



- (51) **International Patent Classification:**
G05B 19/00 (2006.01)
- (21) **International Application Number:**
PCT/US2015/045478
- (22) **International Filing Date:**
17 August 2015 (17.08.2015)
- (25) **Filing Language:** English
- (26) **Publication Language:** English
- (30) **Priority Data:**
62/037,944 15 August 2014 (15.08.2014) US
- (71) **Applicant:** UNIVERSITY OF CENTRAL FLORIDA RESEARCH FOUNDATION, INC. [US/US]; 12201 Research Parkway, Suite 501, Orlando, FL 32826 (US).
- (72) **Inventors:** NAGENDRAN, Arjun; 100 Grand Avenue, Apt 1108, Oakland, CA 94612 (US). KELLY, Brian; 948 Dewinsberry Drive, Winter Garden, FL 34787 (US).
- (74) **Agents:** HJORTH, Beverly, E. et al.; Posternak, Blankstein & Lund LLP, 800 Boylston Street, Prudential Tower, Boston, MA 02199 (US).
- (81) **Designated States** (*unless otherwise indicated, for every kind of national protection available*): AE, AG, AL, AM, AO, AT, AU, AZ, BA, BB, BG, BH, BN, BR, BW, BY, BZ, CA, CH, CL, CN, CO, CR, CU, CZ, DE, DK, DM,

DO, DZ, EC, EE, EG, ES, FI, GB, GD, GE, GH, GM, GT, HN, HR, HU, ID, IL, IN, IR, IS, JP, KE, KG, KN, KP, KR, KZ, LA, LC, LK, LR, LS, LU, LY, MA, MD, ME, MG, MK, MN, MW, MX, MY, MZ, NA, NG, NI, NO, NZ, OM, PA, PE, PG, PH, PL, PT, QA, RO, RS, RU, RW, SA, SC, SD, SE, SG, SK, SL, SM, ST, SV, SY, TH, TJ, TM, TN, TR, TT, TZ, UA, UG, US, UZ, VC, VN, ZA, ZM, ZW.

- (84) **Designated States** (*unless otherwise indicated, for every kind of regional protection available*): ARIPO (BW, GH, GM, KE, LR, LS, MW, MZ, NA, RW, SD, SL, ST, SZ, TZ, UG, ZM, ZW), Eurasian (AM, AZ, BY, KG, KZ, RU, TJ, TM), European (AL, AT, BE, BG, CH, CY, CZ, DE, DK, EE, ES, FI, FR, GB, GR, HR, HU, IE, IS, IT, LT, LU, LV, MC, MK, MT, NL, NO, PL, PT, RO, RS, SE, SI, SK, SM, TR), OAPI (BF, BJ, CF, CG, CI, CM, GA, GN, GQ, GW, KM, ML, MR, NE, SN, TD, TG).

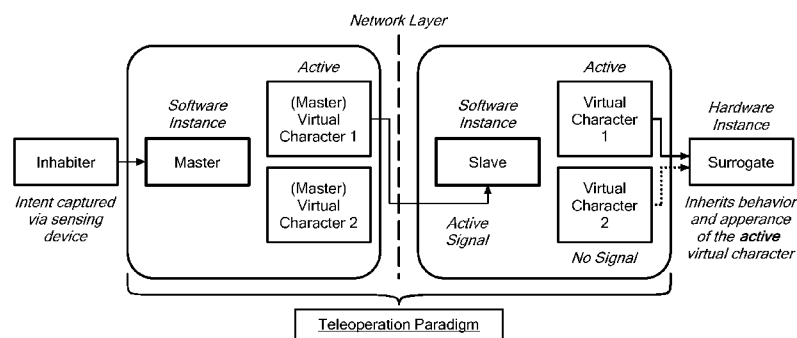
Declarations under Rule 4.17:

- *as to applicant's entitlement to apply for and be granted a patent (Rule 4.17(ii))*
- *as to the applicant's entitlement to claim the priority of the earlier application (Rule 4.17(iii))*
- *of inventorship (Rule 4.17(iv))*

Published:

- *with international search report (Art. 21(3))*

(54) **Title:** CONTROL INTERFACE FOR ROBOTIC HUMANOID AVATAR SYSTEM AND RELATED METHODS

**FIG. 3A**

(57) **Abstract:** A system and method for controlling a humanoid robot from a remote location are provided. One or more sensing devices are used to sense positions and movements of a user at an inhabiter station. A computing device generates a virtual character based upon the sensed positions and movements of the user and provides a plurality of outputs for controlling the humanoid robot based upon motion of the virtual character. The computing device includes a master controller to generate the virtual character and its behaviors and to transmit instructions to control motion of the humanoid robot to a slave controller located at the humanoid robot, providing smooth and natural motion of the humanoid robot. The method permits switching between multiple virtual characters in a single scene to control and alter the behavior of one or more humanoid robots located in several geographic locations.

TITLE

Control Interface For Robotic Humanoid Avatar
System and Related Methods

CROSS REFERENCE TO RELATED APPLICATIONS

This application claims priority under 35 § 119(e) of U.S. Provisional Application No. 62/037,944, filed on August 15, 2014, entitled “Control Interface for Robotic Humanoid Avatar System and Related Methods”, the disclosure of which is hereby incorporated by reference.

STATEMENT REGARDING FEDERALLY SPONSORED
RESEARCH OR DEVELOPMENT

N/A

BACKGROUND

Previous research has addressed the concepts of inhabiting a humanoid robot through an intuitive user interface. In (Trahianias, Argyros, et al., 2000), the authors’ presented a system, “Tourbot”, which enabled users to remotely explore a museum through a robotic avatar. Instead of allowing the user to directly control the robot’s motion, users were given a set of high level commands such as what exhibit to view, promoting ease of use. Additionally, (Marn, Sanz, et al., 2005) created a highly general control system for manipulating a robot arm that allowed both high level voice commands and low level remote programming through Java. Another project (Bell, Shenoy, et al., 2008) explored a noninvasive brain computer interface using electroencephalography (EEG). Because of the large amount of noise associated with EEG, the authors focused on robot control through simple high level commands. This work focused on indirect control, in which individual joints are not mapped onto a robot.

In other research, (Tachi, Komoriya, et al. 2003) presented a “telexistence cockpit” capable of remotely controlling a robot. The user interface consisted of two “master arms,” a head mounted display (HMD), and a motion base located in the cockpit. A user would manipulate the master arms to control the slave robot arms. Additionally, the motion base was used to communicate robot translation and the HMD adjusted the view perspective.

Other researchers have explored how human motion can be realistically mapped onto a robot. In (Pollard, Hodgins, et al., 2002), the authors prerecord human motion and map the

resulting animation onto a robot. Because a human has many more degrees of freedom than the robot, the human body was divided into segments and the overall rotations from these segments were mapped onto the robot. Furthermore, the authors scaled the original animation to fit inside the range of motion of the robot's joints.

5 Another related paper (Nakaoka, Nakazawa, et al., 2003) addressed the more specific problem of mapping a human dance onto a robot surrogate. The researchers motion captured a human dance performance, and the resulting animation was then divided into a set of important postures for a robot to assume. The authors were interested in full body motion, and as a result, had to modify the captured data to ensure the robot would maintain its
10 balance.

The previous approaches are limited by the quality of the motion capture software, and in some cases may not replicate the motion as desired. In (Calinon and Billard, 2007), the authors presented a method of teaching a robot a gesture through a two phase process. In the first phrase, a human coach, wearing several motion sensors, performed a gesture while the
15 robot observed. In the second phase, the coach was allowed to grab and move the robot arms to refine the gesture.

SUMMARY OF THE INVENTION

The present invention relates to a user control interface to generate a virtual character
20 and map the virtual character's behavior to a remotely located physical humanoid robot that functions as a user's surrogate. More particularly, a control interface is provided using a master-slave architecture to allow a user to interact with elements in a remote environment via a physical entity, a humanoid robot, located in the remote environment. The user, who may also be called an inhabiter, is able to inhabit the robot by controlling a virtual character
25 via a master controller. The virtual character behaves as a master that sends control signals to a remote slave (robot) coupled to the master controller. The system can employ simple, inexpensive, and noninvasive user interface devices that are operated by the user as inputs to a master controller that generates animations of the virtual character based on the inputs. This approach obviates the need to directly link the joints of the user and the robot; instead, it
30 relates the virtual character to the robot. Position, velocity and torque profiles are determined to reduce the discrepancy between the virtual character and the robot. This approach also addresses the limited degrees of freedom and range of motion of the robot relative to the degrees of freedom available to a virtual character. An increase or reduction in the degrees of freedom can be performed to preserve motion profiles between a virtual character and a

robot. This approach does not require the capture of human motion through a set of motion sensors. The use of an animation for a virtual character eliminates the need for training the robot.

Other aspects of the method and system include the following:

- 5 1. A system for controlling a humanoid robot from a remote location, the system comprising:
 - one or more sensing devices to sense positions and movements of the user; and
 - a computing device in communication with the one or more sensing devices, the computing device comprising one or more processors and memory, and including computer-
 - 10 executable instructions stored in the memory that, upon execution by the one or more processors, generate a virtual character based upon the sensed positions and movements of the user and provide a plurality of outputs for controlling the humanoid robot based upon motion of the virtual character.
- 15 2. The system of item 1, wherein the one or more processors of the computing device include a master controller operative to generate the virtual character and to translate motion of the virtual character into instructions to control motion of the humanoid robot, the master controller in communication with a slave controller located at the humanoid robot to transmit the instructions to the slave controller.
- 20 3. The system of any of items 1-2, wherein the master controller includes a graphics rendering engine and a physics engine including instructions operative to translate motions of the user into a three-dimensional animation of the virtual character.
4. The system of any of items 1-3, wherein the virtual character is generated by creating an animation from the sensed positions and motions of the user.
5. The system of any of items 1-4, wherein the animation comprises an interpolation
- 25 between key frames or between time-positions or a combination thereof.
6. The system of any of items 1-5, wherein the computing device is operative to provide a real time mapping between motions of the user and motions of the humanoid robot.
7. The system of any of items 1-6, wherein the computing device is operative to determine a torque or force to apply to a joint of the humanoid robot based on one or more
- 30 accelerations of one or more joints of the virtual character.
8. The system of any of items 1-7, wherein a number of degrees of freedom of motion of the virtual character is greater than a number of degrees of freedom of motion of the humanoid robot.

9. The system of any of items 1-8, wherein the virtual character has a number of joints, each joint corresponding to a joint of the user, and the computing device is operative to assign a uniform coordinate system to each joint of the virtual character.

10. The system of any of items 1-9, wherein the computing device is operative to store in memory data comprising at least one of pitch, yaw, and roll orientation for each joint, the joints including at least one of a shoulder joint, an elbow joint, a wrist joint, a finger joint, a neck joint, a waist joint, a hip joint, a knee joint, and an ankle joint.

11. The system of any of items 1-10, wherein the virtual character has a number of joints, each joint corresponding to a joint of the user, and the computing device is operative to store an orientation of each joint of the virtual character as a quaternion, and includes instructions to transform the orientation of each joint of the virtual character to an orientation of each joint of the humanoid robot stored as an Euler angle.

12. The system of any of items 1-11, wherein the transform instruction comprises mapping the set of orientations for each joint of the virtual character to the set of orientations for each joint of the humanoid robot.

13. The system of any of items 1-12, further comprising constructing a new set of orientations for each joint of the virtual character by applying a rotation operation to each joint.

14. The system of any of items 1-3, further comprising mapping multiple joints of the virtual character to a single joint of the humanoid robot.

15. The system of any of items 1-14, wherein the mapping comprises generating a reduced set having a reduced number of orientations of the virtual set equal to the number of joints of the humanoid robot, the reduced set including a quaternion equal to a top rotation of a hierarchy of joints and a rotation of a leaf joint of the humanoid robot.

16. The system of any of items 1-15, further comprising converting the reduced set of orientations to a converted set of orientations of the joints of the humanoid robot utilizing a Denavit-Hartenberg convention applicable to a corresponding joint of the humanoid robot.

17. The system of any of items 1-16, further comprising extracting an Euler angle for the corresponding joint of the humanoid robot.

18. The system of any of items 1-17, wherein:

the computing device provides a plurality of selectable micro-poses on the display and receives a selected micro-pose via input from the user; and

the computing device causes the humanoid robot to perform the selected micro-pose.

19. The system of any of items 1-18, wherein the one or more sensing devices comprise a physical input device coupled to the computing device to input physical motions of the user to the computing device.
20. The system of any of items 1-19, wherein the physical input interface comprises a joystick device, a magnetic tracking device, a motion sensing device, an image-based motion recognition device, a depth-based motion recognition device, an infrared (IR) motion sensing device, or combinations thereof.
21. The system of any of items 1-20, wherein the one or more sensing devices comprise one or more reflective markers positioned on a head of the user, and an infrared (IR) image sensor coupled to the computing device to determine a head position of the user based upon tracking of the one or more reflective markers.
22. The system of any of items 1-21, wherein the computing device controls head motion of the humanoid robot based upon a head position of the user.
23. The system of any of items 1-223, wherein the computing device controls arm motion of the humanoid robot based upon an arm position of the user.
24. The system of any of items 1-23, wherein the computing device provides corresponding pluralities of outputs for controlling a plurality of humanoid robots in different locations.
25. The system of any of items 1-24, wherein the computing device provides a menu stored in the memory and displayable on a display for selecting a current humanoid robot from the plurality thereof.
26. The system of any of items 1-25, wherein the computing device provides a plurality of virtual characters configurations stored in memory and receives a selected virtual character configuration from the user.
27. The system of any of items 1-26, wherein the computing device is operative to switch between selected virtual characters in real time upon input of an updated virtual character selection from a user.
28. The system of any of items 1-27, wherein any of the plurality of virtual characters can be used to control the humanoid robot.
29. The system of any of items 1-28, wherein the computing device includes stored in the memory a plurality of motions of the virtual character, each motion representative of a user behavior.

30. The system of any of items 1-30, further comprising a display coupled to the computing device to provide a third person view of the virtual character as a representation of a user and a first person view from the at least one humanoid robot;

31. The system of any of items 1-30, wherein the display comprises a first display device coupled to the computing device to provide the third person view of the virtual character and a second display device coupled to the computing device to provide the first person view from the humanoid robot.

32. The system of any of items 1-31, wherein the computing device includes instructions operative to display the third person view of the virtual character and the first person view from the humanoid robot as separate screens or windows on one display device or on two display devices.

33. The system of any of items 1-33, wherein the system further includes a microphone to record audio from the user in real-time and the computing device includes instructions to transmit recorded or live audio to the humanoid robot synchronized with the sensed positions and movements of the user.

34. A humanoid robot system comprising:
the system for controlling a humanoid robot from a remote location of any of items 1-33; and
a humanoid robot in communication with the control system.

35. The system of any of items 1-34, wherein the computing device comprises a master controller, comprising a processor and memory, to generate the virtual character and to translate motion of the virtual character into instructions to control motion of the humanoid robot, and

further comprising a slave controller, comprising a processor and memory, to receive instructions from the master controller, the slave controller located at and in communication with the at least one robotic humanoid avatar.

36. The system of any of items 1-35, wherein the humanoid robot includes one or more joints, an actuator connected to each joint to control rotation of the connected joint, and the computing device is operative to transmit instructions to the slave controller to actuate rotation of a selected joint via the connected actuator.

37. A method for controlling a humanoid robot from a remote location, comprising:
sensing positions and movements of a user;
storing sensed positions and movements of the user as a virtual character in a computing device;

sending outputs to the humanoid robot to control motion of the humanoid robot based on the sensed positions and movements of the user.

38. The method of item 37, further comprising generating the virtual character at a master controller of the computing device, translating motion of the virtual character into instructions to control motion of the humanoid robot, and transmitting the instructions to a slave controller located at the humanoid robot.

39. The method of any of items 37-38, further comprising creating an animation of the virtual character from the sensed positions and motions of the user.

40. The method of any of items 37-39, further comprising mapping in a real time motions of the virtual character to motions of the humanoid robot.

41. The method of any of items 37-40, further comprising determining a torque or force to apply to a joint of the humanoid robot based on one or more accelerations of one or more joints of the virtual character.

42. The method of any of items 37-41, further comprising reducing a number of joints of the virtual character to a reduced set corresponding to a number of joints of the humanoid robot.

43. The method of any of items 37-42, further comprising storing an orientation of a joint of the virtual as a quaternion, and transforming the quaternion to an orientation of a joint of the humanoid robot as a Euler angle.

44. The method of any of items 37-43, further comprising mapping multiple joints of the virtual character to a single joint of the humanoid robot.

45. The method of any of items 37-44, further comprising sensing positions and movements of the user by receiving input signals from a physical input device, the physical input device comprising one or more of a joystick device, a magnetic tracking device, a motion sensing device, or an infrared motion sensing device.

46. The method of any of items 37-45, further comprising sensing positions and movements of the user by tracking one or more reflective markers positioned on a head of the user with an infrared image sensor to determine a head position of the user.

47. The method of any of items 37-46, further comprising controlling arm motion of the humanoid robot based upon arm motion of the user.

48. The method of any of items 37-47, further comprising controlling head motion of the humanoid robot based upon head motion of the user.

49. The method of any of items 37-48, further comprising providing a plurality of virtual character configurations and receiving a selected virtual character from a user.

50. The method of any of items 37-49, further comprising switching between selected virtual characters in real time upon input of an updated virtual character selection from a user.

51. The method of any of items 37-50, wherein any of the plurality of virtual characters can be used to control the humanoid robot.

5 52. The method of any of items 37-51, further comprising sending outputs to a second humanoid robot to control motions of the second humanoid robot based on the sensed positions and movements of the user.

DESCRIPTION OF THE DRAWINGS

10 The invention will be more fully understood from the following detailed description taken in conjunction with the accompanying drawings in which:

Fig. 1 is a schematic illustration of a control interface system and method for controlling a humanoid robot;

Fig. 2 is a schematic illustration of an embodiment of an inhabitant's station;

15 Fig. 3A is a schematic illustration of an embodiment of a master/slave paradigm of the control interface for controlling a humanoid robot from a remote location;

Fig. 3B is a schematic illustration of an embodiment of a hardware instantiation of a master/slave paradigm;

Fig. 4 is a schematic illustration of a virtual character;

20 Fig. 5 is a schematic illustration of a humanoid robot;

Fig. 6 is an illustration of testing of a magnetic tracking device;

Fig. 7 is a schematic illustration of a further embodiment of an architecture of a control interface for controlling a humanoid robot at a remote location;

25 Fig. 8A is a schematic illustration of a virtual character displayed in an environment based on a remote environment at which a humanoid robot is located;

Fig. 8B is a schematic illustration of a humanoid robot in a real environment; and

Fig. 9 is a schematic illustration of a further application of a control interface for controlling a humanoid robot at a remote location.

30 DETAILED DESCRIPTION OF THE INVENTION

This application incorporates by reference the entire disclosure of U.S. Provisional Application No. 62/037,944, filed on August 15, 2014, entitled "Control Interface for Robotic Humanoid Avatar System and Related Methods".

A surrogate can be defined as a substitute or stand-in for a person in a context-specific role. Human surrogates can take several forms, ranging from a virtual character in a virtual world to a robotic or fully physical representation of a specific person in a real environment. The surrogate provides access to an environment that is otherwise not readily accessible to the person controlling the surrogate for reasons such as a physical disability that prevents the person from being present in the environment, maintaining anonymity, lack of time or resources to be physically present in the environment, accessing relatively dangerous locations, and the like.

The present invention relates to control of a humanoid robot that provides a physical representation in a real environment of a surrogate of a person located remotely from the robot. Referring to Fig. 1, the system 10 provides an inhabitier 14 (a user) the ability to control a virtual character 16 that in turn controls a robot 18. A robot as controlled herein can offer advantages to the user controlling it (the inhabitier) by supporting bi-directional communication involving both verbal and non-verbal cues to another person interacting with the robot at the robot's location. Such humanoid robots can have several applications, including training, rehabilitation, and education. So that people can interact naturally with the robot, the robot can be capable of (i) appearing natural without destroying the beliefs of the people interacting with it (plausibility) and (ii) eliciting behavioral responses in the people with whom it interacts. For instance, the robot can exhibit natural looking motion in the form of kinematics and facial expressions as well as auditory stimuli to evoke an empathetic response. The robot can exhibit dynamic stability if a need for contact or touch between the robot and a real person arises.

Toward these ends, a control interface is provided with minimally invasive sensing that captures and transmits an inhabitier's intent, via, for example, head and arm motions, so the inhabitier can take control of movement of the robot. The paradigm supports a first person view of an environment via the robot. This allows an inhabitier to see "through the eyes" of the robot and perform actions in the remote environment appropriately. The paradigm includes algorithms that interpret the data received over the network (with a chance of packet-loss) to create smooth and natural motion profiles at the robot's end. It reduces cognitive loads on the inhabitier and reduces the amount of network traffic transmitted during inhabiting. Such robots can be used in all walks of life, including training (for instance, when the inhabitier is a subject matter expert (SME) offering advice to a remotely-located group of people), therapy (where a patient converses with a therapist in a distant geographic location

via the robot), or education (where a student attends a classroom lecture via a robot, enabling the student to interact naturally with other classmates.

The control interface also allows a humanoid robot to seamlessly switch between a number of different surrogates, automatically adopting behaviors associated with each surrogate, as controlled by the inhabitant. Such multiple surrogate embodiments can be used, for example, in cultural awareness training. The control interface can also support hybrid control of a robot using both artificial intelligence and direct human control.

Humanoid robots that can be inhabited using the control interface described herein are known and are commercially available. Such humanoid robots include a head, typically with a camera to provide a video feed of what the robot sees. Some humanoid robots can also include features such as a vacuumformed face and a rear-projected head allowing the robot to change its facial appearance. Humanoid robots also typically include a torso supporting a neck that mounts the head as well as arms and hands. Legs or other mobility mechanisms (such as wheels or whegs, i.e., wheeled legs) can also be provided in some robots. Joints provide rotation—yaw, pitch, and/or roll—of the various body parts to effect movement. Any suitable actuator, such as electric servo motors or pneumatic or hydraulic actuators, can be used to effect rotation of the joints. Some robots use a combination of electric and pneumatic actuation with passive compliance elements paired in an antagonistic manner to support highly natural looking motion. For example, pneumatic fluidic muscles in the form of a fiber cylinder that contracts in length and expands in diameter when filled with air can be used to provide naturalistic motion in arms or legs of a robot. For instance, a robot can be fitted with fluidic muscles to control arm joints, such as a shoulder joint (roll, pitch, and yaw), an elbow joint (pitch), and a wrist joint (pitch). Independent servo motors can be used to control the head (roll, pitch and yaw) and the torso (roll, pitch and yaw). Fingers can be actuated in a binary manner (close/open) using a directional control valve with three air ports, and two positions (commonly referred to as a 3/2 pneumatic valve). The natural low impedance characteristics of the pneumatic actuators make them easily back-drivable.

Certain humanoid robots (such as the Robothespian) may employ a user interface that permits graphical user interface (GUI)-based programming of certain pre-defined robot behaviors. This mode is well suited for narration and recorded animations supplied by the present system. Each individual actuator on the robot is also controllable via an application program interface (API), thereby permitting real-time control of the robot. The present control interface interacts with the actuators of the robot to realize and transmit the

inhabiter's intent accurately in real-time, while adapting the closed-loop response of the robot's hybrid actuation mechanism to faithfully represent this action.

In some embodiments, the control interface interacts with the robot to control the arms, hands, head, neck, and torso, whereas the legs are not used. Embodiments in which the legs are not used may be suitable for environments where other humans may be present and it is desired to limit movement of the robot for safety concerns. In other embodiments, the legs of the robot can be used as well, allowing the robot to move about its environment.

One example of a commercially available robot that can be used to support human interaction is the Robothespian from Engineered Arts Limited, UK. This robot employs fluidic muscles as described above. The upper torso of the Robothespian has a total of 24 independently controllable degrees of freedom. The fingers of each hand can be opened and closed, although the thumbs are fixed. The force exerted when the fingers are closed and the lack of pronation/supination of the thumb makes it unsuitable for gripping objects. If desired, the legs can also be actuated using proportional pneumatic valves (fluidic muscles).

A user who remotely operates a humanoid robot can also be referred to as an "inhabiter". The inhabiter has the ability to view the remote environment through the robot's eyes and perform actions in the remote environment by transmitting his intent to the robot. To facilitate this, the inhabiter occupies a location referred to as an inhabiter station. See Fig. 2. The inhabiter station is an environment that includes a plurality of sensing devices that allow an inhabiter's intent to be captured via various movements, such as head and arm movements. This data forms what is referred to as "sensory affordances" of the inhabiter station. The sensory affordances are the set of possible joint movements that a user can make using an input device. A set, $A_{virtual}$, of affordances can include, for example, at least one of pitch, yaw, and roll orientation for each joint, the joints including at least one of a shoulder joint, an elbow joint, a wrist joint, a finger joint, a neck joint, a waist joint or a hip joint, a knee joint, and an ankle joint. In order to support portability and instantaneous ability to inhabit a surrogate from any location, the set $A_{virtual}$ of sensory affordances can be kept to a minimum.

The paradigm of the control interface is that of a master-slave system in which any robot (slave) is driven by its virtual counterpart (master). See Figs. 3A and 3B. The system includes hardware, including a computing device having a master controller and memory, displays, and sensing devices, to support inhabiting and a control system including algorithms and other software to control the virtual character (master). A slave controller (at the robot) in communication with the master controller via a network

layer 90 receives instruction from the master to reproduce the virtual character's behavior faithfully by the robot hardware instance in the remote location, thereby serving as the inhabitant's surrogate.

While it is possible to directly map an inhabitant's motion to a robot, this generally requires motion sensing on every joint on the kinematic chain. This approach applies a one-to-one mapping in which every joint of both the inhabitant and the robot is tracked and requires that errors between the joints across the inhabitant and the robot be minimized. It requires "literal" motion on the part of the inhabitant and a calibrated kinematic configuration of the robot to faithfully reproduce this motion by the robot. Also, to facilitate first-person control, a first-person view of the environment through the eyes of the robot must be provided. Assuming that tracking and motion can be faithfully reproduced with minimum latency, when the inhabitant moves his head, the robot correspondingly moves its head. A screen with a video feed from a camera at the robot's eyes can allow the inhabitant to look around in the remote environment. However, when the inhabitant looks upwardly or too far out to the side, he is unable to see the screen in front of him, if the screen or display surface is not immersive. If the display is immersive, such as a head-worn display, any latency or delay in the update between the motion of the inhabitant and the motion of the robot may also result in motion-sickness, since the inhabitant's physical motion is not synchronous with the updated video feed from the camera at the robot. An immersive display also reduces the portability and non-intrusive sensing aspects of the inhabiting paradigm.

The present system overcomes these drawbacks by employing a master-slave paradigm in which the inhabitant controls a virtual character instance 36, which can be viewed by the inhabitant 32 on a screen display 38. (See Fig. 2.) In addition, a view through the eyes of a robot 42, for example, of a participant 46, is also provided on a screen display 44. The control interface provides a mapping of the motion of the virtual character to the robot despite a varying number of degrees of freedom. This is achieved by a combination of a coordinate system transform (Equations 2-4 below) to map each joint of the virtual character coordinate space to the joints of the robot in the robot's coordinate space and a reduction transform (Equation 5 below) for instances when the master virtual character has more degrees of freedom at a specific joint than the slave robot, as described further below. The master controller then transmits to the slave controller at the robot standard control instructions comprising a set of joint orientations described, for example, as a set of angular accelerations, to control torques on the joints by the appropriate actuator to effect a desired degree of rotation without affecting other joints in the kinematics chain (Equation 6 below).

In this manner, motion of the robot's joints can be effected smoothly and naturally in real time.

A virtual character is created at the master controller that is representative of the user to be the user's surrogate at the robot. The virtual character can be designed to have sufficient degrees of freedom to kinematically represent all motions of humans. This character is then rigged and animated with a set of essential poses that contain a combination of key-frame and time-dependent blending routines. The poses can be chosen based on the required functionality of the surrogate. Also, some commercially available robots can incorporate the virtual character's appearance by virtue of possessing a rear-projected head with vacuum surface.

The master controller can include a graphics rendering engine to generate an animation of the virtual character. The graphics rendering engine's animation system uses built-in routines to interpolate between the key-frames and time-positions depending on the chosen blending method. In essence, this becomes a free inverse-kinematic solver for the real-world surrogate. The graphics rendering engine can also be combined with a physics engine to solve for the inverse dynamics of the system allowing for decoupled joint control on the surrogate. For instance, given the accelerations at each joint as determined by the master controller using the physics engine, controllable torques on the robot joints are computed using the standard Recursive Newton Euler formulation of Inverse Dynamics:

$$\tau = I(\theta) \ddot{\theta} + c(\dot{\theta}, \theta) \dot{\theta} + g(\theta) \quad (1)$$

where τ is the vector of joint torques, θ is the n-dimensional joint-space position vector, $\dot{\theta}$ is the n-dimensional joint-space angular velocity vector, $\ddot{\theta}$ is the n-dimensional joint-space angular acceleration vector, $I(\theta)$ is an $n \times n$ symmetric inertia matrix, $c(\dot{\theta}, \theta)$ is the $n \times n$ tensor or velocity coupling matrix, where n is the number of joints, and g is the $n \times 1$ gravity-dependent vector. The robot's joints can then be controlled, for example, by driving a DC motor, a pneumatic or hydraulic linear actuator, or actuating a fluidic muscle to produce the desired torque or force. Software modules or routines to control a robot based on inverse dynamics are known and readily available, as are various graphics rendering engines and physics engines.

Inhabiters can also map a character's motion (master) to their intent during a calibration phase (described further below), which gives each inhabiter the ability to modify the controls to the real-world surrogate while keeping cognitive loads to a minimum.

The robot is considered a slave during this operation. Robots can have a varying number of degrees of freedom depending on their complexity. Some of these degrees of freedom can be manipulable in the continuous-time domain while others can be discrete. Together, this forms the set A_robot of “robot affordances.” A co-ordinate transform function maps the “sensory affordances” of an inhabiter to the “robot affordances” via the virtual character (master) in real-time. In some instances, this is a one-one mapping, while in others, a difference transform between relevant joint space variables is applied to achieve the required configuration.

The need for a coordinate transform function stems from differences in coordinate frames between the virtual character and the robot. In the case of the virtual character, a three dimensional engine can be employed to assign a uniform coordinate system to each joint. However, the robot uses the Denavit–Hartenberg (DH) convention for its joint space parameters. The DH convention standardizes coordinate frames for links in a kinematic chain. More formally, a function is provided that maps $A_virtual$ to A_robot as defined below.

$$A_virtual = \{vx_i, vy_i, vz_i, vw_i \mid i = 1 : n\} \quad (2)$$

$$A_robot = \{rx_j, ry_j, rz_j \mid j = 1 : m\} \quad (3)$$

where n is the number of joints for the virtual character, m is the number of joints for the robot, (vx_i, vy_i, vz_i, vw_i) is the orientation of joint i in the virtual character stored as a quaternion, and (rx_i, ry_i, rz_i) is the orientation of joint j in the robot stored using Euler angles.

In order to create this mapping, an initialization pose from $A_virtual$ is equated to an initialization pose from A_robot . Upon start up, a robot assumes an initialization pose, generally looking forward. The orientation of the actuators and joints of the robot’s initialization pose are specified by the robot’s manufacturer, for example, in the robot’s documentation. Similarly, the virtual character assumes an initialization pose. These poses must be equated so that motion of the virtual character can be coordinated with motion of the robot. To do this, for each joint in the virtual character, a new joint is constructed that is equal to the identity quaternion when $A_virtual$ represents the initialization pose. These new joints are children of their corresponding virtual character joint causing them to follow any

animations that take place. Essentially, a new set of affordances is defined by applying a rotation to each joint.

$$A_equated = A_initialPose^{-1}A_virtual \quad (4)$$

5

where $A_equated$ is the new set of affordances and $A_initialPose$ is $A_virtual$ when the virtual character is in the initialization pose. To be clear, all operations on sets are performed on each set element individually.

In some cases, such as for the robot's head, this single step is enough to convert a joint from the virtual character; however, for most cases, additional transformations are required. The second step involves converting a joint into the appropriate space. When the robot and virtual character share the same joint, the virtual character's joint orientation must be transformed to local space. However, sometimes the conversion is more complex because the robot has fewer joints than the virtual character. If this occurs, multiple joints in a parent and child hierarchy must be mapped to a single joint by computing the difference between the child (or leaf) joint's rotation and the local rotation of the entire hierarchy. As an example, illustrated by reference to Figs. 4 and 5, the master character 92 has two joints, J1 at the elbow (the parent joint) and J2 at the wrist (a child or leaf joint), that control the orientation of the palm. J1 contains an "arm twist" and J2 contains a "wrist pitch." The robot 94, however, does not have a function equivalent to an "arm twist." Instead, it only has one joint K1 to control "wrist orientation." A reduction transform, as shown in Equation 5 below, computes the local transform K1 on the robot as a function of the transforms J1 and J2 on the virtual character such that there is a minimal difference in the final pose achieved between the two. This step essentially maps $A_equated$ to a new set, $A_reduced$, with the cardinality of A_robot .

$$A_reduced = \{T_j^{-1}L_j \mid (j = 1 : m) \wedge (T_j \in A_equated) \wedge (L_j \in A_equated)\} \quad (5)$$

where m is the number of joints in the robot, T_j is a quaternion equal to the top rotation of the hierarchy (J1 in the example above), and L_j is the rotation of the leaf joint (J2 in the example above).

The next step involves converting $A_reduced$ to the DH convention used for that particular joint. Specifically, the coordinate space of the joint in $A_reduced$ is transformed to one where all of the axes align with the DH convention for the joint. Due to differences in the

handedness of the coordinate systems, some axes may need to point in the opposite direction. Ultimately, the direction of the axes is irrelevant because the direction of rotation is set to match the DH convention. More formally,

$$A_converted = \{DHSpace_j A_reduced_j \mid j = 1 : m\} \quad (6)$$

where m is the number of joints in the robot and $DHSpace$ is a set of m rotations defining the coordinate system of each joint.

Once $A_converted$ is established, its Euler angles can be extracted. Overall, there are many different ways to express an Euler angle varying in which order the rotations are applied and the direction of the rotation. For each joint in the robot, the rotation order and rotation direction must be defined. The resulting Euler angles denote the rotational difference between the virtual character's current pose and the robot's initialization pose. From these values, A_robot can be found and clamped so that its angles lie within the range of the robot's joints.

Exemplary embodiments of software routines that can perform the calculations of the above Equations are attached at the Appendix. It will be appreciated that other routines and algorithms can be used to perform the calculations herein.

Referring to Figs. 2, 3A, and 3B, the system includes an inhabiter station to support inhabiting the humanoid robot. The inhabiter station can be located remotely from the humanoid robot and can be portable so that a user can control the humanoid robot from any location. The inhabiter station includes one or more sensing devices 34 in communication with a computing device 71. A sensing device can be a physical input device to input physical positions and motions of the inhabiter to the computing device. The computing device includes one or more processors and memory for storing machine-readable instructions that, upon execution by the one or more processors, provide a plurality of outputs to the humanoid robot for controlling the at least one humanoid robot based upon the sensed positions and movements of the user and the input from the user. The station includes at least one and in some embodiments two display devices in communication with the computing device. One display device 38 displays an image of the virtual character. The other display device 44 displays a view seen by the humanoid robot, for example, a view of another person 46 with whom the inhabiter 32 is interacting via the virtual character 36. In another embodiment, the computing device can display two split windows, in which one window

shows a view of the master virtual character that is being manipulated by the inhabiter while the second window shows a first-person view from the robot's perspective.

In one embodiment, a sensing device 34 can be a motion and orientation detection device that detects the position and orientation of a user, such as are used in the video game industry. Suitable devices include a joystick-based controller, a hand controlled magnetic tracking device (such as the Hydra game controller from Razer), a motion sensing device, an image-based motion recognition device (such as a standard web camera), a depth-based motion recognition device (such as a laser scanner or the Microsoft KINECT[®]), or an infrared (IR) motion sensing device (such as the head tracking device TrackIR from NaturalPoint).

Multiple sensing devices can be used. More than one tracking device can be used with the inhabiter station. For example, referring again to Fig. 2, two magnetic tracking devices 52, one for each user's hand, with an associated base 54 can be combined with a reflective marker or IR signal emitter 56 worn on a user's head that is detectable by an IR sensor 58 located in a region to detect motion of the user's head. For example, the IR sensor can be located on top of the display device 38 that displays the virtual character.

In one embodiment, a magnetic tracking device includes a base 54 and two hand-held controllers 52. Each controller has six buttons, a control stick, and an analog trigger. The tracking device tracks the three-dimensional position and orientation of both controllers with respect to the base, providing the user with six degrees of freedom on each controller. Internally, the device software assigns a controller two sets of positions and rotations, one for each hemisphere around the base. Because the device sensors are unable to determine which hemisphere the controller lies inside, a brief calibration procedure is required upon initialization that involves pointing both controllers toward the base. Using this information, the correct hemisphere can be found; however, the user will be restricted to a single hemisphere during operation.

In order to evaluate the accuracy of the tracking device, several tests were performed using a Hydra Razer. These tests evaluated the ability of the device to replicate the same coordinates when positioned at the same location in the environment. To do this, three points were selected nearby the base of the tracking device and their distance from the base was measured. Next each controller was placed at the predefined point and the coordinates were recorded for two seconds. In total, each controller was tested in this fashion three times for each point. See Fig. 6. The table below shows the standard deviation of each controller at each point across the three tests. These standard deviations were converted from the tracking

device space to inches by computing a scaling factor to center the data points onto their ground truth value.

Trial	Coordinates (inches) [X, Y, Z]	Std Dev (left controller) [X, Y, Z]	Std Dev (right controller) [X, Y, Z]
1	[-22.5, -4, 10]	[0.0403, 0.0219, 0.0442]	[0.0664, 0.0211, 0.0205]
2	[25.25, 4, 22.5]	[0.2556, 0.1509, 0.2796]	[0.1265, 2.3619, 0.2637]
3	[0.1, -10, 18.5]	[0.0108, 0.0361, 0.0821]	[0.0040, 0.0199, 0.0245]

In one embodiment of an imaging sensor, an optical tracking device is provided. The tracking device can have any suitable frame rate, such as 120 fps, and field of view, such as 51.7°. An inhabiter can wear, for example, a cap fitted with a retro-reflective pattern whose geometry is known. The position and rotation of the inhabiter's head, in 6 degrees of freedom, can be computed using information about the spacing and size of the "blobs" in the image plane. This information can be used to infer the inhabiter's head motion. Commercial imaging sensor systems are available. One suitable system is TrackIR.

In one embodiment, an inhabiter uses both an imaging sensor and a tracking device to control the virtual character. Using the imaging sensor, a one-to-one mapping is created between the orientation of the user's head and the rotation of the virtual character's neck joint. All other joints are indirectly controlled through the tracking device by way of micro-poses. A micro-pose is a simplified animation that contains two key frames or two distinct poses. As a result, the beginning of the micro-pose animation is the first key frame while the end of the micro-pose animation is the last key frame. Any point in the middle of the animation is a mixture of both poses. By giving the inhabiter the ability to select micro-poses and their time positions, a highly expressive real time user interface can be created without the need to specify individual joint orientations.

The simplest method for controlling the virtual character restricts the inhabiter to a single micro-pose at every time instant. Each micro-pose that the character can assume is mapped to two three-dimensional positions representing the positions of the left and right tracking device controllers. This mapping is generated during a calibration procedure discussed further below. Consequently,

$$calibration = \{lx_i, ly_i, lz_i, rx_i, ry_i, rz_i \mid i = 1 : n\} \quad (7)$$

where n is the number of micro-poses, i is an integer id for a micro-pose, and (lx_i, ly_i, lz_i) and (rx_i, ry_i, rz_i) represent the positions of the left and right controllers of the tracking device respectively. Once this mapping is established, the system measures the euclidean distance between the current tracking device controller positions and each of the micro-poses. From these measurements, each micro-pose is scored as shown below.

$$dist(x_1, y_1, z_1, x_2, y_2, z_2) = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2 + (z_1 - z_2)^2} \quad (8)$$

$$score = dist(lx_i, ly_i, lz_i, cx, cy, cz) + dist(rx_i, ry_i, rz_i, cx, cy, cz) \quad (9)$$

where (cx, cy, cz) represent the current position of the tracking device. If the inhabiter positions the tracking device controllers such that the lowest scoring micro-pose is constant and different from the current micro-pose for a period of time, then the system will automatically switch the current micro-pose to the lowest scoring micro-pose. The transition however, is not instantaneous. Instead, the new pose's weight is increased at a set rate while the old pose's weight is decreased at the same rate. In this manner, micro-pose transitions appear smooth and seamless to the inhabiter and anyone engaged with the robot. Furthermore, there is a short time delay from when the inhabiter assumes a pose with the tracking device and when the micro-poses begin to blend. This delay prevents the user from unintentionally assuming a micro-pose calibrated to lie in between the current pose and the desired pose.

The time position of the assumed micro-pose is controlled by the roll of the tracking device controllers. This allows the inhabiter to move the character without changing the micro-pose. To promote smooth animations, simple exponential smoothing is applied to the roll values, and the rate of change is clamped to be under a set value.

Alternatively, it is also possible to bind a micro-pose to a single hand making it possible for multiple micro-poses to be assumed at the same time. In order to generalize the prior control algorithm, the notion of pose types is introduced. Overall, there are four different pose types: "both," "left," "right," and "main." Micro-poses marked as "both" are controlled as in the previous algorithm requiring both hands close to the correct position. Conversely, the "left" and "right" pose types only depend on the position of a single controller. Additionally, these animations are set to only manipulate the joints on one side of

the body allowing the combination of “left” and “right” micro-poses without fighting. As a result, the scoring function is re-written as

$$score = \begin{cases} dist(lx_i, ly_i, lz_i, cs, cy, cz) + dist(rx_i, ry_i, rz_i, cs, cy, cz) & \text{if type} = \text{both} \\ 2dist(lx_i, ly_i, lz_i, cs, cy, cz) & \text{if type} = \text{left} \\ 2dist(rx_i, ry_i, rz_i, cs, cy, cz) & \text{if type} = \text{right} \end{cases} \quad (10)$$

5

Anytime single handed micro-poses are used, a single “main” micro-pose must be defined as well. The “main” micro-pose animates the joints that are not controlled by the “left” or “right” micro-poses. Furthermore, the “main” micro-pose can be calibrated as both a “left” and “right” handed micro-pose. In this manner, for each hand, the inhabiter can choose among the set of single handed poses for the corresponding hand and the default pose.

10

As previously mentioned, the robot joints can be made of actuators that are either electric (DC motors) or pneumatic (fluidic muscles), coupled with passive impedance elements (linear or torsional springs).

Each of these joints must be tuned based on its anticipated step responses during inhabiting. The actuators can be controlled in either position mode or velocity mode. Position mode typically involves a predetermined velocity profile to move between start and end positions—this profile is different for the electric and the pneumatic actuators. The response also results in a motion that does not capture an inhabiter’s intent faithfully since there is a loss of velocity information. The constant velocity response in position mode can also cause aliasing artifacts where a new position demand is received before completion of the previous request. This can affect the perceived gestures at the remote site during inhabiting. In order to overcome this limitation, the joints are tuned in velocity mode. This can be achieved by varying the torque to the actuation mechanism but open-loop current-based control of the torque is noisy and inaccurate. Adding a feedback sensor to achieve this can be expensive and complicate the mechanical design. For present purposes, the inhabiter’s velocity information is used to close the loop on the position without actually monitoring the applied torques. This helps achieve a much smoother and representative response of the inhabiter’s intent. For DC motors, this is fairly straight-forward since it involves computing the PID gains required to achieve the desired response. Tuning is performed using the Pessen Integral method. The fluidic muscles are coupled with spring elements. Velocity control of these actuators therefore takes into account the passive impedance of the springs. By

15

20

25

30

calibrating the applied torque against the joint space position, an equation for the balancing torque (τ_b) is obtained in the form

$$\tau_b = \gamma_1(\theta^2) + \gamma_2(\theta) - \gamma_3 \quad (11)$$

where the coefficients γ_1 , γ_2 , γ_3 are dependent on the specific joint under consideration. Additional torque must compensate for the error e between desired (θ_d) and actual (θ_a) positions. Control torque required for any joint ‘n’ can therefore be written as:

$$\tau_n = \tau_{bn} + K_p(e) + K_i \int(e \cdot dt) + K_d(de/dt) \quad (12)$$

The problem becomes more complex when using feed-forward terms, where the torque from each link is included in the equations to compensate for the coupling effects during joint motion and is not addressed in this work.

As mentioned above, the micro-poses for a virtual character must be calibrated in order to use the system. The number of micro-poses for each virtual character is pre-selected based on several factors such as desired level of fidelity during an interaction. In addition, there are limits to the number of programmable micro-poses for a character depending on the signal-noise ratio of the chosen control or tracking device. The physical robot is opaque to the actual virtual character and its micro-poses. This offers the ability to switch between multiple virtual characters (and hence, robotic surrogates) in real time. Correspondingly, the robot can inherit the behavioral and appearance properties of the active virtual character. The benefits to this approach include:

1) To a person interacting with this robot, an almost instantaneous change in behavior and appearance of the robot can be observed. As an example, a robot with the ability to change both appearance and behavior can be particularly useful during cross-cultural training and communication.

2) To a person controlling this robot, the ability to “teleport” between characters exists. This allows an inhabiter to proverbially be in “two places at the same time”. By jumping back-forth between the virtual characters, corresponding instances of robotic surrogates can be activated.

Fig. 7 shows two possible architectures supported by the paradigm. In the single surrogate configuration (left), an inhabiter can have several virtual characters in the scene and

opt to inhabit any one of them. Once activated, the robot simply inherits the appearance and behaviors (micro-poses) of the active virtual character (e.g., Surrogate 1 corresponding to Virtual Character 1 in Fig. 7). One can imagine the use of this system to make the robot appear as two different characters with varying demographics depending on the subject who is interacting with it in the remote environment. In the multiple surrogate configuration (right), the inhabiter's interface remains the same as in the previous case. However, multiple slaves can be launched in the remote environment. The slaves can be located in the same or in physically separate environments (multiple remote locations). When a master virtual character is activated, the corresponding slave character receives and transmits these signals to its hardware instance locally. When combined with the other infrastructure of the inhabiter station that includes support for video feeds and conversations via the surrogate in the remote setting, a feeling of "teleportation" may be experienced by an inhabiter.

The master virtual character can be created in a virtual environment that is representative of the slave robot's physical environment. For example, the virtual environment can be constructed using a suite of sensors on the slave robot or in the robot's environment. For example, furniture items in the robot's physical environment can be reproduced in the virtual character's virtual environment and can be seen on the display. See Fig. 8. Although described particularly herein as being controlled by a human user, in another embodiment, the virtual character can be controlled using a hybrid model of artificial intelligence and direct human control. In yet another embodiment, a collaborative application can employ the present control interface in two separate locations, such that two users can each inhabit a robot at the other's location. See Fig. 9.

The underlying paradigm offers the added functionality of being able to calibrate each slave virtual character independently. This helps in reducing the cognitive load of an inhabiter while controlling multiple surrogate instances. For instance, a group of surrogates could have different greeting gestures depending on their demographics, but the inhabiter could create the same calibration for all of the poses. With this setup, the inhabiter can still greet people in the remote environment by performing the same standard action while the surrogate at the remote end automatically handles the culturally appropriate greeting.

Specifically, the calibration routine assigns two three-dimensional positions corresponding to the positions of each of the tracking device controllers for each micro-pose. An inhabiter can enter calibration mode prior to taking control of the robotic surrogate. In the calibration mode, the virtual character will cycle through all of its pre-defined poses. The inhabiter then positions the tracking device controllers into their preferred position in

correspondence to the currently displayed micro-pose. A time-dependent routine automatically records the controller positions and advances to the next micro-pose for that virtual character. After calibrating, the inhabiter is then given a chance to test the new settings and redo the calibration procedure if necessary. In this manner, the user interface can be tailored to a specific inhabiter's needs.

The master controller can be part of a computer system that executes programming for controlling the humanoid robot as described herein. The computing system can be implemented as or can include a computing device 71 that includes a combination of hardware, software, and firmware that allows the computing device to run an applications layer or otherwise perform various processing tasks. Computing devices can include without limitation personal computers, work stations, servers, laptop computers, tablet computers, mobile devices, hand-held devices, wireless devices, smartphones, wearable devices, embedded devices, microprocessor-based devices, microcontroller-based devices, programmable consumer electronics, mini-computers, main frame computers, and the like.

The computing device can include a basic input/output system (BIOS) and an operating system as software to manage hardware components, coordinate the interface between hardware and software, and manage basic operations such as start up. The computing device can include one or more processors and memory that cooperate with the operating system to provide basic functionality for the computing device. The operating system provides support functionality for the applications layer and other processing tasks. The computing device can include a system bus or other bus (such as memory bus, local bus, peripheral bus, and the like) for providing communication between the various hardware, software, and firmware components and with any external devices. Any type of architecture or infrastructure that allows the components to communicate and interact with each other can be used.

Processing tasks can be carried out by one or more processors. Various types of processing technology can be used, including a single processor or multiple processors, a central processing unit (CPU), multicore processors, parallel processors, or distributed processors. Additional specialized processing resources such as graphics (e.g., a graphics processing unit or GPU), video, multimedia, or mathematical processing capabilities can be provided to perform certain processing tasks. Processing tasks can be implemented with computer-executable instructions, such as application programs or other program modules, executed by the computing device. Application programs and program modules can include

routines, subroutines, programs, drivers, objects, components, data structures, and the like that perform particular tasks or operate on data.

The computing device includes memory or storage, which can be accessed by the system bus or in any other manner. Memory can store control logic, instructions, and/or data.

5 Memory can include transitory memory, such as cache memory, random access memory (RAM), static random access memory (SRAM), main memory, dynamic random access memory (DRAM), and memristor memory cells. Memory can include storage for firmware or microcode, such as programmable read only memory (PROM) and erasable programmable read only memory (EPROM). Memory can include non-transitory or nonvolatile or persistent
10 memory such as read only memory (ROM), hard disk drives, optical storage devices, compact disc drives, flash drives, floppy disk drives, magnetic tape drives, memory chips, and memristor memory cells. Non-transitory memory can be provided on a removable storage device. A computer-readable medium can include any physical medium that is capable of encoding instructions and/or storing data that can be subsequently used by a
15 processor to implement embodiments of the method and system described herein. Physical media can include floppy discs, optical discs, CDs, mini-CDs, DVDs, HD-DVDs, Blu-ray discs, hard drives, tape drives, flash memory, or memory chips. Any other type of tangible, non-transitory storage that can provide instructions and/or data to a processor can be used in these embodiments.

20 The computing device can include one or more input/output interfaces for connecting input and output devices to various other components of the computing device. Input and output devices can include, without limitation, keyboards, mice, joysticks, microphones, displays, monitors, scanners, speakers, and printers. Interfaces can include universal serial bus (USB) ports, serial ports, parallel ports, game ports, and the like.

25 The computing device can access a network over a network connection 90 that provides the computing device with telecommunications capabilities. Network connection enables the computing device to communicate and interact with any combination of remote devices, remote networks, and remote entities via a communications link. The communications link can be any type of communication link, including without limitation a
30 wired or wireless link. For example, the network connection can allow the computing device to communicate with remote devices over a network, which can be a wired and/or a wireless network, and which can include any combination of intranet, local area networks (LANs), enterprise-wide networks, medium area networks, wide area networks (WANs), the Internet, or the like. Control logic and/or data can be transmitted to and from the computing device via

the network connection. The network connection can include a modem, a network interface (such as an Ethernet card), a communication port, a PCMCIA slot and card, or the like to enable transmission of and receipt of data via the communications link.

The computing device can include a browser and a display that allow a user to browse and view pages or other content served by a web server over the communications link. A web server, server, and database can be located at the same or at different locations and can be part of the same computing device, different computing devices, or distributed across a network. A data center can be located at a remote location and accessed by the computing device over a network.

The computer system can include architecture distributed over one or more networks, such as, for example, a cloud computing architecture. Cloud computing includes without limitation distributed network architectures for providing, for example, software as a service (SaaS), infrastructure as a service (IaaS), platform as a service (PaaS), network as a service (NaaS), data as a service (DaaS), database as a service (DBaaS), backend as a service (BaaS), test environment as a service (TEaaS), API as a service (APIaaS), and integration platform as a service (IPaaS).

It will be appreciated that the various features of the embodiments described herein can be combined in a variety of ways. For example, a feature described in conjunction with one embodiment may be included in another embodiment even if not explicitly described in conjunction with that embodiment.

The present invention has been described in conjunction with certain preferred embodiments. It is to be understood that the invention is not limited to the exact details of construction, operation, exact materials or embodiments shown and described, and that various modifications, substitutions of equivalents, alterations to the compositions, and other changes to the embodiments disclosed herein will be apparent to one of skill in the art.

APPENDIX

Script Files

scripts/RobotInitialization.cs

5 scripts/GenerateRobotSkeleton.cs

scripts/MapToRobot.cs

scripts/HandController.cs

scripts/PoseToRobot

10 Interface Files

interface/Robothespian.dll

interface/NTURobot.dll

interface/GenericRobotInterface.cs

interface/NTURobotInterface.cs

15 interface/RobothespianInterface.cs

Hardware Plugins

RTJoint.cpp

RTJoint.h

20 UnityWrapper.cpp

UnityWrapper.h

RobotInitialization.cs

```

using UnityEngine;
using System;

// Class to initialize and uninitialize the Robot
// Attach this class to some GameObject in the main scene (preferably the one with
// Initialization).
public class RobotInitialization : MonoBehaviour
{
    // The type of Robot to initialize
    public enum RobotType {None, Robothespian, NTURobot};
    public RobotType robotType = RobotType.Robothespian;

    // If a verbal confirmation should be given
    public bool verbalConfirm = true;

    private static bool othersObserved = false;

    // Returns if more than one robot in the active stage is being observed
    // If you call this function before onPostInitialization(), there may be no
    // active stage and false will be returned
    public static bool areOthersObserved ()
    {
        return othersObserved;
    }

    // Get callbacks
    void Awake ()
    {
        MenuSystem.requestSetupComplete(onSetupComplete);
        MenuSystem.requestWillUnload(onWillUnload);
    }

    // Initializes the robot
    private void onSetupComplete ()
    {
        if (SettingsMgr.isParticipant())
        {
            if (robotType == RobotType.Robothespian &&
!GenericRobotInterface.startup(new RobothespianInterface(), verbalConfirm))
            {
                Debug.Log("RobotInitialization: Failed to initialize
the Robothespian.");
            }
            else if (robotType == RobotType.NTURobot &&
!GenericRobotInterface.startup(new NTURobotInterface(), verbalConfirm))
            {
                Debug.Log("RobotInitialization: Failed to initialize
the NTU Robot.");
            }
            else if (robotType == RobotType.None)
            {
                Debug.Log("RobotInitialization: Skipped robot
initialization. If this was not the intended behavior, make sure you select the
appropriate option in the inspector.");
            }
        }

        Stage.getStageChangeNotification(onStageChange, true);
    }

    // When the active stage changes update the number of robots that are
    // observed
    private void onStageChange (Stage stage)

```

```
RobotInitialization.cs
{
    int numObserved = 0;
    foreach (Puppet puppet in stage.getAllPuppets())
    {
        if (SettingsMgr.isObserved(puppet.getName()))
        {
            numObserved++;
        }
    }
    othersObserved = numObserved > 1;
}

// Uninitialize if we are returning to the main menu
private void onWillUnload ()
{
    GenericRobotInterface.shutdown();
    othersObserved = false;
}

// Uninitialize if we are quitting
void OnApplicationQuit ()
{
    GenericRobotInterface.shutdown();
}
}
```

```

GenerateRobotSkeleton.cs

using UnityEngine;
using System;
using System.Collections.Generic;

// Sets up a skeleton for the Robothespian automatically (attaches MapToRobot
// automatically).
// Attach this script to the root of the Puppet
// Avatar is assumed to initially be standing up with arms straight at side
[RequireComponent (typeof (Puppet))]
public class GenerateRobotSkeleton : MonoBehaviour
{
    public bool mapToRobot = true;
    // If false, the puppet joints will not be mapped so the puppet will not move.
    public bool skipBlendForRobot = true;
    // Skips animations blending on the user side (decreases robot latency)
    public float networkFPS = 4;
    // How many times every second the robot is updated
    public List<GameObject> debugJoints = new List<GameObject>(); // A
    list of joints that should be in debug mode

    // The joints on the avatar
    public GameObject head = null;
    public GameObject torso = null;
    public GameObject leftShoulder = null;
    public GameObject leftElbow = null;
    public GameObject leftWrist = null;
    public GameObject rightShoulder = null;
    public GameObject rightElbow = null;
    public GameObject rightWrist = null;

    // Setup the skeleton by adding MapToRobot scripts.
    // The MapToRobot Selectables are added to the puppet manually.
    // We must do this before any animations start running so we cannot check
the SettingMgr here
    void Awake ()
    {
        // Get a setup complete callback
        Initialization.requestPostInitialization(onPostInitialization);

        Puppet puppet = GetComponent<Puppet>();
        MapToRobot comp;
        int id = 0;

        // Head (yaw, pitch, roll)
        comp = setupComponent(puppet, head, gameObject);
        setJoint(comp, MapToRobot.Direction.yaw,
RobothespianInterface.Joints.headYaw, id++);
        setJoint(comp, MapToRobot.Direction.pitch,
RobothespianInterface.Joints.headPitch, id++);
        setJoint(comp, MapToRobot.Direction.roll,
RobothespianInterface.Joints.headRoll, id++);
        setAngles(comp, new Vector3(-30, -45, -20), new Vector3(30, 45, 20),
new Vector3(150, 135, 160), new Vector3(210, 225, 200));
        comp.flipX = true;
        comp.flipY = true;
        comp.flipZ = false;
        puppet.addSelectable(comp);

        // Torso (pitch, yaw, roll)
        id = 0;
        comp = setupComponent(puppet, torso, gameObject);
        setJoint(comp, MapToRobot.Direction.pitch,
RobothespianInterface.Joints.torsoPitch, id++);

```

```

GenerateRobotSkeleton.cs
    setJoint(comp, MapToRobot.Direction.yaw,
RobothespianInterface.Joints.torsoYaw, id++);
    setJoint(comp, MapToRobot.Direction.roll,
RobothespianInterface.Joints.torsoRoll, id++);
    setAngles(comp, new Vector3(-15, -20, -6), new Vector3(15, 20, 6),
new Vector3(165, 160, 174), new Vector3(195, 200, 186));
    comp.flipX = false;
    comp.flipY = true;
    comp.flipZ = true;
    puppet.addSelectable(comp);

    // left shoulder (yaw, roll, pitch)
    id = 0;
    comp = setupComponent(puppet, leftShoulder, torso);
    setJoint(comp, MapToRobot.Direction.yaw,
RobothespianInterface.Joints.leftShoulderYaw, id++);
    setJoint(comp, MapToRobot.Direction.roll,
RobothespianInterface.Joints.leftShoulderRoll, id++);
    setJoint(comp, MapToRobot.Direction.pitch,
RobothespianInterface.Joints.leftShoulderPitch, id++);
    setAngles(comp, new Vector3(0, -10, 0), new Vector3(95, 40, 53), new
Vector3(95, 170, 92), new Vector3(190, 220, 145));
    comp.flipX = true;
    comp.flipY = true;
    comp.flipZ = true;
    puppet.addSelectable(comp);

    // left elbow (pitch)
    id = 0;
    comp = setupComponent(puppet, leftElbow, leftShoulder);
    setJoint(comp, MapToRobot.Direction.pitch,
RobothespianInterface.Joints.leftElbowPitch, id++);
    setJoint(comp, MapToRobot.Direction.yaw, id++);
    setJoint(comp, MapToRobot.Direction.roll, id++);
    setAngles(comp, new Vector3(45, 0, 0), new Vector3(100, 0, 0), new
Vector3(135, 0, 0), new Vector3(190, 0, 0));
    comp.flipX = true;
    comp.flipY = true;
    comp.flipZ = false;
    puppet.addSelectable(comp);

    // left wrist (roll, pitch, yaw)
    id = 0;
    comp = setupComponent(puppet, leftwrist, leftShoulder);
    setJoint(comp, MapToRobot.Direction.roll,
RobothespianInterface.Joints.leftwristPitch, id++);
    setJoint(comp, MapToRobot.Direction.pitch, id++);
    setJoint(comp, MapToRobot.Direction.yaw,
RobothespianInterface.Joints.leftwristRoll, id++);
    setAngles(comp, new Vector3(0, -90, -40), new Vector3(0, 90, 40),
new Vector3(0, 90, 140), new Vector3(0, 270, 220));
    comp.flipX = false;
    comp.flipY = true;
    comp.flipZ = true;
    comp.lastJointToLastDir = true;
    puppet.addSelectable(comp);

    // right shoulder (yaw, roll, pitch)
    id = 0;
    comp = setupComponent(puppet, rightShoulder, torso);
    setJoint(comp, MapToRobot.Direction.yaw,
RobothespianInterface.Joints.rightShoulderYaw, id++);
    setJoint(comp, MapToRobot.Direction.roll,

```

```

GenerateRobotSkeleton.cs
RobothespianInterface.Joints.rightShoulderRoll, id++);
    setJoint(comp, MapToRobot.Direction.pitch,
RobothespianInterface.Joints.rightShoulderPitch, id++);
    setAngles(comp, new Vector3(0, -10, 0), new Vector3(95, 40, 53), new
Vector3(95, 170, 92), new Vector3(190, 220, 145));
    comp.flipX = true;
    comp.flipY = false;
    comp.flipZ = false;
    puppet.addSelectable(comp);

    // right elbow (pitch, yaw, roll)
    id = 0;
    comp = setupComponent(puppet, rightElbow, rightShoulder);
    setJoint(comp, MapToRobot.Direction.pitch,
RobothespianInterface.Joints.rightElbowPitch, id++);
    setJoint(comp, MapToRobot.Direction.yaw, id++);
    setJoint(comp, MapToRobot.Direction.roll, id++);
    setAngles(comp, new Vector3(45, 0, 0), new Vector3(100, 0, 0), new
Vector3(135, 0, 0), new Vector3(190, 0, 0));
    comp.flipX = true;
    comp.flipY = false;
    comp.flipZ = true;
    puppet.addSelectable(comp);

    // right wrist (roll, pitch, yaw)
    id = 0;
    comp = setupComponent(puppet, rightwrist, rightShoulder);
    setJoint(comp, MapToRobot.Direction.roll,
RobothespianInterface.Joints.rightwristPitch, id++);
    setJoint(comp, MapToRobot.Direction.pitch, id++);
    setJoint(comp, MapToRobot.Direction.yaw,
RobothespianInterface.Joints.rightwristRoll, id++);
    setAngles(comp, new Vector3(0, -90, -40), new Vector3(0, 90, 40),
new Vector3(0, 90, 140), new Vector3(0, 270, 220));
    comp.flipX = false;
    comp.flipY = false;
    comp.flipZ = false;
    comp.lastJointToLastDir = true;
    puppet.addSelectable(comp);
}

// Turn off animation blending if we need to
// We need to wait for post-init because we need Puppet initialization to be
complete
private void onPostInitialization ()
{
    if (SettingsMgr.isParticipant() && skipBlendForRobot)
    {
        Puppet puppet = GetComponent<Puppet>();

        // Just in case we don't have an AnimationMgr even though we
should
        if (puppet.getAnimationMgr() != null)
        {
            puppet.getAnimationMgr().blendSpeed = 0;
        }
    }
}

// Adds a MapToRobot component to "cur" with respect to "back"
// Debug mode is also setup here
// The created component is returned
private MapToRobot setupComponent (Puppet puppet, GameObject cur, GameObject

```

```

GenerateRobotSkeleton.cs

back)
{
    cur.AddComponent<MapToRobot>();
    MapToRobot comp = cur.GetComponent<MapToRobot>() as MapToRobot;
    comp.joints = new List<RobothespianInterface.Joints>();
    comp.directions = new MapToRobot.Direction[3];
    comp.debug = isDebugJoint(cur);
    comp.coordinateSystem = transform;
    comp.parent = back.transform;
    comp.networkFPS = networkFPS;
    return comp;
}

// Returns if this joint should be in debug mode
private bool isDebugJoint (GameObject cur)
{
    foreach (GameObject obj in debugJoints)
    {
        if (cur == obj)
        {
            return true;
        }
    }
    return false;
}

// Maps a puppet joint to the component
// This function maps one joint
// id specifies the order of the direction convention
private void setJoint (MapToRobot comp, MapToRobot.Direction direction,
RobothespianInterface.Joints joint, int id)
{
    if (mapToRobot)
    {
        comp.joints.Add(joint);
    }
    comp.directions[id] = direction;
}

// Maps a direction for a joint to the component. The puppet joint is left
out. // id specifies the order of the direction convention.
private void setJoint (MapToRobot comp, MapToRobot.Direction direction, int
id)
{
    comp.directions[id] = direction;
}

// Sets up the angles for MapToRobot
private static void setAngles (MapToRobot comp, Vector3 minAngle, Vector3
maxAngle, Vector3 minRobot, Vector3 maxRobot)
{
    comp.minAngle = minAngle;
    comp.maxAngle = maxAngle;
    comp.minRobot = minRobot;
    comp.maxRobot = maxRobot;
}
}

```

MapToRobot.cs

```

using UnityEngine;
using System;
using System.Collections.Generic;

// Maps an arbitrary joint to a set of joints on the Robothespian
// The euler angles can be calculated using a variety of different conventions
public class MapToRobot : Selectable
{
    public enum Direction {pitch, yaw, roll};
    public Direction[] directions = new Direction[3];           //
    Specifies the order of the calculation
    public List<GenericRobotInterface.Joints> joints;           //
    Specifies the joints that will receive the measurements (in the order of directions)
    public bool lastJointToLastDir = false;                     //
    Forces the last joint to be associated with the last direction instead of going in
    order

    public Transform coordinateSystem;
    // The initial coordinate system the calculations are done in
    public Transform parent;
    // The parent of the coordinate system
    public Vector3 minAngle = new Vector3(-180, -180, -180);    // A
    clamp on minimum angle values from this joint
    public Vector3 maxAngle = new Vector3(180, 180, 180);        // A
    clamp on maximum angles values from this joint
    public Vector3 minRobot = new Vector3(0, 0, 0);             //
    The robot angle to which the minimum angle corresponds
    public Vector3 maxRobot = new Vector3(360, 360, 360);        //
    The robot angle to which the maximum angle corresponds
    public bool flipX = false;
    // If true, then the pitch is negated
    public bool flipY = true;
    // If true, then yaw is negated
    public bool flipZ = true;
    // If true, then roll is negated
    public bool debug = true;
    // If true, debug will print

    public float networkFPS = 4;
    // How many times every second the robot will be updated with this joint.

    private GameObject newCoor;
    // A rotated coordinate system used to compute euler angles for various conventions

    private GameObject reference;
    // The gameObject with the rotation in question
    private Vector3 scale;
    // A multiplier to convert to robot space

    private Vector3 lastRobot = Vector3.zero;

    private float networkTimer = 0;
    private bool[] enabledJoints = new bool[3]{true, true, true};

    // Disables and enables this scripts control over particular joints on the
    Robothespian
    // Returns if this script can have control over the argument joint
    public bool setEnabledJoint (GenericRobotInterface.Joints joint, bool
enable)
    {
        int id = joints.IndexOf(joint);
        if (id < 0 || id >= 3)
        {

```

```

                                MapToRobot.cs
        return false;
    }

    enabledJoints[id] = enable;
    return true;
}

// Disables/Enable this scripts control over all of its assigned joints
public void setEnabledAll (bool enable)
{
    enabledJoints[0] = enable;
    enabledJoints[1] = enable;
    enabledJoints[2] = enable;
}

// Initialize robt control algortihm based on initial robot orientation
public override void onWillRegister()
{
    // Setup the coordinate system
    if (coordinateSystem.gameObject != parent.gameObject)
    {
        GameObject coorObj = new GameObject();
        coorObj.transform.rotation = coordinateSystem.rotation;
        coorObj.transform.parent = parent;
        coordinateSystem = coorObj.transform;
    }

    // Setup the adjusted coordinate system
    // Unity components euler angles with one convention: z then x then
y.
the first axis we want to apply // We rotate the coordinate system so that the z axis corresponds to
it point in the // The x axis should be the second axis we want to apply
right direction // The z axis should be the third axis we want to apply
// Sometimes we have to flip an axis because there is no way to make
newCoor = new GameObject();
newCoor.transform.parent = coordinateSystem;

    // XYZ
    if (isConvention(Direction.pitch, Direction.yaw, Direction.roll))
    {
        newCoor.transform.localRotation = Quaternion.Euler(0, 90,
90);
    }
    // XZY
    else if (isConvention(Direction.pitch, Direction.roll,
Direction.yaw))
    {
        newCoor.transform.localRotation = Quaternion.Euler(0, 90,
0);
        flipZ = !flipZ;
    }
    // YXZ
    else if (isConvention(Direction.yaw, Direction.pitch,
Direction.roll))
    {
        newCoor.transform.localRotation = Quaternion.Euler(270, 0,
0);
        flipZ = !flipZ;
    }
    // YZX
    else if (isConvention(Direction.yaw, Direction.roll,

```

```

MapToRobot.cs
Direction.pitch))
{
    newCoor.transform.localRotation = Quaternion.Euler(270, -90,
0);
}
//ZXY
else if (isConvention(Direction.roll, Direction.pitch,
Direction.yaw))
{
    newCoor.transform.localRotation = Quaternion.identity;
}
// ZYX
else if (isConvention(Direction.roll, Direction.yaw,
Direction.pitch))
{
    newCoor.transform.localRotation = Quaternion.Euler(0, 0,
90);
    flipX = !flipX;
}
else
{
    Debug.Log("MapToRobot: Unsupported rotation convention.");
    shutdown(true);
    return;
}

// Setup a reference to zero the initial rotation
reference = new GameObject();
reference.transform.rotation = newCoor.transform.rotation;
reference.transform.position = transform.position;
reference.transform.parent = transform;

// Determine scale for conversion to robot
scale = Vector3.Scale((maxRobot - minRobot), compInv((maxAngle -
minAngle)));
}

// Setup the network mode and error check
protected override void onInitialize ()
{
    // If we are not a user, we don't need to run
    if (!SettingsMgr.isParticipant())
    {
        shutdown(true);
    }

    // Error checking
    if (directions == null || directions.Length > 3)
    {
        Debug.Log("MapToRobot: There must be three angles associated
with a joint.");
        shutdown(true);
        return;
    }

    // Run this script on the user side only
    setNetworkMode (false, true);
}

// Computes angles of the joint according to the convention and map them to
the robot
void LateUpdate ()
{

```

```

                                MapToRobot.cs
networkTimer += Time.deltaTime;
if ((networkTimer > 1 / networkFPS) && SettingsMgr.isUser() &&
SettingsMgr.isObserved(getPuppet().getName()))
{
    // Reset timer
    networkTimer = 0;

    // If a single robot maps to multiple characters, we only
map to the active one
    if (getPuppet().getMode() == "inactive" &&
RobotInitialization.areOthersObserved())
    {
        return;
    }

    // Compute the angles and convert them to robot space
    Vector3 angles = getAngles();
    Vector3 robot = clamp(minRobot + Vector3.Scale(angles -
minAngle, scale), minRobot, maxRobot);

    // Debug mode
    if (debug)
    {
        Debug.Log("Raw: " + transform.rotation.eulerAngles +
"      " +
                                "Calculated: " + angles + "      " +
                                "Robot: " + robot);
    }

    // Map it to the robot
    for (int i = 0; i < directions.Length && i < joints.Count;
i++)
    {
        if (enabledJoints[i])
        {
            Direction dir = directions[i];
            if (lastJointToLastDir && i == joints.Count
- 1)
            {
                dir =
directions[directions.Length-1];
            }

            if (dir == Direction.pitch)
            {
                GenericRobotInterface.updateRotation(joints[i], robot.x);
                lastRobot.x = robot.x;
            }
            else if (dir == Direction.yaw)
            {
                GenericRobotInterface.updateRotation(joints[i], robot.y);
                lastRobot.y = robot.y;
            }
            else
            {
                GenericRobotInterface.updateRotation(joints[i], robot.z);
                lastRobot.z = robot.z;
            }
        }
    }
}

```

```

MapToRobot.cs

    }
}

// Called when this script should run
protected override void onSelection ()
{
    enabled = true;
}

// Called when this script should not run
protected override void onDeselection ()
{
    enabled = false;
}

// Defines the modes in which this script runs
protected override bool runInMode (String mode)
{
    return mode == "active" || mode == "inactive";
}

// Returns the angle measurement using the class's convention
private Vector3 getAngles ()
{
    // Get a quaternion in local space (with respect to the coordinate
system)
    Quaternion local = Quaternion.Inverse(newCoor.transform.rotation) *
reference.transform.rotation;

    Vector3 angles = Vector3.zero;
    for (int i = 0; i < directions.Length; i++)
    {
        // Make a vector mask to indicate which euler angle we are
computing
        Vector3 vec = Vector3.zero;
        if (directions[i] == Direction.pitch)
        {
            vec.x = 1;
        }
        else if (directions[i] == Direction.yaw)
        {
            vec.y = 1;
        }
        else
        {
            vec.z = 1;
        }

        // The first component in the directions convention is the
first to be applied
        // This component must be the z component since it is the
first one Unity applies
        if (i == 0)
        {
            angles += vec * local.eulerAngles.z;
        }
        // Second component is x
        else if (i == 1)
        {
            angles += vec * local.eulerAngles.x;
        }
        // Thrid component is y
        else

```

```

        MapToRobot.cs
    {
        angles += vec * local.eulerAngles.y;
    }

    return toAngles(angles);
}

// Converts 0 to 360 euler angles to -180 to 180 euler angles
// Angles are flipped according to flipX, flipY, and flipZ
private Vector3 toAngles (Vector3 angles)
{
    return new Vector3(toAngle(angles.x, flipX), toAngle(angles.y,
flipY), toAngle(angles.z, flipZ));
}

// Converts a 0 to 360 angle to a -180 to 180 angle
// This is a helping function for toAngles()
// flip specifies that this angle should be inverted
private float toAngle (float angle, bool flip)
{
    if (flip)
    {
        angle = -angle;
    }

    if (angle > 180)
    {
        angle -= 360;
    }
    if (angle < -180)
    {
        angle += 360;
    }

    return angle;
}

// Returns if the argument convention matches the class's convention
private bool isConvention (Direction first, Direction second, Direction
third)
{
    Direction[] testDir = new Direction[3];
    testDir[0] = first;
    testDir[1] = second;
    testDir[2] = third;

    for (int i = 0; i < directions.Length; i++)
    {
        if (directions[i] != testDir[i])
        {
            return false;
        }
    }

    return true;
}

// Takes a component-wise inverse of a vector
// Vector3.Scale(b, compInv(a)) divides b by a
private static Vector3 compInv (Vector3 a)
{
    return new Vector3(1.0f/a.x, 1.0f/a.y, 1.0f/a.z);
}

```

```

MapToRobot.cs
}

// A component-wise clamp on a Vector3
private static Vector3 clamp(Vector3 vector, Vector3 min, Vector3 max)
{
    return new Vector3(Mathf.Clamp(vector.x, min.x, max.x),
                       Mathf.Clamp(vector.y, min.y, max.y),
                       Mathf.Clamp(vector.z, min.z, max.z));
}

// Returns the orientation of the euler angles using the class's convention
// This function was used for debug purposes
// You cannot use Quaternion.Euler() because this script may not be using
Unity's z then x then y convention
private Quaternion getOrientation (Vector3 angles)
{
    Quaternion rotation = coordinateSystem.rotation;
    foreach (Direction dir in directions)
    {
        if (dir == Direction.pitch)
        {
            rotation = Quaternion.AngleAxis(angles.x,
coordinateSystem.right) * rotation;
        }
        else if (dir == Direction.yaw)
        {
            rotation = Quaternion.AngleAxis(angles.y,
coordinateSystem.up) * rotation;
        }
        else
        {
            rotation = Quaternion.AngleAxis(angles.z,
coordinateSystem.forward) * rotation;
        }
    }

    return rotation;
}

// Logs the orientation of this joint for the robot
// The log data is under "Robot<Name of GameObject>"
protected override LogEntree[] onLog()
{
    LogEntree[] log = new LogEntree[]{new LogEntree("Robot" +
gameObject.name, getPuppet().getName())};

    if (isSelected())
    {
        log[0].addLog("Angles", lastRobot.x + "," + lastRobot.y +
", " + lastRobot.z, false);
    }
    else
    {
        log[0].addLog("Angles", "None", false);
    }

    return log;
}
}

```

HandController.cs

```

using UnityEngine;
using System;
using System.Collections.Generic;

// Controls the hand of the Robothespian according to the hydra bumpers.
// This script does not effect the motion of the virtual character, only the robot
// itself.
// This script only works on the Robothespian (NTURobot not supported).
[RequireComponent (typeof (NetworkView))]
public class HandController : Selectable
{
    // What the hand will do (close palm or point)
    public enum Mode {palm, point};
    public Mode mode = Mode.palm;

    private HashSet<RobothespianInterface.Fingers> disabledFingers = new
    HashSet<RobothespianInterface.Fingers>();
    private bool leftHand = false;
    private bool rightHand = false;

    // Disable or re-enable this scripts control over a finger
    // When you disable a finger, the finger will not re-open if it is currently
    closed
    disable) public void disableFinger (RobothespianInterface.Fingers finger, bool
    {
        if (disable)
        {
            disabledFingers.Add(finger);
        }
        else
        {
            disabledFingers.Remove(finger);
        }
    }

    // Shorthand for re-enabling all of the fingers
    // By default, all fingers are enabled so you don't need this function
    unless you call disableFinger()
    public void enableAllFingers ()
    {
        disabledFingers.Clear();
    }

    // Initialization: Run on both server and client
    protected override void onInitialize ()
    {
        Initialization.checkNetworkView(this);
        setNetworkMode (true, true);
    }

    // Called when the puppet is selected
    protected override void onSelection ()
    {
        enabled = true;
    }

    // Called when the puppet is deselected
    protected override void onDeselection ()
    {
        enabled = false;

        // Force update to the base position

```

```

                                HandController.cs
    leftHand = false;
    rightHand = false;
    update();
}

// Read the input devices and broadcast network messages
public override void onInteractorControl ()
{
    if (isSelected() && HydraInterface.refresh())
    {
        // Get the latest information from the hydra
        // Flip according to the pose mirroring settings
        bool newLeftHand;
        bool newRightHand;
        if (!Initialization.getMirrorPose())
        {
            newLeftHand =
HydraInterface.getControllers()[0].getButton(HydraController.Button.Bumper);
            newRightHand =
HydraInterface.getControllers()[1].getButton(HydraController.Button.Bumper);
        }
        else
        {
            newLeftHand =
HydraInterface.getControllers()[1].getButton(HydraController.Button.Bumper);
            newRightHand =
HydraInterface.getControllers()[0].getButton(HydraController.Button.Bumper);
        }

        // If we have an update
        if (leftHand != newLeftHand || rightHand != newRightHand)
        {
            leftHand = newLeftHand;
            rightHand = newRightHand;

            // Network update
            if (SettingsMgr.isConnected())
            {
                networkView.RPC("updateHands",
RPCMode.Others, leftHand, rightHand, (int)mode);
            }
        }
    }

    // Update any clients when they join the session
    public override void onSynchronizeClient (NetworkPlayer player)
    {
        networkView.RPC("updateHands", RPCMode.Others, leftHand, rightHand,
(int)mode);
    }

    // Update the finger joints
    void update ()
    {
        if (SettingsMgr.isUser() &&
SettingsMgr.isObserved(getPuppet().getName()))
        {
            if (mode == Mode.palm)
            {
                setFinger(RobothespianInterface.Fingers.rIndex,
rightHand);
                setFinger(RobothespianInterface.Fingers.rMiddle,

```

```

        HandController.cs
    rightHand);
    rightHand);
    rightHand);
    leftHand);
    leftHand);
    leftHand);
    leftHand);
    leftHand);
    }
    else
    {
        setFinger(RobothespianInterface.Fingers.rIndex,
        setFinger(RobothespianInterface.Fingers.rMiddle,
        setFinger(RobothespianInterface.Fingers.rRing,
        setFinger(RobothespianInterface.Fingers.rBaby,
        setFinger(RobothespianInterface.Fingers.lIndex,
        setFinger(RobothespianInterface.Fingers.lMiddle,
        setFinger(RobothespianInterface.Fingers.lRing,
        setFinger(RobothespianInterface.Fingers.lBaby,
    !rightHand);
    true);
    true);
    true);
    !leftHand);
    true);
    true);
    true);
    }
    }
}

// Sets a finger to open or close if the finger is enabled
private void setFinger (RobothespianInterface.Fingers finger, bool close)
{
    if (!disabledFingers.Contains(finger))
    {
        RobothespianInterface.controlFinger(finger, close);
    }
}

// Update everyone with the correct state
[RPC]
private void updateHands (bool leftHand, bool rightHand, int mode)
{
    this.leftHand = leftHand;
    this.rightHand = rightHand;
    this.mode = (Mode)mode;
}
}

```

PoseToRobot.cs

```

using UnityEngine;
using System;
using System.Collections.Generic;

// Script to cause the robot to assume a predefined pose when certain animations are
// running
// This script expects to be attached to the root of the Puppet containing the
// Animation and AnimationMgr Components
// This script expects an input file for each act located at
// Config/<ActName>/<PuppetName>Robot.cfg
[RequireComponent (typeof (Puppet))]
public class PoseToRobot : Selectable
{
    public List<GenericRobotInterface.Joints> jointsToControl = new
List<GenericRobotInterface.Joints>
(
    new
GenericRobotInterface.Joints[] {GenericRobotInterface.Joints.headYaw,
    GenericRobotInterface.Joints.headPitch,
    GenericRobotInterface.Joints.headRoll,
    GenericRobotInterface.Joints.torsoYaw,
    GenericRobotInterface.Joints.torsoPitch,
    GenericRobotInterface.Joints.torsoRoll,
    GenericRobotInterface.Joints.leftShoulderYaw,
    GenericRobotInterface.Joints.leftShoulderPitch,
    GenericRobotInterface.Joints.leftShoulderRoll,
    GenericRobotInterface.Joints.leftElbowPitch,
    GenericRobotInterface.Joints.leftWristPitch,
    GenericRobotInterface.Joints.leftWristRoll,
    GenericRobotInterface.Joints.rightShoulderYaw,
    GenericRobotInterface.Joints.rightShoulderPitch,
    GenericRobotInterface.Joints.rightShoulderRoll,
    GenericRobotInterface.Joints.rightElbowPitch,
    GenericRobotInterface.Joints.rightWristPitch,
    GenericRobotInterface.Joints.rightWristRoll}
);

    public List<RobothespianInterface.Fingers> fingersToControl = new
List<RobothespianInterface.Fingers>
{
    RobothespianInterface.Fingers.rIndex,
    RobothespianInterface.Fingers.rMiddle,
    RobothespianInterface.Fingers.rRing,
    RobothespianInterface.Fingers.rBaby,
    RobothespianInterface.Fingers.lIndex,
    RobothespianInterface.Fingers.lMiddle,
    RobothespianInterface.Fingers.lRing,
    RobothespianInterface.Fingers.lBaby
};

    // Data structure to store robot poses for an act
    private class ActData
    {
        public Dictionary<String, float[]> robotPoses = null;
    }

    // Script references
    private AnimationMgr animationMgr = null;
    private List<MapToRobot> skeleton = new List<MapToRobot>();
    private HandController handController = null;

    // Config file data
    private List<ActData> actData = new List<ActData>();

```

```

PoseToRobot.cs
private Dictionary<String, float[]> robotPoses = null;
// Initialization: Read in config files for all acts
// Get script references
protected override void onInitialize ()
{
    // Only run on the user side
    if (!SettingsMgr.isUser())
    {
        shutdown(true);
        return;
    }

    readInputFiles();

    animationMgr = getPuppet().getAnimationMgr();
    if (animationMgr == null)
    {
        Debug.Log("PoseToRobot: No AnimationMgr was associated with
" + getPuppet().getName());
        shutdown(true);
        return;
    }
    animationMgr.requestAnimationNotification(onAnimationChange, true);

    // Get the MapToRobot scripts
    MapToRobot[] array = GetComponentsInChildren<MapToRobot>();
    foreach (MapToRobot mapToRobot in array)
    {
        skeleton.Add(mapToRobot);
    }

    // Get the HandController if one exists
    handController = GetComponentInChildren<HandController>();

    setNetworkMode(true, true);
}

// Get the data from the correct config file and make sure it is good.
protected override void onLoadConfig ()
{
    int id = getStage().getActId();
    if (actData.Count <= id || actData[id].robotPoses == null)
    {
        Debug.Log("The robot pose input file for " + getPuppet() + "
in " + getStage().getActNames()[id] + " is missing or invalid.");
        shutdown(false);
    }
    else
    {
        robotPoses = actData[id].robotPoses;
    }
}

// Re-enable any joints we took over since we are being deactivated
protected override void onDeselection ()
{
    foreach (MapToRobot mapToRobot in skeleton)
    {
        mapToRobot.setEnabledAll(true);
    }

    if (handController != null)

```

```

                                PoseToRobot.cs
                                {
                                handController.enableAllFingers();
                                }
                                }

// Process animation changes
private void onAnimationChange (List<String> playingAnimations)
{
    // If we are not selected then we don't do anything
    if (!isSelected() || playingAnimations.Count == 0)
    {
        return;
    }

    // Find the first playing animation that is keyed in the dictionary
    foreach (String playing in playingAnimations)
    {
        if (robotPoses.ContainsKey(playing))
        {
            // Loop through all of the joints and set them
            float[] angles = robotPoses[playing];
            for (int i = 0; i < jointsToControl.Count; i++)
            {
                GenericRobotInterface.Joints joint =
                jointsToControl[i];

                // we have an angle value so take control of
                the joint and set the rotation
                if (angles[i] >= 0)
                {
                    setEnabledJoint(joint, false);
                }
                GenericRobotInterface.updateRotation(joint, angles[i]);
            }
            // we do not have an angle value so make
            sure MapToRobot is enabled
            else
            {
                setEnabledJoint(joint, true);
            }
        }

        // Loop through all of the fingers and set them
        accordingly
        for (int i = 0; i < fingersToControl.Count; i++)
        {
            if (angles[jointsToControl.Count + i] >= 0)
            {
                // Disable interactor control over
                the finger so we can set the value here
                if (handController != null)
                {
                    handController.disableFinger(fingersToControl[i], true);
                }

                bool close =
                angles[jointsToControl.Count + i] > 0.5f;
                RobothespianInterface.controlFinger(fingersToControl[i], close);
            }
            // Since we are not controlling this finger,

```

```

PoseToRobot.cs
enable interactor control
    else if (handController != null)
    {
handController.disableFinger(fingersToControl[i], false);
    }
    }
    return;
}

// Re-enable any scripts we disabled previously since no relevant
animations are playing
if (handController != null)
{
    handController.enableAllFingers();
}

foreach (MapToRobot mapToRobot in skeleton)
{
    mapToRobot.setEnabledAll(true);
}

// Run this script if we are (observed and active) OR (inactive and no one
else but this Puppet is observed)
protected override bool runInMode (String mode)
{
    if (SettingsMgr.isObserved(getPuppet().getName()))
    {
        return (mode == "active") || (mode == "inactive" &&
!RobotInitialization.areOthersObserved());
    }

    return false;
}

// Enable or disable a robot joint
private void setEnabledJoint (GenericRobotInterface.Joints joint, bool
enable)
{
    foreach (MapToRobot mapToRobot in skeleton)
    {
        mapToRobot.setEnabledJoint(joint, enable);
    }
}

// Read in the input files for all acts and report error messages
private void readInputFiles ()
{
    foreach (String actName in getStage().getActNames())
    {
        ActData data = new ActData();
        data.robotPoses = new Dictionary<String, float[]>();

        TokenReader input = new TokenReader(Application.dataPath +
"/Config/" + actName + "/" + getPuppet().getName() + "Robot.cfg");
        if (!input.isOpen())
        {
            actData.Add(data);
            continue;
        }
    }
}

```

```

PoseToRobot.cs

String animationName = input.getToken();
while (animationName != null)
{
    if (animation[animationName] == null)
    {
        Debug.Log("PoseToRobot: The animation named
" + animationName + " does not exist for " + getPuppet().getName() + ".");
        data.robotPoses = null;
        break;
    }

    bool error = false;
    float[] angles = new float[jointsToControl.Count +
fingersToControl.Count];
    for (int i = 0; i < angles.Length; i++)
    {
        if (!readFloat(input, out angles[i]))
        {
            error = true;
            break;
        }
    }
    if (error)
    {
        Debug.Log("PoseToRobot: The animation " +
animationName + " for " + getPuppet().getName() + " should be followed by " +
(jointsToControl.Count +
fingersToControl.Count) + " values indicating the robot angles.");
        data.robotPoses = null;
        break;
    }

    data.robotPoses.Add(animationName, angles);
    animationName = input.getToken();
}

actData.Add(data);
input.close();
}

// Read in a float value from the TokenReader
// Returns if the value is a float. angle contains the actual value.
// The token -- can be used to set the angle to -1
private bool readFloat (TokenReader input, out float angle)
{
    String token = input.getToken();
    if (token == "--")
    {
        angle = -1;
        return true;
    }

    return Single.TryParse(token, out angle);
}
}

```

```

<?xml version="1.0" encoding="UTF-8"?>
<!--this document was created with pestudio 5.0-->
<image name="c:\users\arjun\documents\robothespian\interface\robothespian.dll">
  <overview>
    <description>n/a</description>
    <version>n/a</version>
    <creation>13:08:2015 - 20:06:06</creation>
    <cpu>32</cpu>
    <size>61440</size>
    <type>executable</type>
    <subsystem>Console</subsystem>
    <signature/>
    <md5>0C0EF1CBE02547A38540F4DF7F29DCE6</md5>
    <sha1>CEE9E81DCE09410CD4B8760DCE7174BAB9D9DBD7</sha1>
    <imphash>742E3B2819A89E1D4A016B1B925D31E3</imphash>
  </overview>
  <indicators count="12">
    <indicator id="3049">The file references the Event Log</indicator>
    <indicator id="4023">The count (9) of Process and Thread Functions has reached the maximum threshold (1) provided</indicator>
    <indicator id="4033">The count (3) of SEH Functions has reached the maximum threshold (1) provided</indicator>
    <indicator id="1432">The count of blacklisted strings has reached the maximum threshold (30) provided</indicator>
    <indicator id="1102">The file uses Address Space Layout Randomization (ASLR) as Mitigation technique</indicator>
    <indicator id="1036">The file checksum is invalid</indicator>
    <indicator id="1620">The file has no version information</indicator>
    <indicator id="1632">The count of file extensions detected has reached the maximum threshold (5) provided</indicator>
    <indicator id="1040">The file is not signed with a Digital Certificate</indicator>
    <indicator id="1264">The file imports 82 decorated Symbols</indicator>
    <indicator id="1266">The count of imported blacklisted functions has reached the maximum threshold (1) provided</indicator>
    <indicator id="1270">The count of Antidebug imported functions has reached the maximum threshold (1) provided</indicator>
  </indicators>
  <libraries count="6">
    <library>
      <name>c:\users\arjun\documents\robothespian\interface\libioserveclient.dll</name>
      <type>Implicit</type>
      <imported_symbols>11</imported_symbols>
      <size>13000</size>
      <linker_version>10.0</linker_version>
      <checksum>0x00000000</checksum>
      <imported_symbols>11</imported_symbols>
      <checksum>n/a</checksum>
    </library>
    <library>
      <name>c:\windows\system32\kernel32.dll</name>
      <type>Implicit</type>
      <imported_symbols>21</imported_symbols>
      <size>F0000</size>
      <linker_version>12.10</linker_version>
      <checksum>0x0009A1EB</checksum>
      <imported_symbols>21</imported_symbols>
      <checksum>n/a</checksum>
    </library>
    <library>
      <name>c:\windows\system32\msvcp100.dll</name>
      <type>Implicit</type>
      <imported_symbols>60</imported_symbols>
      <size>006B</size>
      <linker_version>100.82</linker_version>
      <checksum>0x656C7564</checksum>
      <imported_symbols>60</imported_symbols>
      <checksum>n/a</checksum>
    </library>
    <library>
      <name>c:\users\arjun\documents\robothespian\interface\boost_thread-vc100-mt-1_45.dll</name>
      <type>Implicit</type>
      <imported_symbols>9</imported_symbols>
      <size>13000</size>
      <linker_version>10.0</linker_version>
      <checksum>0x00000000</checksum>
      <imported_symbols>9</imported_symbols>
      <checksum>n/a</checksum>
    </library>
    <library>
      <name>c:\users\arjun\documents\robothespian\interface\boost_date_time-vc100-mt-1_45.dll</name>
      <type>Implicit</type>
      <imported_symbols>2</imported_symbols>
      <size>13000</size>
      <linker_version>10.0</linker_version>
      <checksum>0x00000000</checksum>
      <imported_symbols>2</imported_symbols>
      <checksum>n/a</checksum>
    </library>
    <library>

```

49

```
<string>j@@</string>
<string>jBjB</string>
<string>jGjG</string>
<string>@jIjI</string>
<string>jrjr</string>
<string>Wjnjn</string>
<string>jIjI</string>
<string>j}}</string>
<string>j-j</string>
<string>jjjj</string>
<string>jkjk</string>
<string>jIjI</string>
<string>jmjmn</string>
<string>Y_</string>
<string>9^Tu</string>
<string>_ ^</string>
<string>PQSV</string>
<string>9^Tt</string>
<string>8^Pt89^Tt</string>
<string>SVWP</string>
<string>S<PPR</string>
<string>KTQVj</string>
<string>sGQV</string>
<string>Y_</string>
<string>SVWP</string>
<string>W<PPR</string>
<string>9wDu</string>
<string>Y_</string>
<string>CHVW9</string>
<string>KTPVWQ</string>
<string>WPRQ</string>
<string>PQSVW</string>
<string>Y_</string>
<string>AQWV</string>
<string>_ ^Y</string>
<string>h@m6</string>
<string>SVWP</string>
<string>Y_</string>
<string>PQSW</string>
<string>STRVj</string>
</strings>
- <virustotal>
  <results ratio="0/0"/>
</virustotal>
</image>
```

```

<?xml version="1.0" encoding="UTF-8"?>
<!--this document was created with pestudio8.50-->
<image name="c:\users\arjun\documents\robothespian\interface\nturobot.dll">
- <overview>
    <description>n/a</description>
    <version>n/a</version>
    <creation>13:08:2015 - 20:06:06</creation>
    <cpu>32</cpu>
    <size>16384</size>
    <type>executable</type>
    <subsystem>GUI</subsystem>
    <signature/>
    <md5>3FC81C0C0D61274F6AF85FA8875B3ADE</md5>
    <sha1>0B2B2D0C3F09D08DB43E2958DB9E19AF740AF727</sha1>
    <imphash>F17DD003D6AC7DF1F4D74DB557A454CA</imphash>
  </overview>
- <Indicators count="19">
    <Indicator id="3102">The file references a communications device</Indicator>
    <Indicator id="3103">The file monitors a communications device</Indicator>
    <Indicator id="3042">The file references the resources of an executable</Indicator>
    <Indicator id="4003">The count (11) of Memory Management Functions has reached the maximum
      threshold (1) provided</Indicator>
    <Indicator id="4013">The count (9) of Communication Functions has reached the maximum threshold (1)
      provided</Indicator>
    <Indicator id="4023">The count (9) of Process and Thread Functions has reached the maximum threshold
      (1) provided</Indicator>
    <Indicator id="4033">The count (5) of SEH Functions has reached the maximum threshold (1)
      provided</Indicator>
    <Indicator id="4035">The count (3) of File Management Functions has reached the maximum threshold
      (1) provided</Indicator>
    <Indicator id="1432">The count of blacklisted strings has reached the maximum threshold (30)
      provided</Indicator>
    <Indicator id="1102">The file uses Address Space Layout Randomization (ASLR) as Mitigation
      technique</Indicator>
    <Indicator id="1620">The file has no version information</Indicator>
    <Indicator id="1152">The Debug file name is different than the file name (nturobot)</Indicator>
    <Indicator id="1632">The count of file extensions detected has reached the maximum threshold (5)
      provided</Indicator>
    <Indicator id="1040">The file is not signed with a Digital Certificate</Indicator>
  </Indicators>
- <libraries count="2">
- <library>
    <name>c:\windows\system32\kernel32.dll</name>
    <type>Implicit</type>
    <imported_symbols>42</imported_symbols>
    <size>F0000</size>
    <linker_version>12.10</linker_version>
    <checksum>0x0009A1EB</checksum>
    <imported_symbols>42</imported_symbols>
    <checksum>n/a</checksum>
  </library>
- <library>
    <name>c:\windows\system32\msvcr100.dll</name>
    <type>Implicit</type>
    <imported_symbols>30</imported_symbols>

```

```

        <size/>
    </library>
</libraries>
- <debuggers>
    - <debugger>
        <format>RSDS</format>
        <path>c:\users\lenovo\desktop\dev\code\nturobottest\nturobotdll\release</path>
        <guid>CCE49A66-CF02-4EF9-A07F-78B8A74CAC62</guid>
        <age>29</age>
    </debugger>
</debuggers>
- <strings count="174">
    <string>.This.program.cannot.be.run.in.DOS.mode.</string>
    <string>Rich..=</string>
    <string>.text</string>
    <string>`.rdata</string>
    <string>@.data</string>
    <string>.rsrc</string>
    <string>@.reloc</string>
    <string>QSVW3</string>
    <string>tPWS</string>
    <string>t1WS</string>
    <string>PQSVW</string>
    <string>t=9p</string>
    <string>Y_^[</string>
    <string>uGh0B</string>
    <string>u9h|B</string>
    <string>gfff</string>
    <string>}.9y</string>
    <string>4?VQ</string>
    <string>Y_^[</string>
    <string>Y_^[</string>
    <string>~.WP</string>
    <string>PQVW</string>
    <string>WQRV</string>
    <string>VPVS</string>
    <string>z_^[]</string>
    <string>t.Vhv.</string>
    <string>8csm</string>
    <string>QQS3</string>
    <string>VW9E</string>
    <string>9hLA</string>
    <string>@_^[</string>
    <string>u.WPS</string>
    <string>uCWVS</string>
    <string>t.WVS</string>
    <string>Y_^[</string>
    <string>_^[]</string>
    <string>Y_^[</string>
    <string>Y_^[</string>
    <string><.Xf</string>
    <string>=L9o</string>
    <string>Edgar_Dll_Debug.txt</string>
    <string>Helloworld....Initializing.Edgar</string>
    <string>COM10</string>

```

```

<string>.[DONE]</string>
<string>Cannot.open.Communication.Port.</string>
<string>Cannot.Configure.Communication.Port</string>
<string>Cannot.Configure.Communication.Timeouts</string>
<string>Error.setting.communications.event.mask</string>
<string>Error:.setJointRotation...Cannot.write.string.to.RS232.</string>
<string>....</string> <string>Cannot.open.Communication.Port.Please</
string> <string>Quit.the.program.and.Re.start.your.PC.</string>
<string>GetCommState.Error</string> <string>SetCommState.Error</
string> <string>StCommTimeouts.function.failed</string>
<string>RSDSf</string>
<string>C:\Users\Lenovo\Desktop\dev\code\NTURobotTest\NTURobotdll\Release\EDGAR.pdb</string>
<string>MultiByteToWideChar</string>
<string>GetLastError</string>
<string>LoadResource</string>
<string>LockResource</string>
<string>SizeofResource</string>
<string>FindResource</string>
<string>FindResourceEx</string>
<string>Sleep</string>
<string>SetCommMask</string>
<string>CloseHandle</string>
<string>CreateFile</string>
<string>GetCommState</string>
<string>SetCommState</string>
<string>GetCommTimeouts</string>
<string>SetCommTimeouts</string>
<string>WriteFile</string>
<string>KERNEL32.dll</string>
<string>??3@YAXPAX@Z</string>
<string>free</string>
<string>memcpy_s</string>
<string>wmemcpy_s</string>
<string>memmove_s</string>
<string>_localtime64</string>
<string>_time64</string>
<string>fopen</string>
<string>fprintf</string>
<string>fclose</string>
<string>printf</string>
<string>MSVCR100.dll</string>
<string>?terminate@@@YAXXZ</string>
<string>_vsprintf</string>
<string>_unlock</string>
<string>__dllonexit</string>
<string>_lock</string>
<string>_onexit</string>
<string>_malloc_crt</string>
<string>_encoded_null</string>
<string>_initterm</string>
<string>_initterm_e</string>
<string>_amsg_exit</string>
<string>__CppXcptFilter</string>

```

```
<string>_crt_debugger_hook</string>
<string>?_type_info_dtor_internal_method@type_info@@QAEXXZ</string>
</strings>
. <virustotal>
    <results ratio="0/0"/>
</virustotal>
</image>
```

GenericRobotInterface.cs

```

using System;
using UnityEngine;

// Generic robot class
// Derive from this class to create a robot plugin
public abstract class GenericRobotInterface
{
    // Controllable joints of the Robot
    public enum Joints
    {
        headYaw,
        headPitch,
        headRoll,
        torsoYaw,
        torsoPitch,
        torsoRoll,
        leftShoulderYaw,
        leftShoulderPitch,
        leftShoulderRoll,
        leftElbowPitch,
        leftWristPitch,
        leftWristRoll,
        rightShoulderYaw,
        rightShoulderPitch,
        rightShoulderRoll,
        rightElbowPitch,
        rightWristPitch,
        rightWristRoll
    };

    // Delegates
    protected delegate float OnRequestRotation (Joints joint);
    protected delegate void OnUpdateRotation (Joints joint, float angle);
    protected delegate bool OnStartup (bool confirm);
    protected delegate void OnShutdown ();

    // Callbacks
    protected static OnRequestRotation onRequestRotation;           // Called to
    get the rotation of a particular joint
    protected static OnUpdateRotation onUpdateRotation;             // Called to
    set the rotation of a particular joint
    protected static OnStartup onStartup;                           // Called
    when the robot is initialized
    protected static OnShutdown onShutdown;                         // Called
    when the robot is uninitialized

    // This function should assign the delegates to their appropriate function
    protected abstract void registerDelegates ();

    // Function to initialize the generic robot
    // Robot provides an instance to the class to initialize
    // confirm indicates if a verbal confirmation should be provided
    public static bool startup (GenericRobotInterface robot, bool confirm)
    {
        robot.registerDelegates();
        return onStartup(confirm);
    }

    // Function to shutdown the generic robot
    public static void shutdown ()
    {
        if (onShutdown != null)
        {

```

```
GenericRobotInterface.cs
onShutdown();
    }
}

// Function to set the rotation of the specified joint to the specified angle
public static void updateRotation (Joints joint, float angle)
{
    if (onUpdateRotation != null)
    {
        onUpdateRotation(joint, angle);
    }
}

// Function to get the rotation of the specified joint
public static float requestRotation (Joints joint)
{
    if (onRequestRotation != null)
    {
        return onRequestRotation(joint);
    }
    return -1;
}
}
```

```

NTURobotInterface.cs

using System;
using System.Text;
using System.Runtime.InteropServices;
using System.Collections;
using UnityEngine;

// Interface for the NTURobot.
// Place Robothespian.dll in the Assets folder.
// Dependent dlls must be placed in the root folder of the project.
public class NTURobotInterface : GenericRobotInterface
{
    // The Joints enum is now defined in GenericRobotInterface
    // You can access it from here just like before because we inherit from
    GenericRobotInterface

    // Why does getJointRotation() return an int? The public version implicitly
    converts this int to a float.
    // Private dll functions
    [DllImport("Assets/Plugins/Robothespian/NTURobot.dll")]
    private static extern int getJointRotation(int joint);
    [DllImport("Assets/Plugins/Robothespian/NTURobot.dll")]
    private static extern bool initialize(int verbalConfirm);

    // Uninitializes the system.
    [DllImport("Assets/Plugins/Robothespian/NTURobot.dll")]
    public static extern void close();

    // Sets the angle of the specified joint.
    [DllImport("Assets/Plugins/Robothespian/NTURobot.dll")]
    public static extern void setJointRotation(Joints joint, float angle);

    // This function must be implemented if we inherit from GenericRobotSkeleton
    // We need to set GenericRobotInterface's delegates to the appropriate
    functions so that they will be called as needed
    protected override void registerDelegates ()
    {
        // initialization callback
        onStartUp = NTURobotInterface.initialize;

        // uninitialization callback
        onShutdown = NTURobotInterface.close;

        // get joint rotation callback
        onRequestRotation = NTURobotInterface.getJointRotation;

        // set joint rotation callback
        onUpdateRotation = NTURobotInterface.setJointRotation;
    }

    // I added this function. It does two things that the old initialize() did
    not:
    // 1. It uses a boolean instead of an integer for verbalConfirm
    // 2. It sets the neutral pose (RobothespianInterface.cs does the same
    thing)
    // Initializes the system and returns if successful.
    public static bool initialize (bool verbalConfirm)
    {
        bool initOk;
        if (verbalConfirm)
        {
            initOk = initialize(1);
        }
    }
}

```

```

                                NTURobotInterface.cs
    else
    {
        initOk = initialize(0);
    }

    if (initOk)
    {
        setNeutralPose();
    }

    return initOk;
}

// Returns the current angle of the specified joint
public static float getJointRotation(Joints joint)
{
    // getJointRotation returns an int and it is implicitly casted to a
float?    return getJointRotation((int)(joint));
}

// Sends the robot to its neutral pose
// This function is called automatically upon initialization
public static void setNeutralPose()
{
    setJointRotation(Joints.headYaw, 180);
    setJointRotation(Joints.headPitch, 180);
    setJointRotation(Joints.headRoll, 180);
}
}

```

```

RobothespianInterface.cs

using System;
using System.Text;
using System.Runtime.InteropServices;
using System.Collections;
using UnityEngine;

// Interface for the Robothespian.
// Place Robothespian.dll in the Assets folder.
// Dependent dlls must be placed in the root folder of the project.
public class RobothespianInterface : GenericRobotInterface
{
    // Hands of the robot
    public enum Fingers
    {
        rIndex = 0,
        rMiddle,
        rRing,
        rBaby,
        lIndex,
        lMiddle,
        lRing,
        lBaby
    };

    // Private dll functions
    [DllImport("Assets/Plugins/Robothespian/Robothespian.dll")]
    private static extern bool getJointRotation (int joint, ref float rotation);
    [DllImport("Assets/Plugins/Robothespian/Robothespian.dll")]
    private static extern void controlFinger(int finger, int open);
    [DllImport("Assets/Plugins/Robothespian/Robothespian.dll")]
    private static extern bool initialize (int verbalConfirm);
    [DllImport("Assets/Plugins/Robothespian/Robothespian.dll")]
    private static extern void speak (uint[] content, int length);

    // Uninitializes the system.
    // No action is taken if the system is not initialized.
    // The system can only be initialized once. If you call this function you
cannot re-initialize until closing the application.
    [DllImport("Assets/Plugins/Robothespian/Robothespian.dll")]
    public static extern void close ();

    // Sets the angle of the specified joint.
    // No action is taken if the system is not initialized.
    [DllImport("Assets/Plugins/Robothespian/Robothespian.dll")]
    public static extern void setJointRotation (Joints joint, float angle);

    // Register generic robot callbacks (called by GenericRobotInterface)
    protected override void registerDelegates ()
    {
        onStartUp = RobothespianInterface.initialize;
        onShutdown = RobothespianInterface.close;
        onRequestRotation = RobothespianInterface.getJointRotation;
        onUpdateRotation = RobothespianInterface.setJointRotation;
    }

    // Initializes the system and returns if successful.
    // No action is taken if the system is initialized.
    // If you call close(). Initialization will fail until the application is
restarted.
    // verbalConfirm indicates if the robot should say "login complete" on
successful initialization
    public static bool initialize (bool verbalConfirm)
    {

```

```

        RobothespianInterface.cs
    bool success = false;
    if (verbalConfirm)
    {
        success = initialize(1);
    }
    else
    {
        success = initialize(0);
    }

    if (success)
    {
        setNeutralPose();
    }

    return success;
}

// Causes the robot to speak
// Content is what the robot will say
public static void speak (String content)
{
    char[] charArray = content.ToCharArray();
    uint[] uintArray = new uint[charArray.Length];

    for (int i = 0; i < charArray.Length; i++)
    {
        uintArray[i] = (uint)(charArray[i]);
    }

    speak(uintArray, uintArray.Length);
}

// Returns the current angle of the specified joint
public static float getJointRotation (Joints joint)
{
    float angle = new float();
    if (getJointRotation((int)(joint), ref angle))
    {
        return angle;
    }

    return -1;
}

// wrapper to assist boolean call in C#
// Sets the angle of all joints of the palm.
// No action is taken if the system is not initialized.
public static void controlFinger(Fingers finger, bool close)
{
    if (close)
    {
        controlFinger((int)finger, 1);
    }
    else
    {
        controlFinger((int)finger, 0);
    }
}

// Sends the robot to its neutral pose
// This function is called automatically upon initialization
public static void setNeutralPose ()

```

```
RobothespianInterface.cs
{
    setJointRotation (Joints.headYaw, 180);
    setJointRotation (Joints.headPitch, 180);
    setJointRotation (Joints.headRoll, 180);
    setJointRotation (Joints.torsoYaw, 180);
    setJointRotation (Joints.torsoPitch, 180);
    setJointRotation (Joints.torsoRoll, 180);

    setJointRotation (Joints.leftShoulderYaw, 180);
    setJointRotation (Joints.leftShoulderPitch, 95);
    setJointRotation (Joints.leftShoulderRoll, 92);
    setJointRotation (Joints.leftElbowPitch, 180);
    setJointRotation (Joints.leftWristPitch, 180);
    setJointRotation (Joints.leftWristRoll, 180);

    setJointRotation (Joints.rightShoulderYaw, 180);
    setJointRotation (Joints.rightShoulderPitch, 95);
    setJointRotation (Joints.rightShoulderRoll, 92);
    setJointRotation (Joints.rightElbowPitch, 180);
    setJointRotation (Joints.rightWristPitch, 180);
    setJointRotation (Joints.rightWristRoll, 180);

    controlFinger(Fingers.rBaby, false);
    controlFinger(Fingers.rMiddle, false);
    controlFinger(Fingers.rIndex, false);
    controlFinger(Fingers.rRing, false);
    controlFinger(Fingers.lBaby, false);
    controlFinger(Fingers.lMiddle, false);
    controlFinger(Fingers.lIndex, false);
    controlFinger(Fingers.lRing, false);
}
}
```

RTJoint.cpp

```

#include "RTJoint.h"
#include<iostream>
#include <conio.h>
using namespace std;

RTJoint::RTJoint():
    mReadId(0),
    mWriteId(0),
    mJointVal(0),
    mJointTorque(0),
    mDemandPos(1800),
    Kp(0.5),
    Kd(0.1)
{
}

RTJoint::RTJoint( IoServeClient* client, int read_id, int write_id, float kp, float
kd, float coeffs[3])
{
    mReadId = read_id;
    mWriteId = write_id;
    mClient = client;
    mJointVal = 0;
    mDemandPos = 1800;
    mJointTorque = 0;
    running = true;

    for(int i = 0; i < 3; i++)
        mCoEffe[s[i]] = coeffs[i];

    Kp = kp;
    Kd = kd;
}

RTJoint::~~RTJoint()
{
}

void RTJoint::run()
{
    // Formula for conversion between position and spring balancing torque:
    // Torque = 0.0001 * pos^2 + 0.076 * pos - 184.95;
    // pos = -0.0012 * torque^2 + 2.8597 * torque + 1018.2;

    int jointError = 0;
    int prevJointError = 0;
    int deltaError = 0;

    float balancingTorque = 0;

    try
    {
        while(running)
        {
            //std::cout << "test./..." << std::endl;
            boost::this_thread::interruption_point();
            // Set Joint Position
            int demandPos = getDemandPos();
            send_output_value(mClient, mWriteId, demandPos);
            //if(get_value(mClient, mReadId, mJointVal) != -1)
            //{
            //}
        }
    }
}

```

```

                                RTJoint.cpp

                                if(_kbhit())
                                    break;
                                }
                                }
                                catch( const boost::thread_interrupted& )
                                {
                                    cout << "Thread was interrupted." << endl;
                                    send_output_value(mClient, mwriteId, 0);
                                }
                                catch( const std::exception& exc )
                                {
                                    cout << "(Exception caught) " << exc.what() << std::endl;
                                    send_output_value(mClient, mwriteId, 0);
                                }
                                }

int RTJoint::get_value(IoServeClient* client, int ID, int &value)
{
    IoServeCommand* command;
    IoServeResponse *response;
    char xml[LIBIOSERVE_READ_BUFFER_SIZE] = "";
    int id = 0;

    handleError( ios_response_create(&response), "ios_response_create" );
    handleError( ios_command_create(client, "output_management", "read",
&command), "ios_command_create" );
    handleError( ios_command_add_int_parameter(command, "id", ID),
"ios_command_create" );

    handleError( ios_write(client, command, response), "ios_write" );
    //Sleep(1000);
    handleError( ios_response_get_int_requestor_id(response, &id),
"ios_response_get_int_requestor_id" );
    handleError( ios_response_get_result_xml(response, xml,
LIBIOSERVE_READ_BUFFER_SIZE), "ios_response_get_result_xml" );
    handleError( ios_command_destroy(command), "ios_command_destroy" );
    handleError( ios_response_destroy(response), "ios_response_destroy" );

    //cout << "XML response: " << xml << endl;
    xmlDocPtr doc; // the resulting document tree
    xmlNodePtr cur;

    //memset(response, 0, sizeof(char) * LIBIOSERVE_READ_BUFFER_SIZE);

    doc = xmlReadMemory(xml, strlen(xml), "noname.xml", NULL, 0);
    //Sleep(100);
    if (doc == NULL) return -1;
    cur = xmlDocGetRootElement(doc);
    if (cur == NULL)
    {
        xmlFreeDoc(doc);
        return -1;
    }
    cur = cur->xmlChildrenNode;
    while (cur != NULL)
    {
        if ((xmlStrcmp(cur->name, (const xmlChar *) "value")) == 0)
        {
            int val;
            xmlChar *key = xmlNodeListGetString(doc,
cur->xmlChildrenNode, 1);

```

```

                                RTJoint.cpp
char *sNum=(char*)key;
if (sNum)
{
    int nv=sscanf(sNum,"%d",&val);
    xmlFree(key);
    if (nv==1 && val < 10000)
    {
        value=val;
        break;
    }
    else
        return -1;
}
}
cur = cur->next;
}
return 0;
}

int RTJoint::send_output_value(IoServeClient* client, int ID, int value)
{
    IoServeCommand* command;

    handleError( ios_command_create(client, "output_management", "write",
&command), "ios_command_create" );

    //ios_command_add_parameter(command, "number", output_number);
    //ios_command_add_parameter(command, "value",val);

    //err = ios_write(client, command, NULL);

    //ios_command_destroy(command);

    handleError( ios_command_add_int_parameter(command, "id", ID),
"ios_command_add_int_parameter" );
    handleError( ios_command_add_int_parameter(command, "value", value),
"ios_command_add_int_parameter" );
    handleError( ios_write(client, command, NULL), "ios_write" );
    handleError( ios_command_destroy(command), "ios_command_destroy" );

    return 0;
}

void RTJoint::handleError( IOSERVE_CLIENT_ERROR errorCode, const std::string&
functionName )
{
    if( errorCode != IOSERVE_CLIENT_NO_ERROR )
    {
        cout << "Error in " << functionName << "()": " << errorCode << ": ";

        switch( errorCode )
        {
            case IOSERVE_CLIENT_NO_ERROR:
                cout << "IOSERVE_CLIENT_NO_ERROR"; break;
            case IOSERVE_CLIENT_INVALID_SOCKET:
                cout << "IOSERVE_CLIENT_INVALID_SOCKET"; break;
            case IOSERVE_CLIENT_OPEN_SOCKET_FAILED:
                cout << "IOSERVE_CLIENT_OPEN_SOCKET_FAILED"; break;
            case IOSERVE_CLIENT_WRITE_FAILED:
                cout << "IOSERVE_CLIENT_WRITE_FAILED"; break;
            case IOSERVE_CLIENT_READ_FAILED:

```

```

        RTJoint.cpp
        cout << "IOSERVE_CLIENT_READ_FAILED"; break;
case IOSERVE_CLIENT_CONNECT_HOST_FAILED:
        cout << "IOSERVE_CLIENT_CONNECT_HOST_FAILED"; break;
case IOSERVE_CLIENT_INVALID_PARAMETER:
        cout << "IOSERVE_CLIENT_INVALID_PARAMETER"; break;
case IOSERVE_CLIENT_PASSED_BUFFER_TO_SMALL:
        cout << "IOSERVE_CLIENT_PASSED_BUFFER_TO_SMALL";
break;

case IOSERVE_CLIENT_TOO_MANY_PARAMETERS:
        cout << "IOSERVE_CLIENT_TOO_MANY_PARAMETERS"; break;
case IOSERVE_CLIENT_INVALID_RESPONSE:
        cout << "IOSERVE_CLIENT_INVALID_RESPONSE"; break;
case IOSERVE_CLIENT_INVALID_NOTIFICATION:
        cout << "IOSERVE_CLIENT_INVALID_NOTIFICATION";
break;

case IOSERVE_CLIENT_TIMEOUT:
        cout << "IOSERVE_CLIENT_TIMEOUT"; break;
case IOSERVE_CLIENT_UNKNOWN_ERROR:
        cout << "IOSERVE_CLIENT_UNKNOWN_ERROR"; break;
case IOSERVE_CLIENT_EMPTY_STRING:
        cout << "IOSERVE_CLIENT_EMPTY_STRING"; break;
default:
        break;
    }
    std::cout << endl;
}

}

/*
        if(get_value(mClient, mReadId, mJointVal) != -1)
        {
            int demandPos = getDemandPos();
            jointError = demandPos - mJointVal;
            deltaError = jointError - prevJointError;
            // Read Joint Params
            cout << "Joint Demand " << demandPos << "
Joint Position " << mJointVal << "
Joint Error" << jointError << endl;

            // P - D Controller
            balancingTorque = mCoEfs[0] * mJointVal *
mJointVal + mCoEfs[1] * mJointVal + mCoEfs[2];
            // cout << balancingTorque << endl;
            mJointTorque = balancingTorque + Kp *
(jointError) + Kd * deltaError;

            // Set Joint Torque
            send_output_value(mClient, mWriteId,
demandPos);
            prevJointError = jointError;
        }
    */

```

```

                                RTJoint.h

#ifndef _RTJOINT_H
#define _RTJOINT_H

#include "ThreadBase.h"
#include <IOServeClient.h>
#include <IOServeXmlCommand.h>
#include <IOServeXmlResponse.h>
#include "IOServeXmlNotification.h"
#include <libxml/parser.h>

class RTJoint : public SrealUtils::ThreadBase
{
    private:
        int mReadId;
        int mWriteId;
        int mDemandPos;
        int mJointVal;
        int mJointTorque;
        float mCoEffe[3];
        float kp, kd;
        IoServeClient* mClient;
        bool running;
        mutable boost::mutex mMutex;          ///<
Locking mutex.

    public:
        RTJoint();
        RTJoint(IoServeClient* client, int read_id, int write_id, float kp,
float kd, float coeffs[3]);
        ~RTJoint();
        virtual void run();

        int get_value(IoServeClient* client, int ID, int &value);
        int send_output_value(IoServeClient* client, int ID, int value);

        int getCurrentValue()
        {
            boost::lock_guard<boost::mutex> lock( mMutex );
            return mJointVal;
        }

        int getJointId ()
        {
            return mReadId;
        }

        int getDemandPos()
        {
            boost::lock_guard<boost::mutex> lock( mMutex );
            return mDemandPos;
        }

        void setDemandPos(int pos)
        {
            boost::lock_guard<boost::mutex> lock( mMutex );
            if(pos < 0)
                mDemandPos = pos + 3600;
            else
                mDemandPos = pos;
        }
}

```

```
                                RTJoint.h
void shutdown ()
{
    running = false;
}
void handleError( IOSERVE_CLIENT_ERROR errorCode, const std::string&
functionName );
};
#endif
```

Unitywrapper.cpp

```

#include <vector>
#include "Unitywrapper.h"

#include <IOServeXmlCommand.h>
#include <IOServeXmlResponse.h>
#include "IOServeXmlNotification.h"
#include <libxml/parser.h>
#include <fstream>
#include "boost/date_time/posix_time/posix_time.hpp"

#include "RTJoint.h"

bool initOk = false;
bool hasInit = false;
#define NULL 0

IoServeClient *client = NULL;
std::vector <RTJoint *> RTJointVec;
std::vector<RTJoint*> fingers;

// Initializes the system
// verbalConfirm is a boolean indicating if the robot should verbally confirm
// initialization
bool initialize (int verbalConfirm)
{
    if (!initOk && !hasInit)
    {
        std::ifstream file;
        file.open("RobothespianIP.cfg");
        if(!file.is_open())
        {
            return false;
        }

        std::string robotIP;
        file >> robotIP;
        file.close();

        if(!handleError( ios_library_initialise()))
            return false;
        //if(!handleError( ios_create_connection("robothespian.ist.ucf.edu",
7766, &client))) //(10.190.96.27)
        if(!handleError( ios_create_connection(robotIP.c_str(), 7766,
&client))) //(10.190.96.27)
            return false;

        boost::posix_time::ptime now =
boost::posix_time::microsec_clock::local_time();
        std::string name = "myTestClient-" + to_simple_string(now);

        if(!handleError( ios_authenticate(client, name.c_str(), "root",
"ABC_r00ty_123", "3"))))
            return false;

        if (verbalConfirm == 1)
        {
            ios_text_to_speech(client, "Laura", "Log in complete!");
        }

        initializeAllJoints();
        initializeFingers();
        initOk = true;
        hasInit = true;
    }
}

```

```

Unitywrapper.cpp

    }
    return initOk;
}

// Uninitializes the system
void close ()
{
    if (initOk)
    {
        for (int i = 0; i < RTJointVec.size(); i++)
        {
            RTJointVec[i]->shutdown();
            RTJointVec[i]->stop();
            delete RTJointVec[i];
        }

        for (int i = 0; i < fingers.size(); i++)
        {
            fingers[i]->shutdown();
            fingers[i]->stop();
            delete fingers[i];
        }

        handleError( ios_close_connection(client));
        handleError( ios_library_destroy());
        initOk = false;
    }
}

// Gets the rotation associated with the joint and assign it to rotation
bool getJointRotation (int joint, float* rotation)
{
    if (initOk && joint < RTJointVec.size() && joint >= 0)
    {
        *rotation = RTJointVec[joint]->getCurrentValue() / 10.0;
        return true;
    }

    return false;
}

// Sets the specified joint to the specified rotation
void setJointRotation (int joint, float angle)
{
    if (initOk && joint < RTJointVec.size() && joint >= 0)
    {
        RTJointVec[joint]->setDemandPos(angle*10);
    }
}

// Opens and closes the specified fingers
void controlFinger(int finger, int open)
{
    if (initOk && finger >= 0 && finger < fingers.size())
    {
        if (open == 0)
        {
            fingers[finger]->setDemandPos(0);
        }
        else
        {
            fingers[finger]->setDemandPos(1);
        }
    }
}

```

```

Unitywrapper.cpp

    }
}

// Causes the Robot to speak
// content is a uint array containing the text (without the null terminator)
// length is the length of this array (excluding null)
void speak (unsigned int* content, int length)
{
    if (initOk && length > 0)
    {
        // Convert the uint array to a char array
        char* charArray = new char[length+1];
        for (int i = 0; i < length; i++)
        {
            charArray[i] = (char)(content[i]);
        }
        charArray[length] = '\0';
        ios_text_to_speech(client, "Laura", charArray);
        delete charArray;
    }
}

bool handleError( IOSERVE_CLIENT_ERROR errorCode)
{
    return(errorCode == IOSERVE_CLIENT_NO_ERROR);
}

IOSERVE_CLIENT_ERROR ios_text_to_speech(IoServeClient *client, const char *voice,
const char *text)
{
    IOSERVE_CLIENT_ERROR err;
    char buffer[1000] = "";
    char fmt[1000] = "![CDATA[voice=%s\nspeech=%s]]";
    //char fmt[1000] = "![CDATA[voice=%sspeech=%s]]";

    IoServeCommand* command;
    IoServeResponse *response;

    if(!handleError( ios_response_create(&response)))
        return IOSERVE_CLIENT_INVALID_RESPONSE;

    ios_command_create(client, "output_management", "write", &command);

    sprintf(buffer, fmt, voice, text);

    ios_command_add_parameter(command, "id", "257");
    ios_command_add_parameter(command, "value", buffer);
    ios_command_add_parameter(command, "type", "output");
    ios_command_add_parameter(command, "format", "ascii");

    err = ios_write(client, command, response);

    ios_command_destroy(command);

    return err;
}

// Initializes head and elbow joints

```

Unitywrapper.cpp

```

void initializeAllJoints()
{
    // Head Yaw
    initializeJoint(RTJointVec, 116, 116);
    // Head Pitch
    initializeJoint(RTJointVec, 87, 87);
    // Head Roll
    initializeJoint(RTJointVec, 88, 88);

    // Torso Yaw
    initializeJoint(RTJointVec, 174, 174);
    // Torso Pitch
    initializeJoint(RTJointVec, 173, 173);
    // Torso Roll
    initializeJoint(RTJointVec, 175, 175);

    // Left Shoulder Yaw
    initializeJoint(RTJointVec, 68, 68);
    // Left Shoulder Pitch
    initializeJoint(RTJointVec, 70, 70);
    // Left Shoulder Roll
    initializeJoint(RTJointVec, 80, 80);

    // Left Elbow Pitch
    initializeJoint(RTJointVec, 67, 67);
    // Left Wrist Pitch
    initializeJoint(RTJointVec, 72, 72);
    // Left Wrist Roll
    initializeJoint(RTJointVec, 113, 113);

    // Right Shoulder Yaw
    initializeJoint(RTJointVec, 65, 65);
    // Right Shoulder Pitch
    initializeJoint(RTJointVec, 64, 64);
    // Right Shoulder Roll
    initializeJoint(RTJointVec, 66, 66);
    // Right Elbow Pitch
    initializeJoint(RTJointVec, 71, 71);
    // Right Wrist Pitch
    initializeJoint(RTJointVec, 73, 73);
    // Right Wrist Roll
    initializeJoint(RTJointVec, 114, 114);
}

// Initializes palm
void initializeFingers()
{
    //Right Index
    initializeJoint(fingers, 110, 110);
    //Right Middle
    initializeJoint(fingers, 93, 93);
    //Right Ring
    initializeJoint(fingers, 125, 125);
    //Right Baby
    initializeJoint(fingers, 126, 126);

    //Left Index
    initializeJoint(fingers, 106, 106);
    //Left Middle
    initializeJoint(fingers, 107, 107);
    //Left Ring
    initializeJoint(fingers, 108, 108);
    //Left Baby

```

Page 4

```

    initializeJoint(fingers, 109, 109);
}

// Initializes a single joint and places it in the argument vector
void initializeJoint (std::vector<RTJoint*> &jointVec, int writeId, int readId)
{
    float controlCoeffs[3] = {0, 0, 0};
    RTJoint* joint = new RTJoint(client, readId, writeId, 0.7, 0.1,
controlCoeffs);
    jointVec.push_back(joint);
    joint->start();
}

```

Unitywrapper.h

```
#ifndef _Unitywrapper_H
#define _Unitywrapper_H

#include <IOServeClient.h>
#include "RTJoint.h"

// helping functions
bool handleError( IOSERVE_CLIENT_ERROR errorCode );
IOSERVE_CLIENT_ERROR ios_text_to_speech(IoServeClient *client, const char *voice,
const char *text);
void initializeAllJoints();
void initializeFingers();
void initializeJoint (std::vector <RTJoint*> &jointVec, int writeId, int readId);

// dll functions
extern "C" __declspec(dllexport) bool initialize (int verbalConfirm);
extern "C" __declspec(dllexport) void close ();
extern "C" __declspec(dllexport) bool getJointRotation (int joint, float* rotation);
extern "C" __declspec(dllexport) void setJointRotation (int joint, float angle);
extern "C" __declspec(dllexport) void controlFinger(int finger, int open);
extern "C" __declspec(dllexport) void speak (unsigned int* content, int length);

#endif
```

CLAIMS

What is claimed is:

1. A system for controlling a humanoid robot from a remote location, the system comprising:
 - one or more sensing devices to sense positions and movements of the user; and
 - a computing device in communication with the one or more sensing devices, the computing device comprising one or more processors and memory, and including computer-executable instructions stored in the memory that, upon execution by the one or more processors, generate a virtual character based upon the sensed positions and movements of the user and provide a plurality of outputs for controlling the humanoid robot based upon motion of the virtual character.
2. The system of claim 1, wherein the one or more processors of the computing device include a master controller operative to generate the virtual character and to translate motion of the virtual character into instructions to control motion of the humanoid robot, the master controller in communication with a slave controller located at the humanoid robot to transmit the instructions to the slave controller.
3. The system of claim 2, wherein the master controller includes a graphics rendering engine and a physics engine including instructions operative to translate motions of the user into a three-dimensional animation of the virtual character.
4. The system of claim 1, wherein the virtual character is generated by creating an animation from the sensed positions and motions of the user.
5. The system of claim 4, wherein the animation comprises an interpolation between key frames or between time-positions or a combination thereof.
6. The system of claim 1, wherein the computing device is operative to provide a real time mapping between motions of the user and motions of the humanoid robot.

7. The system of claim 1, wherein the computing device is operative to determine a torque or force to apply to a joint of the humanoid robot based on one or more accelerations of one or more joints of the virtual character.
8. The system of claim 1, wherein a number of degrees of freedom of motion of the virtual character is greater than a number of degrees of freedom of motion of the humanoid robot.
9. The system of claim 1, wherein the virtual character has a number of joints, each joint corresponding to a joint of the user, and the computing device is operative to assign a uniform coordinate system to each joint of the virtual character.
10. The system of claim 9, wherein the computing device is operative to store in memory data comprising at least one of pitch, yaw, and roll orientation for each joint, the joints including at least one of a shoulder joint, an elbow joint, a wrist joint, a finger joint, a neck joint, a waist joint, a hip joint, a knee joint, and an ankle joint.
11. The system of claim 1, wherein the virtual character has a number of joints, each joint corresponding to a joint of the user, and the computing device is operative to store an orientation of each joint of the virtual character as a quaternion, and includes instructions to transform the orientation of each joint of the virtual character to an orientation of each joint of the humanoid robot stored as an Euler angle.
12. The system of claim 11, wherein the transform instruction comprises mapping the set of orientations for each joint of the virtual character to the set of orientations for each joint of the humanoid robot.
13. The system of claim 11, further comprising constructing a new set of orientations for each joint of the virtual character by applying a rotation operation to each joint.
14. The system of claim 11, further comprising mapping multiple joints of the virtual character to a single joint of the humanoid robot.

15. The system of claim 14, wherein the mapping comprises generating a reduced set having a reduced number of orientations of the virtual set equal to the number of joints of the humanoid robot, the reduced set including a quaternion equal to a top rotation of a hierarchy of joints and a rotation of a leaf joint of the humanoid robot.
16. The system of claim 15, further comprising converting the reduced set of orientations to a converted set of orientations of the joints of the humanoid robot utilizing a Denavit-Hartenberg convention applicable to a corresponding joint of the humanoid robot.
17. The system of claim 16, further comprising extracting an Euler angle for the corresponding joint of the humanoid robot.
18. The system of claim 1, wherein:
the computing device provides a plurality of selectable micro-poses on the display and receives a selected micro-pose via input from the user; and
the computing device causes the humanoid robot to perform the selected micro-pose.
19. The system of claim 1, wherein the one or more sensing devices comprise a physical input device coupled to the computing device to input physical motions of the user to the computing device.
20. The system of claim 19, wherein the physical input interface comprises a joystick device, a magnetic tracking device, a motion sensing device, an image-based motion recognition device, a depth-based motion recognition device, an infrared (IR) motion sensing device, or combinations thereof.
21. The system of claim 1, wherein the one or more sensing devices comprise one or more reflective markers positioned on a head of the user, and an infrared (IR) image sensor coupled to the computing device to determine a head position of the user based upon tracking of the one or more reflective markers.
22. The system of claim 1, wherein the computing device controls head motion of the humanoid robot based upon a head position of the user.

23. The system of claim 1, wherein the computing device controls arm motion of the humanoid robot based upon an arm position of the user.
24. The system of claim 1, wherein the computing device provides corresponding pluralities of outputs for controlling a plurality of humanoid robots in different locations.
25. The system of claim 24, wherein the computing device provides a menu stored in the memory and displayable on a display for selecting a current humanoid robot from the plurality thereof.
26. The system of claim 1, wherein the computing device provides a plurality of virtual characters configurations stored in memory and receives a selected virtual character configuration from the user.
27. The system of claim 26, wherein the computing device is operative to switch between selected virtual characters in real time upon input of an updated virtual character selection from a user.
28. The system of claim 26, wherein any of the plurality of virtual characters can be used to control the humanoid robot.
29. The system of claim 26, wherein the computing device includes stored in the memory a plurality of motions of the virtual character, each motion representative of a user behavior.
30. The system of claim 1, further comprising a display coupled to the computing device to provide a third person view of the virtual character as a representation of a user and a first person view from the at least one humanoid robot;
31. The system of claim 1, wherein the display comprises a first display device coupled to the computing device to provide the third person view of the virtual character and a second display device coupled to the computing device to provide the first person view from the humanoid robot.

32. The system of claim 1, wherein the computing device includes instructions operative to display the third person view of the virtual character and the first person view from the humanoid robot as separate screens or windows on one display device or on two display devices.
33. The system of claim 1, wherein the system further includes a microphone to record audio from the user in real-time and the computing device includes instructions to transmit recorded or live audio to the humanoid robot synchronized with the sensed positions and movements of the user.
34. A humanoid robot system comprising:
the system for controlling a humanoid robot from a remote location of claim 1; and
a humanoid robot in communication with the control system.
35. The system of claim 34, wherein the computing device comprises a master controller, comprising a processor and memory, to generate the virtual character and to translate motion of the virtual character into instructions to control motion of the humanoid robot, and
further comprising a slave controller, comprising a processor and memory, to receive instructions from the master controller, the slave controller located at and in communication with the at least one robotic humanoid avatar.
36. The system of claim 35, wherein the humanoid robot includes one or more joints, an actuator connected to each joint to control rotation of the connected joint, and the computing device is operative to transmit instructions to the slave controller to actuate rotation of a selected joint via the connected actuator.
37. A method for controlling a humanoid robot from a remote location, comprising:
sensing positions and movements of a user;
storing sensed positions and movements of the user as a virtual character in a computing device;
sending outputs to the humanoid robot to control motion of the humanoid robot based on the sensed positions and movements of the user.

38. The method of claim 37, further comprising generating the virtual character at a master controller of the computing device, translating motion of the virtual character into instructions to control motion of the humanoid robot, and transmitting the instructions to a slave controller located at the humanoid robot.
39. The method of claim 37, further comprising creating an animation of the virtual character from the sensed positions and motions of the user.
40. The method of claim 37, further comprising mapping in a real time motions of the virtual character to motions of the humanoid robot.
41. The method of claim 37, further comprising determining a torque or force to apply to a joint of the humanoid robot based on one or more accelerations of one or more joints of the virtual character.
42. The method of claim 37, further comprising reducing a number of joints of the virtual character to a reduced set corresponding to a number of joints of the humanoid robot.
43. The method of claim 37, further comprising storing an orientation of a joint of the virtual as a quaternion, and transforming the quaternion to an orientation of a joint of the humanoid robot as a Euler angle.
44. The method of claim 37, further comprising mapping multiple joints of the virtual character to a single joint of the humanoid robot.
45. The method of claim 37, further comprising sensing positions and movements of the user by receiving input signals from a physical input device, the physical input device comprising one or more of a joystick device, a magnetic tracking device, a motion sensing device, or an infrared motion sensing device.
46. The method of claim 37, further comprising sensing positions and movements of the user by tracking one or more reflective markers positioned on a head of the user with an infrared image sensor to determine a head position of the user.

47. The method of claim 37, further comprising controlling arm motion of the humanoid robot based upon arm motion of the user.
48. The method of claim 37, further comprising controlling head motion of the humanoid robot based upon head motion of the user.
49. The method of claim 37, further comprising providing a plurality of virtual character configurations and receiving a selected virtual character from a user.
50. The method of claim 49, further comprising switching between selected virtual characters in real time upon input of an updated virtual character selection from a user.
51. The method of claim 49, wherein any of the plurality of virtual characters can be used to control the humanoid robot.
52. The method of claim 37, further comprising sending outputs to a second humanoid robot to control motions of the second humanoid robot based on the sensed positions and movements of the user.

1/9

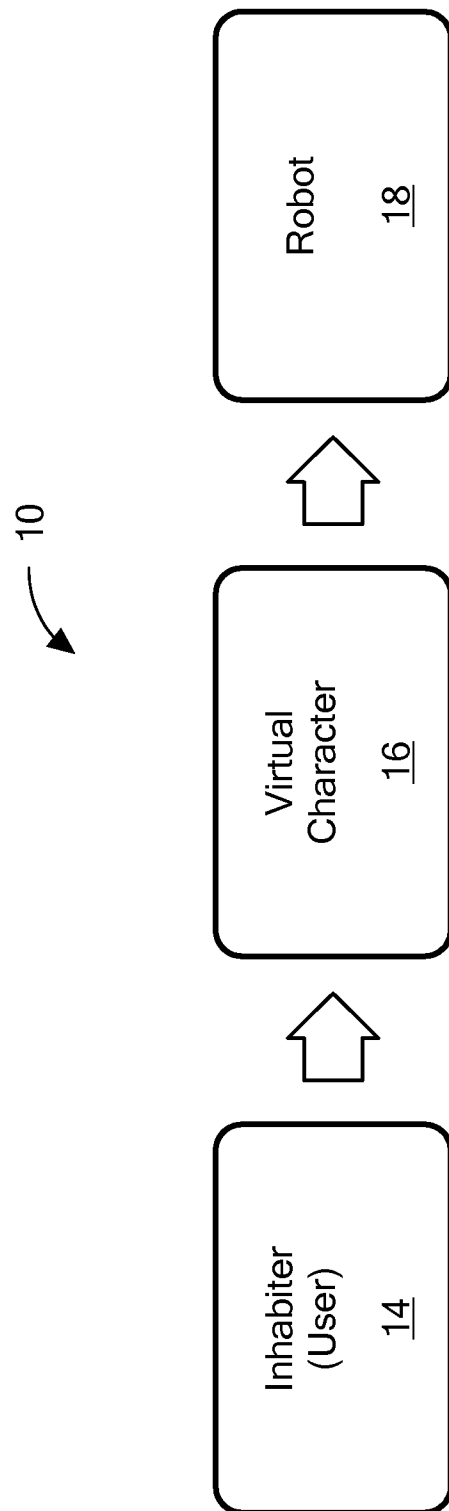
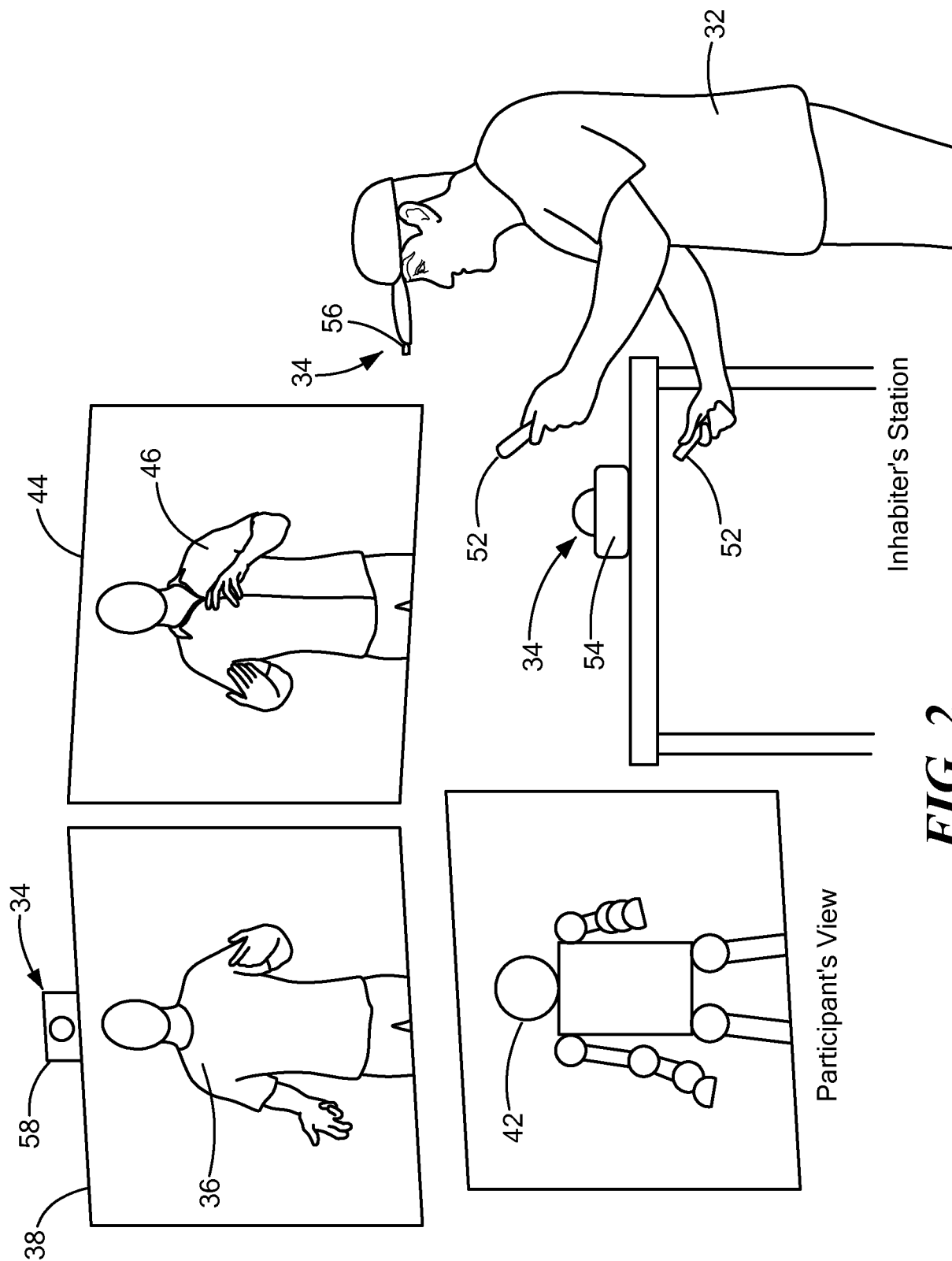


FIG. 1



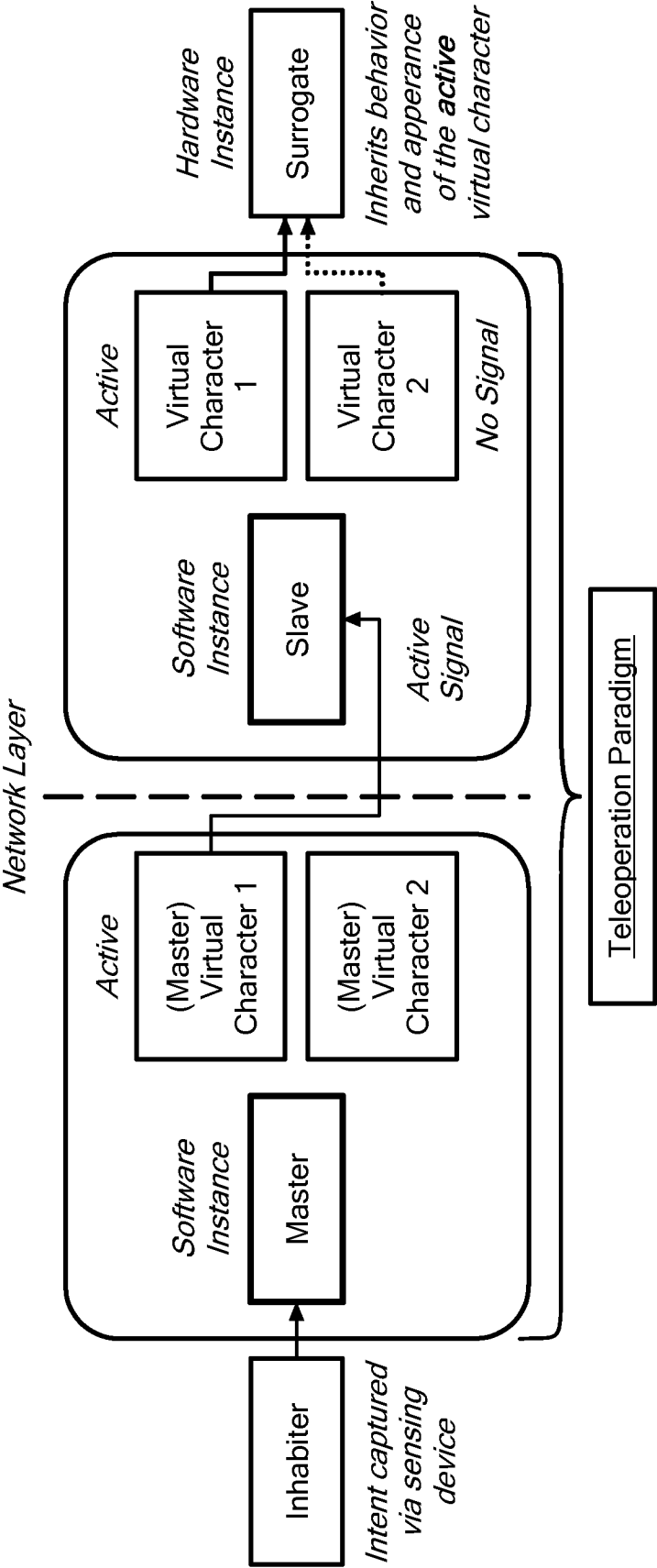
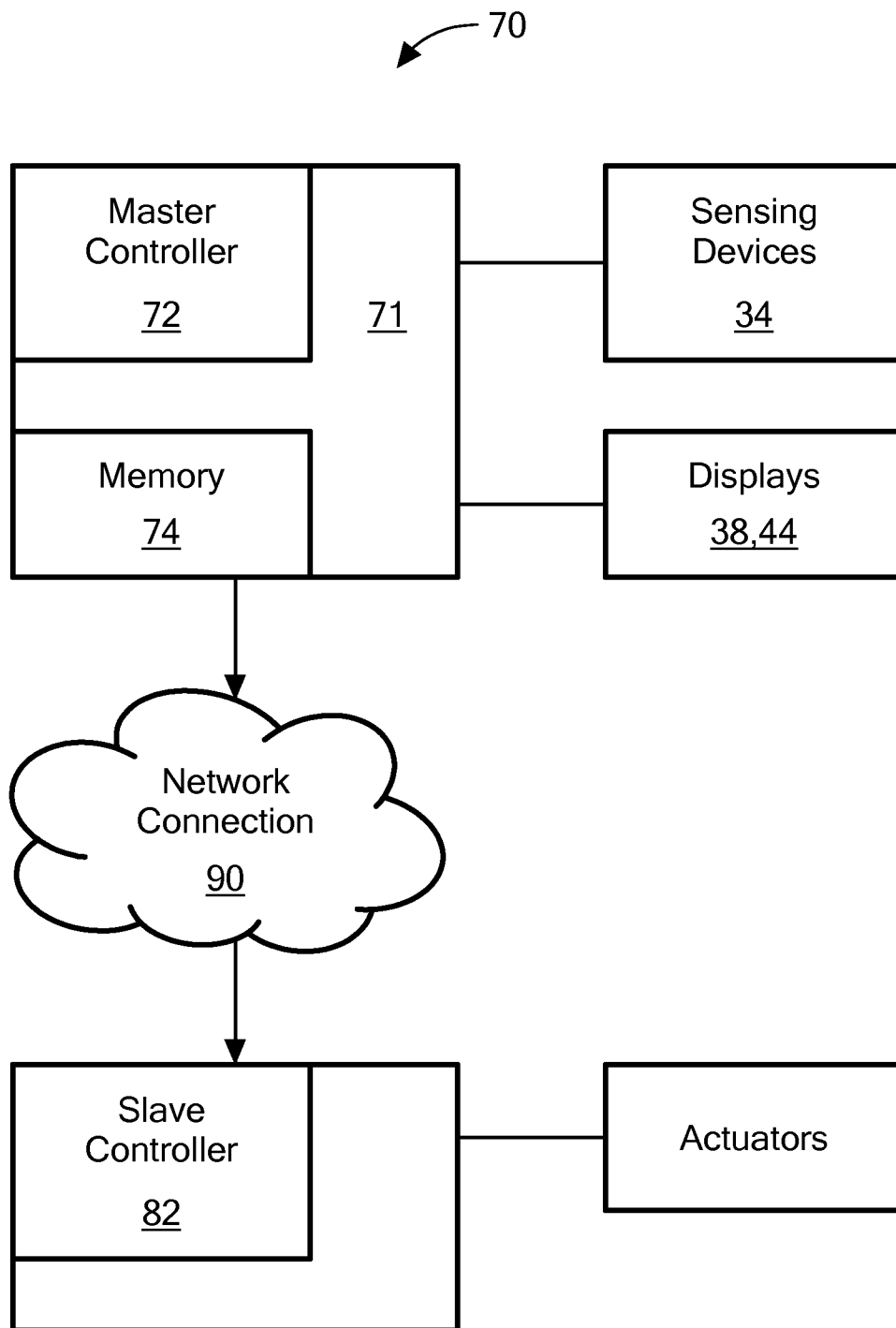


FIG. 3A

4/9

**FIG. 3B**

5/9

FIG. 4

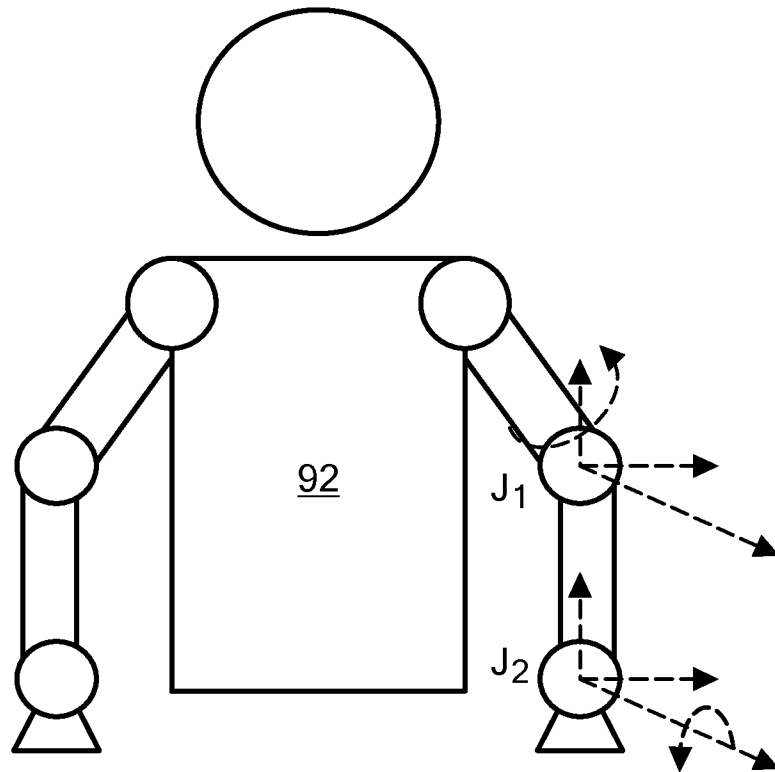
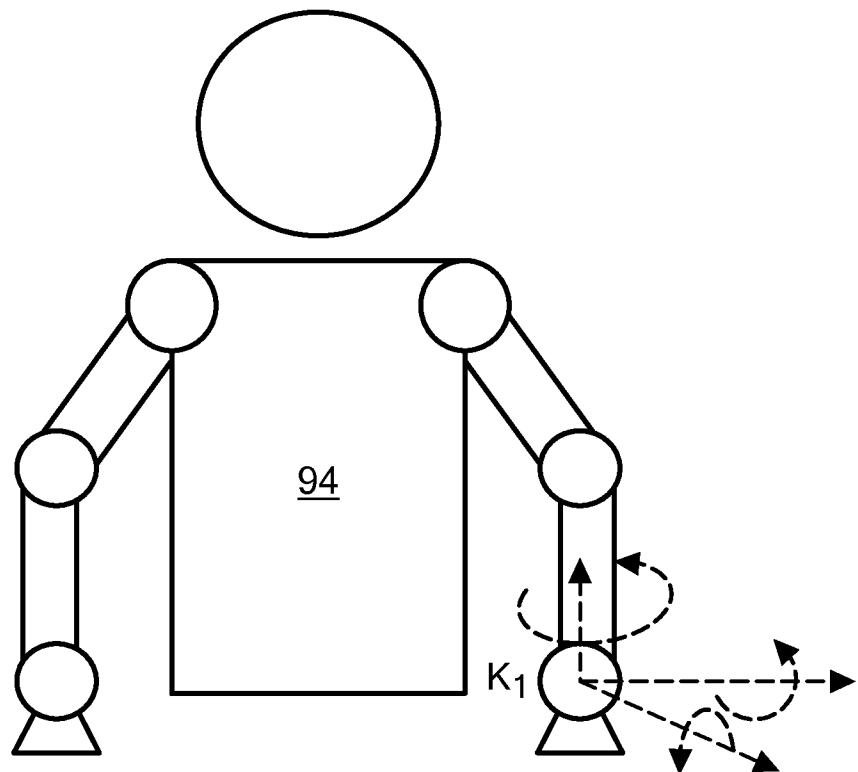
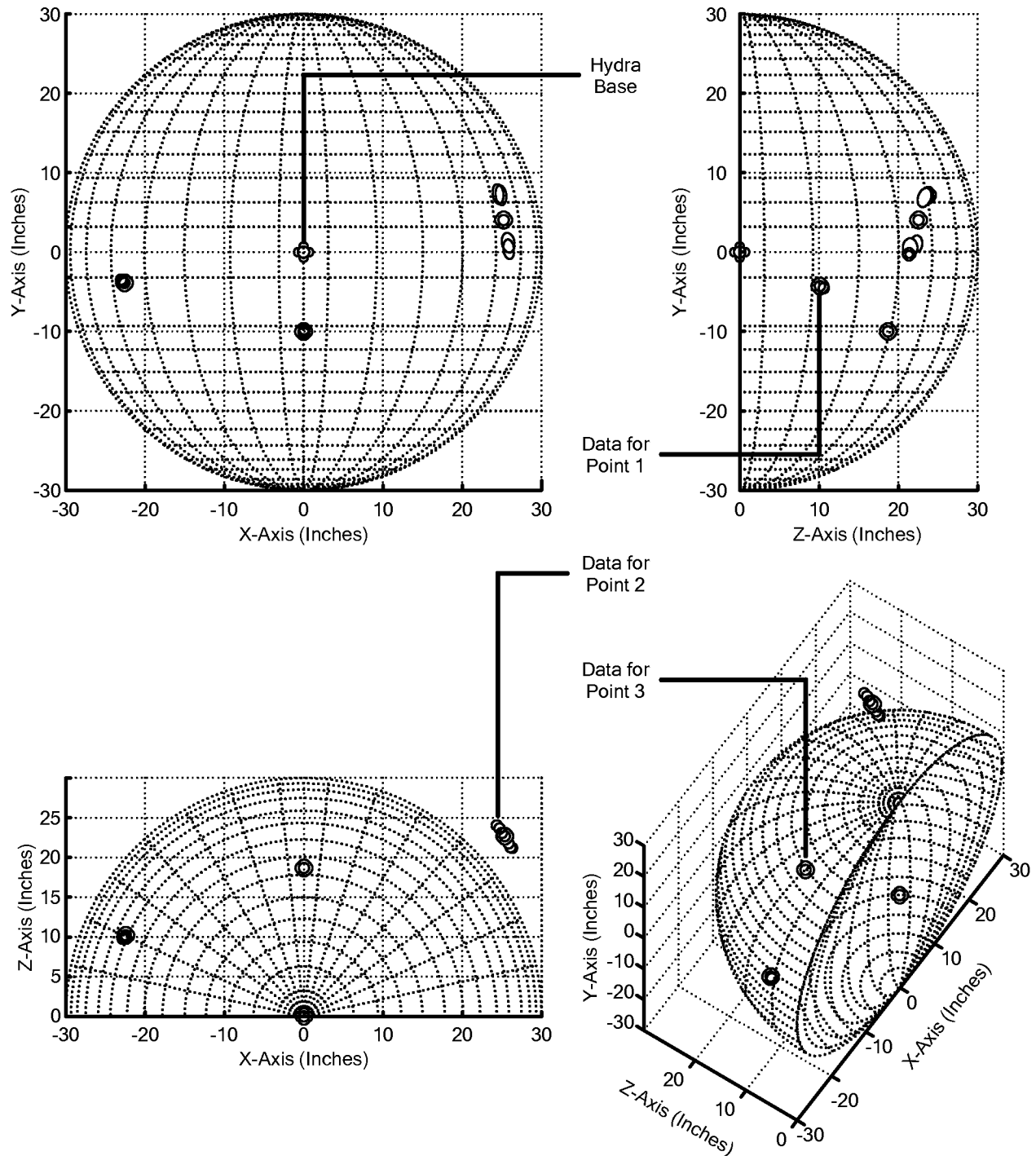


FIG. 5



6/9

**FIG. 6**

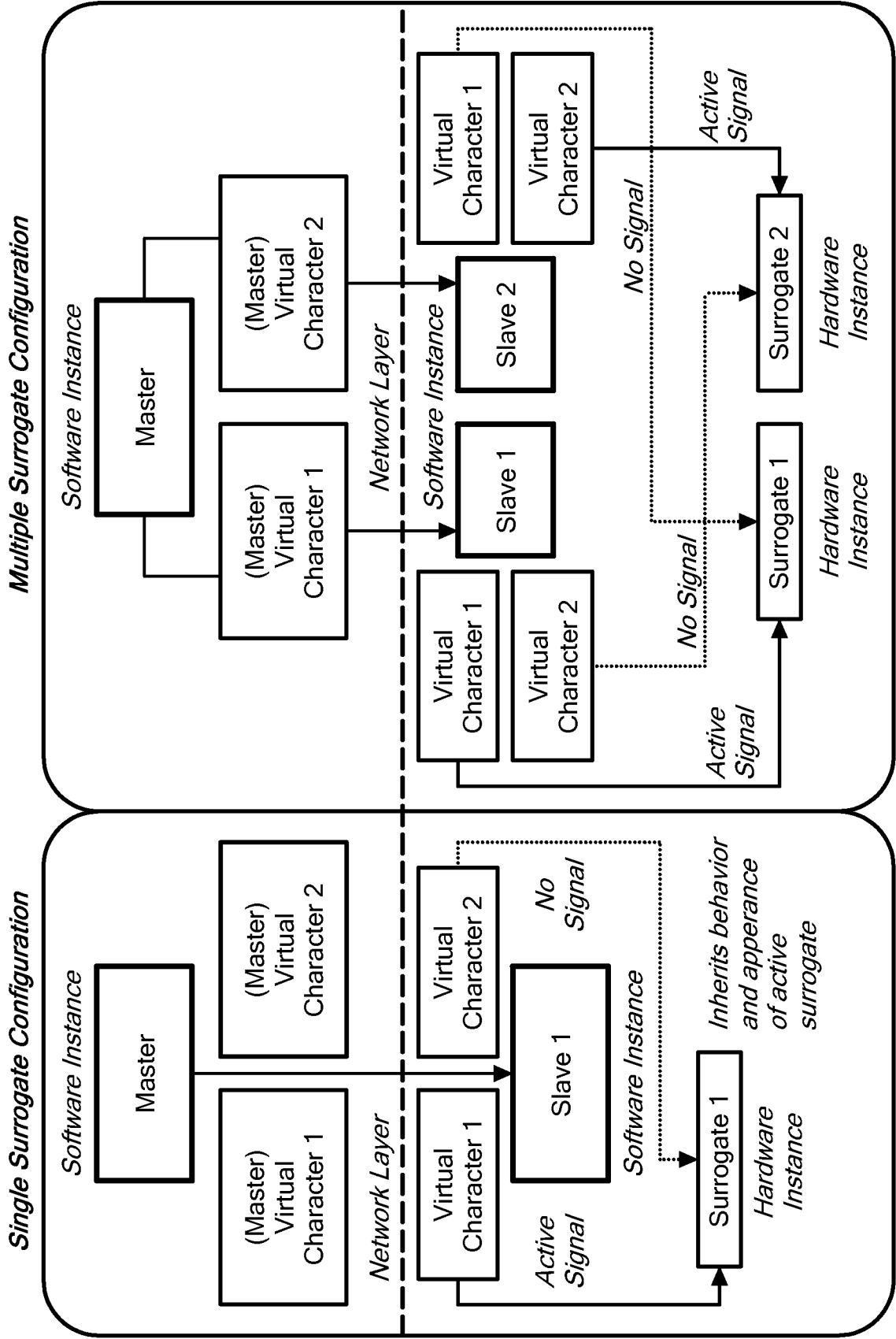


FIG. 7

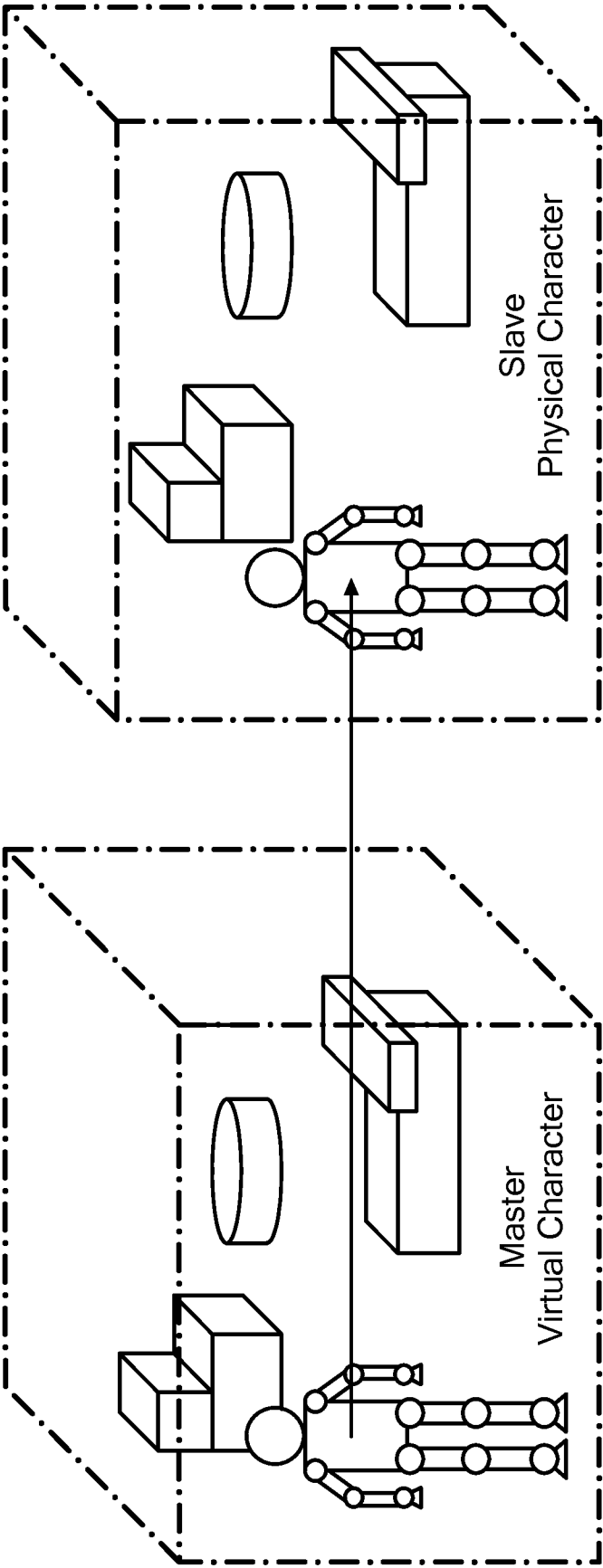


FIG. 8B

FIG. 8A

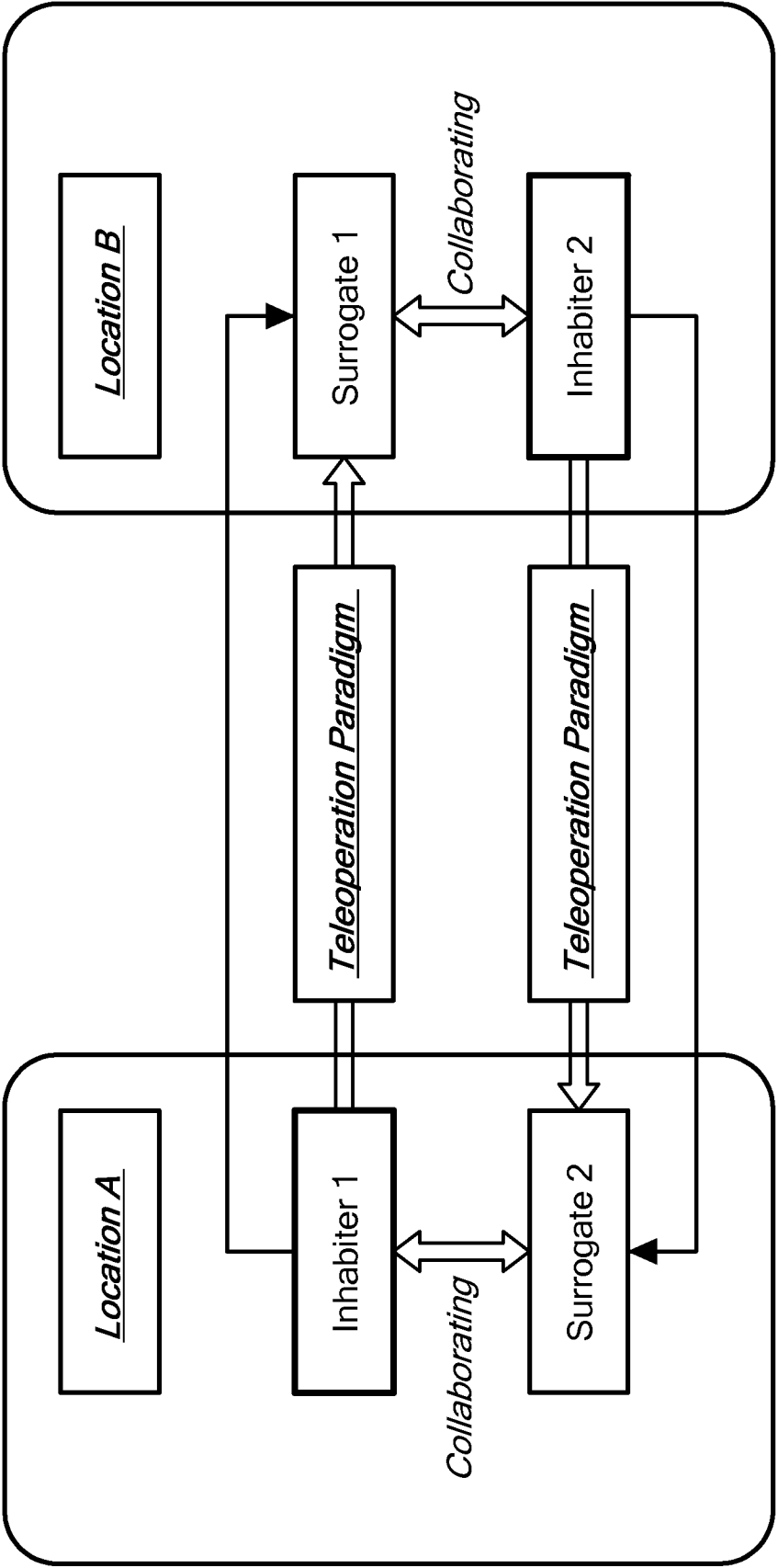


FIG. 9

INTERNATIONAL SEARCH REPORT

International application No.

PCT/US 15/45478

A. CLASSIFICATION OF SUBJECT MATTER IPC(8) - G05B 19/00 (2015.01) CPC - G05B 19/425; B25J 9/1671; G05B 19/427 According to International Patent Classification (IPC) or to both national classification and IPC		
B. FIELDS SEARCHED Minimum documentation searched (classification system followed by classification symbols) IPC(8): G05B 19/00 (2015.01) CPC: G05B 19/425; B25J 9/1671; G05B 19/427 Documentation searched other than minimum documentation to the extent that such documents are included in the fields searched IPC(8): G05B 19/00 (2015.01); CPC: G05B 2219/36159, A61B 19/22; G05B 19/425; B25J 9/1671; G05B 19/427; USPC: 700/264; 700/245; 715/961; 715/977; 901/1 (keyword limited; terms below) Electronic data base consulted during the international search (name of data base and, where practicable, search terms used) PatBase; Google(Web); Search terms used: robotic humanoid avatar virtual character sensed user position movement joint transform mapping control		
C. DOCUMENTS CONSIDERED TO BE RELEVANT		
Category*	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
X - Y	US 2008/0082214 A1 (Haskell et al.) 03 April 2008 (03.04.2008), entire document especially abstract; Fig. 1, 2; para [0005], [0009], [0018]-[0028]	1, 4-10, 18-20, 22-24, 26, 28, 29, 34, 37, 39-41, 45, 47, 49, 51, 52 2, 3, 11-17, 21, 25, 27, 30-33, 35, 36, 38, 42-44, 46, 48, 50
Y	US 2012/0041599 A1 (Townsend et al.) 16 February 2012 (16.02.2012), abstract; para [0033]-[0035]	2, 3, 35, 36, 38
Y	US 2006/0061574 A1 (Ng-Thow-Hing et al.) 23 March 2006 (23.03.2006), para [0051]-[0066]	11-17, 42-44
Y	US 2012/0038739 A1 (Welch et al.) 16 February 2012 (16.02.2012), para [0053], [0059], [0060], [0069], [0070], [0073], [0087], [0090], [0095], [0099], [0123]	21, 33, 46, 48
Y	US 2011/0288682 A1 (Pinter et al.) 24 November 2011 (24.11.2011), Fig 14; para [0058], [0077], [0074]	25
☒ Further documents are listed in the continuation of Box C. ☐		
* Special categories of cited documents: "A" document defining the general state of the art which is not considered to be of particular relevance "E" earlier application or patent but published on or after the international filing date "L" document which may throw doubts on priority claim(s) or which is cited to establish the publication date of another citation or other special reason (as specified) "O" document referring to an oral disclosure, use, exhibition or other means "P" document published prior to the international filing date but later than the priority date claimed "T" later document published after the international filing date or priority date and not in conflict with the application but cited to understand the principle or theory underlying the invention "X" document of particular relevance; the claimed invention cannot be considered novel or cannot be considered to involve an inventive step when the document is taken alone "Y" document of particular relevance; the claimed invention cannot be considered to involve an inventive step when the document is combined with one or more other such documents, such combination being obvious to a person skilled in the art "&" document member of the same patent family		
Date of the actual completion of the international search 02 October 2015 (02.10.2015)		Date of mailing of the international search report 23 NOV 2015
Name and mailing address of the ISA/US Mail Stop PCT, Attn: ISA/US, Commissioner for Patents P.O. Box 1450, Alexandria, Virginia 22313-1450 Facsimile No. 571-273-8300		Authorized officer: Lee W. Young PCT Helpdesk: 571-272-4300 PCT OSP: 571-272-7774

INTERNATIONAL SEARCH REPORT

International application No.

PCT/US 15/45478

C (Continuation). DOCUMENTS CONSIDERED TO BE RELEVANT		
Category*	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
Y	US 2014/0019374 A1 (Huang et al.) 16 January 2014 (16.01.2014), para [0008], [0053], [0079], [0110], [0111]	27, 50
Y	US 2012/0299827 A1 (Osborn et al.) 29 November 2012 (29.11.2012), para [0015], [0040], [0041]	30-32