A Generic Approach for Interfacing VRML Browsers to Various Input Devices and Creating Customizable 3D Applications

Frank Althoff*, Herbert Stocker+, Gregor McGlaun* and Manfred K. Lang*

*Institute for Human-Machine-Communication, Technical University of Munich, Arcisstr. 16, 80290 Munich, Germany

+blaxxun interactive, Elsenheimerstr. 61-63, 80687 Munich, Germany

althoff@ei.tum.de, herbert@blaxxun.de, mcglaun@ei.tum.de, lang@ei.tum.de

ABSTRACT

In this work we present a generic architecture for interfacing various input devices to VRML browsers. Concentrating on the aspect of navigation, our system supports the full range of potential input devices from conventional haptic devices like keyboard and mouse over special Virtual-Reality devices like spacemouse and joystick to, as a special feature, semantically higher level input like speech and gesture recognition. The communication between the individual components of the system is based on a context free grammar, allowing abstract modeling of the various devices and handling both discrete and continuous navigation information. Two new node extensions support the VRML author in creating highly customizable 3D applications: The DeviceSensor node allows grabbing arbitrary user input in a systematic way and the Camera node gives full control over the scene view by specifying velocity vectors and thus enabling arbitrary navigation modes. Finally, the proof of concept is given by a prototypical implementation in VRML.

Categories and Subject Descriptors

H.5.1 [**Information Interfaces and Presentation**]: Multimedia Information Systems; I.3.6 [**Computer Graphics**]: Methodology and Techniques

1. INTRODUCTION

In the course of time the development of user interfaces (UIs) has become a significant factor in the software design process, as growing functional complexity, restriction to mostly haptic interaction and extensive learning periods lead to increased user frustration. Therefore, various interface types and interaction paradigms have been introduced[10]. Virtual-Reality (VR) interfaces currently resemble the latest step in the development of man-machine interfaces, providing a highly intuitive system approach. Particularly interesting for the average computer user, 3D interfaces can be worked with effectively and intuitively.

A fundamental task in designing VR systems is handling the problem of orientation, navigation and manipulation tasks in 3D space. In this way, our work especially concentrates on the aspect of navigating in arbitrary VRML worlds. The framework includes mechanisms for implementing various navigation modes based on the standard modes WALK, FLY and EXAMINE. From our experience with 3D applications we can derive the requirements of a UI framework to design customizable UIs. On the low end, this toolkit has to deal with primitive browser-intrinsic navigation features and on the cutting edge, it must be able to handle the integration and coordination of several complex applications.

In general, multimodal system access provides the user with more naturalness and flexibility, and it makes the system working more error-robust (e.g. see [11][6]). Both experts and normal computer users highly enjoy having the possibility to freely choose among multiple input devices and work with the system according to their personal interaction styles as found out in [1]. Therefore, our system supports a wide range of potential input devices in a generic way, facilitating interaction by both conventional haptic devices like mouse and keyboard and special VR hardware like joystick and spacemouse, and, as a special feature, semantically higher level modalities (SHM) like natural speech and dynamic hand and head gestures. Finally, the mechanisms introduced in this work are easy to use without having to deal with 3D math into depth.

2. SYSTEM ARCHITECTURE

The framework that we describe in this paper defines a set of modules that allow the user to freely select amongst various ways of communicating with the VRML browser. Although we primarily concentrate on the navigation task, our framework can be extended to other interaction paradigms. An overview of the underlying system architecture is given in figure 1. A series of input modules interprets input of the semantically higher level modalities and manipulations of the haptic devices. The discrete integrator handles the commands received from the various input modules and resolves

Web3D'02, February 24-28, 2002, Tempe, Arizona, USA.

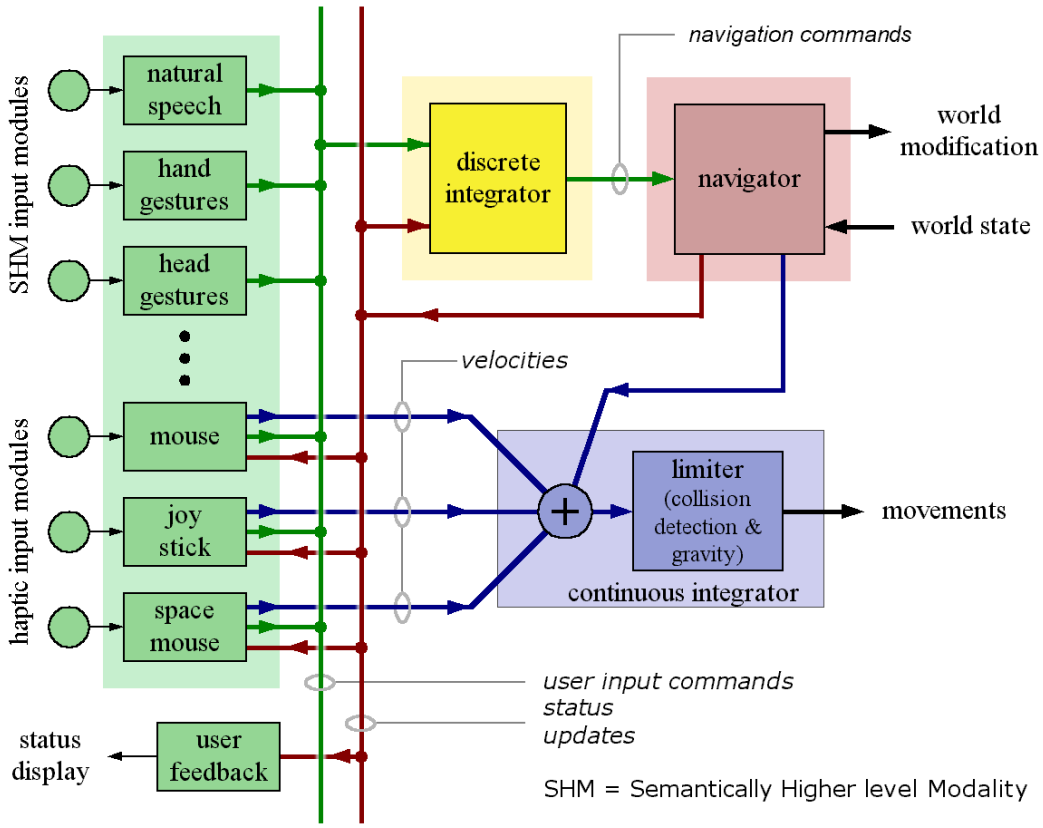

Figure 1: Overview of the system architecture for integrating user input

redundant, complementary and competing multimodal information contents. Converted to navigation commands, the information is transmitted to the navigator. Besides executing these commands, the navigator unit maintains the individual parameters that characterize navigation and browser status and informs both the input modules and the discrete integrator of any changes. Finally, the continuous integrator sums up the speed values of all inputs and passes the sum to the limiter that performs collision detection and gravity simulation. If the system is to be extended to an additional new input device, this can easily be done by adding a new input module to the framework. In selective cases, the logic of the discrete integrator component may additionally be subject to some changes to take into account the specific features of the new input source.

2.1 Input Modules

In general, the input modules can be divided into two groups: On the one side, the SHM modules interpret semantically higher level input like the output of speech and gesture recognition units. One the other side, the haptic modules represent the front-end for both conventional haptic input devices like mouse and keyboard and special VR hardware like spacemouse and joystick.

SHM modules typically do not send velocity vectors to the continuous navigator, but instead generate commands that represent, as the name indicates, semantically higher level user intentions. In most cases, these commands are incomplete and either have to be interpreted in the navigation context of previous commands or rely on additional information provided by another modality. An SHM module does not receive and evaluate status updates. Interpreting this information is directly handled in the discrete integrator. Details on SHM modules are given in [2], [12] and [9].

Haptic modules, in turn, typically generate velocity vectors that describe navigation movements. While SHM modules mainly output abstract user expressions, haptic modules completely interpret manipulations of the input devices: The binding of device manipulations to velocities depends on the current state of the navigator, which is mainly characterized by the navigation mode and speed. The haptic modules receive status updates from the navigator and adapt the interpretation process to these changes. In EXAMINE mode, the modules primarily generate a rotational vector, whereas in WALK/FLY mode a translatory vector with a mostly forward oriented direction is generated. The magnitude of the movements depends on the instantaneous navigation speed.

Besides sending velocities to the continuous integrator, haptic modules can send the full spectrum of available commands to the discrete integrator component. Thus, they can issue mode and stepsize changes, as well as initiating discrete movements, which are typical for a key pad device. The status information contained in the navigator unit can be used externally, too. Sending it back to the associated devices offers an interesting opportunity of providing direct system feedback to the user. A possible application could be the implementation of a force feedback effect that is initiated when a collision with geometry is detected. Moreover, the navigation mode could be shown on the screen or announced via the loudspeaker system.

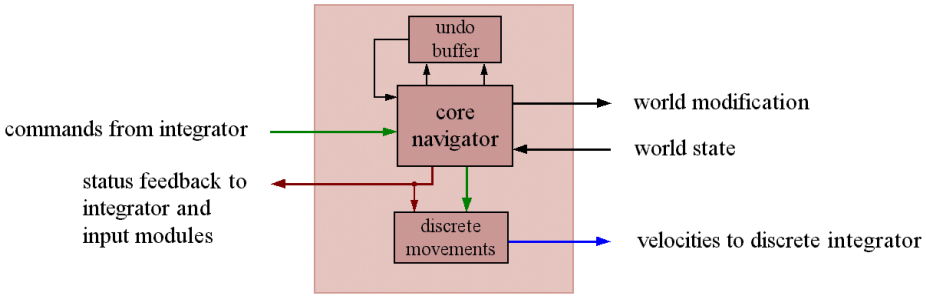

Figure 2: Structure of the navigator component

2.2 Discrete Integrator

The discrete integrator module resolves and combines the incoming multimodal information stream of the various SHM and haptic modules as well as the status updates provided by the navigator component. As a result of this integration process, a set of browser commands is generated that can directly be executed by the navigator. A context free grammar (CFG) serves as the communication basis for coordinating the individual information sources. By applying specific look-up tables, both the output of the individual input modules and the status updates are transformed to the grammar formalism. For example, the semantic meaning of a recognized word chain is mapped on the appropriate CFG elements by a separate unit, interpreting the results of the word processor.

To integrate the pieces of multimodal information in real-time, we mainly use a classical straight-forward, rule-based approach, that is well established in various research applications (e.g. [5][8][13]). Thereby, the semantic unification process is carried out with reference to our CFG, i.e. taking into account context sensitive timing windows, individual grammar elements are combined in a meaningful way to correspond to the users primary intention. Additionally, we are experimenting with statistical methods based on evolutionary computation. While the communication formalism is extensively discussed in section 3, the precise details of the integration algorithm are described in [2].

2.3 Navigator

The navigator component receives the resolved commands from the discrete integrator and executes them. It performs actions like viewpoint animation or step based movements. Additionally, this unit maintains diverse status information like the currently active navigation mode, the navigation speed, the position in the viewpoint stack or potential collision events. Each time, one of these status flags changes, the navigator notifies both the haptic modules and the discrete integrator of these events.

To allow for an undo command, the navigator provides an n-stage **undo buffer**. Before executing a command, it saves the current state (the current viewpoint position for movement commands, the navigation mode for mode switching commands, etc.) to the buffer. When the undo command is received, the last stored state is read in LIFO order and properly restored.

For movement commands, the navigator contains a submodule, entitled **discrete movements**, that generates velocity vectors and sends them to the continuous integrator. For step based commands, it generates the velocities only for a limited amount of time, but with a magnitude that is proportional to the navigation speed. For continuous commands these movements are generated until the stop command or another, conflicting command is received. A detailed view of the navigator unit is shown in figure 2.

2.4 Continuous Integrator

The continuous integrator module consists of two submodules. The adder sums up the velocities that it receives from the haptic modules. Afterwards, this sum is passed to the limiter. To restrict user movements with regard to the given world geometry, this unit performs collision detection and gravity simulation. Before routed to the limiter, the individual velocity components are combined, resulting in a mathematically precise notation of movements in 3D that are entirely described by the six degrees of freedom. To implement restricted navigation modes, several of these degrees can be blocked and then the adder only performs the summation in the allowed directions.

3. COMMUNICATION FORMALISM

For communicating information between the individual modules of the system, we make use of an adapted and massively extended context-free grammar that is based on a formalism first introduced in [3]. Completely describing the functionality vocabulary of the browser, the grammar model facilitates the abstract representation of domain- and device independent multimodal information contents. Thus for example, both natural speech utterances, hand gestures and spacemouse interaction can be described in the same formalism. In a classical client-server approach pieces of multimodal information are exchanged in the form of simple string messages over TCP/IP sockets. Each of these messages is prefixed by two unique identifiers. While the first ID specifies the target destination (e.g. a certain integrator or application module), the second ID indicates the message source, i.e. the device, modality or internal system module, the command comes from. The formal language defined by the grammar represents the multitude of all theoretically possible system interactions. A word of the grammar corresponds to a command, subcommand or a status message of the interface. Multiple words form a sentence describing a sequence of events and actions. The initial sequences of the grammar are given in Backus-Naur Form (BNF) below.

```
<S>    ::= <CSEQ>
<CSEQ> ::= <CMD> | <CMD> <CSEQ> | <>
<CMD>  ::= <MOV> | <POS> | <CTRL>
```

The class of user input commands that are communicated between the input modules and the discrete integrator denote the users intention. While the CFG model is mainly used for transferring discrete navigation commands, the continuous velocities, resulting from haptic interactions, are a set of periodically updated floating point values, one for each degree of freedom. The set of user input commands can further be divided in various command clusters. In general, three major information blocks can be identified. Movement commands (`MOV`) indicate direct navigation information, position oriented commands (`POS`) denote movements to specific locations and finally control commands (`CTRL`) describe changes of the browser parameters and the feedback flow. The grammar blocks are discussed in more detail.

3.1 Movement Commands

The group of movement commands describes navigation information, resulting in movements of the scene view in the following form: <`mode`> <`type`> <`direction`> and can optionally be prefixed with start. The <mode> slot denotes the specific navigation mode, in which the movement should be performed. This mode is sent to the haptic modules which interpret future manipulations on their devices in this mode. Without the `start` prefix, the commands express a movement in form of a single step. The length of such a step is controlled by a set of `control stepsize` commands. If the `start` prefix is present, the movement in the currently given direction infinitely continues until the `stop` command or a command denoting another movement is received. Some of the continuous commands (e.g. `start walk trans forward`) are predestinated to be used simultaneously in combination with haptic devices. In this case, the haptic devices would then change the direction of movements. The following BNF represents the set of potential navigation movements.

```
<MOV>  ::= start <MCMD> | <MCMD> | stop
<MCMD> ::= walk <WLK> | fly <FLY> | examine <EXM>
<WLK>  ::= trans <ADIR> | rot <LR>
<FLY>  ::= trans <ADIR> | rot <LRUD> | roll <LR>
<EXM>  ::= trans <MDIR> | rot <LRUD> | roll <LR>
<ADIR> ::= <MDIR> | <DIAG>
<MDIR> ::= left | right | up | down |
           forward | backward
<DIAG> ::= lfwd | rfwd | lbwd | rbwd
<LRUD> ::= left | right | up | down
<LR>   ::= left | right
```

Sometimes, the CFG transcription of an isolated user articulation can be ambiguous, representing an incomplete command that does not correspond to the exact syntax of the formal grammar model. To cope with this problem, the discrete integrator unit also accepts subsets of the grammar and combines them with context information of previous commands. Thus, for example, a `rot left` or even only a single direction can be interpreted by the discrete integrator component.

3.2 Position Oriented Commands

Purely position oriented commands denote movements to specific positions. An overview of the corresponding BNF notation is given in figure 3. In that syntax, "f" is used as a short form for <float> and "#" denotes comments, shortly explaining the meaning of the corresponding grammar entry. Some of the position oriented commands (`gothere, lookat` and `exacenter`) need an additional, `indicated` command to be interpreted as a valid navigation command. Thereby, indicated commands express e.g. gestures that convey some relevant position and/or orientation in 3D space. A special set of commands (`orientto, moveto` and `beamto`) express the users wish to be oriented or moved to a specified position. Orientations can either exactly be described by the axis of rotation and the rotation angle (like transformations in VRML) or by specifying a point in 3D space where the optical axis of the scene camera is to be rotated to (offering one additional degree of freedom around that axis that has to be handled explicitely). Moreover, the grammar provides a `beamto nearpos` command which moves the user close to a given position. Thereby, the interpretation of the exact location can be handled according to currently given geometrical constraints. This command may be equivalent to a `gothere` (provided by an SHM input) and `indicated geometry` (given by a haptic module). The effect is that the scene view is changed in a way, that the user looks at the specified position and the avatar is moved to a location nearby, taking into account the geometrical constraints of the surrounding area.

3.3 Control Commands

In addition to the individual movement commands, the grammar contains a section for describing and manipulating the various status parameters of the VRML browser, e.g. the handling of lighting conditions, viewpoints, collision detection and gravity flags as well as changing the basic navigation modes. A detailed description is given by the following BNF.

```
<CTRL> ::= ctrl <CCMD> | repeat | undo | quit
<CCMD> ::= mode <SMOD> | stepsize <SPS>  |
           viewpoint <VIP> | sendstatus <SVAR>
<CCMD> ::= light <OOT> | collision <OOT> |
           gravity <OOT>  | straighten | balance
<SMOD> ::= set <MVAR> |
           restrict <x y z yaw pitch roll>
<MVAR> ::= walk | fly | examine
<SPS>  ::= inc | dec | reset | set <float>
<VIP>  ::= prev | next | reset | set <float>
<OOT>  ::= on | off | toggle
<SVAR> ::= all | sstepsize | smode | slight |
           scollision | sgravity
```

The output of the integrator is a subset of the commands of the grammar discussed in the sections above. Each of these commands describes one action to be performed independently from previous or following commands. Thus, the integrator works as a kind of context sensitive integration unit, maintaining its own state machine for the history of previous commands, but the commands emitted to the navigator are full words with regard to the formal language model of the context free grammar.

3.4 Status Feedback

In addition to the command stream, the system provides a feedback stream that selectively reports the current navigation mode and the status of the various system parameters. The information can be triggered on demand by external modules attached to the system. Moreover, it can either be sent automatically each time, the appropriate state value

```
<POS> ::= gothere                                  # moves to a specified position
<POS> ::= lookat                                   # orients to a specified position
<POS> ::= exacenter                                # sets the position as the examine center
<POS> ::= indicated geometry   f f f               # denotes a position on a geometry
<POS> ::= indicated pos        f f f               # denotes a position
<POS> ::= indicated posori     f f f f f f f       # denotes position and orientation
<POS> ::= indicated ori        f f f f             # denotes an orientation
<POS> ::= orientto pos         f f f               # orients to a given position
<POS> ::= moveto pos           f f f               # moves to a given position (exact)
<POS> ::= moveto nearpos       f f f               # moves near to a given position
<POS> ::= moveto ori           f f f f             # rotates to a given orientation
<POS> ::= moveto posori        f f f f f f f       # changes to a position and orientation
<POS> ::= beamto pos           f f f               # moves and orients to a position (exact)
<POS> ::= beamto nearpos       f f f               # moves and orients near to a position
```

Figure 3: BNF notation of position related commands

changes or it is sent in a fixed periodic feedback cycle. As the exact definition of the BNF defining the feedback grammar is similar to that of the control commands, it is not explicitly given here, but instead only the differences are pointed out. The `stepsize` feedback reports a parameter in form of a floating point value, that specifies a factor to be multiplied with the default navigation speed or current step size. The `viewpoint` value informs about the index of the currently bound viewpoint in the list of all viewpoints, the size of this list - which may change over time - and an associated name of that viewpoint, given as a string. At the start of a viewpoint transition, an `isbeaming` flag is sent, which serves as an indicator for the input modules to immediately stop any kind of movements that are not directly user initiated. This mechanism is of essential importance for the navigator whenever it generates movements for about a second after a user interaction. Those movements should not interfere with the viewpoint animations or even outlast them. The `undoing` flag informs the other modules either about the kind of action that has been undone or about the fact that the undo buffer is empty. This flag is most essential for both implementing user feedback behavior and maintaining an external context sensitive control unit.

4. IMPLEMENTATION DETAILS

VRML content becomes independent of available input hardware if the framework is a fixed part of the browser. But applications definitely need a possibility to adopt the behavior of input devices to their particular needs, or to implement special navigation paradigms for a certain type of audience. In the framework described in this paper, we use the mechanism of ROUTEs to break up the rigid event routing of conventional browser implementations. Additional devices can plug in the event dispatcher module of the browser and dispatch their input to a *DeviceSensor* node in the VRML scene graph, thereby giving the author full access to all kinds of conceivable user input and output. The *Camera* node facilitates the control of the navigation: By setting velocity vectors, the VRML author can animate the user position. A Script node connects both nodes, transforming user input to velocities. These nodes have already been implemented in the blaxxun Contact browser, allowing us to realize most parts of our framework in VRML. Both nodes are now discussed in the following two sections.

4.1 DeviceSensor Node

Based on the PROTO mechanism, the DeviceSensor node offers an elegant possibility to represent arbitrary input devices in VRML. Therefore, without any further requirements for additional syntax in VRML parsers, devices can be modeled in VRML as they are, not just as a flat array of floats. The DeviceSensor is not limited to input devices, even feedback can be routed to eventIns on the PROTO and thus to the device. One DeviceSensor node instance suffices to represent the complete input device, giving full access to the individual input elements. Due to the usage of the PROTO mechanism, parsers are independent of supported devices and browser vendors can create an extensible architecture that allows independent programmers to add support for further devices. The node definition is given below.

```
DeviceSensor {
  exposedField   SFBool    enabled    TRUE
  exposedField   SFString  device     ""
  exposedField   SFString  eventType  ""
  exposedField   SFNode    event      NULL
  eventOut       SFBool    isActive
}
```

The `device` field names the device that should be represented by the DeviceSensor node, e.g. "JOYSTICK". An optional number allows to distinguish between multiple devices of the same type. The `event` field refers to a PROTO instance with the device specific fields. Data can be routed to or from fields on that PROTO instance, but, additionally, `event` can also act as an eventOut, emitting the whole node when some fields on it are subject to changes. Moreover, `eventType` provides some means to pass initialization parameters to the device support and the `enabled` flag is used to enable the DeviceSensor. Related to this flag, the `isActive` flag evaluates to true if the following three conditions hold: the `enabled` flag is set, the device is supported by the browser and the device is physically present. Using the DeviceSensor in VRML is done in two steps:

The first step is to declare the PROTO for the device.

```
PROTO JoyStick [
   eventOut SFVec2f  stick
   eventOut SFBool   button1
   eventOut SFBool   button2
] {}
```

In the example given above, a typical joystick with two buttons is described. The definition part of the PROTO is empty because it is supplied by the browser. It is important that the fields of the PROTO and its association to a device name must be standardized for a device to allow interoperability between browsers (e.g. one for "MOUSE", one for "SPACEMOUSE", etc.), but browsers can support their own proprietary or experimental devices. If browsers do not force VRML content to declare all the standardized fields of the PROTO, the specification for a device can be extended. Thus, new content can still run on old browsers, and vice versa. For example, in the content definition, force feedback information could be routed to an eventIn SFVec2f force field and the class of browsers that support this feature would pass it to the joystick.

The second step is to define the DeviceSensor and associate an instance of the PROTO to its `event` field.

```
DEF DS DeviceSensor {
   device "JOYSTICK"
   event DEF JS JoyStick {}
}
```

This instructs the browser that the event node (the node that is assigned to event) is associated with the "JOYSTICK" device. If the browser supports joysticks, it sets the `isActive` flag to true, reads the joysticks status and supplies it to the fields on the event node. The content reads joystick data by routing the fields of the event node away, either individually or the complete event node at once from the `event` field of the DeviceSensor. In order to allow a modular design, the DeviceSensor is not a bindable node. Thus, multiple DeviceSensor nodes in different PROTOs can query the same device, e.g. one PROTO queries the joystick buttons to fire weapons or select objects, while another PROTO queries the stick for navigation.

4.2 Camera Node

The Camera node allows VRML content to directly communicate with the internal navigation modules of the browser. In a traditional browser, mouse and keyboard input is interpreted and directly routed to the navigation module. With the Camera node, this strict routing is broken up. In contrast to the Viewpoint node, the author can move the user on non-predefined paths through the virtual world without any deep knowledge of 3D mathematics. Additionally, the author can control the collision detection, the center of rotation for examine mode, the third person view, and various other parameters. The node definition is given below.

```
Camera {
   eventIn       SFVec3f   xyz
   eventIn       SFVec3f   ypr
   eventIn       SFVec3f   opr
   exposedField  SFBool    collide         TRUE
   exposedField  SFBool    gravity         TRUE
   exposedField  SFVec3f   examineCenter   0 0 0
   eventIn       SFVec3f   moveTo
   eventIn       SFVec3f   orientTo
   eventIn       SFVec3f   beamTo
   exposedField  SFFloat   duration        2
   exposedField  SFVec3f   offset          0 0 0
   exposedField  SFBool    enabled
   exposedField  SFBool    disableDefault
}
```

Typically, the content will gain user input from an input device through a DeviceSensor, transform it in a Script node and then supply velocities and other commands to a Camera node. Coordinates routed to the Camera node are always interpreted relative to the currently bound viewpoint, since this is essential for navigation. In the following, we briefly explain the individual events and fields of the Camera node. For more details on the implementation, please refer to [7].

xyz, ypr, opr: Values routed to these eventIns are interpreted as velocity components in various directions. Thereby, `xyz` specifies translatory velocities to the right, up and backwards. While the components of `ypr` represent angular velocities around the y-, x-, and z-axis (yaw, pitch, roll), `opr`, in turn, denotes movements for the EXAMINE mode. This mode can be described as a rotation of the viewpoint around an examine center. Components of `opr` specify angular velocities around the x axis, y axis and viewing axis (the connection of view point and examine center).

collide, gravity: These boolean values are used to control the currently active status of the collision detection as well as the gravity simulation.

examineCenter: The component `examineCenter` allows the content author to set the center of rotation in the EXAMINE mode, which is a long missing feature in VRML.

moveTo, orientTo: Independent of the currently active navigation mode, when a position is routed to `moveTo`, the browser starts a viewpoint animation that moves the viewpoint to that position without any kind of rotation. A position routed to `orientTo` starts an animation that only rotates the user to look at the received point.

beamTo: Based on these two types of movement events, `beamTo` is a combination of translational and rotational movements with regard to the specified position. The browser must find a suitable position and orientation to transform the current viewpoint, i.e. a location from which the referenced object can be well seen and interacted with. If the target object of the `beamTo` movement is a picture on a virtual wall, it does not make much sense to move the avatar directly into the picture, but, instead, position him in a way that the complete picture is within the current field of view.

duration: The duration of the respective translational and rotational movements are controlled by the `duration` field. Thereby, the values are given in seconds.

offset: This field allows for third person view by utilizing the metaphor of moving the virtual camera out of the users body. When this value is nonzero, browsers that support `offset` move the camera away from the position of the user and insert an avatar into the scene. The scene is still influenced by the unmodified position, e.g. with ProximitySensor or collision detection, only the perspective of the scene view is changed. The virtual camera is always oriented to bring the avatar's head to the center of the screen. While the first value of `offset` defines its distance to the avatar, the others are two angles defining the direction from which the avatar is seen.

enable, disableDefault: If the `enabled` flag is FALSE, all fields on the Camera node are ignored. Velocities of each enabled Camera node are added to the movements of the built-in navigation module. For other eventIns on multiple enabled Camera nodes, the same rules apply as if the fields were on the same node. By setting the flag `disableDefault` in an enabled Camera node, the built-in navigation of the browser will be deactivated.

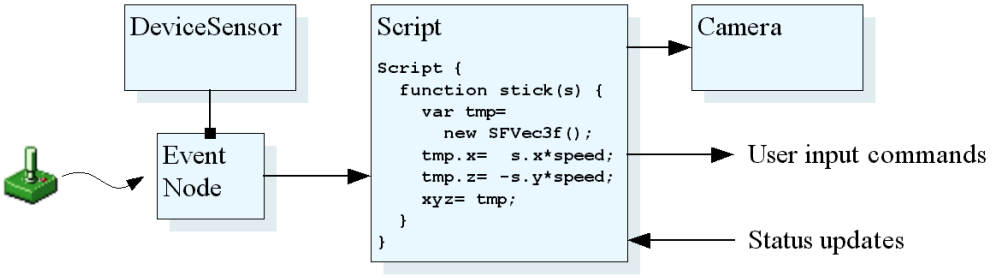

Figure 4: Structural elements for customizable applications

5. CUSTOMIZABLE APPLICATION

As a proof of concept for the developed system architecture and the interaction mechanisms, we have implemented a prototypical application in the proposed framework. Both the SHM modules and the integrator are not described here, since they are extensively discussed in [2]. This section concentrates on setting up a software framework for the browser front-end to create an application specific navigation handling. We make use of the VRML browser blaxxun Contact[7] because it has already implemented and successfully tested the Camera and DeviceSensor nodes described above.

5.1 Sample Implementation in VRML

The navigator module, containing undo buffer and discrete movements, as well as the haptic input modules have been implemented as VRML content, which gives us high flexibility to adopt things. In the case of the Interpreter, this VRML content can mainly rely on navigation features in standard VRML. In some selected cases, it additionally needs some special extensions of the blaxxun browser, e.g. an eventOut type list of the all Viewpoint nodes. The VRML content of the discrete integrator is represented, like any other module, as an EXTERNPROTO. Internally, it utilizes a DeviceSensor that loads a TCP plug-in to connect to the real integrator, running on another machine. The command language is represented as MFString events containing one terminal symbol in every array element. Thus, parsing the formalism in EcmaScript becomes rather simple.

5.2 Usage of DeviceSensor and Camera Node

We want to show the structure of a haptic input module in more detail because it demonstrates the typical usage of the DeviceSensor and the Camera nodes. The adder in the continuous integrator is part of the Camera node implementation, and the subsequent collision and ground detection are part of the VRML Browser. Therefore, the haptic input modules route their velocities to a Camera node. A DeviceSensor identifies the device and reads the appropriate input data. A Script node interprets the user input according to the state information that it receives from the interpreter. It either sends continuous movements to the Camera node as velocities, or commands to the discrete integrator. Figure 4 shows the major structural elements, needed in VRML to create customizable applications.

5.3 Application Scenarios

The framework introduced in this paper is currently used in two totally different applications. In the first scenario, the VRML browser is integrated in a HTML frameset used for interactively exploring a virtual model of our institute. In addition to the 3D area, the interface provides a 2D overview and several text areas. The user can walk through the rooms and different laboratories, access multimedia documents and communicate with the various professors and researchers.

In the second scenario, the framework is used as the visual 3D front-end for a multimodal interface to operate various comfort devices (radio, cd-player, telephone, etc.) in an automotive environment. Thereby, the conventional 2D display paradigm is broken up. Presented on a standard touschscreen, the interface extensively makes use of the additional dimension by providing alternative information arrangements and intuitive interaction techniques.

6. POTENTIAL EXTENSIONS

In this section, we briefly outline some potential extensions to our framework, that we are currently working on. Thereby, we concentrate on the integration of new input sources and adapting the concept of our system architecture to handle other interaction paradigms besides navigation.

6.1 Positional Devices

In the current form, our framework handles haptic input devices that generate data in the sense of velocities. Another way of gaining navigation input from users is to directly measure positions and orientations, e.g. by trackers that sense the position of the users head or hands. An extension to such devices involves allowing input modules to generate positions in addition to velocities. The continuous integrator would combine these movements additively to the velocities. The output of a positional input device can be expressed as a series of delta movements. These can be transmitted in an additional communication channel parallel to the one for velocities. In a simulation loop based implementation, the delta movement during each frame is relevant. On the other hand, velocities when multiplied with the current frame time (1/frame rate) also result in a delta movement for each frame. Since the collision detection module would probably need such deltas anyway, adding the deltas based on velocity to the deltas based on positions would combine both types of input.

6.2 Adding Communication Formalism to the Camera Node

Navigation in 3D space is a complex task, especially when it comes to give content designers a way to design their own user interface. Standardizing the formalism that is sent to the navigator module and providing an eventIn on the Camera node would give the content author a simple, yet power-

ful way to design user interfaces or to control the navigation through a world. EAI could give access to navigation for Java applets or other applications, that use the browser as a render engine. The beamto commands in combination with an EcmaScript function, that is fed by the mouse cursor position from the DeviceSensor and returns the 3D position under the cursor, could give finer control to custom beamto navigation in comparison to a simple SFVec3f typed eventIn. Profiles as used in conjunction with X3D could specify the set of supported browser commands. Such profiles could be the empty command set, the one sent to the navigator or the whole command set allowing the author to implement a complete input module. Similar to input, the formalism used for status updates could be provided as an eventOut on the Camera node and would allow the content to react on states and events in the navigation module, e.g. mode changes or collision events while navigating. A potential definition is given below.

```
Camera {
    eventIn  MFString  command
    eventOut MFString  event
}
```

In this way, the **command** would accept words or subwords of the formal grammar model like `ctrl, viewpoint` or `next`. On the contrary **event** would send status messages like `colliding` or `true` according to a separately defined formal grammar.

6.3 Extension to the Manipulation Paradigm

Navigation in 3D applications is very important, but interacting with the scene inheres at least the same importance. We briefly describe what will be necessary to extend our system to manipulation. The most important thing in manipulation is moving objects, either as a piece of the world e.g. in a furnishing application or as an abstract thing, e.g. the knob on a slider control. Another type of manipulation is activating things, the domain of the TouchSensor. Both need some way of selecting objects. Pointing devices like the conventional 2D mouse are capable of this, but others like the joystick cannot, unless they simulate a pointing device by moving a 2D or 3D cursor. A new `select` command, combined with an indicated geometry would express the gesture of selecting an object for manipulation. An `activate` command would issue the touchTime event on a TouchSensor, or connect the velocities of input modules with a drag sensor. A `deselect` command would reconnect the modules to navigation. Existing drag sensors do not cover the full range of movements in 3D space, as they are designed for 2D devices. We think that a general MoveSensor should cover all six degrees of freedom and it should have a field that defines which of these degrees are active. This would make existing drag sensors special cases of the MoveSensor. A CollisionSensor node as proposed in [4] would allow to move objects in complex geometry like in the furnishing example.

Our frame set should be extended by a manipulator module that executes manipulation commands in a similar way as the navigator does for navigation. Similarly, the continuous integrator would contain a corresponding module for manipulation. To allow authors to override the browser built-in behavior of input devices and modalities, the MoveSensor node should have velocity eventIns similar to those of the Camera node. This would allow to control specific objects. To override the behavior for all movable objects, an ObjectMover node could accept velocities, and supply them to the currently selected object. The structure DeviceSensor → Script → Camera or ObjectMover repeats.

7. CONCLUSION

In this work we presented a software architecture to support generic navigation devices in VRML browsers, covering conventional haptic input, special VR devices as well as advanced input modalities like speech and gestures. Both discrete and continuous navigation information is handled in the system. We introduced a DeviceSensor for grabbing arbitrary input and a Camera node to control the scene view and implemented both constructs in blaxxun Contact. By tailoring applications to user profiles as shown in the prototypical implementation, we can enable a broader use of VRML 3D applications. Finally, we identified potential extensions to our system to facilitate other interaction styles and support the VRML author in creating highly customizable applications.

8. REFERENCES

[1] F. Althoff, G. McGlaun, and M. Lang. Combining multiple input modalities for VR navigation - A user study. In *9.th Int. Conf. on HCI*, August 2001.

[2] F. Althoff, G. McGlaun, and M. Lang. Using multimodal interaction to navigate in arbitrary virtual VRML worlds. In *Workshop on Perceptual User Interfaces (PUI 2001)*, Orlando, Nov. 2001.

[3] F. Althoff, T. Volk, G. McGlaun, and M. Lang. A generic user interface framework for VR applications. In *9.th Int. Conf. on HCI*, New Orleans, August 2001.

[4] M. Brelot and J. Dufourd. Ideas for new BIFS sensors and applications. *At 50th. MPEG meeting*, Dec. 1999.

[5] A. Cheyer and L. Julia. Designing, developing and evaluating multimodal applications. In *WS on Pen/Voice Interfaces (CHI 99)*, Pittsburgh 1999.

[6] K.-H. Engelmeier et al. Virtual reality and multimedia human-computer interaction in medicine. *IEEE WS on Multimedia Signal Processing*, pages 88–97, Los Angeles, December 1998.

[7] Developer site of blaxxun interactive (Jan 2002). http://www.blaxxun.com/developer/index.html.

[8] M. Latoschik et al. Multimodale Interaktion mit einem System zur Virtuellen Konstruktion. *Informatik '99, 29. Jahrestagung der Gesellschaft für Informatik, Paderborn*, pages 88–97, October 1999.

[9] P. Morguet et al. Comparison of approaches to continuous hand gesture recognition for a visual dialog system. *Proc. of ICASSP 99*, pages 3549–3552, 1999.

[10] J. Nielsen. *Usability Engineering*. Morgan Kaufmann Publishers Inc., 1993.

[11] S. L. Oviatt. Multimodal interface research: A science without borders. *Proc. of 6th Int. Conference on Spoken Language Processing (ICSLP 2000)*, 2000.

[12] B. Schuller, F. Althoff, G. McGlaun, and M. Lang. Navigating in virtual worlds via natural speech. In *9.th Int. Conf. on HCI*, New Orleans, August 2001.

[13] A. Waibel, M. T. Vo, P. Duchnowski, and S. Manke. Multimodal interfaces. *Artificial Intelligence Review*, 10(3-4):299–319, August 1995.