Documentation for PAGI World: A Physically Realistic Simulation Environment for Developmental AI Systems

John Licato

February 25, 2018

Abstract

This document is a partial documentation for getting started with PAGI World. It is by no means meant to be a complete tutorial, and may not make much sense to anyone who has not taken the EHLACS class with John Licato. We are working on a more complete tutorial and hope to complete it soon.

Contents

Update History

10/10/14 Created document. —*JL*

02/19/18 Updated document to show how to utilize JSON formatted commands and messages. —*AE*

Known Bugs

- none

1 Getting and Starting Up PAGI World

PAGI World can be accessed from several places:

- `http://www.reddit.com/r/PAGIWorld` - Subreddit for discussion about PAGI World
- `http://rair.cogsci.rpi.edu/projects/pagi-world/` - Official website of PAGI World
- `https://github.com/RespeckKnuckles/PAGIworld` - Github repository containing full source and downloadable executable files

You can run PAGI World by downloading an executable file, running it through Unity (only recommended for advanced users), or executing it through the command line.

1.1 Command Line

If you download the standalone version of PAGI World, as of version 0.1.7 you can execute it with some command line options. Note that if you're using Mac OS, the executable file is in pagi_mac.app/Contents/MacOS/pagi_mac. Some are listed below:

- `-p P` Sets the port number which PAGI World is listening on to `P`. It is 42209 by default. Note that some ports are locked down depending on your network settings, so be sure to check the console after PAGI World opens to see if it started listening on that port successfully, or if there was an error.
- `-popupwindow` Forces PAGI World to start up without a frame.
- `-show-screen-selector` Forces the screen resolution selector to pop up.

2 All Commands

What follows is a list of all the possible commands that can be sent to PAGI World through TCP/IP. PAGI World make use of the JSON data interchange format, and each command sent into PAGI World shares the same general format, but has slightly different parameters. An example JSON-formatted string is shown below. Every one of these should be terminated with an endline (`\n`) character.

{”messageType”:”print”,”stringContent”:”Hello World”,”floatContent”:0.0,
”vectorContent”:{”x”:0.0,”y”:0.0},”otherStrings”:[],”messages”:[]}

2.1 Command Format

As shown above, a typical command sent into PAGI world is enclosed in brackets and contains six parameters that store different information for each command. The first, messageType, is the type of command being issued (all commands will be discussed below), and the rest are dependent on the type of command being sent. It is important that this format is followed exactly, down to the last character, otherwise it will not be interpreted by PAGI World.

2.2 Sensor Requests

This is a complete listing of all sensors available, how to request their values, and what to expect in return. The command format is shown below where the messageType is set to ”sensorRequest” and the stringContent is set to the code of the sensor being requested. All sensor codes are listed further below.

{”messageType”:”sensorRequest”,”stringContent”:”SENSOR-NAME”,”floatContent”:0.0,
”vectorContent”:{”x”:0.0,”y”:0.0},”otherStrings”:[],”messages”:[]}

2.2.1 Body tactile sensors

For each of the body sensors (which are numbered 0 to 7), the sensor codes are `B0` - `B7`. The body is surrounded by these 8 sensors, which are placed at equal distances along the body’s circumference. So in order to request the value of body sensor 5, you would send the following string:

{”messageType”:”sensorRequest”,”stringContent”:”B5”,”floatContent”:0.0,
”vectorContent”:{”x”:0.0,”y”:0.0},”otherStrings”:[],”messages”:[]}

Returns: A string of the form

{”sensorCode”:”B5”,”p”:0,”temp”:0.0,”texture”:[0.0,0.0,0.0,0.0],”endorphins”:0.0}

Where:

- `sensorCode` - the sensor's code
- `p` - 0 if the sensor detects nothing (in which case all following values will be 0), 1 if something was detected
- `temp` - The temperature value detected, as a float
- `texture[0-3]` - The texture detected. This is four floats meant to describe the quality of the texture.
- `endorphins` - Body sensors only. This is the amount of ”direct pleasure” the agent is currently feeling from that sensor. e.g., if the B0 sensor is touching a reward object, then this sensor will return a positive value for e. If it is touching a pain object, then it will return a negative value.

2.2.2 Hand tactile sensors

Each hand sensor has 5 tactile sensors, which are like the body sensors. The codes for the left hand's tactile sensors are `L0` - `L4`, while the right hand's tactile sensors have the codes `R0` - `R4`. As above, these codes are input in quotation marks as the stringContent parameter.
Returns: Same as the returned string for the body tactile sensors.

2.2.3 Velocity sensors

Tells you the current velocity of the body. Use sensor code `S` as the stringContent parameter.
Returns: A string of the form {”type”:”S”,”x”:0.0,”y”:0.0}, where `x,y` are float values representing the x and y velocities of PAGI guy's body.

2.2.4 Body position sensors

Tells you the current absolute coordinates of the body. Use sensor code `BP` as the stringContent parameter.
Returns: A string of the form {”type”:”BP”,”x”:-2.7,”y”:-2.7}, where `x,y` are float values representing the x and y coordinates of PAGI guy's body.

2.2.5 Proprioception sensors

Tells you the positions of the hands relative to the body. Use sensor code `LP` or `RP` in the stringContent parameter for the left or right hands, respectively. Note that these are relative positions, so the proprioception coordinates of the hands will change relative to the rotation and position of the body.
Returns: A string of the form {"type":"s","x":-2.25,"y":4.32}, where `s` is the sensor code you sent, and `x,y` are float values representing the x and y coordinates of the hand relative to PAGI guy's body.

2.2.6 Rotation sensor

Tells you the current rotation of the body. Use sensor code `A` in the stringContent parameter.
Returns: A string of the form {"type":"A","x":0.0,"y":0.0}. where `x` is the current rotation of the body in radians, and `y` is unused.

2.2.7 Vision sensors

There are two categories of vision sensors, the *detailed* and the *peripheral* vision sensors. The detailed vision sensors, whose sensor codes are `V0.0 - V30.20`, are located at intervals of 0.15 world coordinate units (approximately 15 pixels), measured from the bottom-left corner of the detailed visual field. For example, V4.7 is the visual sensor that is located 4x15=60 pixels to the right of sensor V0.0, and 7x15=105 pixels above. V30.20 is on the top-right corner of the visual field.

The peripheral visual sensors, whose sensor codes are `P0.0 - P15.10`, are located at 0.667 world coordinate units (approximately 66.7 pixels), measured from the bottom-left corner of the peripheral visual field. For example, P4.7 is the visual sensor located 4x66.7=267 pixels to the right of sensor P0.0, and 467 pixels above.

Simple mode is turned on by default, and simply means that more information is given by the visual sensors. Currently, simple mode cannot be turned off.

Returns: A string of the form:
{"sensorCode":"V0.3","vq":[0.0,0.0,0.0,0.0],"type":"defaultType","name":"Background"}, where:

- `sensorCode` - the vision sensor's code
- `vq[0-3]` - floats which capture the visual description by the sensor. This will correspond to things like color, fuzziness, etc. currently just returns 0 for each.

- `type` - [only in simple mode] a string describing the type/category of the object, e.g. “chair” or “human”.
- `name` - [only in simple mode] a string of the unique id of the object, e.g. “chair121” or “Bob”.

2.2.8 Map vision sensors

Sometimes you will find it too slow to request single sensor values at a time, and will need to download the entire visual field in one shot. There are two commands available for this. Sensor code `MDN` returns the data from the detailed sensors, and `MPN` does the same for the peripheral sensors.

Returns: A string of the form `"type":"s","content":"n0.0,n1.0,..."` where `s` is either `MDN` or `MPN`, depending on which command you sent. Each `nx,y` is the name/unique id of the object seen by the sensor. If no object was seen by that sensor, then it is an empty string (so you’ll see two commas with nothing in between). The values are sent in row-major order, so for example with detailed visual sensors the order will be `n0.0, n1.0, n2.0, ..., n30.0, n0.1, n1.1, ..., n30.20`.

2.2.9 Object searches

Although it is not quite low-level, the option is also provided to do a search within the visual field for an object of a specific name. This is done by sending the special command (note this is not a sensor request)

{”messageType”:”findObj”,”stringContent”:”objName”,”floatContent”:0.0,
”vectorContent”:{”x”:0.0,”y”:0.0},”otherStrings”:[”searchMode”],”messages”:[]}

where `objName` is the name of the object you are searching for. `searchMode` describes the options for the search, which are either `P` for peripheral visual sensors only, `D` for detailed visual sensors only, and `PD` for both.

Returns: A string of the form `findObj,objName[,s1][,s2]...` where:

- `objName` - the name of the object being searched for (see below)
- `si` - the visual sensor at which the object was found.

Although this is not a complete list of all the possible `objName` values (since it will be updated frequently), let this serve as a partial guide:

- Ramps: `rightRamp, leftRamp, floatingRightRamp, floatingLeftRamp`
- Wall and block pieces: `wallBlock, horizontalWall, verticalWall, floatingWallBlock, floatingHorizontalWall, floatingVerticalWall, iceBlock, lavaBlock, blueWall, greenWall, redWall`
- Dynamite: `redDynamite, greenDynamite, blueDynamite`
- Rewards/Punishments: `apple, bacon, redPill, bluePill, poison, steak`
- Other: `soldier, medkit`

2.3 Force Effectors

Force effectors allow you to essentially control PAGI guy's movement. In keeping with the philosophy of PAGI World, you don't move PAGI guy directly; rather you can send forces in particular directions. For example, to move one of his hands downwards, you send a downward force to the hand, and this will exert an equal and opposite force on his body. With enough force, you can even get him to push downwards with his hands to get him to "stand up" using his arms as legs!

All force effector commands are of the form displayed below. In messageType we have "addForce", in stringContent is `e`, the effector code that the force is being applied to, and in otherStrings is `v`, the expression to evaluate for the amount of force (this is necessary for all but ignored by some effector codes, see the descriptions that follow). If using direct values, v can be plugged into the vectorContent x parameter (Note that the x parameter is used even if the force being applied is in the vertical direction). ll effector codes return a string of the form:

{"type":"Update","content":"e,1"}

if the force was successfully applied, or:

{"type":"Update","content":"e,0"}

if there was an error, where `e` is the effector code that was sent.

{"messageType":"addForce","stringContent":"e","floatContent":0.0,
"vectorContent":{"x":3.14,"y":0.0},otherStrings":["v"],"messages":[]}

You can replace `v` with a basic arithmetic expression. To do this, wrap it in square brackets, and insert the expression as a string in otherStrings. Expressions can contain the basic arithmetic operators ($+$, $-$, $/$, and $*$), float values, and sensor aspect codes (Section 2.4.2). So for example, if you want to send a force to the body to move vertically, where the force is equal to twice the y coordinate of the body, you would use:

{"messageType":"addForce","stringContent":"BMV","floatContent":0.0,
"vectorContent":{"x":0.0,"y":0.0},"otherStrings":["[2*BPy]"],"messages":[]}

2.3.1 Hand and body forces

To apply forces to the left hand, use effector codes `LHV` and `LHH`, for vertical and horizontal movement respectively. `v` can be positive or negative. Likewise, to move the right hand use `RHV` or `RHH`, and for the body use `BMV` and `BMH`. Currently you are allowed to add arbitrary force in the up/down directions, so effectively you can make him fly. But this is likely to be removed in later versions, so try to avoid using it!

You also have the option of sending vertical and horizontal movements in one command. Use effector codes `LHvec`, `RHvec`, and `BMvec` for the left hand, right hand, and body respectively. For example, if you want to send a force vector to the right hand of 3.1 in the x direction and 2.0 in the y direction, you'd use the command:

{”messageType”:”addForce”,”stringContent”:”RHvec”,”floatContent”:0.0,
”vectorContent”:{”x”:3.1,”y”:2.0},”otherStrings”:[],”messages”:[]}

Similarly, if you wanted to specify this command but with expressions, you would insert the expressions into the otherStrings parameter, as shown below.

{”messageType”:”addForce”,”stringContent”:”RHvec”,”floatContent”:0.0, ”vectorContent”:{”x”:0.0,”y”:0.0},
”otherStrings”:[”[3.1*BMV]”,”[2.0/BMH]”],”messages”:[]}

2.3.2 Jumping

Use effector code `J` to make PAGI guy jump. Use a force of about 30,000 for a good jump. A jump will also exert an equivalent downward force on whatever object the bottom of the body was touching at the time of the jump. If the bottom of the body is not touching anything, the jump will not be carried out and the following will be returned:

{”type”:”Update”,”content”:”J,0”}

2.3.3 Rotation

To rotate the body, use effector code `BR`. `v` is the amount of torque to use to rotate the body (can be positive or negative).

2.3.4 Gripping and releasing

To have the hands grip or release, send the relevant effector codes with `v` set to 0 (`v` is ignored here). To make the right or left hand grip, use effector codes `RHG` and `LHG` respectively. To make the right or left hands release, use effector codes `RHR` or `LHR`. A hand is in a gripping state until it is sent a command to release.

2.4 States and Reflexes

Although TCP/IP communication is normally quite fast, there are some times when you will need PAGI guy to react much quicker, or to notify the AI side when a certain combination of sensors is met. For this reason, a *states and reflexes* system is built into PAGI World. Setting reflexes involves the most complex commands available within PAGI World, and **this section is only recommended for advanced users who have exhausted the means available in the other sections.**

A *state* in PAGI World is in a simple binary state: it is either active, or it is not. To establish a state, send a command in the following form:

{"messageType":"setState","stringContent":"s","floatContent":d,
"vectorContent":{"x":0.0,"y":0.0},"otherStrings":[],"messages":[]}

Where `s` in stringContent is the unique name of this state, and `d` in floatContent is the duration that this state will remain active, measured in milliseconds. If you want a state to remain active indefinitely, set `d` to -1.0. If you want to remove, or deactivate a state, set `d` to 0.0.

A *reflex* in PAGI World allows PAGI guy to automatically respond to a certain state of the world automatically, much faster than if this were dependent on TCP/IP. However, we do not recommend making your program reflex-heavy; reflexes should be reserved for those actions that require precisely timed or incredibly quick actions. To create a reflex, send a command in the following form:

{"messageType":"setReflex","stringContent":"n","floatContent":0.0,
"vectorContent":{"x":0.0,"y":0.0},"otherStrings":["C"],"messages":[A]}

`setReflex,n,C[,A]` where:

- `stringContent: n` - The unique name of this reflex. If a reflex with this name already exists, this will replace the older reflex.
- `otherStrings[0]: C` - A listing, where each item is separated by a semicolon, of the conditions that must be met for this reflex to be triggered. There are two types of conditions: state conditions, and sensory conditions.
  - **State conditions** simply are used to check if a state is active or not. Each state condition is of the form `[-]s` where `s` is simply the name of the state. The hyphen, which is optional, makes it so that the condition is only satisfied if the state `s` is *not* active.
  - **Sensory conditions** are met if a particular sensor aspect has a value that is above, below, or equal to a certain value. Sensor aspects are the float values that are associated with each sensor. For example, each

of the hand sensors checks for temperature, a four-dimensional texture vector, and endorphins. Here the temperature value, the endorphin value, and each of the four floats making up the texture vector are all sensor aspects. Each of these sensor aspects has a unique code which is listed below.

Each sensory condition is of the form `q|o|v` where `q` is the code of the sensor aspect (see Section 2.4.2), `o` is the operator to use to compare (which is either $<, >, =$, { for $\leq$, } for $\geq$, or ! for $\neq$) and `v` is the value to use. For example, `L0_tx1|{|0.1` means that we want to check the first float of the texture vector detected by left hand sensor 0. If it is less than or equal to 0.1, then this sensory condition is satisfied. For the equality operator ($=$), there is a tolerance of 0.01, so that for the expression `L0_tx1|=|5.0`, any values of `L0_tx1` from 4.9 to 5.1 will be accepted.

- `messages[]: A` - (optional) A JSON array of messages to be sent to PAGI world that contain the actions to be made when a reflex is fired. The reflex is fired when every condition in `C` is met. Each action is another command, except for another reflex command. For example, to create a reflex named 'myCustomReflex' that makes the PAGI World guy say 'hello world' when the world state changes to a previously defined state called 'myCustomState', the following command would need to be sent:

{"messageType":"setReflex","stringContent":"myCustomReflex","floatContent":0.0, "vectorContent":{"x":0.0,"y":0.0},"otherStrings":["myCustomState"],"messages":[{"messageType":"say","stringContent":"hello world","floatContent":10.0, "vectorContent":{"x":5.0,"y":-3.0},"otherStrings":["P"],"messages":[]}]}

Here's another example which makes use of the states and reflexes system. Let's say that you defined a state called "`testState`". If `testState` is active, you want a reflex to fire that will have two actions take place: You want PAGI guy to move right, and you want him to jump. In order to create this reflex, you would use:

{"messageType":"setReflex","stringContent":"myCustomReflex","floatContent":0.0,
"vectorContent":{"x":0.0,"y":0.0},"otherStrings":["testState"],
"messages":[{"messageType":"addForce","stringContent":"BMH","floatContent":0.0,
"vectorContent":"x":100.0,"y":0.0,"otherStrings":[],"messages":[]},
{"messageType":"addForce","stringContent":"J","floatContent":0.0,
"vectorContent":{"x":30000.0,"y":0.0},"otherStrings":[],"messages":[]}] }

Now let's say you wanted to add another reflex that has two conditions: the downward velocity of PAGI guy's body has to be greater than 100 (`Sx|>|100`), and the state `testState` must be active. You want it to have zero actions (this

is sometimes useful when you only want the notification that the conditions were met). You would use the command:

{”messageType”:”setReflex”,”stringContent”:”myCustomReflex”,”floatContent”:0.0,
”vectorContent”:{”x”:0.0,”y”:0.0},”otherStrings”:[”testState;Sx—¿—100”],”messages”:[]}

2.4.1 Other state and reflex commands

To remove a state, send the command

{”messageType”:”setState”,”stringContent”:”s”,”floatContent”:0.0,
”vectorContent”:{”x”:0.0,”y”:0.0},”otherStrings”:[],”messages”:[]}

where `s` in stringContent is the name of the state you want to disable, and 0.0 is specified in floatContent.

To remove a reflex, simply send the command:

{”messageType”:”removeReflex”,”stringContent”:”n”,”floatContent”:0.0,
”vectorContent”:{”x”:0.0,”y”:0.0},”otherStrings”:[],”messages”:[]}

Where `n` in stringContent is the name of the reflex to remove.

You can get a listing of all active states with the command:

{”messageType”:”getStates”,”stringContent”:””,”floatContent”:0.0,
”vectorContent”:{”x”:0.0,”y”:0.0},”otherStrings”:[],”messages”:[]}

It returns a string of the form `activeStates,s1,s2,...` where each `si` is the name of a state that is currently active.

To get a listing of all active reflexes, use the command:

{”messageType”:”getReflexes”,”stringContent”:””,”floatContent”:0.0,
”vectorContent”:{”x”:0.0,”y”:0.0},”otherStrings”:[],”messages”:[]}

It returns a string of the form `activeReflexes,r1,r2,...` where each `ri` is the name of a reflex that is currently active.

2.4.2 All sensor aspect codes

L0_tmp - Left hand sensor 0 temperature
L0_tx1 - Left hand sensor 0 texture vector component 1
L0_tx2 - Left hand sensor 0 texture vector component 2
L0_tx3 - Left hand sensor 0 texture vector component 3
L0_tx4 - Left hand sensor 0 texture vector component 4
L0_e - Left hand sensor 0 endorphins
Likewise for L1 through L4, R0 through R5 for the right hand, and B0 through B7 for the body's tactile sensors. Simply replace 'L0' with the appropriate sensor code.

Sx - x velocity of body
Sy - y velocity of body
BPx - absolute x coordinate of body
BPy - absolute y coordinate of body
LPx - x coordinate of left hand relative to body
LPy - y coordinate of left hand relative to body
RPx - x coordinate of right hand relative to body
RPy - y coordinate of right hand relative to body
A - rotation of the body in radians
V0.0_vq1 - Detailed visual sensor description vector component 1
V0.0_vq2 - Detailed visual sensor description vector component 2
V0.0_vq3 - Detailed visual sensor description vector component 3
V0.0_vq4 - Detailed visual sensor description vector component 4
Likewise for V0.1 through V30.20 for the detailed visual sensors, and P0.0 through P15.10 for the peripheral visual sensors.

2.5 Creating Items Dynamically

Through commands, you can create items in real-time.

2.5.1 Creating Menu Items Dynamically

Some items can be created during runtime by sending commands from the AI side. Simply send a command of the form:

{”messageType”:”dropItem”,”stringContent”:”n”,”floatContent”:0.0, ”vectorContent”:{”x”:0.0,”y”:0.0}, ”otherStrings”:[”n”],”messages”:[]}

Where:

- `stringContent n` - The object name of the item to generate.
- `vectorContent x,y` - The x,y positions where the item should initially appear.
- `otherStrings n` - an additional note (usually optional, but required for some item types. See item codes below for more details. If the item type doesn't require this, it is ignored.)

The object names are the same identifiers that are returned by sensors, e.g. the `objName` field in the `findObj` command.

2.5.2 Creating and Controlling Custom Objects

(As of version 0.1.7) It is possible to create your own objects in PAGI World, which you can then control by sending force commands. This is done by calling the command:

{”messageType”:”createItem”,”stringContent”:”fp”,”floatContent”:1.0,
”vectorContent”:{”x”:0.0,”y”:0.0}, ”otherStrings”:[”name,ph,r,e,k”],”messages”:[]}

- `stringContent: fp` - The path to the image to be used. The image must be either a png, jpg, or jpeg. The filename must have no commas or this string will be messed up! Also note that the image file is *not* saved with task files, so if you have a task file that has custom items, use relative directories and always include the image files with your task files. The image will be loaded at a size of 50 pixels per world unit. Also, the size of the bounding box will *always* be rectangular, regardless of the shape of the image. You may load png files if you want partially transparent images, but the bounding box will still be calculated based on the total image size.
- `vectorContent x,y` - The x,y positions where the item should initially appear.
- `floatContent m` - The initial mass of the item. For reference, the mass of an apple is 1, and the mass of a wall piece is 50.
- `otherStrings["name,py,r,e,k"]` a single string containing five parameters, where:
- `name` - The unique string identifier for this individual item. If you use a name that is already taken, then the previous item with this name will be deleted.
- `ph` - The physics material to use for this item as an int value. This can be one of the following:
  - 0 - Low friction, low bounciness
  - 1 - Normal friction, low bounciness (Most items should be this setting by default)
  - 2 - High friction, low bounciness
  - 3 - Low friction, high bounciness
  - 4 - Normal friction, high bounciness
  - 5 - High friction, high bounciness

- `r` - The initial rotation of the item, in radians as a float type.
- `e` - The amount of endorphins as a float type the object provides. If this is 0, then it will be have like a normal object. If it is non-zero, then after touching PAGI guy's body, it will disappear.
- `k` - Kinematic properties of the item as an int value. An object can be kinematic (meaning it will not move or rotate, even when sent force commands), fixed angle (meaning it will move but not rotate), and backgrounded (meaning it will not interact with other items in the world, and is invisible to all of PAGI guy's sensors). The value you send for `k` will determine which combination of properties the object has:
  - 0 - Kinematic, and backgrounded (kinematic automatically implies fixed angle).
  - 1 - Kinematic, and not backgrounded.
  - 2 - Not kinematic, fixed angle, and backgrounded.
  - 3 - Not kinematic, fixed angle, and not backgrounded.
  - 4 - Not kinematic, not fixed angle, and backgrounded.
  - 5 - Not kinematic, not fixed angle, and not backgrounded (the vast majority of items in the world will be this setting).
  - 6 - Kinematic, and *pseudo-backgrounded* (same as option 0 except this option makes the item visible to vision sensors. Like option 0, the object will not interact with other objects in any other way).

After an item has been created, you can send force commands to it using:

{"messageType":"addForceToItem","stringContent":"name","floatContent":0.0,
"vectorContent":{"x":0.0,"y":10.0}, "otherStrings":[],"messages":[]}

Note that you can only apply `addForceToItem` to items that you created using `createItem`!
Where:

- `stringContent name` - The unique name you gave to this item when you first created it. Errors will be returned if the name is not found (which might happen if the object was destroyed.
- `vectorContent x,fy` - The amount of force in the horizontal and vertical directions (NOT relative to the item's rotation)
- `floatContent fa` - The amount of rotational force to send to this item.

You are allowed to send forces to custom created items, but not to directly edit or modify their positions or velocities, in order to avoid unexpected behavior with PAGI World's physics engine. To get information about an item, call:

{"messageType":"getInfoAboutItem","stringContent":"name","floatContent":0.0,
"vectorContent":{"x":0.0,"y":0.0},"otherStrings":[],"messages":[]}

Where `name` is the unique name of this item. This will return an error message if the item was deleted or the name was not found. Otherwise, this command will return:

Returns: A string of the form `getInfoAboutItem,name,px,py,vx,vy` where:

- `name` - The name of the item
- `px,py` - The x and y coordinates of the item
- `vx,vy` - The x and y velocities of the item

Frequent use of `getInfoAboutItem` is **strongly discouraged**, as using any sort of perceptual input about things in the world that are not obtained through PAGI guy's sensors is against the intention of PAGI World. Do not get too attached to this command—future versions may severely limit its use somehow or remove it entirely!

You can also destroy custom created items, by calling:

{"messageType":"destroyItem","stringContent":"name","floatContent":0.0,
"vectorContent":{"x":0.0,"y":0.0},"otherStrings":[],"messages":[]}

Where `name` is the unique name of this item.

2.6 Others

2.6.1 Loading Tasks / Resetting Tasks

In order to load a new task, or to reset the current task (assuming it is saved as a .TSK file), use the command:

{"messageType":"loadTask","stringContent":"f","floatContent":0.0,
"vectorContent":{"x":0.0,"y":0.0},"otherStrings":[],"messages":[]}

where `f` in stringContent is a file path of a TSK file. To create TSK files, just press tab while PAGI World is loaded, and you can save the current task.

2.6.2 Printing Text to the Screen

To display text in the PAGI World console (obtained by pressing the "backquote" key, usually the same key that contains the tilde ( )), use the `print` command:

{"messageType":"print","stringContent":"S","floatContent":0.0,
"vectorContent":{"x":0.0,"y":0.0},"otherStrings":[],"messages":[]}

where `S` in stringContent is the string to print. Note that everything in S is printed until the end-of-line character is reached. So if you were to send the string `"this is a test, so is ``this''`, the console will display:

```
AI-side says:  this is a test, so is ``this''
```

2.6.3 Saying Things / Creating Speech Bubbles

You can have PAGI guy say things (and have them show up as speech bubbles). You can also create speech bubbles that are spoken by custom items (see Section 2.5.2), or have a speech bubble not attached to any speaker. Simply call the command:

{"messageType":"say","stringContent":"text","floatContent":10.0, "vectorContent":{"x":5.0,"y":-3.0},
"otherStrings":["speaker"],"messages":[]}

- `otherStrings: speaker` - If you want PAGI guy to be the speaker, this is `"P"`. If you want a custom item you created to be the speaker, this is the name of that item. Otherwise, if there is no speaker, use `"N"`.
- `stringContent: text` - The text for the speaker to say. Do not use commas in this text.
- `floatContent: duration` - The length (in seconds) that this bubble will be visible.
- `vectorContent: posX, posY` - The position of the speech bubble, relative to the speaker. So if you use `0,10`, and the speaker is PAGI guy, the speech bubble will appear slightly above where PAGI guy is. If you have no speaker, these coordinates are treated as absolute (so that `0,0` makes the bubble appear in the center of the screen).

You can see speech bubbles using PAGI guy's vision, and the `name` parameter returned will be the text in the bubble. You cannot otherwise interact with speech bubbles physically.

2.7 Unprovoked Notifications

There are many messages that may be sent from PAGI World to the AI side that are related to events that happened in the world which may not have been directly triggered by the AI side. For example, if a *trigger box* is activated, a message will be sent to the user. Likewise if a reflex is activated (see Section 2.4 for information on those notifications).

Messages sent to the AI side follow a generic format of:

{"type":"update","content":"message"} or: {"type":"error","content":"message"}

2.7.1 Endorphins: Rewards and Punishments

The basic reward / punishment system utilized by PAGI World is the *endorphin*, which is a property of items in the environment. If an item which has a nonzero endorphin value touches the body, a message of the following form is sent to the AI side:

{"type":"update","content":"RD,e,l"}

, where `e` is the endorphin value (positive is good, whereas negative is bad), and `l` is the location on the body which is reporting contact with the endorphin, which ranges from 0-7 depending on which body sensor was closest.

2.7.2 Spoken commands

PAGI World can also "hear" commands in plain text which are given to it by the environment. Currently the only way to do this is to type a message in the command box, which can be made visible through one of the menus (see Section **??**). The command received is of the form `SC,x`, where `x` is the string which was sent through the command box.

An alternate is to have PAGI guy "hear" things spoken by custom items in the environment by using speech bubbles (Section 2.6.3).

2.7.3 Triggers

One of the items that can be created using PAGI World's task creator is a *trigger box*, which is essentially a rectangular region of the environment that does not interact with the rest of the world in any physical way. The trigger boxes can be configured to fire if PAGI guy performs certain actions while within the box's boundaries (see Section **??**). This will trigger a message to be sent to the AI side of the form:

{"type":"update","content":"TB,n,c"}

where `n` is the name of the trigger box, and `c` is the action that the box detected within its boundaries. The possible values for `c` are:

- LE/LX - Left hand entered or exited the box, respectively
- RE/RX - Right hand entered or exited the box, respectively
- BE/BX - Body entered or exited the box, respectively
- RG/RR - Right hand gripped or released its grip in the box, respectively
- LG/LR - Left hand gripped or released its grip in the box, respectively
- OE/OX - Another object (not a hand or the body) entered or exited the box, respectively

Note: Remember that the trigger box must be configured to fire on each event type that you want it to watch for!

References