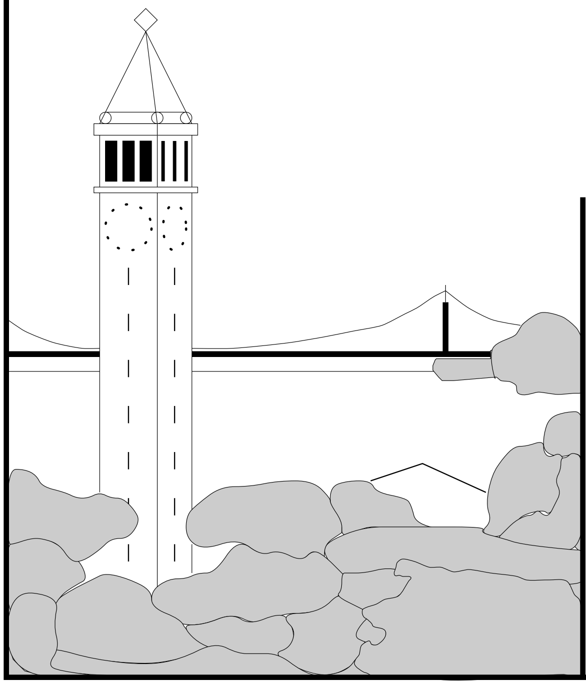
VIGL: Visualization Graphics Library

Allen B. Downey



Report No. UCB/CSD 93/764

June 1993

Computer Science Division (EECS)
University of California
Berkeley, California 94720

VIGL: Visualization Graphics Library*

Allen B. Downey

June 1993

1 Introduction

VIGL is a library of C procedures for displaying graphical (X window) representations of two-dimensional data. They can be called directly from C or FORTRAN, or through the FIDIL I/O Library. This document presents the C interface to these procedures.

There are six kinds of objects in the universe of the graphics library: spaces, grids, segments, points, graphs and coordinate transformations. A *space* is a window that corresponds to a problem space you are dealing with. A *grid* is the graphical representation of a two-dimensional data object; it corresponds to a FIDIL map. *Segments* and *points* are simple graphical objects that can be displayed in a problem space. *Graphs* are windows that contain a standard x–y plot of some 1-dimensional data. Finally, *coordinate transformations* are mappings from the index space to the problem space; they are used to create quadrilateral (but non-rectangular) grids.

Grids are displayed as unions of quadrilateral regions filled with colored cells, one for each data point. You can control the mapping between a cell's value and its color with the vigl_set_*something*_parameters procedures.

In general, these objects are created with the procedure named vigl_create_*something* and destroyed with vigl_destroy_*something*. Grids and segments are displayed with vigl_draw_*something*_in_space, and deleted with vigl_erase_*something*.

*This work funded by NSF (DARPA) grant DMS-8919074.

Graphical objects exist and can be manipulated even when they are not being displayed.

2 Spaces

`Space* vigl_create_space(float*` *low*`, float*` *high*`,`
`float*` *min_cell_size*`, int*` *size*`, int*` *loc*`)`

Creates and displays a new window for the problem space, according to the provided dimensions. *loc* is an array of two integers that specifies the upper right-hand corner of the window in screen (pixel) coordinates. *size* (also an array of two integers) gives the *suggested* size of the window.

min_cell_size is an array of two floats giving the dimensions (in problem-space coordinates) of the smallest cell you plan to draw in this space. This is an optional hint that is used by the library to choose the size of the window. The library tries to choose a window size as close as possible to the suggested dimensions subject to the constraint that the minimum cell size be an integral number of pixels. The actual size of the window will replace the suggested values in the *size* array. If *min_cell_size* is 0.0, the suggested size is accepted.

The arrays *high* and *low* contain the dimensions of the problem space. For example, a problem space running from 0.0 to 1.0 in each dimension is indicated by

```
low = {0.0, 0.0};
high = {1.0, 1.0};
```

`vigl_build_attic_for_space(Space*` *space*`)`

An attic is a small space (1/3 the size) attached to a problem space used for temporary storage of various objects. In Select mode (described under *vigl_interact* below), clicking Button 1 on an object moves it into the attic, or back.

```
vigl_set_space_parameters(Space* space, int map_type,
                          float params[])
```

Setting the parameters for a space (or grid) controls the mapping between floating point values and cell colors. The *map_type* is one of the following.

HUE_MAP hue mapping: low values are blue, high values are red

BANDED_HUE_MAP same as `HUE_MAP`, except for periodic black bands

GRAY_MAP gray scale

BANDED_COLOR_MAP color bands: blue $\mapsto$ green, blue $\mapsto$ cyan, blue $\mapsto$ yellow, green $\mapsto$ yellow, green $\mapsto$ cyan, red$\mapsto$magenta, red$\mapsto$yellow, red$\mapsto$white

RANDOM_MAP random mapping

The argument *params* is an array of three elements giving the range of values, and a special value used to represent invalid data. For example, if the range of valid values is 0.0–100.0, we might choose -99.0 to represent invalid data. Then,

```
params[0] = 0.0
params[1] = 100.0
params[2] = -99.0     (invalid data)
```

Cells with values outside of the range are mapped to an attention-getting color, depending on the color map.

Individual grids may carry their own mappings (set with vigl_set_grid_parameters). Any grid without a mapping will use the space's mapping as a default. Attempting to draw a grid without a mapping in a space without a mapping results in an error.

Changing a space's color mapping automatically updates all grids in the space and the space's attic.

```
vigl_destroy_space(Space* space)
```

Erases the space and its window. Subsequent operations on this space may cause unexpected behavior.

3 Grids

`Grid* vigl_create_grid(float` *cell_size*`[], float` *offset*`[])`

Creates a new empty grid with cell dimensions given by *cell_size* problem space coordinates. The vector *offset* gives the location of the lower-left corner of the grid.

To display the grid, you must specify some data using vigl_add_to_grid, and draw the grid with vigl_draw_grid_in_space.

`vigl_add_to_grid(Grid*` *grid*`, float` *data*`[], int` *lwb*`[], int` *upb*`[])`

Adds a rectangle of data to an existing grid. *data* is a pointer to an array of floating point values in row-major form (apologies to FORTRAN users). *lwb* and *upb* are index ranges that specify where in the grid these values fall.

By adding multiple rectangles to a single grid, you can construct an arbitrarily complex grid. Note that areas of the grid that are not covered by data are transparent.

If two regions within a grid overlap, the result is undefined unless the data is identical in the overlapping cells.

`vigl_set_grid_parameters(Grid*` *grid*`, int` *map_type*`, float` *params*`[])`

For an explanation of the mapping between floating point values and cell colors, see vigl_set_space_parameters above. This procedure binds a color mapping to a specific grid. When the grid is drawn, this binding will override the space's color mapping. If the grid is already displayed, it will be updated.

`vigl_draw_grid_in_space(Grid*` *grid*`, Space*` *space*`)`

This procedure causes the grid to appear as a graphical object in the given problem space (or, instead of the space argument, you can pass the constant DEFAULTSPACE). All of the button bindings described in vigl_interact will apply to the new object.

`vigl_erase_grid(Grid*` *grid*`)`

Removes *grid* from the space in which it appears. If it is not being displayed, this procedure has no effect.

`vigl_destroy_grid(Grid*` *grid*`)`

Returns the storage used by *grid*, which is no longer a valid argument to other routines.

4 Segments

`Segments* vigl_create_segments(float` *data*`[], int` *n*`, char*` *color*`)`

Creates a new graphical object consisting of a set of line segments. *data* is an array of $4n$ floating-point values specifying the endpoints of the segments in the format $x_0, y_0, x'_0, y'_0, x_1, y_1, \ldots$, so that line i has endpoints (x_i, y_i) and (x'_i, y'_i). n is the number of line segments. *color* is a string containing standard X color specification. See Color Specification for details.

`vigl_draw_segments_in_space(Segments*` *segment*`, Space*` *space*`)`

This procedure is analogous to vigl_draw_grid_in_space, described above. A set of line segments is considered a single graphics object, and behaves as such (for example, when it is moved back an forth between the attic and the primary space).

`vigl_erase_segments(Segments*` *segments*`)`

Removes a set of line segments from the space in which it is displayed.

`vigl_destroy_segments(Segments*` *segments*`)`

Releases storage for *segments*, which is thereafter an invalid argument.

5 Graphs

`Graph* vigl_create_graph(int` *size*`[], int` *loc*`[], char*` *color*`)`

Creates a new graph and displays its window, according to the dimensions in *size* and the location in *loc*. *color* determines the color of the objects (axes, labels and data lines) in the graph. See Color Specification for details.

`vigl_draw_graph(Graph*` *graph*`, float` *data*`[], int` *mask*`[],`
`int` *n*`, float` *low*`, float` *high*`, float` *axis*`)`

Adds data to an existing graph window. *data* is a vector of floating-point values; *n* is its length. *mask* is an array of integers that indicate the validity (1) or invalidity (0) of the corresponding piece of data. Invalid data will not be displayed in the graph. If *mask* is NULL, all data are considered valid. *params* is an array of three values specifying the range of the data and where the axis falls on the graph. For example, if the data fall in the range 0.0–100.0 and the x-axis is at $y = 0.0$, then:

```
params[0] = 0.0   (low)
params[1] = 100.0 (high)
params[2] = 0.0   (axis)
```

If the first two parameters are the same, the range will be determined automatically.

`vigl_erase_graph(Graph*` *graph*`)`

Erases the data and axes from a graph, but leaves the empty window intact.

`vigl_destroy_graph(Graph*` *graph*`)`

Erases a graph space and recycles the associated resources.

6 Coordinate Transformations

By default, VIGL assumes that the mapping between index space and the problem space is a function of the offset and the size of the grid cells. Specifically,

$$x = (i - lwb) \cdot cell\_size + offset,$$

where i is the index pair, lwb is the index pair of the lower-left corner of the grid, $cell\_size$ is the size of the cells, $offset$ is the position of the lower-left corner of the grid, x is the position of the ith cell, in problem space coordinates.

VIGL provides a mechanism for overriding this default and providing an arbitrary mapping between index space and problem space. This mapping is called a Tform (for coordinate transformation). The following are the procedures for creating, applying, and destroying Tform objects.

```
Tform* vigl_create_tform(int lwb[], int upb[],
                         void (*map)(int *, float *))
```

lwb and *upb* are arrays of two integers, giving the index range over which the coordinate tranformation is valid. If the transformation is applied to any grid with cells outside this range, an error will result.

map is a user-provided function that maps the index space into the problem space. The following is a simple example:

```
map (int *i, float *x)
{
   x[0] = i[0] * 30.0 + 25.0;
   x[1] = i[1] * 40.0 + 20.0;
}
```

This mapping would be equivalent to a grid with the lower-left corner at (25.0, 20.0) and each cell 30 units wide and 40 units high.

```
vigl_apply_tform(Grid* grid, Tform* tform)
```

Applies a coordinate transformation to a grid. The same transformation may be applied to a number of grids. If the grid is displayed in some space, it is updated automatically. If the grid contains a cell that falls outside the range of the transformation, an error results.

`vigl_destroy_tform(Tform*` *tform*`)`

Frees up the transformation. Note that destroying a grid does not destroy any associated transformation.

7 Interactive behavior

`vigl_interact(Space*` *space*`)`

The above procedures can be used to present data in various formats, but they do not allow the user to perform any sort of interactive inspection. The procedure vigl_interact creates a control window with a set of tools for manipulating and inspecting the objects on the screen.

The procedure does not return until the user presses the *Continue...* button in the control window. The other buttons are explained below.

7.1 Select

Puts the user in select mode (the default), in which the following button bindings are active:

- Button 1: moves an object back and forth between the window and its attic. If there is no attic, this button has no effect.
- Button 2: pulls an object to the front of the "pile" of objects in the problem space
- Button 3: pushes an object to the back of the "pile" of objects in the problem space

7.2 Inspect

Switches to inspect mode (the cursor becomes an eye). The following bindings are active:

> Button 1: clicking on a grid cell causes the floating point value of that cell to appear next to the cursor. The value will stay as long as the button is held, then will disappear.

7.3 Plot Line

Switches to plotting mode (the cursor is a cross hair). Button 1 is used to specify the endpoints of a line segment. Once drawn, the line segment can be manipulated by dragging its endpoints (again with Button 1).

A graph will appear that shows the values of any grid cells that fall under the line segment. The graph is updated automatically whenever the user moves the line segment, or adds, erases, or modifies a grid. The graph disappears when the user leaves plotting mode.

`vigl_interact_not(Space*` *space*`)`

If there is a control panel associated with *space*, this procedure destroys it. The control panel persists until this procedure is called or the space associated with it is destroyed.

8 Sample Code

The following code demonstrates the use of some of the library routines. It resides in the *tests* subdirectory of the VIGL directory.

```
#include "vigl.h"

/* some parameters for the problem space */
float low[2] = {0.0, 0.0};
float high[2] = {400.0, 400.0};
float min_cell_size[2] = {1.0, 1.0};
```

```
int size[2] = { 400, 0 };
int loc[2] = { 200, 100 };

/* the index range of the grid, cell size and position */
int lwb[2] = {5, 15};
int upb[2] = {25, 35};
float cell_size[2] = {10.0, 10.0};
float offset[2] = {0.0, 0.0};
float params[3] = {0.0, 15.0, 0.0};

/* the index range of the coordinate transformation */
int mlwb[2] = {5, 16};
int mupb[2] = {25, 36};

/* coordinates for line segments and points */
float coords [12] = {   100, 100, 200, 200,
                        200, 300, 200, 400,
                        350, 150, 250, 150 };

/* declaration of a user procedure that allocates memory
   for the data array and stores the data there */
extern float *create_array ();

/* this procedure calculates the coordinate transformation that
   maps the index space into the problem space */
void map (int *i, float *x)
{
  x[0] = (float)(8*i[0]+2*i[1]+100);
  x[1] = (float)(8*i[1]+2*i[0]+100);
}

main(argc, argv)
int argc;
char *argv[];
{
  Space *space;
  Grid *grid, *grid2;
```

```
Points *points;
Segments *segments;
Tform *tform;
int i;
float *data = create_array();

/* create a new problem space, set the color map, */
/* and create the attic */
space = vigl_create_space (low, high, min_cell_size,
                           size, loc);
vigl_set_space_parameters (space, GRAY_MAP, params);
vigl_build_attic_for_space (space);

/* create a new grid, add the data array to the grid, */
/* and draw it */
grid = vigl_create_grid (cell_size, offset);
vigl_add_to_grid (grid, data, lwb, upb);
vigl_draw_grid_in_space (grid, space);

/* wait for the user to inspect the problem space */
/* and press Continue... */
vigl_interact (space);

/* create a new coordinate transformation and apply */
/* it to the grid. The grid will be redrawn with */
/* the new transformation */
tform = vigl_create_tform (lwb, upb, map);
vigl_apply_tform (grid, tform);

/* create a new set of line segments and display them */
segments = vigl_create_segments (coords, 3, "blue");
vigl_draw_segments_in_space (segments, space);

/* create a new set of point and display them */
points = vigl_create_points (coords, 6, "red");
vigl_draw_points_in_space (points, space);
```

```
  /* wait for the user */
  vigl_interact (space);

  /* erase everything */
  vigl_erase_grid (grid);
  vigl_erase_segments (segments);
  vigl_erase_points (points);

  /* wait for the user */
  vigl_interact (space);

  /* redraw everything */
  vigl_draw_grid_in_space (grid, space);
  vigl_draw_segments_in_space (segments, space);
  vigl_draw_points_in_space (points, space);

  /* wait */
  vigl_interact (space);

  /* destroy everything */
  vigl_destroy_grid (grid);
  vigl_destroy_segments (segments);
  vigl_destroy_points (points);
  vigl_destroy_space (space);
}
```

9 Installation, Configuration, and Compilation

Figure 1 is a diagram of a typical (non-distributed) VIGL application. The user code is linked with the VIGL library. The VIGL library makes calls to the X server, which creates and maintains the display, and generates user events like button presses and keystrokes.

This scheme has two disadvantages. First, the VIGL library is large, so linking time is long and the executable code is large. Second, the user code

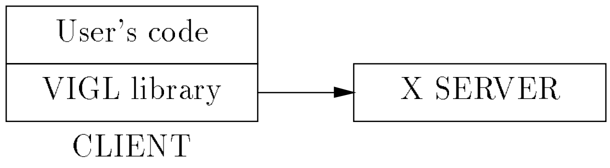


Figure 1: Diagram of a typical, non-distributed VIGL application.

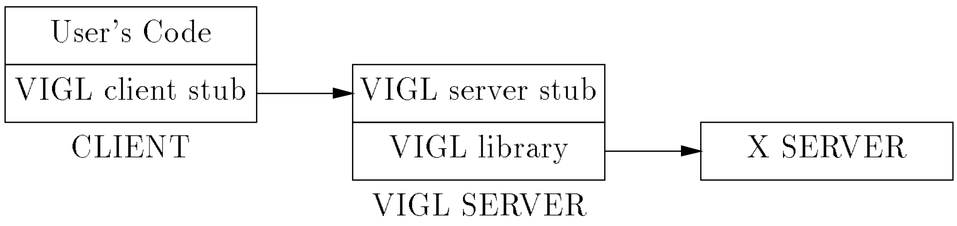


Figure 2: A distributed version of the application in Figure 1.

and the VIGL library have to run on the same machine, which may not be the most efficient use of the hardware.

Figure 2 shows the distributed version of the same application. The user code is linked with the client stubs, which are much smaller than the VIGL library. These stubs communicate with the VIGL server using UNIX sockets. The VIGL server, which contains the VIGL library, invokes the appropriate procedure and generates events for the X server.

9.1 Installation

The VIGL directory contains five subdirectories:

lib contains the VIGL library procedures described above

vigl_server contains the VIGL server stubs

gen_client contains VIGL client stubs for a generic workstation architecture (32-bit words, IEEE Standard floating-point)

cray_client contains VIGL client stubs for a Cray supercomputer

tests contains sample application code, including the sample shown above

Each directory contains a Makefile that builds the relevant object file. In lib, the object file is libvigl.a. This library contains the VIGL procedures. In cray_client and gen_client, there is also a file named libvigl.a, but it contains only the client stub procedures.

The vigl_server directory builds an executable file named "server". In order to run VIGL as a distributed application, "server" must be running on the VIGL server machine.

9.2 Compilation

The following is the procedure for building a non-distributed VIGL application.

- Compile the VIGL library by moving into the lib directory and typing "make".
- Compile the user code and link it with libvigl.a in the lib directory

For example,

```
VLIB = $(VIGL_HOME)/lib
CFLAGS = -I$(VLIB)        #search VLIB for header files
LFLAGS = -L$(VLIB)        #look for libraries in VLIB
gcc $(LFLAGS) test.o -lvigl
```

The following is the procedure for building a distributed VIGL application.

- As above, compile the VIGL library.
- Compile either the generic client or the Cray client by moving to the appropriate directory and typing "make".
- Compile the VIGL server by moving to the vigl_server directory and typing "make".

- Compile the user code and link it with libvigl.a in the client directory, *not the lib directory.*

For example,

```
VLIB = $(VIGL_HOME)/gen_client
CFLAGS = -I$(VLIB)          #search VLIB for header files
LFLAGS = -L$(VLIB)          #look for libraries in VLIB
gcc $(LFLAGS) test.o -lvigl
```

9.3 Configuration

Before you can run either version of a VIGL application, you have to set several environment variables to tell the application where to find the X server and, for distributed applications, the VIGL server.

To set the X server, type:

```
setenv DISPLAY host_name:0.0
```

Usually the default X server is the machine that is running the VIGL application.

To set the VIGL server, type:

```
setenv VIGL_DISPLAY host_name
```

You must make sure that the VIGL server is running on the named host. If it is not, the VIGL applications will print an error message and abort.