THE UNIVERSITY *of* MANCHESTER

Advanced AVS Course

OHP Overviews

ADVISORY GROUP ON Computer Graphics

S Larkin, A J Grant, F Lin, N Hill

Manchester and North HPC T&EC

MAN T&EC

The Manchester and North HPC Training and Education Centre

Edition 6.0 December 1994

Advanced AVS Course

Module Writing

Handouts

- These OHPs
- Practical exercises (including solutions)
- AVS Developers Guide

Pre-requisites

- We will be covering writing modules in C
  (but no FORTRAN)
- We will **not** be teaching C
- We assume a basic knowledge of AVS and using the network editor

Writing modules

- Modules can be written in C and F77
- AVS 5.01 supports ANSI C and C++
- They are compiled using normal system compilers
- Links are made to AVS libraries to describe and control the module
- Most modules can also be used remotely without any further modification

External libraries

- ❑ Other external system and subroutine libraries can be used by modules
- ❑ Care must be taken if you use any “Graphics” or X Windows code in the modules.

Overview of a module

- ❑ They can be subroutines or coroutines
- ❑ subroutines are commonly used for modules
- ❑ A module has two main parts
- ❑ **Description** function: defines input/output ports and parameters
- ❑ **Compute** function: code which executes when module receives new data
- ❑ You can also specify **initialization** and **destruction** functions

Subroutines V Coroutines

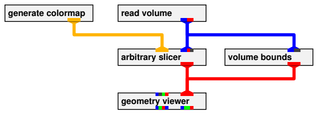
- ❑ Subroutine modules execute their compute function when new input data is received or a parameter value changes
- ❑ They are controlled by the AVS flow executive
- ❑ Coroutine modules are main programs which execute continuously

Subroutines V Coroutines (2)

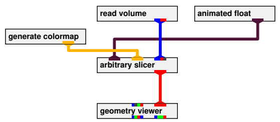
- ❑ They inquire input data and parameter changes whilst sending data to the output port
- ❑ Functions exist to synchronise modules
- ❑ You can mimic a subroutine module using a coroutine (but you wouldn’t do this!)

Subroutines V Coroutines (3)

- Subroutine module



- Coroutine module



- We will concentrate on subroutines

Describing a module

- These are all contained in the description function
- The description function is called when a module is added to AVS and again when instanced
- The description function defines:
  - *module name and class*
  - *input and output ports*
  - *parameters and widgets*
  - *the compute function*

Example of a description function

```
AVSset_module_name("do_threshold", MODULE_FILTER);

in_port = AVScreate_input_port("input",
"field 3D 3-space scalar uniform float", REQUIRED);

out_port = AVScreate_output_port("output",
"field 3D 3-space scalar uniform float");

param = AVSadd_float_parameter("pmin",0.0,0.0,1.0);
AVSconnect_widget(param, "dial");

param = AVSadd_float_parameter("pmax",1.0,0.0,1.0);
AVSconnect_widget(param, "dial");

AVSset_compute_proc(do_threshold_compute);
```


Compute function

- This is called when a module receives new input data or parameters change
- All input, output and parameter values are passed as arguments to the compute function
- Note that output data uses a double indirection **
- float parameters are passed as pointers

Compute function

example

```
int clamp_compute(in, out, pmin, pmax, ts)
AVSfield_float *in;
AVSfield_float **out;
float *pmin;
float *pmax;
int ts;
{
   /* user code to define output data */

   return(1); /* if successful */
}
```


Ex1: threshold

- This example is taken from the AVS example **`/usr/avs/examples/threshold.c`**
- The module will import a 3D array of data with a single scalar floating point data component
- It will have two dial parameters to specify the min and max data values
- The module will threshold the data according to the min and max values

Ex1: threshold algorithm

This code will be inserted into the compute function

```
*output = (AVSfield_float *)
AVSfield_alloc (input,0);

for (k = 0; k < MAXZ(input); k++)
    for (j = 0; j < MAXY(input); j++)
        for (i = 0; i < MAXX(input); i++)

            if (I3D(input, i, j, k) > max)
                I3D(*output, i, j, k) = 0.0;

            else if (I3D(input, i, j, k) < min)
                I3D(*output, i, j, k) = 0.0;

            else
                I3D(*output, i, j, k) = I3D(input, i, j, k);
```


Module generator

- this module will generate:
  - *source code for your module (C and F77)*
  - *optional hints*
  - *makefile*
  - *man page*
- You have to write your own application code though!

The module generator

Do ex1

Makefiles for AVS modules

```
INC_FILE=$(ROOT)/usr/avs/include/Makeinclude
include $(INC_FILE)

AVS_LIBS = $(ROOT)/usr/avs/lib
INC = $(ROOT)/usr/include
BASELIBS=-lgeom -lutil -lm $(LASTLIBS)
CFLAGS =  -I$(INC) $(AOPTCFLAGS) $(LOCAL_CFLAGS) $(G)
DEBUGFLAGS = -I$(INC) -g
CFLOWLIBS=-L$(AVS_LIBS) -lflow_c $(BASELIBS)
CSIMLIBS=-L$(AVS_LIBS) -lsim_c $(BASELIBS)

clamp: clamp.o
$(CC) $(CFLAGS) -o clamp clamp.o $(CFLOWLIBS)

clamp.o: clamp.c
$(CC) $(CFLAGS) -c clamp.c
```


Include files and link libraries

- All C programs need **#include<avs.h>**
- All F77 programs need **INCLUDE(avs.inc)**
- C subroutine modules use **libflow_c.a (-lflow_c)**
- C coroutines modules use **libsim_c.a (-lsim_c)**
- F77 subroutine modules use **libflow_f.a (-lflow_f)**
- F77 coroutines modules use **libsim_f.a (-lsim_f)**

Debugging AVS modules

- this is not inserting lots of **`printf`** statements
- system debuggers are used to debug modules so you can use your favourite one.
- you need to add the flags and libraries that you would normally do for debugging ordinary C or F77 programs
- There are a few steps which must be followed in sequence

Debugging steps

- An example for debugging the module **`threshold.c`** using **`xdb`**
- recompile **`threshold.c`** using the debug flags **`(-g)`**
- type **`avs_dbx -id threshold`**
- a debug window will appear
- Add the module to AVS by clicking on **`threshold.real`** and **not** **`threshold`**

Debugging steps (2)

- switch to source code display mode **`(td)`**
- set a suitable breakpoint in the module **`(b threshold_compute)`**
- Drag the module onto the workspace and a message “fire when ready” will appear
- set the module running (c)

AVS datatypes

- Already seen some example of the AVS Field Data Type
- single scalar values (integer,real,string,boolean)
- Geometry
- Unstructured Cell Data (UCD)
- Colourmaps
- Chemistry Developers Toolkit (CDK)
- User defined data

Using the AVS field data type

- ❑ One way is to describe the data using an ASCII header file
- ❑ Writing a module allows greater flexibility e.g.,
- ❑ reading more complex data formats
- ❑ extracting transient data (time steps)

Describing the data

Dimensions

- ❑ you must specify the dimensions of the data
- ❑ e.g., 1D: list of scattered data 3D: volume of data
- ❑ these specify the **computational** space
- ❑ you must specify the dimensions of the data as it is passed to AVS
- ❑ these specify the **coordinate** space

Describing the data

Mapping

- ❑ as the data is imported it can optionally be transformed
- ❑ this is a mapping from computational space to coordinate space
- ❑ there are three methods of mapping
  - ■ *uniform*
  - ■ *rectilinear*
  - ■ *irregular*

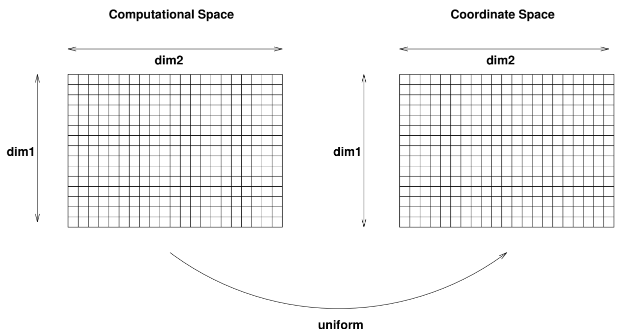
Mapping methods

UNIFORM

- ❑ The data is not transformed as it is imported (direct & implicit)
- ❑ The dimensions of the computational and coordinate space should be the same
- ❑ used to import regular arrays of data

Mapping methods

UNIFORM (2)



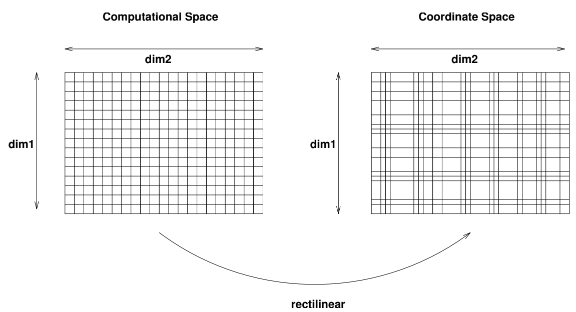
Mapping methods

RECTILINEAR

- Each dimension in the data has an explicit coordinate mapping
- The spacing along the coordinate axes can be non-uniform
- e.g., logarithmic axes
- The dimensions of the computational and coordinate space should be the same

Mapping methods

RECTILINEAR (2)



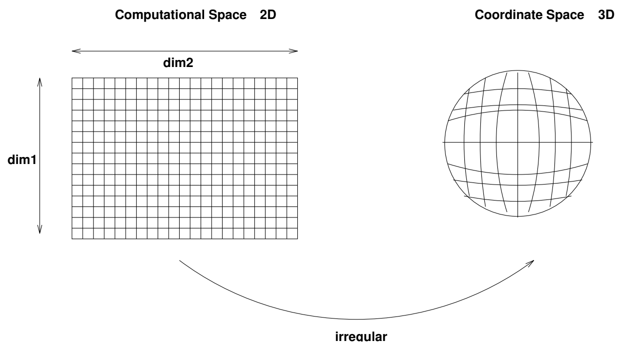
Mapping methods

IRREGULAR

- Each data element is mapped to a point in the coordinate space
- The dimensions of the computational and coordinate space can be different
- e.g., 1D scattered data into 3D space

Mapping methods

IRREGULAR (2)

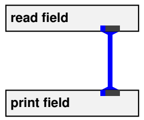


Describing the data elements

- You specify the number of components at each data element and the type e.g., byte,integer,float,double
- AVS allows you to describe each component:
  - *specifying the name*
  - *specifying the units*

Debugging

- A useful module to include in your networks is **`print_field`**



- The module will display information on the header and data elements
- N.B. by default only the first data element is shown; to display the rest use the dial on the module parameters labelled “Max Elements”.

AVS field: C structure

- You access and define the field via a C structure
- All source code files must have

  `#include<avs/field.h>`
- Some access is direct using **->** e.g., **`input->points`**
- Most other accesses are via routines or C MACROS
- It is important that you always use the correct routines and MACROS and do not misuse the -> method
- This is a safeguard incase the C structure changes in later versions

Extract from `/usr/avs/include/field.h`

```
typedef struct {
    int ndim;
    int nspace;
    int veclen;
    int type;
    int size;
    int uniform;
    int *dimensions;
    float *points;
    float *data;
    float *min_extent;
    float *max_extent;
    char *labels;
    float *minimum;
    float *maximum;
} AVSfield_float;
```


Allocating fields

- ❑ There are two methods
- ❑ Using the input field as a **template**
- ❑ Describing the field using an ASCII string

Allocating fields - an example

- ❑ using a template:

```
*output = (AVSfield_float *)
AVSfield_alloc (input,0)
```

- ❑ using a description:

```
int dims[2]
dims[0] = 5;
dims[1] = 10;
*temp_field = (AVSfield_float *)AVSdata_alloc ("field 2D
    2-space scalar uniform float",dims);
```


Using the C structure

- ❑ From "**`field 3D 3-space 5-vector irregular real`**"

```
ndim            3
nspace          3
veclen          5
type            AVS_TYPE_REAL
size            sizeof(float)
uniform         IRREGULAR
dimensions      dims[3]={x,y,z}
points          coords[3][z][y][x]
data            data[z][y][x][5]
```

Other allocation functions

❑ Checking if the field is already allocated

```
AVSfield_float **conc_field;

if (*conc_field != NULL)
{
   /* if so then free the space */

   AVSfield_free (*conc_field);
}
```


Accessing the C structure

(MACROS)

❑ to obtain the x dimension of a field

```
MAXX(field)
AVSfield *field

xdim = MAXX(input)
xdim = MAXX(*output)
```


Accessing the C structure

(MACROS) (2)

❑ to obtain the y coordinate for a 3D data element

```
COORD_Y_3D(field,i,j,k)
AVSfield *field
int i,j,k
ycoord = COORD_Y_3D(input,i,j,k)
```


Accessing the C structure

(routines)

❑ to obtain the number of dimensions

```
in_ndim = AVSfield_get_int
(field_in,AVS_FIELD_NDIM);
```

❑ to obtain the units labels

```
AVSfield_get_units
(field_in,in_unit_str,in_unit_delim)
```

Accessing the data

```
for (i=0; i < dims[0]; i++)
   for (j=0; j < dims[1]; j++)
   {
      /* add the data to the field */

      fscanf (file_ptr,"%f",&datavalue);
      I2D(*output,i,j) = datavalue;
   }
```


Accessing the coordinates

```
AVSfield_float **conc_field;
float *coords;

   coords = (*conc_field)->points;

   for (i=0; i < dims[0]; i++)
   {
      fscanf(file_ptr,"%f%f%f%f",&x,&y,&z,&dv);
      *I1DV(*conc_field,i) = dv;

      coords[i] = x;
      coords[i + dims[0]] = y;
      coords[i + (2*dims[0])] = z;
   }
```


AVS errors

- every module will get them
- saves a lot of time if you add these error checks
- AVS is not graceful at indicating the location of the error in a module
- most functions have a successful or erroneous return code

Detecting AVS errors

- this code checks for two specific errors
- no file has been specified (not a real error)
- the **`datafile`** argument is always NULL on a modules first instanciation
- opening an unreadable or non-existent file

```
if (!datafile) return (1);
if (!(file_ptr = fopen (datafile,"r")) )
{

   AVSerror("Error cannot open file\n%s\n",datafile);
   return(0);
}
```

The practical session

- The practical involves ex2,ex3 and ex7
- ex7 is not a real exercise but an illustration on how to use and allocate fields
- ex2 and ex3 involve reading a specified, but non-standard, datafile

read_temps - ex2

- the data is 2D uniform and has a single scalar real value at each point
- the first two numbers are the x and y dimensions of the field
- here is an example datafile (ordered x then y)

```
31 31
213.8787
213.8795
213.881
.......
```


read_concs - ex3

- the data is 1D irregular and has a single scalar value at each point
- the first number is the x dimension of the field
- each datavalue has an associated x,y,z coordinate
- here is an example datafile

```
7831
18.37 5.00 10.00 .06
18.94 5.00 10.00 .07
17.75 6.00 10.00 .04
.....
```

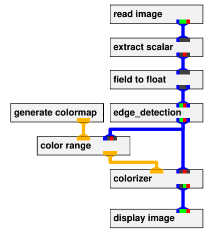

Starting the exercises

- Use the module generator to generate the skeleton code
- Ask a demonstrator for help
- Take a peek at the solution

Do ex2/ex3

Ex8: sobel edge detection

- The aim of this exercise is to write a module to apply the sobel edge detection filter to an input field
- The result can be viewed using the following network:



Evaluating the new pixels

- The edge detection filter is a 3x3 filter which is passed over the input image
- The resultant pixel $P*$ is calculated as follows:

B	C	D
A	P	E
H	G	F

- $X = (B + 2C + D) - (H + 2G + F)$
  $Y = (D + 2E + F) - (B + 2A + H)$
  $P* = \mathrm{SQRT}(X^2 + Y^2)$

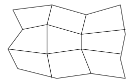
Sobel edge detection applied to avs.x

AVS geometry data type

- All files must have **`#include<avs/geom.h>`**
- More detailed information can be found in appendix G of developers guide
- The objects which are defined are usually interpreted by the module **`geometry_viewer`**
- A GEOM object is a single primitive e.g.,
  - *polylines*
  - *mesh objects*
  - *spheres*
  - *polytriangle*

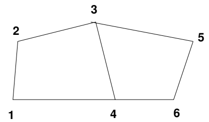
Some GEOM primitives

Mesh



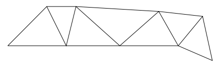
M = N = 3
(M-1)(N-1) quadrilaterals

Polyhedron



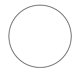
Connectivity List
{ 4, 1, 2, 3, 4, 4, 3, 5, 6, 4, 0}

Polytriangle



N verticies
(N-2) triangles

Sphere

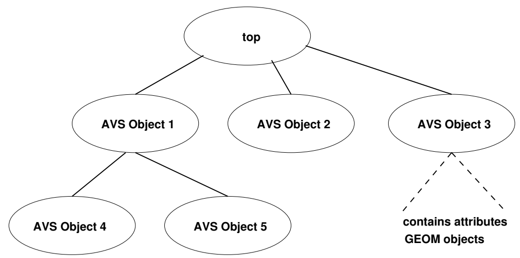


Radius R
Centre X, Y, Z

AVS objects

- An AVS object has **attributes** and **GEOM objects** associated with it
- An AVS object has a parent object and can have child AVS objects
- An AVS object will inherit attributes from its parent object
- By default the AVS Geometry Viewer has an AVS object called **`top`**
- All subsequent geometries which are imported become children of **`top`**

AVS and GEOM objects



AVS GEOM edit lists

- A sequence of commands which are interpreted by the **`geometry_viewer`** module
- some examples of commands are:
  - *object creation*
  - *camera manipulation*
  - *light source control*

Creating objects (filters)

- One way is to create **`.geom`** files using Unix style filters
- the **`.geom`** files can be read by the "Read Object" option of **`geometry_viewer`**
- some examples are in **`/usr/avs/filters`** and **`/usr/avs/data/geometry`**
- e.g., **`polygon < datafile > geomfile`**
- Note that these files are binary!

Creating objects (modules)

- Destroy any previous object which has been created
- Create the object, use GEOM_NULL for unknown values as these can be bound later
- Initialise the edit list (instructions to **`geometry_viewer`**)
- Add additional information such as colour, normals
- Edit the geometry list to add the object

C code to create an object

```
GEOMedit_list *scatter_obj;

   *scatter_obj = (GEOMedit_list *)
      GEOMinit_edit_list(*scatter_obj);

   spheres = GEOMcreate_sphere (NULL,
         centres,
         GEOM_NULL,
         NULL,
         NULL,
         dims[0],
         GEOM_COPY_DATA);
```

C code to create an object (2)

```
GEOMadd_float_colors(spheres,colours,
                     dims[0],GEOM_COPY_DATA);

GEOMauto_transform (spheres);

GEOMedit_geometry(*scatter_obj,"sphere",
                  spheres);

GEOMdestroy_obj(spheres);
```


Hints and tips for GEOM

- You must not modify the GEOM object once it has been added to the edit list
- therefore always call **`GEOMdestroy_obj`** after **`GEOMedit_geometry`**
- It is quicker to create one object containing 10,000 lines than to keep adding a single line object with 1 line
- If your module executes successfully but **`geometry_viewer`** crashes then check the object your passing
- use the examples

Practical: ex4

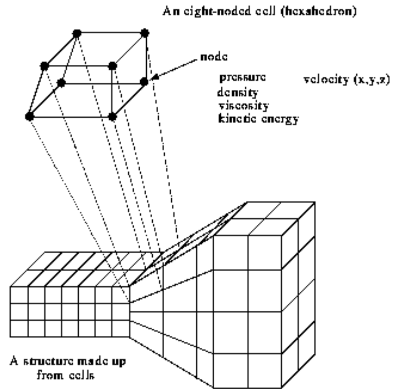
- this uses the datafile from ex3
- write a module which accepts **`1D 3-space irregular float`** data and outputs an GEOM edit list
- the module should create a sphere object
  - the sphere centres are the x,y,z info from the input
  - the colour of the sphere should use the data value
  - a dial to alter the radii should be used
- use the module generator

Do ex4

Unstructured Cell Data (UCD)

- used to represent discrete objects with data associated with them
- typically output from finite element analysis but many other applications have this scattered data
- the data type consists of:
  - nodes: contain coordinate information and data components
  - cells: built up by referencing nodes, can be 2D or 3D

An example UCD structure



Creating UCD

- there are two ways

  1. use the **`read_ucd`**
  2. write a module
- using **`read_ucd`** is easy but tied to changes made by AVS
- writing a module is **more** flexible

Using `read_ucd`

- This module reads a certain format of ASCII file and outputs a UCD structure
- the format is shown below:

Using `read_ucd` (2)

```
n-nodes n-cells ndata cdata sdata

node-id x y z

cell-id material-id cell-type cell-verts

n-data < size * ndata >

< label , units * ndata >

node-id < data * ndata >
```


An example UCD file (`hex.inp`)

```
8 1 1 0 0
1 0.000 0.000 1.000
2 1.000 0.000 1.000
3 1.000 1.000 1.000
4 0.000 1.000 1.000
5 0.000 0.000 0.000
6 1.000 0.000 0.000
7 1.000 1.000 0.000
8 0.000 1.000 0.000
1 1 hex 1 2 3 4 5 6 7 8
1 1
..................{cont'd over}
```


An example UCD file (`hex.inp`) (2)

```
....................{con't}
stress, lb/in**2
1 4999.9999
2 18749.9999
3 37500.0000
4 56250.0000
5 74999.9999
6 93750.0001
7 107500.0003
8 5000.0001
```


Writing a UCD module

- all files need **`#include <avs/ucd_defs.h>`**
- the C definiton is

```
UCD_structure **out_UCDstruct
```

- initially allocate the structure

```
*out_UCDstruct =
   (UCD_structure *)UCDstructure_alloc
```

Writing a UCD module (2)

- define the node x,y,z and extent

```
UCDstructure_set_node_position
UCDstructure_set_extent
```

- specify the cell types and node verticies

```
UCDcell_set_information
```


Writing a UCD module (2)

- specify the node data components

```
UCDstructure_set_node_components
UCDstructure_set_node_active
UCDstructure_set_node_labels
UCDstructure_set_node_units
```

- add the node data

```
UCDstructure_set_node_data
UCDstructure_set_node_minmax
```


Practical: ex5

- The example reads a defined data file and maps this onto the UCD datatype
- The example data file was generated from one of the example test runs from Harwell's Flow3d package. (Thanks to Dr I P Jones, CFDS, Harwell Labs, Didcot, UK)
- Another good example of using UCD is

```
/usr/avs/examples/read_ucd.c
```


The input data format for ex5

```
num-cells num-nodes num-materials

nodeid x y z

cellid < 8 node ids >

num-time-steps

nodeid velx vely velz pres dens visc KE
```

Coroutines

RECAP

- ❑ very useful for simulations
- ❑ generate data independently of the AVS flow executive
- ❑ number of routines for synchronising
- ❑ only use them when you have to

Coroutines

Scheme

```
Description routine:

    describes inputs, outputs,
    parameters in normal method

Main routine:

    Call AVScorout_init with ``Description routine'' name
```


Coroutines

Scheme (2)

```
Infinite Loop
{

    Call AVScorout_wait ( wait for a change )

    Call AVScorout_input ( to obtain input and params )

    Optionally

    Call AVSinput_changed or AVSparameter_changed

    COMPUTE PART

    Call AVScorout_output ( new output data )

}
```


Some real C code

```
int main(argc, argv)
{
    AVScorout_init(argc,argv,discs_desc);

    while (1) {

        if (!sleep) AVScorout_mark_changed() ;

        AVScorout_wait ();

        output_discs = (GEOMedit_list)
            GEOMinit_edit_list(output_discs);

        GEOMedit_geometry (output_discs,"disc",obj);

        AVScorout_output(output_discs);
}
```

Practical: ex6

- The coroutine outputs a number of spheres along a route with:
  - - x coordinate: *increasing linearly*
  - - y coordinate: *randomly changing +ve and -ve*
  - - z coordinate: *fixed*
- The simulation starts when the user click's the "sleep" button

What haven't we covered

- FORTRAN modules
- Using Colourmaps and CDK
- Memory Allocation debugging
- Upstream data
- Automatic connection of ports
- User defined data
- Modules and processes
- Command Line Interpreter
- But we only have ONE day

Closing comments

- The International AVS Center, North Carolina Supercomputing center collect and port public domain AVS modules as part of their remit
- Public domain modules are held under anonymous FTP:

  `avs.ncsc.org 128.109.178.23`
  `ftp.mcc.ac.uk 130.88.203.12 (shadow)`
- There are 600+ public domain modules
- If you write any modules then **please** submit them to the International AVS Center (avs.ncsc.org)