Report Number: WUCS-TM-92-04

1992-08-01

CtoVis: An Interface between C Programs and Pavane Visualizations

Kenneth C. Cox

School of Engineering & Applied Science

CtoVis: An Interface between C Programs and Pavane Visualizations

Kenneth C. Cox

WUCS-TM-92-04

August 1992

Department of Computer Science
Washington University
Campus Box 1045
One Brookings Drive
Saint Louis, MO 63130-4899

Printed 4/12/93

1. Introduction

Visualizations in Pavane consist of three concurrent components. The *underlying computation* is a program whose behavior is to be visualized; we assume that the computation can be characterized by a state which undergoes a series of atomic transformations. The *visualization computation* examines the state of the underlying computation and, through application of a collection of rules, produces a set of tuples called the animation space. The *rendering computation* transforms the animation space into visual form.

This paper describes the use of *CtoVis*, a package of functions which allows a C (or C++) program to act as the underlying computation of a Pavane visualization. Through the use of CtoVis, the animator (the constructor of the visualization) initiates the visualization and rendering computations, selects which portions of the C program state are to be examined, and identifies those points in the code where the state information is to be transmitted to the visualization computation.

The remainder of this paper is divided into five parts. The first three cover respectively the specification of the part of the state that is monitored, the initialization of the visualization and rendering computations, and the transmission of state information. The fourth section describes how to compile programs that use the CtoVis package. The final section contains several complete sample uses of the package.

2. Selection of monitored data

Monitoring refers to the selection of one or more C variables whose contents are to be examined and transmitted to the visualization computation. The main operation involved in monitoring is the conversion of C data types into Pavane data types. We will first discuss the various data types, then return to the routine that handles monitoring of the variables.

CtoVis permits monitoring of the following C data types:

- Numeric data types (`char`, `short`, `int`, `long`, `float`, and `double`)
- Strings (null-terminated arrays of characters) represented either by pointers to character arrays (`char *`) or as arrays of characters (`char []`)
- Arrays (`[]`) of any monitorable type; the dimension of the array must be fixed.
- Structures (`struct`) whose components are any monitorable type.

This is, of course, a recursive definition, so it is possible to monitor a variable which is an array of structures, each containing a `short`, a `double`, and a `char *`; examples of such conversions are provided below. Pointer data types (other than `char *` as a string) are not supported at this time. Each C variable that is monitored must be transformed into one of the following Pavane data types:

- Integer (a fixed-point number with range equivalent to that of a C `long`).
- Real (a floating-point number with range equivalent to a C `double`).
- Boolean (either true or false).
- Strings (sequences of up to 100 characters).
- Arrays of any Pavane data type.
- Structures of any Pavane data type.

The conversion between these types must be indicated. The following conversions are permitted:

- Any C numeric type may be converted to a Pavane integer, real, or boolean data type, with conversion rules the same as in C. Thus, converting a C `double` to a Pavane integer is equivalent to assigning a `double` to a `long`, with the associated problems of loss of precision, while assigning any type to a Pavane boolean uses the C "zero is false, non-zero is true" test.
- C strings (`char *` and `char []`) may be converted into a Pavane string; of course, the latter type can also be considered as an array of character and converted into a Pavane array of

integer, real, or boolean. During the conversion to a Pavane string, Pavane's 100-character limit is silently enforced by the truncation of the input strings.
- C arrays may be converted into Pavane arrays.
- C structures may be converted into Pavane structures

This is again recursive, the conversions for the elements of arrays and components of structures must also be given.

Each separate monitored variable is transformed into a single Pavane tuple with a type-name selected by the user. The C data is also transformed into Pavane's internal representation. Thus, four things must be specified when a variable is selected for monitoring: the Pavane type name, the location of the variable, the Pavane data type, and the C data type. These form the arguments of the `VisualMonitor` function, a variadic function declared as:

```
VisualMonitor(char *typename, void *data, ...);
```

The first argument, `typename`, is the name of the Pavane tuple type. `data` is the address of the C data that is to be monitored. The remaining, variadic arguments, all of type `long`, specify the type-conversions between the output (Pavane) data types and the corresponding input (C) data types. The following predefined constants are to be used in this section:

`VISMONITOR_PAVANE_INTEGER` — The output data type is to be a Pavane integer. This constant must be followed by a C numeric type, specified by one of the following constants: `VISMONITOR_C_CHAR`, `VISMONITOR_C_SHORT`, `VISMONITOR_C_INT`, `VISMONITOR_C_LONG`, `VISMONITOR_C_FLOAT`, or `VISMONITOR_C_DOUBLE`.

`VISMONITOR_PAVANE_REAL` — The output data type is to be a Pavane real. This constant must be followed by a C numeric type, using the constants listed above.

`VISMONITOR_PAVANE_BOOLEAN` — The output data type is to be a Pavane boolean. This constant must be followed by a C numeric type, using the constants listed above.

`VISMONITOR_PAVANE_STRING` — The output data type is to be a Pavane string. This must be followed by a C string type, using the constant `VISMONITOR_C_CHAR_PTR` (for a `char *` data type) or `VISMONITOR_C_CHAR_ARRAY(N)` (for a `char [N]` datatype). In the latter case, `N` must be fixed and between 1 and 16384.

`VISMONITOR_PAVANE_ARRAY` — The output data type is to be a Pavane array. This constant must be followed by the size of the array, cast as a `long`, which in turn must be followed by the type-conversion of the elements of the array. For example, an array of 10 C `long`'s that is to be represented by a Pavane array of 10 integers would be specified by the conversion

```
VISMONITOR_PAVANE_ARRAY, (long)10,
VISMONITOR_PAVANE_INTEGER, VISMONITOR_C_LONG.
```

Note the recursion, with the type-conversion of the array elements specified in the same manner as the type-conversion for a single variable of that type.

`VISMONITOR_PAVANE_STRUCT` — The output data type is to be a Pavane structure. This constant must be followed by the number of components in the structure, cast as a `long`, which in turn must be followed by the type-conversions of the components in the order that they appear in the C structure. For example, a C structure containing a `long`, a `long`, and a two-dimensional array of 10 by 10 of `double`'s that is to be converted into a similar Pavane structure using integers for `long`'s and reals for `double`'s would be specified by the conversion

```
VISMONITOR_PAVANE_STRUCT, (long)3,
VISMONITOR_PAVANE_INTEGER, VISMONITOR_C_LONG,
VISMONITOR_PAVANE_INTEGER, VISMONITOR_C_LONG,
VISMONITOR_PAVANE_ARRAY, (long)10,
VISMONITOR_PAVANE_ARRAY, (long)10,
VISMONITOR_PAVANE_REAL, VISMONITOR_C_DOUBLE
```

Again note the recursion, with a separate type-conversion for each of the three components of the structure.

Three rules apply to the use of the `VisualMonitor` function. First, all monitored variables must be selected and initialized before the visualization is started using `VisualOpen`. Next, all type names must be distinct and are limited to 63 characters. Finally, the data must exist from the time `VisualMonitor` is called until the visualization process is halted (using `VisualClose`). This means that some care must be used when monitoring variables that are declared in a subroutine; since such variables are normally allocated on the C runtime stack, they cease to exist when the subroutine exits. Such variables must be moved outside the subroutine into C's "file scope".

2. Initiating and terminating a visualization

Once all monitored variables have been identified and initialized, the function `VisualOpen` is used to start the visualization and rendering programs. The visualization program must be the result of compiling a collection of visualization rules (the rendering program is fixed). `VisualOpen` has the declaration

```
VisualOpen(char *program, ...);
```

The first argument is the name of the visualization program that is to be started. The remaining arguments, all of type `char *`, are a NULL-terminated list of the arguments to be given to the visualization, in the same order as in the visualization's declaration. For example, a visualization which was declared as

```
visualization AllPairs(integer N, boolean flag)
```

would require two argument strings and then the NULL. A corresponding use of `VisualOpen`, assuming that the visualization code had been compiled into the file `AllPairsVis`, would be

```
VisualOpen("AllPairsVis", "8", "true", (char *)NULL);
```

Note that the arguments must be strings; some conversion may be required. If, for example, the value corresponding to N was known only at run-time, a sequence something like the following might be used:

```
char buf[16];
...
sprintf(buf,"%d",number_of_nodes);
VisualOpen("AllPairsVis", buf, "true", (char *)NULL);
```

`VisualOpen` starts the visualization program and sends the initial state of the computation (the values of all the monitored variables). For this reason, these variables should be initialized before `VisualOpen` is called.

To terminate a visualization, use `VisualClose`. When `VisualClose` is called, the visualization and rendering computations are halted and the CtoVis monitoring package is re-initialized. Another visualization may then be set up and started with calls to `VisualMonitor` and `VisualOpen`.

3. Transmitting state information

Once a visualization is started with `VisualOpen`, three routines may be used to send state information (the values of all the monitored variables) to the visualization. The first of these is `VisualUpdate`, which simply sends all the state information when it is called.

A second method is provided by the functions `VisualBeginAtomic` and `VisualEndAtomic`. As the names imply, this pair of functions is used to delineate the boundaries of logical "atomic transitions" in the C code, i.e., blocks of code that are considered to represent a single state change. Whenever an atomic transition is complete (defined as an equal number of calls to `VisualBeginAtomic` and `VisualEndAtomic`) the state is transmitted to the visualization. Internally, these routines modify the value of a counter; `VisualBeginAtomic` increments the counter, while `VisualEndAtomic` decrements it and, if the value is zero after decrementing, sends the state

information. The purpose here is a matter of encapsulation. A sorter, for example, could be written using the two calls to transmit its state information and, if run "stand-alone", would transmit its state at appropriate points. The same code could be used as a subroutine in a larger program with the call enclosed in a `VisualBeginAtomic` / `VisualEndAtomic` pair; as a result, the state would be transmitted only after the sort was complete, with all the intermediate `VisualEndAtomic` calls inside the sorter code having no effect.

`VisualUpdate` may be used with `VisualBeginAtomic` and `VisualEndAtomic`. Calls to `VisualUpdate` have no effect on the counter used by the other two routines.

4. Compilation

A program that is to use the CtoVis package must contain the line

```
#include "/usr/people/pavane/C_INTERFACE/CtoVis.h"
```

This file contains declarations for the CtoVis functions and definitions of various constants.

When compiling, either the C or C++ CtoVis library must be loaded, depending on which compiler you use. These libraries are located in the `/usr/people/pavane/C_INTERFACE` directory, named `CtoVis_c.a` and `CtoVis.a` respectively. A typical C++ compilation line would be

```
CC -o AllPairs AllPairs.c /usr/people/pavane/C_INTERFACE/CtoVis.a
```

5. Example programs

We provide the skeletons of two C programs using the CtoVis package. The first is an all-pairs shortest path algorithm (the Floyd-Warshall algorithm). The second is from an elevator-control simulation and uses somewhat complex data structures.

5.1. All-pairs algorithm

The all-pairs visualization contains the following code segments:

```
visualization AllPairs(integer NNodes)
   ...
input space
   << array of NNodes array of NNodes real D :: d(D) >>;
   << integer K :: scan_counter(K) >>;
```

This indicates that the visualization program (which is compiled into the file "AllPairsVis") must have one argument, an integer *NNodes*. It is also expecting tuples of two types. The type *d* must contain an *NNodes* by *NNodes* array of real numbers, while the type *scan_counter* must contain an integer. These conversions are specified in the following C code.

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

/* ARRAYSIZE corresponds to NNodes in the visualization code */
#define ARRAYSIZE 8

static void initgraph(double dist[][ARRAYSIZE]) {
   /* code, of no interest to this presentation, which initializes
    * the given graph
    */
}
```

```
main()
{
long i,j,k;
char abuf[8];
double d;
double      dist[ARRAYSIZE][ARRAYSIZE],
            distprime[ARRAYSIZE][ARRAYSIZE];

   /* Two variables are of interest here: dist and k.  We must
    * indicate that these variables are to be monitored and
    * how they are to be converted from C to Pavane.  Note that,
    * although these variables are on the C stack, they will not
    * leave scope until the visualization is completed; we can
    * therefore safely monitor them.
    */

   /* dist will become the tuple "d" (see above).  It is an array
    * of ARRAYSIZE arrays of ARRAYSIZE doubles, to be converted
    * into the equivalent two-dimensional Pavane array using
    * Pavane reals.  We pass (void *)dist because dist is already
    * the address of the array.
    */
   VisualMonitor("d",(void *)dist,
      VISMONITOR_PAVANE_ARRAY, ARRAYSIZE,
      VISMONITOR_PAVANE_ARRAY, ARRAYSIZE,
      VISMONITOR_PAVANE_REAL, VISMONITOR_C_DOUBLE);

   /* k will become the tuple "scan_counter".  It is a long, to
    * be represented by a Pavane integer.  We have to take the
    * address of the variable k and cast it to void *.
    */
   VisualMonitor("scan_counter",(void *)(&k),
      VISMONITOR_PAVANE_INTEGER, VISMONITOR_C_LONG);

   /* Initialize the monitored variables before VisualOpen is
    * called (since that routine will transmit the values of the
    * variables to the visualization)
    */
   initgraph(dist);
   k = 0;

   /* Start the visualization; this also sends the current state
    * of the monitored variables.
    */
   sprintf(abuf,"%d",ARRAYSIZE);
   VisualOpen("AllPairsVis",abuf,(char *)NULL);


   /* We now execute the algorithm.  The portion between the
    * VisualBeginAtomic and VisualEndAtomic calls form a single
    * "atomic transition" for this algorithm, in which all the
    * array entries are updated at once.  The state is sent when
    * VisualEndAtomic is called.
    */
   while (k < ARRAYSIZE) {
      VisualBeginAtomic();
      for (i = 0; i < ARRAYSIZE; i++) {
            for (j = 0; j < ARRAYSIZE; j++) {
```

```
                    d = dist[i][k] + dist[k][j];
                    if (d < dist[i][j])
                          distprime[i][j] = d;
                    else
                          distprime[i][j] = dist[i][j];
              }
        }
        for (i = 0; i < ARRAYSIZE; i++)
              for (j = 0; j < ARRAYSIZE; j++)
                    dist[i][j] = distprime[i][j];
        k++;
        VisualEndAtomic();
    }

    /* To exit cleanly, we call VisualClose.  Simply returning
     * from main() or calling exit() would also terminate the
     * visualization, but not as cleanly.
     */
    VisualClose();
}
```

5.2. Elevator control simulation

The elevator visualization contains the following code segments:

```
visualization ElevatorVis(integer NElevators, integer NFloors)
   ...
types
   elevator_type ==
      struct of   integer floor,
                  integer dir,
                  integer vel,
                  boolean open,
                  array of NFloors boolean buttons;
   button_type ==
      struct of   boolean up,
                  boolean down;

input space
   << array of NElevators elevator_type E :: elevators(E) >>;
   << array of NFloors button_type B :: buttons(B) >>;
```

The C code (as seen below) has corresponding structures. The main area of interest in this example is the specification of the conversion between the types.

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

#define NELEVATORS 2
#define NFLOORS 6

static struct elevator {
   long     floor;
   long     direction;
   long     velocity;
   short    open;
   short    buttons[NFLOORS];
} elevators[NELEVATORS];
```

```
static struct     {
   short up;
   short down;
} call_buttons[NFLOORS];

static void InitializeElevators(void) {
   /* some code of no particular interest which initializes
    * the elevators and call_buttons data structures
    */
}

static void InitiateMonitoring(void) {
char ebuf[16], fbuf[16];

   /* elevators is an array of NELEVATORS structures, so its
    * type-conversion is "VISMONITOR_PAVANE_ARRAY, NELEVATORS"
    * followed by the type-conversion for the structure.  The
    * structure has five components (floor, directions, velocity,
    * open, and buttons), so its type-conversion is
    * "VISMONITOR_PAVANE_STRUCT, 5" followed by the
    * type-conversions of the components in the same order that
    * they appear in the structure.
    */
   VisualMonitor("elevators",(void *)elevators,
      VISMONITOR_PAVANE_ARRAY, NELEVATORS,
      VISMONITOR_PAVANE_STRUCT, 5,
      VISMONITOR_PAVANE_INTEGER, VISMONITOR_C_LONG,
      VISMONITOR_PAVANE_INTEGER, VISMONITOR_C_LONG,
      VISMONITOR_PAVANE_INTEGER, VISMONITOR_C_LONG,
      VISMONITOR_PAVANE_BOOLEAN, VISMONITOR_C_SHORT,
      VISMONITOR_PAVANE_ARRAY, NFLOORS,
      VISMONITOR_PAVANE_BOOLEAN, VISMONITOR_C_SHORT
   );

   /* Similarly, call_buttons is an array of NFLOORS structures,
    * each having two elements.
    */
   VisualMonitor("buttons",(void *)call_buttons,
      VISMONITOR_PAVANE_ARRAY, NFLOORS,
      VISMONITOR_PAVANE_STRUCT, 2,
      VISMONITOR_PAVANE_BOOLEAN, VISMONITOR_C_SHORT,
      VISMONITOR_PAVANE_BOOLEAN, VISMONITOR_C_SHORT
   );

   sprintf(ebuf,"%d",NELEVATORS);
   sprintf(fbuf,"%d",NFLOORS);
   VisualOpen("ElevatorVis", ebuf, fbuf, (char *)NULL);
}

static void ChangeElevators(void) {
   /* more code of no particular interest which simulates the
    * pressing of buttons and the movement of the elevators
    */
}


main() {
```

```
    /* initialize the state variables (elevators, buttons) */
    InitializeElevators();

    /* monitor the two variables and start the visualization */
    InitiateMonitoring();

    /* this simulation runs forever, with one update after each
     * modification of the state.  We could get the same effect
     * by placing ChangeElevators within a VisualBeginAtomic /
     * VisualEndAtomic pair.
     */
    for (;;) {
       ChangeElevators();
       VisualUpdate();
    }
}
```