A Generic User Interface Framework

Thomas Volk
blaxxun interactive
Elsenheimerstr 61-63
80687 Munich, Germany
+49-89-544 628 63
thomas.volk@blaxxun.de

Frank Althoff and Manfred Lang
Institute for Human-Machine Communication
Technical University of Munich
80290 Munich, Germany
{althoff/lang}@ei.tum.de

Holger Grahn
blaxxun interactive
holger.grahn@blaxxun.de

ABSTRACT

Current VRML browser implementations lack the flexibility of adaptable user interfaces and navigation modes, in massive contradiction to the primary motivation for using 3D, namely to give the user a more natural understanding of how to interact with computer systems and, in return, to make computer work more time-efficient. Multimodal interaction, such as the use of gesture and speech recognition, promises further improvement of the usability of 3D applications but can only be seen in dedicated applications thus far. We present a user interface framework including the definition of a node set and the interfaces to other devices: As a key feature, we propose a DeviceSensor node that allows grabbing arbitrary user input, and a Camera node to realize arbitrary navigation modes. The interface is based on the abstract formalism of a context-free grammar providing the representation of domain- and device-independent multimodal information contents. By changing the grammer the behaviour of the system can easily be modified.

Keywords

UI design, User Interface, HCI, 3D, Navigation

1. HISTORY OF USER INTERFACES

The study of user interfaces (UIs) has long drawn significant attention as an independent research field, especially as a fundamental problem of current computer systems is that due to their growing functionality they are often complex to handle and therefore they adaptation by the user to a high degree. However, those interfaces would be particularly desirable whose handling can be learned in a short time and that can be worked with quickly, easily and, above all, intuitively. Computers should be established as a common tool for everyday people.

In the course of time, the design of user interfaces has undergone some development leading to different generations of interfaces. According to Hennig[1] user interfaces can generally be classified in the the following six categories:

1) Purely physical machine interfaces existed mainly in the vacuum tube era. Interfaces of this generation consisted entirely of fixed mechanical and electromechanical hardware, where reprogramming the machine actually equaled rebuilding it. The user must have profound knowledge of the system architecture as he both had to construct and operate it. These early machines were mostly used by experts for programming complex calculations.

LEAVE BLANK THE LAST 3.81 cm (1.5")
OF THE LEFT COLUMN ON THE FIRST PAGE
FOR THE COPYRIGHT NOTICE

2) Batch systems (zero-dimensional UIs) have been the first entirely software-based method to operate computers. The user must have native programming skills. Typically he designed a job (programming phase), submited the job to the machine (execution phase) and then had to wait for the results. The main drawback of this early approach was the lack of any kind of system feedback, the results of the job were not known until the job had been completely processed.

3) Line-oriented interfaces (one-dimensional UIs) have been the first step toward using the computer as an everyday tool, also for non-programmers. The user communicated with the machine via an alphanumerical terminal. In a classical question-and-answer style, the user was guided by very simple and strictly hierarchically structured dialogs. A disadvantage of this approach is that the user is caught in the dialog scheme and thus is never in direct control of the interaction.

4) Full screen interfaces (two-dimensional UIs) shift the interaction from fixed question-and-answer dialogs to a kind of form-filling user interaction where the user is much more in control how to interact with the system. As he is able to survey a certain amount of upcoming tasks, he can better plan his actions according to his intentions. Developing interfaces of this generation involves defining a set of commands, designing a menu system and providing some kind of screen navigation.

5) Graphical user interfaces (2.5 dimensional UIs) have enhanced the full screen interface, introducing the desktop metaphor [2], which served as a basis for most current desktop systems. Composed of two-dimensional coordinates, the elements of the interface are defined in planar areas which can overlap on the screen due to the current window focus. The interface provides an easy mechanism to freely operate the underlying programs according to the classical WIMP-paradigm (Window-Icon-Menu-Pointing). Besides direct manipulation [3] the interfaces also provide early forms of multimodal man-machine interaction.

6) Virtual reality (VR) interfaces (three dimensional UIs) resemble the highest step in the development of man-machine interfaces. The individual elements of the interface are defined in 3D space containing both planar regions and spacial view volumes. Highly interactive and immersive, they provide the

most intuitive approach to communicate with the machine [4], especially for non-skilled computer users.

Virtual reality user interfaces

Over the last decades, three-dimensional user interfaces as an integral part of VR have undergone some development on their own, too. Characterised by decreasing hardware prices and continously growing computational power, the interfaces have considerably improved from the development of the first HMD by Sutherland[5] in the late sixties through the first immersive flight simulators by Furness[6] up to generic VR toolkits like WorldToolKit[7] in the early nineties.

The Information Visualizer[8] developed at Xerox PARC is an experimental system which explores a 3D user interface paradigm suitable for applications to manipulate large amounts of information. Supporting various display techniques like perspective walls for linear structured information or cone trees for hierachical structured information it provides an intuitive way of accessing documents.

Partly influenced by the Information Visualizer project, Leach[9] took various concepts of the 2.5D desktop metaphor that proved to be working and getting users' acceptance and transfered them into a genuine 3D environment. Like in conventional desktop GUIs, he uses windows, icons, as much as an area to visualize the individual elements and an input device to manipulate the cursor. The novelty of his approach was the introduction of a real 3D window manager and the integration of a 3D cursor controlled by a mouse with six degrees of freedom (6DOF).

The WebBook[10] provides a different kind of information visualisation. Based on a standard WWW-browser, website information filtered according to user-specified criteria is arranged on a virtual book page. Diverse pages are then combined to a book in which the user can read and skim through like in a real book. A further development is the WebForager, an application that embeds the WebBook and other objects in a hierarchical 3D workspace. The information space can contain multiple pages occurring simultaneously or groups of pages in the form of WebBooks, which the user can order and group according to his needs.

Engelmeier[11] et. al introduced a system for the visualization of 3D anatomical data, derived from Magnetic Resonance Imaging (MRI) or Computed Tomography (CT) enabling the physician to navigate through a patient's 3D scans in a virtual environment. They presented an easy to use multimodal human-machine interface combining natural speech input, eye tracking and glove-based hand gesture recognition. Using the implemented interaction modalities they found out that speed and efficiency of the diagnosis could be considerably improved.

In the course of the SGIM project Latoschik[12] et. al developed an interface for interacting with multimedia systems by evaluating both the user's speech and gestural input. The approach is motivated by the idea of overcoming the physical limitations of common computer displays and input devices. Large-screen displays (wall projections, workbenches, caves) enable the user to communicate with the machine in an easier and more intuitive way. The last two projects represent typical examples for the tendency towards applying multimodal strategies in VR-interfaces.

Imposing the highest constraints on hard- and software, virtual reality evolved to a cutting-edge technology integrating information-, telecomunication- and entertainment issues[13]. Research in 3D user interfaces is a highly interdisciplinary task relying on computer and cognitive science, psychology and human factors analysis. VR aims at enabling the average user to intuitively communicate with information systems to easily realize and manipulate complex data content. Therefore the applications of VR cover multiple domains ranging from 3D animation and computer games through scientific visualization up to complete virtual environments.

2. MOTIVATION AND TARGETS

Nowadays VRML-based 3D applications are typically embedded in an HTML frameset since most applications also use HTML to show text-based content. Quite often the user is overwhelmed by the richness of functionality that a web page offers. Besides user interface elements in the 3D window, the user has to cope with others in HTML, naturally these differ considerably from the look and feel of the 3D elements, not to mention all the elements of the desktop system, which leads to an overall decrease of usability. Usability tests underline that fact, revealing that even a 3D walkthrough could be overkill for the end user. On the other side, VRML applications in the CAD/CAM domain and VRML-based games lack more elaborate navigation features. Neither can the browser’s built-in navigation features be adapted to the needs of the application, nor does the 3D author have any flexibility to create a unified UI.

From our experience with commercial 3D applications, we can derive the requirements of a UI toolkit to design customizable UIs and navigation modes. On the low end this toolkit has to deal with a primitive navigation driven by the cursor keys and switching on a light bulb by pressing another key; on the cutting edge it has to cope with complex applications that involve several runtime modules themselves, make use of highly realistic 3D models, and are controlled by speech recognition systems and 6DOF input devices. The GUI of the browser functionality should be an integral part of the HTML frameset or of the 3D window.

Features of the UI-toolkit:

- Allows the implementation of arbitrary navigation modes with VRML
- Easy to use, no need to bother 3D math into depth
- Support for all kind of input devices
- Enables the design of arbitrary UIs

2.1 VRML Navigation Features and Event Handling

Regarding the event handling of a browser implementation we will find a rigid event routing (fig. 1) between the input events like mouse movements and key strokes and the input handling, that realizes the navigation, other UI elements and the VRML sensors, e.g. a mouse drag over a anchor node is handled by the anchor node, outside the anchor the movement serves as the input for the navigation module. The processed events of the sensor nodes fire in return events connected to event consumer fields by ROUTE mechanism. The author has no influence on the event flow and the event handling. Neither he can change the reaction of the navigation module on user interaction nor he can grab events in VRML directly for implementing a 3D UI.

The navigation module exposes only a small set of its functionality by the *Viewpoint* node. Various viewpoints can be defined and activated by binding the viewpoint to the browser. By wrapping viewpoints in *Transform* nodes and applying animations to them, guided tours can be implemented. Even some simple navigation modes can be implemented by catching user input with *TouchSensors* using it as an input for computing a new user position and orientation in a *Script* node. As a drawback no collision detection is performed and a lot of 3D math is involved.

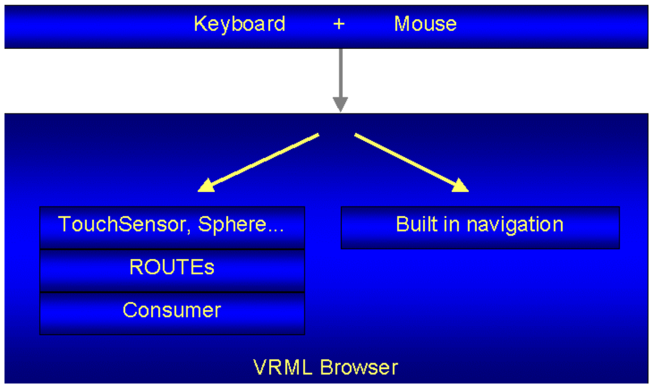

Figure 1: Event flow of common browser implementations

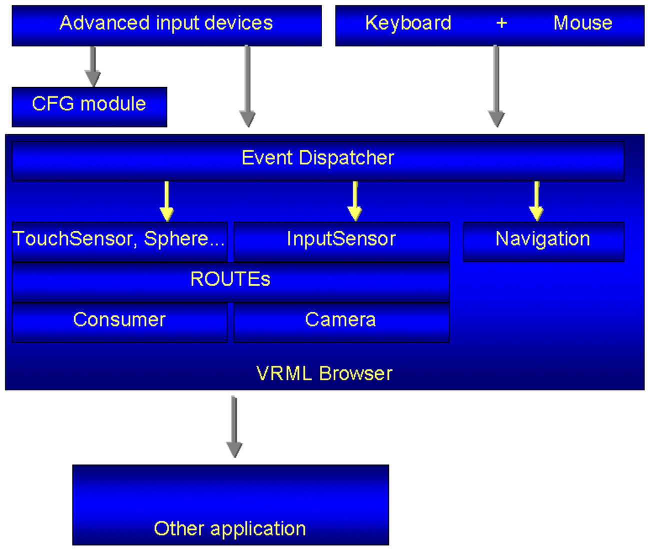

Figure 2: Event flow of blaxxun contact 5.0 with new architecture

In the framework described in this paper we use the mechanism of ROUTEs to break up the rigid event routing (fig. 2). Additional devices can plug in the event dispatcher module of the browser and dispatch their raw input to a *DeviceSensor* node in the VRML scene graph. In addition, a device can trigger certain browser actions by using CFG (context free grammar) module commands as proposed in 4. At VRML scope the newly defined *DeviceSensor* node, a *KeyboardSensor* and a *MouseSensor* give access to all kinds of conceivable user input. The *Camera* node gives superb control over the camera used for rendering: By setting velocity vectors, the VRML author animates the camera. The camera computes the distance vector for the current frame under consideration of previous frame times and the results of the collision detection. Besides a 6DOF navigation mode the node supports an EXAMINE and a BEAMTO mode, thus covering all conceivable navigation modes while hiding the nasty details of 3D math from the VRML content author. However, in the absence of a *Camera,* a default built-in navigation should encompass a minimal navigation feature set allowing all basic navigation actions such as the WALK, SLIDE, PAN, EXAMINE modes.

As a consequence of the flexible event handling any given VRML geometry can be used to implement a user interface. World-specific navigation panels and menus can be a integral part of the scene. A panel can easily be set up with HUDs incorporating *TouchSensor* nodes. Menus can configure the browser by dedicated vrml-script calls to the browser and even drive other applications by using *eventOut*-observers of the EAI.

2.2 Interface to Advanced Input Devices and other Applications

Human beings are able to process several interfering perceptions at a high level of abstraction so that they can meet the demands of the prevailing situation. Most of today's technical systems are incapable of emulating this ability yet, although the information processing of a human being works at a plainly lower throughput than it can be reached in modern network architectures. Therefore many researchers propose to apply multimodal system interfaces, as they provide the user with more natural, expressive power and flexibility. Multimodal operating systems work more steadily than unimodal ones do because they integrate redundant information shared between the individual input modalities. These additional pieces of information are evaluated to enhance the robustness of the system, especially in error-prone environments.

Here we present an approach for handling multimodal information in a VRML browser which can be generated by arbitrary and also multiple input devices. Therefore advanced input devices like data gloves, motion capture systems or even higher-level components like speech and hand/head gesture recognition modules can be plugged into the eventdispatcher in addition to the standard control devices like keyboard and mouse. For interfacing to the browser we are using an abstract communication formalism based on a context-free grammar.

Based on this abstract model we are able to represent domain- and device-independent multimodal information contents. The various events and user interactions are combined in a semantic unification process and thereby mapped on an abstract model of the functionality vocabulary. Thus for example, both natural speech utterances and hand gestures are described in the same formalism. Of course the multimodal information sources have to be integrated in a meaningful way to cope with typical problems like redundant and concurrent information and different running times of the participating recognition components.

As the individual input devices all share the same formalism, it makes no difference to the browser module by which exact input device a specific event has been generated. The browser module just operates on the formal model of the grammar. An additional advantage of this approach is that arbitrary new information sources can easily be integrated into the interface, too.

3. Node Proposal - Extension Nodes for Customizable Input Handling and Navigation

3.1 Camera

```
Camera {
  exposedField   SFString mode „SIXDOF“,
„EXAMINE“, „BEAMTO"
  eventIn       SFBool    set_bind
  eventIn        SFVec3f  xyz             0 0 0
  eventIn        SFVec3f  ypr             0 0 0
  eventIn        SFVec3f  moveTo          0 0 0
  eventIn        SFVec3f  orientTo        0 0 0
  eventIn        SFVec3f  examineCenter   0 0 0
  eventIn        SFInt32  examineRadius   0
  eventIn        SFTime   duration        0
  exposedField   SFVec3f  positionOffset
  exposedField   SFRotation orientationOffset
  exposedField   SFBool     collision   TRUE
  exposedField   SFBool     gravity      TRUE
  exposedField   SFVec3f    down          -1 0 0
  exposedField   SFFloat    fieldOfView   0.785398
  exposedField   SFBool     jump           TRUE
  exposedField   SFRotation orientation    0 0 1 0
  exposedField   SFVec3f    position       0 0 10
  field          SFString   description    ""
  eventOut       SFTime     bindTime
  eventOut       SFBool     isBound
}
```

The ***mode*** specifies the basic mode of the camera, SIXDOF, EXAMINE, or BEAMTO. All standard modes like WALK, SLIDE, PAN, FLY are regarded as a subset of the SIXDOF mode and are realized by applying different values to the ***ypr*** and the ***xyz*** field. The ***xyz*** field specifies the speed in m/s, the *ypr* (yaw, pitch, roll) field is in rad/s. It is under consideration to also introduce an ***acceleration***, but by now we see the gradual increase or decrease of the velocity vectors as a sufficient alternative. The current *position* and *orientation* of the camera are always updated according to the velocity and can also be set even if the velocity vectors are set.

In the EXAMINE mode, the camera is moved on a virtual sphere with the center ***examineCenter*** and the radius ***examineRadius.*** The camera target (look-at point) is not changed, i.e. the viewer moves on the virtual sphere but does not look to the center automatically. To change the camera target see ***orientTo***. The x and y components of the *xyz* field are interpreted as a velocity vector on the virtual shpere.

In the BEAMTO mode the viewer position is animated in ***duration*** seconds to the ***moveTo*** position and the viewing direction is turned to see ***orientTo*** point.

positionOffset and ***orientationOffset*** are additional offsets applied to the current camera position and orientation while maintaining the values of ***position*** and ***orientation***. The ***orientationOffset*** could implement a keyboard-driven look up/down left/right while controling the standard WALK mode with the mouse.

If the ***gravity*** field is set to TRUE, the built-in ground detection is enabled. To detect the ground, a ray hit test is performed with a ray defined by the ***down*** vector. Thus velocity vectors in contrary direction of the down-vector are disregarded. The ***collision***-field switches the collision detection of the browser.

The *Camera* node is derived from the *Viewpoint* node and inherits all *Viewpoint* fields, since its basic functionality is to control the camera, too. For the documentation of the *Viewpoint* fields please refer to the VRML97 specifiction. Like the *Viewpoint*-node, the *Camera* is bindable and is queued in the same stack as the viewpoints. To activate a certain *Camera,* it has to be in front of all other cameras and viewpoint definitions at file scope or has to be bound explicitly. Of course, the bind mechanism can be used to switch between various navigation modes if corresponding configured cameras are present in the scene. If velocity vectors are set, the *position* and *orientation* should also updated if the camera is not bound. If for example a train is chased by various cameras each with its own *fieldOfView* etc. it is essential to have real-time positions in every Camera-node to allow camera changes.

3.2 DeviceSensor

The *DeviceSensor*-node is capable of observing arbitrary input devices such as a six degrees-of-freedom mouse or a speech recognition system. The device data is wrapped in an *Event* node which already covers a considerable amount of possible event types. Special-purpose devices can replace the default implementation with their own *Event* node, guaranteeing maximum flexibility for the support of all possible input devices.

```
DeviceSensor {
  exposedField SFBool enabled TRUE
  exposedField MFString device
"<device>:<subDevice>:<deviceParam>"
  exposedField MFString  eventType
“mouseup“, “mousedown“,...

  eventOut SFNode      event
  eventOut SFBool      isActive
 }
```

The *device* field specifies the hardware device which is observed by the node. A subdevice string allows further refinement of the selection of the device output data. By setting

Device = “MOUSE:LBUTTON”

all events generated by the left mouse button are reported by the DeviceSensor.

<device> ::= MOUSE | KEYBOARD | JOYSTICK | SIXDOF

If the device is MOUSE, the following subdevice values are valid:

<subdevice> ::= LBUTTON | MBUTTON | RBUTTON | MWHEEL

KEYBOARD:

<subdevice> ::= NUMPAD, CURSOR, ALPHANUMERIC

Additionally the ***eventType*** field determines which event types to oberserve. The ***eventType*** accpets multiple type values that are specified in W3C-DOM-Level2 [14] (see Apendix 7.1). For devices not mentioned here a new device name can be created. The node uses this string for identifying the device driver plugged into the event-dispatcher (see 1.1).

3.3 Event

The *Event* node is modeled after the W3C-DOM Events [14].

```
Event {
      eventIn  SFBool     cancelBubble
      eventIn  SFBool     returnValue

      eventOut SFString   type
      eventOut SFVec2f    screen
      eventOut SFVec2f    client
      eventOut SFVec2f    position
      eventOut SFVec3f    xyz
      eventOut SFVec3f    ypr
      eventOut SFBool     altKey
      eventOut SFBool     ctrlKey
      eventOut SFBool     shiftKey
      eventOut SFInt32    keyCode
      eventOut SFString   dataType
      eventOut SFString    data
      eventOut SFInt32     button
}
```

For a detailed description of the fields refer to the DOM-model [14], see also Appendix 7.1. Nevertheless some additional fields can be found in the definition.

For drag-and drop-events the *dataType*- and *data* fields have been added. The *dataType*-field can have following values:

<dataType> ::= File | URL | HTML | Text | Image

The *data* field delivers the URL of the data.

3.4 MouseSensor

Analogous to the *KeyboardSensor* we define a *MouseSensor* that reports the events of the mouse.

```
MouseSensor {
  exposedField SFBool enabled TRUE

  eventOut SFVec2f     client
  eventOut SFVec2f     position

  eventOut SFBool      lButton
  eventOut SFBool      mButton
  eventOut SFBool      rButton
  eventOut SFFLoat     mouseWheel
  eventOut SFBool      isActive
  eventIn  SFBool      returnValue
 }
```

The *client*-field indicates the coordinate at which the event occurred relative to browser's client window. The *position*-field indicates the normalized coordinate at which the event occurred.

lButton, *mButton*, *rButton* events are generated as the buttons are pressed and released.

The *mouseWheel* field indicates the distance rotated. If the wheel is rotated forward the values are positive, in the other case they are negative. The size of the value depends on the resolution of the mouse-wheel. Typical values are mutiples of 120.

If the *returnValue* is set to `false` no default action associated with the event is excuted by the browser.

3.5 KeyboardSensor

As a shortcoming of the VRML200x proposed KeyboardSensor [15] (see Appendix 7.2) the node catches all keyboard events. Even if certain keys that are normally used by the browser are not processed in the scene, the events get lost for the browser. Following the W3C-DOM-proposal [14] we propose to add a *returnValue* field.

4. Interfacing with the Browser

For simplification we assume that the functionality vocabulary of the interface resp. that of the underlying application can be completely described by an grammar formalism. Based on this abstract model we are able to represent domain- and device-independently both high- and low-level multimodal information contents.

A grammar G=(V,T,P,S) consists of non-terminal symbols (V: alphabet of variables and metasymbols representing the structure of the language) and terminal symbols (T: alphabet of valid words). By applying grammatically correct production rules (P) to the variables new elements can be generated. Finally, S is an element of V representing the start symbol for the substitution process. In the special case of a context-free grammar (CFG) which we are using here, the left side of a production rule must contain only a single variable. Thus, ant any given time any variable can be replaced by the ride side of its rule independent of the current context. The rules themselves are not deterministic, i.e. the right sides of the rules normally consist of a sequence of termial and non-terminal elements.

A single word of this grammar corresponds to a single command or an event of the interface. Multiple words form a sentence which denotes a sequence of actions like a whole session. The language defined by the grammar represents the multitude of all potential interactions. A part of a typical context-free grammar we are using in our projects is given below. It is described in BNF (Backus-Naur Form) and demonstrates a small part of the possibile actions in the WALK-mode. By convention, the variables are written in capital letters and the terminal elements in small letters. According to that grammar a valid command of the functionality vocabulary would e.g. be: "walk trans forward".

```
<S>          ::= <SESSION>
<SESSION>    ::= <COMMAND> <SESSION> | quit
<COMMAND>::= <CONTROL> | <WALK> | <FLY> | ...

<WALK>       ::= walk <WSEQ> | walk <SESSION>
<WSEQ>       ::= <W> | <W> <WSEQ>
<W>          ::= trans <ALLSEQ> | rot <LRSEQ>

<ALLSEQ>     ::= <ALL> | <ALL> <ALLSEQ>
<ALL>        ::= <LR> | <FB> | <UD> | <DIAG>
<UD>         ::= up | down
<LR>         ::= left | right
<FB>         ::= forward | backward
```

Messages and events created by the various low- and high-level devices from simple keystrokes to complex natural speech utterances are semantically unified and mapped on the abstract device-independent formalism of the formal grammar. Based on the created information contents, the CFG-module triggers the specific browser functionalities by using the EAI. For navigation, this mainly concentrates on routing the commands to the above described camera node. The concept has proven to be working in a prototypical implementation which can be adapted by the user according to his individual needs.

The key feature of our approach using the CFG-modul as a meta-device interfacing the various input devices to the browser is that it provides a high level access to the functionality of the individual recognition moduls. By changing the underlying context-free grammar the interaction behaviour of the target application can easily be modified also by non-experts without having to deal with the technical specifications of the input devices. The commands of the grammar are comprehensible straight forward as they inherently make sense.

5. Case Studies

To demonstrate the benefit of customizable navigation modes two reference implementations taking advantage of the *Camera* and the *DeviceSensor*-node have been made [16]. The formerly used navigation panel and right-click menu is replaced by VRML-based implementations. With vrml-script calls the VRML-author can get the current status of the browser with the *getOptions*-function, e.g. status of texture smoothing or texture dithering, and set new values with the *setOptions*-function.

5.1 Implementation of a Minimal Navigation

A very intuitive method of navigating in 3D is to animate the camera to the point which has been selected by a mouse click. Because inexperienced 3D users are overwhelmed by a “drag navigation” they try to reach their target by clicking on it. By using the selection test of the browser triggered by a mouse click of the user the target point is determined. This point or a point close to it is used as the *moveTo* and the *orientTo* point of a *Camera* with the BEAMTO-mode activated.

5.2 Advanced Navigation Features

Experienced users prefer to explore their 3D environment using various navigation modes acting with the mouse and the keyboard simultanously. For the implementation of a third person view and a look up/down mechanism we use the *positionOffset* and the *orientationOffset* in the *Camera*. In a mulituser (MU) environment the position of the avatar corresponds to the *position* and *orientation* of the camera, whereas the offsets have no effect on the avatar. Regarding a human body the velocity vectors have effect on the corpus and the head while a *orientationOffset* is only applied to the head. Thus it has to be checked in a further step in how far this movement of the avatar head would fit into the H-Anim specification, and how the head movement could be sent over the network in a MU-environment.

In our sample we use the cursor-keys to control the *orientationOffset* allwoing the user to look left and right, up and down while walking and without switching the mode. Furthermore the used script implements a third person view by adding a *positionOffset*. Thinking of games a car racing could use the velocity vectors to move the car and allow the player to look out of the side window by setting an *orientationOffset*. By setting a *positionOffset* the player could see his car.

6. Conclusion

By tailoring applications to user profiles as shown in our case studies we can enable a broader use of VRML applications. The proof of concept is already given by the implementation of the nodes in blaxxun contact 5.0 and several sample applications. Therefore we propose to add the new nodes to the VRML200x standard.

Working with the *DeviceSensor* and advanced input devices such as the 6DOF mouse we came to the result that even further extensions seem to be desirable:

Although one can use other devices than the mouse for navigation the pointing device used to activate the sensor nodes (TouchSensor etc) is tied to the mouse. To use other input devices the *DeviceSensor* need to drive the pointing device. Either a new node definition, e.g. *PointingDevice*, could process the device input or the browser gets the position values through a vrml-script call, e.g. browser.setCursorPos(SFVec2f position). For real 3D interaction a 3D version of the *PointingDevice* could be driven by a 6DOF input device such as a data glove or a 6DOF mouse. Instead of a ray-hit test a real collision detection between a geometry specified for pointing such as a hand and the sensor geometry would trigger sensors. In both cases mechnanisms have to be found to give the visual feedback for *isOver* events.
Nodes of the MPEG4 standard [17] promise further improvment of the UI design process. Upcoming 2D node implementations are actually more suitable for implementing user interface elements such as a navigation panel or menus. Since the 2D rendering and the 2D user interaction can be handled more efficiently by browser implementation, frequently observed performance bottlenecks will be avoided.
At last promising proposals dealing with input devices in the MPEG4 domain can serve as basis for a common proposal. A *collisionSensor* defined for 3D as well as for 2D allows to test arbitrary geometry for collision with the scene [18]. In contrast to our generic approach of supporting various input devices with the *DeviceSensor* the MPEG4 proposal describes dedicated sensors such as a *GestureSensor* and a *BodySensor*. Even so the definitions are not contradictive since the special purpose sensors could be implemented as PROTOs receiving the data from the generic *DeviceSensor*.

7. Appendix

7.1 W3C-DOM level2

type

The `type` property represents the event name as a string property.

cancelBubble

The `cancelBubble` property is used to control the bubbling phase of event flow. If the property is set to true, the event will cease bubbling at the current level. If the property is set to false, the event will bubble up to its parent. The default value of this property is determined by the event type.

returnValue

If an event is cancellable, the `returnValue` property is checked by the DOM implementation after the event has been processed by its event handlers. If the `returnValue` is false, the DOM implementation does not execute any default actions associated with the event.

Event types

click

The click event occurs when the pointing device button is clicked over an element. This attribute may be used with most elements.

Bubbles: Yes

Cancellable: Yes

Context Info: screen, client, position, altKey, ctrlKey, shiftKey, button

dblclick

The dblclick event occurs when the pointing device button is double-clicked over an element. This attribute may be used with most elements.

Bubbles: Yes

Cancellable: Yes

Context Info: screen, client, position, altKey, ctrlKey, shiftKey, button

mousedown

The mousedown event occurs when the pointing device button is pressed over an element.

Bubbles: Yes

Cancellable: Yes

Context Info: screen, client, position, altKey, ctrlKey, shiftKey, button

mouseup

The mouseup event occurs when the pointing device button is released over an element.

Bubbles: Yes

Cancellable: Yes

Context Info: screen, client, position, altKey, ctrlKey, shiftKey, button

mouseover

The mouseover event occurs when the pointing device is moved onto an element.

Bubbles: Yes

Cancellable: Yes

Context Info:screen, client, position, altKey, ctrlKey, shiftKey

mousemove

The mousemove event occurs when the pointing device is moved while it is over an element.

Bubbles: Yes

Cancellable: No

Context Info:screen, client, position, altKey, ctrlKey, shiftKey

mouseout

The mouseout event occurs when the pointing device is moved away from an element.

Bubbles: Yes

Cancellable: Yes

Context Info: screen, client, position, altKey, ctrlKey, shiftKey, button

keypress

The keypress event occurs when a key is pressed and released.

Bubbles: Yes

Cancellable: Yes

Context Info: keyCode, charCode

keydown

The keydown event occurs when a key is pressed down.

Bubbles: Yes

Cancellable: Yes

Context Info: keyCode, charCode

keyup

The keyup event occurs when a key is released.

Bubbles: Yes

Cancellable: Yes

Context Info: keyCode, charCode

resize

The resize event occurs when a document is resized.

Bubbles: Yes

Cancellable: No

Context Info: None

screen

`screen.x` indicates the horizontal coordinate at which the event occurred relative to the origin of the screen coordinate system. `screen.y` indicates the vertical coordinate at which the event occurred relative to the origin of the **screen coordinate system**.

client

`client.x` indicates the horizontal coordinate at which the event occurred relative to the DOM implementation's **client area**.
`client.y` indicates the vertical coordinate at which the event occurred relative to the DOM implementation's client area.

position

position`.x` indicates the horizontal coordinate at which the event occurred relative to the DOM implementation's **normalized client area**.
position`.y` indicates the vertical coordinate at which the event occurred relative to the DOM implementation's normalized client area.

altKey

`altKey` indicates whether the 'Alt' key was depressed during the firing of the event.

ctrlKey

`ctrlKey` indicates whether the 'Ctrl' key was depressed during the firing of the event.

shiftKey

`shiftKey` indicates whether the shift key was depressed during the firing of the event.

keyCode

The value of `keyCode` holds the virtual key code value of the key which was depressed if the event is a key event. Otherwise, the value is zero. Currently the raw Win32 keycode is reported.

7.2 Nodes of other proposals

7.38 KeySensor

```
interface KeySensor : KeydeviceSensorNode {
  attribute SFInt32 keyPress
  attribute SFInt32 keyRelease
  attribute SFInt32 actionKeyPress
  attribute SFInt32 actionKeyRelease
  attribute SFBool shiftKey_changed
  attribute SFBool controlKey_changed
  attribute SFBool altKey_changed
  attribute SFBool isActive
}
```

A KeySensor node generates events when the user presses keys on the keyboard. The KeySensor supports the notion of "keyboard focus"; if there are multiple KeyboardSensors and/or StringSensors in a world, only one will generate events at any given time.

keyPress and *keyRelease* events are generated as keys which produce characters are pressed and released on the keyboard. The value of these events is an integer which is the UTF-8 character value for the key pressed. The set of UTF-8 characters that can be generated will vary between different keyboards and different implementations.

actionKeyPress and *actionKeyRelease* events are generated as 'action' keys are pressed and released on the keyboard. The value of these events are in Table 7.9.

shiftKey_changed, *controlKey_changed*, and *altKey_changed* events are generated as the shift, alt and control keys on the keyboard are pressed and released. Their value is TRUE when the key is pressed and FALSE when the key is released.

The KeySensor is activated when it receives an *isActive* event. Only the most recent KeySensor or StringSensor to receive an isActive event can receive input from the keydevice. Thus, all other KeySensors and StringSensors become inactive until the most recently activated KeySensor has its *isActive* field set to FALSE.

The KeySensor is not affected by its position in the transformation hierarchy.

7.66 StringSensor

```
interface StringSensor : KeydeviceSensorNode {
  attribute SFString enteredText;        # initial
value: ""
  attribute SFString finalText;          # initial
value: ""
  attribute SFString terminationText;    # initial
value: "\r" (carriage return character)
  attribute SFInt32  deletionCharacter;  # initial
value: "\b" (backspace character)
  attribute SFBool   isActive;           # initial
value: FALSE
}
```

A StringSensor node generates events as the user presses keys on the keyboard. The StringSensor supports the notion of "keyboard focus"; if there are multiple StringSensors or KeySensors in a world, only one will generate events at any given time.

enteredText events are generated as keys which produce characters are pressed on the keyboard. The value of this event is the UTF-8 string entered including the latest character struck. The set of UTF-8 characters that can be generated will vary between different keyboards and different implementations. If a *deletionCharacter* is entered, the previously entered character in the *enteredText* is removed. The *deletionCharacter* field contains the integer representation of one UTF-8 character. It may be a control character. If the deletionCharacter is 0, no deletion operation is provided.

The *finalText* event is generated whenever a sequence of keystrokes are recognized which match the keys in the *terminationText* string. When this recognition occurs, the *enteredText* is moved to the *finalText* and the *enteredText* is set to the empty string. This causes both a *finalText* event and an *enteredText* event to be generated.

The StringSensor is activated when it receives an *isActive* event. Only the most recent StringSensor or KeySensor to receive an isActive event can receive input from the keydevice. Thus, all other StringSensors or KeySensors become inactive until the most recently activated StringSensor or KeySensor has its *isActive* field set to FALSE.

The StringSensor is not affected by its position in the transformation hierarchy.

CollisionSensor2D

The CollisionSensor2D allows to detect the collision between a node reactiveNode which is a 2D Shape, and the group of objects in which the CollisionSensor2D is inserted. Its use in a scene is explained in Figure 3.

The BIFS proposed syntax for the CollisionSensor2D is described as follows :

```
CollisionSensor2D {
ExposedField        SFBool              enable
ExposedField        SFShape2DNode
          reactiveNode        null
EventOut            SFBool              isOver
EventOut            SFBool              isInside
EventOut            SFBool              isOutside
EventOut            SFTime              EventTime
EventOut            SFFloat             inCoef
          }
```

The coefficient inCoef gives additional information about the collision state. It corresponds to the ratio of the surface of the reactive node which is over the group of sensitive objects. Its value can be computed e.g. as ratio of overlapping pixels over total pixel number. The following table gives practical examples (Figure 4).

The node CollisionSensor2D is not directly bound to a device, as is the case of the other sensors described in this paper. However, the denomination « sensor » seems appropriate for many reasons. Various parameters in the reactive node can be animated with BIFS-Anim. E.g. position, rotation and scale could be animated by a stream coming from another user's terminal. It is also possible to put as reactive node a bitmap displaying a raw video which is segmented in real-time and inserted in the terminal directly. This last configuration is illustrated in Figure 5 showing how the image of a person, segmented in real time and inserted in a scene, can become a « actor » of a scene through a CollisionSensor2D.

CollisionSensor

The node CollisionSensor is strictly equivalent to the previous 2D version, apart from its 3D nature. Here is the syntax proposed for the CollisionSensor :

```
CollisionSensor {
ExposedField        SFBool              enable    TRUE
ExposedField        SFShape3DNode
          reactiveNode
EventOut            SFBool              isOver
EventOut            SFBool              isInside
EventOut            SFBool              isOutside
EventOut            SFTime              EventTime
EventOut            SFFloat             inCoef
       }
```

8. REFERENCES

[1] A. Hennig, Die andere Wirklichkeit, Addison Wesley, 1.Auflage, 1997

[2] D. Smith, C. Irby, R. Kimball, and B. Verplank. Designing the star user interface. Byte, pages 242-282, April 1982.

[3] B. Shneiderman, “Direct manipulation: A step beyond programming languages”, IEEE Computer, 16(8):57-69, August 1983

[4] M. Green, C. Shaw, R. Pausch, “Virtual Reality and Highly Interactive Three Dimensional User Interafces”, CHI 92 Tutorial, ACM Conference on Human Factors in Computing Systems, Mai 92 Monterey, ACM Press 92

[5] I. Sutherland, “The Ultimate Display”, Proceedings of the Fall Joint Computer Conference, Vol 33, 757-764, 1968

[6] T. Furness, D. Kocian, "Putting Humans into Virtual Space", Proceedings of Society for Computer Simulation, Aerospace Conference, Jan. 1986.

[7] WorldToolKit – An advanced cross-platform development environment for high performance real-time 3D graphics applications”, http://www.sense8.com/products/index.html

[8] G. R. Robertson, S. K. Card, and J. D. Mackinlay, “Information visualisation using 3D interactive animation”, Communications of the ACM, 36(4):57-71, 1993.

[9] G. Leach, G. Al-Qaimari, M. Grieve, N. Jinks and C. McKay, “Elements of the Graphical User Interface”, Proc. of INTERACT'97, Sydney, Australia, July 1997

[10] S. K. Card, G. G. Robertson, and W. York, “The webbook and the web forager: An information workspace for theworld-wide web”, CHI 96, pages 111-117, April 1996.

[11] K.-H. Engelmeier, C. Krapichler, M. Haubner, M. Seemann and M. Reiser, “Virtual reality and multimedia human-computer interaction in medicine“, IEEE Workshop on Multimedia Signal Processing, Los Angeles, Dec. 1998

[12] M. E. Latoschik, B. Jung & I. Wachsmuth: Multimodale Interaktion mit einem System zur Virtuellen Konstruktion. In K. Beiersdörfer, G. Engels & W. Schäfer (Hrsg.):Informatik '99, 29. Jahrestagung der Gesellschaft für Informatik, Paderborn, 5.10 - 9.10. 1999, 88-97.

[13] H-J. Bullinger, O. Riedel, A.Rössler, “Virtual Reality as a Focal Point between New Media and Telecommunication”, VR World ´95 – Conference Documentation, IDG 1995

[14] Document Object Model (DOM) Level 2 Specification, http://www.w3.org/TR/1999/WD-DOM-Level-2-19990304/events.html.

[15] Draft VRML 200x-X3D Specification, September 2000. http://www.web3d.org/TaskGroups/x3d/specification/index.html

[16] Developer Site of blaxxun interactive, http://www.blaxxun.com/developer/contact/3d/

[17] MPEG-4 Systems Final Committee Draft : http://garuda.imag.fr/MPEG4/syssite/syspub/docs/w2201.zip

[18] Marc Brelot, Jean-Claude Dufourd, Ideas for new BIFS sensors and applications, M5340