RoboCup Soccer Server 3D Manual

June 29, 2007

Contents

1 System Overview

To get started you should be somewhat familiar with the components of the system. The soccer simulation consists of three important parts: the server, the monitor and the agents.

1.1 Server

In order to work with the server you should be familiar with the SPADES [**?**][**?**] simulation middleware. Some important concepts you should know about: The server is responsible to start an agent process, i.e. it does not wait for an agent to connect as the 2D simulation does. The SPADES library uses a database that contains information how to start different agent types. It is called `agentdb.xml`, located in the `./app/simulator/` directory.

Agents connect via UNIX pipes to a SPADES Commserver. The use a length prefixed format to exchange messages. The Commserver in turn communicates with the server. In the default setup of the soccer server an integrated Commserver is started.

It is to possible start more than one Commserver in order to distribute agent processes across different systems. Please see the SPADES manual for further details about how to start and configure a remote Commserver. In this setup the 3d server has to be configured to wait until all Commservers are connected before it unpauses the simulation. The relevant settings are found in the server startup script `rcssserver3D.rb`.

These settings are 'Spades.RunIntegratedCommserver' and 'Spades.CommServersWanted'. The first setting configures if the integrated Commserver is started. Its default value is 'true'. The second setting gives the number of Commservers the server will wait for, before the simulation is initially unpaused. The integrated Commserver counts as one, so the default value here is 1.

1.2 Monitor

The default monitor is called `rcssmonitor3D-lite`, located in the directory `./app/rcssmonitor3d/lite`. It is also used to replay logfiles that the server automatically creates (use the `--logfile <filename>` option). The automatically generated logfile is called 'monitor.log'. You'll find it in the `Logfiles/` directory below the directory in which you started the server. A set of logfiles from 2004 RoboCup can be found at [**?**].

The implemented monitor protocol supports a command set to implement a trainer, i.e to automatically recreate test situations on the field and to evaluate an agents behavior. A 'monitor library' is provided to help implementing custom monitor and trainer applications, please see the `./app/rcssmonitor3d/lib` directory. The protocol between server and monitor is detailed further down in this text file.

A good starting point for your own agent implementations is the `agenttest` program in the `./app/agenttest/` directory. This agent implements a simple kick and run behavior.

2 Soccer Simulation

2.1 Soccer Team

Your soccer team consists of a number of robots with equal capabilities. The programs you should write to create a team exchange data with the (virtual) low level control system delivered with the robots. Both perceptors and effectors of your robots work with S-expressions, this is the syntax you know already from the 2D soccer simulator, or maybe also from your favorite programming language :).

2.2 Environment

Some technical data of the environment and of your new robots:

The playing field is a plane with FIFA standard soccer field size (length between 100m and 110m, width between 64m and 75m). Goal boxes and the ball are also standard FIFA size: goals are 7.32m wide; the ball has got a diameter of 0.222m and weighs between 0.41kg and 0.45kg. Because our agents are small and cannot jump, goals are only 0.5m high – the official FIFA height is 2.44m.

FIFA does not say too much about gravity (probably because they can't change it anyway), but in our simulation, gravity is fixed to $9.81\text{m}/s$.

Simulator steps are 0.01 seconds long. Connected monitors receive an update every 15th simulator step.

Many of the values contained in this text are subject to change and it is likely that this text does not always reflect the current state of affairs. You'll find the current set of constants in the setup script that the server executes at startup, please see `./app/simulator/rcssserver3D.rb`. After the first run of the server this file is copied to a directory under your home directory, called `/.rcssserver3d` and is read back from there on subsequent runs. Any experimental changes should happen there.

2.3 Players

In the current version of the simulator, robots are represented as spheres (until we can come up with a more sophisticated representation next year). The diameter of all robots is 0.44m, and each robots weighs 75kg.

Robots possess a kind of omnidrive, which adds some physical force to the robot body. By using the omnidrive, it is possible to accelerate into any direction, and it is also possible to jump very little. However, the omnidrive does only work if the robot is actually touching the soccer field. If you stop accelerating, robots will still move for a while, and you also cannot suddenly stop when moving with full speed (but you can use it for breaking). The maximum speed and the maximum height for jumping up if yet to be discovered.

When a player initially connects to the server you have to do two thing in order to get started. At first you must create the robot type you want to use during the game. Currently we are limited to the robot sphere described above. In later versions more sophisticated robot models may be available. It is the job of the create effector to select

and create one robot type at startup. Further the player must receive a number and join a team. This is the job of the init effector.

2.3.1 Create effector

When you initially connect to the simulator, your agent does not have any physical representation. The only thing your agent has got is a "CreatEeffector". The idea of the CreatEeffector is that you can request different effectors, perceptors or robot types. Currently, there is only one fixed robot type, so the CreateEffector ignores all parameters. For now, you should simply do "(create)" at the beginning and you will get the default robot type.

Example command: `(create)`

2.3.2 Init Effector

To set the team name and uniform number, you have to use the InitEffector. Prior to initializing, your effectors and perceptors will not work properly.

Syntax: `(init (unum <number>) (teamname <string>))`

Example: `(init (unum 7) (teamname RoboLog))`

2.3.3 Beam Effector

Similar to the kickeffector, the initial plan was not to introduce any artificial actions like "beaming" agents from one place to another place, similar to the "move" command in 2d soccer server. Still, the plan is to develop the soccer simulation so that beaming can disappear from the set of effectors.

However the problem was that due to limited time we had to do something about moving agents to their half in before kick off mode. To remove "beaming", the referee has to be extended to send yellow or red cards to players that don't behave properly... with your help :) we are going to work on this feature in the near future. Until then, beaming of agents is allowed in `beforekickoff` mode.

The beam effector expects three coordinates, but currently forces the third component to be zero, i.e. agents are only allow to move on the ground along the horizontal plane.

Syntax: `(beam <x> <y> <z>)`

Example: `(beam -6.6 0 0)`

2.3.4 Drive Effector

To use the omnidrive of the agent, you have to use the so called "DriveEffector", which takes a cartesian vector (x y z) with a maximum length of 100 units. The x-coordinate points towards the opponents team side of the field, z points up. With the DriveEffector, you set a kind of motor force, i.e. if you want to drive full speed for a while, it is sufficient to use the DriveEffector *once*. The force you set is applied at each simulator step until you change it again. The DriveEffector works reliable, there is a small error for forces

along each axis (each up to 2% of the applied force). The error is normally distributed around 0.0.

Using the omnidrive consumes battery. You get to know of battery states by reading the AgentStatePerceptor. If the battery is empty, the omnidrive will stop working. It is also possible to push away other robots. Using this feature to push away opponents is discouraged :).

Syntax: `(drive <x> <y> <z>)`
Example command: `(drive 20.0 50.0 0.0)`

2.3.5 Kick Effector

To move the ball, you have the option of simply using the robots to push the ball into a desired direction, or you can use the kickeffector to kick the ball. Originally, we did not intend to create an artificial kickeffector. However, to make use of the 3rd dimension, this was the easiest way. It is intended to remove this kind of kick effector in future versions (not this years' competition) in favor of a real physical device.

The kickeffector can accelerate the ball radially away from the robot body. The kickeffector takes an angle as first argument. This is the latitudal angle (in degrees) for accelerating the ball. It is restricted to a number between 0 and 50. The second argument indicates the kicking power and this is a number between 0 and 100. It is interpreted as the percentile of the maximum available power. The kickeffector adds a force and a torque to the ball. This happens over a fixed number of simulation steps. Currently 10 cycles are used. This corresponds to 1/10s simulation time. To kick the ball, the ball has to be very close to the robot, i.e. it has to be within the so called kickable margin of the player. Currently 0.04m are configured.

You cannot change the kicking angle in the horizontal plane. This means that you have to move the robot so that it can kick into the desired direction. Right now, the kickeffector is not very strong, because something like an offside rule is missing. It should also not be possible to move other robots by kicking the ball against them anymore. (at least not very much :) Like the DriveEffector, the kickeffector does only work if the robot touches the soccer field.

The kickeffector noise has the following parameters:

- The angle error in the x-y plane is quite low and normally distributed around 0.0 with sigma = 0.02. The
- The latitudal angle error is normally distributed around 0.0. This angle error is low with sigma = 0.9 at both extreme positions, i.e. 0 and at 50 degrees. Towards the middle of the range the angle error gets higher with sigma up to 4.5.
- The kick power error is normally distributed around 0.0 with sigma = 0.4

Syntax: `(kick <angle> <power>)`
Example command: `(kick 20.0 80.0)`

2.3.6 Vision Perceptor

Your robots possess a special omnicam with some smart image processing software attached :). Robots have a 360 degrees view. The VisionPerceptor delivers lists of seen objects, where objects are either others robots, the ball, or markers on the field. Currently there are 8 markers on the field: one at each corner point of the field and one at each goal post.

With each sensed object you get:

- The distance between the player and the object.
- The angle in the horizontal plane. Zero degree always points to the opponent goal.
- The latitudal angle. Here zero degree means horizontal.

Contrary to 2D soccer simulation, the vision system does not deliver object velocities. Objects can be occluded by other objects (this is not completely implemented yet). All distances and angles are given relative to the camera position. The camera is currently located at the center of the robot sphere.

The noise parameters of the vision system are as follows:

- A small calibration error is added to the camera position. For each axis, the error is uniformly distributed between -0.005m and 0.005m. The error is calculated once and remains constant during the complete match.
- Dynamic noise normally distributed around 0.0
  - distance error: sigma = 0.0965
  - angle error (x-y plane): sigma = 0.1225
  - angle error (latitudal): sigma = 0.1480

Syntax:

```
(Vision
        (<Type>
        (team <teamname>)
        (id <id>)
        (pol <distance> <horizontal angle> <latitudal angle>)
        )
)
```

Possible values are as follows:

- 'Flag' with `<id>` one of '1_l', '2_l', '1_r', '2_r'
- 'Goal' with `<id>` one of '1_l', '2_l', '2_l', '2_r'
- 'Player' with `<id>` being the uniform number of the player

Example Vision output:

```
(Vision (Flag (id 1_l) (pol 54.3137 -148.083 -0.152227)) (Flag (id
2_l) (pol 59.4273 141.046 -0.131907)) (Flag (id 1_r) (pol 61.9718
-27.4136 -0.123048)) (Flag (id 2_r) (pol 66.4986 34.3644 -0.108964))
(Goal (id 1_l) (pol 46.1688 179.18 -0.193898)) (Goal (id 2_l) (pol
46.8624 170.182 -0.189786)) (Goal (id 1_r) (pol 54.9749 0.874504
-0.149385)) (Goal (id 2_r) (pol 55.5585 8.45381 -0.146933)) (Ball (pol
6.2928 45.0858 -0.94987)) (Player (team robolog) (id 1) (pol 7.33643
37.5782 5.86774)))
```

2.3.7 Say Effector

To broadcast messages to other players, you have to use the SayEffector. Messages can be `sayMsgSize` (for now 512) characters long, where valid characters for say messages are the printing characters* except space and (). Messages players say can be heard within a distance of `audioCutDist` meters (for now 50) by members of both teams. The use of the SayEffector is only restricted by the limited capacity of the players of hearing messages. See the Hear Perceptor section for a list of server variables affecting these capacities.

* In the seven-bit ASCII character set, the printing characters are 0x20 to 0x7E.

Syntax:

```
(say <message>)
```

Example command:

```
(say player10_Pass)
```

2.3.8 Hear Perceptor

You get percepts from this perceptor when a player uses SayEffector and sends a message. The format of the aural sensor message from the is:

```
(hear <time> <direction in degree> <message>)
```

`<time>` indicates the current time.

`<direction in degree>` is relative direction to sender (without noise) if it is another player, otherwise it is `"self"` (without quotation mark).

`<message>` is the message. The maximum length is `sayMsgSize` bytes.

The server parameters that affect the Hear perceptor are:

- `audioCutDist`, default 50.0
- `hearMax`, default 2
- `hearInc`, default 1

- `hearDecay`, default 2
- `sayMsgSize`, default 512

A player can only hear a message if the player's hear capacity is at least `hearDecay`, since the hear capacity of the player is decreased by that number when a message is heard. Every cycle the hear capacity is increased with `hearInc`. The maximum hear capacity is `hearMax`. To avoid a team from making the other team's communication useless by overloading the channel the players have separate hear capacities for each team. With the current values this means that a player can hear at most one message from each team every second perceptor update.

If more messages arrive at the same time than the player can hear the messages actually heard are undefined (The current implementation choose the messages according to the order of arrival). This rule does not include messages from oneself. In other words, a player can hear a message from himself and hear a message from another player in the same perceptor output.

A message said by a player is transmitted only to players within `audioCutDist` meters from that player. For example, a defender, who may be near his own goal, can hear a message from his goal-keeper but a striker who is near the opponent goal can not hear the message.

Example Hear output:

```
(hear 0.8 -179.99 Test_1)
(hear 0.4 self Test_2)
```

2.3.9 GameStatePerceptor

The GameStatePerceptor tells you about the current status of the game. The first percept you get from this perceptor tells you about some of the game variables, like ball weight and field size additionally.

Syntax: `(GameState (<Name> <Value>) ...)`

Possible values for `<Name>` are:

- `time` gives the current simulation time (as a float value) passed in seconds
- `playmode` gives the current playmode as a string. Possible playmodes are `BeforeKickOff`, `KickOff_Left`, `ickOff_Right`, `PlayOn`, `KickIn_Left`, `KickIn_Right`, `corner_kick_left`, `corner_kick_right`, `goal_kick_left`, `goal_kick_right`, `offside_left`, `offside_right`, `GameOver`, `Goal_Left`, `Goal_Right`, `free_kick_left`, `free_kick_right`, `unknown`.

For an up to day list of all playmodes refer to `./plugin/soccer/soccertypes.h`

Example GameState output:

```
(GameState (time 0) (playmode BeforeKickOff))
```

2.3.10 AgentState perceptor

The AgentStatePerecptor tells you about the current state of your agent, currently its battery level and temperature.

Syntax:

```
(AgentState
        (battery <battery level in percent>)
        (temp <temperature in degree>)
)
```

Example AgentState output: `(AgentState (battery 100) (temp 23))`

3 Monitor and Trainer Protocol

The default monitor port for the soccer simulation is 12001. The server periodically sends you lines of text that contain S-Expressions. The monitor log file, that contains the recorded sequence of all expressions sent to the monitor is further used as the log file format. It is automatically generated in `Logfiles/monitor.log` relative to the server directory.

3.1 init Expression

Initially one `Init` expression is sent. An example `init` expression is given below. Note that S-Expressions from the server are received as a single line. Their are reformatted here for readability.

```
(Init
      (FieldLength 104)(FieldWidth 68)(FieldHeight 40)
      (GoalWidth 7.32)(GoalDepth 2)(GoalHeight 0.5)(BorderSize 10)
      (FreeKickDistance 9.15)(WaitBeforeKickOff 2)(AgentMass 75)
      (AgentRadius 0.22)(AgentMaxSpeed 10)(BallRadius 0.111)
      (BallMass 0.425878)(RuleGoalPauseTime 3)(RuleKickInPauseTime 1)
      (RuleHalfTime 300)
      (play_modes BeforeKickOff KickOff_Left KickOff_Right PlayOn
      KickIn_Left KickIn_Right corner_kick_left corner_kick_right
      goal_kick_left goal_kick_right offside_left offside_right
      GameOver Goal_Left Goal_Right free_kick_left free_kick_right)
      )
```

Each subexpression of the init expression is a name value pair that gives one parameter that the current instance of the simulation uses. The meaning of the different parameters:

- `FieldLength`,`FieldWidth`,`FieldHeight`: dimensions of the soccer field in meter
- `GoalWidth`, `GoalDepth`, `GoalHeight`: dimensions of the goals in meter

- `BorderSize`: the simulated soccer field is surrounded by an off field area. `BorderSize` gives the extra space in meters relative to the regular field dimensions in meters
- `FreeKickDistance`: gives the distance in meters that agents of the opposite have to adhere when a player carries out a free kick.
- `WaitBeforeKickOff`: gives the time in seconds the server waits before automatically starting the game
- `AgentMass`: the mass of each agent in `kg`
- `AgentRadius`: the radius of each agent in `m`
- `AgentMaxSpeed`: the maximum speed of each agent in `m/s`
- `BallRadius`: the radius if the ball in `m`
- `BallMass`: the mass of the ball in `kg`
- `RuleGoalPauseTime`: the time in seconds that the server waits after a goal is scored before switching to kick off playmode
- `RuleKickInPauseTime`: the time in seconds that the server waits after the ball left the field before switching to the kick in playmode
- `RuleHalfTime`: the length of one half time in seconds
- `play_modes`: lists the the different `play_modes` of the soccer simulation. Later on `play_modes` are referenced by a zero based index into this list.

3.2 info Expression

After the initial init message is sent only Info expressions are sent. These expressions contain the full state of the current simulation state. An example `Info` expression is given below:

```
(Info (time 0)(half 1)(score_left 0)(score_right 0)(play_mode 0) (P
(pos 0 0 0))(P (pos 0 0 0))(P (pos 0 0 0))(P (pos 0 0 0)) (P (pos 0 0
0))(P (pos 0 0 0))(P (pos 0 0 0))(P (pos 0 0 0)) (P (pos 0 0 0))(P
(pos 0 0 0))(F (id 1_l)(pos -52 -34 0)) (F (id 2_l)(pos -52 34 0))(F
(id 1_r)(pos 52 -34 0))(F (id 2_r)(pos 52 34 0)) (G (id 1_l)(pos -52
-3.66 0))(G (id 2_l)(pos -52 3.66 0)) (G (id 1_r)(pos 52 -3.66 0))(G
(id 2_r)(pos 52 3.66 0)) (B (pos 0 0 10)) )
```

Each subexpression of the `info` expression is a name value pair that contains information about one aspect of the current simulation state. Not all subexpressions are repeated. This concerns the positions of the field flags and the names of the two teams. This information is only sent once. Further game state information like the score count, and the current game state is only sent if it changed. The meaning of the different expressions:

- `Die`: notifies the monitor that the soccer simulation is about to terminate
- `time`: the current simulation time in seconds
- `half`: the current game half, 0 means the first, 1 means the second game half
- `score_left`, `score_right`: the score count of the left and right team respectively
- `team_left`, `team_right`: gives the names of the left and right team respectively; the information is only sent once as it remains static
- `play_mode`: the current play mode as 0 based index into the `play_modes` list given in the init expression
- `P`: gives information about a player. This expression may contain further subexpressions.
  - `s`: gives the team the player belongs to; 0 for the left, 1 for the right team
  - `id`: gives the uniform number of the player
  - `pos`: gives the position of the player as a three component vector
  - `last`: if this subexpression is present, the player was the last to touch the ball
  - `say`: this expression gives the string the player sent using the optional `SayEffector`
- `F`: gives information about a flag on the field. Information about a flag is only sent once, as it remains static
  - `pos`: gives the position of the flag as a three component vector
  - `id`: gives the name of the flag
- `B`: gives information about the ball
  - `pos`: gives the position of the ball as a three component vector
- `ack`: acknowledges a command that is carried out by the server; carries a user defined cooky string as parameter; see below for further explanation

3.3 Monitor Command Parser

A connected monitor can further send commands as S-Expressions to the server using the monitor connection. These commands allow a connected monitor to set the current playmode and to move players and the ball to arbitrary positions on the field. This allows for the implementation of trainer clients.

Supported expressions are:

- `(kickOff)`: start the soccer game, tossing a coin to select the team that kicks off first
- `(playMode <play_mode>)`: set the current playmode. Possible playmodes are given as strings in the `play_modes` expression of the init expression the monitor receives when it connects. Example: `(playMode corner_kick_left)`
- `(agent(team [R,L])(unum <uniform number>(pos <x,y,z>)(vel <vx,vy,vz))`. This expression sets the position and velocity of the given player on the field. Example: `(agent (team L)(unum 1)(pos -52.0 0.0 0.3)(vel 0.0 0.0 0.0))`
- `(ball (pos <x,y,z>))`: set the position of the ball on the field. Example: `(ball (pos 10,20,1))`
- `(dropBall)`: drop ball at its current position and move all players away by the free kick radius.
- `(getAck <cooky string>)`: experimental feature, currently disabled. Requests an `(ack <cooky>)` reply from the server. The server will send the answer as soon as the command is carried out. This is used to synchronize a trainer implementation wit the server. The getAck expression is appended behind one of the above commands. Example: `((kickOff)(getAck kicked_off))`

4 Scene Description Language

Spark provides access to the managed scene graph in several ways. Besides the internal C++ interface and external access via Ruby script language, an extensible mechanism for scene description languages is implemented. This allows for both a procedural and a description-based scene setup.

A scene is imported using one of any number of registered scene importer plugins, each supporting a different scene description language.

4.1 RubySceneGraph language

Currently one S-expression-based importer is implemented. This reference language is called *RubySceneGraph*. It maps the scene graph structure to the nesting of Lisp-like `s-expressions`.

An s-expression is a list of elements. Each element is either an `atom` or is itself another `list` of atoms. An atom is either a predefined keyword or a non empty string literal that has no further syntactic structure. The syntax of s-expressions, notated using EBNF is given in Listing 1.

```
character    -> "A" | ... | "Z" | "1" | ... | "9",
atom         -> character+
list         -> "(" s_expression* ")"
s_expression -> atom | list
```

Listing 1: EBNF notation of s-expressions

On the semantic side the *RubySceneGraph* interpreter recognizes a set of special atoms. The first atom in each subexpression determines its type. The set of keywords comprises four atoms that allow the interpreter to distinguishes five different expression types.

- The `RubySceneGraph` expression is the header expression of every scene graph file.
- The `node` expression declares a new scene graph node.
- The `importScene` expression is replaced with the content of another scene graph file.
- The `template` expression declares a set of parameters for a following scene fragment that can later be reused like a macro.
- Every other expression type is interpreted as a `method call`.

Apart from the different expression types listed above a replacement mechanism is implemented. Every atom literal starting with a dollar sign is interpreted as a template parameter and replaced with its actual value.

We shall describe the semantic of the different expression types below together with some small usage examples and a partial reference of available node types and methods.

4.2 File structure

The top level structure of a ruby scene file consists of two s-expressions. The first expression must be the header expression. It allows the parser to confirm the file type and to get information about the version of the used language.

The syntax of the header expression is `(RubySceneGraph <major Version> <minor Version>)`. Currently the only valid header states `0` for the major and `1` for the minor version.

The header is followed by a single s-expression that contains the scene graph body. Any further expression is discarded. The body expression consists of an optional single template expression and a set of node expressions. The resulting structure is outlined in listing 2. Note that lines starting with a semicolon are comment lines.

```
; the header expression
(RubySceneGraph 0 1)
(
 ; the body of the file starts here

 ; declare this file as a template
 (template $lenX $lenY $lenZ $density $material)

 ; declare the top level scene graph node
 (node Box
       ; children of the top level node go here
       (node DragController
       )
```

```
  )
)
```

Listing 2: File Structure

4.3 Node Expression

The scene graph consists of a tree of object instances, called nodes. Each node in the scene graph is declared with the `(node <ClassName>)` expression. The *ClassName* argument gives the name of a class registered to the Zeitgeist class factory system.

The semantic of a node expression is to instantiate a new scene graph object of the given class type. The importer therefore relies on the Zeitgeist class factory system to create the requested object. It is then installed as a child of the nearest enclosing node expression. If there is no enclosing node expression then the node is a top level node of the expressed scene graph.

The set of top level nodes are installed as children of the node below which the current graph is imported. This is either the global root node of the system, or an insertion point defined with the importScene expression within another scene graph file. The nesting of node expressions therefore defines directly the structure of the resulting scene graph with a very small syntactic overhead.

4.4 Scene Graph templates

The language further allows the reuse of scene graph parts in a macro like fashion. This enables the construction of a repository of predefined partial scenes, or complete agent descriptions. The macro concept is available through the `(importScene <filename> <parameter>*)` expression. This expression recursively calls the importer facilities of the system. It takes the nearest enclosing node expression as the relative root node to install the scene graph described within the given file.

Note that the given file must not necessarily be another RubySceneGraph file but any file type registered to the importer framework. This allows the nesting of scene graph parts expressed in different graph description languages. An example application of this feature is that parts of the resulting scene could be created by application programs better suited to create 3D models. By now, we do not exploit this feature yet.

The list of parameters given to the importScene expression is passed on to the responsible importer plugin. If another ruby scene graph file is imported that declares a template, they are substituted with its formal parameters.

A template declaration within the imported file has to meet the following syntax: `(template <parameterName>*)`. A parameter name is a string literal that is prefixed with a dollar sign, see listing 2 for an example declaration. All parameter names that follow within the body of the file are replaced with their actual content.

The usage example in listing 3 below assumes a `box.rsg` file. It uses that to construct boxes with varying sizes and colors according to the template expression given in listing 2.

```
(RubySceneGraph 0 1)
```

```
(
 (node Transform
       (importScene box.rsg 1 3 0.8 10 matRed)
 )
 (node Transform
       (importScene box.rsg 2 4 0.4 8 matBlue)
 )
)
```

Listing 3: importScene example

4.5 Method Calls

A node created with the node expression above can further be parameterized with method calls in order to modify its default properties. Every expression that does not match one of the expression types described above is interpreted as a method call. It is read as an s-expression that starts with the name of the function, followed by an optional list of parameters, i.e. `(<method name> <parameter>*)`.

Each method call is evaluated in the context of the nearest enclosing node expression. The semantic of a method call is to invoke the Ruby script interface of the corresponding C++ class, hence the name of this language.

This design decision allowed us to rapidly implement a complete scene description language during the development of the simulator, that is automatically extended as new methods are exported to Ruby, our primary scripting language.

Implementing a completely new scene description language with its own set of methods and property names would require the reimplementation of functionality that was otherwise readily available from our script interface.

An example usage is the setup of a transform node. These node types are used to position and orient nodes along a path in the scene graph relative to their respective parent node. The transform node therefore provides a method `SetLocalPos` to set the offset relative to its parent node. Likewise a box node provides a method to set the extents of the represented box.

```
(RubySceneGraph 0 1)
(
 (node Transform
       (setLocalPos 10 20 5)
       (node Box
             (setExtents 1 1 1)
       )
 )
)
```

Listing 4: Method call example

4.6 Language Reference

In this section we shall list the most common node types together with their most important methods. The complete reference would be to extensive and is a moving target as the simulator is constantly extended.

4.6.1 BaseNode

The `BaseNode` type is the base class for all scene graph nodes. The available methods are:

- `(importScene <string fileName>)` imports the given scene as described in section 4.4
- `(setName <string name>)` sets the user defined name of the node. This is used to reference nodes within path expressions.

4.6.2 Transform

The `Transform` node type positions and orients its child nodes relative to its respective nearest parent transform node. The available methods are available:

- `(setLocalPos <float x> <float y> <float z>)` defines the relative position
- `(setLocalRotation <float x> <float y> <float z>))` defines the relative rotation in degrees
- `(setLocalTransform <float m00> ... <float m33>)` defines the local 4x4 rotation matrix

4.6.3 SingleMatNode

The `SingleMatNode` is an abstract base class for all nodes that display an object using a single material property, e.g axis, box, sphere and capped cylinder. The following method is available:

- `(setMaterial <string name>)` uses the given material to draw the node

4.6.4 Axis

The `Axis` node is a SingleMatNode that displays axes of coordinates using colored perpendicular lines. The default is to draw lines of unit length. The following method is available:

- `(setSize <float size>)` sets the length of each draw axis line

4.6.5 Box

The `Box` node is a SingleMatNode that displays a cube with the given extents. The default is to draw a unit box. The following method is available:

- `(setExtents <float x>) <float y> <float z>` sets the extent of the box along the corresponding axis.

4.6.6 Sphere

The `Sphere` node is a SingleMatNode that displays a sphere with the given radius. The default is to draw a unit sphere. The following method is available:

- `(setExtents <float radius>)` sets the radius of the sphere

4.6.7 CCylinder

The `CCylinder` node is a SingleMatNode that displays a capped cylinder with the given radius and length. The default is to draw a cylinder with unit length and radius. The following method is available:

- `(setParams <float radius> <float length>)` sets the length and the radius of the capped cylinder

4.6.8 Body

The `Body` node represents the physical aspect of a simulated object. The following methods are available:

- `(enable)` allows the body to take part in the physical simulation
- `(disable)` prevents the body to take part in the physical simulation
- `useGravity <bool f>)` specifies whether the body is influenced by gravity
- `setSphere <float density> <float radius>)` sets the mass distribution of for a sphere
- `setSphereTotal <float mass> <float radius>)` sets the mass distribution for a sphere
- `setBox <float density> <float size>)` sets the mass distribution for a box
- `setBoxTotal <float mass> <float size>)` sets the mass distribution for a box
- `setCylinder <float density> <float radius> <float length>)` sets the mass distribution for a cylinder

- `setCylinderTotal <float mass> <float radius> <float length>)` sets the mass distribution for a cylinder
- `setCappedCylinder <float density> <float radius> <float length>)` sets the mass distribution for a capped cylinder
- `setCappedCylinderTotal)` sets the mass distribution for a capped cylinder
- `setMass <float mass>)` sets the total mass of the body
- `setVelocity <float x> <float y> <float z>)` sets the initial velocity of the body
- `setAngularVelocity <float x> <float y> <float z>)` sets the initial angular velocity in degrees

4.6.9 Collider

The `Collider` is an abstract base class for all supported collision primitives. It further manages the set of CollisionHandler child nodes that react to collision events detected by a Collider instance. The available method is:

- `(addCollisionHandler <string ClassName>)` installs a collision handler node below the Collider. This i a convenience function that has the same effect as installing the collision handler using a node expression. If no collision handler is installed the default ContactJoint handler is installed during runtime.

4.6.10 BoxCollider

The `BoxCollider` is a Collider node that implements the box collision primitive. The default is to create a unit sized box collider. The available method is:

- `(setBoxLengths <float x> <float y> <float z>)` sets the extents of the collision primitive.

4.6.11 SphereCollider

The `SphereCollider` is a Collider node that implements the sphere collision primitive. The default is to create a unit sized sphere collider. The available method is:

- `(setRadius <float radius>` sets the radius of the collision primitive.

4.6.12 CCylinderCollider

The `CCylinderCollider` is a Collider node that implements the capped cylinder collision primitive. The default is to create a capped cylinder with unit length and radius. The available method is:

- `(setParams <float radius> <float length>)` sets the radius and length of the collision primitive.
- `(setRadius <float radius>)` sets the radius of the collision primitive.
- `(setLength <float length>)` sets the length of the collision primitive.

4.6.13 CollisionHandler

The `CCylinderCollider` node is an abstract base class for handlers that take action in response to a collision. To each Collider instance one ore more CollisionHandlers are registered. There are no methods available in this base class.

4.6.14 ContactJointHandler

The `ContactJointHandler` is a CollisionHandler node that creates an ODE contact joint between the two bodies associated with the two affected collision primitives. Contact joints are used to resolve a collision, i.e. to generate the appropriate force that does not allow two bodies to interpenetrate. The available methods are available:

- `(setContactBounceMode <bool set>)`
- `(setContactBounceValue <float value>)`
- `(setMinBounceVel <float velocity>)`
- `(setContactSoftERPMode <bool set>)`
- `(setContactSoftERP <float erp>)`
- `(setContactSoftCFMMode <bool set>)`
- `(setContactSoftCFM <float cfm>)`
- `(setContactSlipMode <bool set>)`
- `(setContactSlip <float slip>)`
- `(setContactMu <float mu>)`

All methods above enable and set the corresponding value of the ODE contact structure. Please refer to the ODE user guide [**?**] for details.

4.6.15 RecorderHandler

The `RecorderHandler` is a CollisionHandler that accumulates collision information of the Collider it belongs to. It aids for example in the implementation of bumper sensors or in the implementation of game rules. There it is used to detect if agents are in certain areas of the playing field. These areas are for example expressed as box colliders. No methods further are available through the script interface.

4.6.16 Joint

The `Joint` node is an abstract base class for all joints. It defines a relationship (a constraint) that is enforced between two bodies so that they can only have certain positions and orientations relative to each other.

Note that the joint geometry parameter setting functions should only be called after the joint has been attached to bodies, and those bodies have been correctly positioned, otherwise the joint may not be initialized correctly. If the joint is not already attached, these functions will do nothing.

Note that joint nodes are positioned and orientated according to the parent transform nodes. The available methods are:

- `attach(<string path1> <string path2>)` attaches the joint to some new bodies. If the joint is already attached, it will be detached from the old bodies first. To attach this joint to only one body, omit the second path parameter. This effectively attaches the body to the static environment.

  The path expression follow the common Unix syntax and are relative to the joint node. Object instances are referred by the name set using the SetName function, using `"/"` as the path separator and `".."` to refer to the parent node.

  - `(setAnchor <float x> <float y> <float z>)` sets the anchor point in local coordinates relative to the joint node.

- `setHighStopDeg)`
- `(setLowStopDeg)`
- `(setHighStopPos)`
- `(setLowStopPos)`
- `(setBounce)`
- `(setCFM)`
- `(setStopCFM)`
- `(setStopERP)`
- `(setSuspensionERP)`
- `(setSuspensionCFM)`
- `(setLinearMotorVelocity)`
- `(setAngularMotorVelocity)`
- `(setMaxMotorForce)`

The last block of methods above set the corresponding values of the ODE joint structure. Please refer to the ODE user guide [**?**] for details.

4.6.17 FixedJoint

The `FixedJoint` node represents a joint that maintains a fixed relative position and orientation between two bodies, or between a body and the static environment.

4.6.18 BallJoint

The `BallJoint` node represents a *ball and socket joint* connects two bodies at an anchor point. It enforces a constant distance of the two bodies to this anchor. Further it keeps a constant facing of each body towards the anchor point. The two bodies can otherwise rotate freely around the anchor point.

4.6.19 HingeJoint

The `HingeJoint` node represents a *hinge joint*. This joint type connects two rigid bodies along a single axis that passes through a defined anchor point. The axis is fixed to the z axis in the local coordinate system of the node. This can be adjusted using a parent transform node. Like the ball and socket joint it enforces a constant distance and facing with respect to the anchor point. The rotational freedom is however restricted to the defined axis.

4.6.20 Hinge2Joint

The `Hinge2Joint` node represents a *two-hinge joint*. It acts like two hinge joints connected in series. Each hinge joint defines a different hinge axis but shares the same anchor point. The two axis are fixed to the x and z axis in the local coordinate system of the node. This can be adjusted using a parent transform node. This joint type is commonly used to simulate the steering wheel of a car. In this case the first axis allows the steering of the wheel and the second axis allows the wheel to rotate.

4.6.21 UniversalJoint

The `UniversalJoint` node represents a so called *universal joint*. It acts like a ball and socket joint that constrains an extra degree of rotational freedom. The universal joint works on two perpendicular axes, one defined on each body. These axes, fixed to the x and z axes in the local coordinate system of the joint are forced to stay perpendicular. This means that the rotation of the two bodies about the axis perpendicular to the two other axes will be equal. So, if any of the two bodies turns around the axis, the other will turn as well.