数理解析研究所講究録
1106 巻 1999 年 174-185

PDL robots represented in VRML environment*

Zoltán Nagylaki and Géza Horváth, 1999

Abstract

A robot is an autonomous unit, an object of the real world. It has the ability to move and reflect to the changes of its environment. The PDL is one of the robot controlling languages. Its simplicity makes it powerful and easy-to-use. The VRML is an 3D modelling language with solutions for not only static but dynamic worlds representation. It has the ability that it can be published through Internet, which probably makes it universal in the near future.

In this paper we discuss the way of simulation of PDL robots in VRML environment. We consider the properties of VRML for simulation puposes and properties of PDL to be simulated. We take the PDL structures one by one and consider their representation. We focus on the PDL working method and implement its behavior. We show a possible way of representation and discuss the aspects of a more complete solution.

Introduction to PDL

PDL is an acronym for Process Description language. PDL is a language developed for programming autonomous agents (robots) in a dynamical way. A PDL-program consists of a set of quantities and a set of processes operating over these quantities. Quantities are objects representing values. A quantity has a name, an upper and lower limit and initial value. These are defined in the PDL-program and can never be changed. The value of the quantity is dynamic only. After it is set to the initial value when the PDL program starts, it may change during execution of the program. It will stay within the bounds specified by the upper and lower limit. Quantities can be changed by the internal dynamics of the PDL program via the PDL statement add_value as well as by the external dynamics via connection slots. With add_value a process can propose the addition of a value to a quantity. At the end of a loop all proposals are collected and added to the quantity.

Processes are objects operating over quantities. They group a sequence of actions which must be taken one after another. Typically these actions include investigating and proposing new values to some quantities. All processes defined in a PDL-program run in parallel. This parallelism is simulated by the PDL-engine. Processes represent the internal dynamics of the autonomous agent.

*The research has been supported by the Hungarian National Foundation for Scientific Research Grant OTKA T-19501and T-030140.

The connection slots establish a link to the real world if these external variables represent actuators or sensors. Connection slots exists in two types: sensor and actuator slots. A variable assigned to a sensor slot is read by the PDL-engine and written by the robot. A variable assigned to an actuator slot is read by the robot and written by the PDL-engine. These variables are refreshed in every loop executed by the PDL-engine.

The PDL-engine is responsible for initialising, quitting, pausing and executing a PDL program. It manages accesses and proposals to quantities, the execution of processes and the communication with connection slots. It starts with the initialisation phase, where the connection slots are initialised and the init function is called. Then the PDL-loop is activated. This loop will loop through the functions: gets sensors' quantities, run processes (run every given process once), update quantities (calculate the new values from the proposals) and update connections (update linked quantities and external variables). The loop stops after a number of loops or runs forever as it was specified. At last, the quit function is called for normal shutdown.

Connectors are the communication mechanism of the PDL-engine. The system can communicate with the real world or some other applications (e.g. a problem solver) by connecting quantities to external variables. These external variables are set or read by a function which is called every loop by the PDL system. Typically, connection slots are used for connecting sensors and actuators to the system.

Because all processes run in parallel, it is impossible to set the value of a quantity directly. It avoids the interference of processes. The final value of a quantity is only given at the end of a loop and can therefore not be accessed during this loop. Therefore, the processes can only propose changes to quantities. This can be done with the add_value statement. During the PDL-loop every processes get the same values of the quantities, the values were set up at the begining of the PDL-loop. At the end of a cycle, all proposed changes are summed up and added to the current value. If the result is higher or lower than the upper or lower limit, the value is truncated to these limits respectively.

Introduction to VRML

VRML is an acronym for Virtual Reality Modeling Language. VRML is a file format for describing interactive 3D objects and worlds. VRML is designed to be used on the Internet too. VRML is capable of representing static and animated dynamic 3D and multimedia objects with hyperlinks to other media such as text, sounds, movies, and images. VRML supports an extensibility model that allows new dynamic 3D objects to be defined.

A VRML file consists of the following major functional components: the header, the scene graph, the prototypes, and event routing. The contents of this file are processed for presentation and interaction by a program known as a browser. The scene graph contains nodes which describe objects and their properties. These properties are fields, exposedFields, eventIns and eventOuts. All of them have a type. The field is a classic property with a characteristical value. The scenegraph contains hierarchically grouped geometry to provide an audio-visual representation of objects, as well as nodes that participate in the event generation and routing mechanism. Prototypes allow the set of VRML node types to be extended by the user. Some VRML nodes generate events in response to environmental changes or user interaction. Event routing gives authors a mechanism, separate from the scene graph hierarchy, through which these events can be propagated to effect changes in other nodes. Once generated, events are sent to their routed destinations in time order and processed by the receiving node. This processing can change the state of the node, generate additional events, or change the structure of the scene graph. For every node it is defined the set of events to receive and send. These are called eventIn and eventOut respectively. The routing can be given by route statements. Each route statement links one node's eventOut to another node's eventIn, where eventOut and eventIn must be the same type. A name can be assigned to a node by DEF reserved world. These nodes can be referenced by their names, it is achieved by the USE reserved word. The VRML has standard types. They can be single or multiple types. Single is one value only while multiple is a list of values. The atomic types can be string, boolean, integer, float or a node, a pair or triple or quadraple of integer or float and time. Some type of it is used later: SFBool, SFFloat and SFTime are boolean, floating number and timestamp types respectively. SFVec2f and SFRotation is a pair or quadraple of floating numbers respectively. MFRotation is a list of a quadraple of float numbers. MFString is a list of strings.

The script node

Script nodes allow arbitrary, author-defined event processing. An event received by a Script node causes the execution of a function within a script which has the ability to send events through the normal event routing mechanism, or bypass this mechanism and send events directly to any node to which the Script node has a reference. Scripts can also dynamically add or delete routes and thereby changing the event-routing topology.

Each Script node has associated programming language code, that is executed to carry out the Script node's function. The script node can be initialized and shut down. The script node has user-defined events handled by identically named functions.

Programming language of script nodes discussed later is Javascript. In Javascript the initialising function is initialize() and the function for shutting down is shutdown().

Architecture

There is a world where the robot is situated, it consists of the objects surrounding the robot. This world must be a VRML file. The robot's 3D objects must be included in this file too. Further, in this VRML file there is a script node named pdl. This node contains the PDL-engine e.g. the functions implementing the PDL statements. The user's PDL program which is a set of Javascript functions must be placed in the pdl script node after the PDL-engine's functions.

World nodes	World description nodes	
	Nodes sensed by sensors	
	Nodes affected by actuators	
Robot nodes	Nodes of robot body	
	Sensor nodes	
	Actuator nodes	
PDL script node	Variables of sensors and actuators	
	Nodes of sensors and actuators for referencing	
	PDL-engine	
	User's PDL program	Pdl_main function
		Functions of processes
		Other functions

Implementing PDL structures in VRML

The PDL has two elementary objects, the quantities and processes, these need to be represented. At a given quantity must store the name, the current value, the new value, the upper and the lower limit. This requires a record structure with five fields. At a given process the process's name must be stored. These records have to be arranged into a list or an array. Further the value of these variables must be conserved between the steps of the loop. VRML doesn't support lists of user defined types. Static variables are not supported by Javascript. The solution is to use VRML field variables with simple types and manage them in Javascript.

Thus we make two arrays for sensor's quantities, the first contains the sensor's names, the second contains the values. If we put these two arrays next to each other we have the array we need.

```
field MFString      sensornames   [" ", ...]
field MFRotation    sensors       [0.0 0.0 0.0 0.0, ...]
```

The values of sensors array are value of the quantity, new value, the upper and lower limit. For referencing a sensor first we find it in sensornames and with its index we index the sensors array. Furthermore the number of sensors is necessary to know, it is stored in the following field named num_sensors.

```
field SFFloat       num_sensors   0.0
```

The actuators are represented in the same way:

```
field MFString      actuatornames [" ", ...]
field MFRotation    actuators     [0.0 0.0 0.0 0.0, ...]
```

The number of actuators:

```
field SFFloat   num_actuators 0.0
```

The processes are represented using the same technique. There is an array for processes' names with elements of string type. The number of processes are also stored.

```
field MFString  processes      [" ", ...]
field SFFloat   num_processes 0.0
```

There are two more variables For PDL loop management. One for storing the fixed number of PDL loops and one for the loop counter.

```
field SFFloat   loop_counts        -1.0
field SFFloat   loop_counter        1.0
```

Finally one eventIn event is defined for the pdl script node. Whenever the PDL-loop have to be executed a boolean event has to be sent to it.

```
eventIn SFBool pdl_loop
```

The body of the script node starts with the PDL-engine. The PDL-engine consists of Javascript functions implementing PDL statements.

The add_sensor function create a new sensor quantity according to the given parameters. p_sensor specifies the name of the quantity, p_upper_limit and p_lower_limit gives the upper and lower limit of quantity respectively. Finally p_initial_value specifies the initial value of the quantity. The function increases the number of sensors and fills the appropriate elements of the sensors array.

```
function add_sensor(p_sensor, p_upper_limit, p_lower_limit,
        p_initial_value)
```

The get_sensor_index function is a technical function for resolving the name to an index value.

```
function get_sensor_index(p_sensor)
```

The value function gets a sensor p_sensor and returns its current value.

```
function value(p_sensor)
```

The add_actuator function creates a new actuator quantity named p_actuator and fills it appropriatly with the parameter values.

```
function add_actuator(p_actuator, p_upper_limit, p_lower_limit,
        p_initial_value)
```

The get_actuator_index is the pair of get_sensor_index function.

```
function get_actuator_index(p_actuator)
```

The add_value function serves for modifying the value of actuator. It gets an actuator name p_actuator and a value p_value. It adds value to the new value of the actuator. The p_value treated as signed number.

```
function add_value(p_actuator, p_value)
```

The actuator_value function gets a actuator p_actuator and returns its current value. It is not part of standard PDL it is an obvious extension.

```
function actuator_value(p_actuator)
```

The add_process function registers a new process named p_process.

function add_process(p_process)

The init_pdl function is for compatibility only. It has no effect.

function init_pdl ()

The run_pdl function sets the variables for the loop management. It initializes loop_counter to one and sets the loop_counts to p_loop_counts parameter. If the loop_counts is zero the PDL-loop never executes if it is smaller than zero then the PDL-loop runs forever.

function run_pdl (p_loop_counts)

The pdl_loop_begin function starts up the PDL-loop. First it checks PDL-loop management variables and decides whether the PDL-loop to be executed. Afterwards it processes the VRML objects of the sensors and calculates the values of the sensor quantities. Finally it sets the sensor values.

function pdl_loop_begin()

The pdl_loop_end function shuts down the PDL-loop. First it sets the quantities to their new values. In the array of actuators the new value is copied into the value if it is between the limits. If it is out of limits the value gets the appropriate limit's value. Afterwards every VRML objects belonging to a given actuator have to be changed according to the actuator's new value. Finally, it increases the loop counter.

function pdl_loop_end()

The pdl_loop function implements the kernel of the PDL-loop. Its parameters are not used, their purpose is to fit the VRML to Javascript interface only. First it calls the pdl_loop_begin function to start the loop. Afterwards it executes every processes defined by the user. Finally it finishes the PDL-loop by calling the pdl_loop_end function.

function pdl_loop(value, timestamp)

The standard named initialize function is executed when the VRML source is loaded, hence this is the place for definition and declarations. It simply calls the pdl_main function.

function initialize()

The pdl_main is the user's function. He has to put here the main part of PDL program. This function's body consists of calls of add_actuator, add_sensor, add_process, init_pdl and run_pdl functions.

function pdl_main()

After these functions the user's functions follow in the script node body. Some of these functions will be specified as processes. A process can contain calls of value and add_value functions.

The implementetion of connect_sensor and connect_actuator PDL statements discussed in the next section.

Sensors and actuators in VRML

After implementing the PDL-engine and providing the environment for PDL-program's execution, only one task is remained. The sensors and actuators have to be simulated. They are some objects in the VRML world belong to the robot. They provide values to the sensor quantities and the actuator quantities are assigned to these objects. Because of the simulation they also have to be simulated. Since they differ from case to case they cannot be modelled in a general way. We try to give a technique how to implement various sensors and actuators in the above discussed VRML-PDL environment. Furthermore, the VRML is a 3D modelling language only and it does not contain every property of the reality. For example the objects have not mass, temperature, impulse or energy. These physical properties also have to be simulated if a sensor or actuator requires it. In sum, not only the robot have to be simulated but the world too.

These physical properties have to be modelled by VRML objects, e.g. nodes. In the case of the sensors all information of the physical properties modelling the reality which can have affect the sensor have to be known to be able to evaluate what the sensor senses. From the results of evalution can be calculated the current value of the sensor's quantity. For this evaluation these nodes have to be accessible from the pdl script node. Therefore, all these nodes have to be defined in the script node definition part.

There are two ways for realising actuators. The first is similar to the case of sensors. That is the nodes of actuator have to make accessible. The proposed changes of properties of nodes are carried out at the time of writing their values in the body of script node.

The other way for realising actuators rely on the event routing mechanism of VRML. Besides of changing the properties of nodes directly, we send events them. It requires the appropriate definitions of eventIns and eventOuts and the necessary route statments. These eventOuts can be used as variables in the body of script node with restriction. These variables are not readable but values can be assigned to them in Javascript. In this case the proposed changes of properties of nodes of actuators are sent by the event routing mechanism after the script node body finishes.

These two techniques can be used together. Sometimes they can be extended by dynamically creating new nodes and routes and deleting existing ones. For example the new path of moving of a robot is realised by creating a new interpolator node and creating a new route statement from it to the robot and the old ones are deleted.

The above actuator and sensor representations are the solution frames of pdl_loop_begin function "get sensors" part and pdl_loop_end function "set actuators" part. In sum, implementation of sensors and actuators requires from the user creation of the VRML nodes of them at first, placing SFNode declarations for these nodes in the definition part of pdl script node at second, writing the "get sensors" part at third and writing the "set actuators" of VRML objects part at fourth. These impelemtations steps are the equivalents of the connect_sensor and connect_actuator pdl statements. This implementation of PDL-engine has two disadvantages. Firstly, the source of the engine itself is not hided from the user. Secondly, these definitions and codes overload the pdl script node's functionality.

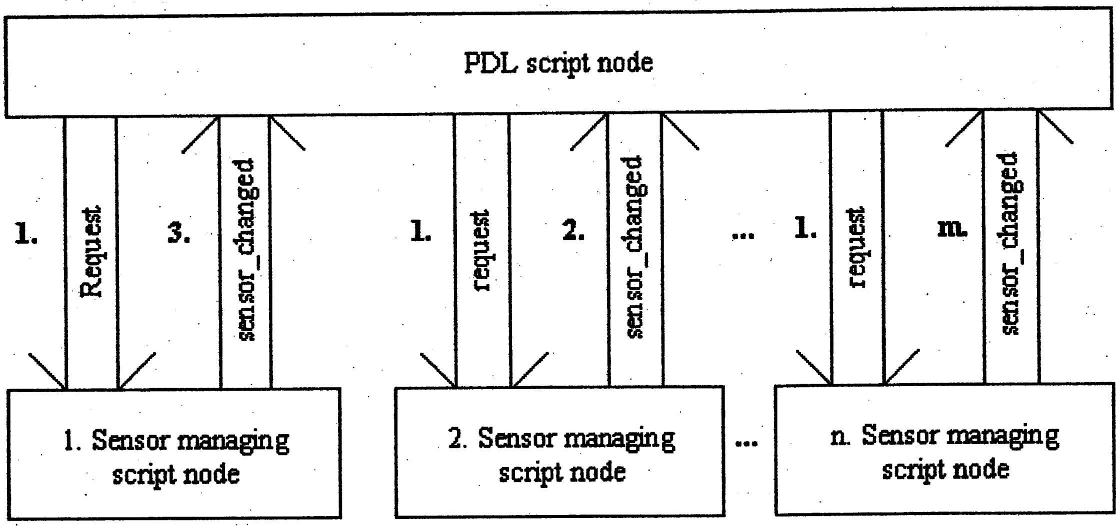
We provide a solution for connect_sensor and connect_actuator pdl statements, which solves the above problems. In this case each sensor has a managing script node. This script node gives the current value of the sensor when he is asked for. It means that the objects belonging to the sensor and required for evaluating the value of the sensor have to be acessible from this managing script node and not from the pdl script node. Further this managing script node have to know the upper and lower limits. It has a SFBool eventIn named request and has an SFRotation eventOut named sensor_changed which contains the sensor's values in the style of sensors array. The function named request calculates the sensor's value and sets the eventOut sensor_changed. So this calculation task is separated from the pdl script node. The new_value field of sensors array will be used for an identification number. This identification number have to be unique among the sensors. So the manging script node sends data and identifies who sent the event. After these, the connect_sensor pdl statement can be implemented:

```
connect_sensor(<sensor-name>, <identification-number>)
```

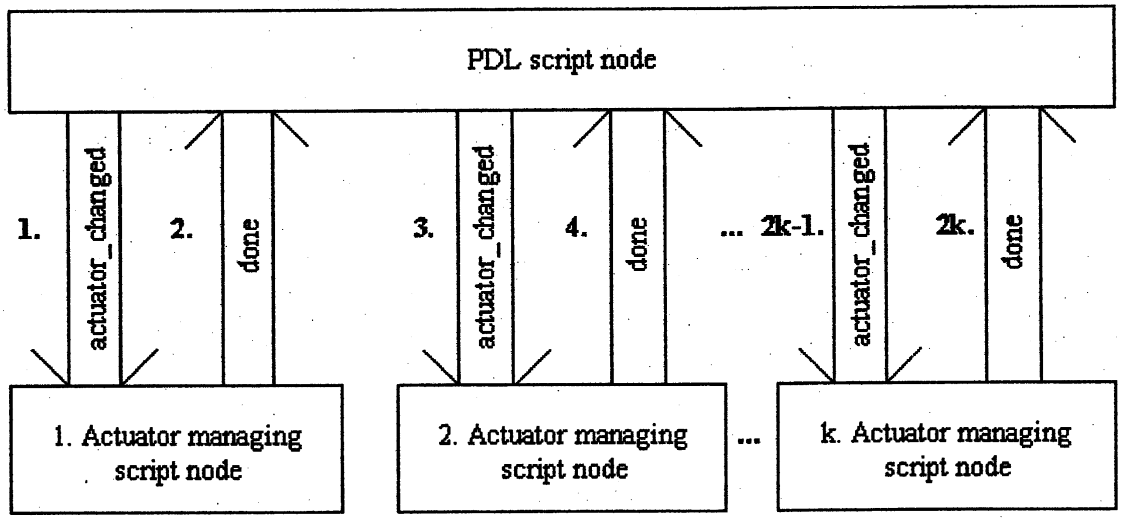
The first parameter is the name of the sensor. The second is the uniqe identification number. This pairings can be stored in a field typed array of SFVec2f type. In this array the first column is the identification number the second column is not the sensor name but its index in the sensors array. For having this index the <sensor-name> is resolved by get_sensor_index in the body of connect_sensor. In accordance the pdl script node has to be modified. It have to be extended by a new eventIn named set_sensor, which can receive the eventOut sensor_changed of managing script nodes. A new boolean eventOut named request have to add, by this the pdl script node can ask for sensor values. Some route statments have to be added.

```
Route <pdl-script-node>.request To <managing-script-node>.request
Route <managing-script-node>.sensor_changed To <pdl-script-
      node>.set_sensor
```

The PDL-loop work is also modified. It receives an event that a PDL-loop have to start. It examines the loop_counter and loop_counts and it decides if the PDL-loop need to run. If it needs to run then it generates request eventOuts to every sensor managing script node, and it finishes the processing of PDL-loop. Afterwards the managing nodes calculates the sensor values and sends them to the pdl script node. When it receives such an event then it copies the values to its sensors array. The received identification number and the array for connect_sensor are for determining the sensor quantity. The value is copied in accordance with the upper and lower limits. After it examines if it received events from every managing script node he had requested. It can be stored in a number field whose value is the number of managing script nodes answered. If everybody answered then continues the processing of PDL-loop. It can be decided by comparing the number of answers and the total number of sensors.



The actuators can be represented similarly. For each actuator a managing script node is created. In such a script node the objects belonging to a given actuator have to be made accessible.



The actuator managing script node has an eventIn named **set_actuator** of SFRotation type. It is used in the same style of **sensor_changed** eventIn in the case of sensors. Each actuator has an identification number which have to be unique in the circle of actuator managing script nodes. When it receives an event it checks if the value of the new_value field is equal to its identification number. If not there is nothing to do. If equals, then calculates and affects the VRML objects of actuators according to the received values. Afterwards it answers, it has an SFBool eventOut named **done**, and sends it. By the identification numbers the connect_actuator pdl statement can be defined:

connect_actuator(<actuator_name>, <identification-number>)

The first parameter is the name of the actuator. The second is the uniqe identification number. This pairings can be stored similarly to the connect_sensor. Further, the pdl script node has to be modified. It has an SFRotation eventOut named actuator_changed and an SFBool eventIn named done. The pdl_loop_end function of the PDL-engine has to be modified. The "set actuators' VRML objects" part starts with setting the values of the first actuator and send these values to it by finishing the pdl script node. It receives, processes and sends a done eventOut back. When the pdl-script node receives an eventIn done, then sets the values of the second actuator and sends it. It repeats until event sent to every actuators. It can be achieved by a counter similarly to the case of sensors. Finally, it sets the loop_counter and finishes. It requires route statements for each actuator managing script node:

```
Route <pdl-script-node>.actuator_changed To <managing-script-
      node>.set_actuator
Route <managing-script-node>.done To <pdl-script-node>.done
```

These route statements implies when the pdl script node sends an eventOut then this event arrives to every actuator managing script node. But one of them is the real addresse only. The identification number is required for singling it out. This broadcasting can be avoided by dynamically creating and removing route statements. Before sending an actuator_changed eventOut to the given actuator then we create a route to it and create a route back from it. After it sent the done eventIn back then we removes the routes to it. This can increase performance in the case of large number of actuators.

Example - Chicken selector robot

The Chicken selector is an industrial robot, its task is separation of chickens arriving on the production line. This separation is done by the chickens' weight which is in the limits of 1 to 10 kilograms. If a chicken arrives at the end of the production line then its weight is measured. If its weight is greater than 5 kilograms then it is dropped into the white hole. If its weight is less than 5 kilograms then it is dropped into the black hole. The environment of the robot consists of:

- A "chicken loader" machine, which puts a chicken on the production line from time to time.
- A production line, which transfers the chicken to the scale.
- A white hole for big chickens.
- A black hole for little chickens.

There is only one sensor, it is the scale. Their values are in the limits of 0 to 10 which reprents the mesaured weight. 0 represents that the scale is empty. 1 to 10 represents that there is a chicken on the scale and the value is its weight.

There is only one actuator, it is a "chicken pick up and drop" machine. It can have 0, 1 and 2 values. 0 is for stand by mode. If its value is set to 1 then it picks the chicken up, drops it to the black hole and goes back to stand by mode. If its value is set to 2 then does the same except for dropping the chicken into the white hall.

The sensor, the actuator and the only one process discussed later are defined in the pdl_main function.

```
function pdl_main() {
   init_pdl();
   add_actuator('Loader', 2, 0, 0);
   add_sensor('Libra', 10, 0, 0);
   add_process('set_Loader_process');
   run_pdl(-1.0);
}
```

It starts with initializing the PDL-engine. It creates an actuator for moving in the limits of 0 and 2 and defaults to 0. It creates a sensor for Libra in the limits of 0 and 10 and defaults to 0. It creates a process for controlling the behaviour of the robot. Finally it starts the PDL-engine with unlimited number of loops. The function implementing the process:

```
function set_Loader _process() {
  if (value(' Libra')>0) {
    if (value(' Libra')<=5) {
      add_value(' Loader',1-actuator_value(' Loader'));
    }
    else
      add_value(' Loader',2-actuator_value(' Loader'));
  }
  else
    add_value(' Loader',0-actuator_value(' Loader'));
}
```

This process checks the value of the sensor Libra at first. If its value is smaller than 0 then it actuator Loader to 0 for stand by mode. Since the values of actuators can not be set directly, signed values can be added to them only. Thus setting to 0 can be reached by adding the value of zero minus its current value to it. By the same technique is used for setting the actuator value to 1 or 2. If the value of sensor Libra is smaller or equal to 5 then we set actuator Loader to 1. It results that the chicken is dropped into the black hole. If the sensor value is greater than 5 then the actuator is set to 2 and the chicken arrives into the white hall.

The representation of the sensor and the actuator involves the definitions placed in the pdl script node. The pdl script node receives the value of the sensor from the single sensor managing node named Prog. The pdl script node sends the actuator value to the single actuator managing script node named Javsc2. There are two route statements for establishing this event change.

```
ROUTE    Prog.Ch_w     TO   PDL.Ping
ROUTE    PDL.Acts      TO   Prog.Javsc2
```

Furthermore, the pdl script node sets the sensor quantities to the values just received from the sensor managing script mode. It is in an array of float elements (with only one element in our case). Afterwards it runs the PDL loop and copies the actuator quantities to the eventOut which is sent to the actuator managing script node.

```
eventIn  MFFloat  Ping
eventOut  MFFloat Acts
```

```
function Ping( val ) {
    for(var i=1; (i<=num_sensors); i++){
      sensors[i][0]=val[i-1];
      sensors[i][1]=val[i-1];
    }
    pdl_loop();
    for(var i=1; (i<=num_actuators); i++)
      Acts[i-1]=actuators[i][0];
  }
```

As it can be seen the example follows the implementation method discussed in the second half of the „Sensors and actuators in VRML" above. Since it has one sensor and one actuator only, therefore it simplifies that model. There is no need for identifying the events so it does not broadcast and loads the browser unnecessarily. It does not need to consider the actuator managing script nodes one by one, and it doesn't need to wait for every sensor managing node to send their values.

References

1. Carey, R., Bell, G., Marrin, C.: ISO/IEC 14772-1:1997, Virtual Reality Modeling Language, (VRML97) , San Diego SuperComputing (SDSC), 1997 http://www.vrml.org/Specifications/VRML97
2. Carey, R., Bell, G.: The Annotated VRML 2.0 Reference Manual ISBN: 0-201-41974-2
3. Dewson, T., Irving A.D., Terdik, Gy.: Estimating of Volterra kernels in case of moment hierarchy and orthogonal representation, Comput. Math. Appl. 33 (1997), No 4, pp. 5-14.
4. Gál, Z., Korcsolay, Zs., Terdik, Gy.: UDNET: An Informatics Network at Universitas of Debrecen, Proc. EUNIS Congr. 95, Trend in Academic Information Systems in Europe, pp. 139-145.
5. Iglói, E., Terdik, Gy.: Bilinear Modelling of Chandler Wobble, Theory of Portability and its Applications, v. 44 (1997), 2, pp. 398-400
6. Inniss, M., Parent, J., Engelen, S.: Autonomous Systems 'Robot Football' Report, Vrije Universiteit Brussel, January 1998.
7. Spenneberg, D., Schlottmann, E., Höpfner, T., Christaller, T.: PDL Programming Manual, GMD Working Paper Nr. 1082, June 1997
8. Lewis, M.: Building 3D Virtual Environments, course of The Ohio State University, Course Number: Art 894Z12, Winter Quarter, 1998 http://www.cgrg.ohio-state.edu/mlewis/VRML/Class/syl.html
9. Nadeau, Dave., Moreland, J., Heck, M.: Introduction to VRML 2.0, On-Line Course Materials, The VRML 2.0 course notes from SIGGRAPH 96. http://www.sdsc.edu./siggraph96vrml/
10. San Diego Supercomputing Group, The Virtual Reality Modeling Language Version 2.0, ISO/IEC CD 14772 August 4, 1996 http://vrml.sgi.com/moving-worlds/spec/part1/