................................................................... 17
  3.1 DIMENSIONALITY ............................................................................................................ 17
  3.2 VISUALIZATION IN X3D .................................................................................................. 18
  3.3 PRESENTATION TECHNIQUES .......................................................................................... 22
  3.4 PROTOTYPING .................................................................................................................. 27
4 IMPLEMENTATION .................................................................................................................. 31
  4.1 STRUCTURE OVERVIEW .................................................................................................. 31
  4.2 CORE COMPONENTS ......................................................................................................... 33
    4.2.1 Display ....................................................................................................................... 33
    4.2.2 Settings ...................................................................................................................... 36
  4.3 BASE VISUAL COMPONENTS ............................................................................................ 37
    4.3.2 Rectangle .................................................................................................................... 37
    4.3.3 Layer .......................................................................................................................... 37
    4.3.4 Plane .......................................................................................................................... 37
    4.3.5 Button ....................................................................................................................... 38
    4.3.6 ToggleButton .............................................................................................................. 38
4.3.7 Functionality Overview .................................................................................. 39

4.4 CONVENTIONAL VISUAL COMPONENTS ................................................... 40

4.4.1 TextButton.................................................................................................. 40

4.4.2 TextToggleButton...................................................................................... 40

4.4.3 ControlButton........................................................................................... 41

4.4.4 CheckBox................................................................................................... 42

4.4.5 RadioButton and RadioButtonGroup...................................................... 43

4.4.6 Label........................................................................................................ 44

4.4.7 TextField................................................................................................. 44

4.4.8 ComboBox................................................................................................. 45

4.4.9 HorizontalSlider...................................................................................... 47

4.4.1 Update Mechanisms.................................................................................. 47

4.5 CONTAINER COMPONENTS ........................................................................ 49

4.5.1 Panel........................................................................................................ 49

4.5.2 TabPanel.................................................................................................. 49

4.5.3 Frame....................................................................................................... 50

4.5.4 TaskBar.................................................................................................... 53

4.6 LAYOUTS ...................................................................................................... 54

4.6.1 LayoutManager......................................................................................... 54

4.6.2 BorderLayout.......................................................................................... 55
Figures

Figure 1. Linear accelerator simulator with X3D-based GUI ........................................ 10
Figure 2. Nervous system simulator with HTML-based GUI ........................................ 13
Figure 3. Primitive interactive scenario in X3D .......................................................... 21
Figure 4. HUD technique demonstration .................................................................. 23
Figure 5. HUD-layer penetration ............................................................................. 24
Figure 6. GUI depth-separation example .................................................................. 25
Figure 7. Polygonal (a) and texture-based (b) text rendering in X3D ......................... 26
Figure 8. X3DUI prototype inheritance diagram ...................................................... 33
Figure 9. Button object in unpressed (a) and pressed (b) states .............................. 38
Figure 10. TextButton object in unpressed (a), pressed (b), and disabled (c) states ................................................................. 40
Figure 11. TextToggleButton object in unpressed (a), pressed unreleased (b), pressed released (c), disabled unpressed (d), and disabled pressed (e) states ....... 41
Figure 12. ControlButton object set of a window in normalized (a) and maximized (b) states ................................................................. 42
Figure 13. CheckBox object in unpressed unchecked (a), pressed unchecked (b), unpressed checked (c), pressed checked (d), disabled unchecked (e), and disabled checked (f) states .......................................................... 42

Figure 14. RadioButton object in unpressed unchecked (a), pressed unchecked (b), unpressed checked (c), pressed checked (d), disabled unchecked (e), and disabled checked (f) states .................................. 43

Figure 15. TextField in plain (a), overflow (b), and disabled (c) states .................. 45

Figure 16. ComboBox object collapsed with no selection (a); expanded with no selection (b); collapsed with selection (c); and expanded with selection (d) ....... 46

Figure 17. HorizontalSlider in idle (a), selection (b), and disabled (c) states ...... 47

Figure 18. TabPanel with the first (a), second (b), and third (c) tab activated............ 50

Figure 19. Existing resize cursor variations ................................................................. 51

Figure 20. Frame object in inactive (a), resizing (b), and resized (c) states ........... 52

Figure 21. A virtual desktop using the TaskBar prototype ................................. 53

Figure 22. A Frame object using the GridLayout ....................................................... 57

Figure 23. FlowLayout-based component “folding” on a window resize ............... 58
Table 1. Realization of key OOP concepts in X3D................................................... 28
Table 2. Comparison of base visual prototype attribute sets ........................................... 39
Table 3. Explanation of positive characteristics of X3DUI............................................... 61
Listings

Listing 1. Sample X3D file listing

Listing 2. Script attributes/methods definition example

Listing 3. A simple GUI programmed with X3DUI

Listing 4. Example of BorderLayout usage
Introduction

1.1 Motivation
With 3D graphics firmly entering the domain of the Internet, a new niche of virtual visualization—called WEB3D—has formed. One of the leading technologies united in the realm of WEB3D is the Extensible 3D (X3D) modeling language. Due to immense graphical and scripting capabilities, X3D has become a mature graphics standard with wide recognition among professional organizations, researchers, 3D designers, and WEB3D enthusiasts around the globe. Yet one important feature that the language still lacks is a toolset for creating conventional user interfaces (UIs). Typically, to provide a UI for each new application, the X3D author has to design an entirely different set of interface components. Most of these implementations are very limited in functionality and only serve their ad-hoc purpose.

In this thesis we address the challenge of creating consistent, efficient, easily controllable, and visually appealing interfaces by proposing the Extensible 3D User Interface (X3DUI) library. This library is a wide range of cross-compatible X3D widgets, complete with configurable appearance and behavior. With X3DUI, we attempt to standardize the
UI construction across various X3D developments, and improve the reusability, compatibility, adaptability, readability, and flexibility of many existing applications. X3DUI is composed of traditional Microsoft-Windows-like UI elements, whose configuration parameters in general correspond to analogous realizations in Java or Visual C++.

We further argue that humans are technologically more accustomed to planar interface layouts, and thus the management of virtual 3D content in X3D can be effectively performed through 2D or 2.5D (2D with seeming depth) UIs, rendered using the heads-up-display (HUD) technique. Another reason for reduced dimensionality of the UIs is that the truly three-dimensional interfaces are difficult to operate and communicate via inherently two-dimensional visualization systems, such as computer monitors and overhead projectors.

### 1.2 Outline of This Work

In the rest of this chapter we discuss the origin and key features of WEB3D, and review the major representative technologies. The second chapter presents current research and development efforts in the field of GUI design for X3D-based visualization systems; the common usability- and interactivity-related issues are analyzed. In chapter 3 we examine various presentation-specific aspects of X3DUI and illustrate several usage scenarios. The organization and implementation of the library components are elaborated in chapter 4. We conclude with a brief summary and considerations for future work in chapter 5.
1.3 What is Web3D?

Web3D collectively refers to dozens of visualization frameworks designed to promote the distribution and consumption of 3D content across the Internet. The distinctive feature of any Web3D technology is that it provides a simple mechanism of complementing the existing Web media with custom 3D graphics. Most Web3D formats are readily integrated with HyperText Markup Language (HTML) and displayed via proprietary browser plug-in extensions. (Presently, major Web browsers do not support 3D visualization on their own, as 3D modeling is not yet a part of the HTML specification.) Interactivity is another important aspect of Web3D. To fully take advantage of three dimensions, the user needs instant access to, and effective navigation over the virtual environment; while the developer must be equipped with all the tools necessary to construct the geometry and assign certain dynamic behaviors to it.

1.3 Emergence and History of Web3D

Despite—or due to—the success of desktop 3D applications, much less effort has been put into deploying 3D content on the Web. The cyber-community has been surprisingly reluctant to create and publish Web3D projects, even though the standards and technology, such as VRML, emerged soon after the creation of World Wide Web in the early 1990s [1]. One reason is that thousands of newly-born Web surfers were fully entertained with the suddenly available hypertext and multimedia content. People deliberately avoided the hassle of downloading (mostly via dial-up) a heavy 3D player that there was not much use for. Nevertheless, better protocols, communication channels, and Web lan-
guage frameworks soon followed, and the relevance of the online-3D concept was partially restored.

Here we discuss major Web3D technologies and applications that have been developed over the last fifteen years, from the first releases of VRML to the present-day flavors of X3D.

1.3.1 Virtual Reality Modeling Language

Virtual Reality Modeling (initially, Markup) Language (VRML) was introduced in 1995 as the first 3D graphics standard for the Web. Based on the Open Inventor [2] application programming interface (API), the first version of VRML specification supported a limited set of 3D object definitions and scene descriptions, stored in a plain-text file format. Because VRML 1.0 was not sufficient to provide realistic 3D visualization and greatly lacked interactivity, a significantly extended VRML 2.0 specification was produced a year later. The most complete version of the language was adopted in 1997, when VRML was recognized as an international standard by the International Organization for Standardization (ISO) and the International Electrotechnical Commission [1]. Named VRML97, this standard is still widely used on the Internet along with its successor, X3D.

An advantage of VRML97 over other proprietary standards is that the delivered 3D content is open-source and platform-independent. The VRML code can be viewed and interpreted on any operating system (OS) without prior conversion. As a pioneer in Web3D and a breakthrough in distributed graphics, the language also set high interactivity requirements for all other Web3D standards to come. VRML97 is not merely a set of static scenery, but a dynamic environment allowing the authors to utilize and create various interactive scenarios. The standard has rich scripting capabilities that—through Ex-
ternal Authoring Interface (EAI)—provide access to the structure of the virtual world, enabling the user to modify the existing components and create new components on-the-fly. This functionality is implemented either internally, via VRML scripting nodes; or externally, via Java classes, which can communicate with the 3D world using EAI as well.

The language offers extensive support for numerous image formats and texture types; audio, video, and text nodes; point-set-, line-set-, and polygonal-based geometry; extrapolation; advanced lighting and material properties; object transformations; keyboard-, mouse-, and other sensor-triggered events; navigation scenarios; and so on. The VRML architecture also allows multiple copies of any instantiated object to be reused simultaneously in different parts of the scene.

Most importantly, VRML encourages the composition of complex entities via integrating a number of smaller data structures, called prototypes, in a very object-oriented-like fashion. This feature makes the language easily extensible with various task-specific units that are fully compatible with the existing language structures. However, unlike many other platforms where a set of conventional UI controls with programmable behaviors can be easily applied to the virtual setting, it is the developer’s responsibility to implement (from scratch) all desired interface components in VRML. Several individual attempts have been made to produce a publicly-available library of graphical UI (GUI) components for the language, but, to the best of our knowledge, none of them have been released yet.

1.3.2 Adobe Flash
Developed by Macromedia, the Flash technology was launched in 1996 as a multimedia tool to enrich the user experience on the Web. Used for small games and animation at
first, Flash quickly turned into a popular advertising engine. The excess of annoying
Flash banners deterred many Web surfers and publishers from using the product for some
time. However, the framework soon evolved and eventually became a de-facto standard
for non-HTML interactive multimedia. Currently, Flash supports vector and raster graph-
ics as well as full-duplex audio and video streaming. Although Flash was not truly in-
tended as a 3D platform, a variety of libraries are designed to supplement Flash applica-
tions with quality interactive three-dimensional visualizations.

After Macromedia was acquired by Adobe, Flash technology also became one of the
development environments for Adobe AIR Web 2.0 applications. Notably, Flash is a
nearly ubiquitous Web platform that, according to Adobe’s statistics [3], reaches 99% of
Internet surfers’ desktops. A large number of other Web-enabled devices, such as mobile
phones, now also start supporting Flash applications.

1.3.3 Java 3D
Java 3D is a Java-native API for developing interactive 3D scenes. It was invented in 1998
and has undergone several releases thereafter. Although the technology has not been
widely used in Web applications, Java 3D has extensive visualization and multimedia
processing capabilities that it inherits from its Java layer. Support of object-oriented
paradigm, multithreaded programming, and 3D visual and audio effects are some of the
features offered to Java 3D programmers.

Because Java 3D is not a part of the standard Java Runtime Environment distribution,
most Internet users have to download and install additional libraries to enable the 3D ren-
dering. Another disadvantage of the API is that the Web browsers can only execute Java
code within an applet; and applets tend to not integrate well with HTML pages, both visually and programmatically.

Despite or even because of limited vendor and community support, a number of Java-3D-based Web technologies emerged in the last several years. 3DZZD [4], Strata Live 3D [5], and ARDOR3D [6] are but a few.

1.3.4 Extensible 3D

The X3D language, developed by the WEB3D Consortium [7] as the successor to VRML, is an open-source cross-platform ISO standard for Web-distributed 3D graphics. The X3D architecture embraces the functionality of VRML while offering improved multimedia capabilities. The language is also supplemented with various extensions, supporting Computer-Oriented Design, human animation, non-uniform rational B-splines (NURBS), and geospatial modules. X3D “understands” the classic VRML format and easily consumes the VRML97-based content. Yet one of the main acquisitions of X3D is its support of Extensible Markup Language (XML) encoding. XML makes X3D code more readable and better adapted to the deployment on the Web.

As of this writing, X3D finds more and more use in various professional domains, such as architecture, medicine, mechanical and civil engineering, geospatial navigation and planning, and so forth. A list of projects that employ the X3D architecture is published on the WEB3D Consortium homepage [7]. One particular area of X3D expansion is haptic-enabled visualization systems. For example, SenseGraphics’ H3D API [8] for haptic devices takes advantage of X3D functionality to define virtual environments. The MEDX3D [9] library is designed for medical imaging and volumetric rendering, and is
built on top of H3D. Unfortunately, the absence of any prepackaged UI library is a problem that X3D inherited from VRML.

### 1.3.5 Other Technologies

The list of WEB3D technologies is expanded on a regular basis. The improved computer hardware and growing demands of the Web audience for a rich interactive 3D experience contribute to the formation of new open-source communities and commercial organizations that attempt to deliver a powerful multi-purpose WEB3D standard. Despite the variety of technologies, little success has been achieved in generating a single ubiquitous 3D platform for the Web. Examples of current WEB3D solutions that are not covered in this thesis include 3DMLW [10], Canvas 3D [11], EMMA3D [12], Croquet [13], COLLADA [14], among others.
2 Related Work

2.1 Interface Classification

In this chapter we provide an overview of existing GUI solutions for interactive X3D simulations. Some developers suggest X3D-only implementations, with interface components constructed using the native language definitions—similarly to the library proposed here. Others employ X3D content as a part of composite multimedia environments, backed up with either conventional technologies, such as HTML and JavaScript, or entire proprietary frameworks and APIs. We discuss the flaws and merits of each approach and analyze some of the common issues.

2.2 X3D-Based Interfaces

The most evident and straightforward method to create a GUI in X3D is to interconnect suitable geometric nodes via natively supported scripting extensions. It is, however, difficult to achieve an aesthetically pleasant, functionally rich, and programmatically convenient architecture by dealing with geometric primitives and low-level spatial transformations. For instance, to simulate a button one might need to combine a number of
IndexedFaceSet, IndexedLineSet, and Transform nodes with specific dimensions and material properties, and then superimpose the desired graphics or text on top. To trigger an action on mouse click or imitate the effect of pressing, a TouchSensor and a Script with several functions and field references should be brought into the picture. Every time the button is resized or repositioned, text alignment and border edges might have to be readjusted. That is, even the implementation of such a basic tool as a button might “cost” the developer hundreds of lines of code. (This amount of code would be added for every button with a different size or color, if prototypes were not used.) For this reason, the overwhelming majority of existing X3D-based GUIs merely feature a few basic components that support the minimum required functionality. In what follows, we present several more advanced examples.

Figure 1. Linear accelerator simulator with X3D-based GUI.

In 2006, two X3D-driven Web-based simulation tools for radiation therapy planning procedures were built as a part of 3D Radiation Therapy Treatment (3DRTT) project [15]. The simulators incorporate a set of floating HUD-menus, containing toggle buttons, sliders, and scrolls for manipulating the virtual linear accelerator hardware (figure 1). The
design of the interface follows the traditional approach, described earlier, and only targets one ad-hoc purpose. Despite the simplicity and intuitiveness, this GUI takes up a lot of screen real estate and renders poorly on very high and low zoom levels as well as in stereo mode (because of the false focal distance, as discussed further in section 3.3). Additionally, the menus are prone to visual collisions and can be accidentally moved off the screen view area.

A more systematic methodology is practiced in the CONTIGRA architecture [16], where GUI schemes for the resulting X3D world are defined in XML. The architecture comprises three major levels: SceneGraph, SceneComponent, and Scene. At the SceneGraph level, X3D entities are used to define the geometric components; special grammars are introduced to program the sets of additional nodes for extended behavioral and audio functionality, currently not available in X3D. SceneComponent serves as the markup language of the CONTIGRA architecture and is used to provide interface declarations as well as detailed widget configurations. The Scene level governs the integration procedure and yields executable X3D code. This realization demonstrates powerful abstraction techniques that eliminate bindings to different implementation frameworks, while allowing to write high-level format-independent code. Nonetheless, it is still the developer’s responsibility to create the ultimate building blocks of the GUI, most likely as complementary SceneGraph nodes. Developing such an extension using CONTIGRA architecture would be even more complicated than constructing a GUI library in X3D directly. A similar approach, but with emphasis on XML and Extensible Stylesheet Language Transformations, is presented in [17]. In that case, a sequence of
inter-format translations is performed to analyze, interpret, and merge the model and the scene files into an X3D scene.

Another example of multilayer architecture is explained in [18]. It is proposed to employ the UtiXML [19] UI definition language (UIDL) to describe the scene, which is subsequently converted via VUIToolkit into X3D code for rendering. Fast deployment, however, does not provide high universality and usability of the generated interface. Applicable for individual models, this approach lacks the overall design completeness and integrity.

The common problems of the existing X3D-based GUI implementations include the inability to cover dynamic changes of the interface structure and configuration, necessity to compile each interface individually, questionable rendering quality, and unnatural fusion with the 3D portion of the scene. Utilization of special development tools further reduces the fitness of such solutions for wide use among X3D content creators. Next, we review a series of HTML-based interfaces that overcome many of the presentation- and usability-related shortcomings of X3D-based GUIs.

2.3 HTML-Based Interfaces

To visualize and interact with an X3D world, the user needs special player software. There are currently over a dozen X3D players, with different distribution licenses and levels of X3D component support. Some of these players can be installed both as a stand-alone application and a Web browser plug-in, the latter enabling X3D scenery to blend smoothly into the context of a Web page. With such functionality in place, the interactive tasks of the interface can be effectively delegated to various multimedia ingredients of
the page, including the X3D world, HTML controls, JavaScript scenarios, and so on. Most X3D plug-in manufacturers provide a simple API for runtime access to the geometrical and scripting nodes of the scene via JavaScript code run in the scope of the host page. The backward capability to invoke predefined JavaScript methods from within the X3D environment is often provided as well.

The potential of multimodal HTML-based interfaces for online X3D simulations has been explored in a number of projects developed in the NEWS laboratory [20]. Early versions of one of them, 3DRTT, were discussed in the previous section. The current generation of simulators presented in the 3DRTT project contain a more sophisticated GUI that relies on HTML and Cascading Style Sheets (CSS) for visual representation, and JavaScript for functionality. Control over the virtual environment is ensured by frequent invocations of internal X3D procedures, conducted from the Web page’s JavaScript code as a part of the GUI-event handling.

![Figure 2. Nervous system simulator with HTML-based GUI.](image)

Other HTML-based interfaces for X3D are implemented in the nervous system simulator (figure 2) from the NeuroPathways project, and virtual tensile testing laboratory from
the Virtual Interactive Engineering on the Web project. (Both projects are coordinated by the NEWS laboratory.) The nervous system simulator additionally uses Asynchronous JavaScript and XML (AJAX) technology to dynamically send HyperText Transfer Protocol (HTTP) requests to the server; the server analyzes these requests, queries or updates the database, and replies with an appropriate response. Such architecture is known as AJAX3D, and is designed to combine the benefits of real-time 3D with the power of Web-enabled interfaces. AJAX3D enables the developer to dynamically manage X3D worlds with JavaScript via the Scene Access Interface (SAI)—the analogue of VRML’s EAI.

The Ludos Top project [21] from the Federal University of Uberlândia, Brazil, is another showcase of the AJAX3D paradigm. More specifically, the project demonstrates the feasibility of designing real-time multiplayer 3D online games by expanding one of the existing Web deployment architectures, such as Linux-Apache-MySQL-Python (LAMP), with the X3D module. The AJAX3D schema operating on top of the LAMP software stack, with additional template extensions, also proved effective in the WebScylla [22] project. In this case, an eye-catching HTML-based GUI is combined with realistic X3D animations to allow the user to interactively visualize the colonization of an artificial reef.

Ultimately, the cohort of present-day HTML-based GUIs for X3D simulation systems reveal certain strong sides of WEB3D: interactivity, accessibility, and compatibility with other Web technologies. However, the HTML-based-GUI metaphor entails various limitations on how the X3D content is presented: The visualization can only be interacted with inside the browser. The GUI relies on JavaScript communication, often very intensive, which results in high computation costs and produces delays and jitter in the final visualization. HTML includes only a subset of interface components commonly used in OSs; for
instance, windows, sliders, and tab panels are not normally supported in a Web page. GUIs cannot be easily rendered in stereo mode and are somewhat tied to the screen size and resolution. Elements of the interface usually cannot render above the 3D scene and thus reserve substantial area of the screen. Transparency and visibility are difficult to implement without disrupting the consistency of the GUI. Lastly, sophisticated GUIs with intricate object positioning might be browser-dependent.

A notable effort to consolidate X3D and HTML technologies in the next-generation Web-page-language standard is made in the X3DOM project [23]. The proposed framework is considered for inclusion into the HTML5 specification as a way to support declarative 3D content—defined by X3D nodes—natively in the DOM. This improvement would eliminate the need of a browser plug-in, and enable the 3D scene management via JavaScript operations on DOM tree rather than on SAI (which could also be plug-in-specific).

### 2.4 Other Approaches

While most commonly GUIs for X3D visualizations are implemented internally or in conjunction with HTML, several research teams have decided to invent an entire new UIDL in an attempt to universalize the interface generation. One of them, GUIML, is proposed as an interface markup language for Web3D [24]. The language is described in XML and is comprised of multiple interface components and callback interaction mappings. The authors argue that GUIML has the potential to become a standardized method of GUI abstraction, X3D being a particularly promising 3D interface-instantiation environment. However, the architecture requires a special interpreter to carry out the conversion and
might be better suited for general-case projects. In [25], for example, a very specialized multi-user environment is designed that is unlikely to benefit from the described UIDL. In certain situations it might be more practical to design GUIs that are more applicable in the context of the application, especially if proprietary viewing software is used.

In this chapter we have shown various trends and techniques for creating GUIs in WEB3D applications powered by X3D. We have provided a simple interface classification to point out the virtues and weaknesses of major design approaches; a more elaborate evaluation framework for Web-based visualizations is presented in [26]. In the remainder of this thesis we expound on how our solution targets the problems described, rationalize the utilized design patterns, and work out the details of the implementation.
3.1 Dimensionality

Numerous technical innovations have entered our lives over the last few decades. The means and media of social communication—and information exchange as a whole—have drastically changed. Information is accessed, perceived, and stored in various new configurations and formats. What has remained unchanged for the most part is the 2D nature of content delivery. Maps, newspapers, billboards, Web sites, and TV broadcasting make good examples. Despite living in a 3D space, we are traditionally used to 2D arrangement of information, which provides for effective distribution and interpretation. The intrinsic storage and presentation complexity of physical 3D displays also contribute to this condition. As a result, the vast majority of human-designed interfaces are semantically 2D; that is, having three physical dimensions, including depth, such interfaces are logically confined to their planar equivalents. Some of the customary widgets incorporated in these interfaces include buttons, switchers, sliders, and so on.

The same phenomenon extends to the realm of virtual 3D: Whereas the volumetric representation of the scenery is important, 2D or 2.5D (two dimensions with simulated
depth) implementations are usually more advisable for GUIs [27]. The symbiosis of two- and three-dimensional graphics has proved to be an effective solution for many computer applications, such as video games, modeling software, and various medical, engineering, aeronautic, architectural, and educational simulations.

Practical visualization systems should be able to interact with the user by accepting some sort of human control and generating the proper responses. Generally, the more intuitive and submissive is the UI, the better is the overall operability of the system. In [28] it is shown that the precision of user manipulations could be enhanced by applying simulated surface constraints to the 3D interface of a virtual environment, which once more proves the better fitness of 2D and 2.5D realizations for GUI design.

This principle lays in the foundation of X3DUI. We have attempted to build a library that would preserve the power of X3D to deliver rich volumetric content, while supplying the user with convenient, effective, and—as importantly—conventional tools to control it. Next, we explain the visualization characteristics of X3D and outline how those are applied in X3DUI.

### 3.2 Visualization in X3D

According to the X3D specification [29], “[t]he semantics of X3D describe an abstract functional behavior of time-based interactive 3D multimedia information.” This information is organized in a specific hierarchical structure that conforms to a directed acyclic graph organization, denoted as the scene graph. Typically, the scene graph consists of several groups of components, defining different aspects of the overall appearance and
behavior of the system, such as geometry, lighting, navigation, sensors, scripts, and routes. Let us briefly examine one example of an X3D scene graph.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE X3D PUBLIC "ISO//Web3D//DTD X3D 3.0//EN" "http://www.web3d.org/specifications/x3d-3.0.dtd">
<X3D profile="Immersive">
<head />
<Scene>
  <Viewpoint orientation="0 0 0 0" position="0 0 5" />
  <Background skyColor=".5 .5 .5" />
  <Group>
    <TouchSensor DEF="touchSensor" />
    <Shape>
      <Appearance>
        <Material DEF="material" diffuseColor="1 0 0" />
      </Appearance>
      <Box size="1 1 1" />
    </Shape>
  </Group>
  <Script DEF="script" directOutput="true">
    <field name="material" type="SFNode" accessType="initializeOnly">
      <Material USE="material" />
    </field>
    <field name="boxClicked" type="SFBool" accessType="inputOnly" />
    <![CDATA[
    ecmascript:
      function boxClicked(clicked) {
        if (clicked) {
          if (material.diffuseColor.r == 1) {
            material.diffuseColor = new SFFloat(0, 1, 0);
          } else if (material.diffuseColor.g == 1) {
            material.diffuseColor = new SFFloat(0, 0, 1);
          } else {
            material.diffuseColor = new SFFloat(1, 0, 0);
          }
        }
      }
    ]]>  
  </script>
  <ROUTE fromNode="touchSensor" fromField="isActive" toNode="script" toField="boxClicked" />
</Scene>
</X3D>
```

Listing 1. Sample X3D file listing.
Listing 1 presents the code of a sample XML-encoded X3D scene; the scene contains a box that cyclically changes color from red to green to blue upon receiving mouse touch events (figure 3). Lines 1–3 define the XML encoding and Document Type Definition. Line 5 opens the X3D content, with the node profile specified as Immersive. The head tag in line 6 can optionally contain certain meta data about the virtual world. All of the functional content of the scene is enclosed in the Scene tag, extending from line 8 to line 49. A viewpoint (line 9), or a group of viewpoints, can be declared for further use in the scene. Background is configured in line 11. Lines 16–21 create a shape, whose geometry is a box, and whose appearance is made of a red-colored material. Note that the parameter DEF in line 18 gives the Material node an identifier that is used as a reference within the Script node in line 26. TouchSensor in line 14 is placed in the same group (line 13) with the box to not trigger touching events for anything outside that parent node (provided that there could be more geometric nodes in the scene). Script (lines 24–45) needs to redefine internally any resource that is referenced in one of the functions, as is the case with Material in lines 25–27. Additionally, for every function that is to be referenced outside the script (such as in a route) there has to be a corresponding field included; function boxClicked defined in line 28 is one example. Lines 30–44 demonstrate the use of scripting logic in X3D; multiple methods, operating on previously identified resources, may be declared and invoked, both from inside and outside the Script node. Finally, lines 47–48 illustrate the power of ROUTEs to tunnel events between qualified nodes’ fields, servicing inter-component communication.
To render the X3D scene graph, the hierarchical model needs to be converted into a 3D view; following is the overall picture of such conversion. First, all objects are registered, transformed as needed, and positioned in the global coordinate system. Then the field of view and z-buffer clipping distances are set, resulting in the culling of the hidden geometry. Ultimately, the material properties and lighting are applied to the scene. More intricacies arise with the use of special visual or behavioral components, such as NURBS, interpolations, scripts, routes, and others.

Renderers for X3D have been designed using both primary graphics libraries: OPENGL and DIRECTX. Via OPENGL mappings, in particular, X3D enjoys the benefits of a platform-independent standard. Although the rendering station is expected to demonstrate substantial computational performance, several successful attempts have been made to transport X3D content to mobile phones [30, 31].

Beyond the distinctions in rendering realization, the interpretation of the scene should look and “feel” identically on various platforms. Therefore, in theory X3D authoring comes down to designing the visual content and programming the behaviors in accordance with the language specification. However, X3D player manufacturers rarely support the specification entirely, and sometimes provide proprietary APIs to enable certain visual or behavioral effects, if those are not yet a part of the specification.
We are assured that Contact, by Bitmanagement Software [32], is among the top X3D players, due to its extended capabilities and a convenient interface. In the implementation of our library we use several Contact-embedded X3D components and scripting functions. Although we realize that the employment of a proprietary API may impair the ubiquity of X3DUI, the complementary functions and nodes used are very reasonable candidates for inclusion into the X3D standard, as we argue in the conclusion. Besides, we suggest expanding the library implementation further to accommodate the programming logic to other X3D players.

In the next section we explain how a smooth integration of two- and three-dimensional graphics is accomplished in an X3D scene using both specification-prescribed and proprietary functions as well as conventional and unorthodox visualization techniques.

### 3.3 Presentation Techniques

Despite the seeming easiness, in practice the systematic incorporation of 2D and 3D graphics within one scene is a non-trivial task, especially when dealing with such a crucial ingredient of the visualization as the UI. Because the interface dictates strict accessibility requirements, it is normally visualized in the HUD-layer, in front of everything else. Numerous questions arise regarding the applicability of this technique: Will the UI permanently eclipse the objects in the background? How will avatar orientation changes be treated? How will the interface react to zooming? Could the background scenery partially penetrate the interface controls? These and other questions are thoroughly addressed in the logic of the X3DUI architecture, as explained next.
To accomplish the HUD-like behavior in X3D, a method of routing a ProximitySensor to a Transform node containing the targeted geometry has been adopted: as the user navigates through space, the ProximitySensor detects the viewpoint position and orientation changes, and, using the ROUTE construct, updates the Transform’s translation and rotation fields. The result of this coordination is that visually every child node contained in the Transform appears to remain unaffected by the perspective displacement. Figure 4 illustrates the technique in action: semitransparent rectangle remains in front of the red cube even after the zoom level and avatar orientation change; the normal of the rectangle is still facing the viewer.

Unfortunately, the approach described has several issues, which manifest themselves differently in different X3D players. For instance, when the scene is zoomed in too close or zoomed out too far, the geometry of the HUD-layer might flicker in some players; in others, it will “shake” on any orientation change. Additionally, HUD-objects are penetrated by movable objects that are very near to the viewpoint (figure 5; the red cube is partially penetrating the semitransparent rectangle). Finally, with increasing interest in simulated volumetric visualization, the ability to render a virtual scene in stereo mode becomes very relevant. In case of HUD-based layers the stereo mode produces widely diverged separation for right- and left-eye views, resulting in a bifocal decomposition of
the scene. This phenomenon is stimulated by the false focal distance of the HUD layer, which does not obey to the spatial transformations in the scope of global coordinate system.

As an alternative to the HUD technique, X3DUI library uses the Layer3D node, provided by BitManagement via a prototype, and hence easily transportable to other players (prototyping is explained in the next section). Layer3D allocates a transparent rectangular area of the screen to render an autonomous scene by overlaying it on top of the host scene. This node enables smooth layering and is free of penetration- and stereo-rendering-related problems.

A bigger challenge than merging planar and volumetric geometry in 3D visualization systems, and X3D, in particular, is consistent management of 2D layers in the shared z-plane. Naturally, GUI components placed within one container are rendered at an equal distance to the viewpoint, and nothing prevents their surfaces from interpenetration. This mixture is easily avoided by slightly dispersing the layers along the z-axis. However, in large sets of GUI components, even an insignificant dispersion increment might contribute to an oversized separation range (figure 6), which is very apparent at side-by-side comparison of the affected nodes. If—based on value, order of use, or relationship—certain pieces of the interface have to be readjusted in the depth stack, noticeable visual
permutations are generated. A more appealing, yet even more intricate solution is to instruct the renderer to display in a specific order the 2D items that coincide in z-depth. In analogy to window focus history in many OS interfaces, if window 1 is activated first, window 3 second, and window 2 third, then window 2 will be drawn on top of window 3, which will be drawn on top of window 1. At the same time, the z-coordinate of all three windows remains constant.

Figure 6. GUI depth-separation example.

We program this behavior in X3DUI using OrderedGroup extension node from BitManagement. The node constitutes a simple container, with rendering priority of its children specified in the array-type order parameter. Because sensor nodes in X3D do
not take into account special rendering effects, mere adoption of \texttt{OrderedGroup} is not sufficient to disambiguate the scopes of overlapping sensors. This is why X3DUI also employs the depth-separation technique in cases when accessibility is more important than perfect appearance.

The next essential aspect of utilizing GUI overlays is the occlusion of non-GUI scenery. Possible method to improve the usability is to introduce an appropriate transparency level to the interface components and make them “hideable” or closeable. These effects are readily obtainable in X3D with the aid of \texttt{Material} and \texttt{Switch} nodes combined with simple scripting. Extra degrees of freedom may be provided by making the GUI controls resizable and draggable across—but not beyond—the screen. The cost of such enhanced interactivity is the higher complexity of implementation. For example, it is trivial to use \texttt{PlaneSensor} to allow the translation of objects within a rectangular region of a plane; yet accounting for the dimensions of the manipulated object, and updating the magnitude of translation when X3D player window is resized are more involved procedures. To facilitate the designer’s work, X3DUI library incorporates multiple techniques for efficient use of screen real estate. Resizable, minimizable, and closeable floating windows, support of transparency by all visual components, and control over the size of most objects are to name a few.

![Figure 7. Polygonal (a) and texture-based (b) text rendering in X3D.](image)

One more crucial item in most virtual GUI designs is text. Not only are text-driven interfaces informative, but sometimes essential to the understanding of a control’s function;
even more so if no explicit link exists between the trigger and the event. Nonetheless, text rendering can be more elaborate than rendering of geometric primitives, the reason being a high polygonal count caused by tessellation. To “fill up” the character contours, a great number of varied-size polygons are clustered together in a mesh. While transformations are applied to the text object, every polygon in the mesh is updated with the new transformational matrix, resulting in the visual disruption among adjacent polygons within one character entity (figure 7, left). By supplying the USE_TEXTURE flag in the style attribute of FontStyle node the Contact player can be instructed to render the associated message using a texture, and therefore avoid tessellation-related problems (figure 7).

We have covered several issues of the “2D-in-3D” GUI metaphor that apply to the implementation of X3DUI. More technicalities regarding the individual components of the library are discussed in chapter 4. In the rest of this chapter we expound the mechanism of prototyping in X3D, and point out a number of pitfalls associated with node inheritance.

### 3.4 Prototyping

Prototyping is what truly makes X3D extensible. In X3D authoring, prototypes are a means of constructing new nodes, so that they could be reused in a scene the same way as nodes that are a part of the language specification [33]. Prototype definitions are declared using the ProtoDeclare tag, and prototype objects are instantiated using the ProtoInstance tag. X3D provides an option of storing prototypes in separate files; ExternProtoDeclare tag should be used in such cases. When prototypes are accessed from a single external storage repository instead of multiple discrete locations,
maintaining and deploying a library with many inter-referenced nodes—as is X3DUI—becomes a much easier task.

Table 1. Realization of key OOP concepts in X3D.

<table>
<thead>
<tr>
<th>OOP Concept</th>
<th>X3D Realization</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class</td>
<td>Prototype</td>
</tr>
<tr>
<td>Instance</td>
<td>Prototype Instance</td>
</tr>
<tr>
<td>Method</td>
<td>Field</td>
</tr>
<tr>
<td>Attribute</td>
<td>Field</td>
</tr>
<tr>
<td>Inheritance: Classes inherit and override parent’s methods and attributes.</td>
<td>Inheritance (composition only): Prototypes may build on parent’s exposed fields.</td>
</tr>
<tr>
<td>Encapsulation: Classes hide attributes and methods that should not be accessed directly.</td>
<td>Encapsulation: Prototypes hide fields and methods of a script node.</td>
</tr>
<tr>
<td>Polymorphism: Objects of a child class can be treated as objects of any of the parent-stack classes.</td>
<td>Polymorphism: Only possible if the interface (exposed fields) of prototypes in a parent-child relationship is the same. No casting validation is performed.</td>
</tr>
</tbody>
</table>

On many levels, prototyping in X3D is similar to the object-oriented programming (OOP) paradigm, commonly practiced in numerous programming languages, such as Java, C++, and Python. Table 1 shows the generic X3D incarnation of corresponding OOP concepts. Notably, both methods and attributes are replaced with the concept of a field in X3D. Prototypes can contain multiple fields that each have a unique (in the scope of one prototype) name as well as specific value and access type. Access type defines whether the field will contain a constant value (initializeOnly), may send out new values (outputOnly), may receive new values (inputOnly), or both (inputOutput).
Because methods in X3D can only exist within a script, a prototype that needs to support an interaction unrealizable with ROUTEs, will have to inherit from (that is, include) a Script node. As mentioned earlier in section 3.2, to have access to objects outside the script from the script methods, respective field definitions are required. In a slightly different fashion, Script can also override the fields of the prototype’s interface, thus sending and receiving values on behalf of the component that it is inside of. The only differentiation between a field acting as a method and a field acting as an attribute is whether an identically named function is defined in the scope of the Script.

```xml
<ProtoDeclare name="Sample">
  <ProtoInterface>
    <field name="settings" type="SFNode" accessType="inputOnly" />
    <field name="setup" type="SFBool" accessType="inputOnly" />
    <field name="applyZOrder" type="SFBool" accessType="inputOnly" />
  </ProtoInterface>
  <ProtoBody>
    <Script directOutput="true">
      <field name="CMP_ID" accessType="inputOutput" type="SFInt32" />
      <field name="settings" accessType="inputOnly" type="SFNode" />
      <field name="applyZOrder" accessType="inputOnly" type="SFBool" />
      <IS>
        <connect nodeField="settings" protoField="settings"/>
        <connect nodeField="applyZOrder" protoField="applyZOrder"/>
      </IS>
      <![CDATA[ecmascript:
        function initialize() {
          setAppearance();
        }
        function applyZOrder() {
          // apply z-order
        }
        function setAppearance() {
          // set appearance
        }]]>
    </Script>
  </ProtoBody>
</ProtoDeclare>
```

Listing 2. Script attributes/methods definition example.
Let us consider the structure of the prototype in listing 2. Sample prototype has three inputOnly fields, two of which (lines 4–5) are of type SFBool, and one (line 3) of type SFNode. The body of the prototype contains a scripting node (lines 8–32), which contains three fields (lines 9–11). As indicated in the IS construct (lines 13–16), two of these fields are overriding the prototype’s fields. Additionally, the script holds a function whose name is identical to the name of a field (function applyZOrder, lines 24–26), which makes the corresponding field of the prototype a method rather than an attribute. As an example of encapsulation, the script also contains an internal attribute (line 9) and an internal method (lines 28–30) that can only be referenced from other methods with the same scope (line 21).

A great convenience of X3D prototypes is that they could be nested into each other. That is, X3D authors are not confined to working only with the standard node set at all stages of production. Initially, a prototype is created to accomplish a certain level of abstraction in the global design plan. Subsequently, the original prototype will be delegated a fraction of inheriting prototypes’ tasks, and so on. We make use of this principle in the implementation of X3DUI. To be exact, we repeatedly nest prototypes within each other to branch out the most basic components of the library into dozens of more specialized and better equipped nodes, described in detail in the following chapter.
4 Implementation

4.1 Structure Overview

In this chapter we present the X3DUI library and discuss each of its components individually. First, let us examine the contents and structure of the library. X3DUI presently includes twenty-seven prototypes classified into four categories: system, visual, group, and layout. The system category includes prototypes which organize the work of all widgets and are imperative to the functioning of the entire library; they are Display and Settings. All prototypes that have a visual representation are collected in the visual category; they include Rectangle, Layer, Plane, Label, Button, ControlButton, ToggleButton, TextButton, TextToggleButton, RadioButton, CheckBox, ComboBox, Panel, TabPanel, TextField, HorizontalSlider, Frame, and TaskBar. One additional prototype is HorizontalRunner, but it is auxiliary to HorizontalSlider. The group category holds prototypes that manage the behavior of several nodes in one group, and currently contains one item, RadioButtonGroup. Lastly, the prototypes for laying out elements within a Panel, or Frame container are combined in the layout group,
and consist of LayoutManager, BoxLayout, GridLayout, BorderLayout, and FlowLayout prototypes.

As we briefly covered in table 1, the only inheritance type in X3D is composition, and polymorphism is not supported explicitly. However, the language also allows creating nodes programmatically and therefore inserting them into the scene graph at runtime. It follows that X3D provides three possible forms of bindings between any two prototypes: First is when an ExternProtoDeclare statement is used to notify the interpreter that a specific prototype will be nested inside another prototype by utilizing the ProtoDeclare construct. Second is to assume one of the prototype’s attributes or attribute’s children to be an instance of another prototype; this approach may incur a number of problems if wrong assumptions are made. (The type can be verified via the special getType function, applicable to every internal node or value.) Third is to employ scripting to instantiate a prototype that is already indexed in the scene graph, and attach the resulting object within the host prototype.

Figure 8 illustrates the interrelation among X3DUI components, solid connectors being the first, dashed connectors being the second, and dotted connectors being the third kind of prototype binding, correspondingly. As seen from the diagram, HorizontalSlider has no bindings (except to HorizontalRunner, which is left out of the picture as a supplementary node), since it does not employ—and is not employed in—other components. While also not reflected in the diagram, the Settings prototype is in fact accessed by the majority of other prototypes via type-implicit referencing (second type of binding).
Due to obvious reasons, the most inherited prototype in X3DUI library is \texttt{Rectangle}. Not only many ordinary GUI components have rectangular shape, but they are also better described in 2D space in terms of their width and height. This convention enables easier grouping and packing of interface nodes in higher-level node containers, such as \texttt{Panel} and \texttt{Frame}. In fact, the entire functioning of any layout manager rests upon requesting or determining the dimensions of an element before it can be properly positioned.

\section*{4.2 Core Components}

\subsection*{4.2.1 Display}

\texttt{Display} is the central prototype of X3DUI, for it manages the operation of the entire interface. Implemented as a singleton, in \texttt{children} attribute this prototype encloses an array of the \texttt{Frame}-type objects. All children, along with a \texttt{TaskBar} node, are settled within an instance of \texttt{Layer3D} node, in the body of the prototype; \texttt{Script}, \texttt{MouseSensor}, and several associated \texttt{ROUTE}s follow. Unlike other prototypes,
Display initializes Settings explicitly, in accordance with the first type of binding explained earlier. The primary functions of Display are following: propagation of unique identifiers among GUI nodes participating in the scene graph; window-overlay management; focus management; disambiguation of overlapping touch sensors’ scopes (via intercepting mouse-triggered events); and status synchronization between the windows and the task bar. Let us elaborate on the first two functions in particular.

Every software system with a multitude of constituents typically relies on some identification scheme for the indexing and rapid acquisition of the required items. The X3D language has its own convention, which designates two universal parameters, DEF and USE, for labeling and referring to the nodes, accordingly. However, for objects nested deeply in the hierarchical chain, created dynamically, or controlled from an external prototype, this system fails to provide a dependable way of automating the distribution and registration of unique identifiers. Listing 3 proves the point by demonstrating cases of four-fold nesting in a simplistic GUI, built using X3DUI, and consisting of a single frame with two immediate children and a few more elements altogether. One possible approach to pass every node’s (lines 3–45) identifier to the Display wrapper is by manually providing those nodes with distinct names, cataloging them, and then hard-coding all the names as one of the Display’s input parameters. This procedure still does not take into account the nodes created at runtime, but appears cumbersome and prone to errors even for fairly trivial UIs. Moreover, the inheritance relationships cannot be reflected in a one-dimensional array of identifiers. At the same time, to properly visualize the priority of overlays in listing 3, we must know that, for instance, the TextButton (lines 14–16)
is a child to a Panel (lines 11–21) object; that Panel is placed within a bigger Panel (lines 5–23), which, in its turn, sits inside a Frame (lines 3–45).

Listing 3. A simple GUI programmed with X3DUI.
Our solution to the node identification problem lies in the recursive invocation of overridden functions—namely, setup and applyZOrder—inherited by all visual prototypes from Display, and intended for the runtime component initialization and repainting requests, correspondingly. Since the visualization of X3DUI nodes repeats the nesting order, each parent is held responsible only for its own children. In other words, to broadcast a message, it suffices for the Display to invoke the same method on each of the Frame objects involved in the scene. Thereby the propagation of unique identifiers occurs during object initialization, commenced by the setup function; and reordering of the visual stack takes place by the means of applyZOrder function.

4.2.2 Settings

Settings prototype contains various configurations that define the overall “look-and-feel” of the GUI. A single instance of this prototype is distributed among all visual nodes by Display via the described recursive mechanism, with the exception of passing an attribute instead of calling a function. The configuration fields are declared initializeOnly for encapsulation purposes, and therefore may not be altered after initialization. Setter-methods for Settings's fields are built into Display.

Under the default configuration the appearance of X3DUI is expected to be satisfactory for most users and should enable swift and well-coordinated interactions. If modifications are desired, the authors can fine-tune the settings and evaluate the results locally before publishing the scene. Future contributors to X3DUI will also be able to stylize the library by composing customized themes applied to the graphical components.

One attribute of Settings that deserves special attention is DEBUG, which controls logging in X3DUI. Because error messages displayed by many X3D players, includ-
ing BitManagement Contact, can be very scattered and non-explanatory, identifying the faulty element at the time of debugging becomes a particular burden for the developer. X3DUI integral log statements clarify the GUI initialization sequence and facilitate the search of problematic code snippets.

4.3 Base Visual Components

4.3.2 Rectangle

Rectangle is a container with a small set of basic parameters inherited by most higher-level prototypes. Visually, Rectangle represents a box of a certain size, visibility, color, transparency, and border type; it can host any number of other visual objects. Normally, this prototype should not be instantiated directly, but could be legitimately used as an immediate child to Frame.

4.3.3 Layer

The primary function of Layer is to allow dragging UI objects across the screen. Despite the small field-set, this prototype carries out very critical tasks behind the scenes. For example, Layer ensures that a child remains entirely in the view, remembers its coordinates, and updates its location when the X3D player window is resized. Thanks to Layer, windows can be moved, maximized, and normalized in X3DUI.

4.3.4 Plane

Plane is a descendant of Layer, and is conceived as a “floating” alternative to Rectangle; the method-sets of the two prototypes are identical, apart from the motion-related extension inherited from Layer. Frame is currently the sole descendant of
Plane, partially because Frame is the only permissible root-level visual node type, and hence all its children are movable with the parent and not independently.

4.3.5 Button

The reason that Button and ToggleButton are included in this and not the next section of the thesis is because such generic forms of the button control are rarely used; typically, buttons are supplied with text, as implemented in the TextButton and TextToggleButton prototypes. As a member of X3DUI, Button is the most primitive interactive GUI component, and has only two event-like outputOnly functions: isPressed and isClicked. The first function is called when the button is either being pressed or released, and the second—only upon pressing and then releasing the button. Both functions generate boolean values.

![Button object in unpressed (a) and pressed (b) states.](image)

In terms of animation, the transition between “unpressed” and “pressed” states of a button is achieved by swapping the light and dark edges of its border (figure 9). Because this maneuver is equivalent to inverting the intensity of original border shades, the button appears to be illuminated differently, as if it were “pressed in.” No interactions occur when the button is disabled.

4.3.6 ToggleButton

The main distinction of ToggleButton from Button is the persistence of unpressed and pressed states between user interactions. In other words, once “pressed” by the user, a toggle button will remain pressed until the user outpresses it. When disabled,
the toggle button may still remain in either pressed or unpressed state, although it would not react to the user’s manipulations.

4.3.7 Functionality Overview

In this section we have covered the base visual prototypes, and have shown pieces of their underlying architecture. The base prototypes share a noticeable number of attributes, as presented in table 2. (Although the exposed prototype methods are not incorporated into the table, they largely correlate with the attributes.) However, as prototypes start to deviate into more specialized directions, certain attributes might be dropped, and new ones might be added.

<table>
<thead>
<tr>
<th>Table 2. Comparison of base visual prototype attribute sets.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Rectangle</strong></td>
</tr>
<tr>
<td>settings</td>
</tr>
<tr>
<td>children</td>
</tr>
<tr>
<td>size</td>
</tr>
<tr>
<td>visible</td>
</tr>
<tr>
<td>color</td>
</tr>
<tr>
<td>transparency</td>
</tr>
<tr>
<td>border</td>
</tr>
<tr>
<td>movable</td>
</tr>
<tr>
<td>position</td>
</tr>
<tr>
<td>enabled</td>
</tr>
<tr>
<td>pressed</td>
</tr>
</tbody>
</table>

Next, we proceed to the customary GUI controls available in the X3D UI library. Most of these controls inherit common attributes and methods from one or several of the base
visual prototypes, excluding Layer, which is an intermediate node used by Plane. The coincident attributes are listed at the top of table 2; they include settings, children, size, visible, color, and transparency.

4.4 Conventional Visual Components

4.4.1 TextButton

TextButton is the X3DUI’s implementation of arguably the most traditional GUI control: a rectangular button containing a text label with the summary of the performed function. This prototype extends Button with text-related functionality and one additional graphical state in the animation stack.

![TextButton](a) ![TextButton](b) ![TextButton](c)

Figure 10. TextButton object in unpressed (a), pressed (b), and disabled (c) states.

Figure 10 captures how a text button is drawn while idle (a) and while being pressed (b); if the button is disabled (c), the grayed-out text serves as a visual cue to the user that the control is locked.

4.4.2 TextToggleButton

Analogously to TextButton descending from Button, TextToggleButton is ToggleButton’s child. Because of the visual “sticking” in the pressed mode, this prototype has two additional animation states as compared to TextButton.
Figure 11. `TextToggleButton` object in unpressed (a), pressed unreleased (b), pressed released (c), disabled unpressed (d), and disabled pressed (e) states.

Figure 11 illustrates how a text toggle button behaves when it moves from the unpressed state (a) to the pressed state, before (b) and after (c) the mouse button is released; even while disabled, this control may either be unpressed (d) or pressed (e), according to the semantics of a given interaction scenario. Although the text of the button in figure 11 is the same in both pressed and unpressed states, `TextToggleButton` allows specifying separate messages for each of the two cases.

### 4.4.3 ControlButton

Despite being a typical GUI item, `ControlButton` is accessory to the `Frame` prototype. There is four subtypes of a control button, each with a unique pictogram and a distinct purpose. These subtypes are internally represented with the following self-explanatory flags: `MINIMIZE`, `MAXIMIZE`, `NORMALIZE`, and `CLOSE`. Normally, up to three of these buttons—generally including the minimizing and closing buttons—are placed on the right of a window’s header. Since the maximized and normalized states of the window are codependent, the corresponding control button variants should not be put together, as reflected in figure 12. The window will automatically update the buttons upon the change of its status.
Although \texttt{ControlButton} has the \texttt{enabled} attribute, in the current implementation any control button that is not needed by the parent \texttt{Frame} node is rendered invisible. Whenever a single control button is omitted from a set, the remaining items are still positioned in the original order.

\textbf{4.4.4 CheckBox}

A graphically autonomous component, \texttt{CheckBox} basically matches the functionality of \texttt{TextToggleButton}. However, rather than inheriting from \texttt{Button}, \texttt{CheckBox} inherits from the \texttt{Rectangle} prototype directly. Another distinction from \texttt{TextToggleButton} is that \texttt{CheckBox} does not generate mouse-triggered events per se, such as on pressing or releasing a mouse button; this prototype only notifies the listeners about changing its status from being checked to unchecked, and vice versa.

![Figure 13. CheckBox object in unpressed unchecked (a), pressed unchecked (b), unpressed checked (c), pressed checked (d), disabled unchecked (e), and disabled checked (f) states.](image)

Nevertheless, the primitive purpose does not hinder the check box from having a diversity of visual interactions. Figure 13 defends the point with six animation phases of a check box: the control is neither checked nor pressed on (a); the control is not checked yet, but is being clicked on (mouse button not released) (b); the control has been selected upon receiving a click (mouse button released) (c); the control is still checked, but is be-
ing clicked on (mouse button not released) (d); the control is disabled and not checked (e); and the control is disabled but checked (f).

### 4.4.5 RadioButton and RadioButtonGroup

RadioButton is semantically very close to CheckBox, and even shares similar graphical states, as presented in figure 14, (a) through (f). However, RadioButton can be checked, but cannot be unchecked by clicking on it. In comparison, the check box in figure 13 from state (c) would go to state (d) and then state (a) after being clicked; the radio button in figure 14 from state (c) would go to state (d) and then return to state (c) instead. The justification of such behavior is that radio button objects should normally appear in groups where only one out of several options can be picked at any time, versus the group of check boxes, where each item is independent from the rest.

![RadioButton states](image)

Figure 14. RadioButton object in unpressed unchecked (a), pressed unchecked (b), unpressed checked (c), pressed checked (d), disabled unchecked (e), and disabled checked (f) states.

To provide the single-choice selection feature for related RadioButton instances, we created the RadioButtonGroup prototype. This prototype logically couples several radio buttons by unchecking every item except the one selected last. As a group-type prototype (refer to section 4.1), RadioButtonGroup is instantiated at the Scene level, outside the Display object.
4.4.6 Label

Certain aspects of user-scene interaction often need to be supplemented with visual comments integrated into the GUI. In other cases the realization of an interface component is not inclusive enough to clarify its purpose. An example is a pair of radio buttons, where the first component’s caption says ‘On’, and the other’s says ‘Off’. Without providing further clues or empiric observation, the application of these radio buttons is plain guesswork. Of course, the method of trials and errors might be inapplicable in a military or industrial setting. In any case, a short text message can redress this omission.

In particular, the Label prototype from the X3DUI library is solely dedicated to the in-scene text management. Label supports various text-specific characteristics, such as justification, font size, and font style; although wrapping is not implemented at this time, Label is capable of handling multiline messages. If the preferred width of the label is smaller than some of the lines, the text in those lines will be ellipsized to fit; if impossible, the width will be enlarged to at least accommodate the ellipsis. In the event of vertical overflow, the height of the label is increased to the aggregate height of all lines of text.

4.4.7 TextField

In some situations a keyboard might be better suited for interacting with the UI than a mouse. A typical scenario is when the virtual environment requests a textual response from the user. In our library, this functionality is provided via the TextField prototype. This prototype generates a rectangular field for viewing and editing a string of characters (figure 15); the value of the field can be optionally predefined. TextField recognizes characters typed in lower and upper cases; allows deletion by ‘Backspace’; fin-
ishes editing by ‘Enter’ and ‘Escape’ keys; and supports basic navigation using ‘Home’, ‘End’, and the arrow buttons on the keyboard.

![Figure 15. TextField in plain (a), overflow (b), and disabled (c) states.](image)

While TextField is activated, a blinking cursor indicates the position of the caret (figure 15a). If the width of the field becomes insufficient to display the entire message, only the work section of the string—determined by the location of the cursor—is shown (figure 15b). It is also possible to limit the maximum length of the message by setting the maxLength field. When the node is disabled (figure 15c), it cannot gain focus and will ignore the keyboard input.

TextField is able to interpret key events with the help of KeySensor, which is one of X3D’s standard sensor nodes. KeySensor allows detecting the press- and release-events for regular and action keys as well as their combinations with ‘Shift’, ‘Alt’, and ‘Control’. Although TextField does not currently support text selection, KeySensor makes this feature attainable in the future releases.

### 4.4.8 ComboBox

The ComboBox prototype is our implementation of a common drop-down menu control combined with a text field for editing. Even though ComboBox does not yet support typed input for option selection and submission, most of the remaining functionality is in place.
Figure 16 illustrates a simple user-guided interaction scenario involving a combo box tool: first, no item is selected (a); then the user clicks the blank text area to expand the drop-down (b); after the second option is selected, the control automatically collapses, so the text area now displays the user’s choice (c); and when the user once more expands the list—this time by pressing on the button with an arrow icon—the current selection is highlighted in the list (d).

A peculiarity of the ComboBox control is that the drop-down list has to be drawn over other GUI elements. Not only should the list completely cover the graphics behind it, but should also remain operatable, which is not guaranteed if some other component of the same window has a higher z-proximity to the viewer. The workaround used in X3DUI is to assign to the list area a larger z-elevation, as if it were in a different window hovering on top of everything else. Because any window with an activated component automatically becomes active, every child from an inactive window is already placed farther on the z-axis, and thus the combo box could only interfere with the rest of its parent’s children.
### 4.4.9 HorizontalSlider

Along with ComboBox and TextField, HorizontalSlider is one of the more sophisticated visual non-grouping prototypes in the X3DUI library. Besides the base appearance settings, the configuration of HorizontalSlider is composed of such parameters as minimum, maximum, and selected value; number of mark intervals and their exposure; discrete or continuous selection; and text in the left and right captions. This prototype can operate on negative numbers, work with ascending and descending intervals, and is capable of dynamically detecting and correcting invalid numeric bounds as well as out-of-range selection values. Similar functionality will be programmed into the VerticalSlider, which is not yet a part of X3DUI.

![HorizontalSlider](a) ![HorizontalSlider](b) ![HorizontalSlider](c)

*Figure 17. HorizontalSlider in idle (a), selection (b), and disabled (c) states.*

Figure 17 exhibits a horizontal slider control, first defaulted to the minimum value (a); then during selection with enforced snapping to graduation marks (b); and lastly in the disabled state (c). As mentioned in section 4.1, HorizontalSlider takes advantage of the supplementary HorizontalRunner prototype, which defines the draggable pentagon-shaped runner. Use of highly tailored subcomponents helps us decompose the overall architecture and apply the divide-and-conquer strategy to our implementation.

### 4.4.1 Update Mechanisms

Returning to the discussion started in section 3.4, let us outline the OOP practices and techniques employed in the update mechanisms of the visual nodes described. Every pro-
totype has a collection of specialized setter-methods that are each responsible for a range of related functions dealing with some exterior or behavioral aspect of the node; getter-methods are simulated with accordingly named attributes, whose values are updated within the setter-methods. Since certain parameters are interrelated, updating a single attribute might require updating several others also, which results in a chain of method invocations. Effective at runtime, this logic is inapplicable during the initialization stage, because it produces redundant forwarding sequences, and causes particular setter-methods to be executed before the attribute defaults are overridden with the user-provided values. A simple way to disallow the unwanted propagation is to add a conditional statement that evaluates a boolean-type flag, whose value is flipped immediately after the initialization.

On the other hand, there might be cases when a method needs to be called twice in the process of node instantiation. For example, if the text of a button caption is ellipsized due to the insufficient size of the container, and the original caption has only been one-letter long, then the resultant message would consist of a single ellipsis and measure even wider (“…” versus “a”). To address the increased shortage of space, the button must be resized. At the implementation level, this means that after first calling the `setSize` method and setting the text afterwards, the `setSize` method has to be called once more, to ensure that no further resizing is necessary.
4.5 Container Components

4.5.1 Panel

As in physical interfaces, individual components of virtual GUIs are better pronounced, perceived, and handled when they are arranged into logically cohesive groups. Moreover, coherent organization generally improves the mnemonics of a design. Computer keyboard is a perfect example: buttons with homogenous functions form a number of spatially disjoined blocks; disposition of these blocks is governed by the principles of memorability, usability, and ergonomics. We believe that similar reasoning ought to be embraced in the implementation of visual interface components in X3D.

Panel is the most basic and arguably the most universal grouping container in our library. The Panel prototype can nest multiple X3DUI nodes in a rectangular area with an optional border of lowered, raised, or edging style. The children are positioned according to the specified layout, defaulted to FlowLayout. If the initial area of the panel is too small to fit all elements, it is adjusted to the minimum qualified size.

4.5.2 TabPanel

Economy of space and bent for compactness have driven the developers to create a GUI control that would allow to both cram several sets of smaller widgets into one confined area and also provide unhampered access to them. This is how the tab panel came into existence. At any given time, the tab panel displays the content of the tab that was activated most recently. Selecting a tab is accomplished by clicking on the labeled header at the top of the control.
This mode of operation is threaded into the logic of the TabPanel prototype. Figure 18 shows how a TabPanel object graphically adapts to the alternation of active tabs. Perhaps at even greater extent than in the other widgets, the relief simulation with shaded borders is crucial to visual segregation of active and inactive zones in a tab panel.

4.5.3 Frame

With eighteen attributes and thirty-five methods, Frame is the most complex and multi-functional component of the X3DUI library. Frame derives its name from analogous class in Java programming language, and conceptually implements a window. Besides the work area, a Frame instance can contain a header with the title and a set of control buttons, populated in accordance with the chosen minimization, maximization, and closing flags. A window may be also defined as floating or docked, as well as resizable or static. Children placement and dimension fitting are done with the use of layouts, identically to the Panel prototype.

The Frame widget resides in either active or inactive state (seen in figure 20); activation is the result of any interaction with the widget. However, only a single window can be active simultaneously in an X3DUI-powered virtual desktop. Therefore, by actuating an inactive window, the user causes all other windows to become inactive, as later demonstrated in figure 21.
Resizing mechanisms programmed into the Frame prototype merit special consideration as an unprecedented implementation of that feature in X3D. As a rule, interface components created in X3D lose in usability and operability to their OS-specific counterparts; so, WEB3D authors generally avoid complicating the interface design with additional graphical and behavioral features. In case of X3DUI, the experience of resizing a window demonstrates a high level of intuitiveness and robustness. With respect to technicalities, the resizing operation is a tricky combination of TouchSensor, Switch, and other nodes, backed up with extensive scripting. The sensor is attached to a small transparent padding area adjoining the window’s edges on the inside. Once a touch-event is detected, the default cursor is hidden, and an icon with one of the four “resize” cursors (figure 19)—chosen based on the point of initial contact—is rendered instead. Regardless of scene dimensions and screen resolution, the icon is always scaled to the fixed absolute size, which facilitates navigation on small displays.

![Figure 19. Existing resize cursor variations.](image)

To indicate the prospective area of the window after resizing, a thin textured frame, with edges stretching after the mouse pointer, is visualized during the operation, as presented in figure 20b. After activation the TouchSensor remains engaged only if the pointer aims at “any geometry nodes that are descendants of [its] parent group” [29]. Hence to prevent premature disengagement of the sensor, the attached geometry should move with the pointer. To compensate for trailing effect and jerky motions of bigger than normal amplitude, we temporarily spread out a vast transparent surface in the plane of the
window’s resize margins. This technique keeps the TouchSensor active as long as the mouse button is pressed.

Figure 20. Frame object in inactive (a), resizing (b), and resized (c) states.

Once the size has been updated (figure 20c), all ancillary elements are concealed, and the default system cursor is restored. The updated window size abides by both the global constraints, imposed by the Display’s configuration, and local constraints, imposed by the layout demands.
4.5.4 TaskBar

Task bar is integral component of many virtual desktop environments. Improvement of space efficiency is one purpose that it has in common with a tab panel; only instead of tabs the task bar controls windows. Respectively, in X3DUI the TaskBar prototype is designed to manage Frame prototype instances. Each opened Frame object is represented with a self-titled TextToggleButton control positioned within the TaskBar, in the order of creation.

![Figure 21. A virtual desktop using the TaskBar prototype.](image)

Figure 21 serves as an illustrative example of TaskBar-enabled scene: Three windows, of which one is inactive and one is minimized, are duplicated with a set of desk-bands at the bottom of the screen. The middle desk-band, corresponding to the currently activated window, is pressed down.

Depending on the present status of a Frame object, clicking the respective TextToggleButton can have different effects. If the Frame is minimized or just
inactive, it will become active and will be brought to the front. In case of the already active Frame, the window will be minimized.

4.6 Layouts

4.6.1 LayoutManager

With the prevalence of OOP-derived hierarchical visualization and success of programmatic layout managers, absolute positioning becomes more and more extinct in the modern GUIs. Exploitation of absolute coordinates results in static interfaces that disregard the client’s preferences and capabilities and often require total recasting upon insignificant rearrangements. A properly chosen layout, in contrast, can reduce the designer’s job of organizing and maintaining the GUI to a minimum.

X3DUI is stocked with four popular layout implementations: BorderLayout, BoxLayout, GridLayout, and FlowLayout. Any one of these layouts can be applied towards the content of a Panel or Frame container. Although each layout in the library is represented with a different prototype, those are nothing but templates that are initialized with basic layout parameters, such as vertical or horizontal gap. The actual arrangement of components is performed by the LayoutManager prototype, instantiated inside both Panel and Frame structures, as follows from the inheritance diagram in figure 8 on page 33.

To operate, the LayoutManager requires access to an instance of the preferred layout prototype and children nodes of the affected container. Whenever necessary, the container invokes the LayoutManager’s doLayout function, which forces all content to be rearranged according to the current profile. Based on the layout choice and di-
dimensions of the parent, the children are traversed and individually wrapped into Transform holders with calculated horizontal and vertical offsets. At the end of the doLayout operation the LayoutManager also reports the ascertained minimum width and height of the container, which are used to update—if necessary—the actual size, and properly constrain the future resizes of the container.

4.6.2 BoxLayout

With the use of the BoxLayout prototype, components in a Panel or Frame can be arranged in a single row or column, and additionally aligned vertically to the top, middle, or bottom; and horizontally to the left, center, or right. The ‘Temperature Control’ window from figure 21 shows an example of using a BoxLayout with vertical orientation.

4.6.3 BorderLayout

BorderLayout is employed to place subcomponents in up to five areas: NORTH, SOUTH, WEST, EAST, and CENTER. The unused space is allotted to the CENTER area. This is the only layout that requires the prior knowledge about the area that each child should be assigned to. In practice it implies that every child node is given a unique identifier, which is then served as an input parameter to the layout object. The demonstration can be found in listing 4, which contains the code for the ‘Compass Direction’ window from figure 21. The implementation features five labeled buttons (lines 12–31) embedded into a Frame object (lines 1–34), whose layout attribute nests an instance of BorderLayout (lines 3–9) prototype, with each of the stipulated areas referring to the designated button (lines 4–8).
Listing 4. Example of BorderLayout usage.

When the BorderLayout partitions the container’s surface into several regions, it first compares the available space with the cumulative size of the children’s dimensions. If occlusions are expected, the container has to be enlarged; if, to the contrary, extra space is left, the peripheral components are pushed to the edges, while the center component is placed in the middle of the remaining space.

4.6.4 GridLayout

GridLayout allocates components to individual cells of a grid that contains the requested number of rows and columns. By using the compressHorizontally and
compressVertically flags, the layout can be told to either level all rows in height and all columns in width, or condense each row and column on an individual basis.

![GridLayout Demonstration](image)

Figure 22. A Frame object using the GridLayout.

Figure 22 clarifies how GridLayout handles the situations when the number of children does not equal the number of cells in the grid. In this particular case, five buttons are placed in the first five—counted left to right, top to bottom—cells of a two-by-three grid. If seven or more objects were provided, only the first six would be rendered.

### 4.6.5 FlowLayout

FlowLayout, used by default in every Panel and Frame node, puts elements in rows, “jumping” to a new row every time the remaining horizontal space of the current row is insufficient for the next item. FlowLayout is the most sophisticated of all layouts implemented in X3DUI, because, depending on the current size of the container, the number of components in any given row as well as position of any given component in the row may vary. Additional uncertainty comes from the choice of left-, center-, or right-justification.
An example of dynamic repositioning managed by `FlowLayout` is introduced in figure 23, where the child objects are carried over to the subsequent rows when the width of the parent window is reduced. Resizing of the `Frame` containers that employ `FlowLayout` is handled somewhat differently from those that use other layouts: The minimum width and height that the container can be resized to on a unilateral operation might not agree with the permissible size for a bidimensional operation. In other words, resizing a window horizontally first and vertically second may produce a different result from applying the transformation in both directions simultaneously.

4.7 Deployment

In this chapter we have covered the structure of X3DUI and reviewed each component individually. This section concludes the chapter with a few notes on the deployment process and a discussion of the performance optimization practices used.

In the development environment all twenty-seven prototypes of the library are stored in separate files, located in four directories. Another directory contains the graphical files. While the total size of the graphics is less than 4 KB, and it is not feasible to try merging
them into one file, the source code measures just under 430 KB and can be effectively integrated into a single resource. We have built an automated tool that generates a single file containing all prototypes and devoid of redundant spaces. The tool yields minified X3D code with the size of about 280 KB, which equates to a 35% reduction. More importantly, the scene-loading time is accelerated by fewer file-system requests and prototype-scope sharing, resulting in elimination of external-prototype declarations.

However, when an X3D project is to include the X3DUI library, a separate ExternProtoDeclare statement should be added for every prototype that will be used in the scene; the URL parameter must contain a relative or absolute path to the library file as well as the anchor to the referenced prototype. If the folder with the images is removed from its original location within the library package, the new location has to be specified in the corresponding attribute of the Display prototype. When valid references are observed, X3DUI is readily deployable on the Web as a part of larger X3D visualization systems.
5.1 Summary

This thesis has presented our work on the design and development of X3DUI, a GUI library for the X3D modeling language. We have reviewed and classified several existing approaches to building GUI-enabled X3D-driven visualization systems. We have also identified the main problems of current solutions, and addressed them natively in X3D by using special implementation techniques. The specificity of the language has been considered to ascertain a number of advisable usability-oriented practices further employed in X3DUI. Finally, the organization of the library and essential characteristics of its individual nodes have been discussed.

5.2 Assessment

Although no actual assessment of the X3DUI library has been conducted yet, we are confident that our research and development efforts should be of interest to the WEB3D community. Even in its early stages, X3DUI already evinces the qualities of a promising GUI framework. The major advantages of the library are reflected in table 3.
Table 3. Explanation of positive characteristics of X3DUI.

<table>
<thead>
<tr>
<th>Characteristic</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adaptability</td>
<td>X3DUI dynamically adjusts the GUI appearance to different resolutions and screen sizes, and can be configured with customized visual themes.</td>
</tr>
<tr>
<td>Compatibility</td>
<td>Implemented entirely in X3D, lightweight, and easy-to-deploy, the library is readily integratable into many existing WEB3D solutions, and suitable for both mono- and stereographic imaging.</td>
</tr>
<tr>
<td>Efficiency</td>
<td>Simple geometry, minimal use of texture graphics, and undemanding computation cycles make X3DUI suitable for both desktop and Web-based visualizations.</td>
</tr>
<tr>
<td>Flexibility</td>
<td>Wide functional diversity of the library components allows X3D authors to better tailor the “look-and-feel” of the application to the project-specific needs.</td>
</tr>
<tr>
<td>Neatness</td>
<td>Appealing exterior, smooth rendering, and natural blending of 2D and 3D ingredients enable X3DUI apt for quality production applications.</td>
</tr>
<tr>
<td>Operability</td>
<td>Intuitive navigation across a variety of conventional UI components ensures intelligible and predictable interaction with the scene.</td>
</tr>
<tr>
<td>Readability</td>
<td>Descriptive names are given to all nodes, attributes, and functions in accordance with the OOP naming convention. Tag-based XML nesting improves the clarity of hierarchical dependencies.</td>
</tr>
<tr>
<td>Reusability</td>
<td>The same GUI prototypes can be reused to provide unified look across many applications.</td>
</tr>
</tbody>
</table>

The main limitation of X3DUI is the use of proprietary functions and nodes that are supported by one particular X3D player. Nonetheless, the functionality available through the Bitmanagement Software X3D extension answers the needs of numerous ongoing developments and appears relevant to the modern WEB3D design trends. Hence we believe that the majority of nonstandard elements employed in the realization of X3DUI should
become a part of the language specification. For instance, \texttt{Layer3D} and \texttt{OrderedGroup} nodes are irreplaceable for multi-scene management and dynamic z-depth stack control. The superiority of texture-based text rendering, achievable with \texttt{USE\_TEXTURE} flag, was proved in section 3.3. The ability to request the type, name, and bounding dimensions of an object (accomplished with \texttt{getType}, \texttt{getName}, and \texttt{getBBox} functions, correspondingly) can lead to significant optimizations of scripting and rendering performance. Finally, the \texttt{Browser} object extensions allowing to monitor the current size of the screen (\texttt{windowSize} attribute as well as \texttt{getWindowSizeX} and \texttt{getWindowSizeY} functions); control visibility of the cursor (\texttt{hideCursor} function); programmatically set the navigation mode (\texttt{setNavigationMode} function); and create new nodes at runtime (\texttt{createVrmlFromString} function)—undoubtedly provide better integration with the host OS and enhance to the dynamics of the scene.

\section*{5.3 Future Work}

The development of X3DUl is still in progress and requires substantial revision before a fully functional release is produced. With new features added on a daily basis, even the structure of the library and inheritance linkage among its components might need to be reworked. One of the priority directions for future work is the reduction of recurring code patterns and further delegation of subsidiary tasks to designated prototypes. Another important goal is to tweak the programming logic to recognize different X3D player engines and serve only the player-supported content.
With regards to the component diversity, the library could be expanded with several unimplemented GUI widgets, including text area control, toolbars, file and context menus, dialogs, vertical and horizontal scrolls, lists, icons, and so on. The existing nodes could be supplemented with additional functionality as well. For example, certain designers might want their windows to snap to the sides of the screen, while others would enjoy roll-out panels. Overall interface accessibility could be improved with tabbing navigation and shortcut support.

To minimize the development-to-production overhead, we are building a software tool—similar to Javadoc [34]—to generate user documentation automatically from source code; attribute- and method-specific descriptions are be supplied within XML comment tags. To facilitate the debugging process, the system should respond to invalid input values and runtime exceptions with detailed error messages printed in the console.

Another intriguing idea, inspired by the success of Google Web Toolkit [35], is to build an X3DUI development suite that would translate the GUI written in a modern object-oriented language, such as Java, into X3D code. Because of the higher level of abstraction, the programmer would not be required to have extensive expertise in X3D.
6

References


