

AVS5 Advanced Application Development Techniques

Ian J. Curington
AVS / UNIRAS Ltd.
Montrose House, Chertsey Boulevard,
Hanworth Lane, Chertsey,
Surrey, KT169JX, UK
011-44-932-566608
ianc@avsuk.com

Abstract

While developing large applications using the AVS visual programming system, both networks and modules can become large and complex. Applying careful design with attention to detail can allow the application to have optimal performance and memory usage. The following hints and tips in selected areas may help in achieving this optimal performance. The topics are designed for module writers, where tuning and customizing the module application code is involved. Specific hints are given for Reader design, pointer pass-through, UCD construction, GEOM construction, AVSstatic, UCD_to_geom and cache methods for geometry producing modules.

1. Introduction

Why should I tune my network? If you are happy with the performance, memory usage, and user interface behaviour of your network, then don't. However, if you are concerned about such factors, then some advanced module development may be in order. The first priority for many AVS application developers is the data import and reader module design. Secondly, the user interface layout, control names, control behaviour, etc., is often the next area of focus. This leaves out considerations of the underlying network construction, and the chain of algorithms involved. This paper deals with the "Engine" at the core of your network.

Paying attention to the internal design of your network may be the key to creating a competitive application. At the start, take a look

at the modules that process or refer to the largest field or UCD structure. Follow the progress of this structure down through the network to the viewer. Which modules and algorithms are critical for speed or memory? If you are not sure, or want some quantitative measure, try running with "avs -separate -mod_time" while watching the system process table for the size of the module executing processes, and looking at start/stop timing messages to see which modules are taking the most time.

2. Smart Readers

The module at the top of your network that accesses external data and presents this to the rest of the AVS network is generally either a simple or a smart reader. A simple reader is one that opens an external file, reads that file, and passes the contents of the file to the field or UCD output port. If the input file is potentially larger than the memory of the machine, or beyond the processing ability of the downstream modules, then consider the construction of a "smart reader". A typical design includes the following components:

- open the file (only when filename changes),
- scan or compute an offset list in bytes of subsections of the file (such as scan-lines, planes, time steps, properties),
- add control parameters for the scan-line, planes, time steps, etc.,

- use "fseek" calls and binary "fread" calls on the file to quickly read the desired subsection,
- create a field or UCD output port that contains just this subsection.

This method can be potentially more efficient than the virtual memory handler, as it can use prior knowledge of the file format and anticipated access patterns. An example of such a reader is the IAC module "Read_SDRC".

3. Pass Through Pointers

In general, the standard AVS modules cannot assume that they have write permission on the input ports. This is because AVS may be running in read-only shared memory mode, or that several other modules may be referring to the same input data. If you are constructing an application where you can specify the rules about how modules may or may not be wired together, and can specify the correct permissions, then you may be able to use in-place operators or pointer references within your module. Even if you cannot make the above assumptions, if your module does not modify the data from the input to output ports, you can take advantage of passing a pointer through to the output port.

Simply passing the input port pointer out through the output port argument of a module will not work. This is because the AVS kernel has responsibility to keep track of memory allocation at output ports, and knows exactly which modules reference which parts of memory. When your module stops, AVS will try to free the output port memory. If the input and output ports point to the same place, it will get freed twice! The way around this is to save the output port pointers in static variables, and include an AVSset_destroy_proc function to "disconnect" the output from the input pointer, and set it to NULL. This way AVS will think you have already freed the output and will not do it twice. See the IAC module "field_mux" for a simple example of this technique.

If you want to modify the data passed through with such a pointer, then you can add the MODIFY_IN flag on the input port, and set the switch for "ReadOnlySharedMemory 0" in your .avsrc start-up file.

4. UCD Construction

Due to the large number of possible calls in the UCD library, it may not be clear which are the most useful to construct a new UCD output port system. Several of the standard examples show how to have local data areas from your application copied into the new UCD output structure using the "set" calls. However there is an alternative that avoids both the requirement for the local memory allocation, and the copy operation implied by the "UCDstructure_set..." calls. Notice that "UCDstructure_get_XXX" generally return a pointer to a pre-existing memory structure. Just such a structure is created by the "UCDstructure_alloc", so we can use this to get a direct pointer to the output storage area for nodes, positions, etc., and access this area directly in a portable way. The following code fragment shows the outline of this technique:

```
float *xc, *yc, *zc, *node_data;
*Output = (UCDstructure *)
    UCDstructure_alloc (...);
UCDstructure_get_node_positions
    (*Output, &xc, &yc, &zc);
UCDstructure_get_node_data
    (*Output, &node_data);
xc[i] = ...
node_data[i] = ...
```

Notice no "malloc" calls, and no copy operations! See the IAC module "ACE_to_UCD" for a working example of this technique.

5. Fast GEOM Construction

Many people are not aware of the different performance implications of passing the various flags into the GEOMETRY library calls. By making the choice of flags appropriate for your application, you can save both compute time and memory. Inevitably, the processing of geometry in the AVS network involves a creation module and a viewer module. By changing the GEOM flags, you can

1. Optimize for most efficient rendering in the Geometry Viewer with the smallest polygon count.
2. Optimize for fastest geometry creation speed, at the expense of rendering (a few more polygons).

An expensive operation in the creation calls is to search the current node list to merge in shared nodes where possible. This can take some time with a large structure, but will minimize the number of nodes in the output, and allow for smooth shading by averaging vertex normals over adjacent faces. An example of how to control this is shown below:

```
obj=GEOMcreate_obj(GEOM_POLYHEDRON,0);
GEOMadd_disjoint_polygon(obj,
    verts, NULL, colors, numv,
    GEOM_SHARED,0);
GEOMgen_normals(obj, 0);
GEOMcvt_polyh_to_polytri
    (obj,GEOM_SURFACE );
```

The alternative is to avoid the search, and simply add duplicate vertices into the list. This is very fast to do, but makes a longer list, and does not allow for the smooth shading mentioned above. The rendering time is unlikely to double, while the creation time can drop significantly. The method used in this case is shown below:

```
obj=GEOMcreate_obj(GEOM_POLYHEDRON,0);
GEOMadd_disjoint_polygon(obj,-
    verts, NULL, colors, numv,
    GEOM_NOT_SHARED,0);
GEOMgen_normals(obj, 0);
GEOMcvt_polyh_to_polytri(obj,
    GEOM_SURFACE |
    GEOM_NO_CONNECTIVITY );
```

In both cases, it is important NOT to use "GEOM_WIREFRAME I GEOM_SURFACE" unless you are writing a very simple general purpose module, as this forces TWO complete versions of your geometry to be created. This mode will allow "outline gouraud" mode, or geometry viewer selection of shaded/wireframe display, but is carrying a significant memory overhead. It is advisable to make GEOM_WIREFRAME or GEOM_SURFACE flag controlled by a user parameter for best speed and memory efficiency. In the case where you expect to only

use wireframe or no lighting, the GEOMgen normals call can also be avoided for extra speed. For an example of this technique see the IAC module "ucd_cell_geom".

6. AVSstatic

To get good efficiency out of a large application, it is important to link all the modules into one process, using the "mongo" system of AVS. The AVS flow executive then calls individual modules inside this process as needed. In addition, if you have more than one copy of a module in your network, you do not want AVS to spawn extra processes, so your module needs to be re-entrant. To get this behaviour you must set the COOPERATIVE I REENTRANT flags in the module description. However, a common design problem then arises - how to manage static variables. Any static variables will be trashed if the same code is called re-entrantly in multiple contexts. The AVS flow executive provides a very nice service to manage static areas, called "AVSstatic". Using this handle, the AVS flow executive will keep track of context specific static pointers for you. All you need is to allocate some space, and set the AVSstatic pointer to reference it! The following template illustrates the basics of this technique:

```
/* AVSstatic illustration example */
extern char *AVSstatic;

typedef struct _xxx_stats {
    int *val_list;
} Xxx_Stats_Type;

/* --- module compute routine */
xxx_compute(...)
{
    Xxx_Stats_Type *xxx_stats;
    if TAVSstatic == NULL)
        xxx_stats = (Xxx_Stats_Type *)
            malloc(sizeof(Xxx_Stats_Type));
    else
        xxx_stats =
            (Xxx_Stats_Type *)AVSstatic;
    AVSstatic = (char *)xxx_stats;
    xxx_stats->val_list = input->val_list;
    ....
}

/* --- module init proc */
xxx_init()
{
    AVSstatic = (char *)0;
}

/* --- module destroy proc */
xxx_destroy()
```

```

{
Xxx_Stats_Type *xxx_stats;
if (AVSstatic == NULL) return;
xxx_stats = (Xxx_Stats_Type *)
    AVSstatic;
free (xxx_stats);
}

```

7. Shifting Gears on UCD to Geom

There is a very nice mode in the `ucd_to_geom` module which can be up to 10 times faster than normal! With the "Save Geometry" button on, the module will only create the expensive geometry structures once, save them, then subsequently modify them in place as long as only the property values are changing, like colormap or node data. This mechanism is triggered by the "mesh id" in the UCD structure header. Any time a new UCD is allocated, copied, modified, etc., this number is incremented. If the UCD cell connectivity list, number of nodes, etc., remain constant, then it is possible to keep the "mesh id" the same as before. The `ucd_to_geom` module checks for this case, and will use its saved copy if possible. A good example of this is where you time varying temperature over a static shape. Several commercial AVS developers have exploited this facility for very fast update over large structures. A good example demo of this is the IAC module "ACE_to_UCD".

8. Cache your own GEOM structures

In many applications, such as Oil Reservoir analysis, Fluid Dynamics and Finite Element Structural vibration analysis, the connectivity or topology of the model changes infrequently or never, while time varying properties of the structure are displayed.

In the previous section we examined how to use the cache system built into the standard AVS module `ucd_to_geom` when only values/colors are changing. The basic mechanism behind this is potentially much more general than just changing colors on UCD objects. There is no

reason why this technique could not be applied to any situation where you have a geometry output port, and you know that the connectivity of your structure changes infrequently compared to values, colors, vertex positions, scale, or texture coordinates. To achieve significant speedup, take the following simple steps:

1. Create a GEOMObject structure, saved in a static structure.
2. Add vertex data index tags to uniquely identify your own data.
3. Convert the Object to an optimal polytriangle strip structure.
4. Use the vertex data index to directly update the polytriangle strip.
5. Send the edited Object to the viewer.

The geometry data structure is designed to be managed by the GEOM library calls, so direct access to it is not documented, supported or encouraged. However, the `geom.h` structure has not changed in many releases, and a number of modules at the IAC also use direct access. This includes any module that has GEOM as an input port.

The following example draws five sides of a small cube, saving the converted polytriangle strip. After the first time, any time the "Scalar" parameter changes, a vertex is moved on the object and is re-sent to the viewer. The compute time of the application will be the sum of the Geometry Viewer, this module and any AVS flow executive overhead. By using this technique the Geometry Viewer time will be unaffected, except by making a more or less efficient structure for it to render, while the compute time in the geometry creation module could be as little as 10 percent of the first time through.

```

/*
 * Example of how to cache geometry structures for dynamic update
 * of vertex positions [ianc@avsuk.com, Feb 94]
 */
int geom_compute(Geom_Out, Scalar )
    GEOMedit_list **Geom_Out;
    float *Scalar;
{
    int i, j, v;
    static GEOMobj *obj = NULL;          /* cached Geometry */

    static float xc[8] = { 0.0, 1.0, 1.0, 0.0, 0.0, 1.0, 1.0, 0.0 };
    static float yc[8] = { 0.0, 0.0, 1.0, 1.0, 0.0, 0.0, 1.0, 1.0 };
    static float zc[8] = { 0.0, 0.0, 0.0, 0.0, 1.0, 1.0, 1.0, 1.0 };

    static int faces_count = 5;

    static face_list[5][4] = { 0, 1, 5, 4,
                                1, 2, 6, 5,
                                2, 3, 7, 6,
                                3, 0, 4, 7,
                                3, 2, 1, 0 };

    float verts[4][3];          /* local temporary copy */
    int vdata[4];

    if (obj == NULL) /* Create the object the first time */
    {
        *Geom_Out = (GEOMedit_list *)GEOMinit_edit_list(*Geom_Out);
        obj = GEOMcreate_obj(GEOM_POLYHEDRON, GEOM_NULL); /* open new */

        for (i = 0; i < faces_count; i++) /* loop over each face */
        {
            for (j = 0; j < 4; j++) /* loop over each vertex */
            {
                v = face_list[i][j];
                verts[j][0] = xc[v]; /* make local copy */
                verts[j][1] = yc[v];
                verts[j][2] = zc[v];
                vdata[j] = v; /* remember index! */
            } /* Add one poly at a time */
            GEOMadd_disjoint_polygon(obj, verts, NULL, NULL, 4,
                                     GEOM_NOT_SHARED, 0);
            GEOMadd_vertex_data(obj, vdata, 4, GEOM_COPY_DATA); /* index! */
        }

        GEOMgen_normals(obj, GEOM_FACET_NORMALS);
        GEOMcvt"polyh_to_polytri"(obj, GEOM_SURFACE | GEOM_NO_CONNECTIVITY);
    }
    else /* If already Created, update inplace */
    {
        xc[0] = *Scalar; /* move the first vertex */
        yc[0] = *Scalar;
        zc[0] = *Scalar;

        for (i = 0; i < PT(obj).npts; i++) /* for each polytri strip */
        {
            /* PT is a macro from geom.h */
            for (j = 0; j < PT(obj).ptverts[i].n; j++)
            {
                v = PT(obj).ptvdata[i].1[j]; /* get back index */
                PT(obj).ptverts[i].1[3*j+0] = xc[v]; /* update vert */
                PT(obj).ptverts[i].1[3*j+1] = yc[v];
                PT(obj).ptverts[i].1[3*j+2] = zc[v];
            }
        }

        GEOMedit_geometry(*Geom_Out, "geom_cache", obj); /* send to viewer */

        return(1);
    }
}

```