MotView

A Visualiser of Motion in the Plane[1]

(version 2.1)

Geert-Jan Giezeman[2]

August 16, 1993

[1]This work was partially supported by the Dutch Organisation for Scientific Research (N.W.O).
[2]Department of Computer Science, Utrecht University, P.O. Box 80.089, 3508 TB Utrecht, The Netherlands. email:geert@cs.ruu.nl

Contents

Chapter 1

Introduction

When one designs, implements and tests motion planning algorithms, it is very convenient to have visual feedback. In this way it is easy to discover if something goes wrong (collisions) or if some peculiar path is chosen. The software environment MotView is meant to make it easy to provide this kind of feedback.

MotView stands between the application programmer and the user. To the application programmer MotView offers a number of calls to define a scene and to move a robot around. The application programmer doesn't have to worry about the graphical aspects. The number of MotView routines is very small, so MotView is easily mastered.

The user has the benefit of a single user interface for viewing motions. MotView offers the possibility to view the motion in many different ways. The user can zoom in and out, follow the robot while it is moving, change colors of objects, play back motion, check for collisions, display the sweep volume of the robot etcetera. Also it is possible to generate a postscript file to print a scene on a printer. The user interface is user-friendly and quite simple to use. Clicking the mouse on buttons will do the trick in most cases.

In our model there is a world with obstacles in it and one or more moving objects called robots. A robot can have moving parts (links) that are attached by means of revolute or prismatic joints.

Apart from those objects it is often useful to have extra visual information available. This can help to visualise what happens in the algorithm. Suppose, for example, one wants to solve the problem of a translating polygon in the plane with fixed orientation. This can be solved by reducing the polygon to a single point (its center) and blowing up the obstacles (taking the Minkovski difference). One would probably like to see the Minkovski differences and see how the point moves among them. To this end the Minkovski difference can be added as an extra layer. The problem then can be viewed either in real space or as a point in Minkovski space. When using a potential field method one could show the equipotential lines. One could show a quadtree when using approximate cell decomposition. This kind of information helps to see what happens in the algorithm, where the complexity resides and what can be done about it.

The above examples add layers in a fixed place. Also information associated with the robot can be handy, for example a buffer zone around the robot.

In the viewing program the user can choose at any moment which layers of information are visible.

In a MotView program there is a sequence of three phases: first the layers are defined, then the moves are defined and finally the motion is viewed. Because of this separation, it is not very well possible to show how datastructures change while the algorithm is running.

MotView is developed for Silicon Graphics workstations. The next chapter describes what other libraries are needed to be able to work with MotView. There is also described how the package can be obtained.

The rest of this document is organised as follows: first there is a chapter describing the calls of the MotView library. This chapter is of interest to application programmers. Then there is a chapter describing the user interface. It is followed by a chapter about the motion viewing program. Finally there is a small chapter that says something about future developments.

Chapter 2

The MotView Library

The MotView library has only a few calls. Some deal with the definition and removal of layers, some serve to move the robot, others enable the user to interact with the program. The phases of definition of the layers and motion of the robot are separated: first the layers are defined, then the motion is described. It is not possible for objects to appear during the motion. The next sections describe the calls needed to perform those actions.

The MotView library is written in C++. It is built on several other libraries. Among these are:

- FORMS library for the user interface.
- PlaGeo library for geometrical primitives in the plane.
- RS2 for the definition of robots and configurations.
- GL for rendering.

MotView, FORMS, PlaGeo and RS2 are ftp-able from `archive.cs.ruu.nl` in subdirectories of `/pub/SGI`. They all come with documentation in LaTeXformat.

This manual supposes that the reader is familiar with PlaGeo and RS2, because some data types that are defined there are used in MotView. Knowledge of C++ is not needed for using the routines of MotView. The little that is needed to use PlaGeo and RS2 routines is described in a section of the PlaGeo documentation.

An application program that makes use of the library should include the header file motview.h. On page 6 there is an example program: `example.C`. This program is used as a running example in the rest of this document. To see what this program does it can be compiled:

```
CC -o example example.C -lmotview -lrs2 -lplageo -lforms -lfm_s -lgl_s -lm
```

2.1 Definition of the Layers

The visible objects are described by layers. Layers are constructed with the help of building blocks supplied by the RS2 library. Apart from a purely geometric shape, layers also have a color, a height and a name. Every layer is made of points, line segments and polygons. Other shapes (e.g. arcs) are not yet available and should be approximated. A layer normally contains a set of related objects. They share the same color and can all be made invisible by one click.

RS2, and therefore MotView also, restricts robots to chainlike structures: to every link at most one other link can be attached. This is enough to describe the kind of robots that are used in factories, but is not adequate for more complicated things. E.g. a human body with two arms and two legs can not be described in this way.

Synopsis

```
mv_layer_t mv_define(const pl_geolist &obstacles, int real_object,
                     int col, int height, const char *name)

mv_layer_t mv_define(const pl_robot &robot, int real_object,
                     int col, int height, const char *name)
```

Description

There are four different kinds of layers. First there is a distinction between obstacle layers (non moving) and robot (moving) layers. A layer of obstacles is specified by a `pl_geolist`, a robot layer is specified by a `pl_robot`, which types are both defined in RS2. So, the first `mv_define` routine above defines an obstacle layer, the second defines a robot layer. The routines return an identifier for the layer that was just defined. This identifier is needed to refer to this layer in other routines.

The second distinction that is made is between real layers and phantom layers. The second parameter should be set to true to define a real object. The real layers represent the objects that are actually in the world (the robot and the obstacles), the phantom layers represent help information. What is in the phantom layers is up to the application programmer. Often, layers can be added to show some information about the algorithm, for instance a cell decomposition, a potential field or a Minkovski difference. For MotView, the distinction is important for collision checking; this only happens between real objects.

The third parameter describes the color of the layer. This is an index in the color map. The fourth parameter is the height of the layer. Layers with a bigger height obscure layers with a lesser height if the two layers overlap. The last parameter is the name of the layer. This name is used in the user interface.

A robot phantom is defined in exactly the same way as a robot. A robot phantom, however, can not move on its own; it must always be attached to a robot. Therefore, after the robot phantom has beeen defined and before it is used in a motion, it should be bound to a particular robot, using the routine:

```
mv_bind_phantom(mv_layer_t phantomlayer, mv_layer_t roblayer);
```

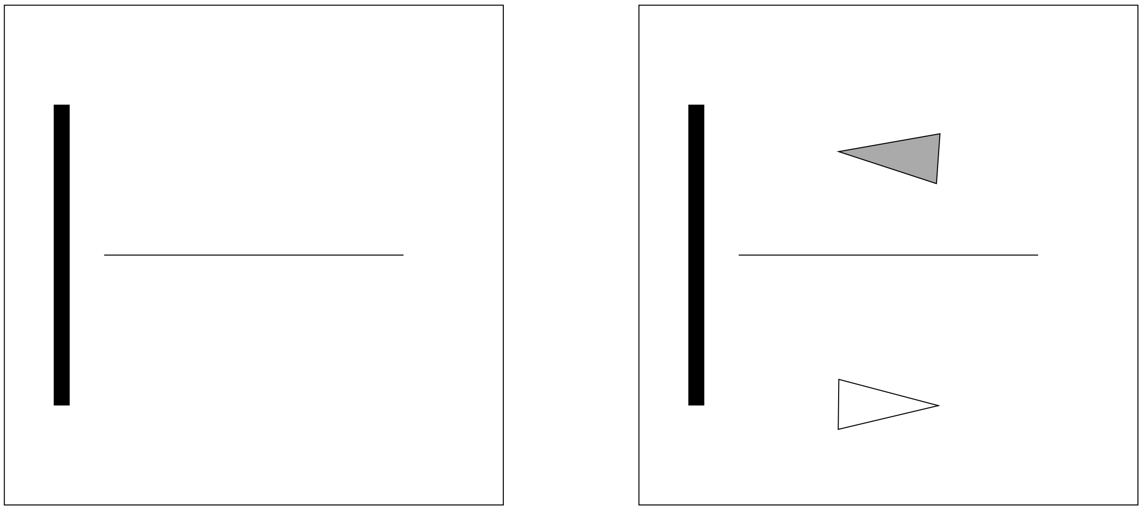
The links of the robot phantom are attached to the corresponding links of the robot. That's why a robot phantom should have the same structure as the robot to which it is bound, i.e., the same number and kind of joints as the robot.

Examples

In figure 2.1 and 2.3 there is the code for two example programs.

```
#include <plageo.h>
#include <rs2.h>
#include <motview.h>

main()
{
    mv_layer_t roblayer;
    pl_robot robot;
    pl_geolist gl;
    pl_vertexreray vrr;

    gl.el.append(pl_edge(pl_vertex(200, 500), pl_vertex(800, 500)));
    vrr.newsize(4);
    vrr.replace(0, pl_vertex(100,200));
    vrr.replace(1, pl_vertex(130,200));
    vrr.replace(2, pl_vertex(130, 800));
    vrr.replace(3, pl_vertex(100, 800));
    gl.pl.append(pl_pgn(vrr));
    mv_define(gl, 1, 0, 0,"obstacles");

    gl.el.newsize(0);
    vrr.newsize(3);
    vrr.replace(0, pl_vertex(0, 100));
    vrr.replace(1, pl_vertex(-50, -100));
    vrr.replace(2, pl_vertex(50, -100));
    gl.pl.replace(0, pl_pgn(vrr));
    robot.replace_link(0, gl);
    roblayer = mv_define(robot, 1, 7, 1, "robot");

    pl_configuration conf;
    conf.set_pose(4.7, pl_vec(500, 200));
    mv_abs_move(roblayer, conf);
    mv_start_motion();
    conf.set_pose(0, pl_vec(950, 563));
    mv_abs_move(roblayer, conf, 3);
    conf.set_pose(1.5, pl_vec(500, 700));
    mv_abs_move(roblayer, conf, 3);
    mv_end_motion();
    mv_view_motions(1);
}
```

Figure 2.1: code of the running example



the obstacles begin and end position

Figure 2.2: running example

The first program is used as a running example. It defines a very simple robot (a triangle, without any joints) and two obstacles. The main program is divided in four sections. In the first section some variables are declared. In the second section the obstacle layer is defined. A `pl_geolist` is created containing an edge and a polygon of four vertices. The robot is defined in the second section. The robot has no joints, so only a single link (a triangle). The last section describes a motion, which will be treated later. Figure 2.2 shows two scenes from this program.

In the second example program a robot and a robot phantom are defined. The example is not a typical robot, but with a few lines it shows the major concepts. In order to assure that the robot and the phantom have the same structure, the robot is copied to the phantom. Figure 2.4 shows the music stand in two positions programs. See how the score moves along with the stand.

In those two examples, the robot is explicitly defined in the program. Of course, it is also possible to define a robot or a set of obstacles in another program and save it into file. RS2 supplies routines to do this.

Removing Layers

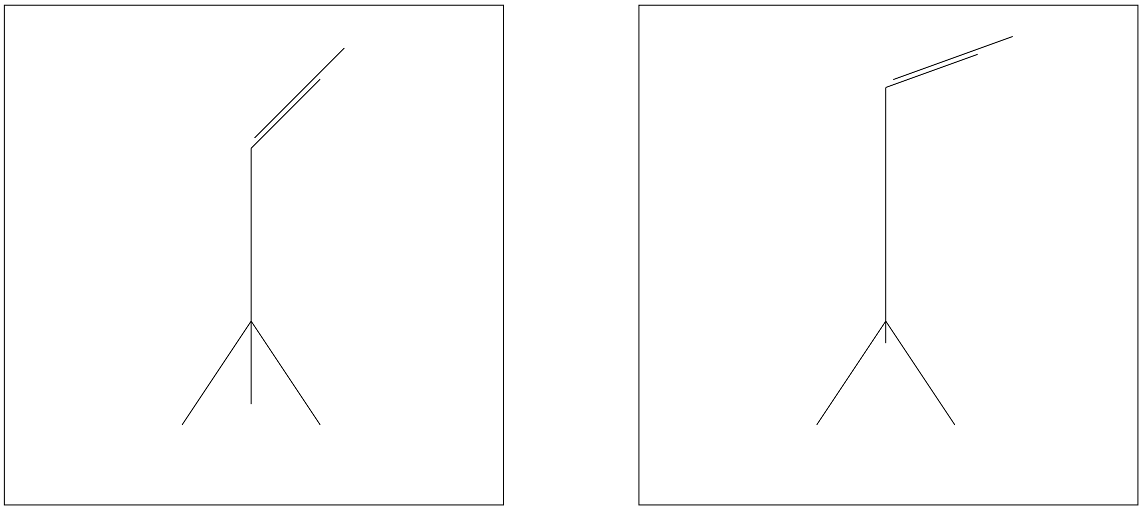
Layers that were defined previously can be removed afterwards. This is useful if one wants to define several motions in a single file. There are two calls which accomplish this:

```
mv_remove_all()
mv_remove(mv_layer_t id)
```

The first call removes all layers that were defined. The second call removes a single layer. The layer is identified by the identifier that was returned when the layer was defined.

```
#include <plageo.h>
#include <rs2.h>
#include <motview.h>

static s_coord jointdescription[] =
                {PRISMATIC, 0, 100, REVOLUTE, 0, 400, -5, 5};

main()
{
    pl_geolist gl;
    pl_robot stand(2, 8, jointdescription);
    pl_robot score;
    score = stand;
    mv_layer_t standlayer, scorelayer;
// define the links of the music stand.
    gl.el.append(pl_edge(pl_vertex(-100, 0), pl_vertex(0, 150)));
    gl.el.append(pl_edge(pl_vertex(100, 0), pl_vertex(0, 150)));
    stand.replace_link(0, gl);
    gl.el.newsize(0);
    gl.el.append(pl_edge(pl_vertex(0, 30), pl_vertex(0, 400)));
    stand.replace_link(1, gl);
    gl.el.newsize(0);
    gl.el.append(pl_edge(pl_vertex(0, 400), pl_vertex(100, 500)));
    stand.replace_link(2, gl);
    standlayer = mv_define(stand, 1, 0, 1,"music stand");
// define the links of the score and attach the score to the stand.
    gl.el.newsize(0);
    gl.el.append(pl_edge(pl_vertex(5, 415), pl_vertex(135, 545)));
    score.replace_link(2, gl);
    scorelayer = mv_define(score, 0, 0, 0, "score");
    mv_bind_phantom(scorelayer, standlayer);
// define the motion of the stand.
    pl_configuration conf(2);
    conf.set_x(500);
    conf.set_y(100);
    mv_abs_move(standlayer, conf);
    mv_start_motion();
    conf.set_joint(0, 1);
    conf.set_joint(1, -0.5);
    mv_abs_move(standlayer, conf, 3);
    mv_end_motion();
    mv_view_motions(1);
}
```

Figure 2.3: music stand program



music stand in zero position with joints moved

Figure 2.4: music stand pictures

2.2 Moving a Robot

After one has defined the layers, the time has come to let things move. The path that the robots follow is described in a motion. A motion is built out of several primitive moves. Primitive is perhaps not the right word here, as in a single move a robot can translate and rotate, and its joints can move as well. If, however, a robot should first move west and then move north, primitive moves should be combined to achieve this.

It is possible to define several motions in the same program. For example, if there are several solutions to move from A to B, every solution can be shown in a different motion. Between two motions layers can be added and removed.

The routines that are described in this section define a motion, they do not immediately show it. This can be compared to the shooting of a video tape.

The calls that deal with moving the robot are:

```
mv_start_motion()
mv_abs_move(mv_layer_t layer, const pl_configuration &goal, double time)
mv_end_motion()
```

The first call starts the motion. `mv_end_motion` is called to conclude the motion. Between these two calls the robots can move around.

The routine `mv_abs_move` takes three parameters. The first parameter must be the identifier of a robot layer. This indicates which robot should be moved. The second parameter is a configuration –this concept is explained in the RS2 documentation– which is the goal configuration. The robot moves from its current position to this goal position. The third parameter is the time (in seconds) that the motion should take, so it determines the speed of the move. This parameter may be negative or left out, in which case the viewer will select a speed itself.

The initial position of robots before the motion starts should be specified with a call of `mv_abs_move` before the `mv_start_motion` call. In this case, the time parameter is of no importance.

To determine the intermediate position of a robot during a move, all parameters are linearly interpolated between begin and end position. This means that the velocity of the joints and of the main link is constant during a single move. (It does not mean that the endpoint of the robot follows a straight line.) Of course, when the angle of begin and end configuration is not the same, the robot can turn clockwise or counterclockwise; the smallest angle is chosen.

As was mentioned, the timing of moves can be left over to MotView. If one is not interested in the exact timing of moves, this will often do. One can influence how MotView chooses the times by means of the function

```
mv_set_autospeed(double speed);
```

By default MotView chooses the speed so that a move of ten units in the x or the y direction takes one second. This speed can be changed with this call. If the speed is set to 100 all moves will go 10 times as fast. This speed only affects moves that do not specify their own time.

In the last section of program 2.1 a motion is defined. The robot is placed initially in position (500, 200) with an orientation of 4.7 radians. It makes two consecutive moves, of three seconds each, to end in position (500, 700) with orientation 1.5 radians (pointing to the left). The second move will make the robot collide with one of the obstacles.

Synchronisation of Moves

When there is more than one robot in the scene, the synchronisation of the moves may be important. All moves of a single robot are executed sequentially in the order in which they are specified. All moves of different robots are parallel. This means that only the order of moves of a single robot is important. The moves of different robots can be mixed in any way. They all have there own, separate timeline.

To synchronise one can keep track of the time oneself. For each move specify how long it should take. To let a robot wait one can let it move to its current position during the desired waiting time.

Another possibility is the use of the following routines.:

```
mv_sync2(mv_layer_t id1, mv_layer_t id2)
mv_sync_all()
```

These routines synchronise the moves of two or all robots. When robots are synchronised, all robots wait at the synchronisation point until all the robots involved have finished their moves before they continue with the rest of their moves.

In figure 2.5 is shown a program fragment where three robots move and are synchronised, together with the timeline of each robot. Take a good look at this figure to understand how this works. Note that the last sync-call does not influence the third robot.

For the case that one wants to synchronise more than two (but not all) robots, one should combine calls in the following way:

```
    mv_sync2(rob1, rob2);
    mv_sync2(rob1, rob3);
    mv_sync2(rob1, rob2);    /* don't forget this */
```

```
mv_sync_all();
mv_abs_move(rob1,conf,3);
mv_abs_move(rob1,conf,2);
mv_abs_move(rob2,conf,2);
mv_sync2(rob1,rob2);
mv_abs_move(rob2,conf,2);
mv_abs_move(rob3,conf,6);
```

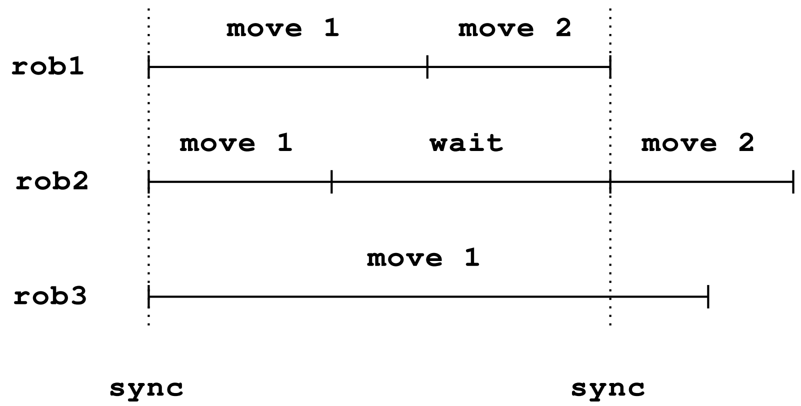


Figure 2.5: Synchronisation of three robots

2.3 Responding to User Input

The user may interact with the application program in several ways. First of all the user may press buttons and so on of the motion viewer. The viewer should then take the appropriate action. Then there are calls the application program can make that explicitly ask for some user response. Finally the application program may do its own stuff.

This section describes the calls to deal with the first two kinds of user interaction. The last type is of course the responsibility of the application program. It describes what the application program should do, not how the user can respond. That is treated in chapter 3.

Enabling User Input

On startup MotView opens a window (actually, it is a Forms Library form). There is a region where the scene is shown that is surrounded by a number of buttons, the user interface. Because the viewer should respond to the input of the user and the view changes as the robot is moving the application program should hand over control to the library after having defined some motions. To this end there is the call:

```
int mv_view_motions(int wait)
```

This call updates the scene and gives the user the opportunity to interact with the viewer. It is normally called after one or more motions have been defined. It takes care that the view is continuously updated, reflecting the progression of the motion and the choices of the user (zooming in and out, showing or hiding certain layers etcetera). If the user is satisfied he can press the return button to indicate this. Then `mv_view_motions` will return (with value 1).

So far the behaviour of `mv_view_motions` was described when called with non zero parameter. If the wait parameter is zero (false), `mv_view_motions` will return immediately after updating the view and handling the pending user requests. It does not wait until the user presses the return button. This does not give the user much time to interact with the viewer, unless called repeatedly. The returned value indicates if the user pressed the return button, so the two following lines are equivalent:

```
mv_view_motions(1);
while (!mv_view_motions(0)) ; /* Empty Loop */
```

Once a motion has been viewed, the two MotView windows remain visible. They can be hidden by calling the procedure:

```
mv_hide();
```

The windows will reappear automatically when another motion is shown, at the next call of `mv_view_motions`.

Place a Robot

There are two routines that give the user the opportunity to place the robot in a position. This can be useful, for example, for asking a start or goal position.

```
mv_place_robot(mv_layer_t, pl_configuration &conf, const char *msg)
mv_place_robot(mv_layer_t, pl_configuration &conf,
               const pl_configuration &constraints, const char *msg)
```

The first parameter must be the identifier of the robot that must be placed. Before calling one of those functions **mv_abs_move** can be called to specify where the robot is placed initially. MotView shows the robot in this position and gives the user the opportunity to move the robot to the desired place. How the user should do this is described in section 3.4. When the user is satisfied with the place he can indicate so and the function returns. The `conf` parameter contains the configuration in which the robot was left. The the joint values will always lie between the minimum and maximum (i.e. between 0 and 1 for a prismatic joint and between the minimal and maximal angle for a revolute joint.

In the `msg` field the user can give a string that is shown when the robot is placed. It could say something like "Give start position.". If no message is required, an empty string should be specified. If msg contains a null-pointer a default message is displayed. New lines (\n) are allowed in strings.

The application can restrict the user in moving the robot by using the second call. The `constraints` parameter is a configuration that indicates whether the user can move the robot in that direction. In fact, this is not a real configuration. All values, of x and y position, angle and joints, should be either **MV_FIXED** or **MV_FREE**, two values defined in the motview header file.

In order to make the creation of those special configurations a bit simpler, there are a number of routines. The following functions return a configuration where every degree of freedom is free, everything is fixed, only the joints are fixed or only the joints are free respectively. The `jointcount` parameter expects the number of joints of the robot.

```
pl_configuration mv_all_free(int jointcount)
pl_configuration mv_all_fixed(int jointcount)
pl_configuration mv_joints_fixed(int jointcount
pl_configuration mv_joints_free(int jointcount)
```

For example, the following example fragment aks the user to place a robot with three joints in a start and goal position. In the start position, only the orientation is fixed. The endposition has the same position as the start position, only the joints may be in a different position.

```
pl_configuration constraints(3), startpos(3), endpos(3);
constraints = mv_all_free(3);
constraints.set_angle(MV_FIXED);
mv_abs_move(rob, startpos);
mv_place_robot(rob, startpos, constraints, "Give the start position.")
mv_abs_move(rob, startpos);
mv_place_robot(rob, endpos, mv_joints_free(3), "Give the goal.")
```

2.4 Error Checking

Not every combination of library calls makes sense. MotView does some checking whether calls are appropriate at a certain moment. If this is not the case, the user is informed and the program is aborted. The messages are intended to be self-explanatory.
Here are some examples of things that are checked:

- Only robots can be moved.
- Robots should not be synchronized outside a motion.
- Robot phantoms must have the same number of joints as the robot.
- Configurations should match robots (equal number of joints).
- A robot phantom is not bound to any robot when a motion is started.

When MotView finds an error, the program will halt. First the user is asked if a core dump should be created. This core dump can be inspected with the help of a debugger to inspect the code at the point where things went wrong.

Chapter 3

The User Interface

MotView offers a great variety of ways to view motions. The user can zoom in and out, show the motion in slow motion, display or hide layers and much more. One always views one motion at a time.

When MotView is started there appears a viewing window where the scene is displayed surrounded by a number of buttons and sliders, the main part of the user interface. Other parts of the user interface can be popped up by pressing some buttons. The user interface windows are called forms. Initially a view is chosen so that a square part of the scene between coordinates (0,0) and (1000,1000) can be seen.

Perhaps the best way to learn how to use the viewer is by running an example program and try out all options. There is an online help facility, but most functions speak for themselves. The buttons and sliders are activated by putting the mouse over them and pressing a mousebutton.

Some actions are performed by pressing a mouse button and then drag the mouse. This is the case for sliders and for mouse actions in the viewing window. Those actions can be controlled more precisely by keeping the left shift-key depressed while dragging.

3.1 The Main Form

In the main form, the part below or beside the viewing window, are the functions that are used most frequently. In figure 3.1 there is a picture of this form. The form is divided in three sections: display control (top left), motion control (bottom) and global control (top right). Actually, the arrangement can be different than is shown here, but all the buttons and sliders should be there.

Display Control

The top left part of the main form is the Display Control section. Here the user can select what layers he wants to see. Apart from the layers defined with the MotView library (the user defined layers), MotView can show a number of features associated with the motion of the robot. Those features can be regarded as extra layers of information defined by MotView. They are called the standard layers in this document. Every layer can be made visible or invisible by selecting or deselecting it.

The names of the user defined layers are displayed in two browsers on the top. The

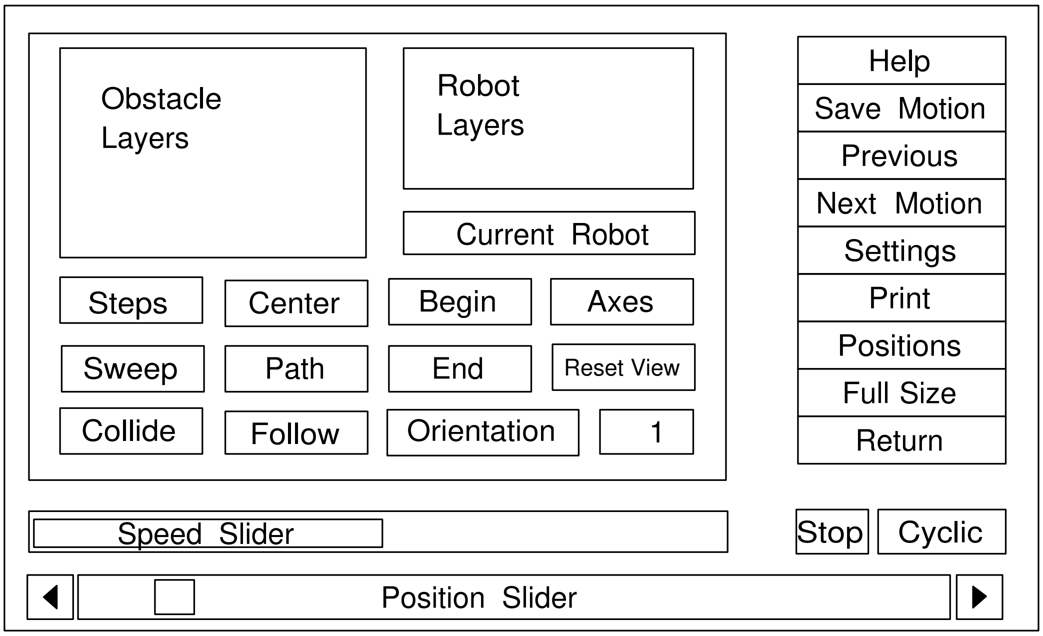


Figure 3.1: the main form

left one contains the obstacle layers, the right one the robot layers. Selecting is done by clicking on the name of the layer.

Below the right browser is a button for selecting the current robot. The current robot is a special robot of which extra information can be displayed. To select another robot as current robot one can press on this field with one of the mouse buttons. The right button causes a drop down menu to appear from which the user can make a choice. The left and right buttons cycle through the possible choices.

The buttons for selecting the standard layers are called Steps, Sweep, Center, Path, Begin, End, and Axes. Other buttons in this section are Reset View, Collide, Follow and Orientation.

Steps displays the current robot in a number of positions on the path. How many steps are shown can be controlled in the settings form. This layer is especially nice if one wants to print a scene. The motion is condensed in a single picture.

Sweep displays the area that the current robot traverses during its motion. If the robot were a piece of chalk this is the trace it would leave. This allows one to see quickly if the robot enters some forbidden region.

Center shows the center of the current robot as a little cross.

Path shows the path of the center of the current robot.

Begin shows the current robot in its begin position.

End shows the current robot in its end position.

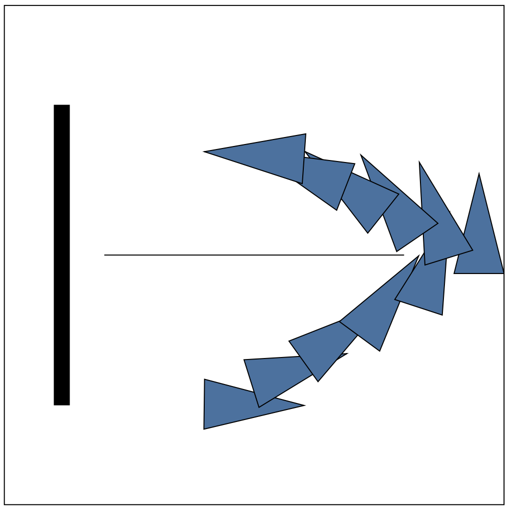
Axes shows x and y axis of the robot frame.

Collide controls whether collision checking is on. When a collision is detected the robots color changes.

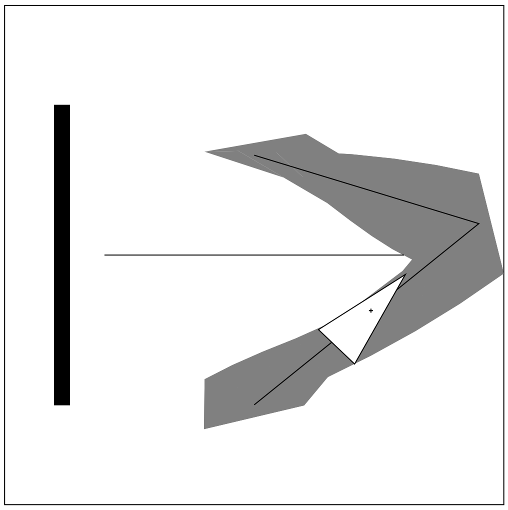
Motion Number The field on the bottom right of this section shows the number of the motion that is viewed.

Apart from controlling what is visible the user can also focus his attention to a particular point. If FOLLOW is selected the camera moves along with the current robot. If ORIENTATION is selected as well then the camera rotates along with the robot. The user can also zoom in and out and shift his attention. This is not controlled from the form but in the viewing window. In order to zoom place the cursor in the viewing window and keep the right mousebutton pressed; to take a closer look move the mouse to the lower part of the viewing window, to get a wider overview, the mouse should be moved to the top. Shifting of the viewpoint is controlled with the left mousebutton. Keep it pressed and move the cursor in the direction you want to see. To undo the zooming and shifting one can press the RESET VIEW button. This button restores the default view. The default view is the view one sees initially, but this can be changed in the settings form.

To give some idea how this looks, here are some pictures that correspond to the motion of program 2.1. The first picture gives the view when steps is pushed. The robot layer is not selected in this picture. The second shows the robot itself, the sweep volume, the center of the robot and the path along which it travels.

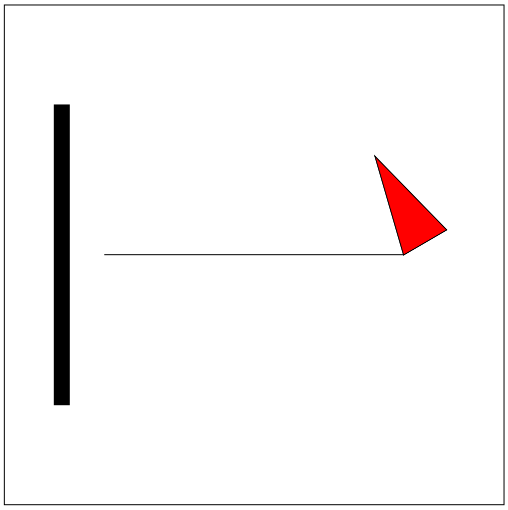


steps

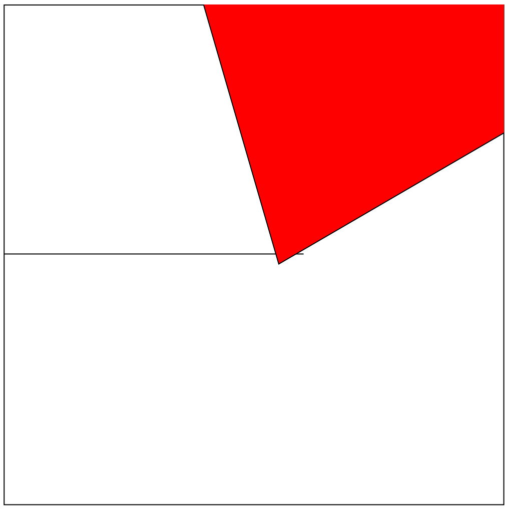


sweep volume, path and center

In the previous picture one can see that the sweep volume intersects with an obstacle. In the following picture we have placed the robot in a position where this collision is apparent. Notice that the color of the robot has changed to the collision color. To the right of this picture is a close-up of the same situation.

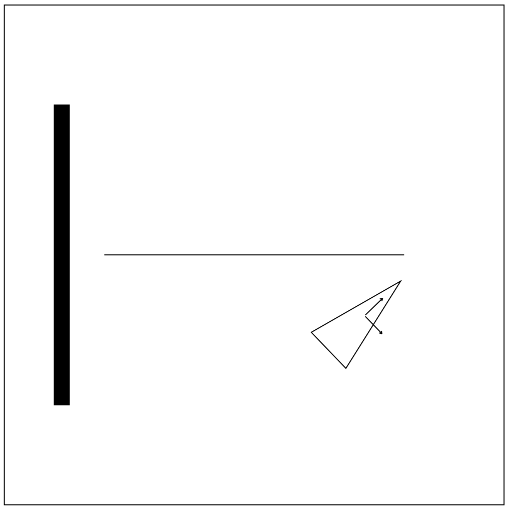


a collision

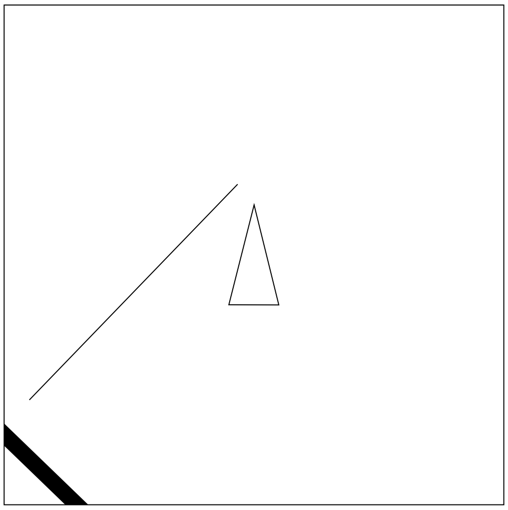


close-up of collision

The next two pictures both show the robot in the same position. In the left picture we see the axes of the robot frame. In the right one the follow orientation button is pushed.



axes of robot frame



follow orientation

Motion Control

Below the display section is the motion control section. In this section the user can control the position and speed of the robot. At the bottom are buttons that work like a tape recorder: forward, backward and stop. The speed slider can be used to control the speed of the motion. Move it to the right to let the robot move faster. The position slider reflects the position of the robot along its path. It is updated as the robot moves (and updating is turned on). Grab this slider to move the robot manually. Those sliders can be controlled more precisely by pressing the shift button while dragging.

Finally there is a button called CYCLIC. Normally the motion will stop when the robot reaches its end position. If this button is on the motion starts again after some seconds. This is nice when giving a demonstration.

Global Control

At the top right is the global control section. The global control gives access to functions which are not so often used.

HELP shows a form with online help messages. In this form the user can select a topic on which he wants some information. The help form can be hidden by pressing the hide button on it.

SAVE MOTION saves the current motions in a file. First one gets the opportunity to select the motions that must be saved. By default all motions are selected. It is also possible to save only the last motion.

Then a file selector is presented to choose a filename. One can press the mouse on a filename to overwrite an existing file, or type in a new name. One can walk through the directory structure by clicking on a directory name (shown with a D in front of it).

By convention motion files have a '.mot' extension, so you should preferably use such a name. The file selector normally shows only files with this extension. This is because the regular expression in the pattern field of the selector is $*$.mot. This pattern acts as a filter: only filenames that pass it are shown. One can edit this pattern by pushing the mouse on it. Change the pattern to $*$ to see all files in a directory.

If a motion file contains several motions another motion is selected by the buttons PREVIOUS and NEXT. The current motion number is displayed in the display control section. When another motion is selected the view stays the same, though the default view is altered. This is handy if the two scenes are related and one wants to focus one's attention to a particular point. If this is not the case, one can always press on Reset View afterwards to get the default view.

SETTINGS shows a form in which the user can adapt various settings. PRINT prints the current scene. Those two options are described in more detail in the next two sections.

POSITIONS shows a form with numerical information about the current position of the current robot. From top to bottom there are the time, the clearance between the current robot and the nearest obstacle, the x, y and theta position of the current robot and the positions of its joints.

FULL SIZE makes the viewing window cover the whole screen. In this case the user interface will be placed at the right side instead of the bottom and it will be arranged differently.

With RETURN one can return control to the application program.

3.2 The Settings Form

The settings form appears when the corresponding button in the global control section is pressed. In this form one can adapt the color of the different layers and some other parameters.

There are two browsers that let the user choose colors. In the left one are the user defined layers. In the right one are the standard layers (including background and collision).

Press on a layer in a browser to select another color for this layer. A palette of colors will then be presented from which one can choose.

The STEPS slider controls how many steps are shown when steps is selected in the main form.

UPDATE controls when the information in the information section of the main form and the position slider is updated. It can have four values:

never never update the info.

special update in special cases: when motion stops or is adapted manually.

periodic same as special, but at least once every ten frames.

always update after drawing of every frame.

If a lot of functions are activated and the scene is complex the motion will be less smooth. To get smoother motion one can turn off collision checking, information updating, the display of the sweep volume or steps.

SET DEFAULT VIEW makes the current view the default view. If the viewpoint is changed afterwards one can restore the default view by pressing the reset view button. When the camera follows the robot this button has no effect.

Then there is a switch for choosing how collisions are checked. There are four modes, each increasing what is checked for. In the simplest mode (robot–obstacle) only collisions between robots and obstacles are detected. The second mode (robot–robot) also checks for intersections between different robots. The third (link–link) and fourth (neighbour links) mode add checking for collisions of links of one robot. In the third mode no checking is done for neighbouring links. This is useful because normally it will be logically impossible for these to collide (as long as the joint position stays within its bounds), even though they might actually overlap to give a nicer picture.

3.3 The Print Form

The print form appears after selecting PRINT in the global control section. This allows one to save the current scene in a PostScript file. One can choose to have a bounding box around the picture.

The user is asked to supply the name of the file. The saved file contains standard postscript and can of course be edited afterwards. One can for instance clip a rectangular instead of a square portion of the scene. One can also insert the file in another document.

The pictures in this document were made in this way. Viewing the example motion file some shots were taken and saved on file. These postscript files were then included in this LaTeXdocument with the help of the epsf style for including Encapsulated Postscript in LaTeXdocuments.

In order to get a nice picyure on paper, it is probably necessary to select different colors for the layers, especially when using a black and white printer. This can be done in the settings form. It is not necessary to choose only grays when using a monochrome printer: colors will be turned into gray values by the Postscript interpreter.

3.4 Placing A Robot

The application program can ask the user to place the robot in a position. When this happens, the main form disappears and another form appears that allows the user to place a robot manually. Like the main form, this form shows the scene, but there is a different user interface. The user can move and rotate the whole robot and move the separate joints. The OK button should be used to indicate that the robot is in the desired position.

There are several ways in which the user can manipulate the position of the robot. First of all, another form can be made visible (by pressing on the button labeled Positions) that shows the x-y-position, orientation and joint positions. Figure 3.2 shows this form.

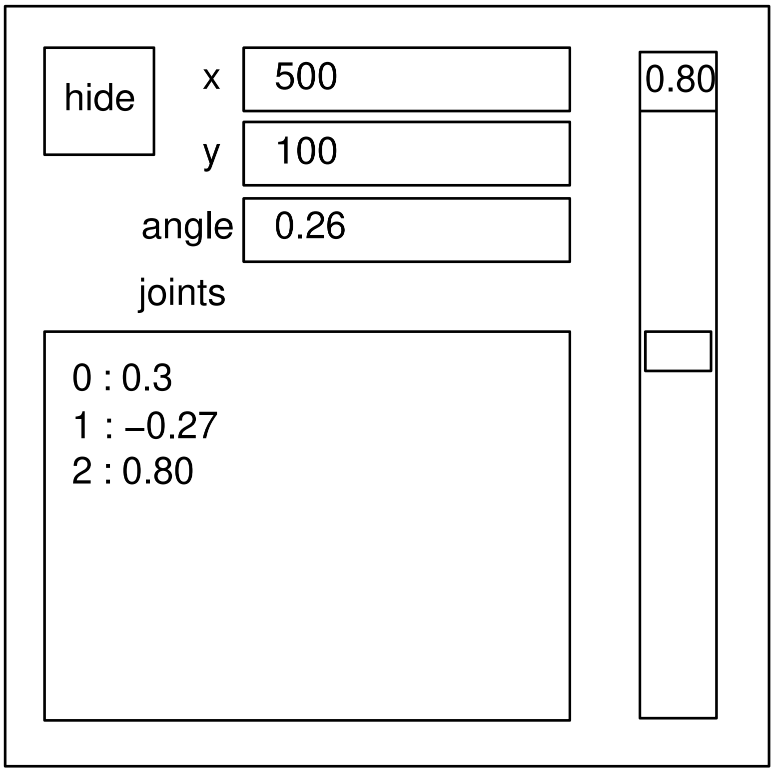


Figure 3.2: the positions form during placing

The x, y and angle field can be edited directly (type in a number and then type return). To change a joint position, first click on the corresponding line of the browser at the bottom. The slider to the right will take the value of the chosen joint. It can be altered by moving the slider.

It is not necessary to use the positions form. In the first form the scene is shown. The robot can be grabbed, by pressing the left mousebutton while the mouse is over the robot, and be moved in the x- or y-direction. If the mouse is not over (or close to) the robot, the whole scene will move, effectively changing the viewing position.

To move the orientation of the robot, press the middle mousebutton and move the mouse the left or right of the form (for clockwise or counterclockwise rotation). The orientation can also be altered in steps, using the left and right arrow key. The shift key can be used in combination with either of those methods to make the rotation go slower.

In order to move a joint, first a joint should be selected with one of the keys 0 to 9. Then, the middle mousebutton and the arrow keys will control this joint instead of the orientation. By pressing the '–' button (next to 0), they will control the orientation again.

Like in the main form, the right mouse controls the zooming in and out. Keeping the right mousebutton pressed, moving the mouse down will zoom in, moving up will zoom out.

The application program can prohibit the robot to move in certain directions. In this case, the actions that were described above will not influence the position of the robot.

Both ways of placing the robot, with the mouse or with the positions form, can be used at the same time. The changes made with the mouse are reflected in the positions form.

Chapter 4

The MotWrite Library

Besides MotView there is another library: MotWrite. MotWrite is offline, that is, when a program is linked with MotWrite instead of MotView, the motion is not shown. Instead, a motion file is created. This file can be viewed later with the viewing program MotView[1]. It is the same type of file as is generated when one saves a file that is viewed by pressing the 'Save' button in the user interface.

To compile a program with the MotWrite library, do:

```
CC -o example example.C -lmotwrite -lrs2 -lplageo -lm
```

One can always choose to link with either of the libraries. The source file need not be rewritten and they both use the same header file. Because MotWrite is offline, there is no possibility of direct interaction with the user. Functions that deal with user interaction are ignored by MotWrite. In this case **mv_view_motions** will always return 1 and the parameters of a place-call will never be altered. Also checking for errors is less strict in the MotWrite library.

There is one call in MotWrite that is not used in MotView:

- **mv_set_file**(char *filename)

This can be called to write the output to a file instead of to standard output. **mv_set_file** implicitly does an **mv_remove_all**, so layers that are used in different files should be defined anew. Remember that motion file names normally end in '.mot'.

Because MotWrite needs a separate creation and viewing step and has less functionality, one will normally link with the MotView library. On the other hand the MotWrite library can be used on computers without special graphic capabilities, not just on SGI machines. The size of executables is a lot smaller when programs are linked with MotWrite.

[1]note that the MotView library and the MotView program are two separate things.

Chapter 5

The Viewing Program

Apart from the two libraries, there is also a viewing program with the name MotView. This program can be used to view motion files that were created with the MotWrite library or saved by a user of the MotView library.

To view a specific file type '`motview` filename'. One can name more than one filename on the command line. If standard input should be read, a dash should be typed instead of a filename.

The user interface that appears is almost the same as that of the MotView library. The only difference has to do with loading and saving motion files. The 'Save' button in the main form is replaced by a 'Load' button. If the user presses this button a file browser is presented. The file that is selected is read and the motions that are contained therein can be viewed. Those motions replace the old motions. If one wants to load several motion files in the viewer this can only be done by specifying more than one file on the command line.

Chapter 6

Future Developments

As long as software is used, there will be a need for change, so hopefully MotView will be extended in the future. Although it is easy to predict that there will be changes, it is harder to foresee what those changes will be. For example, this version of MotView is smaller than the previous one: some functionality has been left out to concentrate on making this a good package for viewing motions, not a mediocre one for animating algorithms.

Some adaptions that are considered are:

- Allow more shapes besides points, line segments and polygons. Especially circles and arcs might be handy in some cases.
- Take away the restriction of chainlike robots. Also allow for treelike structures.
- Allow more kinds of joints than revolute and prismatic.

Finally it would be nice to have a tool like MotView for visualising motion in three dimensions. Where the former changes are mere extensions to MotView, going from the plane to three dimensions will probably be done in a separate program.

Appendix A

Overview of Routines

The following sums up all types, constants and routines defined in motview.h.

```
const MV_NAMELENGTH = 14;

enum mv_constraints {MV_FIXED=0, MV_FREE=1};


typedef mv__layers_index mv_layer_t;

void mv_set_autospeed(double);
mv_layer_t mv_define(const pl_geolist &obstacle, int  real_object,
                 int  col, int  height, const char *name);
mv_layer_t mv_define(const pl_robot &robot, int real_object,
                 int col, int height, const char *name);
void mv_bind_phantom(mv_layer_t phantom_id, mv_layer_t robot_id);
void mv_remove(mv_layer_t layerid);
void mv_remove_all();

void mv_start_motion();
void mv_abs_move(mv_layer_t, const pl_configuration &, double time=-1);
void mv_sync2(mv_layer_t id1, mv_layer_t id2);
void mv_sync_all();
void mv_end_motion();

int mv_view_motions(int wait);
void mv_hide();

void mv_place_robot(mv_layer_t, pl_configuration &, const char *msg = 0);
void mv_place_robot(mv_layer_t, pl_configuration &,
pl_configuration &constraints, const char *msg = 0);
pl_configuration mv_all_free(int jointcount);
pl_configuration mv_all_fixed(int jointcount);
pl_configuration mv_joints_fixed(int jointcount);
```

```
pl_configuration mv_joints_free(int jointcount);

void mv_set_file(const char *);
```