Robot–Box 2D Simulator and Library

(User manual)

Domen Šoberl

October 30, 2018

Contents

Chapter 1

Robot–Box Simulator

1.1 Library reference

1.1.1 Class Playground

A Playground object is a client that connects to the simulator and from the standpoint of the client program, represents the simulator. Multiple instances can be created and connected to multiple simulators on local or remote hosts. Multiple instances can also connect to the same simulator, acting as separate clients. All instances have equal control over all objects in the simulator therefore it is a standard practice to construct only one Playground object in a single client program.

Creation and disposal via interface classes

- `[IPlayground*] IPlayground::create()`

  Constructs a Playground object that connects to local simulator using default port number 24980.

- `[IPlayground*] IPlayground::create(host)`

  Constructs a Playground object that connects to the simulator running on the given host, using default port number 24980.

- `[IPlayground*] IPlayground::create(host, port)`

  Constructs a Playground object that connects to the simulator running on the given host, using given port number.

  Arguments:

`[string]`	`host`	Host name to connect to.
`[string]`	`port`	Port number.

- `IPlayground::dispose(playground)`

  Disposes the given Playground object. Also closes the connection if still open.

  Arguments:

`[IPlayground*]`	`playground`	Playground object to dispose.

Creation and disposal via wrapper classes

- `CPlayground()`

  Constructs a Playground object that connects to local simulator using default port number 24980.

- `CPlayground(host)`

  Constructs a Playground object that connects to the simulator running on the given host, using default port number 24980.

- `CPlayground(host, port)`

  Constructs a Playground object that connects to the simulator running on the given host, using given port number.

  Arguments:

`[string]`	`host`	Host name to connect to.
`[string]`	`port`	Port number.

- `CPlayground(playground)`

  Constructs a new Playground wrapper object that holds the same reference to IPlayground as the given one. Reference count to that Playground interface is increased.

  Arguments:

`[CPlayground]`	`playground`	Object representing the simulator.

- `~CPlayground()`

  If this is the only object holding the same IPlayground reference, it closes the connection and disposes the Playground. If other CPlayground objects exist that hold the same IPlayground reference, the reference count is decreased and only the wrapper class is disposed.

- `operator = (playground)`

  Assigns the IPlayground reference held by the given Playground wrapper. The reference count to the previously held Playground is decreased and object disposed if it reaches 0. The reference count to the newly held Playground is increased.

  Arguments:

`[CPlayground]`	`playground`	Object representing the simulator.

Creation and disposal in Python

- `rbsim2d.Playground()`

  Constructs a Playground object that connects to local simulator using default port number 24980.

- `rbsim2d.Playground(host)`

  Constructs a Playground object that connects to the simulator running on the given host, using default port number 24980. The argument is given as a string.

- `rbsim2d.Playground(host, port)`

  Constructs a Playground object that connects to the simulator running on the given host, using given port number. Both arguments are given as a string.

A Playground is disposed automatically when no variable references it. Note that this also closes the connection with the simulator.

Playground members and methods

- `[bool] connect()`

  Establishes TCP connection with the simulator using host name and port number given at construction. If unsuccessful, error message can be retreived by `Playground::getError()`.

  Returns:

TRUE	If connection is successful.
FALSE	If connection is unsuccessful.

- `[void] disconnect()`

  Stops all running tasks and closes the connection. If connection is not established, this method has no effect.

- `[bool] connected()`

  Checks if connection is open.

  Returns:

TRUE	If connection is open.
FALSE	If connection is closed.

- `[string] getError()`

  Returns the last error message. The Call to this method does not delete the message.

- `[void] stopAllActivities()`

  Stops all running tasks. This means all the robots that are currently executing some task in a separate thread, terminate the thread and stop all motors. This also happens when calling `disconnect()`.

- `[void] waitEvent()`

  Blocks until an event from the simulator is received. This can mean a movement of an object or a mouse event if mouse capturing in the simulator is enabled.

- `[bool] waitEvent(time)`

  Blocks until an event from the simulator is received, but for not longer than the given amount of time.

  Arguments:

`[int]`	`time`	Maximum amount of waiting time in milliseconds.

  Returns:

TRUE	If an event was received.
FALSE	If timeout occurred.

- `[void] waitAllTasks()`

  Blocks until all robots complete their current tasks. If no task is running, this method has no effect.

- `[bool] waitAllTasks(time)`

  Blocks until all robots complete their current tasks, but for not longer than the given amount of time.

  Arguments:

`[int]`	`time`	Maximum amount of waiting time in milliseconds.

  Returns:

TRUE	If no task is running.
FALSE	If timeout occurred.

- `[bool] bodyExists(name)`

  Checks if the body with the given name exists. Body is an object or a robot.

  Arguments:

`[string]`	`name`	The name of the body.

  Returns:

TRUE	If the body exists.
FALSE	If the body does not exist.

  Python: A list of names can be given and the corresponding list of `True / False` values is returned.

- `[bool] robotExists(name)`

  Checks if the robot with the given name exists. Robots are a subset of bodies.

  Arguments:

`[string]` `name` The name of the robot.

Returns:

TRUE If the robot exists.
FALSE If the robot does not exist.

Python: A list of names can be given and the corresponding list of `True / False` values is returned.

- `[vector] listBodies()`

  Returns the list of names of all the bodies. Returns a `std::vector<std::string>` in C++ and a list of strings in Python.

- `[vector] listRobots()`

  Returns the list of names of all the robots. Returns a `std::vector<std::string>` in C++ and a list of strings in Python. Robots are a subset of bodies.

- `[vector] listItems()`

  Returns the list of names of all bodies that are not robots. Returns a `std::vector<std::string>` in C++ and a list of strings in Python.

- `[Body] getBody(name)`

  Returns the body with the given name. Robots can be treated as bodies. If such a body does not exist a *dummy* body is returned. Any operation on a *dummy* body has no effect on the simulator.

  Arguments:

  `[string]` `name` The name of the body.

  Python: A list of names can be given and the corresponding list of bodies is returned.

- `[Body] getRobot(name)`

  Returns the robot with the given name. If such a robot does not exist a *dummy* robot is returned. Any operation on a *dummy* robot has no effect on the simulator.

  Arguments:

  `[string]` `name` The name of the robot.

  Python: A list of names can be given and the corresponding list of robots is returned.

- `[Canvas] getCanvas(name)`

  Returns the canvas of an image identified by the given name. If an image with this name

does not yet exist, it is created.

Arguments:

`[string]` `name` The name of the image.

Python: A list of names can be given and the corresponding list of canvases is returned.

- `[Mouse] getMouse()`

  Returns the object that represents the mouse in the simulator. Each Playground contains only one object of class Mouse.

1.1.2 Class Position

Class Position is a simple container for pairs of floating point numbers x and y. It is an instantiable class and therefore not denoted in the interface / wrapper style as other classes. It is used only in C++ library while in Python tuples `(x, y)` are used instead.

- `[double] x, y`

  Public members to store a position within the simulator.

- `Position()`

  Constructs a position object with $x = 0, y = 0$.

- `Position(x, y)`

  Constructs a position object with x and y set to the given values.

  Arguments:

  `[double]` `x` Value of x.
  `[double]` `y` Value of y.

- `Position(position)`

  Constructs a position object with the same values as the given position.

  Arguments:

  `[Position]` `position` A position to copy its values from.

- `operator = (position)`

  Sets the position to the given values.

  Arguments:

  `[Position]` `position` A position to copy its values from.

- `operator = (value)`

  Sets both x and y to the given value. It is used mostly to simplify resetting a position to $(0, 0)$.

  Arguments:

`[double]`	`value`	New value for x and y.

- `operator == (position)`

  Tests two positions for equality. Two positions are equal if they match in their x and y values.

  Arguments:

`[Position]`	`position`	A position to compare to.

  Returns:

TRUE	If positions are equal.
FALSE	Otherwise.

- `operator != (position)`

  Tests two positions for inequality. Two positions are not equal if they fail to match in their x or y values.

  Arguments:

`[Position]`	`position`	A position to compare to.

  Returns:

TRUE	If positions are not equal.
FALSE	Otherwise.

1.1.3 Class Body

Objects of class Body represent objects within the simulator, or more precisely their bodies, since textures used with objects are not visible to the library. Robots can be treated as bodies. Items and Phantoms are all considered bodies and cannot be distinguished explicitly. For each simulated object exactly one Body is created within a Playground. It is not possible to create a Body object manually. To obtain the body of choice use `Playground::getBody(name)`.

- `[string] name()`

  Returns the name of the body as specified within the simulator.

- `[double] radius()`

  Returns the radius of the circle that circumscribes the body. It is the largest distance from its central point to its border. This can be useful to simplify collision detection for symmetrical robots.

- `[Position] position()`

  Returns the current position of the body. The position of the body is defined as the position of its *central* point as specified in the simulator. Note that this point is not necessarily the body's geometrical center.

  Python: Position is represented as a tuple (x, y).

- `[double] orientation()`

  Returns the current orientation of the body in angular degrees. Returned value is in the range of [0, 360).

- `[double] speed()`

  Returns the current translational speed of the body as measured internally by the library.

- `[double] rotationalSpeed()`

  Returns the current rotational speed of the body as measured internally by the library.

- `[bool] occupies(position)`

  Checks whether the body occupies the given position. A position $p = (x, y)$ is occupied if the point p lies within a body.

  Arguments:

`[Position]`	`position`	Position to be checked.

  Python: A list of positions $[(x_0, y_0), (x_1, y_1), \ldots, (x_{n-1}, y_{n-1})]$) can be given and the corresponding list of `True / False` values is returned.

- `[int] countEdges()`

  Returns the number of edges the shape of this body has, i.e. rectangle 4, triangle 3, circle 1, etc. Can also be used for non-convex shapes. Edges are internally indexed with integers from 0 to `countEdges()` -1.

- `[double] edgeLength(edge)`

  Returns the length of an edge. The edge is determined by the given index. If an invalid index is given, value 0 is returned.

  Arguments:

`[int]`	`edge`	Index of an edge.

  Python: A list of indices can be given and the corresponding list of floating point numbers is received.

- `[double] edgeDirection(edge)`

Returns the direction of an edge expressed in angular degrees. Edge direction is defined as the direction of its perpendicular ray towards the interior of the body. The edge is determined by the given index. Returned value is in the range of $[0, 360)$. If an invalid index is given, value 0 is returned.

Arguments:

`[int]`	`edge`	Index of an edge.

Python: A list of indices can be given and the corresponding list of floating point numbers is received.

- `[Position] borderPoint(edge, tau)`

  A point on the border of a body can be determined by two values: the index of an edge and a normalized parameter `tau` $\in [0, 1]$ on that edge. The center of an edge is identified as `tau` $= 0.5$, while `tau` $= 0$ and `tau` $= 1$ represent its endpoints. Traveling in the CCW direction along the border, `tau` $= 0$ represents the first and `tau` $= 1$ the last point on an edge.

  Arguments:

`[int]`	`edge`	Index of an edge.
`[double]`	`tau`	Normalized edge parameter.

  Python: Each parameter can be given either as a single value or a list. In each case, a Cartesian product is made and all resulting points are returned in a single list.

- `[Position] borderPoint(direction)`

  A point on the border of a body can be determined by ray direction from the body's center. The resulting point is the intersection of the ray with the border. If there are more such points (non-convex shape), the farthest one is returned.

  Arguments:

`[double]`	`direction`	Direction of the ray in angular degrees.

  Python: A list of directions can be given and the corresponding list of points is returned.

- `[double] distanceByCenter(position)`

  Returns the distance between the given position and the body's *central* point. Note that the central point is not necessarily the body's geometrical center.

  Arguments:

`[Position]`	`position`	Position to which the distance is measured.

  Python: A list of positions $[(x_0, y_0), (x_1, y_1), \ldots, (x_{n-1}, y_{n-1})])$ can be given and the corresponding list of distances is returned.

- `[double] distanceByCenter(body)`

  Returns the distance between central points of two bodies.

  Arguments:

`[Body]`	`body`	The body to which the distance is measured.

  Python: A list of bodies can be given and the corresponding list of distances is returned.

- `[double] distanceByBorder(position)`

  Returns the distance between the given position and the body's border, i.e. the closest point on the border.

  Arguments:

`[Position]`	`position`	Position to which the distance is measured.

  Python: A list of positions $[(x_0, y_0), (x_1, y_1), \ldots, (x_{n-1}, y_{n-1})]$) can be given and the corresponding list of distances is returned.

- `[double] distanceByBorder(body)`

  Returns the distance between two bodies according to their borders, i.e. the smallest distance between two points belonging to different bodies.

  Arguments:

`[Body]`	`body`	The body to which the distance is measured.

  Python: A list of bodies can be given and the corresponding list of distances is returned.

- `place(position, orientation)`

  The body is *teleported* to the given position and orientation.

  Arguments:

`[Position]`	`position`	Goal position.
`[double]`	`position`	Goal orientation in angular degrees.

- `waitEvent()`

  Blocks until the body moves. This poses a risk to permanently block the client, therefore the use of `waitEvent(time)` is rather encouraged.

- `[bool] waitEvent(time)`

  Blocks until the body moves, but for not longer than the given amount of time.

  Arguments:

`[int]` `time` Maximum amount of waiting time in milliseconds.

Returns:

TRUE If an event was received.
FALSE If timeout occurred.

- `drawBorder(canvas)`

  Draws the border of the body on the given canvas. Default colors are used. This can be useful to depict certain body positions.

  Arguments:

  `[Canvas]` `canvas` The canvas to draw the border on.

- `drawBorder(canvas, lineColor)`

  Draws the border of the body on the given canvas using the given color for edges and the default color for corners. Color `Canvas::Color::none` can be used to prevent drawing lines.

  Arguments:

`[Canvas]`	`canvas`	The canvas to draw the border on.
`[Canvas::Color]`	`lineColor`	Color of edges.

  Python: Colors are given as strings. Available values are defined as Canvas class members (e.g. `canvas.white`, `canvas.red`, `canvas.none`, etc.)

- `drawBorder(canvas, lineColor, dotColor)`

  Draws the border of the body on the given canvas using the given line and dot colors. Color `Canvas::Color::none` can be used to prevent drawing lines or dots.

  Arguments:

`[Canvas]`	`canvas`	The canvas to draw the border on.
`[Canvas::Color]`	`lineColor`	Color of edges.
`[Canvas::Color]`	`dotColor`	Color of corners.

  Python: Colors are given as strings. Available values are defined as Canvas class members (e.g. `canvas.white`, `canvas.red`, `canvas.none`, etc.)

1.1.4 Class Robot

Robot is a body with controlling functionalities and some other additional properties. The basic functionality is to send output to the robot's left and right wheel. Advanced algorithms to move and rotate the robot or make it follow a given path are also integrated. The robot performs its tasks in a separate thread which can be interrupted at any time. Using this library it is therefore easy to make robots work in parallel without the need to do any thread programming. To obtain the robot of choice use `Playground::getRobot(name)`.

PID profile

Certain control tasks such as moving and rotating require the use of PID controller. When initiating such tasks, the user must provide PID coefficients for which the subclass `Robot::PidProfile` is used.

- `Robot::PIDProfile(double kp, double ki, double kd)`

  Constructs a PidProfile object with the given K_p, k_i, k_d coefficients.

  Arguments:

`[double]`	`kp`	Coefficient for *proportion* component.
`[double]`	`ki`	Coefficient for *integral* component.
`[double]`	`kd`	Coefficient for *derivative* component.

- `[double] kp, ki, kd`

  Members `kp`, `ki` and `kd` are publicly accessible.

Robot members and methods

- `[Position] frontPoint()`

  Returns the position of the robots *bumper*. It is the farthest point from the center of the body in the direction of forward movement. This direction is defined in the simulator under the `<wheels>` tag in the `world.xml` file.

- `[double] headingAngle()`

  Heading angle is defined as *orientation + direction of forward movement* and expressed in degrees. This is useful because the robot may be designed in such a way that its forward movement is not necessarily in the direction of its orientation. The returned value is in the range of $[0, 360)$.

- `[double] angleTo(position)`

  Returns the angular offset of the robot's heading angle from the given position. The returned value is expressed in degrees an in the range of $[-180, 180)$. Its meaning can be interpreted as the amount of rotation needed for the robot to be heading towards the given point.

  Arguments:

`[Position]`	`position`	The point of reference.

  Python: A list of positions $[(x_0, y_0), (x_1, y_1), \ldots, (x_{n-1}, y_{n-1})]$) can be given and the corresponding list of angles is returned.

- `[double] angleTo(body)`

  Returns the angular offset of the robot's heading angle from the central point of the given body. The returned value is expressed in degrees an in the range of $[-180, 180)$. Its meaning

can be interpreted as the amount of rotation needed for the robot to be heading towards the given body.

Arguments:

`[Body]`	`body`	The body of reference.

Python: A list of bodies can be given and the corresponding list of angles is returned.

- `[double] steeringRadius()`

  As the robot moves, the library measures its steering radius. Holding different but steady output values on both wheels, the robot moves in a circle. This circle is computed at every position update. This value is used successfully with PID controllers to control the movement of the robot.

- `[Position] steeringGrip()`

  Additionally to the steering radius, the center of the steering circle can be obtained using this method.

- `setWheels(left, right)`

  Sets the output values to the left and right wheel, making the robot move. Each output value should be in the range of $[-1, 1]$, where a positive value represents forward motion, negative value backward motion, and zero value stops the wheel. The values are held until changed.

  Arguments:

`[double]`	`left`	Output power to the left wheel.
`[double]`	`right`	Output power to the right wheel.

- `moveTo(position, tolerance, pidProfile)`

  Starts the task of moving the robot to the given position. This is done in a separate thread, so the call returns immediately. If some task is still executing it is terminated and new task is started. The robot stops when its distance from the given position is within the given tolerance.

  A PID controller is used, therefore the PID profile (coefficients K_p, K_i and K_d) has to be specified. The coefficients are best determined experimentally, as different values exhibit different robot behavior. The `Robot::PidProfile` class is used to define the PID profile.

  Arguments:

`[Position]`	`position`	Goal position.
`[double]`	`tolerance`	Goal distance tolerance.
`[PidProfile]`	`pidProfile`	PID profile to be used for the task.

  Python: To specify the PID profile, a tuple (K_p, K_i, K_d) of floating point numbers is used.

- `rotateTo(angle, tolerance, pidProfile)`

Starts the task of rotating the robot towards the given orientation until the given tolerance is reached. The given angle applies to the robot's orientation and not its heading direction. The task is done in a separate thread, so the call returns immediately. If some task is still executing it is terminated and new task is started. To move and rotate the robot, same PID profile can be used.

Arguments:

`[angle]`	`position`	Goal orientation expressed in degrees.
`[double]`	`tolerance`	Goal angle tolerance expressed in degrees.
`[PidProfile]`	`pidProfile`	PID profile to be used for the task.

Python: To specify the PID profile, a tuple (K_p, K_i, K_d) of floating point numbers is used.

- `rotateBy(angle, tolerance, pidProfile)`

  Starts the task of rotating the robot by the given angle until the given tolerance is reached. This is done in a separate thread, so the call returns immediately. If some task is still executing it is terminated and new task is started. This method is useful in combination with the `angleTo(position)` and `angleTo(body)` to rotate the robot toward a chosen position or body. The same PID profile can be used as in the above two cases.

  Arguments:

`[angle]`	`position`	Goal orientation expressed in degrees.
`[double]`	`tolerance`	Goal angle tolerance expressed in degrees.
`[PidProfile]`	`pidProfile`	PID profile to be used for the task.

  Python: To specify the PID profile, a tuple (K_p, K_i, K_d) of floating point numbers is used.

- `follow(path, trackWidth)`

  Starts the task of following the given path and returns immediately. If some task is still executing it is terminated and new task is started. The classic *Pure Pursuit* algorithm is used which stops when the robot hits or passes the goal point. Information on track width (distance between left and right wheel) is used to decide on the left and right output at every step. Track width is not known by the simulator and must be set by user. It can be measured by calling the `Robot::findTrackWidth()` method.

  Arguments:

`[Path]`	`path`	The path to follow (see Chapter 1.1.7).
`[double]`	`trackWidth`	Distance between the left and the right wheel.

- `[double] findTrackWidth()`

  Measures robot's track width (distance between the left and the right wheel) by performing a simple maneuver. The method blocks until the task is done and returns the measured value.

- `[bool] busy()`

  Checks is the robot is currently executing a task.

Returns:

TRUE	If the robot is executing a task.
FALSE	If the robot is not executing a task.

- `stop()`

  Terminates the current task and stops the robot. If no task is being executed, the robot is stopped by setting values 0 on both wheels.

- `waitTask()`

  Blocks until the current task is done. If no task is being executed, the method returns immediately. If the robot is set into a position in which it can never reach the given goal, this method blocks forever. Therefore the use of `waitTask()` is rather encouraged.

- `waitTask(time)`

  Blocks until the current task is done, but for not longer than the given amount of time. If no task is being executed, the method returns immediately.

  Returns:

TRUE	If the task is done or no task is being executed.
FALSE	If timeout occurred.

- `enableVisualization(canvas)`

  Enables visualization of robot's tasks on the given canvas. Each task may use a different way of visualizing its process. Default line and dot colors are used.

  Arguments:

`[Canvas]`	`canvas`	The canvas to draw on.

- `enableVisualization(canvas, lineColor)`

  Enables visualization of robot's tasks on the given canvas using the given color for lines and the default color for dots. Color `Canvas::Color::none` can be used to prevent drawing lines.

  Arguments:

`[Canvas]`	`canvas`	The canvas to draw on.
`[Canvas::Color]`	`lineColor`	Color of lines.

  Python: Colors are given as strings. Available values are defined as Canvas class members (e.g. `canvas.white`, `canvas.red`, `canvas.none`, etc.)

- `enableVisualization(canvas, lineColor, dotColor)`

  Enables visualization of robot's tasks on the given canvas using the given line and dot colors. Color `Canvas::Color::none` can be used to prevent drawing lines or dots.

  Arguments:

[Canvas]	canvas	The canvas to draw on.
[Canvas::Color]	lineColor	Color of lines.
[Canvas::Color]	dotColor	Color of dots.

Python: Colors are given as strings. Available values are defined as Canvas class members (e.g. `canvas.white`, `canvas.red`, `canvas.none`, etc.)

- `disableVisualization()`

  Disables visualization of robot's tasks.

1.1.5 Class Canvas

Canvas enables the user to draw simple shapes in the simulator. Shapes belong to different images, so when an image is deleted, shapes not belonging to that image remain visible. Images are identified by their names and can be shared among clients. It is, however, not possible to obtain a list of existing images so it is possible for two clients to unknowingly use the same image if some naming consensus is not set.

The Canvas class represents an image within a simulator. It is obtained by calling the `Playground ::getCanvas(name)` method. If an image with such a name does not yet exist, it is created and its reference stored by the Playground object. When Playground disconnects it also deletes all its images in the simulator.

- `[enum] Color`

  Is a publicly accessible type that specifies all colors supported by the simulator. Possible values are the following: `none`, `white`, `black`, `red`, `green`, `blue`, `cyan`, `yellow`, `magenta`.

  Python: Values are used as strings.

- `clear()`

  Clears all shapes that belong to that Canvas.

- `drawDot(p, color)`

  Draws a dot.

  Arguments:

[Position]	p	Position of the dot.
[Canvas::Color]	color	Color of the dot.

- `drawLine(p1, p2, color)`

  Draws a segment.

  Arguments:

[Position]	p1	First endpoint.
[Position]	p2	Second endpoint.
[Canvas::Color]	color	Color of the line.

- `drawCircle(center, radius, color)`

  Draws a circle.

  Arguments:

`[Position]`	`center`	Center of the circle.
`[double]`	`radius`	Radius of the circle.
`[Canvas::Color]`	`color`	Color of the circle.

- `drawArc(center, radius, alpha, beta, color)`

  Draws an arc. The arc is always drawn in the CCW direction from angle `alpha` to angle `beta`.

  Arguments:

`[Position]`	`center`	Center of the arc.
`[double]`	`radius`	Radius of the arc.
`[double]`	`alpha`	Starting angle in angular degrees.
`[double]`	`beta`	Ending angle in angular degrees.
`[Canvas::Color]`	`color`	Color of the arc.

1.1.6 Class Mouse

The Mouse object allows the user to receive mouse input. This is possible only if in the simulator the *mouse capture* mode is enabled. All clients receive mouse data simultaneously. Mouse object is obtained by calling `Playground::getMouse()`. Each Playground holds one Mouse instance.

Mouse members

- `[enum] ButtonType`

  Type of button which can be one of the following: `undefined`, `left`, `right`.

  Python: Values are used as strings.

- `[enum] ButtonState`

  State of a button: `released`, `pressed`.

  Python: Values are used as strings.

Click events

Mouse click events are represented by subclass `Mouse::ClickEvent`. The class stores the following three values:

Position	Position where the click occurred.
ButtonType	Mouse button that was pressed or released.
ButtonState	Type of event which is either `pressed` or `released`.

`Mouse::ClickEvent` is an instantiable class and therefore not denoted in the interface / wrapper style as other classes. However, values can only be read but not set. Constructors are provided so the object can be returned via the stack.

1. `ClickEvent()`

   Creates a *dummy* click event with `ButtonType` set to `undefined`.

2. `ClickEvent(clickEvent)`

   Creates a copy of the given click event.

3. `operator = (clickEvent)`

   Assigns the values of the given click event.

4. `[Position] position()`

   Return the position where the click occurred.

5. `[ButtonType] buttonType()`

   Return the button that was pressed or released.

6. `[ButtonState] buttonState()`

   Returns whether the button was pressed or released.

Mouse methods

- `[Position] position()`

  Returns the current mouse position.

  Python: Position is represented as a tuple (x, y).

- `[ButtonState] buttonState(buttonType)`

  Returns the current state of the given button.

  Arguments:

`[ButtonType]`	`buttonType`	The mouse button to check.

- `[int] wheelPosition()`

  Returns the current wheel position as the number of ticks from its initial position. The returned number can be positive or negative, depending of the direction of rotation.

- `resetWheelPosition()`

  Resets the wheel position to 0.

- `[bool] clickEventsPending()`

  Returns `true` if click events are waiting to be processed. The history of up to 16 such events

can be stored in the queue. This is necessary because click events are obtained by polling and can therefore be missed by the client. When the queue is full, the oldest event is removed.

- `[ClickEvent] popClickEvent()`

  Removes and returns the oldest click event from the queue.

- `flushClickEvents()`

  Clears the queue of recent click events.

- `waitEvent()`

  Blocks until a mouse event is received. This can be moving the mouse, clicking its button or rotating its wheel.

- `[bool] waitEvent(time)`

  Blocks until a mouse event is received, but for not longer than the given amount of time.

  Arguments:

`[int]`	`time`	Maximum amount of waiting time in milliseconds.

  Returns:

TRUE	If an event was received.
FALSE	If timeout occurred.

- `[ClickEvent] waitClickEvent()`

  Blocks until a mouse click event is received.

  Returns:

ClickEvent	The received click event.

- `[ClickEvent] waitClickEvent(time)`

  Blocks until a mouse click event is received, but for not longer than the given amount of time.

  Arguments:

`[int]`	`time`	Maximum amount of waiting time in milliseconds.

  Returns:

ClickEvent	The received click event. If timeout occurred, a *dummy* click event is returned with `buttonType` set to `undefined`.

- `[ClickEvent] waitClickEvent(buttonState)`

  Blocks until a mouse click event is received, considering only the given type of click events.

  Arguments:

[ButtonState]	buttonState	Events to consider which are either `pressed` or `released`.

Returns:

ClickEvent	The received click event. If timeout occurred, a *dummy* click event is returned with `buttonType` set to `undefined`.

- `[ClickEvent] waitClickEvent(buttonState, time)`

  Blocks until a mouse click event is received, but for not longer than the given amount of time. Only the given type of click events are considered.

  Arguments:

[ButtonState]	buttonState	Events to consider which are either `pressed` or `released`.
[int]	time	Maximum amount of waiting time in milliseconds.

  Returns:

ClickEvent	The received click event. If timeout occurred, a *dummy* click event is returned with `buttonType` set to `undefined`.

1.1.7 Class Path

Class `Path` represents a curve that the robot can follow.

Creation and disposal via interface classes

- `[IPath*] create()`

  Creates a void path.

- `[IPath*] clone(path)`

  Creates a copy of the given path.

  Arguments:

[IPath*]	path	Path to be cloned.

- `dispose(path)`

  Disposes the given path.

  Arguments:

[IPath*]	path	Path to be disposed.

Creation and disposal via wrapper classes

- `CPath()`

  Creates a void path.

- `CPath(path)`

  Creates a copy of the given path.

  Arguments:

`[CPath]`	`path`	Path to be cloned.

- `~CPath()`

  Disposes the path.

Creation and disposal in Python

- `rbsim2d.Path()`

  Creates a void path. The path is automatically disposed when no variable references it.

- `[rbsim2d.Path] clone(path)`

  Creates a copy of the given path.

  Arguments:

`[rbsim2d.Path]`	`path`	Path to be cloned.

Path members and methods

- `clear()`

  Clears the curve and makes the path empty.

- `appendStart(position)`

  Adds a straight segment to the front, between the given position and start position. The current start position becomes a waypoint and the given position becomes the new start position.

  Arguments:

`[Position]`	`position`	New start position.

- `appendGoal(position)`

  Adds a straight segment to the back, between the goal position and the given position. The current goal position becomes a waypoint and the given position becomes the new goal position.

  Arguments:

`[Position]`	`position`	New goal position.

- `appendStartSmooth(position)`

Adds an arc to the front, between the given position and start position, making the joint smooth. The current start position becomes a waypoint and the given position becomes the new start position.

Arguments:

`[Position]`	`position`	New start position.

- `appendGoal(position)`

  Adds an arc to the back, between the goal position and the given position, making the joint smooth. The current goal position becomes a waypoint and the given position becomes the new goal position.

  Arguments:

`[Position]`	`position`	New goal position.

- `[bool] empty()`

  Checks if path is empty.

  Returns:

TRUE	If path is empty.
FALSE	otherwise.

- `[double] length()`

  Returns the length of the path.

- `[Position] startPosition()`

  Returns the start position.

- `[Position] goalPosition()`

  Returns the goal position.

- `[Position] position(distance)`

  Return the positions at the given distance from the start position. If the given value is smaller than 0 or greater than `length()`, the position on the computed extension is returned. The extension is a smooth continuation of the path.

  Arguments:

`[double]`	`distance`	Distance along the curve from the start position.

- `[Position] position(distance, reference)`

  Return the positions at the given distance from the given reference position. If the reference position does not lie on the curve, a projection to the curve is computed and used as a reference. Distance can be negative. If the distance exceeds the distance to the start or goal point,

the position on the computed extension is returned. The extension is a smooth continuation of the path.

Arguments:

`[double]`	`distance`	Distance along the curve from the reference position.
`[Position]`	`reference`	Reference position.

- `[double] travelDistance(position)`

  Returns the distance along the curve from the start position to the given reference position. If the given position does not lie on the curve, its projection to the curve is computed and used as a reference.

  Arguments:

`[Position]`	`position`	The reference position.

- `draw(canvas)`

  Draws the curve on the given canvas using default colors.

  Arguments:

`[Canvas]`	`canvas`	The canvas to draw the curve on.

- `draw(canvas, lineColor)`

  Draws the curve on the given canvas using the given color for edges and the default color for waypoints. Color `Canvas::Color::none` can be used to draw only waypoints.

  Arguments:

`[Canvas]`	`canvas`	The canvas to draw the curve on.
`[Canvas::Color]`	`lineColor`	Color of the curve.

  Python: Colors are given as strings. Available values are defined as Canvas class members (e.g. `canvas.white`, `canvas.red`, `canvas.none`, etc.)

- `draw(canvas, lineColor, dotColor)`

  Draws the curve on the given canvas using the given line and dot colors. Color `Canvas::Color::none` can be used to prevent drawing the curve or waypoints.

  Arguments:

`[Canvas]`	`canvas`	The canvas to draw the curve on.
`[Canvas::Color]`	`lineColor`	Color of the curve.
`[Canvas::Color]`	`dotColor`	Color of waypoints.

  Python: Colors are given as strings. Available values are defined as Canvas class members (e.g. `canvas.white`, `canvas.red`, `canvas.none`, etc.)

1.1.8 Class Pathfinder

Class `PathFinder` finds a smooth path on a polygon from a start to a goal configuration, so that no collision with obstacles occurs.

Creation and disposal via interface classes

- `[IPathFinder*] create()`

  Creates a new pathfinder.

- `[IPathFinder*] clone(pathFinder)`

  Creates a copy of the given pathfinder.

  Arguments:

`[IPathFinder*]`	`pathFinder`	Path finder to be cloned.

- `dispose(pathFinder)`

  Disposes the given pathfinder.

  Arguments:

`[IPathFinder*]`	`pathFinder`	Path finer to be disposed.

Creation and disposal via wrapper classes

- `CPathFinder()`

  Creates a pathfinder.

- `CPathFinder(pathFinder)`

  Creates a copy of the given pathfinder.

  Arguments:

`[CPathFinder]`	`pathFinder`	Path finder to be cloned.

- `~CPathFinder()`

  Disposes the pathfinder.

Creation and disposal in Python

- `rbsim2d.PathFinder()`

  Creates a pathfinder. The pathfinder is automatically disposed when no variable references it.

- `[rbsim2d.PathFinder] clone(path_finder)`

  Creates a copy of the given pathfinder.

Arguments:

`[rbsim2d.PathFinder]` `path_finder` Path finder to be cloned.

PathFinder members and methods

- `clearObstacles()`

  No body is regarded an obstacle.

- `addObstacle(body)`

  Regards the given body as an obstacle.

  Arguments:

  `[Body]` `body` A body to be regarded as an obstacle.

- `removeObstacle(body)`

  The given body is no longer regarded as an obstacle. If the given body was not regarded as an obstacle, the calling of the method has no effect.

  Arguments:

  `[Body]` `body` A body to no longer be regarded as an obstacle.

- `setSpacing(spacing)`

  Sets the spacing between the path and the obstacles. The radius of the robot plus some tolerance should be used for a robot to track the path with no collision.

  Arguments:

  `[double]` `spacing` Spacing between the path and obstacles.

- `[Path] find(startPosition, startOrientation, goalPosition, goalOrientation)`

  Finds and returns a smooth path leading from the start to goal position. Start and goal orientation define initial and final orientation of the robot tracking the path.

  Arguments:

`[Position]`	`startPosition`	Start position.
`[double]`	`startOrientation`	Initial orientation of the robot.
`[Position]`	`goalPosition`	Goal position.
`[double]`	`goalOrientation`	Final orientation of the robot.

- `[Path] find(startPosition, startOrientation, goalPosition)`

  Finds and returns a smooth path leading from the start to goal position. Start orientation defines initial orientation of the robot tracking the path. The goal orientation is chosen automatically for an optimal result.

Arguments:

`[Position]`	`startPosition`	Start position.
`[double]`	`startOrientation`	Initial orientation of the robot.
`[Position]`	`goalPosition`	Goal position.

- `[Path] find(robot, goalPosition, goalOrientation)`

  Finds and returns a smooth path leading from the current position and orientation of the robot to goal position. The robot tracing the resulting path should arrive at the goal position under the given goal orientation.

  Arguments:

`[Robot]`	`robot`	The robot to track the resulting path.
`[Position]`	`goalPosition`	Goal position.
`[double]`	`goalOrientation`	Final orientation of the robot.

- `[Path] find(robot, goalPosition)`

  Finds and returns a smooth path leading from the current position and orientation of the robot to goal position. The final orientation of the robot tracing the resulting path is chosen automatically for an optimal result.

  Arguments:

`[Robot]`	`robot`	The robot to track the resulting path.
`[Position]`	`goalPosition`	Goal position.

- `enableVisualization(canvas)`

  Draws the roadmap that serves as the search space to finding the resulting path on the given canvas using default colors.

  Arguments:

`[Canvas]`	`canvas`	Canvas to draw on.

- `enableVisualization(canvas, lineColor)`

  Draws the roadmap that serves as the search space to finding the resulting path on the given canvas. The given color is used to draw lines and default color is used to draw dots. Color `Canvas::Color::none` can be used to prevent drawing lines.

  Arguments:

`[Canvas]`	`canvas`	Canvas to draw on.
`[Color]`	`lineColor`	Color of lines.

  Python: Colors are given as strings. Available values are defined as Canvas class members (e.g. `canvas.white`, `canvas.red`, `canvas.none`, etc.)

- `enableVisualization(canvas, lineColor, dotColor)`

Draws the roadmap that serves as the search space to finding the resulting path on the given canvas. The given line and dot colors are used to draw lines and dots. Color `Canvas::Color::none` can be used to prevent drawing either.

Arguments:

`[Canvas]`	`canvas`	Canvas to draw on.
`[Color]`	`lineColor`	Color of lines.
`[Color]`	`dotColor`	Color of dots.

Python: Colors are given as strings. Available values are defined as Canvas class members (e.g. `canvas.white`, `canvas.red`, `canvas.none`, etc.)

- `disableVisualization()`

  Disables drawing roadmaps.

Chapter 2

Examples

2.1 Using pathfinder

In this demo the user navigates the robot by clicking on the desired position in the simulator. A pathfinder is used to compute a path to the goal position avoiding collisions with obstacles. The program terminates when the user clicks the right mouse button or if there is no activity for 10 seconds.

```
import rbsim2d

# Create a new playground and connect to the local simulator.
playground = rbsim2d.Playground()
if (not playground.connect()):
    print "Error: ", playground.get_error()
    quit()
print "Connected."

# Check if all objects exist.
if (not playground.robot_exists("robot")):
    print "No robot found!"
    quit()

if (not all(playground.body_exists(["ball", "fence", "box"]))):
    print "At least one of the required objects was not found!"
    quit()

# Get objects
robot = playground.get_robot("robot")
ball = playground.get_body("ball")
fence = playground.get_body("fence")
box = playground.get_body("box")

# Get mouse and canvas to draw a path on.
mouse = playground.get_mouse()
```

```
canvas_path = playground.get_canvas("path")

# Prepare the pathfinder.
pathfinder = rbsim2d.Pathfinder()
pathfinder.set_spacing(1.1 * robot.radius())
pathfinder.add_obstacle([ball, fence, box])

# In order to follow a path, the robot needs to know its
# track width (the distance between the wheels). This can
# be measured by making a simple maneuver.
trackWidth = robot.find_track_width()
print "The distance between the wheels is", trackWidth

# Play along until the user clicks right mouse botton.
play = True
while (play):
    # Wait until the user clicks the mouse, but for no longer than 10 seconds.
    print "Waiting mouse click..."
    click = mouse.wait_click_event("released", 10000)

    if (click.button_type() == "undefined"):
        print "Timeout - finished."
        play = False

    if (click.button_type() == "right"):
    print "Right click - finished."
    play = False

    if (click.button_type() == "left"):
        position = click.position()
        print "User clicked on position", position
        path = pathfinder.find(robot, position)
        if (path.empty()):
            print "No path has been found!"
        else:
            print "Path has been found."
            canvas_path.clear() # Clear the previous path.
            path.draw(canvas_path) # Draw the new path.
            print "Following the path."
            robot.follow(path, trackWidth); # Start tracking.

print "Disconnecting."
playground.disconnect()
```