Computers & Graphics 37 (2013) 193–201

Contents lists available at SciVerse ScienceDirect

Computers & Graphics

journal homepage: www.elsevier.com/locate/cag

Special Section on Touching the 3rd Dimension

Adapting user interfaces for gestural interaction with the flexible action and articulated skeleton toolkit

Evan A. Suma [a,*], David M. Krum [a], Belinda Lange [a], Sebastian Koenig [a], Albert Rizzo [a], Mark Bolas [b]

[a] Institute for Creative Technologies, University of Southern California, 12015 Waterfront Drive, Los Angeles, CA 90094, United States
[b] School of Cinematic Arts, University of Southern California, 900 West 34th Street, Los Angeles, CA 90089, United States

ARTICLE INFO

Article history:
Received 9 September 2012
Received in revised form
25 November 2012
Accepted 30 November 2012
Available online 7 December 2012

Keywords:
Natural interaction
Gesture
User interfaces
Video games
Middleware

ABSTRACT

We present the Flexible Action and Articulated Skeleton Toolkit (FAAST), a middleware software framework for integrating full-body interaction with virtual environments, video games, and other user interfaces. This toolkit provides a complete end-to-end solution that includes a graphical user interface for custom gesture creation, sensor configuration, skeletal tracking, action recognition, and a variety of output mechanisms to control third party applications, allowing virtually any PC application to be repurposed for gestural control even if it does not explicit support input from motion sensors. To facilitate intuitive and transparent gesture design, we define a syntax for representing human gestures using rule sets that correspond to the basic spatial and temporal components of an action. These individual rules form primitives that, although conceptually simple on their own, can be combined both simultaneously and in sequence to form sophisticated gestural interactions. In addition to presenting the system architecture and our approach for representing and designing gestural interactions, we also describe two case studies that evaluated the use of FAAST for controlling first-person video games and improving the accessibility of computing interfaces for individuals with motor impairments. Thus, this work represents an important step toward making gestural interaction more accessible for practitioners, researchers, and hobbyists alike.

1. Introduction

Recent advances in low-cost depth sensing technology have led to a proliferation of consumer electronics devices that can sense the user's body motion. The release of the Microsoft Kinect in late 2010 has sparked the rapid formation of a large and active community that has explored a myriad of uses ranging from informal hobbyist "hacks" to scientific research projects and commercial applications. However, despite the widespread accessibility of full-body motion sensing devices, designing intuitive and powerful gestural interactions remains a challenge for developers. In general, though the Kinect holds the record for the fastest selling consumer electronics device in history, the sales of many commercial Kinect for Xbox 360 game titles have been poor, which has been partially attributed to the lack of well-designed games that integrate body motion seamlessly into the experience [1,2]. Indeed, research has shown that performing physical arm movements and gestures can have a profound impact on the user's attitudinal and emotional responses to visual stimuli [3–5]. These observations point to the need for both a theory of "gesture design" as well as the tools to enable the creation and customization of gestural interactions for 3D user interfaces and interactive media.

An important motivator for our work is the application of video game technology toward advances in the areas of rehabilitation [6] and health [7]. While the clinical value of leveraging motion gaming technology has received increased recognition in recent years, these applications pose several notable challenges for designers. Unlike commercial games, body-based control in a clinical setting is not "one-size-fits-all," and must be customizable based on individual patient medical needs, range of motion, and motivation level. For example, a client with impaired arm movement would require a therapy game that encourages motion just outside the boundary of comfort, but not so far that achieving the required body pose becomes overly frustrating or impossible. Thus, the gestural interactions need to be designed on a per-client basis by the clinician, who often may not possess intimate technical knowledge and programming skills. Furthermore, these interactions need to be easily and immediately adjustable as the patient improves or encounters problems.

To facilitate the integration of full-body control with third party applications and games, we developed a middleware software framework known as the Flexible Action and Articulated Skeleton Toolkit (FAAST). The toolkit enables the adaptation of existing interfaces for

* Corresponding author.
E-mail address: suma@ict.usc.edu (E.A. Suma).

0097-8493/$ - see front matter © 2012 Elsevier Ltd. All rights reserved.
http://dx.doi.org/10.1016/j.cag.2012.11.004

gestural interaction even though they never intended to support input from motion sensors. To achieve the goal of intuitive and transparent gesture design, we defined a syntax for representing complex gestural interactions using rule sets that correspond to the basic spatial and temporal components of an action. These "action primitives" are represented as plain English expressions so that their meaning is immediately discernible for both technical and non-technical users, and are analogous to interchangeable parts on an assembly line—generic descriptors that can be reused to replicate similar gestural interactions in two completely different end-user applications. FAAST provides a complete end-to-end framework that includes a graphical user interface for custom gesture creation, sensor configuration, skeletal tracking, action recognition, and a variety of output mechanisms to control third party applications such as 3D user interfaces and video games. FAAST can either be used to support development of original motion-based user interfaces from scratch or to repurpose existing applications by mapping body motions to keyboard and mouse events, and thus represents an important step toward making gestural interaction design and development more accessible for practitioners, researchers, and hobbyists alike. In this paper, we present the FAAST system architecture (Section 3), our approach for representing and designing gestural interactions (Section 4), supported output modalities for manipulating arbitrary user interfaces (Section 5), and two case studies in which FAAST was evaluated within a specific application domain: controlling first-person video games for entertainment and increasing user interface accessibility for individuals with motor impairments (Section 6).

2. Previous work

2.1. Gesture recognition

Computational analysis of human motion typically requires solving three non-trivial problems: detection, tracking, and behavior understanding [8]. In this paper, the software libraries from OpenNI and Microsoft Research provide both user detection and skeletal tracking, so FAAST subsequently focuses on recognizing the action being performed by the tracked user and generating an appropriate output. The quantity of literature focusing on action and gesture recognition from the computer vision and machine learning communities is vast (see [9], [10] for reviews). While approaches based on the statistical modeling, such as Hidden Markov Models [11] and condensation algorithms [12], are often highly sophisticated, such methods would often be treated as "black box" algorithms by non-technically oriented users. While this may be permissible for a video game scenario where a user wants to recognize several distinct gestures by example, they are not appropriate for use in clinical settings where the fine-tuning of individual motions based on patient requirements is essential. For example, to be useful for motor rehabilitation, the clinician must precisely specify the motion thresholds for gesture activation so that the elicited movements are just beyond the comfortable range of motion, but still within the patient's ability to perform. Thus, the method described in this paper aims to be comprehensible for non-technical users without obscuring the low-level motion mechanics for adjustment at runtime. Our approach is perhaps most conceptually similar to gesture recognition methods based on the spatio-temporal finite-state machines [13], although we represent states as simple lists of plain English expressions that describe their spatio-temporal attributes.

2.2. Human motion representation

The use of animated virtual characters has led to the development of a variety of high-level markup languages for designing behaviors with synchronized gestures, facial expressions, and spoken dialog. Examples include the virtual human markup language (VHML) [14], avatar body markup language (ABML) [15], and numerous others (see [16] for a review). While such languages are useful for designing characters that communicate both verbally and non-verbally, physical behaviors are generally represented as pre-recorded animation files or high-level behaviors that will be calculated through inverse kinematics (e.g. pointing at an object). As they generally do not include specifications for representing the mechanics of the physical motions themselves, they are not appropriate for designing gestural input for real-time user interfaces.

It is perhaps more informative to examine human movement representations from other domains. Laban Movement Analysis (LBA) is a formal and universal language for describing human movement, including both its low-level muscular attributes and high-level expressive features [17]. Although originally derived from the study of dance, LMA has been applied to a variety of domains, such as analyzing the motion of subjects who had been affected by stroke [18]. LMA defines four components of movement encompassing both kinematic and non-kinematic features: Body, Space, Effort, and Shape. While this method provides a rich analytic language for describing human motion, it is also highly complex - LMA certification programs typically require many hours of comprehensive study encompassing both the theory and applications of the framework. Since our goal was to provide a motion representation that is accessible for non-technical users with minimal training, it does not require the rich, complex conceptualization of low-level muscular anatomy nor the high-level analysis of expressivity provided by LMA. Therefore, we focused on defining a relatively simpler plain English format to represent the kinematic aspects of motion without the use of anatomical jargon.

2.3. Software frameworks

The initial version of FAAST was released in late 2010, shortly after the release of the Microsoft Kinect, and supported a very limited set of interactions, such as extending the arm directly forward to activate a key press [19]. After the initial interest in the toolkit, mouse control with the user's hand was subsequently added. The idea for generating virtual input events was inspired in part by GlovePIE, a programmable input emulator that maps signals from a variety of hardware devices such as the Nintendo Wiimote into keyboard, mouse, and joystick commands [20]. Another similar solution is Kinemote, which allows mouse control of Windows applications and games using a floating hand and a "Palm Click & Drag" metaphor [21]. However, to the best of our knowledge, FAAST is the first software toolkit that defines a simple yet powerful action syntax and provides an interface for non-technically oriented users to design gesture recognition schemes in an intuitive way.

3. System overview

FAAST is designed as middleware between the depth-sensing camera skeleton tracking libraries and end-user applications. Currently supported hardware devices include the Kinect sensor using the Microsoft Kinect for Windows SDK and any OpenNI-compliant sensor using the NITE skeleton tracking library from PrimeSense. Fig. 1 illustrates the toolkit's architecture. To make the toolkit as device agnostic as possible, communication with each skeleton tracker is split into separate modules that are dynamically selected and loaded at runtime. Each module reads data from its respective tracker into a generic skeleton data structure. Thus, FAAST is easily extensible by adding new modules to support future devices and software libraries.

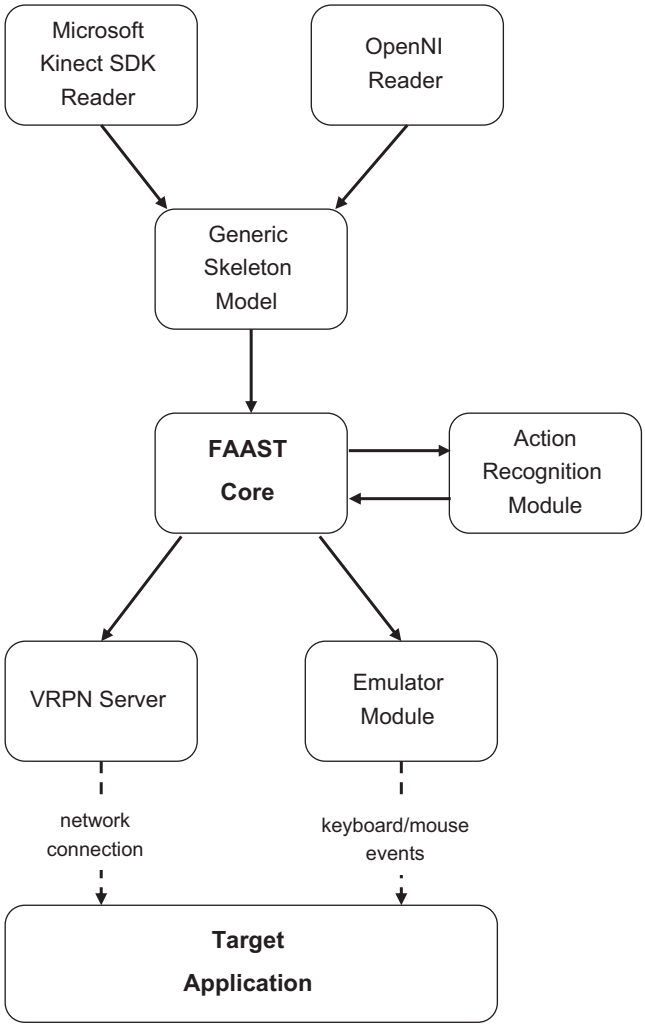

Fig. 1. The FAAST architecture consists of modules that read skeleton data from depth sensing camera libraries, an action recognition module, an emulator module that sends simulated keyboard and mouse events to the active window, and a VRPN server to broadcast the position and orientation of each skeleton joint to networked applications.

Once a skeleton has been read from the sensor, FAAST then processes this data in an action recognition module. Based on the actions being performed, the core application then invokes an emulator module that sends simulated keyboard and mouse events to the active window. The core FAAST application includes a graphical user interface (GUI) that allows the user to create custom gesture schemes and bind these actions to specific emulator events. For example, the user may choose to send a "left arrow key press" event when the user leans to the left by a certain number of degrees (see Fig. 2). It is possible to specify any number of gestures, each of which contains user-defined lists of gestural input criteria and output user interface commands to be generated upon activation. Thus, FAAST allows users to control off-the-shelf applications and games using body motion, even though these interfaces only accept input from standard keyboard and mouse setups. The methods for representing gestural input and generating output for arbitrary user interfaces are described in more detail in the following two sections.

In addition to simulating keyboard and mouse events, FAAST also streams the position and orientation of each skeleton joint using a Virtual Reality Peripheral Network (VRPN) server, which is a network interface that has become a popular standard for facilitating communication between VR applications and physical tracking devices [22]. This allows developers to read skeleton data into their applications in a device-independent manner using the same standard protocols that are commonly used by the community. FAAST is currently being officially supported by two commercial virtual world development environments for this purpose, 3DVIA Studio [23] and the Vizard virtual reality toolkit from WorldViz [24].

The FAAST GUI is composed of three windows that can be moved together or independently: the main window for configuring the toolkit, the viewer for displaying sensor data, and the console for displaying messages to the user (see Fig. 2). The main window is split into four tabs for configuring the sensor hardware, VRPN server, display preferences, and gestural interactions. Initially, the gesture GUI began as a simple text window where the user would type a single input motion and a corresponding output command. However, preliminary feedback with users indicated that the utility of this interface was limited, since it was unclear how one would intuitively combine multiple input motions and output commands within a single gesture. Additionally, remembering all the valid skeleton joint names and output commands placed too much of a burden on users' working memories. Through iterative development and testing, we developed a more intuitive user interface that uses a tree GUI element for representing gestures. A gesture is composed of an input list and an output list, each of which can contain an arbitrary number of items. Items appear as plain English expressions so as to be easily comprehensible to non-technical users, and are constructed and edited through dialog boxes with drop-down menus (see Fig. 3). Each item can also be disabled/enabled, copied, and repositioned in the list through click-and-drag interactions. While the interactions provided by this interface appear to be quite straightforward for novice users, it does not currently incorporate a "visual preview" of the defined gestures using a virtual human model, and this may be an area for future improvement.

4. Representing and designing gestural interactions

In this section, we describe our representation scheme for the individual components that compose gestural interactions, followed by FAAST's capabilities for designing complex gestures using this core mechanic.

4.1. Decomposing complex gestures

In order to enable users to design custom gestural interactions, we considered how to decompose complex body movements into atomic components. These simple action primitives form the conceptual "building blocks" that can be combined in FAAST to form more complicated gesture recognition schemes. To make these actions comprehensible and intuitive for non-technical users, they are constructed by selecting terms from drop down boxes to produce plain English expressions, similar to the way they might be described in everyday conversation (see Fig. 3). For example, to design an action that activates when the user extends the left hand directly out in front of the body, the user would select parameters to form an expression such as: "left hand in front of torso by at least 16 inches." For each action specified, the user must also specify the *comparison* (either "at least" or "at most"), the numeric *threshold* for activation, and the *units* of measurement.

4.1.1. Position constraints

Position constraints refer to the relative spatial relationship between any two skeletal joints, depicted as follows:

```
{body part}{relationship}{bodypart}
  by{comparison}[threshold]{units}
```

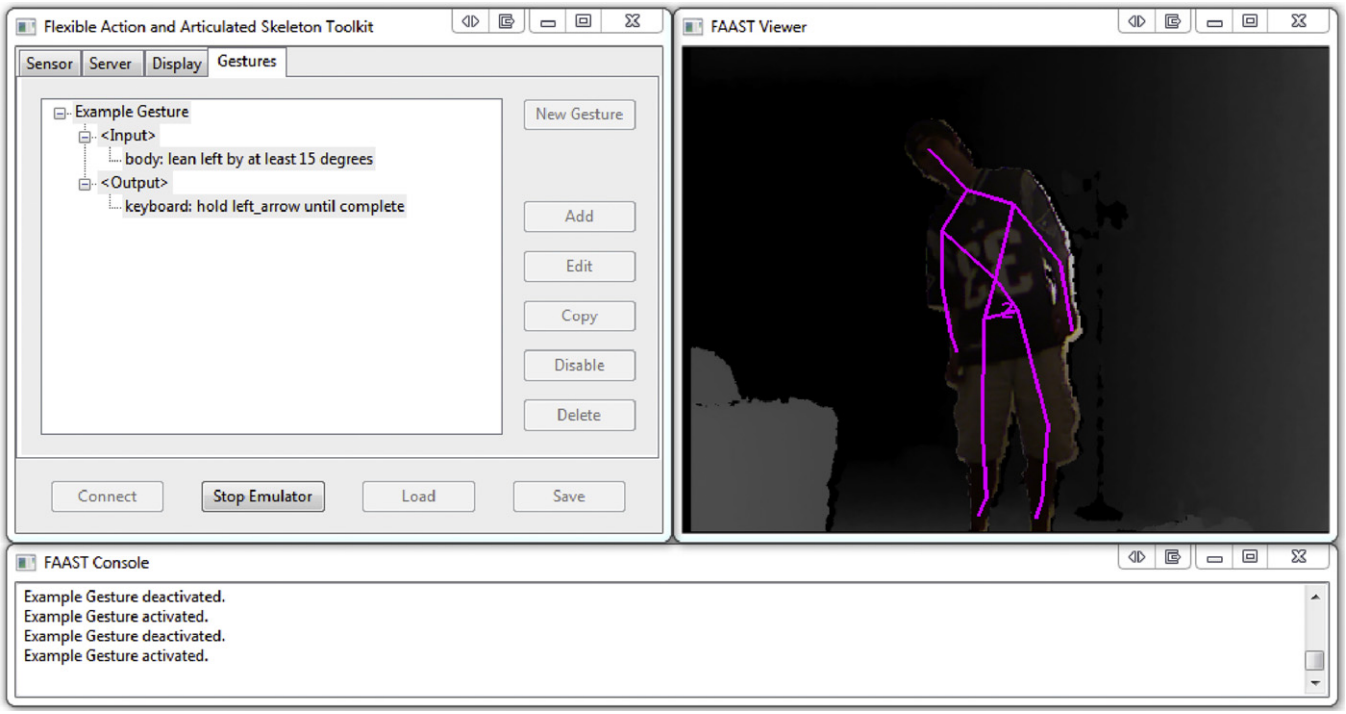

Fig. 2. A screenshot of FAAST's gesture creation interface. The specified example gesture activates when the user leans to the left by 15 degrees. When the user performs this action, a left arrow key down event is sent to the application with current operating system focus. The corresponding key up event is sent when the user returns to the original position.

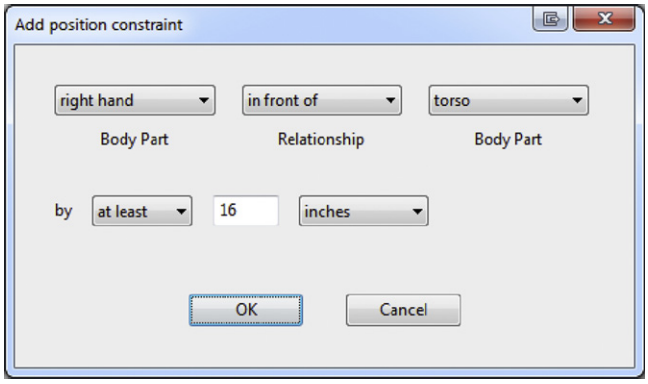

Fig. 3. Rules are defined using drop-down boxes that form plain English expressions. These action primitives form the basis for more complicated gestures.

{body part}=head, neck, torso, waist, left shoulder, left elbow, left wrist, left hand, right shoulder, right elbow, right wrist, right hand, left hip, left knee, left ankle, left foot, right hip, right knee, right ankle, right foot {relationship}=to the left of, to the right of, in front of, behind, above, below, apart from {units}=centimeters, meters, inches, feet

4.1.2. Angular constraints

It is also useful to consider cases when users flex or straighten one of their limbs, determined by calculating the angle of intersection between the two vectors connecting the limb's joint locations. Angular constraints are depicted as follows:

```
{limb}flexed
  by{comparison}[threshold]{units}
```

{limb}=left arm, right arm, left leg, right leg
{units}=degrees, radians

4.1.3. Velocity constraints

Velocity constraints refer to the speed at which a particular body part is moving, depicted as follows:

```
{body part}{direction}
by{comparison}[threshold]{units}
```

{body part}=head, neck, torso, waist, left shoulder, left elbow, left wrist, left hand, right shoulder, right elbow, right wrist, right hand, left hip, left knee, left ankle, left foot, right hip, right knee, right ankle, right foot {direction}=to the left, to the right, forward, backward, up, down, in any direction {units}=cm/sec, m/sec, in/sec, ft/sec.

4.1.4. Body constraints

In addition to the constraints listed above, it is also useful to consider body actions that are more "global," i.e. movements of the whole body relative to the camera, as opposed to positioning individual body parts relative to one another. We define two angular body actions, *lean* and *turn*, depicted as follows:

```
lean{left,right,forward,backward}
by{comparison}[threshold]{units}

turn{left,right}
by{comparison}[threshold]{units}
```

{units}=degrees, radians

Additionally, we also define a *jump* action. Measuring the height of a jump is not immediately obvious, however, because the height value of each skeleton joint is relative to the sensor, not the floor. Thus, to detect jumps, we leverage the fact that these actions occur very quickly, and consider the lowest height value of the feet over a previous window of time to be the height of the floor (experimentally determined to be 0.75 s). The jump action activates when both feet rise above this floor height value by the specified distance threshold. We found that when the user stands

in one spot, which is frequently the case when interacting with depth sensors due to the restricted field of view, jump detection is quite reliable. Jump actions are depicted as follows:

```
jump by{comparison}[threshold]{units}
```

{units}=centimeters, meters, inches, feet

4.1.5. Time constraints

The last type of constraint we consider is the temporal delay between the individual actions that constitute a gesture. In informal testing, we determined that it can be useful to be able to define action timing based on either previous action's start time or stop time, depending on the specific gesture being designed. Thus, time constraints are depicted as

```
wait for [minimum] to [maximum]seconds
after action{starts, stops}
```

4.2. Designing complex gestures

The atomic actions described in the previous section are intentionally simple, and as a result the interaction possibilities provided by each individual constraint are limited. However, by combining these "action primitives" both simultaneously and in sequence, sophisticated gestural interactions can be represented. Thus, we developed a methodology for designing complex gestures using the previously described atomic actions as conceptual building blocks. Gestures are represented as a set of action constraints that can be combined simultaneously, sequentially over time, or any combination of the two.

4.2.1. Simultaneous actions

Any input actions that are added without a time constraint separating them are treated as simultaneous events. The overall gesture will only be activated when all of the input conditions are simultaneously true at a given moment in time. For example, to create an interaction that requires the user to move both hands together in a quick "push" action out in front of the body, one could specify the following gesture:

```
position :  right hand in front of torso
            by atleast 16 inches
position :  left handin front of torso
            by atleast 16 inches
position :  right hand apart fromleft hand
            by at most 10 inches
velocity :  right hand forward
            by at least 5 ft/sec
velocity :  left hand forward
            by at least 5 ft/sec
```

4.2.2. Sequential actions

By combining position, angular, velocity, and body constraints simultaneously, users can finely tune body pose and movement at a singular moment in time. However, many gestures require multiple actions to be performed in sequence. We represent this behavior using time constraints as separators between actions. When a time constraint is encountered, the action recognizer waits until the minimum time has elapsed (which may be zero), and then starts checking for the next action in the sequence to activate. If the actions do not activate before the maximum time has elapsed, the gesture resets and action recognition resumes from the very beginning of the sequence. For example, the following sequence represents a "wave" gesture, that activates when the user's right hand moves rapidly back and forth relative to the elbow:

```
position :  right hand to the right of
            right elbow by at least 4 inches
    time :  wait 0 to 0.5 seconds
            after action stops
position :  right hand to the left of
            right elbow by at least 4 inches
    time :  wait 0 to 0.5 seconds
            after action stops
position :  right hand to the right of
            right elbow by at least 4 inches
```

4.2.3. Combining simultaneous and sequential actions

The combination of simultaneous and sequential actions allows users to design complex gestures that FAAST can recognize in real-time. There is no predetermined bound on the complexity or number of gestures that users may build, and so ultimately they are limited only by their own creativity. Furthermore, because the gestures are created by combining rules that are individually simple, the process of tweaking the parameters of the action is easily discernible. The following example combines the sequential action of flapping the arms with a forward body pose, and represents an interaction that might be appropriate for controlling a flying game:

```
    body :  lean forward by at least 20 degrees
position :  left handto the left of
            left shoulder by at least 16 inches
position :  right hand to the right of right
            shoulder by at least 16 inches
position :  left hand below left shoulder
            by at most 4 inches
position :  right hand below right shoulder
            byatleast4inches
    time :  wait 0 to 0.5 seconds
            after action stops
    body :  lean forward by at least 20 degrees
position :  left handbelow left shoulder
            by at least 16inches
position :  right hand below right shoulder
            byatleast16inches
```

5. Manipulating arbitrary user interfaces

After designing the input criteria for gesture activation, it is necessary to define the output that FAAST should generate when the user performs a gesture. Similar to the representation scheme for gestural input, FAAST provides a robust set of output events, which can be combined simultaneously and sequentially. This approach allows the user to design sophisticated macros that can manipulate arbitrary user interfaces in a variety of ways, ranging from simple events to complex behaviors. All generated output is sent directly to the application that has the current operating system focus via calls to the underlying Windows API.

5.1. Output events

In order to design the mechanisms for manipulating arbitrary interfaces, FAAST supports a number of output event types that correspond to common input methods for off-the-shelf PC applications. Similar to the approach used in Section 4, the rules defining output are constructed by selecting terms from drop

down boxes to form plain English expressions (see Fig. 4). Possible output events include:

- *Keyboard events*: These events refer to singular keystrokes, which can be specified as a "press" (a key down followed immediately by a matching key up event) or a "hold" for either a fixed amount of time or until the gesture is deactivated.
- *Typing events*: These events generate a series of key presses for a complete string of text, allowing sentences and phrases to typed into an interface.
- *Mouse button events*: These events refer to singular action of a mouse button (left, right, or middle), following the same "press" and "hold" rules as keyboard events.
- *Mouse wheel events*: These events refer to a movement of the mouse scroll wheel (up or down) for a specified number of clicks.
- *Mouse move events*: These events refer to a movement of the mouse cursor, specified in either absolute coordinates (a specific position on the desktop) or relative coordinates (a position in relation to the cursor's current location).
- *VRPN button events*: FAAST can also send button signals over a network using the same VRPN server as the skeleton data. A total 255 buttons are supported, which are specified by an index number. These events follow the same "press" and "hold" rules as keyboard events.
- *FAAST events*: These events are special FAAST system commands, which are included for convenience of operating the application remotely. Commands include "stop emulator," "start emulator," and "resume emulator."
- *Time events*: These events specify a temporal delay that can be inserted between output events.

5.2. Output macros

The user can create any number of output events for a single gesture. When the gesture is activated, the output events will be executed, starting at the beginning of the list. If a time event is encountered, then FAAST will wait the specified amount of time before continuing. Any events that are not separated by a time delay will be executed immediately. This architecture allows for the creation of complicated macros that can achieve a wide variety of user interface tasks usually performed with a keyboard and mouse. For example, a gestural application launcher could be created by defining gestures using the following output scheme:

```
keyboard : pressleftwindows
time : wait0.1 seconds
type : [applicationname]
time : wait0.1 seconds
keyboard : pressenter
```

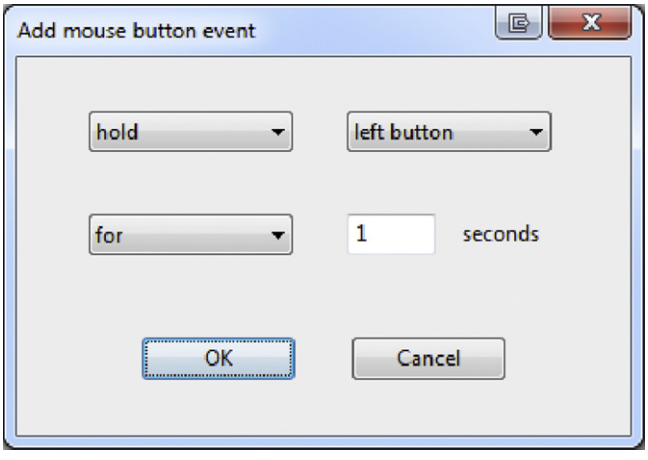

Fig. 4. Similar to our gestural representation scheme, output rules are defined with drop-down boxes that form plain English expressions, which can be combined simultaneously and sequentially to form complex macros.

By default, the set of output events is only executed once per gesture activation. However, there are scenarios where it may also be desirable to execute output events repeatedly until the user stops performing the gesture. Therefore, the user is also given the option of looping the output events with a specified timeout value, which may be zero (at which point, the output events are executed every frame that the gesture is active). For example, Fig. 5 shows a manipulation scheme using four complementary gestures that when executed repeatedly allow users to gradually control mouse cursor by moving the right hand relative to the shoulder.

6. Case studies

Starting as a very simple toolkit, FAAST evolved gradually based on experimentation in a variety of contexts and user feedback from the wider community. In this section, we detail the observations, conclusions, and limitations of using FAAST to adapt user interfaces for gestural interaction in two specific areas: (1) control of first-person video games for entertainment and (2) improving user interface accessibility for individuals with motor impairments.

6.1. First-Person video game control

To assess the effectiveness of using FAAST for control of first-person video games, we attempted to design gestural interactions for the action role-playing game *Elder Scrolls V: Skyrim*, published by Bethesda Softworks. A popular, critically acclaimed game that sold over 3.4 million units in the first two days of release, Skyrim is an ideal test case because it has a control scheme commonly found in first-person video games, but also involves a rich level of interaction with the virtual world.

6.1.1. Locomotion

Locomotion is one of the common and fundamental tasks performed when interacting with 3D user interfaces [25]. To initiate movement in the game world, we used FAAST to simulate a walking-in-place scheme, which has been to positively contribute to the user's sense of presence compared to joystick locomotion [26]. Through trial-and-error experimentation, we determined that two separate gestures, "left foot step" and "right foot step" could be defined to produce walking-in-place locomotion, both of which was mapped to the 'w' key for forward motion in the game. The definition for each gesture is as follows:

```
LeftFootStep
  〈Input〉
    position : leftfootaboverightfoot
          byatleast6 inches
  〈Output〉
    keyboard : holdwfor1.5 seconds
RightFootStep
  〈Input〉
    position : rightfootaboveleftfoot
          byatleast6 inches
  〈Output〉
    keyboard : holdwfor1.5 seconds
```

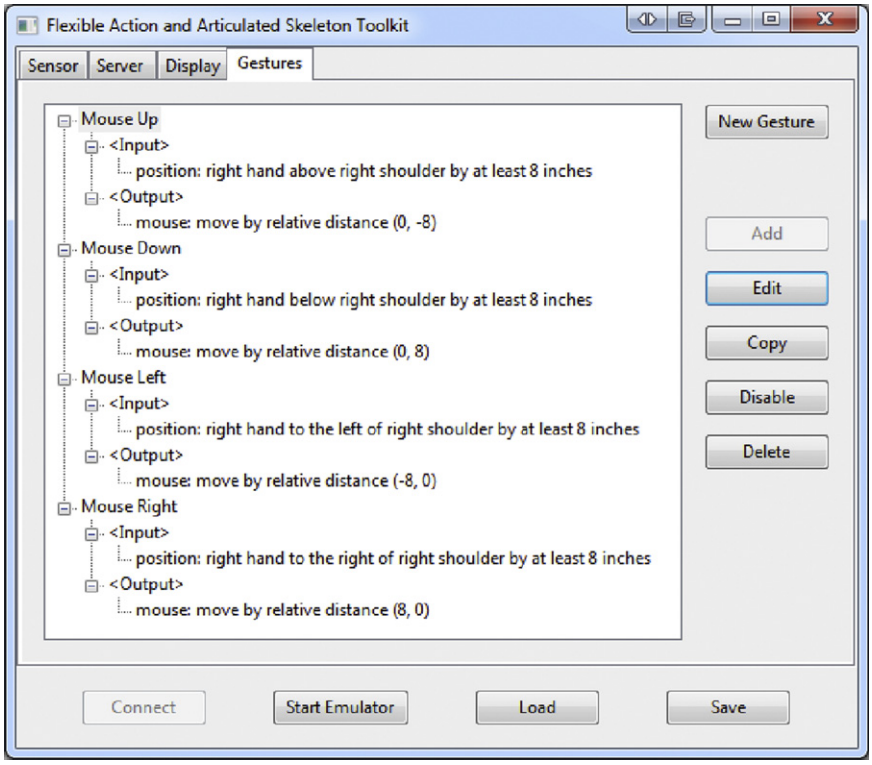

Fig. 5. By executing output events repeatedly in a loop, users can control user interfaces continuously over time. In this example, moving the right hand relative to the shoulder will produce gradual movement of the mouse cursor in one or more directions.

While this locomotion metaphor is reasonably effective, we do note that there is a slight delay in detecting when the user stops walking. In general, there is a tradeoff between responsive detection when the user stops and providing continuous motion between steps. However, the amount of time that the camera is moving after the user stops is relatively small (less than a second), and appears to be within tolerable limits.

However, fine camera control is a more serious limitation of using the Kinect for locomotion. Turning left and right is an important component of aiming in combat oriented games, and our first choice of body lean proved too coarse for effective gameplay. Currently, we believe that gestural input is not well suited for fine camera control in first-person games due to the precision required, and this problem remains an area for future work. Therefore, we would suggest augmenting a gestural control scheme for these types of games with an another interaction method for turning and aiming, such as a handheld controller or an external orientation tracker.

6.1.2. *Manipulation*

While gestural interaction is challenging for locomotion, it is particularly well suited for many manipulation tasks in first-person video games. In Skyrim, combat is achieved through keyboard or mouse commands that refer to the avatar's left hand, right hand, or both simultaneously. The specific function that each hand performs in the game depends on the current activity that is equipped, which could be a variety of weapons, items, or spells. In the case of spellcasting, if the same spell is equipped in both hands, then moving the hands forward together produces a singular, more powerful version of the spell. It proved relatively simple to map quick forward movements of each hand (using both position and velocity constraints) to control these behaviors in an intuitive way.

The action of switching the currently equipped weapon or spell is particularly well suited to gestural input, and it is here that our approach proved particularly effective. Equipping a sword could be achieved by reaching a hand down below the waist, and then quickly drawing the hand upward above the head. Most interestingly, we found that drawing symbols in the air and other complicated hand movements using sequential position constraints were an entertaining method of spellcasting that extremely evocative of fictional works from the fantasy genre. However, while powerful, these gestures require a fair amount of trial-and-error to tweak the positional thresholds until they work seamlessly. Additionally, due to the differences in body types, the parameters for a particular user may not translate perfectly to others. This points to a future need for the capability of automatically calculating these positional thresholds based on the user's motion, or alternatively specifying thresholds as percentages of the user's body size and limb length.

Additionally, it is also common in video games, particularly those from the role-playing genre, for there to be multiple contexts for interaction. For example, we found that occasionally while experimenting with the game, a virtual character would approach the avatar and initiate conversation. With no method of selecting dialog options via gesture, the user would have to approach the mouse and keyboard and perform this task manually. In the future, it would be beneficial to provide multiple interaction contexts, each with their own set of interactions, that the user could switch between using gestures based on the immediate game task. Additionally, it may also be possible to extract context information automatically from third-party applications, but this is an ambitious, non-trivial question beyond the scope of the current work.

6.2. *Accessibility for individuals with motor impairments*

Children with motor impairments such as cerebral palsy and adults with neurological impairments often have severe difficulty manipulating computing interfaces that are designed for people

with typical fine-grained motor control and reaction times. To increase accessibility to interactive entertainment content for these children and adults, a team of physical therapists and psychologists experimented with using FAAST to provide control mechanisms for simple desktop and web based PC games that these children and adults could manipulate. This application highlights the unique advantages for FAAST - it provides the capability for domain experts (non-programmers) to design gestural interactions that can be customized to fit the needs of individual patients that vary widely in individual capabilities. During this testing phase, the researchers and clinicians provided feedback based on each round of evaluation with patients, and we iteratively improved the user interface based on their experiences and suggestions. In this section, we summarize some of the major feedback and problems they encountered, along with our modifications to the toolkit that addressed these issues.

Initial technology testing sessions were completed in the lab setting, followed by user testing sessions in the clinical setting (two clinics in the Los Angeles area). Each of the testing sessions followed the same protocol. Following informed consent, the participant and clinician were provided with information about how the FAAST system works and were provided information about the game they were going to play. Both the participant and clinician were involved in the decision making process of mapping gestures to individual key presses within the game, based on their knowledge about the game and their comfort and ability to perform different gestures. Testing sessions ranged from 30 min to 2.5 h in length, depending on the time limitations of the clinician/participant, the amount of time taken to design the gestures, technical difficulties, and number of games attempted. During the session, participants and clinicians were asked to provide comments and feedback on the system and their experience. Following completion of the testing session, both the participant and clinician were also asked to provide feedback on the session and suggestions for improvements to the system. The researchers also noted technical difficulties and potential changes to the systems based on their experience. The feedback was analyzed and revision suggestions were prioritized and some were implemented before the next testing session. Other revisions were added to the list of future changes. A total of six participants and four clinicians took part in multiple testing sessions.

The application was initially difficult to use when customizing interactions for individual patients because there did not exist an easy way for evaluators to have the patient test single movements/gestures and then perform multiple gestures together without having to set up different files. This increased the time, effort and frustration associated with entering and testing gestures. Based on this feedback, we added the capability to enable and disable gestures on the fly. While a simple modification, this capability saved a substantial amount of time for the evaluator, made the session run more smoothly, and caused less frustration for both the evaluator and the patient.

The evaluators were able to design gestures to activate various keyboard and mouse commands that vary across different games—for example, some games require a button hold, others require a quick single button press and release or multiple button presses in quick succession. For the latter situation, it is difficult for patients to repeat a gesture multiple times in quick succession. This initial feedback provided the motivation for adding the looping behavior that repeatedly generates output events. However, there is still an issue when a third party software or game has multiple rules for one button - for example, 'jump' might result from a single button press of 'w' key, however 'higher jump' or 'flying' might result only when the 'w' key is pressed multiple times in quick succession. This means that the same key must be controlled by different gestures to perform each of these actions. While not insurmountable in the current setup, this does provide an additional layer of complexity that we plan to address in the future.

One major difficulty of using gestures to control third party applications and online web browser games is that the applications not only require key strokes to play the game, but often require complicated navigation through menus to set up the game and during the game to add features, level up, etc. Currently, FAAST is unable to intuitively support the navigation through these menus with many different options, and so the evaluator must intervene using a keyboard and mouse. While the evaluators indicated that the ability to control the mouse cursor with FAAST is a good start toward solving this interaction challenge, but with complicated menus with many options this becomes tedious and impossible for most motor-impaired patients. In the future, it may be possible to detect when text menus are displayed on screen through computer vision techniques and then automatically target specific regions on the screen to make menu selection more intuitive via gestural input.

When playing in a web browser, using the mouse control function, the player is not confined to the active (game) window and therefore, it is difficult for some patients to stay within the game area, causing frustration and providing a limited and less engaging game-play experience. Evaluators suggested that being able to limit the mouse or confine the mouse within an area identified by the user (for example, by drawing a box in which the mouse can move) would be helpful in this situation. This feature is currently being implemented for a future version of the toolkit.

Finally, evaluators also suggested that future versions of FAAST could be augmented with a set of pre-programmed default gestures that can be adapted and modified for individual users and applications. As we collect more information on how FAAST is used by real world practitioners, we plan to analyze this data and identify the useful common gestures that can be provided automatically.

Overall, the use of FAAST with patients has allowed researchers and clinicians to explore how patients with limited or abnormal movements can interact with web browser games and other third party applications. The targeted evaluation has provided us with important feedback that would not have been identified without testing without this specialized group of patients, and has allowed us to expand and improve FAAST's capabilities. Although there are a number of improvements to be made in the future to increase usability for individuals with motor impairments, FAAST is still able to provide patients with a level of accessibility to interact with third party applications and games that has been a major challenge in the past.

7. Conclusion

In this paper, we describe a core mechanic for designing and customizing gestural interactions and present an integrated toolkit that enables the adaptation of existing user interfaces for body-based control. By providing gesture design tools that are transparent and easily discernible to non-technical users, along with the capability of repurposing third party applications, FAAST is particularly useful not only for entertainment purposes, but also for researchers and clinicians working in domains such as rehabilitation and accessibility for individuals with motor impairments. While FAAST was designed to take advantage of OpenNI-compliant depth sensors such as the Microsoft Kinect, we expect that our approach will be extended to support other types of full-body tracking hardware as they become available. With access to

more complex and robust skeleton, hand, and finger tracking data, however, this process of defining rule sets may become more complicated. Thus, in the future, we plan to develop graphical user interface frontends that can generate rule sets automatically from recorded gestures or visual representations of human motion (e.g. keyframing), and explore hybrid methods that combine the fine tuning capabilities of our rule-based approach with the convenience of statistical gesture recognition classifiers.

The toolkit is free software that we have made available for download and use with a non-restrictive license [27]. In the time since initial release, the popularity of FAAST has grown considerably, and the most exciting consequence of this wide adoption has been observing the many creative and novel uses of the toolkit by members of the community. Building on the existing user base, we believe that our easy-to-understand rule-based gesture creation paradigm will further empower hobbyists, researchers, and practitioners to design rich gestural interactions in novel ways that we have not expected.

References

[1] Hughes D. Microsoft Kinect shifts 10 million units, game sales remain poor, Huliq. 〈http://www.huliq.com/10177/microsoft-kinect-shifts-10-million-units-game-sales-remain-poor〉. Accessed on Sept 8, 2012.

[2] Stein S Kinect. 2011: where art thou, motion?, CNET, http://reviews.cnet.com/8301-21539_7-20068340-1039〉. Accessed on Sept 8, 2012.

[3] Cacioppo JT, Priester JR, Berntson GG. Rudimentary determinants of attitudes. II: arm flexion and extension have differential effects on attitudes. J Pers Soc Psychol 1993;65(1):5–17.

[4] Meier BP, Robinson MD. Why the associations between affect and vertical position. Psychol Sci 2004;15(4):243–7.

[5] Tucker D. Towards a theory of gesture design, MFA thesis. University of Southern California; 2012.

[6] Lange B, Suma EA, Newman B, Phan T, Chang C-Y, Rizzo A, et al. 2011. Leveraging unencumbered full body control of animated virtual characters for game-based rehabilitation. In: HCI international. 2011. p. 243–52.

[7] Rizzo A, Lange B, Suma EA, Bolas M. Virtual reality and interactive digital game technology: new tools to address obesity and diabetes. J Diabetes Sci Technol 2011;5(2):256–64.

[8] Wang L, Hu W, Tan T. Recent developments in human motion analysis. Pattern Recognition 2003;36(3):585–601.

[9] Turaga P, Chellappa R, Subrahmanian VS, Udrea O. Machine recognition of human activities: a survey. IEEE Trans Circuits Syst Video Technol 2008;18(11):1473–88.

[10] Mitra S, Acharya T. Gesture recognition: a survey. IEEE Trans Syst Man Cybern Part C 2007;37(3):311–24.

[11] Yamato J, Ohya J, Ishii K. Recognizing human action in time-sequential images using hidden Markov model. In: IEEE computer vision and pattern recognition. 1992. p. 379–85.

[12] Black MJ, Jepson AD. A probabilistic framework for matching temporal trajectories: condensation-based recognition of gestures and expressions. In: European conference on computer vision, vol. I. 1998. p. 909–24.

[13] Hong P, Turk M, Huang TS. Gesture modeling and recognition using finite state machines. In: IEEE automatic face and gesture recognition. 2000. p. 410–15.

[14] Beard S, Reid D. MetaFace and VHML: a first implementation of the virtual human markup language. In: Workshop on embodied conversational agents. 2002.

[15] Kshirsagar S, Magnenat-Thalmann N, Guye-Vuillème A, Thalmann D, Kamyab K, Mamdani E. Avatar markup language. In: Eurographics workshop on virtual environments. 2002. p. 169–177.

[16] Carretero MDP, Oyarzun D, Ortiz A, Aizpurua I, Posada J. Virtual characters facial and body animation through the edition and interpretation of mark-up languages. Comput Graphics 2005;29(2):189–94.

[17] Laban R, Ulmann L. The mastery of movement 3rd edition. Macdonald and Evans; 1971.

[18] Foroud A, Whishaw IQ. Changes in the kinematic structure and non-kinematic features of movements during skilled reaching after stroke: a Laban Movement Analysis in two case studies. J Neurosci Methods 2006;158(1):137–49.

[19] Suma EA, Lange B, Rizzo A, Krum DM, Bolas M. FAAST: the flexible action and articulated skeleton toolkit. In: IEEE virtual reality. 2011. p. 245–6.

[20] 〈http://www.glovepie.org/〉.

[21] 〈http://www.kinemote.net/〉.

[22] Taylor RM, Hudson TC, Seeger A, Weber H, Juliano J, Helser AT. VRPN: a device-independent, network-transparent VR peripheral system. In: ACM virtual reality software and technology. 2001. p. 55–61.

[23] 〈http://www.3dvia.com/studio/〉.

[24] 〈http://www.worldviz.com/products/vizard/〉.

[25] Bowman DA, Kruijff E, LaViola JJ, Poupyrev I. 3D user interfaces: theory and practice. Addison Wesley Longman Publishing Co., Inc.; 2004.

[26] Usoh M, Arthur K, Whitton MC, Bastos R, Steed A, Slater M, et al. Walking, > walking-in-place > flying, in virtual environments. In: ACM conference on computer graphics and interactive techniques (SIGGRAPH). New York, New York, USA:ACM Press; 1999. p. 359–64.

[27] 〈http://projects.ict.usc.edu/mxr/faast/〉.