The Simulation Library: A Basis for Animation Programs

Version 2.0

Susan Hert Dan Reznik

Robotics Laboratory
University of Wisconsin–Madison

Contents

1 Introduction

The Simulation Library provides a foundation for the design of motion simulation programs. The library is written using ANSI C function prototypes. It was developed in the UW Robotics Laboratory to take advantage of the commonality present in many independent projects, all involving the visualization of objects, specifically robots, in motion. As such, the library makes few assumptions about what is meant by a "robot in motion" but provides for efficient animation of these entities.

Provided by the library are some basic C structures that can be used to construct robots and other objects in an environment. These C structures represent primitive geometric entities (points, lines, polyline, polygons, arcs, and circles) that can be combined in any fashion to produce arbitrarily complex environments and robots. Functions are provided for modeling and manipulation of two-dimensional, articulated hierarchical structures, that may easily represent robots. Also, the library provides functions to manipulate a database of environment obstacles and a means by which a user of an application written with the library can change this database interactively to create arbitrarily complex environments.

Also provided are routines for performing simple geometric calculations with the structures, and routines for computing distances, intersections, tangents, etc. between these structures. Using these geometric routines as a basis, any motion planning algorithm can be simulated in arbitrarily complex environments. In fact, though it was designed with mobile robots in mind, the library can be used as a basis for any program that requires geometric structures and calculations.

In addition, there are routines that allow for the development of a user interface to display the motion simulation on either a Silicon Graphics machine (SGI) or a Sun (or, generally, any Xwindows-based) workstation. When used on an SGI, all the features of the standard graphics library (except multiple-windowing capabilities) remain available. The Simulation Library routines not only allow the programmer to design the user interface to meet the needs of the particular simulation, but also provide a basic structure upon which to build. Animation is made possible through built-in features of the user interface (although the programmer must write the animating functions for the moving objects in any program).

Currently the library supports only 2D motion and geometry. When run on an Silicon Graphics machine, it also provides a window for drawing 3D objects, although the functions for geometric computations and drawing in three dimensions are not currently supported in the library. Suggestions for changes and improvements may be directed to `hert@cs.wisc.edu`.

This document introduces to the Simulation Library (the SL) through examples, explanation, and exposition. Sections 1.1 and 1.2 provide information about where the library is located and how to compile programs using the library. Section 1.3 gives a brief overview of the primitives and families of the library. Sections 2 through 6 provide examples that illustrate the use of many of the function modules in the library. The examples in Section 2 illustrate the basics of building and using the user-interface. The remaining examples are for the more advanced features of the library. Section 7 gives a complete description of the structures, constants, and macros provided in the library. The detailed reference manual describing all the SL functions is given in Section 8.

1.1 Where Is the Library

In the UW Robotics Laboratory, the library `libSL.a` and header file `SL.h` are located in the directory `/usr/local/SL/2`. The source code and makefiles for all the example programs in this manual as well as some other programs is given in `/usr/local/SL/2/Examples`.

The library is also available via anonymous ftp from `robios8.me.wisc.edu`. It is in the directory `/pub/SL`. Both IRIX5 and SunOS4 versions are available (as `irisSL.tar.gz` and `sunSL.tar.gz`, respectively). These distribution packages each contains a README file, the header file `SL.h`, the library `libSL.a`, and several example programs and makefiles.

1.2 Compiling with the Simulation Library

In order to use the routines provided by the Simulation Library, the header file `SL.h` must be included using a directive such as

```
    #include </usr/local/SL/2/SL.h>
```

in your program, and the library `/usr/local/SL/2/libSL.a` must be linked with the rest of the program along with some of the Xwindows and GL libraries. For the SGI, these libraries are: libSL, libc, libgl_s, libXirisw, libXaw, libXmu, libXt, libXext, libX11, and libm; for compilation on a Sun, the libraries are libSL, libXaw, libXmu, libXt, libXext, libX11, and libm. Note that since ANSI C prototypes were used in the library, an ANSI C compiler must be used to create the object file(s) for your program.

For example, to compile the SL-application `robot.c` on a Sun workstation, the following command should be used:

```
%gcc -ansi -g -o r robot.c /usr/local/SL/2/libSL.a -lXaw -lXmu -lXt -lXext -lX11 -lm
```

For compilation on an SGI, the command should be:

```
%cc -DSGI -ansi -g -o r robot.c /usr/local/SL/2/libSL.a -lc -lgl_s -lXirisw -lXaw -lXmu
-lXt -lXext -lX11 -lm
```

Sample makefiles are given in `/usr/local/SL/Examples` and are available via anonymous ftp (Section 1.1).

1.3 Overview of the Library

1.3.1 Some Notes on Syntax and Types

- All names native to the simulation library begin with the prefix "SL_" (for Simulation Library). This includes the names of structures, constants, macros, and functions.
- All floating point values are represented as doubles.
- Parameters to functions are either simple types (*e.g.*, integers or doubles) or pointers to structures.

1.3.2 The Library Primitives

The geometric primitives supported in this library are: points, vectors, line segments, arcs, circles, polygons and polylines. They are represented by the following C structures.

```
typedef struct {
   double         x;         /* the x coordinate of the point or vector */
   double         y;         /* the y coordinate of the point or vector */
} SL_Point, SL_Vector;

typedef struct {
   SL_Point       start;     /* the starting point of the segment */
   SL_Point       end;       /* the ending point of the segment */
} SL_Segment;

typedef struct {
   SL_Point       center;    /* the center of the arc */
   double         radius;    /* the radius of the arc */
   double         start_ang; /* CCW radian measurement for starting arc angle */
   double         end_ang;   /* CCW radian measurement for ending arc angle */
} SL_Arc;
```

```
typedef struct {
   SL_Point       center;    /* the center of the circle */
   double         radius;    /* the radius of the circle */
} SL_Circle;

typedef struct {
   int            size;      /* the number of vertices in the polygon or polyline */
   SL_Point       vertex[SL_MAX_VTX];
                             /* counterclockwise list of vertices with
                                the first vertex not repeated */
} SL_Polygon, SL_Polyline;
```

The library also supports a more general structure, the `SL_Shape`, which can represent any of the geometric primitives (Section 7). Four abstract data types (ADTs), `SL_Bug`, `SL_Transf`, `SL_Frame`, and `SL_List` are also available (Sections 8.16, 8.17, 8.19, and 8.20, respectively).

1.3.3 Preview of the Library Modules

The functions in the library are arranged into several distinct modules. Briefly, these modules are:

- **Utilities** - miscellaneous functions not specific to any library primitives
- **Primitive Manipulation Functions** - for performing basic calculations with the geometric primitives of the library
- **Containment** - for determining if one primitive is inside a given polygon or circle
- **Translation** - for moving primitives to specific locations
- **Intersection** - for computing or checking for intersections between pairs of library primitives
- **Distance** - for computing minimum distances between pairs of library primitives
- **Tangent** - for computing tangents between pairs of library primitives
- **User Interface** - for setting up the simulation window
- **Graphics** - for drawing the primitive structures
- **Mouse Editing** - for using the mouse to input or change primitive structures
- **Obstacles** - for adding and modifying obstacles in the simulation environment
- **Bug Algorithms** - for implementing the sensor-based, point automata planar motion planning algorithms Bug2 and Visbug21 [1, 2]
- **Homogeneous Matrices** - for manipulating homogeneous transformation matrices
- **Modeling** - for modeling transformations
- **Frame** - for creating hierarchical geometric structures (*e.g.* arm manipulators)
- **Lists** - for manipulating dynamic collections of data as stacks, queues, or random access structures

These modules will be explained in the following sections through the use of example programs. Descriptions of the functions provided in the various modules are given in Section 8.

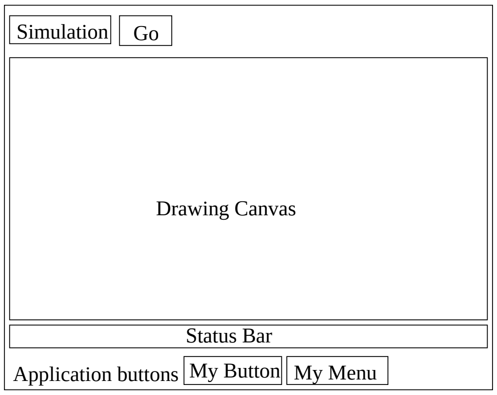


Figure 1: A sample user-interface window produced by an SL-application.

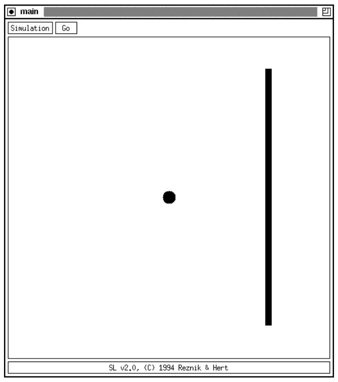


Figure 2: The window produced by program `example1.1.c`

2 The Basics

Any program that uses the Simulation Library is called an *SL-application*. Every SL-application produces a user-interface window with the following features: a simulation menu button, a go button, a drawing canvas, a status bar, and a row of application-specific buttons and labels, which may be empty (Figure 1).

In this section we explain, through the use of several simple SL-applications, the functions in the library that are used for creating and interacting with the user interface and for performing simple geometric calculations (*e.g.*, intersection and distance calculations) with the library primitives.

In Section 2.1 the *user interface loop* (UI-loop), which controls the animation of any SL-application, is explained along with the basic graphics functions, which are used to draw the library primitives. The example in Section 2.2 illustrates how an SL-application can use input from the mouse and keyboard. Section 2.3 explains how application-specific labels, buttons, menus, and dialog boxes may be added to the user interface. We illustrate in Section 2.4 functions that can be used for more sophisticated mouse interaction, as well as the function for drawing text on the canvas.

2.1 The User Interface Loop and Graphics Routines

2.1.1 A Programming Example – `example1.1.c`

What the Program Does

The program shown below is a minimal SL-application that works as follows. When the application's window appears on the screen, a dark blue, circular robot is positioned in the center of the drawing canvas and there is a red, rectangular barrier on the right side of the canvas (Figure 2). (Without a color monitor, both the robot and barrier will be black, of course.) When the Go button at the top of the window is pressed, the robot will move in a horizontal line to the right toward the barrier. When the Go button is pressed again or the robot hits the barrier, the motion stops. If the Simulation button is pressed, a menu will pop up with three options: Redraw, Reset, and Quit. The Redraw option simply refreshes the scene drawn in the window. Choosing the Reset option from this menu causes the robot to be placed back in the center of the canvas. The Quit option will, as expected, cause the program to terminate.

Program `example1.1.c`

```
#include <stdio.h>
#include "SL.h"

#define CVS_SIDE  500

#define SMALL_RADIUS 10.0   /* radius of robot */
#define STEP_SIZE 1.0       /* size of step for robot to take */

static SL_Polygon Barrier = {4, {{400.0,50.0},
                                 {410.0,50.0},
                                 {410.0,450.0},
                                 {400.0,450.0}}};

/* The simulation state variables */
static SL_Circle Robot;

static void SimuInit(void) /* make RobotColor */
{
   SL_MakeColor("RobotColor", 0.4, 0.0, 0.7);
}

static void SimuReset(void)  /* position small robot in center of canvas */
{
   Robot.center.x = CVS_SIDE/2.0;
   Robot.center.y = CVS_SIDE/2.0;
   Robot.radius = SMALL_RADIUS;
}

static void SimuRedraw(void) /* draw circular robot and polygonal barrier */
{
   SL_SetDrawColor("red");
   SL_DrawPolygon(&Barrier, 1);
   SL_SetDrawColor("RobotColor");
   SL_DrawCircle(&Robot, 1);
}

static int SimuStep(void) /* move robot one step to the right */
{
   SL_Circle SaveRobot;

   SaveRobot = Robot;

   Robot.center.x += STEP_SIZE;
   /* check for intersection with the polygonal barrier */
   if (SL_InterPolygonCircleCheck(&Barrier, &Robot))
   {
      /* undo the move since there was an intersection */
      Robot = SaveRobot;
      return(0); /* stop stepping */
   }
   else /* continue stepping */
      return(1);
}

int main()
{
   SL_Init(CVS_SIDE, CVS_SIDE, 0, SimuInit, SimuReset, SimuRedraw,
           NULL, SimuStep, NULL, NULL, NULL);

   SL_Loop();
   return(0);
}
```

Now that you have seen how this program works, let's see why it works the way it does. This program, simple as it is, illustrates several important properties of any SL-application.

2.1.2 Syntactic Notes

- Every SL-application must contain the line `#include "SL.h"` as one of the compiler directives (line 2 of program `example1.1.c`).
- Every function call and structure name specific to the library begins with the prefix "`SL_`".
- Every SL-application will contain a call to the function `SL_Init()`, which is used to initialize the user interface produced by the program, followed by a call to `SL_Loop()`, which sends control of the program to the user interface. These two statements need not be contiguous but the call to `SL_Init()` must appear before the call to `SL_Loop()` (Section 8.12).

2.1.3 Initializing the Simulation

Each SL-application produces a window that contains in the center a rectangular *drawing canvas* on which the graphical output of the application appears. The size and shape of this drawing canvas is specified in the program by the first two arguments in the call to `SL_Init()`:

```
SL_Init(CVS_SIDE, CVS_SIDE, 0, SimuInit, SimuReset, SimuRedraw,
        NULL, SimuStep, NULL, NULL, NULL);
```

These two values specify the length and width of the drawing canvas in pixels. In `example1.1.c`, it is specified that there should be a square, 500 pixel $\times$ 500 pixel, drawing canvas. The drawing canvas is mapped as a normal xy-plane with origin (0,0) in the lower left-hand corner and the x and y axes growing to the right and upwards, respectively.

The third argument to `SL_Init()` is 0 indicating that the obstacle database should not be enabled (Section 3). The fourth argument to `SL_Init()`, the function `SimuInit()` in this example, is a function that is called when the application's window first appears.

```
SL_Init(CVS_SIDE, CVS_SIDE, 0, SimuInit, SimuReset, SimuRedraw,
        NULL, SimuStep, NULL, NULL, NULL);
```

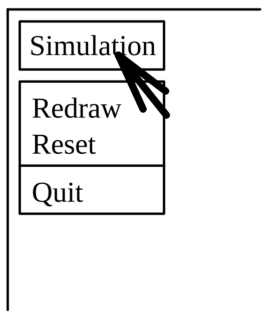


Figure 3: The Simulation menu

This function, which accepts no arguments and returns no value, should contain initialization code that should be executed only once during the run of the application. If there is no such code for your particular application, the fourth argument to `SL_Init()` should be `NULL`. In this example, the initialization that needs to be done is the creation of the color with which the robot will be drawn. This is done by the call:

```
SL_MakeColor("RobotColor", 0.5, 0.2, 0.1);
```

The effect of this call is to establish "RobotColor" as one of the possible *drawing color* names (*i.e.*, to create a new drawing color option). The drawing color is the color with which primitives are drawn on the canvas. It is set by a call to `SL_SetDrawColor()`. (See below.) The three real numbers passed as arguments to `SL_MakeColor()` are the color's RGB (red, green, blue) values. These values should each be in the range $[0.0, 1.0]$. They designate the relative proportions of red, green, and blue in the color. For example, red has RGB values 1.0, 0.0, and 0.0; the RGB values for black are 0.0, 0.0, and 0.0.

`SL_MakeColor()` may be used to define a new color name (as in this example) or redefine an existing color name. There are 11 predefined color names: "white", "black", "red", "green", "blue", "yellow", "cyan", "magenta", "grey", "obstacles", "background". The color "obstacles" is the color with which obstacles in the obstacles database are drawn (Sections 3 and 8.15). This color is originally defined to be black. The "background" color specifies the color of the drawing canvas; the default is white. `SL_MakeColor()` returns 1 if the color was created or redefined successfully, and 0 otherwise.

Note that, in the SL, names of colors are case-insensitive, so "RobotColor" and "robotcolor" are the same color. Also note that, for black and white monitors, colors with intensity < 0.5 will be drawn as black, and those with intensity ≥ 0.5 will be drawn as white. Intensity is measured as: $(0.299 \times red) + (0.587 \times green) + (0.114 \times blue)$.

At the top of an SL-application's window are two built-in buttons: the "Simulation" menu button and the "Go" toggle button. This row of built-in buttons is called the *top row*. Other arguments to `SL_Init()` are used to establish the links between these built-in buttons and a particular application, as explained below.

2.1.4 The Simulation Menu

When pressed, the Simulation button produces a menu with three options: Redraw, Reset, Quit (Figure 3). The functionality of each of these options is explained below.

Redraw The effect of the Redraw option of the Simulation menu is to refresh the scene that appears in the drawing canvas. In `example1.1.c` this is accomplished by the function `SimuRedraw()`. To draw the barrier, the following call to the function `SL_DrawPolygon()` is made:

```
SL_DrawPolygon(&Barrier, 1);
```

Passed as arguments to this function are the address of the `Barrier` structure (Remember that all SL functions are passed pointers to structures even when the structures are not changed by the function (Section 1.3.1).) and the integer 1. The second argument indicates that the interior of the polygon should be filled with the current drawing color. A 0 as this second argument would indicate that only the boundary of the polygon should be drawn. A similar call to `SL_DrawCircle()` is made to draw the robot as a filled circle. Drawing functions are available for all the primitives of the library and work in a similar manner. However, only `SL_Circles`, `SL_Polygons`, and `SL_Arcs` may be filled. Therefore, the functions for the other primitives have only one parameter, which is a pointer to the object to be drawn (Section 8.13.2).

The default drawing color is black. To establish a different drawing color, as is done for both the barrier and the robot in `example1.1.c`, a call to `SL_SetDrawColor()` is made before the drawing is done. This function accepts a character string representing a color name as its argument. If a color with that name has been defined, either because it is one of the predefined colors ("red") or one that has been created with a call to `SL_MakeColor()` ("RobotColor"), the function returns 1 and establishes that color as the current drawing color. It remains the drawing color until another call to `SL_SetDrawColor()` is made. If the color name passed as an argument to `SL_SetDrawColor()` has not been defined, the function returns 0, leaving the drawing color unchanged.

To establish that `SimuRedraw()` is the function to be called when the Redraw option is chosen from the Simulation menu, it is passed as the sixth argument to `SL_Init()` in the sample program:

```
SL_Init(CVS_SIDE, CVS_SIDE, 0, SimuInit, SimuReset, SimuRedraw,
        NULL, SimuStep, NULL, NULL, NULL);
```

This also establishes that this is the function to be called to draw the environment when the application window first appears, when the window is uncovered after being obscured, and to continually update the scene when the Go button has been toggled on. (See below.) The function should accept no arguments and return no value.

Reset The Reset option is used to reinitialize the *simulation state variables* to their starting values. The simulation state variables are the program structures used to capture the current state of the dynamic environment that is being simulated. In `example1.1.c`, the only simulation state variable is `Robot`; `Barrier` is not considered a simulation state variable since it does not change. `SimuReset()` repositions the robot in the center of the drawing canvas (its initial position). Therefore, this is the function that should be called when the Reset option is chosen. To establish this tie, `SimuReset()` is passed as the fifth argument to `SL_Init()`.

```
SL_Init(CVS_SIDE, CVS_SIDE, 0, SimuInit, SimuReset, SimuRedraw,
        NULL, SimuStep, NULL, NULL, NULL);
```

Again, this argument to `SL_Init()` should be a function of no arguments that returns no value. It is not a coincidence that the robot's initial position corresponds to the position assigned in `SimuReset()`. This function is automatically called when the application window first appears.

Quit Choosing this option causes the program to terminate immediately.

2.1.5 The Go Button

The Go button is a toggle that switches the simulation between a *static mode* and an *active mode*. When the simulation is in static mode (as it is when it first starts), none of the objects in the scene is moving. When the Go button is pressed, the simulation switches to an active mode in which the simulation state variables are continuously updated and redisplayed on the drawing canvas. This continues until the Go button is pressed again or the program indicates that there should be no more activity. To produce a smooth animation, the variables should be updated in small increments, which we call *simulation steps*, or simply *steps*. The function that increments the variables by a single step is provided as the eighth argument to `SL_Init()`.

```
SL_Init(CVS_SIDE, CVS_SIDE, 0, SimuInit, SimuReset, SimuRedraw,
        NULL, SimuStep, NULL, NULL, NULL);
```

This function should accept no arguments and return an integer value, which is nonzero if the motion is to continue and 0 if it is to stop. When the simulation is in its active mode, the program **automatically** alternates between calls to this function and the function passed as argument six of `SL_Init()` (the redrawing function) until the Go button is pressed again or the stepping function returns 0. In this way, the animation of the scene is produced.

For our sample program, the continual updating of the simulation state variable and detection of when to stop moving is done by the function `SimuStep()`. A step in this example consists of simply moving the robot to the right by a certain amount (`STEP_SIZE`). The robot should stop moving if it runs into the barrier. To determine if this has occurred, a call to `SL_InterPolygonCircleCheck()` is made:

```
SL_InterPolygonCircleCheck(&Barrier, &Robot)
```

This function determines if the polygon and circle whose addresses are passed as arguments to it intersect. If they do, the function returns 1; if they do not, it returns 0. Similar intersection checking functions are available for all possible pairs of structures in the library (Section 8.9.1). Also available for each pair of structures is an intersection calculation function, which calculates the intersection points (if any) between the two given primitives (Section 8.9.2).

In `SimuStep()`, if the robot has moved to a point where its body intersects the barrier, it is restored to its previous position and the function returns 0, indicating that there should be no more motion. Otherwise, the function returns 1 and the robot continues to step toward the barrier.

2.1.6 The User Interface Loop (UI-loop)

Having explained how the functions in `example1.1.c` are tied to the functionality of the program through the call to `SL_Init()`, it remains to explain the purpose of the call to `SL_Loop()`. This function initiates what is known as the *user interface loop* (UI-loop). Within this loop, the program is continually monitoring its user interface to detect when certain *events* occur. An event corresponds to some sort of action being made by the user of the program either with the mouse or the keyboard (*e.g.*, motion of the mouse in the application window, pressing one of the keys on the keyboard) or with the window itself (*e.g.*, raising or lowering the window, opening or closing the window). When such an event occurs, the program calls the appropriate event-handling routine, also known as a *callback routine*. Which callback routine is appropriate is determined by the function that is *registered* to handle this type of event.

The functions being passed as arguments to `SL_Init()` are being registered as callback routines. This example shows the handling of only five events on the user interface: creating the window; raising or opening the window; the choice of "Redraw" from the Simulation menu; the choice of "Reset" from the Simulation menu; a click on the Go button. These events are handled by the four functions of the the program, as explained above, and are registered by the following call:

```
SL_Init(CVS_SIDE, CVS_SIDE, 0, SimuInit, SimuReset, SimuRedraw,
                    NULL, SimuStep, NULL, NULL, NULL);
```

The callback that handles the event of creating the window is the *initialization callback*, which is the function passed as argument four to `SL_Init()` (`SimuInit()`). Raising and opening the window and selection of the Redraw option from the Simulation menu are all handled by the *redraw callback*, argument six to `SL_Init()` (`SimuRedraw()`). The choice of Reset from the Simulation menu is handled by the function passed as argument five to `SL_Init()`, the *reset callback* (`SimuReset()`). Clicking on the Go button sends the simulation into its active state from which the *step callback* (`SimuStep()`) and *redraw callback* are alternately called until the simulation reverts to its static state. The step callback is passed as argument eight to `SL_Init()`.

If there is nothing for your program to do in case one of these events occurs, the corresponding argument to `SL_Init()` may be `NULL`, as is the case for many of the arguments to `SL_Init()` in `example1.1.c`. However, remember that the reset and redraw callback routines are serving multiple purposes. Each is called when the application window first appears to initialize the scene. The redraw callback is called to refresh the

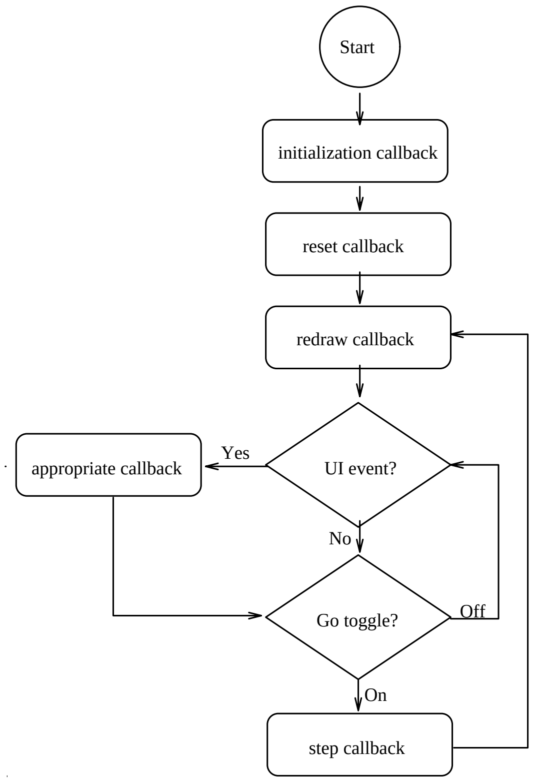


Figure 4: The flow of execution of the UI-loop for an SL-application. The loop starts with a call to `SL_Loop()`. The “appropriate callback” referred to in the diagram is the one that has been registered to handle the type of event that occurred.

scene whenever the window is raised again after becoming partially or completely obscured, and when the simulation is in its active mode to update the scene on the drawing canvas for each step of the simulation.

Finally, note the difference between the initialization callback and the reset callback: the former contains code that is executed only when the window first appears; the latter contains code that is executed when the window first appears **and** every time the Reset option is chosen from the Simulation menu.

The program never returns from the call to `SL_Loop()`; when the Quit option is chosen from the Simulation menu, the program terminates immediately.

The flowchart given in Figure 4 shows the flow of execution of the UI-loop for any SL-application. With this example, we have shown only callback routines that are registered with the call to `SL_Init()`, and we have shown only some of those routines. Later examples expand the set of callback routines that may be registered to handle other types of events, either through the call to `SL_Init()` or through calls to other functions.

2.2 Getting Input from the Mouse and Keyboard

2.2.1 A Programming Example – `example1.2.c`

What the Program Does

Program `example1.2.c` is an expanded version of `example1.1.c` in which we add more functionality through the use of additional callbacks registered with the call to `SL_Init()`. When the application’s window appears on the screen, the only difference between it and the one for `example1.1.c` is the text at

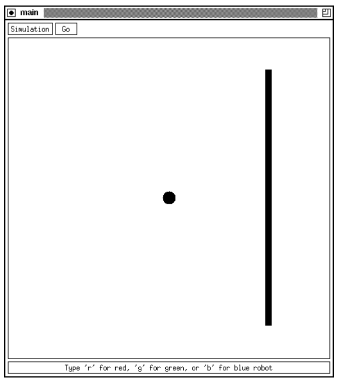


Figure 5: The window produced by program `example1.2.c`

the bottom of the window, which gives instructions on how to change the color of the robot (Figure 5). Typing 'r', 'g', or 'b' changes the color of the robot to red, green, or blue, respectively.

The starting position of the robot may also be changed in this example. If you click the left mouse button with the mouse pointer somewhere in the drawing canvas, the robot will be placed with its center at that position. If you press the right mouse button with the pointer inside the robot's body the robot will follow the mouse pointer until the right button is released. Pressing the middle mouse button does nothing in this example. The built-in buttons in the top row provide the same functionality as in `example1.1.c`. That is, pressing the Go button causes the robot to start moving in a horizontal line from its current position, which may have be specified using the mouse. It moves until it hits the barrier or the Go button is pressed again, and, since the new starting position of the robot may cause it to move above or below the barrier along its horizontal path, a new test has been added to cause the robot to stop moving if it runs into an edge of the drawing canvas. The Reset option of the Simulation menu will reposition the robot in the center of the canvas, just as before.

Program `example1.2.c`

```
#include <stdio.h>
#include "SL.h"

#define CVS_SIDE  500

#define SMALL_RADIUS 10.0  /* the radius of the robot */
#define STEP_SIZE 1.0      /* size of step for robot to take */

SL_Polygon Barrier = {4, {{400.0,50.0},
                          {410.0,50.0},
                          {410.0,450.0},
                          {400.0,450.0}}};

/* The simulation state variables */

int InMotion;  /* indicates if the user is moving the robot with the mouse */

SL_Circle Robot;

static void SimuInit(void) /* make RobotColor and print color changing
                              info to status bar */
{
   SL_MakeColor("RobotColor", 0.4, 0.0, 0.7);
   SL_PrintToStatusBar("Type 'r' for red, 'g' for green, or 'b' for blue robot");
}


static void SimuReset(void)  /* position small robot in center of canvas */
{
   Robot.center.x = CVS_SIDE/2.0;
   Robot.center.y = CVS_SIDE/2.0;
   Robot.radius = SMALL_RADIUS;
   InMotion = 0;
}

static void SimuRedraw(void) /* draw circular robot and polygonal barrier */
{
   SL_SetDrawColor("red");
   SL_DrawPolygon(&Barrier, 1);
   SL_SetDrawColor("RobotColor");
   SL_DrawCircle(&Robot, 1);
}

static int SimuStep(void) /* move robot one step to the right */
{
   SL_Circle SaveRobot;

   SaveRobot = Robot;

   Robot.center.x += STEP_SIZE;

   /* check for intersection with the polygonal barrier & canvas polygon */
   if (SL_InterPolygonCircleCheck(&Barrier, &Robot) ||
       SL_InterCircleCanvasPolygonCheck(&Robot))
   {
      /* undo the move since there was an intersection */
      Robot = SaveRobot;
      return(0); /* stop stepping */
   }
   else /* continue stepping */
      return(1);
}

static void MouseClick(int x, int y, SL_Button btn)
{
   SL_Point LocPnt;
   if (!SL_SteppingOn()) /* mouse clicks have no effect when robot is moving */
      switch (btn)
      {
         case SL_PRESS_LEFT :  /* move robot to the mouse pointer location */
            Robot.center.x = x;
            Robot.center.y = y;
            SL_Redraw();
            break;
         case SL_PRESS_RIGHT :  /* make the robot follow the mouse pointer */
            LocPnt.x = x;
            LocPnt.y = y;
            if ( SL_InsidePointCircle(&LocPnt, &Robot) )
            {
               Robot.center = LocPnt;
               SL_Redraw();
               InMotion = 1; /* robot is now being moved by the user */
            }
            break;
        case SL_RELEASE_RIGHT :
            InMotion = 0; /* robot no longer being moved by the user */
            break;
        default :
            break;
      }
}

static void MouseMove(int x, int y)
{
   if (InMotion)  /* if the robot is being moved by the user */
   {
      Robot.center.x = x; /* make its center current mouse pointer location */
      Robot.center.y = y;
      SL_Redraw();
   }
}

static void ChangeColor(char the_key, int key_pressed)
{
   if (key_pressed) /* do nothing when key is released */
   {
      switch (the_key)
      {
         case 'r' :  /* change robot color to red */
            SL_MakeColor("RobotColor",1.0,0.0,0.0);
            SL_Redraw();
            break;
         case 'g' :  /* change robot color to green */
            SL_MakeColor("RobotColor",0.0,1.0,0.0);
            SL_Redraw();
            break;
         case 'b' :  /* change robot color to blue */
            SL_MakeColor("RobotColor",0.0,0.0,1.0);
            SL_Redraw();
            break;
         default : break;
      }
   }
}

int main()
{
   SL_Init(CVS_SIDE, CVS_SIDE, 0, SimuInit, SimuReset, SimuRedraw,
           NULL, SimuStep, MouseClick, MouseMove, ChangeColor);

   SL_Loop();
   return(0);
}
```

2.2.2 The Status Bar

The area below the drawing canvas in which the message giving instructions for changing the robot's color is printed is known as the *status bar*. Text may be printed to this rectangular area with a call to the function `SL_PrintToStatusBar()`, which accepts as its argument the string of characters to be printed. In `example1.2.c`, this call is made in the initialization callback, `SimuInit()`:

```
SL_PrintToStatusBar(``Type 'r' for red, 'g' for green, or 'b' for blue robot'');
```

2.2.3 The Keyboard Callback

The changing of the robot color is accomplished through the *keyboard callback*. This is the function that is passed as the last argument to `SL_Init()`:

```
SL_Init(CVS_SIDE, CVS_SIDE, 0, SimuInit, SimuReset, SimuRedraw,
    NULL, SimuStep, MouseClick, MouseMove, ChangeColor);
```

Whenever the user presses or releases one of the keys on the keyboard during the run of the application, the keyboard callback routine is called with two arguments: a character indicating which key was either pressed or released and a flag indicating which of these events occurred. In this example, when one of the 'r', 'g', or 'b' keys is pressed, the color "RobotColor", which was defined in the initialization callback, `SimuInit()`, is redefined to the appropriate color with a call to `SL_MakeColor()`. Following this call to redefine the color with which the robot is drawn is a call to `SL_Redraw()`:

```
.
.
.
case 'r' :  /* change robot color to red */
   SL_MakeColor("RobotColor",1.0,0.0,0.0);
   SL_Redraw();
   break;
.
.
.
```

`SL_Redraw()` is a function that causes the scene to be redrawn on the drawing canvas. In this example, this is accomplished in two steps. First, the canvas is cleared with the "background" color. Then, the redraw callback is called. (When the obstacle database has been enabled, a third step is added in which the obstacles are drawn with the color "obstacles" (Section 3).) The call to this function is necessary when the simulation is in its static mode and changes are made to what must be drawn on the canvas. If the call to this function is omitted, the change to the robot color would not be seen until some other event caused the redraw callback to be called. Since `SL_Redraw()` does more than simply call the redraw callback, you should always call it instead of calling your redraw callback directly.

2.2.4 The Mouse Click Callback

The starting position of the robot may be changed in one of two ways using the mouse in this example. The simplest way is by clicking the left mouse button with the mouse pointer at some position on the drawing canvas. The mouse pointer position is assigned to be the robot's new center point. This functionality is accomplished through the *mouse click callback*, which is the function passed as the ninth argument to `SL_Init()`:

```
SL_Init(CVS_SIDE, CVS_SIDE, 0, SimuInit, SimuReset, SimuRedraw,
    NULL, SimuStep, MouseClick, MouseMove, ChangeColor);
```

The mouse click callback is a function that takes two integer arguments representing the x and y positions of the mouse pointer on the screen and an argument of type `SL_Button`, which takes on one of the following six values:

SL_PRESS_LEFT	SL_PRESS_MIDDLE	SL_PRESS_RIGHT
SL_RELEASE_LEFT	SL_RELEASE_MIDDLE	SL_RELEASE_RIGHT

These indicate the type of mouse button event that occurred.

For this example, we are interested in mouse button events only if they occur while the application is in its static mode. To make sure the events are ignored when the application is in its active mode, the mouse click callback routine contains the following statement:

```
if (!SL_SteppingOn()) /* mouse clicks have no effect when robot is moving */
   switch (btn)
   {
      .
      . [ mouse button event handling code ]
      .
   }
```

`SL_SteppingOn()` is a function that simply returns 1 when the Go button has been toggled on and 0 when it has been toggled off. Thus, it indicates if the application is in active or static mode.

When the left mouse button is pressed with the application in its static mode, the following code accomplishes the repositioning of the robot:

```
      case SL_PRESS_LEFT :  /* move robot to the mouse pointer location */
         Robot.center.x = x;
         Robot.center.y = y;
         SL_Redraw();
         break;
```

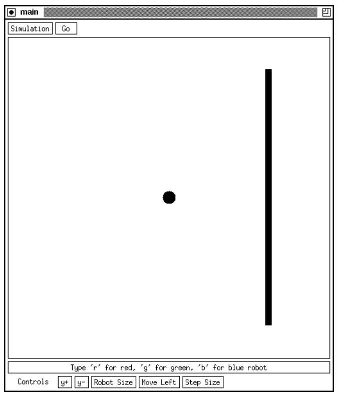
Notice the call to `SL_Redraw()`, which is necessary to draw the updated environment with the robot in its new position.

Clicking with the left mouse button is not the only way the robot may be repositioned with the mouse in this example. The robot's position may also be changed by clicking the right mouse button with the mouse pointer inside the robot's body, moving the mouse pointer to another position on the canvas, and then releasing the right mouse button. During the mouse motion, the robot follows the mouse pointer.

To detect when this sort of repositioning is to begin, every time the right mouse button is clicked, we must check whether the mouse pointer is inside the robot's body. This is accomplished by a call to `SL_InsidePointCircle()`, which takes a pointer to an `SL_Point` and a pointer to an `SL_Circle` and returns 1 if the point is inside the circle and 0 otherwise. There is a similar function for checking if a point is inside a polygon (Section 8.9.1).

```
      case SL_PRESS_RIGHT :  /* make the robot follow the mouse pointer */
         LocPnt.x = x;
         LocPnt.y = y;
         if ( SL_InsidePointCircle(&LocPnt, &Robot) )
         {
            Robot.center = LocPnt;
            SL_Redraw();
            InMotion = 1; /* robot is now being moved by the user */
         }
```

If the mouse pointer is inside the robot's body, the pointer location becomes the new center position of the robot, the scene is redrawn with this new robot location and the simulation state variable `InMotion` is set to 1 to indicate that the user is now moving the robot with the mouse. When the right mouse button is released, this means the user has placed the robot in its new position, so `InMotion` is set back to 0. Notice that `InMotion` is initialized in the reset callback, `SimuReset()`.



Type 'r' for red, 'g' for green, 'b' for blue robot

Figure 6: The window produced by program `example1.3.c`

2.2.5 The Mouse Motion Callback

Changing the robot position to follow the mouse pointer when `InMotion` has been set to 1 is accomplished by the *mouse motion callback*. This is the function that is passed as the tenth argument to `SL_Init()`:

```
SL_Init(CVS_SIDE, CVS_SIDE, 0, SimuInit, SimuReset, SimuRedraw,
    NULL, SimuStep, MouseClick, MouseMove, ChangeColor);
```

This function is called every time the mouse is moved with its pointer inside the drawing canvas. It is passed two integer arguments representing the current x and y position of the mouse pointer. Thus to move the robot along with the mouse pointer, we need only make the point (x, y) the center of the robot and redraw the scene with a call to `SL_Redraw()`. This is accomplished with the function `MouseMove()` in `example1.2.c`.

2.2.6 Intersecting the Canvas Polygon

When the robot has been repositioned using either the left or right mouse buttons, its horizontal path to the right may not cause it to hit the barrier. In this case, we want the robot to stop when it reaches the right edge of the canvas, or, in other words, when its body intersects the *canvas polygon*. The canvas polygon is the rectangular polygon surrounding the drawing canvas. The following function call from `SimuStep()` determines if an intersection has occurred:

```
SL_InterCircleCanvasPolygonCheck(&Robot)
```

This function works just like the other intersection-checking functions: it returns 1 if there is an intersection and 0 otherwise. The only difference is in the number of arguments. There are similar functions for checking for intersections (and calculating intersection points) with the canvas polygon for all the primitives of the library (Section 8.6).

2.3 Customizing the User Interface

2.3.1 A Programming Example – `example1.3.c`

What the Program Does

Program `example1.3.c` is a further expansion of the example introduced in Section 2.1 and built upon in Section 2.2. With this expansion, five application-specific buttons labeled as the "Controls" have been added to the bottom of the user interface (Figure 6).

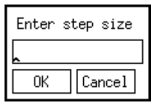


Figure 7: A sample dialog pop-up window

The functionality of `example1.2.c` is retained, so the robot may be repositioned either by clicking the left mouse button with the pointer somewhere in the canvas or by pressing the right mouse button with the pointer in the robot's body, moving the pointer to a new position, and releasing the right button. The new buttons on the application's window provide an additional way for the user to change the position of the robot, as well as means of altering other environment parameters. All these buttons are available for use while the robot is moving.

The vertical position of the robot may be changed using the buttons labeled "y+" and "y-". When a mouse button is pressed with the pointer over one of these buttons, the robot moves in the positive (up) or negative (down) y direction. It continues to move in this direction until the mouse button is released.

The size (radius) of the robot may be changed with the menu button labeled "Robot Size". This menu has three options: Small, Medium, and Large. The robot initially shown on the screen is "Small". Choosing one of the other options on this menu causes the robot to immediately change size. The button labeled "Move Left" is a toggle button that allows the user to choose the horizontal direction the robot will move when the Go button is toggled on. The initial direction is right (just as in the other examples).

The last button, labeled "Step Size", allows the user to change the size of the step the robot takes when when the Go button is toggled on or when the "y+" and "y-" buttons are being used. When a mouse button is clicked over this button, a dialog window like the one shown in Figure 7 will pop up.

When the user types a number in the text area and clicks the "OK" button, the robot's step size is changed to that number, and the window disappears. If the "Cancel" button is clicked, the dialog window disappears and the step size is not changed.

Program `example1.3.c`

```
#include <stdio.h>
#include <stdlib.h>
#include "SL.h"

#define CVS_SIDE  500

#define SMALL_RADIUS 10.0
#define MEDIUM_RADIUS 20.0
#define LARGE_RADIUS 40.0

static SL_Polygon Barrier = {4, {{400.0,50.0},
                                 {410.0,50.0},
                                 {410.0,450.0},
                                 {400.0,450.0}}};

/* The simulation state variables */

static int InMotion;  /* indicates if user is moving robot with mouse */
static int MoveLeft;  /* 1/0 if robot moves left/right when stepping */
static double StepSize; /* size of step for robot to take */

static SL_Circle Robot;

static void SimuInit(void) /* initialize the simulation state variables */
```

```
{
   MoveLeft = 0;
   StepSize = 1.0;
   SL_MakeColor("RobotColor", 0.4, 0.0, 0.7);
   SL_PrintToStatusBar("Type 'r' for red, 'g' for green, 'b' for blue robot");
}

static void SimuReset(void)  /* position small robot in center of canvas */
{
   Robot.center.x = CVS_SIDE/2.0;
   Robot.center.y = CVS_SIDE/2.0;
   Robot.radius = SMALL_RADIUS;
   InMotion = 0;
}

static void SimuRedraw(void) /* draw circular robot and polygonal barrier */
{
   SL_SetDrawColor("red");
   SL_DrawPolygon(&Barrier, 1);
   SL_SetDrawColor("RobotColor");
   SL_DrawCircle(&Robot, 1);
}

static int SimuStep(void) /* move robot one step to the left or right */
{
   SL_Circle SaveRobot;

   SaveRobot = Robot;

   if (MoveLeft)
      Robot.center.x -= StepSize;
   else
      Robot.center.x += StepSize;

   /* check for intersection with the polygonal barrier & canvas polygon */
   if (SL_InterPolygonCircleCheck(&Barrier, &Robot) ||
       SL_InterCircleCanvasPolygonCheck(&Robot))
   {
      /* undo the move since there was an intersection */
      Robot = SaveRobot;
      return(0); /* stop stepping */
   }
   else /* continue stepping */
      return(1);
}

static void MouseClick(int x, int y, SL_Button btn)
{
   SL_Point LocPnt;
   if (!SL_SteppingOn()) /* mouse clicks have no effect when robot is moving */
      switch (btn)
      {
         case SL_PRESS_LEFT :  /* move robot to the mouse pointer location */
            Robot.center.x = x;
            Robot.center.y = y;
            SL_Redraw();
            break;
         case SL_PRESS_RIGHT :  /* make the robot follow the mouse pointer */
            LocPnt.x = x;
            LocPnt.y = y;
            if ( SL_InsidePointCircle(&LocPnt, &Robot) )
            {
               Robot.center = LocPnt;
               SL_Redraw();
               InMotion = 1; /* robot is now being moved by the user */
            }
            break;
         case SL_RELEASE_RIGHT :
            InMotion = 0; /* robot no longer being moved by the user */
            break;
        default :
            break;
      }
}

static void MouseMove(int x, int y)
{
   if (InMotion)  /* if the robot is being moved by the user */
   {
      Robot.center.x = x; /* make its center current mouse pointer location */
      Robot.center.y = y;
      SL_Redraw();
   }
}

static void ChangeColor(char the_key, int key_pressed)
{
   if (key_pressed) /* do nothing when key is released */
   {
      switch (the_key)
      {
         case 'r' :  /* change robot color to red */
            SL_MakeColor("RobotColor",1.0,0.0,0.0);
            SL_Redraw();
            break;
         case 'g' :  /* change robot color to green */
            SL_MakeColor("RobotColor",0.0,1.0,0.0);
            SL_Redraw();
            break;
         case 'b' :  /* change robot color to blue */
            SL_MakeColor("RobotColor",0.0,0.0,1.0);
            SL_Redraw();
            break;
         default : break;
      }
   }
}

static void MoveRobotUp(void)
{
   Robot.center.y += StepSize;
   SL_Redraw();
```

```
}

static void MoveRobotDown(void)
{
   Robot.center.y -= StepSize;
   SL_Redraw();
}

static void ChangeRobotSize(int menu_index)
{
   switch (menu_index)
   {
      case 0 :
         Robot.radius = SMALL_RADIUS;
         break;
      case 1 :
         Robot.radius = MEDIUM_RADIUS;
         break;
      case 2 :
         Robot.radius = LARGE_RADIUS;
         break;
      default : break;
   }
   SL_Redraw();
}

static void ChangeRobotDirection(int toggle_on)
{
   MoveLeft = toggle_on;
}

static void ChangeStepSize(char *step_size)
{
   StepSize = atof(step_size);
}

int main()
{
   char *SizeNames[] = {"Small", "Medium", "Large", NULL};

   SL_Init(CVS_SIDE, CVS_SIDE, 0, SimuInit, SimuReset, SimuRedraw,
   NULL, SimuStep, MouseClick, MouseMove, ChangeColor);

   SL_AddLabel("Controls ");
   SL_AddButton("y+", 1, MoveRobotUp);
   SL_AddButton("y-", 1, MoveRobotDown);
   SL_AddMenu("Robot Size", SizeNames, ChangeRobotSize);
   SL_AddToggle("Move Left", ChangeRobotDirection);
   SL_AddDialog("Step Size", "Enter step size",ChangeStepSize);

   SL_Loop();
   return(0);
}
```

2.3.2 The Bottom Row

The *bottom row* of an SL-application's window is the area below the status bar where the programmer may place labels and buttons to expand the functionality of the user interface. There are four types of buttons that may be attached to the window: push buttons, menu buttons, toggle buttons, and dialog buttons. Each of these is illustrated in this example and explained below.

Buttons and labels are added to the bottom row through various function calls. There may be any combination of buttons and labels; their order on the window is determined by the order in which the corresponding functions are called. Calls to these functions must appear in the program after the call to `SL_Init()` and before the call to `SL_Loop()`. Just as with the buttons on the top row of the window, the bottom-row buttons achieve their functionality through callback routines. These are registered by the calls to the functions that create the buttons.

2.3.3 Labels

The word "Controls" at the beginning of the bottom row in the window shown in Figure 6 is a *label*. It was added to the bottom row of `example1.3`'s window with the following function call in the main program:

```
SL_AddLabel(``Controls '');
```

A label is simply a piece of text placed on the user interface, usually used to explain the uses of the buttons on the bottom row. Unlike the text displayed in the status bar, the text of a label may not be changed as the program is running.

2.3.4 Push Buttons

The buttons labeled "`y+`" and "`y-`" are *push buttons*. They are created by calls to the function `SL_AddButton()`:

```
 SL_AddButton(``y+'', 1, MoveRobotUp);
SL_AddButton(``y-'', 1, MoveRobotDown);
```

The first argument to `SL_AddButton()` is a character string, which is used to label the button. The second argument should be either 1 or 0. This indicates if the button should have auto-repeat capabilities or not. If it does (as in this example), when the user presses a mouse button with the mouse pointer over the push button, the callback function will be continuously called until the mouse button is released. If a push button does not have auto-repeat capabilities, the callback function is called once for each click of the mouse button.

The callback routine, which accomplishes the intended function of the push button, is the third argument in the call to `SL_AddButton()`. It should be a function that takes no arguments and returns no value, as here, for example:

```
static void MoveRobotUp(void)
{
   Robot.center.y += StepSize;
   SL_Redraw();
}
```

Note the call to `SL_Redraw()` at the end of this function. This causes the robot to be drawn in its new position immediately.

2.3.5 Menu Buttons

Adding a *menu button* to the bottom row is accomplished through a call to `SL_AddMenu()`. This function also takes three arguments. The first is a character string with which to label the button; the second is a `NULL`-terminated array of character strings, which are to be the menu options; the third is the callback function. The callback function should be a function with no return value that takes a single integer argument representing the menu option chosen by the user. This integer will be in the range $[0, n-1]$, where n is the number of menu options.

In `example1.3.c`, we created a menu button with three options with the following call:

```
SL_AddMenu(``Robot Size'', SizeNames, ChangeRobotSize);
```

where `SizeNames` was initialized as:

```
char *SizeNames[] = {``Small'', ``Medium'', ``Large'', NULL};
```

The callback function `ChangeRobotSize()` is typical of one for any menu button. It contains simply a switch statement that changes the robot's radius to the appropriate value for the menu option chosen and then redraws the scene.

2.3.6 Toggle Buttons

A *toggle button* much like the Go button in the top row may be added to the bottom row using the function `SL_AddToggle()`. This function takes two arguments: the character string label for the button and the callback function. Each time a mouse button is clicked with the mouse pointer over a toggle button, its callback function is called with either a 1 or a 0 to indicate if the button has been toggled "on" or "off". When the application begins, all toggle buttons are in the "off" position.

In `example1.3.c`, the button labeled "Move Left" is a toggle button. Its callback function, registered with the following call:

```
SL_AddToggle(``Move Left'', ChangeRobotDirection);
```

sets the state variable `MoveLeft` to the toggle-on-flag value. This variable is used in the step callback routine, `SimuStep()`, to decide whether to move the robot to the left or right with each step.

2.3.7 Dialog Buttons

The final button on the bottom row of `example1.3.c`'s window, labeled "Step Size", is a *dialog button*. Clicking a mouse button with the mouse pointer over a dialog button causes a *dialog window* to appear. A dialog window contains a *dialog box*, in which the user may type characters. Above this box is an area in which text may be displayed. This text usually takes the form of instructions to the user about what to type in the dialog box. Below the dialog box are two buttons: one labeled "OK" and the other "Cancel". If the user clicks on the "OK" button, the characters typed in the dialog box are passed as the argument to the callback routine associated with the dialog button, and the window disappears. If the user clicks on the "Cancel" button, the window disappears, but the callback routine is not called.

A dialog button is added to an SL-application's window with a call to `SL_AddDialog()`. This call specifies the character string with which to label the button, the character string to display above the dialog box, and the callback routine to call when the user clicks on the OK button in the dialog window. For `example1.3.c`, the call is:

```
SL_AddDialog(``Step Size'', ``Enter step size'', ChangeStepSize);
```

and the resulting dialog window looks like the one shown in Figure 7.

The callback function `ChangeStepSize()` simply converts the character string representing the number the user typed in the dialog box into a real number and assigns this as the new value of `StepSize`:

```
static void ChangeStepSize(char *step_size)
{
   StepSize = atof(step_size);
}
```

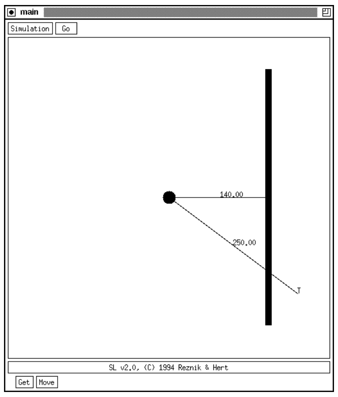


Figure 8: The window produced by program `example1.4.c`

2.4 Mouse Editing, Distances and Drawing Text

2.4.1 A Programming Example – `example1.4.c`

What the Program Does

The example in this section is in much the same spirit as the previous three examples — a circular robot moves along a straight line until a stopping condition is satisfied with the user being able to specify the size and position of the robot (as well as the polygonal barrier). The window produced by program `example1.4.c` is shown in Figure 8. A small, blue circular robot appears in the center of the drawing canvas and a red rectangular barrier is to the right of the robot. There is also a point labeled T, the target point of the robot. A line is drawn from the robot's center to the target point, and this line is labeled with its length. A second line is drawn from a point on the robot's body to a point on the barrier and labeled with its length. This length represents the minimum distance between the circle and the polygon; the line is drawn between the closest points on the two objects.

When the Go button is pressed, the robot will move along a straight line toward the point T and the two distance measurements will be continually updated. The robot will stop when it reaches the target point or it hits the barrier or the canvas polygon. The Reset option of the Simulation menu has no effect for this program, other than to redraw the scene.

The location of the target point can be changed by clicking the left mouse button somewhere inside the drawing canvas. The robot and barrier may also be moved using the "Move" menu in the bottom row. This menu has two options, "Robot" and "Barrier". When one of these options is chosen the program waits for the user to click the mouse somewhere inside the canvas. This defines a reference point. As the mouse is moved within the canvas, the object chosen from the menu moves so as to maintain the same relative position between the mouse pointer and the object center (or centroid) as between the reference point and the original object position. When the object has been moved to its desired location, the user may press the left or middle mouse button to place the object in its new location. If the right mouse button is pressed, the move operation is aborted and the object returns to its original location.

The "Get" menu in the bottom row provides another means by which the robot and barrier may be changed. Again, this menu's options are "Robot" and "Barrier". If the "Robot" option is chosen, the program waits for the user to click either the left or middle mouse button somewhere inside the canvas. This defines the new center for the robot. Then, as the mouse is moved away from this point a circle is drawn with radius equal to the distance between the pointer position and the new center. When the desired radius is reached, the user may click either the left or middle mouse button to define the new robot. Clicking the right mouse button aborts this operation as well and does not change the size or position of the robot. If the "Barrier" option is chosen, the user is allowed to input a new polygonal barrier using the mouse. This is done by clicking the left mouse button in the canvas at the barrier vertex points. When all vertices have been entered using the left mouse button, the middle mouse button should be pressed to close the polygon. As before, clicking the right mouse button at any time aborts the input process. The program will not allow the user to input vertices that cause the polygon to be self-intersecting and if the user enters `SL_MAX_VTX` vertices, the polygon is automatically closed since no more vertices may be added.

Program `example1.4.c`

```
#include <stdio.h>
#include <stdlib.h>
#include "SL.h"

#define CVS_SIDE  500

#define SMALL_RADIUS 10.0
#define STEP_SIZE 1.0

/* The simulation state variables */
SL_Polygon Barrier = {4, {{400.0,50.0},
                          {410.0,50.0},
                          {410.0,450.0},
                          {400.0,450.0}}};
SL_Circle Robot;

SL_Point Target;  /* the target point for the robot */

static void SimuInit(void) /* initialize the simulation state variables */
{
   Robot.center.x = CVS_SIDE/2.0;
   Robot.center.y = CVS_SIDE/2.0;
   Robot.radius = SMALL_RADIUS;
   Target.x = 450.0;
   Target.y = 100;
}

/* draws a line segment between *start and *end and, at the midpoint of this
   segment, displays the distance between the two */
static void DrawDistance(SL_Point *start, SL_Point *end, double dist)
{
   SL_Segment DistLine;
   SL_Point MidPt;
   char dist_text[32];

   SL_SetDrawColor("black");
   DistLine.start = *start;
   DistLine.end = *end;
   SL_SegmentMidpoint(&DistLine,&MidPt);
   sprintf(dist_text,"%.2f",dist);

   SL_DrawSegment(&DistLine);
   SL_DrawText(&MidPt,dist_text);
}


static void SimuRedraw(void) /* draw the robot, barrier, and target point */
```

```
{
   SL_Point PolyPt, CirclePt;
   double dist;

   SL_SetDrawColor("red");
   SL_DrawPolygon(&Barrier, 1);
   SL_SetDrawColor("blue");
   SL_DrawCircle(&Robot, 1);
   SL_DrawPoint(&Target);
   SL_DrawText(&Target,"T");

   dist = SL_DistPointPoint(&(Robot.center),&Target);
   DrawDistance(&(Robot.center),&Target,dist);
   dist = SL_DistPolygonCircle(&Barrier,&Robot,&PolyPt,&CirclePt);
   DrawDistance(&PolyPt,&CirclePt,dist);
}

static int SimuStep(void) /* move robot one step to the left or right */
{
   SL_Circle SaveRobot;
   int AtTarget = 0;

   SaveRobot = Robot;

   if (SL_DistPointPoint(&Robot.center,&Target) <= STEP_SIZE)
   {
      Robot.center = Target;
      AtTarget = 1;
   }
   else
      SL_MoveTowardsPoint(&Robot.center,&Target,STEP_SIZE,&Robot.center);

   /* check for intersection with the polygonal barrier & canvas polygon */
   if (SL_InterPolygonCircleCheck(&Barrier, &Robot) ||
       SL_InterCircleCanvasPolygonCheck(&Robot))
   {
      /* undo the move since there was an intersection */
      Robot = SaveRobot;
      return(0); /* stop stepping */
   }
   else /* continue stepping if robot not at target*/
      return(!AtTarget);
}

static void SetTarget(int x, int y, SL_Button btn)
{
   /* change target only when robot is not moving and left mouse button
      is clicked */
   if (!SL_SteppingOn() && (btn == SL_PRESS_LEFT))
   {
      Target.x = x;
      Target.y = y;
      SL_Redraw();
   }
}
```

```
static void MoveObject(int menu_index)
{
   SL_Point RefPoint;

   switch(menu_index)
   {
      case 0 :
         SL_MouseGetPoint(&RefPoint);
         SL_MouseMoveCircle(&Robot);
         break;
      case 1 :
         SL_MouseGetPoint(&RefPoint);
         SL_MouseMovePolygon(&Barrier);
         break;
      default : break;
   }
   SL_Redraw();
}

static void GetObject(int menu_index)
{
   switch(menu_index)
   {
      case 0 :
         SL_MouseGetCircle(&Robot);
         break;
      case 1 :
         SL_MouseGetPolygon(&Barrier);
         break;
      default : break;
   }
   SL_Redraw();
}


int main()
{
   char *Objects[] = {"Robot", "Barrier", NULL};

   SL_Init(CVS_SIDE, CVS_SIDE, 0, SimuInit, NULL, SimuRedraw,
           NULL, SimuStep, SetTarget, NULL, NULL);
   SL_AddMenu("Get", Objects, GetObject);
   SL_AddMenu("Move", Objects, MoveObject);

   SL_Loop();
   return(0);
}
```

2.4.2 Changing Primitives with the Mouse

The functionality of the Get and Move menus in the bottom row of `example1.4.c`'s application window is provided by the `SL_Mouse*` family of functions (Section 8.14). These functions allow the user to input, move, rotate, and scale library primitives using the mouse. The functions `SL_MouseGetCircle()` and `SL_MouseGetPolygon()` used in this program, each accepts a pointer to the location in which to store the input object. The input process for each primitive is slightly different, as explained in Section 8.14.

To move the robot and barrier using the mouse, the functions `SL_MouseMoveCircle()` and `SL_MouseMovePolygon()`

are used. Again, each accepts a pointer to the object to be moved. When each function is called, the relative positions of the mouse pointer and the given object's center is determined. This relative position is maintained throughout the move operation. To allow the user to specify the point of reference for moving, a call to `SL_MouseGetPoint()` precedes the calls to the `SL_MouseMove*` functions:

```
    case 0 :
       SL_MouseGetPoint(&RefPoint);
       SL_MouseMoveCircle(&Robot);
       break;
         .
         .
         .
```

`SL_MouseGetPoint()` allows the user to input a point by clicking the left or middle mouse button with the mouse pointer somewhere inside the canvas. Without the call to `SL_MouseGetPoint()`, the point of reference would be determined by the point at which the mouse pointer entered the canvas after choosing the appropriate option from the Move menu. This entry point could potentially be very far from the object in question, which may limit the range of mobility for that object.

2.4.3 Moving Toward a Point

The step callback of this program (`SimuStep()`) causes the robot to move a certain distance (`STEP_SIZE`) along a straight line toward its target point, as in the previous three examples. However, since this straight line may be at any orientation, simply adding `STEP_SIZE` to the x coordinate of the robot's center is not sufficient. Instead, the function `SL_MoveTowardsPoint()` is used:

```
   SL_MoveTowardsPoint(&Robot.center,&Target,STEP_SIZE,&Robot.center);
```

This function accepts three input parameters: the starting point (`&RobotCenter`) the ending point (`&Target`) and the amount by which to move (`STEP_SIZE`). Its single output parameter (`&Robot.center`) is the point along the line from the starting point to the target point that is the designated distance from the starting point. The function returns 0 if the starting point and ending point coincide and leaves the output parameter unchanged. Otherwise, the function returns 1. Note that in this program it is first determined if the robot is currently less than a single step away from the target:

```
  if (SL_DistPointPoint(&Robot.center,&Target) <= STEP_SIZE)
  {
     Robot.center = Target;
     AtTarget = 1;
  }
  else
     SL_MoveTowardsPoint(&Robot.center,&Target,STEP_SIZE,&Robot.center);
```

If this test were not done, then in moving toward the target point by a distance of `STEP_SIZE`, the robot could overshoot the target point any number of times with the potential of never actually reaching the target.

2.4.4 Computing Distances

The `SL_Dist*` family of functions (Section 8.10) is used to compute minimum distances between pairs of library primitives. In each case, for each nonpoint primitive, the function returns though an output parameter the point on the primitive to which the distance corresponds. For example, in `example1.4.c`, the following call is made to compute the distance between the robot and the barrier:

```
  dist = SL_DistPolygonCircle(&Barrier,&Robot,&PolyPt,&CirclePt);
```

`PolyPt` and `CirclePt` are the two endpoints of the minimum distance line drawn on the canvas.

2.4.5 Drawing Text

To label the line segments with their respective lengths, the function `SL_DrawText()` is used. This function prints a string of text at a specified location in the current drawing color. In this example, the location at which the text is drawn is the midpoint of the line segment being labeled. The midpoint is calculated using the function `SL_SegmentMidpoint`:

```
SL_SegmentMidpoint(&DistLine,&MidPt);
sprintf(dist_text,"%.2f",dist);

SL_DrawSegment(&DistLine);
SL_DrawText(&MidPt,dist_text);
```

3 Adding Obstacles to the Environment

The motion of a robot within an environment filled with *obstacles* is a feature common to many simulation programs . SL supports this feature by maintaining a list of SL geometric primitives (circles, polygons, etc.) called the *obstacle database* (ODB) SL provides an on-screen interface for creating and editing the ODB's contents. Additionally, SL provides a set of routines that check for inclusion and intersection of geometric primitives and obstacles stored in the ODB. The contents of the ODB can also be retrieved and set at the programming level. This feature allows for simulations involving moving or deforming obstacles. The functions for activating and interacting with the ODB are introduced in this section and describe in more detail in Section 8.15.

3.1 Activating the Obstacle Database

To activate the obstacle database, a non-zero integer must be passed as the third parameter in the call to `SL_Init()`:

```
SL_Init(200, 200, 1, SimuInit, SimuReset, SimuRedraw, NULL, SimuStep, NULL, NULL, NULL);
```

When the window is opened, its top row will contain the usual Simulation menu button and Go button, plus the *Obstacle menu*, as shown in Figure 9.

3.2 Interactive Creation and Editing of the Obstacle Database

The options within the Obstacle menu are grouped into 4 sections: (1) Adding an obstacle (Circle, Triangle, Square, Rectangle, Polygon), (2) Editing an existing obstacle (Move vtx, Move, Copy, Scale, Rotate), (3) Removing obstacle(s) (Delete, Delete All), and (4) File operations (Save, Load, Merge) (Figure 9).

The text below describes the means by which the user interacts with the obstacle database through this menu. In what follows, "clicks" refer to clicks of the left mouse button. All operations explained below can be cancelled at any time by a click of the right mouse button.

3.2.1 Adding Obstacles

The database contains no obstacles when the application starts. To add an obstacle to the environment, the user may choose the appropriate shape from the first section of the Obstacle menu. After either "Circle", "Triangle", or "Square" is chosen from the menu, an outline of the corresponding obstacle shape will follow the mouse pointer until a click "drops" the floating obstacle at its current location. A filled obstacle will be drawn at that location in the color "obstables" (either as predefined (black) or as redefined using `SL_MakeColor()` (Sections 2.1 and 8.13)).

If the "Rectangle" option is chosen from the first section of the Obstacle menu, the user may specify the rectangle by first clicking on the desired top left corner and then moving the pointer to the desired lower right corner and clicking again. After the click for the upper left corner, a rubber-band outline of the

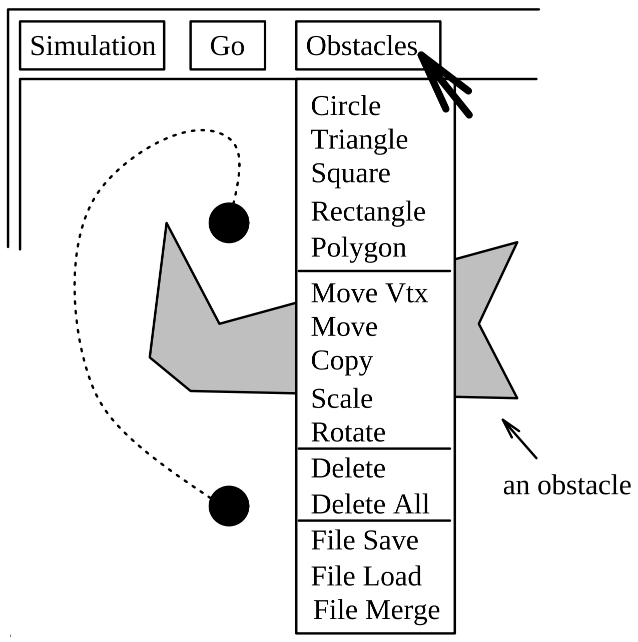


Figure 9: The obstacle database menu.

rectangle will be drawn with its lower right corner at the mouse pointer position. For the "Polygon" option, each polygon vertex is specified by a click of the left mouse button. The middle mouse button is used to close the polygon. Just as for the function `SL_MouseGetPolygon()` (Section 2.4), self-intersecting polygons and polygons with more than `SL_MAX_VTX` vertices are not allowed.

Any number of obstacles can be added to the environment using these first five menu options.

3.2.2 Editing Obstacles

The options in the second section of the Obstacle menu may be used to modify the shape or position of obstacles already in the database.

To change the position of individual polygon vertices, the "Move Vtx" option from the Obstacle menu should be chosen. Then the user must click on the polygon that is to be reshaped (This will turn it into an outline.) and then click as near as possible to the vertex to be moved. From then on the sides meeting at the selected vertex will follow the mouse pointer. When the pointer has been moved to the desired new location for the selected vertex, another click will effectuate the change. Again, self-intersecting polygons will not be accepted.

To move an obstacle currently in the ODB, the "Move" option from the Obstacle menu is used. The user clicks on the obstacle to move (The selected obstacle becomes a pointer-following outline.), and clicks again at the obstacle's new location.

To place a copy of an existing obstacle at another location, the user should select the "Copy" option and click on the obstacle to be copied. (An outline of the selected obstacle will follow the pointer.) The pointer should then be moved to place the copied obstacle somewhere in the environment and then a click will drop it into place.

Scaling (rotating) an obstacle can be accomplished by selecting "Scale" ("Rotate") from the Obstacle menu, and then clicking on the obstacle to be scaled (rotated). This will turn the selected obstacle into an outline that will resize (turn) according to the pointer's location relative to the obstacle's center. A click finishes the operation when the desired size (orientation) is achieved.

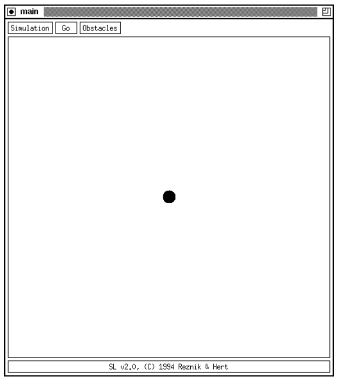


Figure 10: The application window for `obst.c`

3.2.3 Deleting Obstacles

An individual obstacle currently in the database can be deleted by selecting "Delete" from the Obstacle menu and clicking in the interior of the obstacle to be deleted. The deleted obstacle will disappear. To delete all obstacles currently in the database, select "Delete All".

3.2.4 File Operations

The ODB's contents can be saved to a file by selecting "File Save" within the Obstacle menu. A dialog window will appear in which the name of the file in which to save the obstacles may be entered. To load a previously saved file, select "File Load" from the obstacle menu, enter the name of the file you wish to load in the the dialog window, and click on "OK". The contents of the named file will replace the current contents of the ODB. If you wish to merge the contents of an obstacle file with the ODB's current contents, select "File Merge" from the Obstacle menu. See Section 8.5 for a description of the shape file format.

3.3 A Programming Example — `obst.c`

For the purposes of illustration, let us refer back to program `example1.1.c`, presented in Section 2.1. In that program, a circular robot is made to move to the right along a horizontal line until it intersects with a polygonal barrier. Below, a variant of this program is shown in which the barrier becomes the set of obstacles currently in the ODB. The window produced by this program is shown in Figure 10.

```
#include <stdio.h>
#include "SL.h"

#define CVS_SIDE 500        /* window side in pixels */
#define SMALL_RADIUS 10.0   /* radius of robot */
#define STEP_SIZE 1.0       /* size of step for robot to take */

/* The simulation state variables */
static SL_Circle Robot;

static void SimuInit(void)
{
   SL_MakeColor("obstacles", 0.35, 0.1, 0.1);
   SL_MakeColor("RobotColor", 0.4, 0.0, 0.7);
}

static void SimuReset(void)  /* position small robot in center of canvas */
```

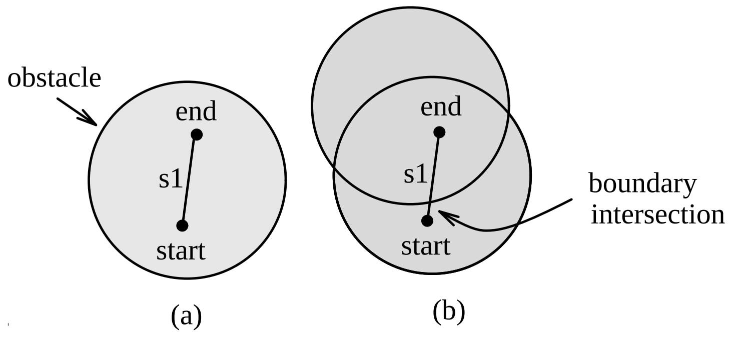


Figure 11: (a) Segment `s1` does not intersect the boundary of the circular obstacle and `SL_InterSegmentObstCheck(&s1)` returns 0. (b) The addition of another obstacle causes the same function call to return 1.

```
{
   Robot.center.x = CVS_SIDE/2.0;
   Robot.center.y = CVS_SIDE/2.0;
   Robot.radius = SMALL_RADIUS;
}

static void SimuRedraw(void) /* draw circular robot */
{
   SL_SetDrawColor("RobotColor");
   SL_DrawCircle(&Robot, 1);
}

static int SimuStep(void) /* move robot one step to the right */
{
   SL_Circle SaveRobot;
   SaveRobot = Robot;
   Robot.center.x += STEP_SIZE; /* move robot to the right */

   /* check for intersection with any obstacle in the ODB */
   if (SL_InterCircleObstCheck(&Robot))
   { /* undo the move since there was an intersection */
      Robot = SaveRobot;
      return(0); /* stop stepping */
   }
   else /* continue stepping */
      return(1);
}

int main()
{
   SL_Init(CVS_SIDE, CVS_SIDE, 1, SimuInit, SimuReset, SimuRedraw,
      NULL, SimuStep, NULL, NULL, NULL);
   SL_Loop();
   return(0);
}
```

The above program is essentially a copy of `example1.1.c` with the following modifications:

- In `main()` the third argument passed to `SL_Init()` has been changed to 1, so as to activate the obstacle database.
- In `SimuInit()` the call `SL_MakeColor(''obstacles'', 0.35, 0.1, 0.1);` has been added. This defines a reddish-brown drawing color for the obstacles.
- The `SL_Polygon Barrier` global variable has been eliminated. (The barrier is now whatever is in the ODB.)
- In the function `SimuStep()`, the call to `SL_InterPolygonCircleCheck(&Barrier, &Robot)` has been replaced by a call to `SL_InterCircleObstCheck(&Robot)`. The intersection test determines if the robot intersects any obstacle in the ODB. Note that the intersection checking functions (`SL_InterSegmentObstCheck()`, `SL_InterPolylineObstCheck()`, etc.) test for boundary intersections only. Figure 11(a) shows an example of a segment `s1` with its two endpoints in the interior of a circular obstacle. Assuming the latter was the only obstacle in the ODB, `SL_InterSegmentObstCheck(&s1)` would return 0, since the segment does not intersect the circle's boundary. However, the same call would return 1 in the case illustrated in Figure 11(b).

  Though the case illustrated in Figure 11(a) is not considered an intersection, from a robotics standpoint it should represent a *collision* between the interior of segment $s1$ and the circular obstacle, since the intersection of the interiors of the shapes is nonempty. The function `SL_CollideShapeObst(SL_Shape *shape_ptr)` returns 1 if the intersection of the interior of `*shape_ptr` with any of the obstacles in ODB is nonempty.
- The calls to `SL_SetDrawColor()` and `SL_DrawPolygon()`, which were issued from `SimuRedraw()` to redraw the polygonal barrier, have been eliminated. As long as the ODB is activated, its contents are automatically redrawn by the system at the end of each execution of the step callback.

3.4 Manipulating the ODB's contents

In program `obst.c`, the contents of the obstacle database could be manipulated only through the Obstacle menu. Functions are provided for manipulation of the obstacles from within a program as well. For example, the contents of the ODB can be completely deleted by issuing a call to:

```
int SL_ObstDeleteAll(void);
```

The functions

```
int SL_ObstLoadFile(char *filename);
int SL_ObstMergeFile(char *filename);
```

provide a means for loading the contents of a shapes file into the ODB. `SL_ObstLoadFile()` will replace the current contents of the ODB with the shapes in the designated file (if it is readable); `SL_ObstMergeFile()` will add the shapes from the file to the current contents of the ODB. The function

```
int SL_ObstSaveFile(char *filename);
```

normalsize can be used to store a set of obstacles in a file for later use. (See Section 8.5 for the file format assumed by these three functions.)

The contents of the ODB can be retrieved into an `SL_List` through the call to:

```
SL_List SL_ObstRetrieveShapes(void);
```

Conversely, an existing `SL_List` can be "loaded" as the new contents of the ODB via a call to:

```
void SL_ObstLoadShapes(SL_List shape_list);
```

The shapes in `shape_list` replace the current contents of the ODB.

The functions `SL_ObstRetrieveShapes()` and `SL_ObstLoadShapes()` allow the programmer to use the ODB for simulations in which obstacles are moving, disappearing or reappearing, and/or changing dimensions. One such application would typically require its step callback to be organized in the following fashion:

```
static int SimuStep(void) /* the step callback */
{
   /* ... */
   SL_List ODB = SL_ObstRetrieveShapes(); /* get contents of ODB */
   /* change position and dimension of obstacles, e.g., using
      SL_ListApply(ODB, ... ) */
   SL_LoadShapes(ODB); /* replace old contents of ODB */

   /* test for collision, move robot, etc. */
   return(1);
}
```

Each time the step callback is executed, the ODB's contents are retrieved and stored in a local `SL_List`. This list is then modified (by changing the number, shape, position, etc. of the obstacles) and reloaded into the ODB, replacing the previous contents. In this way, the obstacles are animated along with the robot.

4 Motion Planning Algorithms

4.1 Bug2 and Visbug21

The on-line motion planning algorithms Bug2 and Visbug21 for bug-like (point automaton) robots moving in the unknown planar environment, containing any number of obstacles have been implemented in the library using the abstract data type `SL_Bug`. The functions for this ADT are introduced here and described in more detail in Section 8.16. Bug2 is designed for a robot with only tactile sensing; Visbug21 assumes the robot is equipped with range sensing. In each of these algorithms, the robot (bug) is given a target point it must reach. It moves along a straight line (the Mline) toward the target point until it reaches the point or runs into (for Bug2) or sees (for Visbug21) an obstacle. Upon encountering an obstacle, the robot turns in its *local direction* (either left or right) to move around the obstacle. When it returns to (or sees) the Mline again, it stops moving around the obstacle and moves along the Mline toward the target again. These algorithms are described in detail in [1] and [2], respectively.

A variable of type `SL_Bug` is initialized using one of the following functions:

```
 SL_Bug SL_Bug2Init(SL_Point *start_ptr, SL_Point *target_ptr,
                    int left_loc_dir, double step_size,
                    int (*coll_fn)(SL_Segment *));
 SL_Bug SL_Visbug21Init(SL_Point *start_ptr, SL_Point *target_ptr,
                        int left_loc_dir, double step_size, double vision_rad,
                        int (*coll_fn)(SL_Segment *));
```

as appropriate for the chosen algorithm. The initialization parameters are the robot's starting and target points, a flag indicating if its local direction is left or not, a step size by which to move at each iteration, a radius of vision for the Visbug21 algorithm, and a collision function that determines if the robot has collided with any of the obstacles in the obstacle database (Section 3). By specifying these parameters in the robot's configuration space, these point-robot algorithms may also be used for nonpoint robots.

Once initialized, a given robot's collision function may not be changed. For all other parameters, however, functions are provided that allow them to be changed.

```
   void SL_BugReset(SL_Bug the_bug, SL_Point *new_start_ptr, SL_Point *new_target_ptr);
   void SL_BugSetPos(SL_Bug the_bug, SL_Point *new_pos_ptr);
   void SL_BugSetLocDir(SL_Bug the_bug, int left_loc_dir);
```

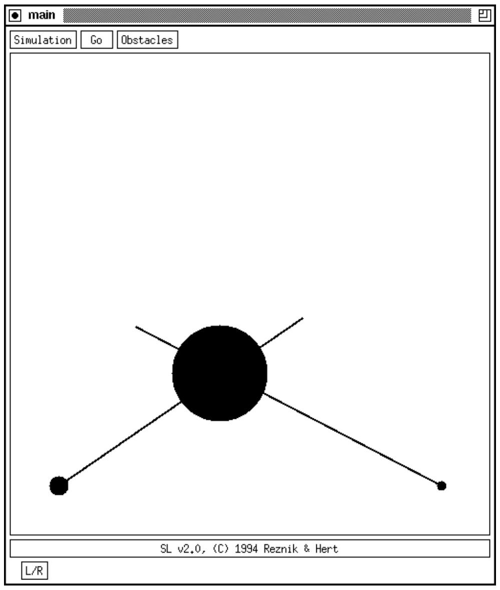


Figure 12: Application window for `bug.c`

```
void SL_BugSetStepSize(SL_Bug the_bug, double step_size);
void SL_BugSetVisionRadius(SL_Bug the_bug,double vision_rad);
```

The current position of the bug may be queried using

```
void SL_BugGetPos(SL_Bug the_bug, SL_Point *pos_ptr);
```

Once a given bug's parameters have been set using the above functions, the function

```
    int SL_BugStep(SL_Bug the_bug);
```

should be used to move the specified bug one step according to the chosen algorithm. This functions returns 1 if the algorithm is to continue or 0 if it should terminate (since the robot has reached its target or determined that it is unreachable).

The function

```
void SL_BugDrawPath(SL_Bug the_bug);
```

provides an easy way to draw the path a given robot has followed from it starting point to its current position.

4.2 A Programming Example — `bug.c`

The following program illustrates the use of the `SL_Bug` ADT in all its glory (well, most of its glory anyway). The application window produced by this program is shown in Figure 12. There are two circular robots in the environment, one (the larger one) using the Bug2 algorithm to plan its motion and the other using Visbug21. The target point for the Bug2 robot can be specified by clicking the left mouse button inside the drawing canvas. The Visbug21 robot's target point can be specified using the middle mouse button. The Mlines for each robot are drawn. Obstacles can be added to the environment using the Obstacle menu (Section 3). For the window shown in Figure 12, a single circular obstacle has been placed in the environment. When the Go button is pressed, the robots will each move along their respective Mlines following their respective algorithms. This motion is accomplished by a single call to `SL_BugStep()` for each robot in the step callback function `SimuStep()`. As the robots move, their paths are drawn using the `SL_BugDrawPath()` function. Choosing the Reset option from the Simulation menu causes the robots to be positioned back at their respective starting points. The toggle button labeled "L/R" in the bottom row switches the local directions of the bugs from left to right and back again.

```
#include <stdio.h>
#include <math.h>
#include "SL.h"

#define CVS_SIDE  500
#define ROBOT_RADIUS 10.0
#define STEP_SIZE 3.0
#define VISION_RADIUS 100.0

/* simulation state variables */
static SL_Bug TactileBug, VisionBug;
static SL_Segment TactileMLine = { {50,50},{100,100} };
static SL_Segment VisionMLine = { {450,50},{350,100} };

/* determines if TactileBug will collide with an obstacle if it moves to the
   end of the given segment */
static int RobotCollision(SL_Segment *trans_ptr)
{
   SL_Circle LocCirc;
   LocCirc.radius = ROBOT_RADIUS;
   LocCirc.center = trans_ptr->end;
   return(SL_InterCircleObstCheck(&LocCirc));
}

/* determines if VisionBug can see an obstacle from the beginning of the
   given segment */
static int VisionCollision(SL_Segment *trans_ptr)
{
   return(SL_InterSegmentObstCheck(trans_ptr));
}

static void SimuInit(void)
{
   SL_MakeColor("obstacles", 0.35, 0.1, 0.1);
   TactileBug = SL_Bug2Init(&(TactileMLine.start), &(TactileMLine.end),
                            0, STEP_SIZE, RobotCollision);
   VisionBug = SL_Visbug21Init(&(VisionMLine.start), &(VisionMLine.end),
                               0, STEP_SIZE, VISION_RADIUS, VisionCollision);
}

static void SimuReset(void)
{
   SL_BugReset(TactileBug, &(TactileMLine.start),&(TactileMLine.end));
   SL_BugReset(VisionBug, &(VisionMLine.start),&(VisionMLine.end));
}

static void SimuRedraw(void)
{
   SL_Circle bug_circle;

   SL_SetDrawColor("blue");
   bug_circle.radius = ROBOT_RADIUS;
   SL_BugGetPos(TactileBug, &(bug_circle.center));
   SL_DrawCircle(&bug_circle, 1);

   bug_circle.radius = ROBOT_RADIUS/2;
   SL_BugGetPos(VisionBug, &(bug_circle.center));
   SL_DrawCircle(&bug_circle, 1);

   SL_SetDrawColor("black");
   SL_BugDrawPath(TactileBug);
   SL_BugDrawPath(VisionBug);

   SL_SetDrawThickness(2);
   SL_SetDrawColor("magenta");
   SL_DrawSegment(&TactileMLine);
   SL_SetDrawColor("red");
   SL_DrawSegment(&VisionMLine);
}

static int SimuStep(void) /* move each bug one step according to their
     respective algorithms */
{
   int tactile_step = SL_BugStep(TactileBug);
   int vision_step = SL_BugStep(VisionBug);
   return(tactile_step||vision_step);
}

static void ClickTarget(int x, int y, SL_Button btn) /* change target point */
{
   if ( !SL_SteppingOn() )
      switch(btn)
      {
         case SL_PRESS_LEFT:
            TactileMLine.end.x = x; TactileMLine.end.y = y;
            SL_BugGetPos(TactileBug, &(TactileMLine.start));
            SL_BugReset(TactileBug, &(TactileMLine.start),&(TactileMLine.end));
            SL_Redraw();
            break;
         case SL_PRESS_MIDDLE:
            VisionMLine.end.x = x; VisionMLine.end.y = y;
            SL_BugGetPos(VisionBug, &(VisionMLine.start));
            SL_BugReset(VisionBug, &(VisionMLine.start),&(VisionMLine.end));
            SL_Redraw();
            break;
   }
}

static void ChangeLocDir(int tog_on) /* change turning direction for bugs */
{
   SL_BugSetLocDir(TactileBug, tog_on);
   SL_BugSetLocDir(VisionBug, tog_on);
}

int main()
{
   SL_Init(CVS_SIDE, CVS_SIDE, 1, SimuInit, SimuReset, SimuRedraw,
   NULL, SimuStep, ClickTarget, NULL, NULL);
   SL_AddToggle("L/R", ChangeLocDir);
   SL_Loop();
   return(0);
}
```

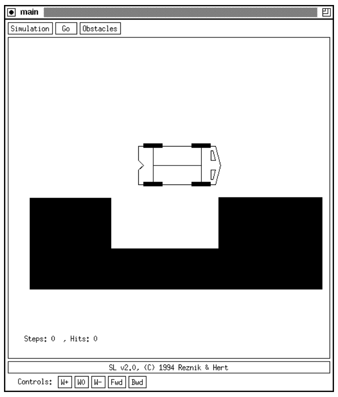


Figure 13: Application window for `car.c`

5 Robot Modeling

5.1 The Current Transformation Matrix and Modeling Transformation Stack

For building and manipulating complicated robot structures, the library provides a set of modeling functions, which are described in Section 8.18. These functions are used to maintain a homogeneous matrix, the *current transformation matrix* (CTM), that encodes a transformed coordinate system with respect to which objects in the scene are specified and drawn. The drawing functions assume objects are specified with respect to the CTM, which is the identity by default. The modeling functions provide a means by which this CTM coordinate system may be translated, rotated, and scaled. Also provided are functions that extract the rotation, translation, or scaling factor from the CTM and functions that transform primitives specified in world coordinates to CTM coordinates, and vice versa.

The *modeling transformation stack* (MTS) is a sequence of CTMs that can be maintained using the functions `SL_ModelPush()` and `SL_ModelPop()`. This is useful for manipulating structures that are naturally hierarchical, as the following program fragment illustrates.

5.2 A Programming Example — `car.c`

The program `car.c`, shown in its entirety in Appendix A, produces a window like the one shown in Figure 13. The car shown in the middle of the screen can be moved using the buttons in the bottom row. The buttons labeled “W+” and “W-” turn the wheels; the “W0” button straightens them. The “Fwd” and “Bwd” buttons move the car forward and backward. The car is a hierarchical structure consisting of a body to which are attached two axes, each with two wheels attached. Extracted below are the functions that are used to draw the car.

```
static ModelAtFrontAxle(Car the_car)
{
   SL_ModelIdentity();
   SL_ModelTranslate(the_car->conf.front_axle.x, the_car->conf.front_axle.y);
   SL_ModelRotateSinCos(-1.0, 0.0);
   SL_ModelRotateSinCos(the_car->conf.axle_sin, the_car->conf.axle_cos);
}

static void DrawHeadLights(void)
{
   static SL_Polygon LeftHL  = {4, {{0,0},{-.2,0},{-.2,.05},{0,.1}}};
   static SL_Polygon RightHL = {4, {{0,0},{.2,0},{.2,.05},{0,.1}}};
   SL_ModelPush();
      SL_ModelTranslate(-0.1, 0.2);
      SL_DrawPolygon(&LeftHL, 0);
      SL_ModelTranslate(0.2, 0.0);
      SL_DrawPolygon(&RightHL, 0);
   SL_ModelPop();
}

static void DrawWheel(double wheel_sin, double wheel_cos)
{
   static SL_Polygon wheel_poly = {4, {{.05,-.2}, {.05,.2},
                                        {-.05,.2}, {-.05,-.2}}};
   SL_ModelPush();
      SL_ModelRotateSinCos(wheel_sin, wheel_cos);
      SL_DrawPolygon(&wheel_poly, 1);
   SL_ModelPop();
}

static void DrawAxle(double wheel_sin, double wheel_cos)
{
   static SL_Segment axle_segm = {{-.4,0},{.4,0}};
   SL_DrawSegment(&axle_segm);
   SL_ModelPush();
      SL_ModelTranslate(-.4, 0.0);
      DrawWheel(wheel_sin, wheel_cos);
      SL_ModelTranslate(.8, 0.0);
      DrawWheel(wheel_sin, wheel_cos);
   SL_ModelPop();
}

void CarDraw(Car the_car)
{
   static SL_Segment CenterAxle = {{0,0},{0,1.0}};
   SL_SetDrawColor(the_car->body_color); /* draw the car body */
   SL_DrawPolygon(&(the_car->body), 1);
   SL_SetDrawColor("black");
   SL_DrawPolygon(&(the_car->body), 0); /* outline it in black */
   SL_ModelPush();
      ModelAtFrontAxle(the_car);
      SL_ModelScale(the_car->length);
      DrawHeadLights();
      DrawAxle(the_car->conf.wheel_sin, the_car->conf.wheel_cos);
      SL_ModelTranslate(0.0, -1.0);
      DrawAxle(0.0, 1.0);
      SL_DrawSegment(&CenterAxle);
   SL_ModelPop();
}
```

The `CarDraw()` function is called from the redraw callback. In this function, first the car body is drawn with respect to the CTM (which is the identity), then this CTM is pushed onto the MTS and the coordinate system is transformed in `ModelAtFronAxle()` so its origin is at the center of the front axle

```
   SL_ModelTranslate(the_car->conf.front_axle.x, the_car->conf.front_axle.y);
```

and its x axis is aligned with the axle in the car's current configuration.

```
   SL_ModelRotateSinCos(-1.0, 0.0);
```

```
SL_ModelRotateSinCos(the_car->conf.axle_sin, the_car->conf.axle_cos);
```

The coordinate system is then scaled to correspond to the length of the car's body:

```
SL_ModelScale(the_car->length);
```

In the function `DrawHeadlights()`, the CTM for the front wheel axle is pushed onto the MTS and then translations are performed to put the coordinate system at the appropriate place for the headlights. When the headlights have been drawn, the call to `SL_ModelPop()` restores the CTM to the front wheel axle coordinate system. Similar manipulations of the CTM and MTS are done in `DrawAxle()` and `DrawWheel()`. Before exiting the `CarDraw()` function, a call to `SL_ModelPop()` is made to restore the CTM to its original state before `CarDraw()` was called. As the structure of the code indicates, each call to `SL_ModelPush()` is generally paired with a call to `SL_ModelPop()`. These calls are used to move up and down the model hierarchy. Using the `SL_Model*()` functions, the programmer is required to travel through the model hierarchy in this way. The `SL_Frame` ADT is a naturally hierarchical data structure that is built using the `SL_Model*()` functions. The functions for the `SL_Frame` ADT, which eliminate the need for manually traversing the hierarchy after it has been built, are described in the following section.

6 Frames

6.1 Frames and Fixtures

Hierarchical geometric structures, such as multi-link arm manipulators, can be built using the `SL_Frame` ADT, as the example in this section illustrates. The functions for this ADT are described in Section 8.19. A *frame* is a transformable coordinate system with which a number of geometric primitives (*fixtures*) are associated. The origin of each frame is considered to be the location of a joint in the geometric structure. The joint may be either fixed, revolute, or prismatic. Each frame that is not a root frame also has a single parent frame associated with it in the hierarchy. All transformations for the child frame are specified with respect to the parent frame. Each frame is created using the function:

```
SL_Frame SL_FrameCreate(SL_Frame the_parent, SL_FrameJoint j_type,
                        int draw_axes);
```

The parameter `j_type` specifies the type of joint to be located at the origin of the new coordinate system (Section 7). The parameter `draw_axes` indicates if the axes of the coordinate systems should be drawn when the geometric structures attached to the frame are drawn.

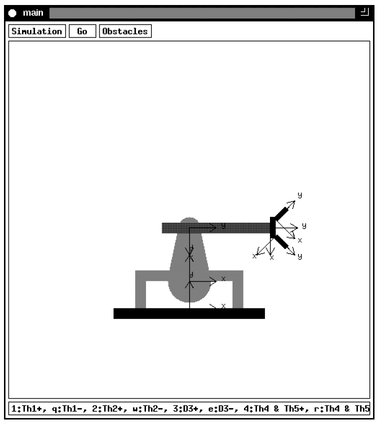
After the call to `SL_FrameCreate()`, the frame is considered to be "open". Transformations (translations, rotations, scalings) with respect to the parent frame can then be done using `SL_FrameTranslate()`, `SL_FrameRotate()`, `SL_FrameRotateSinCos()`, and `SL_FrameScale()`. The location of the frame may also be changed using `SL_FramSet()` and `SL_FramePostMultiply()`. If a frame is created as the root of the hierarchy (by passing NULL as the first argument to `SL_FrameCreate()`), transformations are done with respect to the world coordinate system, the origin of which is in the lower left corner of the drawing canvas.

Fixtures are added to an open frame using the `SL_FrameAdd*()` functions. When all transformations have been performed and all fixtures have been added, a call to `SL_FrameClose()` closes the frame.

Each of the `SL_FrameAdd*()` functions has four parameters

```
void SL_FrameAddCircle(SL_Circle *circ_ptr, int check_collision,
                       char *draw_color, int draw_solid);
void SL_FrameAddPolygon(SL_Polygon *poly_ptr, int check_collision,
                        char *draw_color, int draw_solid);
void SL_FrameAddShape(SL_Shape *shape_ptr, int check_collision,
                      char *draw_color, int draw_solid);
```

The first is a pointer to the geometric primitive that is to be the new fixture. The third and fourth indicate the drawing color and drawing mode (filled or outline) for the primitive. The second is a flag that



```
1:Th1+, q:Th1-, 2:Th2+, w:Th2-, 3:D3+, e:D3-, 4:Th4 & Th5+, r:Th4 & Th5
```

Figure 14: Application window for `arm.c`

indicates if this fixture should be checked for collisions with obstacles or shapes when either of the functions `SL_CollideFrameObst()` or `SL_CollideFrameShape()` is called. These two functions accept a frame that is the root of a hierarchy and check if any of the fixtures attached to the frame hierarchy for which this second argument to the `SL_FrameAdd*()` function was 1 collides with any of the obstacles in the ODB (Sections 3 and 8.15) or with the given shape.

To change the joint values of a specific frame's joint, the functions

```
void  SL_FrameSetJoint(SL_Frame the_frame, double j_value);
void  SL_FrameJointDelta(SL_Frame the_frame, double j_delta);
```

are provided in the library. Because of the hierarchical method of constructing frames, the positions of the frames descendant from `the_frame` are automatically updated to reflect `the_frame`'s joint value change.

To draw the fixtures in a given frame hierarchy, a single call to

```
void SL_FrameDraw(SL_Frame the_frame);
```

is made with the hierarchy's root frame as the argument. All fixtures in the hierarchy are drawn in the color and mode specified when they were added to their respective frames.

Functions are also provided in the library for querying a given frame for its joint value, or its translation vector, rotation angle, or scaling factor with respect to its parent frame or the world coordinate system and for transforming points specified in one frame into another frame. See Section 8.19 for more details.

6.2 A Programming Example – `arm.c`

The application window produced by the following program is shown in Figure 14. The arm shown has two fixed joints (at its base and the base of the gripper), one prismatic joint (at the base of the long rectangle) and four revolute joints. The nonfixed joints are numbered from 1 to 5 and their values may be changed using the keys indicated in the status bar message. Obstacles may be added to the environment using the Obstacle menu. When the Go button is pressed, all the nonfixed joints of the arm will begin to move. If the arm collides with an obstacle, it will rebound off the obstacle and begin to move in the opposite direction. This continues until the Go button is pressed again. The Reset option of the Simulation menu puts the arm back in its starting position (where all joint values are 0).

```
#include <stdio.h>
#include <math.h>
#include <string.h>
#include "SL.h"
```

```
#define CVS_SIDE  500

/* simulation state variables */
static SL_Frame Base, Th1, Th2, D3, Hand, Th4, Th5;

static double arg;
static double arg_step;

static void MakeArm(void)
{
   static SL_Polygon FloorPoly = {4, {{70,-10},{70,0},{-70,0},{-70,-10}}};
   static SL_Polygon BasePoly = {8, {{40,0},{50,0},{50,35},{-50,35},
                                     {-50,0},{-40,0},{-40,25},{40,25}}};
   static SL_Polygon L1Poly = {4, {{19.6,4},{9.8,52},{-9.8,52},{-19.6,4}}};
   static SL_Circle L1Base = {20, {0,0}};
   static SL_Circle L1Tip = {10, {0,0}};
   static SL_Polygon L2Poly = {4, {{5,-25},{5,75},{-5,75},{-5,-25}}};
   static SL_Polygon PalmPoly = {4, {{10,0},{10,5},{-10,5},{-10,0}}};
   static SL_Polygon FingerPoly = {4, {{2.5,0},{2.5,15},{-2.5,15},{-2.5,0}}};

   Base = SL_FrameCreate(NULL, SL_FIXED, 1);
      SL_FrameTranslate(CVS_SIDE*0.5, CVS_SIDE*0.25);
      SL_FrameScale(1.5);
      SL_FrameAddPolygon(&FloorPoly, 0, "black", 1);
      SL_FrameAddPolygon(&BasePoly, 0, "magenta", 1);
   SL_FrameClose();

   Th1 = SL_FrameCreate(Base, SL_REVOL_CW, 1);
      SL_FrameTranslate(0.0, 25.0);
      SL_FrameAddCircle(&L1Base, 1, "grey", 1);
      SL_FrameAddPolygon(&L1Poly, 1, "grey", 1);
   SL_FrameClose();

   Th2 = SL_FrameCreate(Th1, SL_REVOL_CCW, 1);
      SL_FrameTranslate(0.0, 50.0);
      SL_FrameRotate(-SL_PI_OV_2);
      SL_FrameAddCircle(&L1Tip, 1, "grey", 1);
   SL_FrameClose();

   D3 = SL_FrameCreate(Th2, SL_PRISM_Y, 1);
      SL_FrameAddPolygon(&L2Poly, 1, "red", 1);
   SL_FrameClose();

   Hand = SL_FrameCreate(D3, SL_FIXED, 1);
      SL_FrameTranslate(0.0, 75.0);
      SL_FrameAddPolygon(&PalmPoly, 1, "blue", 1);
   SL_FrameClose();

   Th4 = SL_FrameCreate(Hand, SL_REVOL_CW, 1);
      SL_FrameTranslate(-7.5, 5.0);
      SL_FrameRotate(SL_PI*0.25);
      SL_FrameAddPolygon(&FingerPoly, 1, "blue", 1);
   SL_FrameClose();

   Th5 = SL_FrameCreate(Hand, SL_REVOL_CCW, 1);
      SL_FrameTranslate(7.5, 5.0);
      SL_FrameRotate(-SL_PI*0.25);
      SL_FrameAddPolygon(&FingerPoly, 1, "blue", 1);
   SL_FrameClose();
}

static void SimuInit(void)
{
   SL_MakeColor("obstacles", 0.35, 0.1, 0.1);
   MakeArm();
   SL_PrintToStatusBar("1:Th1+, q:Th1-, 2:Th2+, w:Th2-, 3:D3+, e:D3-, 4:Th4 & Th5+, r:Th4 & Th5-");
}

static void SimuReset(void)
{
   arg = 0;
   arg_step = SL_PI*0.01;
   SL_FrameSetJoint(Th1, 0.0);
   SL_FrameSetJoint(Th2, 0.0);
   SL_FrameSetJoint(D3, 0.0);
   SL_FrameSetJoint(Th4, 0.0);
   SL_FrameSetJoint(Th5, 0.0);
}

static void SimuRedraw(void)
{
   SL_FrameDraw(Base);
}

static int SimuStep(void)
{
   arg+=arg_step;
   SL_FrameSetJoint(Th1, sin(arg));
   SL_FrameSetJoint(Th2, sin(2.0*arg));
   SL_FrameSetJoint(D3, 15.0*sin(arg));
   SL_FrameSetJoint(Th4, sin(arg));
   SL_FrameSetJoint(Th5, sin(arg));
   if (SL_CollideFrameObst(Base))
   {
      arg_step*=-1.0;
      arg+=arg_step;
      SL_FrameSetJoint(Th1, sin(arg));
      SL_FrameSetJoint(Th2, sin(2.0*arg));
      SL_FrameSetJoint(D3, 15.0*sin(arg));
      SL_FrameSetJoint(Th4, sin(arg));
      SL_FrameSetJoint(Th5, sin(arg));
   }
   return(1);
}

static void KbdCbk(char the_key, int key_pressed)
{
   if (key_pressed)
   {
      switch(the_key)
      {
         case '1': SL_FrameJointDelta(Th1, SL_PI*0.05);
```

```
                  break;
         case 'q': SL_FrameJointDelta(Th1, -SL_PI*0.05);
                   break;
         case '2': SL_FrameJointDelta(Th2, SL_PI*0.05);
                   break;
         case 'w': SL_FrameJointDelta(Th2, -SL_PI*0.05);
                   break;
         case '3': SL_FrameJointDelta(D3, 5.0);
                   break;
         case 'e': SL_FrameJointDelta(D3, -5.0);
                   break;
         case '4': SL_FrameJointDelta(Th4, SL_PI*0.05);
                   SL_FrameJointDelta(Th5, SL_PI*0.05);
                   break;
         case 'r': SL_FrameJointDelta(Th4, -SL_PI*0.05);
                   SL_FrameJointDelta(Th5, -SL_PI*0.05);
                   break;
      }
      SL_Redraw();
   }
}

int main()
{
   SL_Init(CVS_SIDE, CVS_SIDE, 1, SimuInit, SimuReset, SimuRedraw,
           NULL, SimuStep, NULL, NULL, KbdCbk);
   SL_Loop();
   return(0);
}
```

7 The Structures, Constants, and Macros in the Library

The header file, SL.h, contains the definitions of all structures, constants, and macros used in the library. These are described below.

The following constants are defined in SL.h.

```
/* Some utility constants not specific to the simulation library */
      SL_PI                /* π */
      SL_TWO_PI            /* 2π */
      SL_PI_OV_2           /* π/2 */
      SL_PI_OV_4           /* π/4 */
      SL_1_OV_PI           /* 1/π */
      SL_180_OV_PI         /* 180/π */
      SL_PI_OV_180         /* π/180 */
      SL_E                 /* e, the natural log base */

      SL_SQRT2             /* √2 */
      SL_SQRT3             /* √3 */
      SL_SMALL             /* 1.0e-12, a floating point number close to 0 */
      SL_LARGE             /* 1.0e+12, a large floating point number */

/* Constant used to define the SL_Polygon and SL_Polyline structures */

      SL_MAX_VTX           /* maximum number of vertices for a polygon (polyline) */
```

```
/* The constant values that are used to represent to the mouse click
   callback function which mouse button was either pressed or released.  */

   typedef enum {
      SL_PRESS_LEFT,        /* left mouse button pressed */
      SL_PRESS_MIDDLE,      /* middle mouse button pressed */
      SL_PRESS_RIGHT,       /* right mouse button pressed */
      SL_RELEASE_LEFT,      /* left mouse button released */
      SL_RELEASE_MIDDLE,    /* middle mouse button released */
      SL_RELEASE_RIGHT      /* right mouse button released */
   } SL_Button;

/* The constant values used to indicate the type of shape stored in
   an SL_Shape structure */

   typedef enum {
      SL_POINT,
      SL_VECTOR,
      SL_SEGMENT,
      SL_CIRCLE,
      SL_ARC,
      SL_POLYLINE,
      SL_POLYGON
   } SL_ShapeType;

/* The constant values used to indicate the joint types for the SL_Frame ADT */

   typedef enum {
      SL_FIXED,             /* a fixed joint (i.e., it does not move) */
      SL_REVOL_CW,          /* a clockwise revolute joint */
      SL_REVOL_CCW,         /* a counterclockwise revolue joint */
      SL_PRISM_X,           /* a prisimatic joint moving in the x direction */
      SL_PRISM_Y            /* a prismatic joint moving in the y direction */
   } SL_FrameJoint;
```

Also defined are the following handy macros.

```
      SL_SIGNUM(a)          /* 1, 0, or -1 depending on if a is >, =, or < 0,
                               respectively */
      SL_MAX(a,b)           /* computes the maximum of a and b */
      SL_MIN(a,b)           /* computes the minimum of a and b */
      SL_ABS(a)             /* computes the absolute value of a */
      SL_ODD(a)             /* determines if a is odd or not */
      SL_SQR(a)             /* computes a^2 */
      SL_SWAP(a,b,type)     /* given the type (e.g., int) of the two values a
                               and b will interchange their values */
      SL_Negligible(val)    /* determines if val is sufficiently close to 0
                               to be ignored */
      SL_BoundVal(val,low,hi) /* returns the value in the range [low, hi]
                                 closest to val */
      SL_InClosedInterval(t,low,hi) /* determines if t ∈ [low, hi] */
      SL_InOpenInterval(t,low,hi)   /* determines if t ∈ (low, hi) */
```

The primitive structures of the library are shown below.

```
typedef struct {
   double         x;         /* the x coordinate of the point */
   double         y;         /* the y coordinate of the point */
} SL_Point, SL_Vector;


typedef struct {
   SL_Point       start;     /* the starting point of the segment */
   SL_Point       end;       /* the ending point of the segment */
} SL_Segment;


typedef struct {
   SL_Point       center;    /* the center of the arc */
   double         radius;    /* the radius of the arc */
   double         start_ang; /* CCW radian measurement for starting arc angle */
   double         end_ang;   /* CCW radian measurement for ending arc angle */
} SL_Arc;


typedef struct {
   SL_Point       center;    /* the center of the circle */
   double         radius;    /* the radius of the circle */
} SL_Circle;


typedef struct {
   int            size;      /* the number of vertices in the polygon */
   SL_Point       vertex[SL_MAX_VTX];
                             /* counterclockwise list of vertices with
                                the first vertex not repeated */
} SL_Polygon, SL_Polyline;
```

In addition, a more general structure, the `SL_Shape`, is provided.

```
typedef struct {
   SL_ShapeType   type;      /* indicates which primitive structure this shape is */
   union
   {
      SL_Point    the_point;
      SL_Vector   the_vector;
      SL_Segment  the_segment;
      SL_Circle   the_circle;
      SL_Arc      the_arc;
      SL_Polyline the_polyline;
      SL_Polygon  the_polygon;
   } data;
} SL_Shape;
```

The following abstract data types (ADTs) are also defined:

```
   SL_Bug;
   SL_List;
   SL_Transf;
   SL_Frame;
```

The routines for manipulating objects of type `SL_Bug` are described in Section 8.16; routines for the `SL_Transf` ADT are described in Section 8.17; `SL_Frame` functions are described in Section 8.19; routines for the `SL_List` ADT and described in Section 8.20.

8 A Browse Through the Library

8.1 Utility Functions

The following functions are not specific to any of the library structures.

```
double SL_Modulo(
   double      value              /* a real number to compute modulus of */
   double      modulo             /* the modulo, a positive real number */
)
```

`SL_Modulo()` accepts two real numbers and returns the value of the first number modulo the second. If the second number is negative, the function returns 0.

```
double SL_DegreesToRadians(
   double      degrees            /* the degree measurement */
)
```

`SL_DegreesToRadians()` accepts a number representing a certain degree angle measurement and returns the equivalent angle value in radians.

```
double SL_RadiansToDegrees(
   double      radians            /* the radian angle measurement */
)
```

`SL_RadiansToDegrees()` accepts a number representing a certain radian angle measurement and returns the equivalent angle value in degrees.

```
double SL_RandomNumber(
   double      min,               /* lower bound of range for generated number */
   double      max                /* upper bound of range for generated number */
)
```

`SL_RandomNumber()` returns a random number in the range [`min`,`max`].

```
int SL_QuadraticRoots(
   double      a,                 /* the coefficient of the quadratic term */
   double      b,                 /* the coefficient of the linear term */
   double      c,                 /* the constant term */
   double      *root1_ptr,        /* pointer to first root of equation */
   double      *root2_ptr         /* pointer to second root of equation */
)
```

`SL_QuadraticRoots()` computes the real roots of the quadratic equation $ax^2 + bx + c = 0$. The function returns the number of real roots found. If the equation is of the form $c = 0$, the function returns 0 indicating that there are either no roots or an infinite number of roots. If the roots are imaginary, the function also returns 0. In both of these cases, `*root1` and `*root2` will remain unchanged. If the equation is linear, the single root will be stored in `*root1`, and `*root2` will remain unchanged.

```
void SL_MatrixTranspose2x2(
   double      the_matrix[2][2]   /* matrix to transpose */
   double      transp_mat[2][2])  /* the transposed matrix */
)


void SL_MatrixTranspose3x3(
   double      the_matrix[3][3]   /* matrix to transpose */
   double      transp_mat[3][3])  /* the transposed matrix */
)
```

These two functions compute transposes of given 2×2 or 3×3 matrices.

```
double SL_MatrixDeterminant2x2(
   double      the_matrix[2][2]    /* matrix to compute determinant of */
)


double SL_MatrixDeterminant3x3(
   double      the_matrix[3][3]    /* matrix ot compute determinant of */
)
```

These two functions compute determinants of given 2×2 or 3×3 matrices.

```
int SL_MatrixInvert2x2(
   double      orig_matrix[2][2],  /* the matrix to invert */
   double      inv_matrix[2][2]    /* the inverted matrix */
)


int SL_MatrixInvert3x3(
   double      orig_matrix[3][3],  /* the matrix to invert */
   double      inv_matrix[3][3]    /* the inverted matrix */
)
```

These two functions invert given 2×2 or 3×3 matrices. If the determinant of the original matrix is 0, the function returns 0 leaving `inv_matrix` unchanged. Otherwise, the function returns 1.

```
void SL_MatrixProduct2x2(
   double      m1[2][2],           /* the first matrix */
   double      m2[2][2]            /* the second matrix */
   double      m1xm2[2][2]         /* the product of m1 and m2 */
)


void SL_MatrixProduct3x3(
   double      m1[3][3],           /* the first matrix */
   double      m2[3][3]            /* the second matrix */
   double      m1xm2[3][3]         /* the product of m1 and m2 */
)
```

These two functions compute the matrix product $m_1 m_2$ for two given 2×2 or 3×3 matrices.

```
void SL_MatrixTimesVector2x2(
   double      the_mat[2][2],      /* the matrix */
   double      the_vec[2]          /* the vector to multiply */
   double      res_vec[2]          /* the product the_mat * the_vec */
)


void SL_MatrixTimesVector3x3(
   double      the_mat[3][3],      /* the matrix */
   double      the_vec[3]          /* the vector to multiply */
   double      res_vec[3]          /* the product the_mat * the_vec */
)
```

Each of these two functions computes the product of a given 2×2 or 3×3 matrix `the_mat` and a given column vector `the_vec`. The resulting vector is stored in `res_vec`.

```
int SL_MatrixSolve2x2(
   double      coeff[2][2],       /* row-major matrix of linear coefficients */
   double      rhs[2],            /* column vector of right-hand-side values */
   double      result[2]          /* column vector of solutions */
)
```

`SL_MatrixSolve2x2()` solves a linear system of two equations with two unknowns given in the form:

$$\begin{bmatrix} coeff_{0,0} & coeff_{0,1} \\ coeff_{1,0} & coeff_{1,1} \end{bmatrix} \begin{bmatrix} result_0 \\ result_1 \end{bmatrix} = \begin{bmatrix} rhs_0 \\ rhs_1 \end{bmatrix}$$

If this linear system does not have a solution, the function returns 0 and leaves the array `result` unchanged. Otherwise, the function returns 1.

```
int SL_MatrixSolve3x3(
   double      coeff[3][3],       /* row-major matrix of linear coefficients */
   double      rhs[3],            /* column vector of right-hand-side values */
   double      result[3]          /* column vector of solutions */
)
```

`SL_MatrixSolve3x3()` solves a linear system of three equations with three unknowns given in the form:

$$\begin{bmatrix} coeff_{0,0} & coeff_{0,1} & coeff_{0,2} \\ coeff_{1,0} & coeff_{1,1} & coeff_{1,2} \\ coeff_{2,0} & coeff_{2,1} & coeff_{2,2} \end{bmatrix} \begin{bmatrix} result_0 \\ result_1 \\ result_2 \end{bmatrix} = \begin{bmatrix} rhs_0 \\ rhs_1 \\ rhs_2 \end{bmatrix}$$

If this linear system does not have a solution, the function returns 0 and leaves the array `result` unchanged. Otherwise, the function returns 1.

8.2 Point and Vector Functions

```
double SL_VectorDotProduct(
   SL_Vector   *vec1_ptr,         /* pointer to first vector */
   SL_Vector   *vec2_ptr          /* pointer to second vector */
)
```

`SL_VectorDotProduct()` returns the dot product (scalar product) of two vectors based at the origin.

```
double SL_VectorCrossProduct(
   SL_Vector   *vec1_ptr,         /* pointer to first vector */
   SL_Vector   *vec2_ptr          /* pointer to second vector */
)
```

`SL_VectorCrossProduct` computes the signed magnitude of the z component of `*vec1_ptr` $\times$ `*vec2_ptr`. Since the vectors are two-dimensional, the x and y components of the cross product are trivially zero.

```
double SL_VectorLength(
   SL_Vector   *vec_ptr           /* pointer to vector of which to compute length */
)
```

`SL_VectorLength()` returns the length (magnitude) of the given vector.

```
double SL_VectorLengthSQR(
   SL_Vector   *vec_ptr           /* pointer to vector of which to compute length */
)
```

`SL_VectorLengthSQR()` returns the square of the length of the given vector, thus avoiding the expensive square-root operation.

```
int SL_VectorNull(
   SL_Vector   *vec_ptr           /* pointer to vector in question */
)
```

`SL_VectorNull()` returns 1 if the input vector is of length 0 and returns 0 otherwise.

```
int SL_VectorNormalize(
   SL_Vector   *vec_ptr,          /* pointer to vector to normalize */
   SL_Vector   *unit_vec_ptr      /* pointer to resulting unit length vector */
)
```

`SL_VectorNormalize()` converts a given vector based at the origin into a unit length vector in the same direction. If the length of `*vec_ptr` is 0, the function returns 0, leaving `*unit_vec_ptr` unchanged. Otherwise, the function returns 1 and stores the resulting normalized vector in `*unit_vec_ptr`.

```
double SL_VectorPolarAngle(
   SL_Vector   *vec_ptr           /* pointer to vector of which to compute angle */
)
```

`SL_VectorPolarAngle()` computes the counterclockwise angle the given vector makes with the horizontal. The value returned will be this radian angle measurement in the range $[-\pi, \pi]$. If the vector is of length 0, the function returns 0.

```
int SL_VectorPolarAttributes(
   SL_Vector   *vec_ptr,          /* pointer to the vector in question */
   double      *sin_ptr,          /* pointer to sine of polar angle */
   double      *cos_ptr,          /* pointer to cosine of polar angle */
   double      *length_ptr        /* pointer to length of vector */
)
```

`SL_VectorPolarAttributes()` computes the sine and cosine of the counterclockwise angle the given vector makes with the horizontal and stores these values in `*sin_ptr` and `*cos_ptr`, respectively. It also computes the length of the given vector and stores the result in `*length_ptr`. The function returns 0 if the length of the vector is 0, leaving `*sin_ptr` and `*cos_ptr` unchanged (`*length_ptr` will have the value 0); otherwise, the function returns 1.

```
double SL_VectorToVectorAngle(
   SL_Vector   *vec1_ptr,         /* pointer to vector from which to compute angle */
   SL_Vector   *vec2_ptr          /* pointer to vector to which to compute angle */
)
```

`SL_VectorToVectorAngle()` computes the counterclockwise angle from `*vec1_ptr` to `*vec2_ptr`. The radian angle measurement returned by the function will be in the range $[-\pi, \pi]$. If either of the vectors is of length 0, the function returns 0.

```
void SL_VectorRotate(
   SL_Vector   *vec_ptr,          /* pointer to vector to be rotated */
   double      angle,             /* number of radians by which to rotate vector about
                                     the origin in the counterclockwise direction */
   SL_Vector   *rot_vec_ptr,      /* pointer to vector that results from rotation */
)
```

`SL_VectorRotate()` rotates a vector about the origin by a specified number of radians producing a new vector, `*rot_vec_ptr`. The rotation is done in the counterclockwise direction.

```
void SL_VectorRotateSinCos(
   SL_Vector   *vec_ptr,          /* pointer to vector to be rotated */
   double      the_sin,           /* sine of the angle by which to rotate */
   double      the_cos,           /* cosine of the angle by which to rotate */
   SL_Vector   *rot_vec_ptr       /* pointer to vector that results from rotation */
)
```

`SL_VectorRotateSinCos()` rotates a vector about the origin by a certain angle, the sine and cosine of which are provided. (This avoids the rather expensive calls to the functions to compute the sine and cosine.) The rotation is done in the counterclockwise direction and the resulting vector is stored in `*rot_vec_ptr`.

```
void SL_VectorScale(
   SL_Vector   *vec_ptr,          /* pointer to vector to be scaled */
   double      scale,             /* the amount by which to scale the vector */
   SL_Vector   *scaled_vec_ptr    /* pointer to vector that results from scaling */
)
```

`SL_VectorScale()` multiplies a given vector by a scaling factor storing the vector that results in `*scaled_vec_ptr`.

```
int SL_VectorProjection(
   SL_Vector   *vec1_ptr,         /* pointer to vector to be projected */
   SL_Vector   *vec2_ptr,         /* pointer to vector onto which to project */
   SL_Vector   *proj_vec_ptr      /* pointer to vector that results when *vec1_ptr
                                     is projected onto *vec2_ptr */
)
```

`SL_VectorProjection()` projects one vector (`*vec1_ptr`) onto another (`*vec2_ptr`) producing a third vector (`*proj_vec_ptr`). If the vector onto which the projection is to be done is of length 0, the function returns 0 without changing `*proj_vec_ptr`; otherwise, the function returns 1.

```
void SL_VectorMidpoint(
   SL_Vector   *vec_ptr,          /* pointer to vector of which to find midpoint */
   SL_Point    *mid_ptr           /* pointer to location in which to store midpoint */
)
```

`SL_VectorMidpoint()` computes the coordinates of the point in the middle of a given vector.

```
void SL_VectorNormal(
   SL_Vector   *vec_ptr,          /* pointer to vector to which to find normal */
   SL_Vector   *norm_ptr          /* pointer to location in which to store normal */
)
```

`SL_VectorNormal()` computes a normal to a given vector. The normal calculated is the one that is 90 degrees in the counterclockwise direction from the original.

```
void SL_VectorAdd(
   SL_Vector   *vec1_ptr,         /* pointer to vector to be added */
   SL_Vector   *vec2_ptr,         /* pointer to vector to be added */
   SL_Vector   *v1_plus_v2_ptr    /* pointer to resulting vector */
)
```

`SL_VectorAdd()` adds two given vectors producing a new vector.

```
void SL_VectorSubtract(
   SL_Vector   *vec1_ptr,         /* pointer to vector to subtract *vec2_ptr from */
   SL_Vector   *vec2_ptr,         /* pointer to vector to subtract from *vec1_ptr */
   SL_Vector   *v1_minus_v2_ptr   /* pointer to resulting vector */
)
```

`SL_VectorSubtract()` subtracts `*vec2_ptr` from `*vec1_ptr` storing the result in `*v1_minus_v2_ptr`.

```
int SL_MoveTowardsPoint(
   SL_Point    *current_ptr,      /* pointer to current point */
   SL_Point    *target_ptr,       /* pointer to target point */
   double      distance,          /* the distance to move */
   SL_Point    *new_pos_ptr       /* pointer to new location resulting from moving
                                     the given distance */
)
```

`SL_MoveTowardsPoint()` computes the new point that results from moving from `*current_ptr` a certain distance along the directed line defined by `*current_ptr` and `*target_ptr`. This distance has no bounds (*i.e*, it may be greater than the distance between `*current_ptr` and `*target_ptr` and may be negative). The resulting point is stored in `*new_pos_ptr`. If the two points that define the line segment coincide, the function returns 0 leaving `*new_pos_ptr` unchanged; otherwise the function returns 1.

```
int SL_PointInRightSemiPlane(
   SL_Segment  *seg_ptr,          /* pointer to segment in question */
   SL_Point    *pnt_ptr           /* pointer to point in question */
)
```

`SL_PointInRightSemiPlane()` determines if the given point is in the right semiplane of the line defined by the given segment. If the point lies to the right of the defined line, the function returns 1; if the point lies to the left of the defined line, the function returns 0; if the length of the segment is 0, the function returns -1.

```
void SL_PointInterpolate(
   SL_Point    *start_ptr,        /* pointer to starting interpolation point */
   SL_Point    *end_ptr,          /* pointer to ending interpolation point */
   double      t,                 /* the interpolation parameter */
   SL_Point    *result_ptr        /* pointer to resulting interpolated point */
)
```

`SL_PointInterpolate()` performs a linear interpolation between two points using a given parameter value `t`. For `t` $\in [0, 1]$ the interpolated point (`*result_ptr`) will fall linearly in the interval [`*start_ptr`,`*end_ptr`].

```
int SL_LeftTurn(
   SL_Point    *pnt1_ptr          /* pointer to the first point */
   SL_Point    *pnt2_ptr          /* pointer to the second point */
   SL_Point    *pnt3_ptr          /* pointer to the third point */
)
```

`SL_LeftTurn()` determines if a left turn is made when moving from `*pnt1_ptr` through `*pnt2_ptr` to `*pnt3_ptr` (*i.e.*, if the movement is counterclockwise with respect to `*pnt1_ptr`) returning 1 if this is true and 0 otherwise.

```
int SL_RightTurn(
   SL_Point    *pnt1_ptr          /* pointer to the first point */
   SL_Point    *pnt2_ptr          /* pointer to the second point */
   SL_Point    *pnt3_ptr          /* pointer to the third point */
)
```

`SL_RightTurn()` determines if a right turn is made when moving from `*pnt1_ptr` through `*pnt2_ptr` to `*pnt3_ptr` (*i.e.*, if the movement is clockwise with respect to `*pnt1_ptr`) returning 1 if this is true and 0 otherwise.

```
int SL_Collinear(
   SL_Point    *pnt1_ptr          /* pointer to the first point */
   SL_Point    *pnt2_ptr          /* pointer to the second point */
   SL_Point    *pnt3_ptr          /* pointer to the third point */
)
```

`SL_Collinear()` determines if the three given points are all on the same line and returns 1 if this is true and 0 otherwise.

8.3 Segment Functions

Many of the functions listed in this section are completely analogous to those listed in Section 8.2 for vectors. These functions should be used when dealing with directed segments, which can also be thought of as vectors not based at the origin.

```
double SL_SegmentDotProduct(
   SL_Segment  *seg1_ptr,         /* pointer to first segment */
   SL_Segment  *seg2_ptr          /* pointer to second segment */
)
```

`SL_SegmentDotProduct()` returns the dot product (scalar product) of two segments after each has been translated to the origin. The two segments remain unchanged.

```
double SL_SegmentLength(
   SL_Segment  *seg_ptr           /* pointer to segment of which to compute length */
)
```

`SL_SegmentLength()` returns the length (magnitude) of the given segment.

```
double SL_SegmentLengthSQR(
   SL_Segment  *seg_ptr           /* pointer to segment of which to compute length */
)
```

`SL_SegmentLengthSQR()` returns the squared length of the give segment (thus avoiding the rather expensive square-root operation).

```
int SL_SegmentNull(
   SL_Segment  *seg_ptr           /* pointer to segment in question */
)
```

`SL_SegmentNull()` returns 1 if the given segment is of length 0 and returns 0 otherwise.

```
double SL_SegmentPolarAngle(
   SL_Segment  *seg_ptr           /* pointer to segment of which to compute angle */
)
```

`SL_SegmentPolarAngle()` computes the counterclockwise angle the given segment makes with the horizontal. The value returned will be in radians in the range $[-\pi, \pi]$. If the segment is of length 0, the function returns 0.

```
int SL_SegmentPolarAttributes(
   SL_Segment  *seg_ptr,          /* pointer to the segment in question */
   double      *sin_ptr,          /* pointer to sine of polar angle */
   double      *cos_ptr,          /* pointer to cosine of polar angle */
   double      *length_ptr        /* pointer to length of segment */
)
```

`SL_SegmentPolarAttributes()` computes the sine and cosine of the counterclockwise angle the given segment makes with the horizontal and stores these values in `*sin_ptr` and `*cos_ptr`, respectively. It also computes the length of the given segment and stores the result in `*length_ptr`. The function returns 0 if the length of the segment is 0, leaving `*sin_ptr` and `*cos_ptr` unchanged (`*length_ptr` will have the value 0); otherwise, the function returns 1.

```
double SL_SegmentToSegmentAngle(
   SL_Segment  *seg1_ptr,         /* pointer to segment from which to compute angle */
   SL_Segment  *seg2_ptr          /* pointer to segment to which to compute angle */
)
```

`SL_SegmentToSegmentAngle()` computes the counterclockwise angle from `*seg1_ptr` to `*seg2_ptr`. The radian angle measurement returned by the function will be in the range $[-\pi, \pi]$. If either of the segments is of length 0, the function returns 0.

```
void SL_SegmentRotate(
   SL_Segment  *seg_ptr,          /* pointer to segment to be rotated */
   double      angle              /* number of radians by which to rotate segment
                                     about starting point in the CCW direction */
   SL_Segment  *rot_seg_ptr,      /* pointer to rotated segment */
)
```

`SL_SegmentRotate()` rotates a segment about its starting point by a specified number of radians producing a new segment, `*rot_seg_ptr`. The rotation is done in the counterclockwise direction.

```
void SL_SegmentRotateSinCos(
   SL_Segment  *seg_ptr,          /* pointer to segment to be rotated */
   double      the_sin,           /* the sine of the angle by which to rotate */
   double      the_cos,           /* the cosine of the angle by which to rotate */
   SL_Segment  *rot_seg_ptr       /* pointer to rotated segment */
)
```

`SL_SegmentRotateSinCos()` rotates a segment about its starting point by a certain angle, the sine and cosine of which are provided. (This avoids the rather expensive calls to the functions to compute the sine and cosine.) The rotation is done in the counterclockwise direction and the resulting segment is stored in `*rot_seg_ptr`.

```
void SL_SegmentMove(
   SL_Segment  *seg_ptr,          /* pointer to segment to be translated */
   SL_Point    *new_start_ptr     /* pointer to new starting point for segment */
   SL_Segment  *trans_seg_ptr,    /* pointer to translated segment */
)
```

`SL_SegmentMove()` moves a segment, maintaining its length and orientation, so its starting point is at the point `*new_start_ptr`. The translated segment is returned in the parameter `*trans_seg_ptr`.

```
void SL_SegmentScale(
   SL_Segment  *seg_ptr,          /* pointer to segment to be scaled */
   double      scale,             /* the amount by which to scale the segment */
   SL_Segment  *scaled_seg_ptr    /* pointer to scaled segment */
)
```

`SL_SegmentScale()` resizes a given segment using a given scaling factor storing the resulting segment in `*scaled_seg_ptr`. The scaled segment has the same starting point as the original segment and an adjusted ending point. The scaling factor represents a fraction of the segment length by which to adjust the ending point. For example, if the segment has starting point $(5, 5)$ and ending point $(7, 8)$ and the scaling factor is 3.5, the scaled segment will have starting point $(5, 5)$ and ending point $(12.0, 15.5)$.

```
int SL_SegmentProjection(
   SL_Point    *pnt_ptr,          /* pointer to point to be projected */
   SL_Segment  *seg_ptr,          /* pointer to segment onto which to project */
   SL_Point    *proj_pnt_ptr      /* pointer to projected point */
)
```

`SL_SegmentProjection()` projects a point onto a directed segment producing another point that is on the line defined by the segment. The projected point will not necessarily be between the endpoints of the segment. If the length of the segment is 0, the function returns 0; otherwise, the function returns 1.

```
void SL_SegmentMidpoint(
   SL_Segment  *seg_ptr,          /* pointer to segment of which to find midpoint */
   SL_Point    *mid_ptr           /* pointer to location in which to store midpoint */
)
```

`SL_SegmentMidpoint()` computes the location of the point in the middle of the given segment.

8.4 Polygon and Polyline Functions

Note that all the `SL_Polygon*()` functions described in this section work equally well for variables declared as `SL_Polyline` structures. The one case where the computation differs for the `SL_Polygon` and `SL_Polyline` structures is in deciding if there is a self-intersection. In this case, there is a separate function provided specifically for the `SL_Polyline` structure.

```
int SL_PolygonConvexHull(
   SL_Polygon  *pgon_ptr,         /* pointer to the polygon in question */
   SL_Polygon  *hull_ptr          /* pointer to location for convex hull */
)
```

`SL_PolygonConvexHull()` computes the convex hull of a given polygon and returns the list of vertices of the convex hull in the parameter `*hull_ptr`. The convex hull vertices are listed in counterclockwise order, which may not be the same order as they appear in the original polygon, `*pgon_ptr`. If the `*pgon_ptr` has less than three vertices, the function returns 0, leaving `*hull_ptr` unchanged. Otherwise, the function returns 1.

```
int SL_PolygonInCCWOrder(
   SL_Polygon  *pgon_ptr          /* pointer to the polygon in question */
)
```

`SL_PolygonInCCWOrder()` determines if the vertices of the given polygon are listed in counterclockwise order. If so, the function returns 1. If the vertices are in clockwise order, the function returns 0.

```
void SL_PolygonCentroid(
   SL_Polygon  *pgon_ptr,         /* pointer to polygon in question */
   SL_Point    *centroid_ptr      /* pointer to location for average vertex */
)
```

`SL_PolygonCentroid()` computes the average position of the vertices of a polygon, storing the average vertex in `*centroid_ptr`.

```
void SL_PolygonScale(
   SL_Polygon  *pgon_ptr,         /* pointer to polygon to scale */
   double      scale              /* amount by which to scale */
)
```

`SL_PolygonScale()` scales the vertices of a polyon with respect to the polygon's centroid. That is, if `scale` = 0, all vertices shrink to the centroid; if `scale` = 1, the polygon remains unchanged. The scaled polygon is returned through the parameter `*pgon_ptr`.

```
void SL_PolygonRotate(
   SL_Polygon  *pgon_ptr,         /* pointer to polygon to rotate */
   double      angle              /* angle by which to rotate */
)
```

`SL_PolygonRotate()` rotates all the vertices of a polygon about its centroid in the counterclockwise direction by the given angle. The rotated polygon is returned through the parameter `*pgon_ptr`.

```
void SL_PolygonRotateSinCos(
   SL_Polygon  *pgon_ptr,         /* pointer to polygon to rotate */
   double      the_sin,           /* sine of the rotation angle */
   double      the_cos            /* cosine of the rotation angle */
)
```

`SL_PolygonRotateSinCos()` rotates all the vertices of a polygon about is centroid in the counterclockwise direction by a certain angle, the sine and cosine of which are given. The rotated polygon is returned through the parameter `*pgon_ptr`.

```
void SL_PolygonPointRotate(
   SL_Polygon  *pgon_ptr,         /* pointer to polygon to rotate */
   SL_Point    *pnt_ptr,          /* point around which to rotate */
   double      angle              /* angle by which to rotate */
)
```

`SL_PolygonPointRotate()` rotates all the vertices of a polygon about a given point in the counterclockwise direction by a certain angle. The rotated polygon is returned through the parameter `*pgon_ptr`.

```
void SL_PolygonPointRotateSinCos(
   SL_Polygon  *pgon_ptr,         /* pointer to polygon to rotate */
   SL_Point    *pnt_ptr,          /* point around which to rotate */
   double      the_sin            /* sine of rotation angle */
   double      the_cos            /* cosine of rotation angle */
)
```

`SL_PolygonPointRotateSinCos()` rotates all the vertices of a polygon about a given point in the counterclockwise direction by a certain angle, the sine and cosine of which are given. The rotated polygon is returned through the parameter `*pgon_ptr`.

```
void SL_PolygonMove(
   SL_Polygon  *pgon_ptr,         /* pointer to polygon to be translated */
   SL_Point    *new_centroid_ptr  /* pointer to desired centroid */
)
```

`SL_PolygonMove()` moves a polygon rigidly to make its new centroid corresponds to the point `*new_centroid_ptr`. The repositioned polygon is returned through the parameter `*pgon_ptr`.

```
double SL_PolygonAverageRadius(
   SL_Polygon  *pgon_ptr          /* pointer to polygon in question */
)
```

`SL_PolygonAverageRadius()` computes the average distance of all the polygon's vertices from the polygon's centroid.

```
int SL_PolygonClosestVertex(
   SL_Polygon  *pgon_ptr,         /* pointer to polygon in question */
   SL_Point    *pnt_ptr           /* pointer to point in question */
)
```

`SL_PolygonClosestVertex()` returns the index of the vertex of a given polygon that is closest to a given point. If the polygon has less than 3 vertices, the function returns -1.

```
int SL_PolylineSelfInter(
   SL_Polyline *pline_ptr         /* pointer to polyline in question */
)
```

```
int SL_PolygonSelfInter(
   SL_Polygon  *pgon_ptr          /* pointer to polygon in question */
)
```

`SL_PolylineSelfInter()` (`SL_PolygonSelfInter()`) determines if a polyline (polygon) is self-intersecting. The function returns 1 if any two nonconsecutive edges of the given polyline (polygon) intersect and returns 0 otherwise.

8.5 `SL_Shape` Functions

The `SL_Shape` data type is a generalized library primitive. It can represent any primitive using using appropriate values for its `type` and `data` fields. Generalized functions for computing, for example, distances and intersections between shapes are presented in the appropriate sections. The following two functions are used to save and load `SL_List`s of shapes to and from files. The format of a *shapes file* is as follows. The data for each shape is on a single line in the file. Each line begins with a word indicating the type of shape, followed by the data for that shape as indicated below:

POINT x y
VECTOR x y
SEGMENT x_{start} y_{start} x_{end} y_{end}
POLYLINE *size* x_{v1} y_{v1} x_{v2} y_{v2} $\ldots$
POLYGON *size* x_{v1} y_{v1} x_{v2} y_{v2} $\ldots$
ARC *radius* x_{center} y_{center} *StartAngle EndAngle*
CIRCLE *radius* x_{center} y_{center}

```
void SL_SaveShapesFile(
   char        *filename          /* name of file in which to save shapes */
   SL_List     shape_list         /* list of shapes to save */
)
```

`SL_SaveShapesFile()` stores an `SL_List` of shapes in a file so the shapes may be restored for future use. If the file `*filename` already exists, it will be overwritten.

```
SL_List SL_LoadShapesFile(
   char        *filename          /* name of file from which to load shapes */
)
```

`SL_LoadShapesFile()` reads a collection of shapes from a file and returns an `SL_List` in which each element is an `SL_Shape`. If the file `*filename` does not exist or is unreadable, the function returns `NULL`.

```
int SL_CollideShapeShape(
   SL_Shape    *shape1_ptr        /* pointer to first shape */
   SL_Shape    *shape2_ptr        /* pointer to second shape */
)
```

`SL_CollideShapeShape()` determines if the two shapes `*shape1_ptr` and `*shape2_ptr` collide with each other. Two shapes are said to collide if they intersect or one is completely contained within the other. If the shapes collide, the function returns 1; otherwise it returns 0.

8.6 Canvas Polygon Functions

The *canvas polygon* is the four-vertex polygon that surrounds the rectangular drawing canvas. The vertices of this polygon may be retrieved using the following function:

```
void SL_GetCanvasPolygon(
   SL_Polygon  *pgon_ptr,         /* pointer to the polygon */
)
```

`SL_GetCanvasPolygon()` stores the vertices of the canvas polygon in the parameter `*pgon_ptr`.

Manipulation of the vertices of this polygon will **not** alter the shape or size of the drawing canvas. The following functions are used to test for and calculate intersections with the canvas polygon for each of the primitives in the library. Note that it is **not** necessary to call `SL_GetCanvasPolygon()` before calling these functions.

8.6.1 Canvas Polygon Intersection Testing Functions

```
int SL_InsidePointCanvasPolygon(
   SL_Point    *pnt_ptr,          /* pointer to the point */
)
```

`SL_InsidePointCanvasPolygon()` checks if a point is inside the canvas polygon and returns 1 if it is and 0 otherwise.

```
int SL_InterSegmentCanvasPolygonCheck(
   SL_Segment  *seg_ptr           /* pointer to the segment */
)


int SL_InterPolylineCanvasPolygonCheck(
   SL_Polyline *pline_ptr         /* pointer to polyline */
)


int SL_InterPolygonCanvasPolygonCheck(
   SL_Polygon  *pgon_ptr          /* pointer to polygon */
)


int SL_InterArcCanvasPolygonCheck(
   SL_Arc      *arc_ptr           /* pointer to arc */
)


int SL_InterCircleCanvasPolygonCheck(
   SL_Circle   *circ_ptr          /* pointer to circle */
)


int SL_InterShapeCanvasPolygonCheck(
   SL_Shape    *shape_ptr         /* pointer to shape */
)
```

Each of the above functions checks if a given library primitive (segment, polyline, polygon, arc, circle, or shape) intersects some edge of the canvas polygon and returns 1 if it does and 0 otherwise. Line segments or edges that are collinear and overlap are not considered to intersect.

8.6.2 Canvas Polygon Intersection Calculation Functions

```
int SL_InterSegmentCanvasPolygon(
   SL_Segment  *seg_ptr           /* pointer to the segment */
   SL_Point    **inter_ptr        /* pointer to the array of intersection points */
)
```

```
int SL_InterPolylineCanvasPolygon(
   SL_Polyline *pline_ptr         /* pointer to polyline */
   SL_Point    **inter_ptr        /* pointer to the array of intersection points */
)
```

```
int SL_InterPolygonCanvasPolygon(
   SL_Polygon  *pgon_ptr          /* pointer to polygon */
   SL_Point    **inter_ptr        /* pointer to the array of intersection points */
)
```

```
int SL_InterArcCanvasPolygon(
   SL_Arc      *arc_ptr           /* pointer to arc */
   SL_Point    **inter_ptr        /* pointer to the array of intersection points */
)
```

```
int SL_InterCircleCanvasPolygon(
   SL_Circle   *circ_ptr          /* pointer to circle */
   SL_Point    **inter_ptr        /* pointer to the array of intersection points */
)
```

```
int SL_InterShapeCanvasPolygon(
   SL_Shape    *shape_ptr         /* pointer to shape */
   SL_Point    **inter_ptr        /* pointer to the array of intersection points */
)
```

Each of the above functions computes the intersection points (if any) of a given library primitive (segment, polyline, polygon, arc, circle, or shape) and the canvas polygon. If the primitive does not intersect the canvas polygon, the function returns 0 and `**inter_ptr` remains unchanged. Otherwise, the intersection of the primitive with each side of the canvas polygon is tested and the points of intersection are stored in (`*inter_ptr`)[0], ..., (`*inter_ptr`)[$n-1$], where n is the return value of the function. Note that the memory for the array of intersection points is allocated within this function. The function ignores intersection points that result when a line segment is collinear with and overlaps any side of the canvas polygon.

8.7 Containment Functions

The following functions are used to test for the inclusion of one object inside another. For every geometric primitive, there are two functions: one to test for containment in a polygon and the other to test for containment in a circle. In each case, if the first argument passed to the function is completely contained in the second, the function returns 1, otherwise it returns 0. Note that the line segment connecting the endpoints of an arc is not considered part of the arc.

```
int SL_InsidePointPolygon(
   SL_Point    *pnt_ptr,          /* pointer to the point */
   SL_Polygon  *pgon_ptr          /* pointer to the polygon */
)
```

```
int SL_InsidePointCircle(
   SL_Point    *pnt_ptr,          /* pointer to the point */
   SL_Circle   *circ_ptr          /* pointer to the circle */
)


int SL_InsidePointShape(
   SL_Point    *pnt_ptr,          /* pointer to the point */
   SL_Shape    *shape_ptr         /* pointer to the shape */
)
```

`SL_InsidePointShape()` returns 0 if `*shape_ptr` is not a polygon or a circle.

```
int SL_InsideSegmentPolygon(
   SL_Segment  *seg_ptr,          /* pointer to the segment */
   SL_Polygon  *pgon_ptr          /* pointer to the polygon */
)


int SL_InsideSegmentCircle(
   SL_Segment  *seg_ptr,          /* pointer to the segment */
   SL_Circle   *circ_ptr          /* pointer to the circle */
)


int SL_InsidePolylinePolygon(
   SL_Polyline *pline_ptr,        /* pointer to the polyline */
   SL_Polygon  *pgon_ptr          /* pointer to the polygon */
)


int SL_InsidePolylineCircle(
   SL_Polyline *pline_ptr,        /* pointer to the polyline */
   SL_Circle   *circ_ptr          /* pointer to the circle */
)


int SL_InsidePolygonPolygon(
   SL_Polygon  *pgon1_ptr,        /* pointer to the first polygon */
   SL_Polygon  *pgon2_ptr         /* pointer to the second polygon */
)


int SL_InsidePolygonCircle(
   SL_Polygon  *pgon_ptr,         /* pointer to the polygon */
   SL_Circle   *circ_ptr          /* pointer to the circle */
)


int SL_InsideArcPolygon(
   SL_Arc      *arc_ptr,          /* pointer to the arc */
   SL_Polygon  *pgon_ptr          /* pointer to the polygon */
)


int SL_InsideArcCircle(
   SL_Arc      *arc_ptr,          /* pointer to the arc */
   SL_Circle   *circ_ptr          /* pointer to the circle */
)
```

```
int SL_InsideCirclePolygon(
   SL_Circle   *circ_ptr,          /* pointer to the circle */
   SL_Polygon  *pgon_ptr           /* pointer to the polygon */
)


int SL_InsideCircleCircle(
   SL_Circle   *circ1_ptr,         /* pointer to the first circle */
   SL_Circle   *circ2_ptr          /* pointer to the second circle */
)


int SL_InsideShapeShape(
   SL_Shape    *shape1_ptr,        /* pointer to the first shape */
   SL_Shape    *shape2_ptr         /* pointer to the second shape */
)
```

`SL_InsideShapeShape()` returns 0 if `*shape2_ptr` is something other than a polygon or a circle.

8.8 Translation Functions

The following functions are used to translate geometric primitives rigidly by a given translation vector.

```
void SL_TranslateVector(
   SL_Vector   *vec_ptr,           /* the vector to translate */
   SL_Vector   *transl_ptr         /* the vector by which to translate */
)
```

`SL_TranslateVector()` may also be used to translate objects of type `SL_Point`.

```
void SL_TranslateSegment(
   SL_Segment  *segm_ptr,          /* the segment to translate */
   SL_Vector   *transl_ptr         /* the vector by which to translate */
)


void SL_TranslatePolygon(
   SL_Polygon  *pgon_ptr,          /* the polygon to translate */
   SL_Vector   *transl_ptr         /* the vector by which to translate */
)
```

`SL_TranslatePolygon()` may also be used to translate objects of type `SL_Polyline`.

```
void SL_TranslateArc(
   SL_Arc      *arc_ptr,           /* the arc to translate */
   SL_Vector   *transl_ptr         /* the vector by which to translate */
)


void SL_TranslateCircle(
   SL_Circle   *circ_ptr,          /* the circle to translate */
   SL_Vector   *transl_ptr         /* the vector by which to translate */
)


void SL_TranslateShape(
   SL_Shape    *shape_ptr,         /* the shape to translate */
   SL_Vector   *transl_ptr         /* the vector by which to translate */
)
```

8.9 Intersection Functions

There are two sets of functions in this section. The first set return either 1 or 0 depending on whether the intersection condition they were designed to test is true or not. Intersection checking functions are provided for all pairs of objects.

The second set of functions are the intersection calculation routines that compute the actual points of intersection of any two structures in the library. The return values of these functions represent the number of intersections between the two structures.

In all cases, edges or segments that are collinear and overlap are not considered to intersect.

8.9.1 Intersection Checking Functions

```
int SL_PointOnSegment(
   SL_Point    *pnt_ptr,           /* pointer to the point */
   SL_Segment  *seg_ptr            /* pointer to the segment */
)
```

`SL_PointOnSegment()` determines if the given point is on the given line segment. If the point is on the line defined by the segment and between the two segment endpoints, the function returns 1; otherwise, the function returns 0.

The remaining functions in this section all work in the same manner. In each case, the functions check for intersections between two entities of the type implied by the name of the function. If the entities intersect, the function returns 1, otherwise it returns 0.

```
int SL_InterSegmentSegmentCheck(
   SL_Segment  *seg1_ptr,          /* pointer to the first segment */
   SL_Segment  *seg2_ptr           /* pointer to the second segment */
)


int SL_InterSegmentPolylineCheck(
   SL_Segment  *seg_ptr,           /* pointer to the segment */
   SL_Polyline *plin_ptr           /* pointer to the polyline */
)


int SL_InterSegmentPolygonCheck(
   SL_Segment  *seg_ptr,           /* pointer to the segment */
   SL_Polygon  *pgon_ptr           /* pointer to the polygon */
)


int SL_InterSegmentArcCheck(
   SL_Segment  *seg_ptr,           /* pointer to the segment */
   SL_Arc      *arc_ptr            /* pointer to the arc */
)


int SL_InterSegmentCircleCheck(
   SL_Segment  *seg_ptr,           /* pointer to the segment */
   SL_Circle   *circ_ptr           /* pointer to the circle */
)


int SL_InterPolylinePolylineCheck(
   SL_Polyline *pline1_ptr,        /* pointer to the first polyline */
   SL_Polyline *pline2_ptr         /* pointer to the second polyline */
)
```

```
int SL_InterPolylinePolygonCheck(
   SL_Polyline *pline_ptr,        /* pointer to the polyline */
   SL_Polygon  *pgon_ptr          /* pointer to the polygon */
)


int SL_InterPolylineArcCheck(
   SL_Polyline *pline_ptr,        /* pointer to the polyline */
   SL_Arc      *arc_ptr           /* pointer to the arc */
)


int SL_InterPolylineCircleCheck(
   SL_Polyline *pline_ptr,        /* pointer to the polyline */
   SL_Circle   *circ_ptr          /* pointer to the circle */
)


int SL_InterPolygonPolygonCheck(
   SL_Polygon  *poly1_ptr,        /* pointer to the first polygon */
   SL_Polygon  *poly2_ptr         /* pointer to the second polygon */
)


int SL_InterPolygonArcCheck(
   SL_Polygon  *pgon_ptr,         /* pointer to the polygon */
   SL_Arc      *arc_ptr           /* pointer to the arc */
)


int SL_InterPolygonCircleCheck(
   SL_Polygon  *pgon_ptr,         /* pointer to the polygon */
   SL_Circle   *circ_ptr          /* pointer to the circle */
)


int SL_InterArcArcCheck(
   SL_Arc      *arc1_ptr,         /* pointer to the first arc */
   SL_Arc      *arc2_ptr          /* pointer to the second arc */
)


int SL_InterArcCircleCheck(
   SL_Arc      *arc_ptr,          /* pointer to the arc */
   SL_Circle   *circ_ptr          /* pointer to the circle */
)


int SL_InterCircleCircleCheck(
   SL_Circle   *circle1,          /* pointer to the first circle */
   SL_Circle   *circle2           /* pointer to the second circle */
)


int SL_InterShapeShapeCheck(
   SL_Shape    *shape1_ptr,       /* pointer to the first shape */
   SL_Shape    *shape2_ptr        /* pointer to the second shape */
)
```

8.9.2 Intersection Calculation Functions

The functions in this section compute the points of intersection between any two structures in the library. (Again, edges and segments that are collinear and overlap are not considered to intersect.) The return values of these functions represent the number of points of intersection between the two structures. When there can be only one or two intersection points (between two segments, a segment and an arc, a segment and a circle, two arcs, an arc and a cricle, or two circles), the memory for the intersection point(s) is assumed to be allocated outside these functions (avoiding the costly calls to `malloc()`). In all other cases, since there can be an arbitrary number of intersection points, the memory for the array of intersection points is allocated within the intersection function. In this case, the function's third argument is a pointer to an array of intersection points, `**inter_ptr`. The coordinates of the points of intersection are then stored in (`*inter_ptr`)[0], ..., (`*inter_ptr`)[$n-1$], where n is the return value of the function. Note that this memory allocation is expensive, and each call to one of these intersection function causes a new array to be allocated, which can cause a drain on the available memory for your program.

```
int SL_InterSegmentSegment(
   SL_Segment  *seg1_ptr,         /* pointer to the first segment */
   SL_Segment  *seg2_ptr,         /* pointer to the second segment */
   SL_Point    *inter_ptr         /* pointer to the intersection point */
)
```

```
int SL_InterSegmentPolyline(
   SL_Segment  *seg_ptr,          /* pointer to the segment */
   SL_Polyline *pline_ptr,        /* pointer to the polyline */
   SL_Point    **inter_ptr        /* pointer to the array of intersection points */
)
```

```
int SL_InterSegmentPolygon(
   SL_Segment  *seg_ptr,          /* pointer to the segment */
   SL_Polygon  *pgon_ptr,         /* pointer to the polygon */
   SL_Point    **inter_ptr        /* pointer to the array of intersection points */
)
```

```
int SL_InterSegmentArc(
   SL_Segment  *seg_ptr,          /* pointer to the segment */
   SL_Arc      *arc_ptr,          /* pointer to the arc */
   SL_Point    inter[2]           /* array of intersection points */
)
```

```
int SL_InterSegmentCircle(
   SL_Segment  *seg_ptr,          /* pointer to the segment */
   SL_Circle   *circ_ptr,         /* pointer to the circle */
   SL_Point    inter[2]           /* array of intersection points */
)
```

```
int SL_InterPolylinePolyline(
   SL_Polyline *pline1_ptr,       /* pointer to the first polyline */
   SL_Polyline *pline2_ptr,       /* pointer to the second polyline */
   SL_Point    **inter_ptr        /* pointer to the array of intersection points */
)
```

```
int SL_InterPolylinePolygon(
   SL_Polyline *pline_ptr,        /* pointer to the polyline */
   SL_Polygon  *pgon_ptr,         /* pointer to the polygon */
   SL_Point    **inter_ptr        /* pointer to the array of intersection points */
)

int SL_InterPolylineArc(
   SL_Polyline *pline_ptr,        /* pointer to the polyline */
   SL_Arc      *arc_ptr,          /* pointer to the arc */
   SL_Point    **inter_ptr        /* pointer to the array of intersection points */
)

int SL_InterPolylineCircle(
   SL_Polyline *pline_ptr,        /* pointer to the polyline */
   SL_Circle   *circ_ptr,         /* pointer to the circle */
   SL_Point    **inter_ptr        /* pointer to the array of intersection points */
)

int SL_InterPolygonPolygon(
   SL_Polygon  *pgon1_ptr,        /* pointer to the first polygon */
   SL_Polygon  *pgon2_ptr,        /* pointer to the second polygon */
   SL_Point    **inter_ptr        /* pointer to the array of intersection points */
)

int SL_IntertPolygonArc(
   SL_Polygon  *pgon_ptr,         /* pointer to the polygon */
   SL_Arc      *arc_ptr,          /* pointer to the arc */
   SL_Point    **inter_ptr        /* pointer to the array of intersection points */
)

int SL_InterPolygonCircle(
   SL_Polygon  *pgon_ptr,         /* pointer to the polygon */
   SL_Circle   *circ_ptr,         /* pointer to the circle */
   SL_Point    **inter_ptr        /* pointer to the array of intersection points */
)

int SL_InterArcArc(
   SL_Arc      *arc1_ptr,         /* pointer to the first arc */
   SL_Arc      *arc2_ptr,         /* pointer to the second arc */
   SL_Point    inter[2]           /* array of intersection points */
)

int SL_InterArcCircle(
   SL_Arc      *arc_ptr,          /* pointer to the arc */
   SL_Circle   *circ_ptr,         /* pointer to the circle */
   SL_Point    inter[2]           /* array of intersection points */
)

int SL_InterCircleCircle(
   SL_Circle   *circ1_ptr,        /* pointer to the first circle */
   SL_Circle   *circ2_ptr,        /* pointer to the second circle */
   SL_Point    intersect[2]       /* array of intersection points */
)
```

```
int SL_InterShapeShape(
   SL_Shape    *shape1_ptr,       /* pointer to first shape */
   SL_Shape    *shape2_ptr,       /* pointer to second shape */
   SL_Point    **inter_ptr        /* pointer to array of intersection points */
)
```

8.10 Distance Functions

The functions in this section of the library are used to compute the minimum distance between any two library structures (excluding arcs). Each function's return value is a double representing the computed distance. In each case, if the two objects intersect, the return value is 0.

For each function that computes the distance using a structure other than `SL_Point`, there is an output parameter for each non-point object that will contain the point on that object to which the minimum distance corresponds. If the return value of a function is 0, these output parameters remain unchanged.

```
double SL_DistPointPoint(
   SL_Point    *pnt1_ptr,         /* pointer to the first point */
   SL_Point    *pnt2_ptr          /* pointer to the second point */
)
```

```
double SL_DistPointPointSQR(
   SL_Point    *pnt1_ptr,         /* pointer to the first point */
   SL_Point    *pnt2_ptr          /* pointer to the second point */
)
```

`SL_DistPointPointSQR()` returns the square of the Euclidean distance between the two given points (thus avoiding the rather expensive call to the square-root function).

```
int SL_DistPointPointNull(
   SL_Point    *pnt1_ptr,         /* pointer to the first point */
   SL_Point    *pnt2_ptr          /* pointer to the second point */
)
```

`SL_DistPointPointNull()` returns 1 if the distance between the two points provided is negligible and 0 otherwise.

```
double SL_DistPointSegment(
   SL_Point    *pnt_ptr,          /* pointer to point to compute distance from */
   SL_Segment  *seg_ptr,          /* pointer to segment to compute distance to */
   SL_Point    *closest_ptr       /* pointer to point on *seg_ptr closest to
                                     *pnt_ptr */
)
```

`SL_DistPointSegment()` returns the distance between a point and a line segment. If the projection of the point onto the line defined by the segment lies between the endpoints of the segment, then the value returned by the function is the distance between the point and the projected point. If the projection lies somewhere outside the segment, the value of the function is the distance between the given point and the closer endpoint of the segment.

```
double SL_DistPointPolyline(
   SL_Point    *pnt_ptr,          /* pointer to point */
   SL_Polyline *pline_ptr,        /* pointer to polyline */
   SL_Point    *closest_ptr       /* pointer to point on *pline_ptr closest to
                                     *pnt_ptr */
)
```

```
double SL_DistPointPolygon(
   SL_Point    *pnt_ptr,          /* pointer to point */
   SL_Polygon  *pgon_ptr,         /* pointer to polygon */
   SL_Point    *closest_ptr       /* pointer to point on *pgon_ptr closest to
                                     *pnt_ptr */
)
```

```
double SL_DistPointCircle(
   SL_Point    *pnt_ptr,          /* pointer to point */
   SL_Circle   *circ_ptr,         /* pointer to circle */
   SL_Point    *closest_ptr       /* pointer to point on *circ_ptr closest to
                                     *pnt_ptr */
)
```

If point `*pnt_ptr` is contained in `*circ_ptr`, `SL_DistPointCircle()` returns 0 and `*closest_ptr` remains unchanged.

```
double SL_DistSegmentSegment(
   SL_Segment  *seg1_ptr,         /* pointer to first segment  */
   SL_Segment  *seg2_ptr,         /* pointer to second segment */
   SL_Point    *closest1_ptr,     /* pointer to point on *seg1_ptr closest to
                                     *seg2_ptr */
   SL_Point    *closest2_ptr      /* pointer to point on *seg2_ptr closest to
                                     *seg1_ptr */
)
```

```
double SL_DistSegmentPolyline(
   SL_Segment  *seg_ptr,          /* pointer to segment */
   SL_Polyline *pline_ptr,        /* pointer to polyline */
   SL_Point    *closest1_ptr,     /* pointer to point on *seg_ptr closest to
                                     *pline_ptr */
   SL_Point    *closest2_ptr      /* pointer to point on *pline_ptr closest to
                                     *seg_ptr */
)
```

```
double SL_DistSegmentPolygon(
   SL_Segment  *seg_ptr,          /* pointer to segment */
   SL_Polygon  *pgon_ptr,         /* pointer to polygon */
   SL_Point    *closest1_ptr,     /* pointer to point on *seg_ptr closest to
                                     *pgon_ptr */
   SL_Point    *closest2          /* pointer to point on *pgon_ptr closest to
                                     *seg_ptr */
)
```

```
double SL_DistSegmentCircle(
   SL_Segment  *seg_ptr,          /* pointer to segment */
   SL_Circle   *circ_ptr,         /* pointer to circle */
   SL_Point    *closest1_ptr,     /* pointer to point on *seg_ptr closest to
                                     *circ_ptr */
   SL_Point    *closest2_ptr      /* pointer to point on *circ_ptr closest to
                                     *seg_ptr */
)
```

```
double SL_DistPolylinePolyline(
   SL_Polyline *pline1_ptr,       /* pointer to first polyline */
   SL_Polyline *pline2_ptr,       /* pointer to second polyline */
   SL_Point    *closest1_ptr,     /* pointer to point on *pline1_ptr closest to
                                     *pline2_ptr */
   SL_Point    *closest2_ptr      /* pointer to point on *pline2_ptr closest to
                                     *pline1_ptr */
)


double SL_DistPolylinePolygon(
   SL_Polyline *pline_ptr,        /* pointer to polyline */
   SL_Polygon  *pgon_ptr,         /* pointer to polygon */
   SL_Point    *closest1_ptr,     /* pointer to point on *pline_ptr closest to
                                     *pgon_ptr */
   SL_Point    *closest2_ptr      /* pointer to point on *pgon_ptr closest to
                                     *pline_ptr */
)


double SL_DistPolylineCircle(
   SL_Polyline *pline_ptr,        /* pointer to polyline */
   SL_Circle   *circ_ptr,         /* pointer to circle */
   SL_Point    *closest1_ptr,     /* pointer to point on *pline_ptr closest to
                                     *circ_ptr */
   SL_Point    *closest2_ptr      /* pointer to point on *circ_ptr closest to
                                     *pline_ptr */
)


double SL_DistPolygonPolygon(
   SL_Polygon  *pgon1_ptr,        /* pointer to first polygon */
   SL_Polygon  *pgon2_ptr,        /* pointer to second polygon */
   SL_Point    *closest1_ptr,     /* pointer to point on *pgon1_ptr closest to
                                     *pgon2_ptr */
   SL_Point    *closest2_ptr      /* pointer to point on *pgon2_ptr closest to
                                     *pgon1_ptr */
)


double SL_DistPolygonCircle(
   SL_Polygon  *pgon_ptr,         /* pointer to polygon */
   SL_Circle   *circ_ptr,         /* pointer to circle */
   SL_Point    *closest1_ptr,     /* pointer to point on *pgon_ptr closest to
                                     *circ_ptr */
   SL_Point    *closest2_ptr      /* pointer to point on *circ_ptr closest to
                                     *pgon_ptr */
)


double SL_DistCircleCircle(
   SL_Circle   *circ1_ptr,        /* pointer to first circle */
   SL_Circle   *circ2_ptr,        /* pointer to second circle */
   SL_Point    *closest1_ptr,     /* pointer to point on *circ1_ptr closest to
                                     *circ2_ptr */
   SL_Point    *closest2_ptr      /* pointer to point on *circ2_ptr closest to
                                     *circ1_ptr */
)
```

```
double SL_DistShapeShape(
   SL_Shape    *shape1_ptr,       /* pointer to first shape */
   SL_Shape    *shape2_ptr,       /* pointer to second shape */
   SL_Shape    *closest1_ptr,     /* pointer to point on *shape1_ptr closest to
                                     *shape2_ptr */
   SL_Shape    *closest2_ptr      /* pointer to point on *shape2_ptr closests to
                                     *shape1_ptr */
)
```

SL_DistShapeShape() returns as its value the minimum distance between two shapes and returns through its parameters *closest1_ptr and *closest2_ptr the point on each shape to which this distance corresponds. However, if either or both shapes are points, the corresponding output parameter(s) remains unchanged. If one of the shapes is either a vector or an arc or if the two shapes intersect, the function returns 0, leaving the output parameters unchanged.

8.11 Tangent Functions

```
SL_TangentPointPolygon(
   SL_Point    *pnt_ptr,          /* pointer to the point in question */
   SL_Polygon  *pgon_ptr,         /* pointer to the polygon in question */
   SL_Point    tangents[2]        /* the location for the two tangent points */
)
```

SL_TangentPointPolygon() calculates the two tangent points from a given point in the plane to the *convex hull* of a given polygon. If the point is contained inside the convex hull of the polygon or the polygon has fewer than 3 vertices, the function returns 0 leaving tangents unchanged, otherwise the function returns 1 and stores the coordinates of the two tangent points in the array tangents.

```
SL_TangentPointCircle(
   SL_Point    *pnt_ptr,          /* pointer to the point in question */
   SL_Circle   *circ_ptr,         /* pointer to the circle in question */
   SL_Point    tangents[2]        /* the location for the two tangent points */
)
```

SL_TangentPointCircle() calculates the two tangent points from a given point in the plane to a given circle. If the point is contained inside the circle, the function returns 0 and leaves tangents unchanged, otherwise the function returns 1 and stores the coordinates of the two tangent points in the array tangents.

```
SL_TangentCircleCircle(
   SL_Circle   *circ1_ptr,        /* pointer to the first circle */
   SL_Circle   *circ2_ptr,        /* pointer to the second circle */
   SL_Point    tangents[8]        /* location for the four pairs of tangent points */
)
```

SL_TangentCircleCircle() calculates the points of tangency between two given circles. For two circles that do not intersect, there are four tangent lines and the function returns 4. The four pairs of tangent points are returned in the array tangents. The tangent points for *circ1_ptr are returned in the array entries with odd indices (1, 3, 5, 7) with the corresponding tangents for *circ2_ptr in the even numbered entries (0, 2, 4 and 6, respectively). If the two circles intersect, there are only two tangent lines. The function returns 2 in this case, and the tangent points for these line are returned in tangents[0...3], leaving tangents[4...7] unchanged. If one circle is completely contained within the other, there are no tangent lines, so the function returns 0, leaving the entire tangents array unchanged.

8.12 User Interface Functions

```
void SL_Init(
   int         width,             /* width of the drawing canvas in pixels */
   int         height,            /* height of the drawing canvas in pixels */
   int         obst_enable,       /* 1 to enable the use of the obstacle database */
   void        (*InitCbk)(void),  /* callback for initialization */
   void        (*ResetCbk)(void), /* callback for resetting simulation */
   void        (*RedrawCbk)(void),/* callback for redrawing simulation */
   void        (*GoCbk)(int stepping_on),
                                  /* callback evoked when 'Go' button pressed */
   int         (*StepCbk)(void),  /* callback that updates simulation by one step */
   void        (*ClickCbk)(int x, int y, SL_Button btn),
                                  /* callback for mouse button clicks */
   void        (*MotionCbk)(int x, int y),
                                  /* callback for mouse motions */
   void        (*KeyboardCbk)(char the_key, int key_pressed)
                                  /* callback for keyboard input */
)
```

`SL_Init()` is the function called to initialize the user interface. It should be called once before any of the other user-interface routines are called. The parameters `height` and `width` specify the size of the rectangular drawing canvas. If the SL-application is to use the obstacle database (Section 3) the parameter `obst_enable` should be passed a value of 1, otherwise it should be 0. The other parameters of SL_Init() are the callback functions that are called during the UI-loop when certain events occur (Sections 2.1 and 2.2). They are:

`void (*InitCbk)(void)`

This parameter defines the function that will be called once when the application's window appears on the screen. The function is called with no arguments and should contain initialization code that is to be performed only once during a run of an application. If no such initialization is needed, this argument should be `NULL`.

`void (*ResetCbk)(void)`

This parameter defines the function that will be called when the "Reset" option on the built-in "Simulation" menu is selected. It is also called immediately after the `InitCbk()` when the SL-application is first started. This function is called with no arguments and should generally be used to restore simulation-specific data (*e.g.*, a robot's location) to some initial state (Section 2.1). This argument may be `NULL` if no such resetting is desired or required.

`void (*RedrawCbk)(void)`

This parameter defines the function that will be called when the "Redraw" option on the built-in "Simulation" menu is selected. This function should redraw the simulation environment and will be called with no arguments. This is also the function called during the UI loop to update the scene after a step (as defined by `StepCbk()`) has been made and thus produces the simulation's animation. This argument to `SL_Init()` may also be `NULL` if you never wish to redraw the scene, but it will probably make for a rather uninteresting simulation.

`void (*GoCbk)(int stepping_on)`

This parameter defines the function that will be called each time the built-in "Go" button is pressed. This function will be called with a single integer argument that has a value of 1 or 0 to indicate if the Go button is being toggled on or off. If there is not need to detect when this button is pressed, this argument to `SL_Init()` should be `NULL` (Section 2.1).

`int (*StepCbk)(void)`

This parameter defines the function that will be called when the simulation is in active mode (*i.e.*, the "Go" button has been pressed to start the motion of the objects in the environment). This function will be called with no arguments and should generally move the objects in the environment a single step. The function should return a 0 if the stepping is to stop (*e.g.*, if the robot reaches its target point) and 1 otherwise (Section 2.1).

`void (*ClickCbk)(int x, int y, SL_Button btn)`

This parameter defines the function that will be called when a mouse button is pressed or released with the mouse pointer inside the drawing canvas. This will be called with three integer arguments: the x coordinate of the position of the pointer on the canvas, the y coordinate of the position of the pointer on the canvas, an integer value of type `SL_Button` (Section 7) that indicates which button (left, middle or right) was either pressed or released and whether it was pressed or released. If there is no need to detect when the mouse is clicked, this argument to `SL_Init()` should be `NULL` (Section 2.2).

`void (*MotionCbk)(int x, int y)`

This parameter defines the function that will be called when the mouse pointer is moved inside the drawing canvas. The function will be called with two integer arguments: the x and y coordinates of the current mouse pointer position. If there is no need to detect when the mouse is moved, this argument to `SL_Init()` should be `NULL` (Section 2.2).

`void (*KeyboardCbk)(char the_key, int key_pressed)`

This parameter defines the function that will be called when a key on the keyboard is pressed or released. The function will be called with two arguments, one (`the_key`) representing the key for which the action occurred and the other (`key_pressed`) indicating whether the key was pressed or released. If there is no need to detect when a key is pressed, this argument to `SL_Init()` should be `NULL` (Section 2.2).

```
void SL_SetUp3dCanvas(
   int         cube_rad,           /* the dimensions of the 3d drawing cube */
   void        (*redraw_fn)(void)  /* the function that redraws the 3d scene */
)
```

`SL_SetUp3dCanvas()` adds a double-buffered three-dimensional drawing area to the right of the two-dimensional canvas specified by the call to `SL_Init()`. If this function is used, it must be called right after the call to `SL_Init()`. A toggle button, labeled "Draw3d," is automatically added to the top of the 3d canvas, allowing the user to choose whether the 3d objects are drawn or not. Pose control buttons are added to the bottom of the 3d canvas to allow the user to adjust the viewpoint from which the objects are seen. This function is available only when working on the SGI machines.

The user interface generally consists of a single window with a drawing canvas in the center on which the motion simulation is displayed. There is an area below the drawing canvas (the *bottom row*) into which the buttons and labels defined with the following routines will be placed. Using these routines, the user interface can be tailored to the particular needs of a specific application. The order in which these routines are called determines the order in which the buttons and labels are placed on the user interface (Section 2.3).

```
void SL_AddLabel(
   char        *label              /* the string that is to be printed */
)
```

`SL_AddLabel()` puts a string of characters in the bottom row below the drawing canvas.

```
void SL_AddButton(
   char        *btn_label,         /* the string labeling the button */
   void        (*cback_fn)(void)  /* the function to be called when the
                                     button is clicked with the mouse */
)
```

`SL_AddButton()` adds a button to the bottom row of the user interface and labels it with the specified string. This function also associates the callback function `(*cback_fn)()` with the button so that whenever the mouse is clicked on this button, the function `(*cback_fn)()` will be called with no arguments.

```
void SL_AddToggle(
   char        *tog_label,        /* label for toggle button */
   void        *cback_fn(int tog_on)
                                  /* function to call when the toggle
                                     button is clicked */
)
```

`SL_AddToggle()` adds a toggle button to the bottom row of the user interface and labels it with the specified string. The function also associates the callback function `(*cback_fn)()` with the button. Whenever the mouse is clicked with the pointer over this button, the function will be called. If the toggle button is “on”, `(*cback_fn)()` will be called with a 1, otherwise it will be called with a 0. The toggle is initially in the “off” state, so the first click will turn it “on”.

```
void SL_AddDialog(
   char        *btn_label,        /* the string labeling the button */
   char        *dialog_label,     /* the text to appear at top of dialog window */
   void        (*cback_fn)(*the_string)
                                  /* the function to be called from the
                                     dialog window with the entered text */
)
```

`SL_AddDialog()` adds a button labeled with a specified string to the bottom row of the user interface. When this button is pressed, a window that contains an area for the user to input text will pop up. At the top of this dialog window will be the text passed to `*dialog_label` (*e.g.*, a prompt to the user). Also in the dialog window there will be two buttons: one labeled “OK” and the other labeled “CANCEL” (Figure 7). The function `(*cback_fn)()` will be called when the user clicks on the “OK” button. This callback function should accept one argument that is a pointer to the character string entered by the user. If the “CANCEL” button is clicked, the dialog window will disappear and the callback function will not be called.

```
void SL_AddMenu(
   char        *btn_label,        /* the string labeling the menu button */
   char        **entry_labels,    /* the names of the distinct menu options */
   void        (*cback_fn)(int menu_index)
                                  /* the function to be called when a menu
                                     option is selected */
)
```

`SL_AddMenu()` adds a menu button to the bottom row of the user interface. This menu button will be labeled with the string `*btn_label`. The second parameter to this function, `entry_labels`, is a `NULL`-terminated array of `char` pointers. Each element of this array is a string that represents the name of one of the distinct menu options. When the menu button is pressed, the menu will pop up. When the user selects one of the options in the menu, the function `(*cback_fn)()` will be called with a single integer argument that represents the number of the menu option selected. The first menu option is option 0, so if there are N options displayed in the menu, `(*cback_fn)()` will be called with the argument value in the range $[0, N-1]$.

Once the user interface has been designed and the proper callback routines have been established, the following function should be called to invoke the UI loop.

```
void SL_Loop(void)
```

`SL_Loop()` is the function that passes control to the user interface. It should be called when the whole user interface is set up and the program is ready to fall into an *event-loop* in which the callback routines provided by the functions described in this section will be called whenever appropriate.

```
int SL_SteppingOn(void)
```

`SL_SteppingOn()` may be called to determine if the Go button is currently toggled on or off. The function has no parameters and returns 1 if the Go button is on and 0 if not.

8.13 Graphics Functions

This section contains descriptions of the routines that are used to produce graphical output for a simulation.

Drawing is displayed on the drawing canvas of the application's window (Figure 1). This area is mapped as a normal xy-plane, with the origin $(x, y) = (0, 0)$ in the lower left corner and the x and y axes growing to the right and upwards, respectively. The maximum values of x and y are defined by the first two arguments of the call to `SL_Init()` (Sections 2.1 and 8.12).

8.13.1 Colors and Drawing Thickness

```
int SL_MakeColor(
   char        *color_name        /* name of the color to change or create */
   double      red                /* amount of red in new color */
   double      green              /* amount of green in new color */
   double      blue               /* amount of blue in new color */
)
```

`SL_MakeColor()` is used to create or change a color by specifying the appropriate red, green, and blue (RGB) values. The `color_name` is a case-insensitive character string designating the name by which the color is to be referred. There are 11 predefined color names:

"white"	"black"	"yellow"
"green"	"blue"	"red"
"grey"	"cyan"	"magenta"
"obstacles"	"background"	

The color "obstacles", used to draw elements of the obstacle database (Sections 3 and 8.15), is black by default; "background", the color of the canvas, defaults to white. Any of these predefined colors may be redefined (although it probably makes sense for only "obstacles" and "background"). The red, green, and blue values should each be in the range $[0, 1]$. The function returns 1 if the color is created or changed successfully and 0 if no more colors can be created.

```
int SL_GetColor(
   char        *color_name        /* name of the color */
   double      *red_ptr           /* amount of red in color */
   double      *green_ptr         /* amount of green in color */
   double      *blue_ptr          /* amount of blue in color */
)
```

`SL_GetColor()` returns the RGB values of an existing color through the parameters `*red_ptr`, `*green_ptr`, or `*blue_ptr`. If the color name does not exist, the function returns 0 and leaves `*red_ptr`, `*green_ptr`, and `*blue_ptr` unchanged, otherwise the function returns 1.

```
int SL_SetDrawColor(
   char        *color_name        /* name of the color */
)
```

`SL_SetDrawColor()` defines the color with which primitives are to be drawn. (The default drawing color is black.) If the argument is one of the predefined color names or a color name defined by `SL_MakeColor()`, the function returns 1, otherwise it returns 0. Matching of color names is case-insensitive. Note that, on non-color monitors, colors with intensity < 0.5 are drawn as black and those with intensity ≥ 0.5 are drawn as white. Intensity is measured as: $(0.299 \times red) + (0.587 \times green) + (0.114 \times blue)$.

```
void SL_SetDrawThickness(
   int         thickness          /* an integer greater than 0 specifying the
                                     desired drawing thickness in pixels */
)
```

`SL_SetDrawThickness()` defines the width (in pixels) with which the boundary of primitives are to be drawn. The default drawing thickness is one pixel.

8.13.2 The Primitive Drawing Functions

The following functions are used to draw the library primitives on the drawing canvas. The primitive is drawn in the default color (black) or the color indicated by the most recent call to `SL_SetDrawColor()`. The functions for drawing `SL_Polygon`s, `SL_Arc`s, `SL_Circle`s and `SL_Shape`s accept two arguments, the second of which is a flag which should be 0 if the primitive should be drawn as an outline only or 1 if it is to be filled with the current drawing color.

```
void SL_DrawPoint(
   SL_Point    *pnt_ptr           /* pointer to the point to be drawn */
)
```

Note that calls to `SL_SetDrawThickness()` do not affect how large a point will be drawn. It will always be rather small. This function should also be used to draw `SL_Vector`s.

```
void SL_DrawSegment(
   SL_Segment  *seg_ptr           /* pointer to the segment to be drawn */
)

void SL_DrawPolyline(
   SL_Polyline *pline_ptr         /* pointer to the polyline to be drawn */
)

void SL_DrawPolygon(
   SL_Polygon  *pgon_ptr,         /* pointer to the polygon to be drawn */
   int         filled             /* flag indicting if polygon should be filled */
)

void SL_DrawArc(
   SL_Arc      *arc_ptr,          /* pointer to the arc to be drawn */
   int         filled             /* flag indicating if arc should be filled */
)

void SL_DrawCircle(
   SL_Circle   *circ_ptr,         /* pointer to the circle to be drawn */
   int         filled             /* flag indicating if circle should be filled */
)

void SL_DrawShape(
   SL_Shape    *shape_ptr,        /* pointer to the shape to be drawn */
   int         filled             /* flag indicating if shape should be filled */
)
```

The `filled` parameter has no effect when the shape is other than a polygon, arc, or circle.

8.13.3 Drawing text and other things

```
SL_DrawText(
   SL_Point    *point            /* pointer to coordinates for the lower left
                                    coordinates of the drawn string */
   char        *text             /* the string to display */
)
```

`SL_DrawText()` draws a string of text at a specified location in the current drawing color. Note that the style of the text will not be affected by calls to `SL_SetDrawThickness()`.

```
SL_PrintToStatusBar(
   char        *what_to_print    /* the text to print */
)
```

`SL_PrintToStatusBar()` displays a string of text in the status bar that appears between the drawing canvas and bottom row.

```
SL_DrawAxes(
   double      length            /* the length of the axes */
)
```

`SL_DrawAxes()` draws a pair of axes of length `length` with origin in the lower left corner of the drawing canvas. The axes are labeled "x" and "y".

```
void SL_ClearCanvas(void)
```

`SL_ClearCanvas()` clears the contents of the canvas with the currently active color, as specified by the latest call to `SL_SetDrawColor()` (Section 2.1). The default color (white) is used if no calls to `SL_SetDrawColor()` have been made.

```
void SL_Redraw(void)
```

`SL_Redraw()` causes three things to happen when it is called: (1) the canvas is cleared to the "background" color; (2) the user's redrawing callback routine, established in the call to `SL_Init()`, is called (Section 2.1); (3) the obstacles are redrawn if the obstacle database has been enabled (Section 3). This function is typically called when a callback routine changes what needs to be drawn but the Go button is not toggled on.

8.14 Mouse Editing Functions

The following functions allow the user to manipulate SL primitives using the mouse. For each primitive, there is a function for inputting the structure with the mouse. All primitives except `SL_Point` and `SL_Vector` have functions that allow them to be moved with the mouse. Functions that move a single vertex of a polygon or polyline are also provided. Functions are available for rotating `SL_Polygon`s, `SL_Polyline`s, and `SL_Arc`s and for scaling these structures as well as `SL_Circle`s using mouse movements.

8.14.1 Inputting Primitives with the Mouse

```
int SL_MouseGetPosition(
   SL_Point    *pnt_ptr          /* pointer to location for input point */
)
```

`SL_MouseGetPosition()` allows the user to determine the position of the mouse pointer on the canvas. If the mouse pointer is inside the drawing canvas, the x and y coordinates of the pointer position are stored in `*pnt_ptr` and the function returns 1. Otherwise, the function returns 0, leaving `*pnt_ptr` unchanged.

```
int SL_MouseGetPoint(
   SL_Point    *pnt_ptr            /* pointer to location for input point */
)
```

`SL_MouseGetPoint()` allows the user to input an `SL_Point` using the mouse. (It may also be used to input an `SL_Vector`.) When the function is called, the program waits for the user to click one of the mouse buttons inside the drawing canvas before proceeding. If the left or middle mouse button is clicked inside the drawing canvas, the function stores in `*pnt_ptr` the x and y coordinates of the mouse pointer position and returns 1. If the right button is clicked, the function returns 0, leaving `*pnt_ptr` unchanged.

```
int SL_MouseGetSegment(
   SL_Segment  *seg_ptr            /* pointer to location for input segment */
)
```

`SL_MouseGetSegment()` allows the user to input an `SL_Segment` using the mouse. When the function is called, the program waits for the user to click one of the mouse buttons inside the drawing canvas. If the left or middle mouse button is clicked, this defines the starting point for the input segment. The ending point is defined by the position of the mouse pointer. This segment is drawn on the canvas as the mouse is moved. This continues until the user presses a mouse button again. If the left or middle mouse button is pressed, the current mouse pointer position is stored as the endpoint of the segment and the function returns 1. If at any time the right mouse button is clicked, the function returns 0, leaving `*seg_ptr` unchanged.

```
int SL_MouseGetRectangle(
   SL_Polygon  *rect_pgon_ptr      /* pointer to location for input rectangle */
)
```

`SL_MouseGetRectangle()` allows the user to enter a rectangular `SL_Polygon` using the mouse. When the function is called, the program waits for the user to click a mouse button inside the drawing canvas. If the left or middle mouse button is clicked, this defines one of the corners of the rectangle. The opposite corner (across the diagonal from the first corner) is defined by the position of the mouse pointer as the mouse is moved in the drawing canvas. As the mouse is moved, this rectangle is drawn on the canvas. This continues until a mouse button is clicked. If the left or middle mouse button is pressed, the coordinates of the rectangle drawn on the screen are stored in `*rect_pgon_ptr`, and the function returns 1. If the right mouse button is clicked at any time, the function returns 0, leaving `*rect_pgon_ptr` unchanged.

```
int SL_MouseGetPolyline(
   SL_Polyline *pline_ptr          /* pointer to location for input polyline */
)

int SL_MouseGetPolygon(
   SL_Polygon  *pgon_ptr           /* pointer to location for input polygon */
)
```

`SL_MouseGetPolyline()` (`SL_MouseGetPolygon()`) allows the user to input an `SL_Polyline` (`SL_Polygon`) using the mouse. When the function is called, the program waits for the user to click one of the mouse buttons inside the drawing canvas. If the left or middle mouse button is clicked, the position of the mouse pointer at that time is recorded as the first vertex of the polyline (polygon). Additional vertices of the polyline (polygon) (up to a limit of `SL_MAX_VTX`) may be defined by moving the mouse pointer to a point on the drawing canvas and clicking the left mouse button. The polyline (polygon) being constructed is drawn on the canvas during this process, with the endpoint of the last edge following the mouse pointer as it is moved. When all the desired vertices have been so specified, a click with the middle mouse button will terminate the input process. The position of the pointer when the middle mouse button is clicked is **not** stored as one of the vertices. If the input process is ended with a click of the middle mouse button or `SL_MAX_VTX` vertices have been defined, the polyline (polygon) that has been specified is stored in `*pline_ptr` (`*pgon_ptr`), and the function returns 1. At any time during this process a click with the right mouse button aborts the input process, causing the function to return 0 without changing `*pline_ptr` (`*pgon_ptr`).

```
int SL_MouseGetArc(
   SL_Arc      *arc_ptr           /* pointer to location for input arc */
)
```

`SL_MouseGetArc()` allows the user to input an `SL_Arc` using the mouse. When the function is called, the program waits for the user to click one of the mouse buttons inside the drawing canvas. A click with the left or middle mouse button specifies the position of the mouse pointer as the center c of the arc. Then as the mouse is moved on the canvas, a line is drawn between c and the pointer position. This line represents the arc radius. Another click of the left or middle mouse button specifies the starting point of the arc and thus defines the arc radius, r. Once the starting point has been specified, the mouse pointer position is used to define the ending point of the arc. This point is the point on the circle of radius r centered at c that makes the same angle with the line from c to the starting point as a point at the mouse pointer position does. The arc so defined is drawn on the canvas as the mouse is moved. Note that the arc is always drawn counterclockwise from the starting point to the ending point. Another click of the left or middle mouse button causes the arc specified by the previous clicks of the mouse and the current mouse pointer position to be stored in `*arc_ptr`. The function then returns 1. If at any time the right mouse button is clicked, the function returns 0, leaving `*arc_ptr` unchanged.

```
int SL_MouseGetCircle(
   SL_Circle   *circ_ptr          /* pointer to location for input circle */
)
```

`SL_MouseGetCircle()` allows the user to input an `SL_Circle` using the mouse. When the function is called, the program waits for the user to click one of the mouse buttons inside the drawing canvas. If the left or middle mouse button is clicked, the position of the mouse pointer at that time is stored as the center of the circle. Then, the distance of the mouse pointer from this center point is used as the radius of the circle. This circle of changing size is drawn on the canvas as the mouse is moved. If the left or middle mouse button is clicked, the circle that has been specified by the position of the mouse pointer and the center point is stored in `*circ_ptr` and the function returns 1. If the right mouse button is clicked at any time, the function returns 0, leaving `*circ_ptr` unchanged.

```
int SL_MouseGetShape(
   SL_Shape    *shape_ptr         /* pointer to location for input shape */
)
```

`SL_MouseGetShape()` allows the user to input an `SL_Shape` using the mouse. The argument to this function, `*shape_ptr`, must have had its `type` field set to one of the seven values of type `SL_ShapeType` (Section 7) before this function is called. Then this function performs just as the corresponding primitive-getting functions previously described in this section. If the primitive is entered successfully, it is stored in the `data` field of `*shape_ptr` and the function returns 1. Otherwise, the function returns 0, leaving `*shape_ptr` unchanged.

8.14.2 Moving Primitives with the Mouse

```
int SL_MouseMoveSegment(
   SL_Segment  *seg_ptr           /* pointer to segment to move */
)
```

`SL_MouseMoveSegment()` allows the user to reposition an `SL_Segment` using the mouse. When this function is called, the segment `*seg_ptr` is moved, without changing length or direction, so its starting point coincides with the mouse pointer position. Then, as the mouse is moved within the canvas, the segment follows the mouse pointer with its starting point attached to the mouse pointer position. This continues until the user clicks one of the mouse buttons. If the left or middle button is clicked, the endpoints of the segment, translated so its starting point is at the mouse pointer position, are stored in `*seg_ptr` and the function returns 1. If the right mouse button is clicked at any time, the function returns 0, leaving `*seg_ptr` unchanged.

```
int SL_MouseMovePolyline(
   SL_Polyline *pline_ptr         /* pointer to polyline to move */
)
```

```
int SL_MouseMovePolygon(
   SL_Polygon  *pgon_ptr          /* pointer to polygon to move */
)
```

SL_MouseMovePolyline() and SL_MouseMovePolygon() allow the user to reposition SL_Polyline and SL_Polygons using the mouse. When this function is called, *pline_ptr (*pgon_ptr) is moved so its centroid coincides with the mouse pointer position. Then, as the mouse is moved within the canvas, the polyline (polygon) is translated to keep the centroid at the mouse pointer position. This continues until the user clicks one of the mouse buttons. If the left or middle button is clicked, the new vertices of the polyline (polygon), translated to have its centroid at the mouse pointer position, are stored in *pline_ptr (*pgon_ptr) and the function returns 1. If the right mouse button is clicked at any time, the function returns 0, leaving *pline_ptr (*pgon_ptr) unchanged.

```
int SL_MouseMoveArc(
   SL_Arc      *arc_ptr           /* pointer to arc to move */
)
```

```
int SL_MouseMoveCircle(
   SL_Circle   *circ_ptr          /* pointer to circle to move */
)
```

SL_MouseMoveArc() (SL_MouseMoveCircle()) allows the user to reposition an SL_Arc (SL_Circle) using the mouse. When the function is called, arc *arc_ptr (circle *circ_ptr) is moved so its center coincides with the mouse pointer position. Then, as the mouse is moved within the canvas, the arc (circle) follows the mouse pointer with its center attached to the mouse pointer position. This continues until the user clicks one of the mouse buttons. If the left or middle button is clicked, the mouse pointer position is stored as the new center of *arc_ptr (*circ_ptr), and the function returns 1. If the right mouse button is clicked at any time, the function returns 0, leaving *arc_ptr (*circ_ptr) unchanged.

```
int SL_MouseMoveShape(
   SL_Shape    *shape_ptr         /* pointer to shape to move */
)
```

SL_MouseMoveShape() allows the user to reposition an SL_Shape using the mouse. The argument to this function, *shape_ptr, must have had its type field set to one of the seven values of type SL_ShapeType (Section 7) before this function is called. If this value is either SL_POINT or SL_VECTOR, this function has no effect on its data field and returns 0. Otherwise, the function performs just as the corresponding primitive-moving functions previously described in this section. If the primitive is moved successfully (*i.e.*, the right mouse button is not clicked to abort the moving), it is stored in the data field of *shape_ptr and the function returns 1. Otherwise, the function returns 0, leaving *shape_ptr unchanged.

8.14.3 Changing Vertices and Endpoints with the Mouse

```
int SL_MouseMoveSegmentEndpoint(
   SL_Segment  *seg_ptr           /* pointer to segment which is to change */
)
```

SL_MouseMoveSegmentEndpoint() allows the user to reposition one endpoint of an SL_Segment using the mouse. When the function is called, the program waits for the user to click one of the mouse buttons near one of the segment's endpoints. If the left or middle button is pressed, the function finds the closer endpoint and assigns the mouse pointer position as the new location of that endpoint. The endpoint follows the mouse pointer as it is moved within the canvas until the user clicks a mouse button again. If the left or middle button is clicked, the position of the mouse pointer is stored in `*seg_ptr` as the new segment endpoint, and the function returns 1. If the user presses the right mouse button at any time, the function returns 0 leaving `*seg_ptr` unchanged.

```
int SL_MouseMovePolylineVertex(
   SL_Polyline *pline_ptr         /* pointer to polyline whose vertex is to change */
)


int SL_MouseMovePolygonVertex(
   SL_Polygon  *pgon_ptr          /* pointer to polygon whose vertex is to change */
)
```

`SL_MouseMovePolylineVertex()` (`SL_MouseMovePolygonVertex()` allows the user to reposition a single vertex of an `SL_Polyline` (`SL_Polygon`) using the mouse. When the function is called, the program waits for the user to click one of the mouse buttons near a vertex of the polyline `*pline_ptr` (polygon `*pgon_ptr`). If the left or middle mouse button is clicked, the function finds the vertex of `*pline_ptr` (`*pgon_ptr`) closest to the mouse pointer position and assigns the mouse pointer position as the new position of this vertex. The vertex follows the mouse pointer as it is moved on the canvas until the user clicks a mouse button again. If the left or middle button is clicked, the position of the mouse pointer is stored as the new position of the moved vertex in `*pline_ptr` (`*pgon_ptr`), and the function returns 1. If the user clicks the middle mouse button at any time, the function returns 0, leaving `*pline_ptr` (`*pgon_ptr`) unchanged.

8.14.4 Rotating with the Mouse

```
int SL_MouseRotatePolyline(
   SL_Polyline *pline_ptr         /* pointer to polyline to rotate */
)


int SL_MouseRotatePolygon(
   SL_Polygon  *pgon_ptr          /* pointer to polygon to rotate */
)
```

`SL_MouseRotatePolyline()` (`SL_MouseRotatePolygon()`) allows the user to rotate an `SL_Polyline` (`SL_Polygon`) using the mouse. When the function is called, the polar angle made by the vector from `*pline_ptr`'s (`*pgon_ptr`'s) centroid to the mouse pointer position is measured and all vertices are rotated by this angle about the centroid. Then, as the mouse pointer is moved in the canvas, the polyline (polygon) continues to rotate by the angle dictated by the pointer position until the user clicks a mouse button again. If the left or middle button is pressed, the rotated vertices corresponding to the current pointer position are stored in `*pline_ptr` (`*pgon_ptr`) and the function returns 1. If the user presses the right mouse button at any time, the function returns 0 leaving `*pline_ptr` (`*pgon_ptr`) unchanged.

```
int SL_MouseRotateArc(
   SL_Arc      *arc_ptr           /* pointer to arc to rotate */
)
```

`SL_MouseRotateArc()` allows the user to rotate an `SL_Arc` using the mouse. When the function is called, the polar angle made by the vector from the arc's center to the mouse pointer position is measured. This becomes the new starting angle for the arc. Then, as the mouse pointer is moved in the canvas, the arc continues to rotate by the angle dictated by the pointer position until the user clicks a mouse button again. If the left or middle button is pressed, the angles for the rotated arc corresponding to the current pointer position are stored in `*arc_ptr` and the function returns 1. If the user presses the right mouse button at any time, the function returns 0 leaving `*arc_ptr` unchanged.

```
int SL_MouseRotateShape(
   SL_Shape    *shape_ptr         /* pointer to shape to rotate */
)
```

`SL_MouseRotateShape()` allows the user to rotate an `SL_Shape` using the mouse. The argument to this function, `*shape_ptr`, must have had its `type` field set to one of the seven values of type `SL_ShapeType` (Section 7) before this function is called. If this value is other than `SL_POLYGON`, `SL_POLYLINE` or `SL_ARC`, this function has no effect on its data field and returns 0. Otherwise, the function performs just as the corresponding primitive-rotating functions previously described in this section. If the primitive is rotated successfully (*i.e.*, the right mouse button is not clicked to abort the rotation), it is stored in the `data` field of `*shape_ptr` and the function returns 1. Otherwise, the function returns 0, leaving `*shape_ptr` unchanged.

8.14.5 Scaling with the Mouse

```
int SL_MouseScalePolyline(
   SL_Polyline *pline_ptr         /* pointer to polyline to scale */
)
```

```
int SL_MouseScalePolygon(
   SL_Polygon  *pgon_ptr          /* pointer to polygon to scale */
)
```

`SL_MouseScalePolyline()` (`SL_MouseScalePolygon()` allows the user to scale an `SL_Polyline` (`SL_Polygon`) using the mouse. When the function is called, the distance between the current mouse pointer position and the polyline's (polygon's) centroid is measured. This distance divided by the average distance to any polyline (polygon) vertex from the centroid is used as the scaling factor for the polyline (polygon). As the mouse is moved in the canvas, this scaling factor is recomputed and the polyline (polygon) grows and shrinks accordingly. This continues until the user clicks a mouse button again. If the left or middle button is pressed, the scaled polyline (polygon) vertices determined by the current pointer position are stored in `*pline_ptr` (`*pgon_ptr`) and the function returns 1. If the user presses the right mouse button at any time, the function returns 0 leaving `*pline_ptr` (`*pgon_ptr`) unchanged.

```
int SL_MouseScaleArc(
   SL_Arc      *arc_ptr           /* pointer to arc to scale */
)
```

```
int SL_MouseScaleCircle(
   SL_Circle   *circ_ptr          /* pointer to circle to scale */
)
```

`SL_MouseScaleArc()` (`SL_MouseScaleCircle()`) allows the user to scale an `SL_Arc` (`SL_Circle`) using the mouse. When the function is called, the distance between the current mouse pointer position and the arc's (circle's) center is measured. This distance is assigned as the new arc (circle) radius. As the mouse is moved in the canvas, the arc (circle) grows and shrinks depending on the relative positions of the mouse pointer and the arc's (circle's) center. This continues until the user clicks a mouse button again. If the left or middle button is pressed, the radius of the scaled arc (circle) determined by the current pointer position is stored in `*arc_ptr` (`*circ_ptr`) and the function returns 1. If the user presses the right mouse button at any time, the function returns 0 leaving `*arc_ptr` (`*circ_ptr` unchanged.

```
int SL_MouseScaleShape(
   SL_Shape    *shape_ptr         /* pointer to shape to scale */
)
```

SL_MouseScaleShape() allows the user to scale an SL_Shape using the mouse. The argument to this function, *shape_ptr, must have had its type field set to one of the seven values of type SL_ShapeType (Section 7) before this function is called. If this value is other than SL_POLYGON, SL_POLYLINE, SL_CIRCLE or SL_ARC, this function has no effect on its data field and returns 0. Otherwise, the function performs just as the corresponding primitive-scaling functions previously described in this section. If the primitive is scaled successfully (*i.e.*, the right mouse button is not clicked to abort the scaling), it is stored in the data field of *shape_ptr and the function returns 1. Otherwise, the function returns 0, leaving *shape_ptr unchanged.

8.15 Obstacle Database Functions

The functions in this section are used to interact with the obstacle database, which is enabled by passing a 1 as the third argument to SL_Init() (Sections 3 and 8.12). When the database is enabled, an Obstacle menu button appears in the top row of the user interface next to the Go button. This menu allows the user to manipulate (add, delete, move, scale, etc.) the primitives included in the database. These primitives are automatically drawn on the canvas whenever the scene is redrawn and are not generally available for manipulation within a program, except through the following functions. Provided in this section of the library are functions for saving and loading obstacles to and from files or lists and for testing for intersections or collisions with the obstacles in the database.

For the following three functions, the file format used is the shape file format described in Section 8.5).

```
void SL_ObstSaveFile(
   char        *filename          /* name of file in which to save obstacles */
)
```

SL_ObstSaveFile() saves the current set of obstacles in the named file. If the file already exists, it will be overwritten.

```
int SL_ObstLoadFile(
   char        *filename          /* name of file from which to load obstacles */
)
```

SL_ObstLoadFile() loads a set of obstacles from a named file, replacing the current set of obstacles in the database, if any. If the named file (filename) is unreadable or does not exist the function returns 0, leaving the obstacle database unchanged. Otherwise, the function returns 1.

```
int SL_ObstMergeFile(
   char        *filename          /* name of file from which to merge obstacles */
)
```

SL_ObstMergeFile() loads a set of obstacles from a named file, and adds them to the current set of obstacles in the database. If the named file (filename) is unreadable or does not exist the function returns 0, leaving the obstacle database unchanged. Otherwise, the function returns 1.

```
void SL_ObstLoadShapes(
   SL_List     *shape_list        /* list of new obstacle shapes */
)
```

SL_ObstLoadShapes() transfers a set of shapes stored in *shape_list to the obstacle database, replacing the current set of obstacles in the database.

```
SL_List SL_ObstRetrieveShapes(void)
```

SL_ObstRetrieveShapes() copies the shapes in the obstacle database into an SL_List of SL_Shapes. If there are no obstacles in the database, the return value of this function is NULL.

```
void SL_ObstDeleteAll(void)
```

SL_ObstDeleteAll() removes all existing obstacles from the database.

```
int SL_InsidePointObst(
   SL_Point    *pnt_ptr           /* pointer to point to test for inclusion */
)
```

SL_InsidePointObst() determines if the point *pnt_ptr is inside any of the obstacles in the database, returning 1 if it is and 0 otherwise.

The following functions determine if a given library primitive (segment, polyline, polygon, arc, circle, or shape) intersects any of the obstacles in the database. If it does, the function returns 1; otherwise it returns 0.

```
int SL_InterSegmentObstCheck(
   SL_Segment  *seg_ptr           /* pointer to segment to test for intersection */
)


int SL_InterPolylineObstCheck(
   SL_Polyline *pline_ptr         /* pointer to polyline to test for intersection */
)


int SL_InterPolygonObstCheck(
   SL_Polygon  *pgon_ptr          /* pointer to polygon to test for intersection */
)


int SL_InterArcObstCheck(
   SL_Arc      *arc_ptr           /* pointer to arc to test for intersection */
)


int SL_InterCircleObstCheck(
   SL_Circle   *circ_ptr          /* pointer to circle to test for intersection */
)


int SL_InterShapeObstCheck(
   SL_Shape    *shape_ptr         /* pointer to shape to test for intersection */
)


int SL_CollideShapeObst(
   SL_Shape    *shape_ptr         /* pointer to shape to test for collision */
)
```

SL_CollideShapeObst() determines if the shape *shape_ptr collides with any of the obstacles in the database. Collision is defined as one shape intersecting with another or one shape being inside the other. If the given shape collides with any of the obstacles the function returns 1, otherwise it returns 0.

```
double SL_ObstRayHit(
   SL_Point    *ray_from_ptr      /* pointer to base of ray */
   SL_Point    *ray_to_ptr        /* pointer to final point of the ray */
)
```

SL_ObstRayHit() indicates where along a ray being fired from *ray_from_ptr to *ray_to_ptr the first obstacle intersection occurs. Note that the ray does not extend beyond the point *ray_to_ptr. The return value of this function is a value in the range $[0.0, 1.0]$ indicating the linear displacement from *ray_from_ptr toward *ray_to_ptr of the first obstacle intersection point. For example, a return value of 0.5 indicates the ray first hits an obstacle at its midpoint. If the ray hits no obstacle or is of length 0, the function returns 1.0.

8.16 Bug Algorithm Functions

The following functions are used to implement the sensor-based motion planning algorithms Bug2 and Visbug21 for bug-like point automata moving in the plane. These algorithms are designed to move a point robot from a starting point to a target point in an unknown environment cluttered with obstacles. The same algorithm may be used to move nonpoint robots if the algorithm is implemented in the robot's configuration space. This functionality is provided here as well by simply allowing the programmer to specify the function that determines if a collision with an obstacle has occurred. For nonpoint robots, the bug parameters (starting point, target point, etc.) are given in configuration-space coordinates. The collision function tests for the collision of the entire nonpoint robot structure with the obstacles. If the robot in the given configuration collides with an obstacle, then the point robot moving in the robot's configuration space has collided with a configuration-space obstacle. See Section 4 for an example of how these functions are used with nonpoint robots.

```
SL_Bug SL_Bug2Init(
   SL_Point    *start_ptr,        /* the starting point for the bug (robot) */
   SL_Point    *target_ptr,       /* the target point for the robot */
   int         left_loc_dir,      /* 1/0 of local turning direction is left/right */
   double      step_size,         /* size of the step to be taken by the robot */
   int         (*coll_fn)(SL_Segment *)
                                  /* the function used to check for collision
                                     of robot with obstacles in environment */
)
```

```
SL_Bug SL_Visbug21Init(
   SL_Point    *start_ptr,        /* the starting point for the bug (robot) */
   SL_Point    *target_ptr,       /* the target point for the robot */
   int         left_loc_dir,      /* 1/0 if local turning direction is left/right */
   double      step_size,         /* size of the step to be taken by the robot */
   double      vision_rad,        /* the robot's radius of vision */
   int         (*coll_fn)(SL_Segment *)
                                  /* the function used to check for collision
                                     of robot with obstacles in environment */
)
```

`SL_Bug2Init()` initializes a data structure that represents a bug, or point automaton, that may be moved in the simulation environment in accordance with the Bug2 algorithm [1]. The bug is initialized to have a starting point (`*start_ptr`) and a target point (`*target_ptr`). In addition, a local turning direction in which the bug will turn to move around an obstacle is established by the flag `left_loc_dir`. If this flag is 1, the robot will turn left; otherwise the robot will turn right. The distance the bug travels in a single step is initialized to `step_size`. Finally, `coll_fn()` is a function that takes as an argument a pointer to an `SL_Segment` and returns 1 or 0 depending on if there would be a collision with an obstacle if the bug move from the starting point to the ending point of this segment.

`SL_Visbug21Init()` similarly initializes a data structure that represents a point automaton that may be moved in accordance with the Visbug21 algorithm [2]. The parameter `vision_rad` is the robot's radius of vision.

```
void SL_BugReset(
   SL_Bug      the_bug,           /* the bug that is to be reset */
   SL_Point    *new_start_ptr,    /* new starting point for the robot */
   SL_Point    *new_target_ptr    /* new target point for the robot */
)
```

`SL_BugReset()` establishes a new start and target point for `the_bug` and places `the_bug` at the point `*new_start_ptr`.

```
void SL_BugGetPos(
   SL_Bug      the_bug,           /* the bug whose position is of interest */
   SL_Point    *pos_ptr           /* the current position of the robot */
)
```

SL_BugGetPos() returns through *pos_ptr the current position of the_bug.

```
void SL_BugSetPos(
   SL_Bug      the_bug,           /* the bug for which to set position */
   SL_Point    *new_pos_ptr       /* the new position of the robot */
)
```

SL_BugSetPos() places the_bug at the position *new_pos_ptr.

```
void SL_BugSetLocDir(
   SL_Bug      the_bug,           /* the bug for which to set local direction */
   int         left_loc_dir       /* 1 if new direction is to be left; 0 if right */
)
```

SL_BugSetLocDir() establishes a new local turning direction for the_bug. If the parameter left_loc_dir is passed a nonzero value, the local turning direction is set to Left. Otherwise, the local turning direction is set to Right.

```
void SL_BugSetStepSize(
   SL_Bug      the_bug,           /* the bug for which to set step size */
   double      step_size          /* the new step size for the robot */
)
```

SL_BugSetStepSize() establishes step_size as the new distance the_bug will move in a single step.

```
int SL_BugSetVisionRaidus(
   SL_Bug      the_bug,           /* the bug for which to set radius of vision */
   double      vision_rad         /* the new radius of vision for the robot */
)
```

SL_BugSetVisionRadius() establishes vision_rad as the new radius of vision for the_bug, assuming it was created by a call to SL_Visbug21Init(). If the_bug was created by a call to the function SL_Bug2Init(), it has no radius of vision. This function, therefore, has no effect in this case.

```
int SL_BugStep(
   SL_Bug      the_bug            /* the bug that is to move a single step */
)
```

SL_BugStep() moves the_bug a single step according to either the Bug2 or VisBug21 algorithm. Which of these algorithms is used depends on whether SL_Bug2Init() or SL_Visbug21Init() was used to create the_bug.

```
void SL_BugDrawPath(
   SL_Bug      the_bug            /* the bug whose path is to be drawn */
)
```

SL_BugDrawPath() draws a series of points on the drawing canvas that represents the path of the_bug from its starting point to its current position.

8.17 Homogeneous Matrix Functions (SL_Transf Functions)

The following set of functions are used for manipulating homogeneous transformation matrices and are part of the ADT `SL_Transf`.

```
void SL_TransfCopy(
   SL_Transf   t_from,            /* the transformation matrix to be copied */
   SL_Transf   t_to               /* the place to put the copied matrix */
)
```

`SL_TransfCopy()` makes a copy of a given transformation matrix.

```
void SL_TransfLoadIdentity(
   SL_Transf   the_transf         /* the place to put the identity matrix */
)
```

`SL_TransfLoadIdentity()` loads the identity transformation into a given matrix.

```
void SL_TransfLoadTranslation(
   double      x,                 /* the amount of translation in the x direction */
   double      y,                 /* the amount of translation in the y direction */
   SL_Transf   the_transf         /* the transformation representing the translation */
)
```

`SL_TransfLoadTranslation()` loads the transformation matrix corresponding to a given xy translation into a given matrix (`the_transf`).

```
void SL_TransfLoadScale(
   double      scale,             /* the scaling factor */
   SL_Transf   the_transf         /* the transformation representing the scaling */
)
```

`SL_TransfLoadScale()` loads the transformation matrix corresponding to a given scaling factor (`scale`) into a give matrix (`the_transf`).

```
void SL_TransfLoadRotation(
   double      rad_angle,         /* the angle by which to rotate */
   SL_Transf   the_transf         /* the transformation representing the rotation */
)
```

`SL_TransfLoadRotation()` loads into a given matrix (`the_transf`) the transformation matrix corresponding to a rotation through a certain angle given in radians (`rad_angle`).

```
void SL_TransfLoadRotationSinCos(
   double      sin,               /* sine of the angle by which to rotate */
   double      cos,               /* cosine of the angle by which to rotate */
   SL_Transf   the_transf         /* the transformation representing the rotation */
)
```

`SL_TransfLoadRoatationSinCos()` loads into a given matrix (`the_transf`) the transformation matrix corresponding to a rotation through a certain angle, the sine and cosine of which are given.

```
void SL_TransfPostMultiply(
   SL_Transf   t1,                /* the first transformation matrix */
   SL_Transf   t2,                /* the second transformation matrix */
   SL_Transf   t1_times_t2        /* the product matrix */
)
```

SL_TransfPostMultiply() creates a new transformation matrix (t1_times_t2) that is the result of the multiplication of two other matrices (t1t2).

```
void SL_TransfPostMultiplyD(
   SL_Transf   t1_in_out,         /* the first transformation matrix and
                                     then the product */
   SL_Transf   t2                 /* the second transformation matrix */
)
```

SL_TransfPostMultiplyD() multiplies two transformation matrices, t1_in_out and t2 and replaces the first matrix with the resulting matrix (t1_in_outt2).

```
void SL_TransfTranslate(
   double      x,                 /* the amount to translate in the x direction */
   double      y,                 /* the amount to translate in the y direction */
   SL_Transf   the_transf         /* the transformation to change */
)
```

SL_TransfTranslate() alters a given transformation matrix to reflect a translation in the xy-plane.

```
void SL_TransfScale(
   double      scale,             /* the scaling factor */
   SL_Transf   the_transf         /* the transformation to change */
)
```

SL_TransfScale() alters a given transformation matrix to reflect a uniform scaling by a given factor.

```
void SL_TransfRotate(
   double      rad_angle,         /* the radian angle by which to rotate */
   SL_Transf   the_transf         /* the transformation to change */
)
```

SL_TransfRotate() alters a given transformation matrix to reflect a rotation by a certain angle given in radians.

```
void SL_TransfRotateSinCos(
   double      the_sin,           /* the sine of the angle by which to rotate */
   double      the_cos,           /* the cosine of the angle by which to rotate */
   SL_Transf   the_transf         /* the transformation to change */
)
```

SL_TransfRotateSinCos() alters a given transformation matrix to reflect a rotation by a certain angle, the sine and cosine of which are given.

```
void SL_TransfGetTranslation(
   SL_Transf   the_transf,        /* the transformation matrix in question */
   SL_Vector   *origin_ptr        /* pointer to place to store translated origin */
)
```

SL_TransfGetTranslation() extracts from a given transformation matrix the translation vector encoded there. This may also be seen as the origin of the transformed coordinate system. The translation information is stored in *origin_ptr.

```
double SL_TransfGetScale(
   SL_Transf   the_transf         /* the transformation matrix in question */
)
```

`SL_TransfGetScale()` extracts from the given transformation matrix the scaling factor encoded there. This scaling factor is given as the return value of the function.

```
void SL_TransfGetRotationSinCos(
   SL_Transf   the_transf,        /* the transformation matrix in question */
   double      *sin_ptr,          /* pointer to the sine of the rotation angle */
   double      *cos_ptr           /* pointer to the cosine of the rotation angle */
)
```

`SL_TransfGetRotationSinCos()` extracts from the given transformation matrix the sine and cosine of the rotation angle encoded there.

```
double SL_TransfGetRotation(
   SL_Transf   the_transf         /* the transformation matrix in question */
)
```

`SL_TransfGetRotation()` extracts from the given transformation matrix the rotation angle encoded there. This angle, given in radians, is the return value of the function.

```
void SL_TransfTimesPoint(
   SL_Transf   the_transf,        /* the transformation matrix */
   SL_Point    *point_ptr,        /* the point to transform */
   SL_Point    *result_ptr        /* the transformed point */
)
```

`SL_TransfTimesPoint()` transforms the position of a point (`*point_ptr`) using a given transformation matrix. The transformed point is given in `*result_ptr`.

```
void SL_TransfGetWorld(
   SL_Transf   the_transf,        /* the transformation matrix */
   SL_Point    *local_ptr,        /* the point in the local coordinate system */
   SL_Point    *world_ptr         /* the point in the world coordinate system */
)
```

`SL_TransfGetWorld()` converts a point given in a local coordinate system (`*local_ptr`) as represented by the given transformation matrix (`the_transf`) to one in the world coordinate system (`*world_ptr`). (Note that this function is identical to `SL_TransfTimesPoint()`. I'm not sure why we have both.)

```
void SL_TransfGetLocal(
   SL_Transf   the_transf,        /* the transformation matrix */
   SL_Point    *world_ptr,        /* the point in the world coordinate system */
   SL_Point    *local_ptr         /* the point in the local coordinate system */
)
```

`SL_TransfGetLocal()` coverts a point given in the world coordinate system (`*world_ptr`) to one in the local coordinate system (`*local_ptr`), represented by the given transformation matrix.

```
void SL_TransfConvert(
   SL_Transf   source_transf,     /* the transformation to convert from */
   SL_Transf   target_transf,     /* the transformation to convert to */
   SL_Point    *source_ptr,       /* pointer to the point to convert */
   SL_Point    *target_ptr        /* pointer to the converted point */
)
```

`SL_TransfConvert()` converts the coordinates of a point (`*source_ptr`) given in one coordinate system (`source_transf`) to coordinates in a second coordinate system (`target_transf`).

```
void SL_TransfInvert(
   SL_Transf   the_transf,        /* the transformation to invert */
   SL_Transf   the_transf_inv     /* the inverted transformation */
)
```

`SL_TrasfInvert()` inverts a given transformation matrix (`the_transf`). The inverted matrix is stored in `the_transf_inv`. Any point that is transformed first by `the_transf` and then by `the_transf_inv` will not change.

8.18 Modeling Functions

The functions described in this section are modeling transformation functions. These functions use the homogeneous matrix transformation functions described in Section 8.17 to maintain a single matrix, the current transformation matrix (CTM), that encodes a transformed model coordinate system. The CTM may be thought of as a moving and rotating xy-coordinate system. Each new transformation should always be specified with respect to this moving coordinate system. Further, the drawing functions of Section 8.13 assume that the primitives to be drawn are specified in the CTM coordinate system. When these modeling functions are not used in a program, the CTM is by default the identity matrix.

The functions `SL_ModelPush()` and `SL_ModelPop()` allow for the maintenance of a modeling transformation stack (MTS), which is simply a sequence of CTMs. This is useful for manipulating geometric structures that are naturally hierarchical. See Section 5 for an example of how these functions are used in a program.

```
void SL_ModelIdentity(void)
```

`SL_ModelIdentity()` sets the CTM to the identity matrix.

```
void SL_ModelTranslate(
   double      x,                 /* the amount to translate in the x direction */
   double      y                  /* the amount to translate in the y direction */
)
```

`SL_ModelTranslate()` postmultiplies the CTM by a homogeneous transformation matrix representing a translation in the xy-plane.

```
void SL_ModelRotate(
   double      angle              /* the radian angle by which to rotate */
)
```

`SL_ModelRotate()` postmultiplies the CTM by a homogeneous transformation matrix representing a rotation in the xy-plane through a certain angle given in radians.

```
void SL_ModelRotateSinCos(
   double      the_sin,           /* the sine of the angle by which to rotate */
   double      the_cos            /* the cosine of the angle by which to rotate */
)
```

`SL_ModelRotateSinCos()` postmultiplies the CTM by a homogeneous transformation matrix representing a rotation in the xy-plane through a certain angle, the sine and cosine of which are given.

```
void SL_ModelScale(
   double      scale              /* the scaling factor */
)
```

`SL_ModelScale()` postmultiplies the CTM by a homogeneous transformation matrix representing a scaling of the coordinate system by a certain factor.

```
void SL_ModelPush(void)
```

`SL_ModelPush()` pushes the CTM on the top of the MTS. The CTM remains unchanged.

```
void SL_ModelPop(void)
```

`SL_ModePop()` removes the top of the MTS and stores in the CTM.

```
void SL_ModelGet(
   SL_Transf   the_transf         /* place to store the CTM */
)
```

`SL_ModelGet()` retrieves the CTM and stores it in `the_transf`.

```
void SL_ModelSet(
   SL_Transf   the_transf         /* the matrix to which the CTM should be set */
)
```

`SL_ModelSet()` replaces the CTM with a given transformation matrix.

```
void SL_ModelPostMultiply(
   SL_Transf   the_transf         /* the matrix by which to postmultiply the CTM */
)
```

`SL_ModelPostMultiply()` postmultiplies the CTM by a given transformation matrix. The result of the multiplication is stored as the new CTM.

```
void SL_ModelGetTranslation(
   SL_Vector   *transl_ptr        /* pointer to place to store translation vector */
)
```

`SL_ModelGetTransation()` retrieves the translation vector encoded in the CTM (*i.e*, the origin of the transformed coordinate system) and stores it in `*transl_ptr`.

```
double SL_ModelGetRotation(void)
```

`SL_ModelGetRotation()` retrieves the rotation angle encoded in the CTM. The radian measurement is returned as the function's value.

```
void SL_ModelGetRotationSinCos(
   double      *sin_ptr,          /* pointer to place to store sine of rot. angle */
   double      *cos_ptr           /* pointer to place to store cosine of rot. angle */
)
```

`SL_ModelGetRotateSinCos()` retrieves the sine and cosine of the rotation angle encoded in the CTM.

```
double SL_ModelGetScale(void)
```

`SL_ModeGetScale()` retrieves the scaling factor encoded in the CTM.

The following functions find the word-coordinate representations of library primitives that are given in CTM coordinates.

```
void SL_ModelGetWorldPoint(
   SL_Point    *local_ptr,        /* pointer to point in CTM coordinates */
   SL_Point    *world_ptr         /* pointer to place to store world-coord. point */
)
```

SL_ModelGetWorldPoint() may also be used to find the world coordinates of an SL_Vector given in CTM coordinates.

```
void SL_ModelGetWorldSegment(
   SL_Segment  *local_ptr,        /* pointer to segment in CTM coordinates */
   SL_Segment  *world_ptr         /* pointer to place to store world-coord. segment */
)
```

```
void SL_ModelGetWorldPolygon(
   SL_Polygon  *local_ptr,        /* pointer to polygon in CTM coordinates */
   SL_Polygon  *world_ptr         /* pointer to place to store world-coord. polyon */
)
```

SL_ModelGetWorldPolygon() may also be used to find the world coordinates of an SL_Polyline given in CTM coordinates.

```
void SL_ModelGetWorldArc(
   SL_Arc      *local_ptr,        /* pointer to arc in CTM coordinates */
   SL_Arc      *world_ptr         /* pointer to place to store world-coord. arc */
)
```

```
void SL_ModelGetWorldCircle(
   SL_Circle   *local_ptr,        /* pointer to circle in CTM coordinates */
   SL_Circle   *world_ptr         /* pointer to place to store world-coord. circle */
)
```

```
void SL_ModelGetWorldShape(
   SL_Shape    *local_ptr,        /* pointer to shape in CTM coordinates */
   SL_Shape    *world_ptr         /* pointer to place to store world-coord. shape */
)
```

The following functions find the CTM-coordinate-system representation of library primitives given in world coordinates.

```
void SL_ModelGetLocalPoint(
   SL_Point    *world_ptr,        /* pointer to point in world coordinates */
   SL_Point    *local_ptr         /* pointer to place to store CTM-coord. point */
)
```

SL_ModelGetLocalPoint() may also be used to find the CTM coordinate system representation of an SL_Vector.

```
void SL_ModelGetLocalSegment(
   SL_Segment  *world_ptr,        /* pointer to segment in world coordinates */
   SL_Segment  *local_ptr         /* pointer to place to store CTM-coord. segment */
)
```

```
void SL_ModelGetLocalPolygon(
   SL_Polygon  *world_ptr,        /* pointer to polygon in world coordinates */
   SL_Polygon  *local_ptr         /* pointer to place to store CTM-coord. polygon */
)
```

`SL_ModelGetLocalPolygon()` may also be used to find the CTM coordinate system representation of an `SL_Polyline`.

```
void SL_ModelGetLocalArc(
   SL_Arc      *world_ptr,        /* pointer to arc in world coordinates */
   SL_Arc      *local_ptr         /* pointer to place to store CTM-coord. arc */
)


void SL_ModelGetLocalCircle(
   SL_Circle   *world_ptr,        /* pointer to circle in world coordinates */
   SL_Circle   *local_ptr         /* pointer to place to store CTM-coord. circle */
)


void SL_ModelGetLocalShape(
   SL_Shape    *world_ptr,        /* pointer to shape in world coordinates */
   SL_Shape    *local_ptr         /* pointer to place to store CTM-coord. shape */
)
```

8.19 `SL_Frame` Functions

The functions described in this section allow the programmer to create geometric structures in a hierarchical manner through the use of *frames*. A frame is a coordinate system that has a number of geometric primitives (its *fixtures*) associated with it. Each frame is assumed to be the location of a joint in the structure. This joint may be either fixed (implying no articulation), prismatic, or revolute. Frames may be created in a hierarchical manner, with each frame (except a root frame) having an associated parent and any number of child frames. The coordinate transformation for each frame is defined relative to its parent frame. Drawing or manipulation of a parent frame automatically causes its children to be drawn or appropriately changed to reflect the parent's movement. See Section 6 for an example program that uses these functions.

```
SL_Frame SL_FrameCreate(
   SL_Frame      the_parent,      /* parent frame associated with new frame */
   SL_FrameJoint j_type,          /* type of joint located at frame's origin */
   int           draw_axes        /* 1/0 if frame axes are/are not to be drawn */
)
```

`SL_FrameCreate()` creates a new `SL_Frame` and returns it as its value. The frame is placed in the hierarchy as a child of `the_parent`. If this frame is to be the root of the hierarchy, `the_parent` should be NULL. The type of joint located at the origin of this frame is specified by the parameter `j_type` (Section 7). If the frame axes are to be drawn with each call to `SL_DrawFrame()`, `draw_axes` should be 1; otherwise, it should be 0. After this function is called, the frame created is considered to be "open" and is ready to be repositioned and have fixtures attached to it.

```
void SL_FrameClose(void)
```

`SL_FrameClose()` closes the most recently created frame, which means that no more changes (either transformations or additions of fixtures) are to be made to this frame. Each call to `SL_FrameCreate()` must be paired with a call to `SL_FrameClose()`. In between these two calls are the calls that transform the frame and add fixtures to it.

```
void SL_FrameIdentity(void)
```

`SL_FrameIdentity()` sets the currently open frame's modeling transformation matrix to the identity.

```
void SL_FrameTranslate(
   double      x,                 /* the amount to translate in the x direction */
   double      y                  /* the amount to translate in the y direction */
)
```

`SL_FrameTranslate()` translates the currently open frame by a given amount in the xy-plane.

```
void SL_FrameRotate(
   double      rad_angle          /* the radian angle by which to rotate */
)
```

`SL_FrameRotate()` rotates the currently open frame by a certain angle, given in radians.

```
void SL_FrameRotateSinCos(
   double      the_sin,           /* sine of angle by which to rotate */
   double      the_cos            /* cosine of angle by which to rotate */
)
```

`SL_FrameRotateSinCos()` rotates the currently open frame by a certain angle, the sine and cosine of which are given.

```
void SL_FrameScale(
   double      scale              /* the scaling factor */
)
```

`SL_FrameScale()` scales the currently open frame by a given scaling factor.

```
void SL_FrameSet(
   SL_Transf   the_transf         /* the new modeling transformation matrix */
)
```

`SL_FrameSet()` sets the currently open frame's modeling transformation matrix to a given transformation matrix (`the_transf`).

```
void SL_FramePostMultiply(
   SL_Transf   the_transf         /* the transformation matrix by which to multiply */
)
```

`SL_FramePostMultiply()` postmultiplies the currently open frame's modeling transformation matrix by a given transformation matrix (`the_transf`).

```
void SL_FrameAddCircle(
   SL_Circle   *circ_ptr,         /* pointer to the circle to add to frame */
   int         check_collision,   /* indicates if circle should be checked for
                                     collisions with obstacles */
   char        *draw_color,       /* name of color with which to draw circle */
   int         draw_solid         /* indicates if circle should be filled or not */
)


void SL_FrameAddPolygon(
   SL_Polygon  *poly_ptr,         /* pointer to polygon to add to frame */
   int         check_collision,   /* indicates if polygon should be checked for
                                     collisions with obstacles */
   char        *draw_color,       /* name of color with which to draw polygon */
   int         draw_solid         /* indicates if polygon should be filled or not */
)
```

```
void SL_FrameAddShape(
   SL_Shape     *shape_ptr,         /* pointer to shape to add to frame */
   int          check_collision,    /* indicates if shape should be checked for
                                       collisions with obstacles */
   char         *draw_color,        /* name of color with which to draw shape */
   int          draw_solid          /* indicates if shape should be filled or not */
)
```

The above three functions are used to add fixtures to the frame. The fixture is a library primitive specified in the current frame's coordinate system. If the value of `check_collision` is 1, the fixture will be tested for collision with obstacles in the environment when `SL_CollideFrameObst()` is called or for the given shape when `SL_CollideFrameShape()` is called; otherwise it will not be checked for collisions. When `SL_FrameDraw()` is called for the fixture's frame hierarchy, the primitive will be drawn in the color `draw_color`. If `draw_solid` is 1 (and the primitive is either an arc, circle, or polygon), it will be filled; if it is 0, the primitive will be drawn as an outline only.

```
void SL_FrameGet(
   SL_Frame     the_frame,          /* the frame in question */
   SL_Transf    the_transf          /* frame's transformation relative to its parent */
)
```

`SL_FrameGet()` retrieves a given frame's transformation matrix relative to its parent and places it in `the_transf`.

```
void SL_FrameGetTranslation(
   SL_Frame     the_frame,          /* the frame in question */
   SL_Vector    *transl_ptr         /* the translation vector */
)
```

`SL_FrameGetTranslation()` retrieves the translation vector of a given frame relative to its parent frame. The vector is returned in `*transl_ptr`.

```
double SL_FrameGetRotation(
   SL_Frame     the_frame           /* the frame in question */
)
```

`SL_FrameGetRotation()` retrieves the rotation angle of a given frame relative to its parent frame. This radian angle measurement is returned as the function's value.

```
void SL_FrameGetRotationSinCos(
   SL_Frame     the_frame,          /* the frame in question */
   double       *sin_ptr,           /* the sine of the frame's rotation angle */
   double       *cos_ptr            /* the cosine of the frame's rotation angle */
)
```

`SL_FrameGetRotationSinCos()` retrieves the sine and cosine of a given frame's rotation angle.

```
void SL_FrameGetWorld(
   SL_Frame     the_frame,          /* the frame in question */
   SL_Transf    the_transf          /* the transformation from the world origin
                                       to the frame origin */
)
```

`SL_FrameGetWorld()` retrieves the transformation matrix of a given frame relative to the world origin.

```
void SL_FrameGetWorldTranslation(
   SL_Frame     the_frame,          /* the frame in question */
   SL_Vector    *transl_ptr         /* pointer to the translation vector */
)
```

SL_FrameGetWorldTranslation() retrieves the translation vector of a given frame relative to the world origin. The vector is return in *transl_ptr.

```
double SL_FrameGetWorldRotation(
   SL_Frame    the_frame          /* the frame in question */
)
```

SL_FrameGetWorldRotation() retrieves the rotation angle of a given frame relative to the world origin. This radian angle measurement is returned as the function's value.

```
void SL_FrameGetWorldRotationSinCos(
   SL_Frame    the_frame,         /* the frame in question */
   double      *sin_ptr,          /* the sine of the frame's rotation angle */
   double      *cos_ptr           /* the cosine of the frame's rotation angle */
)
```

SL_FrameGetWorldRotationSinCos() retrieves the sine and cosine of a given frame's rotation angle.

```
void SL_FrameGetWorldPoint(
   SL_Frame    the_frame,         /* the frame in which *local_ptr is defined */
   SL_Point    *local_ptr,        /* pointer to a point defined in the_frame's
                                     coordinate system */
   SL_Point    *world_ptr         /* pointer to place to store point in
                                     world coordinates */
)
```

SL_FrameGetWorldPoint() returns the world coordinates of a point that is specified in a given frame's coordinate system.

```
void SL_FrameGetLocalPoint(
   SL_Frame    the_frame,         /* the frame in question */
   SL_Point    *world_ptr,        /* pointer to point defined in world coordinates */
   SL_Point    *local_ptr         /* pointer to place to store point in
                                     frame coordinates */
)
```

SL_FrameGetLocalPoint() returns the coordinates of a point specified in world coordinates converted to coordinates in a given frame's coordinate system.

```
void SL_FrameConvertPoint(
   SL_Frame    source_frame,      /* the frame to convert from */
   SL_Frame    target_frame,      /* the frame to convert to */
   SL_Point    *source_ptr,       /* pointer to the point to convert */
   SL_Point    *target_ptr        /* pointer to the converted point */
)
```

SL_FrameConvertPoint() converts the coordinates of a point (*source_ptr) given in one frame's coordinate system (source_frame) to coordinates in a second frame's coordinate system (target_frame).

```
double SL_FrameGetJoint(
   SL_Frame    the_frame          /* the joint's frame */
)
```

SL_FrameGetJoint() returns as its value the joint value of the joint associated with a given frame.

```
void SL_FrameSetJoint(
   SL_Frame    the_frame,         /* the joint's frame */
   double      j_value            /* the value to which to set the joint */
)
```

SL_FrameSetJoint() sets the joint value of given frame's joint to a specified quantity.

```
void SL_FrameJointDelta(
   SL_Frame    the_frame,         /* the joint's frame */
   double      j_delta            /* the amount by which to change the joint value */
)
```

SL_FrameJointDelta() changes the joint value of a given frame by a specified amount (j_delta).

```
void SL_FrameDraw(
   SL_Frame    the_frame          /* the frame to draw */
)
```

SL_FrameDraw() draws the hierarchy of shapes attached to a given frame. All fixtures attached to the current frame are drawn as well as the fixtures attached to any descendant frames (children, grandchildren, etc.).

```
int SL_CollideFrameShape(
   SL_Frame    the_frame,         /* the frame to check for collisions */
   SL_Shape    *shape_ptr         /* pointer to shape with which to check collisions */
)


int SL_CollideFrameObst(
   SL_Frame    the_frame          /* the frame to check for collisions */
)
```

SL_CollideFrameShape() (SL_CollideFrameObst()) checks if any of the fixtures attached to a given frame or its descendants that were marked as ones for which to check collisions (See the SL_FrameAdd*() routines.) collides with the given shape (any of the obstacles in the obstacle database). Two shapes are said to collide if they intersect or one is contained within the other. Note that the_frame must be a root frame (*i.e*, its parent must be NULL).

8.20 SL_List Functions

Described here are the functions for manipulating variables of type SL_List. Variables of this type are useful for storing collections of data elements that grow and shrink as the program runs or for which it is difficult to determine an upper bound on the number of elements. The functions that are available for this ADT allow elements to be added and deleted from lists at any position. Entire lists may be copied, concatenated, or destroyed, and their lengths may be queried. Single elements of a list may be copied with or without removing them from the list. A function is also provided that traverses an entire list and deletes all elements that satisfy a given predicate function. Functions are provided for traversing an entire list (either forward or backward) and applying a given function to each element of the list. For general traversal of a list, functions are available for indexing a list so it may be treated much like an array.

```
SL_List SL_ListInit(
   int         data_size          /* the size (in bytes) of the each list
                                     element's data field */
)
```

SL_ListInit() initializes a linked list that can hold elements of size data_size. The initialized list is provided as the function's return value. A call to this function must be made before calling any of the other functions described in this section. After this function is called, the list is empty.

```
SL_List SL_ListDup(
   SL_List     source_list        /* the list that is to be duplicated */
)
```

`SL_ListDup()` makes a copy of a given list. The duplicate list is given as the function's return value.

```
SL_List SL_ListReverse(
   SL_List     source_list        /* the list that is to be reversed */
)
```

`SL_ListReverse()` makes a copy of a given list such that the elements in the new copy of the list are in the reverse order of the original. The duplicate list is given as the function's return value.

```
SL_List SL_ListConcat(
   SL_List     list1,             /* the list after which list2 will be catenated */
   SL_List     list2              /* the list to concatenate after list1 */
)
```

`SL_ListConcat()` creates a new list with elements that are copies of those in `list1` and `list2`. In the new list, which is the function's return value, the elements of `list1` appear first in the same order as in `list1`. The elements of `list2` follow the `list1` elements and appear in the same order as the elements of `list2`. The effect is to concatenate the elements of `list1` and `list2`.

```
int SL_ListEmpty(
   SL_List     the_list           /* the list in question */
)
```

`SL_ListEmpty()` determines if a given list is empty. If `the_list` contains no elements, the function returns 1; otherwise it returns 0.

```
int SL_ListLength(
   SL_List     the_list           /* the list whose length is being determined */
)
```

`SL_ListLength()` determines the number of elements in a given list. The length of `the_list` is returned as the function's value.

```
int SL_ListFirst(
   SL_List     the_list           /* list from which first element is being copied */
   char        *data_ptr          /* pointer to where the list's first element
                                     should be copied */
)


int SL_ListLast(
   SL_List     the_list           /* list from which last element is being copied */
   char        *data_ptr          /* pointer to where the list's last element
                                     should be copied */
)


int SL_ListNth(
   SL_List     the_list,          /* list from which nth element is being copied */
   int         the_n,             /* number of element to be copied from the list
                                     (1 corresponds to the first element) */
   char        *data_ptr          /* pointer to where the list's element
                                     should be copied */
)
```

The above three functions copy the first, last, or n^{th} element, respectively, of a given list into a given location (`*data_ptr`). If the list is empty (or contains fewer than n elements for `SL_ListNth()`), the function returns 0, leaving `*data_ptr` unchanged. Otherwise, the function returns 1. The given list is unchanged by these functions.

```
char *SL_ListInsertNth(
   SL_List     the_list,          /* the list into which to insert an element */
   int         the_n,             /* the list index for the new element */
   char        *data_ptr          /* a pointer to the element to be inserted */
)
```

`SL_ListInsertNth()` inserts a new element (`*data_ptr`) as the n^{th} in `the_list`. The value of `the_n` must be between 1 and the number of elements in the list. If `the_n` is not in this range, the function returns NULL. Otherwise, the function returns a pointer to the new list element created.

```
int SL_ListDeleteNth(
   SL_List     the_list,          /* the list from which to delete an element */
   int         the_n,             /* the list index of the element to be deleted */
   char        *data_ptr          /* pointer to place to store deleted element */
)
```

`SL_ListDeleteNth()` deletes the n^{th} element from `the_list` and copies the data from the deleted element into a given location (`*data_ptr`). If `the_list` contains fewer than `the_n` elements, the function returns 0, leaving `*data_ptr` unchanged. Otherwise, the function returns 1.

```
char *SL_ListPush(
   SL_List     the_list,          /* list to which a new element is to be added */
   char        *data_ptr          /* pointer to the element to be added */
)
```

`SL_ListPush()` adds a new element to the beginning of a given list (*i.e*, it pushes an element onto the list). The function returns a pointer to the new list element created that contains `*data_ptr`.

```
int SL_ListPop(
   SL_List     the_list,          /* list from which to remove an element */
   char        *data_ptr          /* pointer to place to store the removed element */
)
```

`SL_ListPop()` deletes the first element from a given list and copies the data from the deleted element into `*data_ptr` (*i.e.*, it pops an element off the list). If the list is empty, the function returns 0, leaving `*data_ptr` unchanged. Otherwise, the function returns 1.

```
char *SL_ListAppend(
   SL_List     the_list,          /* list to which a new element is to be added */
   char        *data_ptr          /* pointer to the element to be added */
)
```

`SL_ListAppend()` adds a new element to the end of a given list. The function returns a pointer to the new list element created that contains `*data_ptr`.

```
int SL_ListUnappend(
   SL_List     the_list,          /* list from which an element is to be deleted */
   char        *data_ptr          /* pointer to place to store deleted element */
)
```

`SL_ListUnappend()` deletes the last element from a given list and copies the data from the deleted element into `*data_ptr`. If the list is empty, the function returns 0, leaving `*data_ptr` unchanged. Otherwise, the function returns 1.

```
void SL_ListDispose(
   SL_List     the_list           /* list that is to be destroyed */
)
```

SL_ListDispose() disposes of (*i.e.*, frees the memory occupied by) all the elements in the_list. After a call to this function the_list is empty. It need not be reinitialized.

```
int SL_ListDeletePred(
   SL_List      the_list,          /* the list from which elements should be removed */
   int          (*pred_fn)(char *data_ptr, char *handle),
                                   /* predicate function that indicates if an
                                      element satisfies the deletion condition */
   char         *handle            /* pointer to some data that is to be passed
                                      to each call of pred_fn */
)
```

SL_ListDeletePred() deletes all the elements from a given list (the_list) that satisfy a given predicate. This predicate is supplied in the form of a function (pred_fn()) that takes two arguments and returns 1 or 0 depending on whether the given element satisfies the deletion condition. The two arguments to pred_fn() are a pointer to the data element in question and a pointer to another data element necessary to determine if the list element should be deleted. This second argument will be the same for every call to pred_fn() while processing the_list and must be supplied as the third argument to SL_ListDeletePred(). If no such data element is needed to determine the return value of pred_fn(), the third argument of SL_ListDeletePred() may be NULL. Note, however, that pred_fn() must still accept two arguments.

```
int SL_ListApply(
   SL_List      the_list,          /* the list that is being traversed */
   int          (*apply_fn)(char *data_ptr, char *handle),
                                   /* function to call for each element of list */
   char         *handle            /* pointer to some data item that is passed
                                      to each call of apply_fn */
)
```

```
int SL_ListApplyBwd(
   SL_List      the_list,          /* the list that is being traversed */
   int          (*apply_fn)(char *data_ptr, char *handle),
                                   /* function to call for each element of list */
   char         *handle            /* pointer to some data item that is passed
                                      to each call of apply_fn */
)
```

SL_ListApply() (SL_ListApplyBwd()) traverses through the elements of the_list from first to last (last to first), applying the function apply_fn() to each element. This function takes two arguments and returns 1 or 0 to indicate if the traversal of the list elements should continue or stop. The two arguments to apply_fn() are a pointer to a list element and a pointer to another data element that is necessary to process the list element. The second argument will be the same for every call to apply_fn() while processing the_list and must be supplied as the third argument to SL_ListApply(). If no such data element is needed for processing the list elements, the third argument of SL_ListApply() may be NULL. Note, however, that apply_fn() must still accept two arguments.

```
char *SL_ListFind(
   SL_List      the_list,          /* the list that is being indexed */
   int          (*pred_fn)(char *data_ptr, char *handle),
                                   /* function indicating when an element is found */
   char         *handle            /* pointer to data element passed to each call
                                      of pred_fn */
)
```

SL_ListFind() traverses through the elements of the_list to find the first element that satisfies a given predicate. The predicate is applied to the elements of the_list from first to last until it finds an element for which the function pred_fn() returns 1. The predicate function takes two arguments and returns an integer. The two arguments to pred_fn() are a pointer to a list element and a pointer to another data element that is necessary to determine the predicate value. The second argument will be the same for every call to pred_fn() while processing the_list and must be supplied as the third argument to SL_ListFind(). If no such data element is needed, the third argument of SL_ListFidn() may be NULL. Note, however, that pred_fn() must still accept two arguments. The return value of SL_ListFind is a pointer to the first element that satisfies the given predicate function. If no elements satisfy the predicate, the function returns NULL.

The following set of functions are used to traverse through a list's elements by using an index, much as one would traverse through the elements of an array using an index. The index value is set by the function SL_ListSetI() and may then be changed with SL_ListIncI(), SL_ListDecI() and SL_ListFindI(). The index value is 1-based, which means that the first element of the list corresponds to index value 1. The element corresponding to the current index value of a list may be retrieved or deleted using SL_ListGetI() or SL_ListDelI(), respectively. The function SL_ListInsI() may be used to insert a new element into a list at the position corresponding to the current index value.

```
void SL_ListSetI(
   SL_List     the_list,          /* the list that is being indexed */
   int         the_I              /* the 1-based index number for the list */
)
```

SL_ListSetI() sets the index value of the_list to the_I.

```
int SL_ListIncI(
   SL_List     the_list           /* the list that is being indexed */
)
```

SL_ListIncI() increases the index value of the_list by 1.

```
int SL_ListDecI(
   SL_List     the_list           /* the list that is being indexed */
)
```

SL_ListDecI() decreases the index value of the_list by 1.

```
char *SL_ListFindI(
   SL_List     the_list,          /* the list that is being indexed */
   int         (*pred_fn)(char *data_ptr, char *handle),
                                  /* function indicating when an element is found */
   char        *handle            /* pointer to data element passed to each call
                                     of pred_fn */
)
```

SL_ListFindI() finds and returns a pointer to the first element of a given list that satisfies a certain predicate. The predicate is supplied in the form of a function (pred_fn()) that takes two arguments and returns 1 or 0 depending on whether the given element satisfies a certain condition. The two arguments to pred_fn() are a pointer to the data element in question and a pointer to another data element necessary to determine if the list element should be deleted. This second argument to pred_fn() must be supplied as the third argument to SL_ListDeletePred(). If no such data element is needed to determine the return value of pred_fn(), the third argument of SL_ListFindI() may be NULL. Note, however, that pred_fn() must still accept two arguments. After a call to this function, the index value of the_list will be the index value of the first element that satisfies the given predicate. If there is no list element for which pred_fn() returns 1, SL_ListFindI() will return NULL and the index value of the_list will be one more than the number of elements in the_list.

```
char *SL_ListGetI(
   SL_List     the_list           /* the list that is being indexed */
)
```

`SL_ListGetI()` returns a pointer to the data element corresponding to the current index value.

```
void SL_ListDelI(
   SL_List     the_list           /* the list that is being indexed */
)
```

`SL_ListDelI()` deletes the list element corresponding to the current index value.

```
char *SL_ListInsI(
   SL_List     the_list,          /* the list that is being indexed */
   char        *data_ptr          /* pointer to data element to be inserted at
                                     the_list's index position */
)
```

`SL_ListInsI()` inserts a new element into `the_list` at the position corresponding to the current index value. The function returns a pointer to the new list element where `*data_ptr` was stored.

References

[1] V. Lumelsky and A. Stepanov, Path planning strategies for a point mobile automaton moving amongst unknown obstacles of arbitrary shape, *Algorithmica*, 3(4): 403-440, 1987.

[2] V. Lumelsky and T. Skewis, Incorporating range sensing in the robot navigation funciton, *IEEE Transactions on Systems, Man, and Cybernetics*, 20(5): 1058-1069, October 1990.

A car.c code

In the directory /usr/local/SL/2/Examples, this program is split into two files car.c and park.c with a header file car.h. Here we present the program as if it appeared in a single file.

```
#include <stdio.h>
#include <math.h>
#include <string.h>
#include <malloc.h>

#include "SL.h"

typedef struct {
   SL_Point front_axle;
   double axle_sin, axle_cos;
   double wheel_sin, wheel_cos;
   double velocity;
} CarConf;

struct _Car
{
   CarConf conf;
   char *body_color;
   double length, lenght_inv;
   SL_Polygon body;
   int collisions, time_steps;
   double path_length;
};

typedef struct _Car *Car;

static Car MyCar;

static ModelAtFrontAxle(Car the_car)
{
   SL_ModelIdentity();
   SL_ModelTranslate(the_car->conf.front_axle.x, the_car->conf.front_axle.y);
   SL_ModelRotateSinCos(-1.0, 0.0);
   SL_ModelRotateSinCos(the_car->conf.axle_sin, the_car->conf.axle_cos);
}

static void DrawHeadLights(void)
{
   static SL_Polygon LeftHL  = {4, {{0,0},{-.2,0},{-.2,.05},{0,.1}}};
   static SL_Polygon RightHL = {4, {{0,0},{.2,0},{.2,.05},{0,.1}}};
   SL_ModelPush();
      SL_ModelTranslate(-0.1, 0.2);
      SL_DrawPolygon(&LeftHL, 0);
      SL_ModelTranslate(0.2, 0.0);
      SL_DrawPolygon(&RightHL, 0);
   SL_ModelPop();
}

static void DrawWheel(double wheel_sin, double wheel_cos)
{
   static SL_Polygon wheel_poly = {4, {{.05,-.2}, {.05,.2},
                                       {-.05,.2}, {-.05,-.2}}};
   SL_ModelPush();
      SL_ModelRotateSinCos(wheel_sin, wheel_cos);
      SL_DrawPolygon(&wheel_poly, 1);
   SL_ModelPop();
}

static void DrawAxle(double wheel_sin, double wheel_cos)
{
   static SL_Segment axle_segm = {{-.4,0},{.4,0}};
   SL_DrawSegment(&axle_segm);
   SL_ModelPush();
      SL_ModelTranslate(-.4, 0.0);
      DrawWheel(wheel_sin, wheel_cos);
      SL_ModelTranslate(.8, 0.0);
      DrawWheel(wheel_sin, wheel_cos);
   SL_ModelPop();
}

static void ComputeCarBody(Car the_car)
{
   static SL_Polygon BodyPoly = { 8, {{.4,.3},{0,.4},{-.4,.3},
                                      {-.4,-1.3},{-.1,-1.3},{0,-1.2},
                                      {.1,-1.3},{.4,-1.3}} };
   SL_ModelPush();
      ModelAtFrontAxle(the_car);
      SL_ModelScale(the_car->length);
      SL_ModelGetWorldPolygon(&BodyPoly, &(the_car->body));
   SL_ModelPop();
}

/* Non-holonomic kinematics */

static void CarRoll(Car the_car)
{
   CarConf loc_conf;
   loc_conf = the_car->conf;
   the_car->conf.front_axle.x += /* v cos [ theta + phi ] */
      loc_conf.velocity*(loc_conf.axle_cos*loc_conf.wheel_cos-
 loc_conf.axle_sin*loc_conf.wheel_sin);
   the_car->conf.front_axle.y +=  /* v sin [ theta + phi ] */
      loc_conf.velocity*(loc_conf.axle_sin*loc_conf.wheel_cos+
 loc_conf.axle_cos*loc_conf.wheel_sin);
   {
      double factor = loc_conf.velocity*loc_conf.wheel_sin/the_car->length;
      double norm = sqrt(1+factor*factor);
      the_car->conf.axle_sin += factor*loc_conf.axle_cos;
      the_car->conf.axle_sin /= norm;
      the_car->conf.axle_cos -= factor*loc_conf.axle_sin;
      the_car->conf.axle_cos /= norm;
   }
}

static int CarGuardedRoll(Car the_car)
{
   CarConf save_conf;
   SL_Polygon save_body;
   the_car->time_steps++;
   save_conf = the_car->conf;
   save_body = the_car->body;
   CarRoll(the_car);
   ComputeCarBody(the_car);
   if ( CarCollision(the_car) )
   {
      the_car->conf = save_conf;
      the_car->body = save_body;
      the_car->conf.velocity *= -0.25;
      the_car->collisions++;
      return(0);
   }
   else
   {
      the_car->path_length += fabs(the_car->conf.velocity);
      return(1);
   }
}

void CarSetWheel(Car the_car, double wheel_deg_angle)
{
   double wheel_rad_angle = SL_DegreesToRadians(wheel_deg_angle);
   the_car->conf.wheel_cos = cos(wheel_rad_angle);
   the_car->conf.wheel_sin = sin(wheel_rad_angle);
}

double CarGetWheel(Car the_car)
{
   return(SL_RadiansToDegrees(atan2(the_car->conf.wheel_sin,
                                    the_car->conf.wheel_cos)));
}

void CarReset(Car the_car, double deg_orient, double pos_x, double pos_y)
{
   double ori_rad = SL_DegreesToRadians(deg_orient);
   the_car->collisions = the_car->time_steps = 0;
   the_car->path_length = 0.0;
   the_car->conf.front_axle.x = pos_x;
   the_car->conf.front_axle.y = pos_y;
   the_car->conf.axle_cos = cos(ori_rad);
   the_car->conf.axle_sin = sin(ori_rad);
   the_car->conf.velocity = 0.0;
   ComputeCarBody(the_car);
   CarSetWheel(the_car, 0.0);
}

Car CarCreate(double base_length, char *body_color)
{
   Car the_car = (Car)malloc(sizeof(struct _Car));
   the_car->body_color = body_color;
   the_car->length = base_length;
   CarReset(the_car, 0.0, 0.0, 0.0);
   return(the_car);
}
```

```
int CarGetTimeSteps(Car the_car)
{
   return(the_car->time_steps);
}

double CarGetPathLength(Car the_car)
{
   return(the_car->path_length);
}

void CarDraw(Car the_car)
{
   static SL_Segment CenterAxle = {{0,0},{0,1.0}};
   SL_SetDrawColor(the_car->body_color);
   SL_DrawPolygon(&(the_car->body), 1);
   SL_SetDrawColor("black");
   SL_DrawPolygon(&(the_car->body), 0);
   SL_ModelPush();
      ModelAtFrontAxle(the_car);
      SL_ModelScale(the_car->length);
      DrawHeadLights();
      DrawAxle(the_car->conf.wheel_sin, the_car->conf.wheel_cos);
      SL_ModelTranslate(0.0, -1.0);
      DrawAxle(0.0, 1.0);
      SL_DrawSegment(&CenterAxle);
   SL_ModelPop();
}

int CarGetCollisions(Car the_car)
{
   return(the_car->collisions);
}

int CarCollision(Car the_car)
{
   return(SL_InterPolygonObstCheck(&(the_car->body))||
          SL_InterPolygonCanvasPolygonCheck(&(the_car->body)));
}

int CarStep(Car the_car, double step_size)
{
   the_car->conf.velocity = step_size;
   return(CarGuardedRoll(the_car));
}

static void SimuInit(void)
{
   SL_MakeColor("obstacles", 0.35, 0.1, 0.1);
   SL_ObstLoadFile("spot1");
   MyCar = CarCreate(75.0, "green");
}

static void SimuReset(void)
{
   CarReset(MyCar, 0.0, 300.0, 300.0);
}
```

```
static void SimuRedraw(void)
{
   static SL_Point DrawPos = {25,25};
   char buf[64];
   sprintf(buf, "Steps: %-3d, Hits: %-2d",
   CarGetTimeSteps(MyCar), CarGetCollisions(MyCar));
   CarDraw(MyCar);
   SL_DrawText(&DrawPos, buf);
}

static void WheelPos(void)
{
   double new_wheel = CarGetWheel(MyCar)+WHEEL_STEP;
   if (new_wheel <= WHEEL_MAX)
   {
      CarSetWheel(MyCar, new_wheel);
      SL_Redraw();
   }
}

static void WheelNeg(void)
{
   double new_wheel = CarGetWheel(MyCar)-WHEEL_STEP;
   if (new_wheel >= -WHEEL_MAX)
   {
      CarSetWheel(MyCar, new_wheel);
      SL_Redraw();
   }
}

static void WheelZero(void)
{
   CarSetWheel(MyCar, 0.0);
   SL_Redraw();
}

static void MoveFwd(void)
{
   if ( !CarStep(MyCar, STEP_SIZE) )
      fprintf(stderr, "\a");
   SL_Redraw();
}

static void MoveBwd(void)
{
   if ( !CarStep(MyCar, -STEP_SIZE) )
      fprintf(stderr, "\a");
    SL_Redraw();
}

int main()
{
   SL_Init(CVS_SIDE, CVS_SIDE, 1, SimuInit, SimuReset, SimuRedraw,
           NULL, NULL, NULL, NULL, NULL);
   SL_AddLabel("Controls:");
   SL_AddButton("W+", 1, WheelPos);
   SL_AddButton("W0", 0, WheelZero);
   SL_AddButton("W-", 1, WheelNeg);
   SL_AddButton("Fwd", 1, MoveFwd);
   SL_AddButton("Bwd", 1, MoveBwd);
   SL_Loop();
   return(0);
}
```

Index