Scientific Visualization Programming Workshop

Science & Technology Support Group
High Performance Computing
Ohio Supercomputer Center
1224 Kinnear Road
Columbus, OH 43212
Visualization Workshop Agenda

• Day 1: An introduction to AVS Express
  – Morning session, cover the introductory concepts of AVS Express V Programming Language
  – Afternoon session, introduction to C programming techniques.
  – Available through the statewide licensing program through OSC

• Day 2: An introduction to programming plugins for VolSuite
  – Morning session, cover basic structure of VolSuite and concepts of developing modules
  – Developed at OSC, Jason Bryan, Thomas Kerwin, and Dennis Sessanna
  – Available through open-source distribution from OSC
VolSuite

- Open-source, portable, and highly extensible
- Dynamically loadable modules (plugins)
- High-level abstraction of core classes with emphasis on polymorphism
- Built-in network support
- Virtually limitless extensibility for integrating algorithms for visualization, analysis and modification

Isosurface function implemented for Metaballs plugin to VolSuite. Developed at OSC.
AVS Express

STM (See The Molecule) module, developed by Ken Flurchick, et.al.
Course Outline

- The V Command Processor (VCP)
- C Programming and Add Module
- Additional Resources
What is AVS/Express?

- An object-oriented tool for visualization
- Extremely powerful and flexible
- Modular
- Hierarchical
- Extensible
- Many predefined application components
- Can analyze, display, manipulate and interact with data
AVS/Express Editions

• Visualization Edition
  – Basic tool for scientific and data visualization
  – Will sometimes refer to this as “VizExpress” or “vxp” or “Express”

• Developer’s Edition
  – All the functionality of the Visualization Edition, plus
    • Access to low level functionality
    • Ability to create stand-alone applications

• Both editions have the same fundamental architecture
• We’ll be talking almost entirely about VizExpress...
Basic Architecture

- Access to the OM occurs via three separate interfaces
The V Programming Language

- A general language for interfacing with the Object Manager
- Can be used for all AVS/Express programming
  - Object definitions, templates, active instanced objects
  - Data references (connections)
  - Parameters and their values
- Describes the structure of modules
- Also an interactive tool for debugging or manipulating the AVS/Express state
- By default the V command processor (VCP) is launched on startup
  - Unix users should not run AVS/Express in the background!
The V Command Processor

AVS/Express Visualization Edition
Version: 5.0 R992fcs sg6
Project: /tmp/XPtrain

----------------------------------------
OM(Root)  -> MultiWindowApp{
warning: jumping from: Root to Root.Applications.MultiWindowApp
OM(MultiWindowApp)  ->  $list
AU.MultiWindowApp.UI UI
GDM.Uviewer Uviewer
Field Field
MODS.select_cells select_cells
MODS.extract_mesh extract_mesh#1
OM(MultiWindowApp)  ->  Field{
OM(Field)  ->  $list
int nnodes
int nspace
Grid.coordinates coordinates
int ncell_sets
Field.cell_set cell_set
Xform.xform+opt xform
int nnodes_data
Node_Data.node_data node_data
OM(Field)  ->  }
OM(MultiWindowApp)  ->  

User commands
Application Programmer Interface

- You can extend the capabilities of AVS/Express by writing your own modules
- An API provides the interface to the Object Manager
- There is an Add Module tool to help with the creation of modules
The AVS/Express Data Model

- AVS/Express uses a single unified structure to represent data: the field
- It describes:
  - Your data
  - The coordinate mesh on which it lives
  - Geometry data
- AVS/Express visualization objects know how to work with fields
- General structure:

\[
\text{Field} = \text{Mesh} + \text{Data}
\]
Availability through OSC

- AVS/Express is available on the SGI Origin 2000 and Cluster systems at OSC
- It is also available through OSC’s Statewide Software Licensing Program for local use
- Available for many platforms, including
  - Compaq Alpha (Tru64)  Intel Windows NT 4.0
  - HP-UX 10.20 or higher  Intel Windows 95/98/00
  - IBM AIX 4.3.1  Digital Alpha Windows
  - SGI Irix 6.5 (n32 or n64 ABIs)  Sun Solaris 2.7
- For PCs, requires a Pentium-class system and at least 32 MB of memory
- Can use OpenGL for hardware rendering
Running AVS/Express under Unix

- AVS/Express requires that certain environment variables be set before it can be run. The most important of these are
  - `LM_LICENSE_FILE`
    - Must point to the FLEXlm license file
  - `XP_FEATURE`
    - Must be `EXPRESS` (for the Developer’s Edition) or `VIZ_EXPRESS` (for the Visualization Edition)
  - `MACHINE`
    - AVS/Express machine architecture specification (`sgi`, `sgN32`, ...)
  - `LD_LIBRARY_PATH`
    - Should point to the directory `lib` in the AVS/Express installation directory
- Ideally this is done automatically by a script (here we use “vxp”), which sets these variables and then runs the program
Running AVS/Express under Windows

- AVS/Express requires that certain environment variables be set before it can be run. For Win9x/NT, the most important of these are
  - **LM_LICENSE_FILE**
    - Name of the FLEXlm license file (e.g. `C:\flexlm\license.dat`)
  - **XP_FEATURE**
    - Must be **EXPRESS** (for the Developer’s Edition) or **VIZ_EXPRESS** (for the Visualization Edition)
  - **MACHINE**
    - Should be set to `pc`

- Ideally this is done automatically by a script which sets these variables and then runs the program
- If you obtain AVS/Express through OSC, a basic script `run_xp.bat` is provided
Creating an Application

• Single-window Data Viewer
  – Viewer and user interface are in a single window

• Multi-window Data Viewer
  – Viewer and user interface are in separate windows

• Application
  – No default data viewer

• Module Stack
  – Application without a default viewer but including the “module stack” which allows access to the module UI panels

• Scratch Pad
  – Does not activate or “instance” objects
  – Used for editing module templates
Basic Object Types

- **Primitive objects**
  - int, float, string, ...
- **Hierarchical objects**
  - macro, group
- **Arrays of objects** can contain primitives, hierarchical objects, or combinations
- **The icon in the upper left corner** identifies the object type
Object Types

- Libraries
  - Collections of objects
- Macros
  - Combinations of modules and links
- Modules
  - Parameters and methods that interface to application code
- Parameters
  - Data to be operated on by methods
- Methods
  - Interface to application code
- Links
  - Data connections into/out of objects
The V Command Processor (VCP)

- Overview
- Navigating the object hierarchy
- Basic V commands
- Intrinsic functions
The V Command Processor (VCP)

- AVS/Express provides its own language, called V, for interacting with the Object Manager
- It can be used to define and manipulate the state of objects
  - What objects are instanced
  - Data connections (references)
  - Current parameter values
- The VCP provides a command prompt that allows you to interactively enter V commands
- You can use this to traverse the object hierarchy, display and modify object values, and create and destroy objects
- Very useful for examining the values of sub-objects as an aid in debugging
Navigating the Object Hierarchy

- When you start AVS/Express, the OM is currently set to edit Root, the object at the top of the hierarchy.
- You navigate down to sub-objects by entering the name of the object followed by a left curly brace ({}).
- To close the current object and move back up to the previous level, enter };
Basic V Commands

- **$list**
  - List the sub-objects of the current object

- **$int, $ptr, $real, $str**
  - Print the value of a scalar data object of the specified type

- **$get_array**
  - Print an array object

- **$sub_array**
  - An interactive command that allows you to print a subset of an array. Useful if the array is very large.

- **$print**
  - Print the full definition of an object, including references to other objects and any value it currently has
Examples

```
OM(Root)           -> MultiWindowApp{
   warning: jumping from: Root to Root.Applications.MultiWindowApp
OM(MultiWindowApp) -> $list
AU.MultiWindowApp.UI UI
GDM.Uviewer2D Uviewer2D
int y
int x
AGHLM.Graph Graph
AGHLM.GraphViewport GraphViewport
AGHLM.GraphLegend GraphLegend
AGHLM.GraphWorld GraphWorld
AGHLM.XYAxis XYAxis
AGHLM.GraphWorld GraphWorld#1
OM(MultiWindowApp) -> $get_array x
{0,1,2,3,4,
  5,6,7,8,9}
OM(MultiWindowApp) -> $print y
int y<NEportLevels={0,1},NEx=88.,NEy=88.>[10] => {
  1,3,4,2,6,5,7,3,1,4
};
OM(MultiWindowApp) -> }
OM(Root)           ->
```
Intrinsic Functions

- V has a number of intrinsic functions you can use in modules
- Basic math functions (sin, cos, tan, exp, etc.) are supported
- There are also some useful functions you can use to set up and manipulate arrays, among them
  - `init_array(size, start, end)`
    - Creates an array of size uniformly spaced elements starting at `start` and ending at `end`
  - `concat_array(ar1, ar2, ...)`
    - Concatenate two or more arrays
  - `prod(ar1, ar2, ...)`
    - Return the product of all elements in the listed arrays
  - `sum(ar1, ar2, ...)`
    - Return the sum of all elements in the listed arrays
Managing Projects

- AVS/Express projects
- Creating a project
- The `avsenv` file
AVS/Express Projects

• Projects are the AVS/Express mechanism for organizing applications
  – Defines your development and execution environment
• Each project has a specific location in the directory tree
• You are always working in a project
  – Default project is located in the AVS/Express installation directory
    • Generally not writable on a Unix system
    • May well be writable under Windows!
• The project directory contains
  – Custom macros and modules
  – Completed applications (networks)
  – All configured libraries
• Changes to the default (installation) project are tracked
The avsenv File

- The file avsenv in the project directory indicates the project’s derivation hierarchy

```
gromit$ more avsenv
#   WARNING: this file is program generated.
#   Remove these two lines to protect the file from overwrites.
XP_PATH=/tmp/XPtrain /usr/local/express64
```

- This indicates that the project /tmp/XPtrain is derived from the default (read-only) project /usr/local/express64

- It is often necessary to modify this file when installing a project from another user, as the directories are not necessarily correct on your machine

- This should be the only thing you have to do, assuming all file references in modules are relative (to the project directory)
User Defined Data

- You can also define your own custom data types
- Create an AVS/Express object to represent it, e.g. using the `Group` macro

```c
# Patient Scan Data
group {
    int pid
    string pname
    Field scandata
    int DateofScan
}
```

- Other modules could process field
- You could create a User Data Type for holding the data
Third Party Modules - The IAC

• The International AVS Centre
  – Located at the Manchester Visualization Center of the University of Manchester, UK

• IAC objectives:
  – support and expand the AVS user base
  – foster discipline-specific module development
  – act as a liaison between users and vendors

• Large (and growing) repository of user-supplied modules
• They also maintain a mailing list and provide a quarterly newsletter (AVS Network News)
• Provide assistance for AVS/Express users
• Adheres to the AVS/Express project mechanism
IAC Project Files

- Allows straightforward integration of IAC projects with your own
- Basic instructions are in the Project Organization Guidelines
- The IAC base project files only need to be incorporated once per AVS/Express project
- Additional IAC modules can then be added easily
IAC Project Files

- Create your own project, or start from an existing project
- Download the IAC project files, which will create subdirectories `iac_lib` and `iac_proj`, among others
- Modify files (`templ.v`) according to the instructions in the file `INSTALL`
- A separate IAC library is added
V language :: Introduction

- V is AVS/Express' language for defining objects and applications. V is a textual alternative to the Network Editor.
- AVS/Express stores all object definitions as V text files or binary V files. This includes objects you create and save in the Network Editor.
V components :: Introduction

• A V file can contain comments, statements, and commands.
• Comments
  – Document a V file. AVS/Express ignores them.
• Statements
  – Create, modify, and delete objects.
• Commands
  – Perform special operations. For example:
    • $print - prints an object's definition.
    • $include - includes a V file.
    • #define - allows you to define conditional code.
V Command Processor :: Introduction

• The V Command Processor (VCP) is a tool for viewing and maintaining V code within AVS/Express.
• The VCP allows you to
  – enter V statements and commands
  – navigate the object hierarchy
V Command Processor

- VCP allows you to interactively manipulate AVS/Express objects using the V language.
- VCP gives you a prompt that shows you the name of the object that you are currently editing.
- Uses indentation to indicate the number of levels into the object hierarchy that the current object lives.
- Each command is executed when you terminate a command with a newline character.
- To begin, here are several examples of the VCP.
- Enter the text to the right of the -> prompt.
V Command Processor

- Example 1: current object is Root.
- VCP lines create objects var1 and UIfield as immediate subobjects of SingleWindowApp
- Set the value of UIfield's x subobject, then navigate back up to Root

```plaintext
OM(Root) -> Applications {
  OM(Applications) -> SingleWindowApp {
    OM(SingleWindowApp) -> int var1;
    OM(SingleWindowApp) -> UIfield UIfield {
      OM(UIfield) -> x = 10;
      OM(UIfield) -> 
    }
    OM(SingleWindowApp) -> 
  }
  OM(Applications) -> 
}
OM(Root) ->
```
V Command Processor

• Example 2
  – Assume that SingleWindowApp has a subobject called UIfield
  – The following VCP lines print the value of UIfield's x subobject
  – Navigate back up to Root

```
OM(Root) -> SingleWindowApp{
  OM warning: jumping from: Root to Root.Applications.SingleWindowApp
  OM(SingleWindowApp) -> UIfield{
    OM(UIfield) -> $real x
    10
    OM(SingleWindowApp) -> }
  OM(SingleWindowApp) -> }
OM(Root) ->
```
V Command Processor

- **Example 3**
  - The following VCP lines create a group object called grp1
  - Create subobjects for the group
  - Navigate back up to Root

```plaintext
OM(Root) -> SingleWindowApp{
  OM warning: jumping from: Root to Root.Applications.SingleWindowApp
  OM(SingleWindowApp) -> group grp1{
    OM(grp1) -> int x, y, z;
    OM(grp1) -> }
  OM(SingleWindowApp) -> }
OM(Root) ->
```
V Command Processor

- **Example 4**
  - Assume that SingleWindowApp has a subobject called grp1
  - The following lines navigate from Root to grp1
  - Modify grp1's x subobject
  - Add a subobject called w
  - Navigate back up to Root

```om
OM(Root) -> SingleWindowApp.grp1 {
OM warning: jumping from: Root to Root.Applications.SingleWindowApp.grp1
OM(grp1) -> x = 3;
OM(grp1) -> int w;
OM(grp1) -> }
OM(Root) ->
```
VCP’s appearance

- The VCP appears in the window where you started AVS/Express. This is the VCP prompt:
- The name of the current object appears in parentheses
- When you start AVS/Express, the current object is always Root
- This is the VCP prompt:

OM(Root) ->
Entering statements and commands

- You enter V statements and commands to the right of the prompt
- Press Return at the end of a line
- A V statement can continue across lines.
- V command must appear all on one line
- For example, here is the $list command that lists the subobjects of the current object:

  OM(Root) -> $list
Navigating object hierarchy

- The VCP considers one object in the object hierarchy to be current
- At start-up, the current object is Root
- The current object is important because commands and statements you enter work in the context of the current object
  - Several commands, such as $print and $list, operate on a particular object. The object must be the current object or is defined to be relative to the current object
  - In a create statement, the object you create becomes an immediate subobject of the current object
  - In a modify or delete object statement, the target object is defined to be relative to the current object
- The name of the current object appears in parentheses in the VCP prompt
Navigating (cont’d)

• You change the current object by navigating the object hierarchy.
• The object you navigate to becomes the current object.
• To navigate:
  – Enter the name of an immediate subobject, followed by an open brace.
  – You can list immediate subobjects with the $list command.
• Down one level:
  – For example, assume that grp1 is an immediate subobject of SingleWindowApp:

```plaintext
OM(SingleWindowApp) -> grp1 {
OM(grp1) ->
```
Navigating (cont’d)

• Up one level
  – Enter a close brace followed by a semicolon

  \[ OM(grp1) \rightarrow \{ \}; \]
  \[ OM(SingleWindowApp) \rightarrow \]

• Down several levels at once
  – Enter a pathname followed by an open brace
  – The pathname starts with an immediate subobject
  – For example, assume that grp2 is an immediate subobject of grp1

  \[ OM(SingleWindowApp) \rightarrow grp1.grp2 \{ \]
  \[ OM(grp2) \rightarrow \]
Navigating (cont’d)

• Up several levels at once
  – After navigating down several levels at once, enter a close brace followed by a semicolon
  – For example:

  OM(grp2) -> }; 
  OM(SingleWindowApp) ->
Navigating (cont’d)

• Navigating through Applications and libraries
  – The Applications object and many library objects have the global characteristic
  – In the context of VCP navigation, global means that you can navigate to an immediate subobject directly
  – AVS/Express prints a warning message
  – All objects defined with the base type library are global
  – You can tell if other objects are global if the definition of their type contains +global
  – For example, Applications has the global characteristic, so you can navigate directly from Root to SingleWindowApp:

        OM(Root) -> SingleWindowApp {
        OM warning: jumping from: Root to Root.Applications.SingleWindowApp
        OM(SingleWindowApp) ->
Navigating (cont’d)

• If Applications did not have the global characteristic, you would have to do one of the following:

```
OM(Root) -> Applications {
OM(Applications) -> SingleWindowApp {
OM(SingleWindowApp) ->
```

• or

```
OM(Root) -> Applications.SingleWindowApp {
OM(SingleWindowApp) ->
```
Navigating (cont’d)

• You can navigate through several levels of the object hierarchy at once
• The object, at each level, must have the global characteristic
• This is often the case with libraries:

```om
OM(Root) -> UIButton {
OM warning: jumping from: Root to Root.Templates.UI.Controls.UIbutton
OM(UIButton) -> 
};
```
Navigating (cont’d)

• When necessary, you can specify a partial pathname
• In the following example, you navigate to Templates.MODS.Mappers.isosurface
• Templates and Mappers have the global characteristic, so you can omit them in the pathname
• MODS does not, so must be included:

```
OM(OM(root) -> MODS.isosurface { 
OM warning: jumping from: Root to Root.Templates.MODS.Mappers.isosurface
OM(OM(isosurface) ->
```
Navigating (cont’d)

• Navigation commands are really V modify statements
• When you navigate down or up the object hierarchy, you are really entering V modify statements
• For example, assume you have a group object called grp1 and you want to assign a value to subobject x:

```plaintext
grp1 {
  x = 3;
};
```

• The object's name and the open brace indicate that you want to work with the object's immediate subobjects
• The close brace and semicolon at the end indicate that you are finished working with the subobjects.
Navigating (cont’d)

• V statement rules apply
  – Because navigation commands are really modified V statements, V statement rules apply
  – You can split a navigation command across several lines
  – Or enter multiple navigation commands on the same line

• Example 1
  – Here is how you typically enter navigation commands:

```plaintext
OM(Root) -> SingleWindowApp{
  OM(SingleWindowApp) -> grp1 {
    OM(grp1) -> x = 3;
    OM(grp1) -> int w;
    OM(grp1) -> }
  OM(SingleWindowApp) -> }
OM(Root) ->
```
Navigating (cont’d)

• Example 2
  – The same example, but with the object name and open brace on separate lines

```om
OM(Root) -> SingleWindowApp
> {
OM(SingleWindowApp) -> grp1
> {
OM(grp1) -> x = 3;
OM(grp1) -> int w;
OM(grp1) ->
OM(SingleWindowApp) ->
OM(Root) ->
```
Navigating (cont’d)

• Example 3
  – Here is the same example, but with consecutive }; on the same line

```
OM(Root) -> SingleWindowApp {
OM(SingleWindowApp) -> grp1 {
OM(grp1) -> x = 3;
OM(grp1) -> int w;
OM(grp1) -> }
OM(Root) ->
```

• Example 4
  – Same example, but with everything strung together on the same line. It is difficult to read, but is valid V syntax

```
OM(Root) -> SingleWindowApp{grp1{x=3;int w};};
OM(Root) ->
```
V commands

- V commands perform special operations
  - $print prints an object's definition
  - $include includes a V file
  - #define allows you to define conditional code.

- General coding rules
  - Format
    - Commands consist of a command name followed by zero or more arguments
    - Commands always begin with a $ character (except for the #define command)
    - Commands do not end with a semicolon.

- No continuation lines
  - Commands can start anywhere on a line, but cannot be split across lines

- Separators
  - A command name and each of its arguments must be space- or tab-separated

- Case sensitive
  - AVS/Express is case sensitive: $print is a command, but $Print is not.
V commands (cont’d)

- Specifying an object name
  - Most commands take an optional object name as one of their arguments:
  - Default object
    - The object name is usually optional, and defaults to the current object
    - In the syntax diagram for an object, optional arguments appear in brackets.
    - For example: $print [object]
  - Subobjects
    - When you do specify an object name, you must reference a subobject of the current object
    - To refer to an immediate subobject, specify an unqualified name
    - To refer to a lower-level subobject, either navigate to the object before issuing the command, or specify a pathname
V commands (cont’d)

• Redirecting output to a file
  – You can redirect any command's output to a file, by specifying UNIX-style redirection at the end of the command
  – Valid on Windows systems as well

• The following example redirects the output of the $save command to a file
  – $save prints an object's V definition
  – the command with redirection saves the object's V definition to a file
  – is equivalent to the File->Save Object pull-down command in the Network Editor

```
$save myapp > myV/myapp.v
```

• Default directory is the directory in which AVS/Express started.
### V commands (cont’d)

- Summary of commands by category

<table>
<thead>
<tr>
<th>Category</th>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Printing/saving an object's definition</td>
<td>$\text{print}$, $\text{dprint}$, $\text{save}$, $\text{save}_\text{usr}$</td>
<td>Print/save an object's definition.</td>
</tr>
<tr>
<td>Printing an object's value</td>
<td>$\text{int}$, $\text{ptr}$, $\text{real}$, $\text{str}$</td>
<td>Print a scalar data object's value.</td>
</tr>
<tr>
<td></td>
<td>$\text{get_array}$, $\text{sub_array}$</td>
<td>Print an array object's value.</td>
</tr>
<tr>
<td>Printing/setting object references</td>
<td>$\text{link}$</td>
<td>Connect objects.</td>
</tr>
<tr>
<td></td>
<td>$\text{obj_pval}$, $\text{obj_val}$, $\text{obj_ref}$, $\text{obj_ref_ref}$</td>
<td>Print connected objects.</td>
</tr>
<tr>
<td></td>
<td>$\text{refcnt}$</td>
<td>Print number of references to object.</td>
</tr>
<tr>
<td></td>
<td>$\text{refs_to}$</td>
<td>Print objects connected to.</td>
</tr>
</tbody>
</table>
### V commands (cont’d)

- Summary of commands by category

<table>
<thead>
<tr>
<th>Category</th>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Printing/setting other object characteristics</td>
<td>$array_dims,</td>
<td>Print array dimensions and size.</td>
</tr>
<tr>
<td></td>
<td>$array_size</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$get_data_type</td>
<td>Print data type.</td>
</tr>
<tr>
<td></td>
<td>$get_obj_att</td>
<td>Print an attribute.</td>
</tr>
<tr>
<td></td>
<td>$get_obj_prop</td>
<td>Print a property.</td>
</tr>
<tr>
<td></td>
<td>$obj_path</td>
<td>Print full pathname.</td>
</tr>
<tr>
<td></td>
<td>$set_data_type</td>
<td>Set data type.</td>
</tr>
<tr>
<td></td>
<td>$set_obj_name</td>
<td>Set name.</td>
</tr>
<tr>
<td></td>
<td>$user_template</td>
<td>Print object's template name.</td>
</tr>
<tr>
<td></td>
<td>$move</td>
<td>Moves an object in the hierarchy.</td>
</tr>
</tbody>
</table>
### V commands (cont’d)

- **Summary of commands by category**

<table>
<thead>
<tr>
<th>Category</th>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Listing subobjects</td>
<td>$list</td>
<td>List immediate subobjects.</td>
</tr>
<tr>
<td>Validating and debugging</td>
<td>$cur_seq, $obj_seq,</td>
<td>Print sequence number.</td>
</tr>
<tr>
<td></td>
<td>$obj_seq_base, $obj_seq_ptrs</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$echo</td>
<td>Echo string.</td>
</tr>
<tr>
<td></td>
<td>$obj_id</td>
<td>Print object's id and process id.</td>
</tr>
<tr>
<td></td>
<td>$match, $ematch</td>
<td>Test whether objects match.</td>
</tr>
<tr>
<td></td>
<td>$notify</td>
<td>List notifications on an object.</td>
</tr>
<tr>
<td></td>
<td>$deps</td>
<td>List dependencies for a method.</td>
</tr>
<tr>
<td></td>
<td>$resolve</td>
<td>Resolve object references.</td>
</tr>
</tbody>
</table>
V commands (cont’d)

- Summary of commands by category

<table>
<thead>
<tr>
<th>Category</th>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Validating and debugging</td>
<td>$set_trace,</td>
<td>Trace certain events.</td>
</tr>
<tr>
<td>(cont’d)</td>
<td>$unset_trace</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$valid</td>
<td>Test whether object has valid value.</td>
</tr>
<tr>
<td></td>
<td>$obj_proc</td>
<td>Get process id of an object.</td>
</tr>
<tr>
<td>Controlling execution</td>
<td>$pause</td>
<td>Pause execution.</td>
</tr>
<tr>
<td></td>
<td>$push, $pop</td>
<td>Delay delivery of events.</td>
</tr>
<tr>
<td>Including V files</td>
<td>$include</td>
<td>Include a V file.</td>
</tr>
<tr>
<td>Making code conditional</td>
<td>#define, #undef, #ifdef, #else, #endif</td>
<td>Conditionally parse a block of V code.</td>
</tr>
</tbody>
</table>
### Summary of commands by category

<table>
<thead>
<tr>
<th>Category</th>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Others</td>
<td>$help</td>
<td>Print V-command help.</td>
</tr>
<tr>
<td></td>
<td>$open</td>
<td>Parse a buffered object.</td>
</tr>
<tr>
<td></td>
<td>$quit</td>
<td>Quit AVS/Express.</td>
</tr>
<tr>
<td></td>
<td>$local_quit</td>
<td>Quit local process only.</td>
</tr>
<tr>
<td></td>
<td>$compile</td>
<td>Compile an object.</td>
</tr>
<tr>
<td></td>
<td>$generate</td>
<td>Generate code for an object.</td>
</tr>
<tr>
<td></td>
<td>$type</td>
<td>Print AVS/Express base types.</td>
</tr>
<tr>
<td></td>
<td>$server_info</td>
<td>Print server information.</td>
</tr>
<tr>
<td></td>
<td>$shell</td>
<td>Execute a shell command.</td>
</tr>
</tbody>
</table>
V commands (cont’d)

- Summary of commands by category

<table>
<thead>
<tr>
<th>Category</th>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Others (cont’d)</td>
<td>$timer_start,</td>
<td>Get timing information.</td>
</tr>
<tr>
<td></td>
<td>$timer_get</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$set_file_env</td>
<td>Make an environment variable accessible as an AVS/Express file</td>
</tr>
<tr>
<td></td>
<td></td>
<td>variable.</td>
</tr>
</tbody>
</table>
V comments

• V comments document a V file.
• You can insert comments into the body of a V file.
• AVS/Express ignores comments unless they are defined with a special syntax.
• You can use:
  – C-style comments
  – C++-style comments
  – Special syntax comments
• C-style comments
  – C-style comment begins with the delimiter /* and ends with the delimiter */
  – The delimiters can appear anywhere in the code and can span multiple lines
  – AVS/Express treats everything between the delimiters as part of the comment
V comments (cont’d)

• C-style comments (cont’d)
  – For example:

    /* This is a comment. 
       AVS/Express ignores it. */

  – or

    /* Below is a statement, with a comment on the same line. */
    int var1;  /* This is a comment. */
V comments (cont’d)

• C++-style comments
  – A C++-style comment begins with the delimiter //.
  – AVS/Express treats everything to the right of the delimiter to the end of the line as part of the comment.
  – For example:

    ```
    // This is a comment.
    // AVS/Express ignores it.
    ```

  – or

    ```
    // Below is a statement, with a comment on the same line.
    int var1; // This is a comment.
    ```
V comments (cont’d)

• Special comments
  – Special comment in AVS/Express keeps track of the comment
  – The comment is defined as a special AVS/Express object that is kept as a child of the current object
  – This comment is displayed in the Network Editor and is saved and restored along with the current object
  – If the comment is moved or resized in the Network Editor, its size and position are maintained.

• Special comments are defined using this syntax:

  /*= Here is the body of the comment */

• Attach a size and position to the comment

  /*=100x50+30+30 Here is the text of the comment */
V comments (cont’d)

- Creates a comment that is 100x50 units in size and is offset 30 units in X and 30 units in Y.
- Units are in the same coordinate system as the NEx, NEy, NEwidth, and NEheight properties (usually pixels on most systems).

• Special comments are created with a unique name.
  - Use the $list command to see the comments

```oml
OM(grp1) -> $list
comment _comment
comment _comment#1
```

- Delete the object with the _comment name

```oml
OM(grp1) -> -_comment;
```
V statements

• V statements add, modify, and delete objects. This section first provides an overview of the general coding rules that apply to V statements and then describes each available statement.

• General coding rules
  – Clauses
    • A V statement consists of one or more clauses, such as object type, name, and value expression, terminated by a semicolon.

```plaintext
int var1<NEportLevels=2>[] => {a,b};
/* The clauses in this statement are
• int Object's type
• var1 Object's name
• <NEportLevels=2> Objects' property specification
• [] Array dimension declaration
• => {a,b} Connection expression */
```
V statements (cont’d)

– Order of clauses
  • Clauses must appear in the order indicated by the syntax.
  • If a statement specifies both properties and an array declaration, the properties clause appears first:

    ```
    int var1 <NEportLevels=2> [] => {a,b};
    ```

– Separators
  • Clauses must be separated from one another by spaces, tabs, or newlines
  • May insert any number of these between clauses
  • Certain clauses use special characters, which can also act as separators.
    • Reference-mode clause uses the special characters ^, &, and *
    • The array-declarator clause uses [ and ]
    • The properties clause uses < and >
    • The subobjects clause uses { and }
  • Do not have to code a space, tab, or newline on either side of them
V statements (cont’d)

- Separators are not required before the semicolon that terminates a statement

```plaintext
group &var1[3]; // Valid.
group&var1[3]; // Valid.
groupvar1[3]; // Invalid
```

- In the last example, the object's type and name must be separated from one another with a space, tab, newline, or special character
  - group var1[3];
V statements (cont’d)

– Continuation lines
  • A statement can appear on one line or be split across multiple lines. No special formatting is required for continuation lines

  // The following statements are equivalent:

  int var1[] => {a, b};
  int var1[] => {a,b} ;

– Continuing character strings
  • You can continue a character string literal to another line, but the newline character is considered part of the string

– Case sensitivity
  • AVS/Express is case sensitive

  Int var1[3]; // Invalid
VCP Exercises

Exercise 1.

• Directly create, modify and examine primitive objects

In the Network Editor, NE, open a DefaultApplication. In the VCP move through the hierarchy to the application.

    DefaultApplication {

Create some primitive objects.

    int a;
    int b=4;
    a c;

Notice that they appear in the NE. Type in commands to examine the objects.

    $print
    $int obj
    $float obj
    $list

Set some values.

    a=1;
    c=a+b;
Exercise 1 (cont’d).

Examine the objects again. Change some values.

\[ a = 2; \]

Notice that c doesn’t change. Make it reference a and b.

\[ c => a + b; \]
\[ a = 3; \]
\[ b = 5; \]

Remember to examine the object after each change. Delete some objects.

\[-a;\]
VCP Exercises

Exercise 2.

- Directly create, modify and examine group objects

Create some groups:

```c
    group grp1;
    group grp2 {
        float x=7.2;
    };
    group grp2 grp3;
```

Examine and modify the groups created:

```c
    group1 {
        int y=4;
    };
```

Groups cannot be connected directly like primitives:

```c
    group grp4=>grp1;
```

This will return an error. Group references must be used instead.
Exercise 2 (cont’d).

A group reference specifies the minimum number and type of object that a group must have for a connection to be made. The group reference will attain any additional members that the groups have:

```c
group &grpref1 {
  float a;
};

group grpa {
  float a=4;
  int b=2;
};

grpref1=>grpa;
```

Objects can access other objects at different hierarchical levels using ‘<-.’ to move up a level and ‘.object’ to move down:

```c
int a=2;
int b;
```
Exercise 2 (cont’d).

    group grp4 {
        int a1;
        int b2=7;
    };

    grp4.a1=<-a;
    b==grp4.b2;

Exercise 3:
• Learn how to use V code to directly create, modify, and examine array objects

    Creating primitive arrays:
        float a[2];
        float b[]={1,3,2};
        a c;
VCP Exercises

Exercise 3 (cont’d):

Examine the objects as you make changes to see what’s happening. To examine the arrays, use.

\[ \text\{$get\_array\ obj$} \]

Setting the values.

\[
\text{float } x=3.5; \\
\text{float } y=7.8; \\
a=\{x,y\}; \\
x=6.3;
\]

Make an array reference its value.

\[
a=>\{x,y\}; \\
y=8.3;
\]
Exercise 4.

- Use the loop module to create a macro, `AnimFileName`, that generates a filename from supplied `prefix` and `postfix` strings as well as a `number` from the loop module.

Open a ScratchPad and use the VCP to create a group called `AnimFileNameGen`. Add two string parameters for the `prefix` and `postfix`, then add an `int` for the file `number`.

To create the filename add another string to the group, set the strings value using the built-in `str_format` function.

```plaintext
=>str_format("%s%.3%s", prefix, number, postfix)
```

Because a reference is used filename will change each time a parameter changes. The ‘.3’ tells the function to pad the number with zeros to 3 digits.

Set the ports to be exported properly. Try to do this for a parameter with NE and then examine the object in the VCP. Use the VCP to set the export property on the remaining parameters.

Drag down a loop module, found in `Main.Data_IO`, and connect its output to
VCP Exercises

Exercise 4 (cont’d).

Drag down a loop module, found in Main.Data_IO, and connect its output to AnimFilenameGen. Cut and paste these modules into a macro called AnimFilename. Store finished macro in a Workspace and save project.

In an application test the macro is working by setting values for the parameters.

Exercise 5.

• Calculating Values from Arrays

Create the V code module ArrayFns in the VCP to demonstrate how to construct new arrays from parts of other arrays and how to calculate values from arrays using V’s functions. Taking two float arrays as inputs construct a new array from the first four values of each of the two arrays. Calculate the sum and magnitude of this array. Generate and array of the maximum values of the input arrays.

Open an application in the NE and then move to this in the VCP.
VCP Exercises

Exercise 5 (cont’d).

Open an application in the NE and then move to this in the VCP.

Create two arrays of floats as data for the module, giving each at least four values.

```c
float a[] = {10.2, 11.5, 12.6, 8.4, 21.9, 11.3};
float b[] = {5.6, 9.3, 8.9, 2.3, 12.5};
float c[];
float sumC;
float d[];
string message;
```

Create the group ArrayFns in the VCP; look at the NE to examine the group. Connect the data arrays to ArrayFns’ inputs.

Create the array c from the first four values for each input.

```c
c=>{a[0:3], b[0:3]};
```

Calculate its sum by by setting to appropriate the sumC member of ArrayFns.

```c
sumC=>sum(c);
```
Exercise 5 (cont’d).

Set array $d$ to hold the maximum values of the inputs.

\[ d => \{ \text{max\_array}(a), \text{max\_array}(b) \} ; \]

The \textit{str\_format} function, has a syntax similar to C’s \textit{printf} function. The \textit{message} object in ArrayFns may be set.

\[ \text{message} => \text{“The sum of the \%d values in } c \text{ is } \%f”, \text{ array\_size}(c), \text{ sumC} ; \]
Developing Modules

- Introduction
- Module Overview
- Target Functions
Modules Introduction

- A module is an AVS/Express object
- Contains parameters and methods
  - Parameters define aspects of a module’s operation
  - Methods access and modify parameter values
- Adding new modules
  - AVS/Express provides two tools for adding new modules
    - V command language
    - Add Module Tool in the NE
  - Modules run by executing functions written in C, C++ or Fortran.
  - Execution of these commands supported by a third tool, the Object Manager APIs.
- Add Module Tool
- Series of dialog boxes
- Structured interface
- Easy to set the necessary properties for the module, methods and parameters.
Modules Introduction

• Can use Add Module Tool in two ways
  – Create a module that encapsulates code
    • Code already written
    • Integrate into AVS/Express
    • Connect the module into an AVS/Express network
  – Build a module in Add Module Tool
    • Uses the structure defined in AMT
    • Generates skeleton code
    • Edit the code

• Add Module Tool leads you through three steps
  – Name your module and define one or more methods
  – Define parameters for the module and interaction between parameters
  – Specify code management properties that control how code is integrated
Modules Overview

• A module contains parameters and methods.
  /* In this module, Image1, Image2, and outImage are parameters
  * and update is a method
  */
  Module AddImage {
    Image+read+notify &Image1;
    Image+read+notify &Image2;
    Image+write outImage;
    omethod+notify_inst update = “AddImage_update”;  
  }

• Parameters define aspects of a modules operation
  – Primitive data object (int, float, byte, …)
  – Group data object (base type is group)
  – Another module

• Methods access and modify parameter values
  – Associated with specific target function (C, C++ or Fortran)
  – Executed when method is triggered
Modules Overview

• Typical trigger events
  – Instancing (creating)
  – deinstancing (destroying)
  – Changing a parameter value

• Different events may trigger the same method
  – flags in methods indicate which event triggered the method
  – may check which parameters changed

• Three types of methods correspond to three languages

<table>
<thead>
<tr>
<th>Method type</th>
<th>Target function language</th>
<th>Method value</th>
</tr>
</thead>
<tbody>
<tr>
<td>omethod</td>
<td>C</td>
<td>String value specifies name of the function</td>
</tr>
<tr>
<td>cxxmethod</td>
<td>C++</td>
<td>Optional string value interpreted as source code</td>
</tr>
<tr>
<td>fmethod</td>
<td>Fortran</td>
<td>String value specifies name of the function</td>
</tr>
</tbody>
</table>
Modules Overview

• Care should be taken when naming methods
  – Additional code is generated binding the module to the code
  – Symbols used derived from the names of AVS/Express objects
    • String values of AVS/Express method objects (omethod and cxxmethod)
    • C++ classes

• Should be cautious when naming target functions

• General avoidance rules
  – Prefixes used for AVS components:
    • OM, UE, FLD, DV, GD, GMOD, AG, PAL, IP, and AC
    • Simple names that may conflict with standard C functions, read, write, select, …

• If there are name conflicts error messages appear during compilation
Modules Overview

- Setting method and parameter attributes
  - Method attributes
    - notify_inst attribute
      - Instance method
      - Method’s function executed whenever the module is instanced
    - notify_deinst attribute
      - Deinstance method
      - Method’s function executed whenever the module is deinstanted
  - Parameter attributes
    - Read/noread and write/nowrite attributes
      - Module reads and/or writes the parameter
      - Read and nowrite turn on the behaviour
      - Noread and nowrite turn off the behaviour
Modules Overview

- Setting method and parameter attributes
  - Parameter attributes (cont’d)
    - Req(uire) attribute
      - Associated method is triggered only when the parameter
      - Unlike other attributes, req is automatically off
      - Prevents the triggering of the method if parameter does not have valid value
    - Notify/nonotify attribute
      - Determines whether a change in the parameter’s value triggers the method
      - Changing the parameter’s value queues an event to trigger
      - Event is delivered when current operation is completed
Modules Overview

• Target Functions
  – Must provide a C, C++, Fortran target function that the method executes when triggered
  – Mix of native code and calls to Object Manager API routines
  – API routines
    • Enable perform tasks of getting and setting objects
    • Navigating object hierarchy
    • Creating and deleting objects
    • Controlling execution
  – Writing target functions
    • Define the module
    • Set for the type of target function (C, C++, Fortran)
    • Compile the process
      – Select the module in the NE
Modules Overview

• Target Functions
  – Writing target functions
    • Compile the process (cont’d)
      – Select the module in the NE
      – Select Project->Compile from NE menu bar
        » Express generates a template source file
        » Places it in \textit{my\_proj\_dir/my\_build\_dir/mod\_name.lang}
    • Edit template source file
  – Editing source code
    • Set the \texttt{EDITOR} environment variable
    • In NE select the object whose source file requires editing
    • Select the Project->Edit Source pull-down command
Using C Modules Exercises

Exercise 1.
• Using the **Add Module** tool
• Takes two *floats* as inputs and writes their sum to *float* output
• Calls a C function method called *addflts*
• Source code for method resides in addfloats.c

Select *Workspace_1* in *Library_Workspaces*. Open the **Add Module** tool from the Object menu.

In the Add Module tool, rename the module and add the appropriate type of method. Give the method the correct name. Press **Done** and **Next** to finish adding methods.

Add the two input parameters and give them appropriate names, f1 and f2. Make their input ports visible. Set the method behavior to **read**, **req** and **notify** for each of their parameters.
Using C Modules Exercises

• Exercise 1 (cont’d)

Repeat the process for the output parameter, result. But the method dependency should be simply write access. Press Done and Next to finish adding parameters.

Set the compilation details, choosing the correct process and source code file.

Select Edit Source to edit the existing file, addfloats.c.

Save the source and choose Compile in the Add Module tool. After compilation close Add Module and save the project, under the Project menu.
C API Exercises
Additional Resources

- Ken Flurchick’s visualization training resources, www.osc.edu/~pete/kenf/Visualization/vizTrainingIndex.html
  - Includes information on advanced topics such as module writing
- The International AVS Center, http://www.iavs.org/
- Usenet group comp.graphics.apps.avs

Please contact me, Peter Carswell, if you have any comments, suggestions or questions.
Notes
What is VolSuite?

• An application framework, not an application
• Open architecture relies upon dynamic modules (aka plugins) to implement all functionality
• Facilitates construction of tools for analysis, manipulation, and visualization of scientific data
VolSuite:
A portable scientific application framework

An introduction

Jason Bryan, Ohio Supercomputer Center (jbryan@osc.edu)
What is VolSuite?

• Integrated network capabilities for remote collaboration and exchanges
• All code is portable C/C++ with emphasis on abstraction/polymorphism
• GUI based on FLTK, a cross-platform, open-source, and easy-to-use toolkit
• Ported to Unix, Linux, Windows (Mac forthcoming)
What is VolSuite?

• Good for users
  – If you have data to process/visualize and don’t really need to
develop custom modules, sign up to the mailing list to stay abreast
of updates and make feature requests

• Good for developers
  – Architecture strives to make it easy to develop/integrate new
modules. Well-documented, portable, and open-source means
unlimited extensibility
Where can I get it?

• Version 2.3.0 and prior (auto-redirect to new site once 3.0 is out)
  – www.osc.edu/~jbryan/VolSuite/

• Version 3.0 and later (not yet released)
  – www.osc.edu/VolSuite/

• Mailing list
  – http://email.osc.edu/mailman/listinfo/volsuite
Demo

1. Start VolSuite
2. Import example dataset
3. Open rendering window
4. Play with volume
5. Clipping planes
6. Transfer functions
7. Render Parameters
8. Importing data
9. Exporting data
Guts of VolSuite

- Objects
- Documents
- Hierarchy of pure-virtual classes from which all modules are made
VolSuite Objects

- Provides physical storage of and access to scientific data
- Enhanced with a hierarchical data interface for storing attributes, user-info, and other “meta-data”
- In current version (2.3.0), objects are regular 3D grids (i.e. volumes)
- Next version (3.0) will support truly arbitrary data types (e.g. molecular dynamics, geophysical data, irregular volumetric grids, etc.)
VolSuite Documents

- Linear container for Objects (i.e. a 1D array)
- Enhanced with a hierarchical data interface for storing attributes, user-info, and other “meta-data”
- Can be saved/loaded along with GUI state information. Objects are either embedded or saved by reference
VolSuite Class Hierarchy (2.3.0)

- **Attribs**
  - GlobalConfig
  - VolDoc
  - Volume

- **ExportVolumeModule**
  - SaveVolumeModule

- **ImportVolumeModule**
  - LoadVolumeModule

- **View**
  - DocFilterModule
    - DocViewerModule
      - GLDocViewerModule
  - GlobalConfig
  - ViewFilterModule
    - GLViewFilterModule
Attribs Class

- Holds attributes or state information describing an object (e.g. built-in types such as strings, ints, floats; also user-defined types)
- Organized into a “data tree” (like a file-system): binary data is inserted at /path/to/data
- Typically not derived from directly
- Example code:
  ```
  makeDir( "/properties" );
  insert( "/properties", "size", size );
  insert( "/properties", "color", color );
  get( "/annotations/datasource", &dataSource );
  ```
GlobalConfig Class

- Descendent of Attribs class; not available for derivation
- Holds global settings and parameters for VolSuite
- Convenient means for storing temporary information that is visible to all modules
- In version 3.0, a portion of GlobalConfig will be automatically saved/loaded to/from disk when VolSuite quits/starts; good for persistent storage of settings
ImportVolumeModule Class

- Used to import an object into VolSuite when user input is required to parameterize the process
- Import data from disk, network, etc; Exactly where is up to the programmer
- In version 3.0, an ImportObjectModule can be used to import exactly one type of data
- Example: ImportRaw, ImportImageStack
LoadVolumeModule Class

• Used to load an object into VolSuite when user input is not required to parameterize the process
• Load data from disk, network, etc; Exactly where is up to the programmer
• Data loadable by a LoadVolumeModule can be embedded into a document
• In version 3.0, a LoadObjectModule can be used to load exactly one type of data
• Example: LoadVOL
ExportVolumeModule Class

• Used to export an object from VolSuite when user input is required to parameterize the process
• Export data to disk, network, etc; Exactly where is up to the programmer
• In version 3.0, an ExportObjectModule can be used to export exactly one type of data
• Example: ExportRAW
SaveVolumeModule Class

• Used to save an object from VolSuite when user input is not required to parameterize the process
• Save data to disk, network, etc; Exactly where is up to the programmer
• Data saveable by a LoadVolumeModule can be embedded into a document
• In version 3.0, a SaveObjectModule can be used to save exactly one type of data
• Example: SaveVOL
View Class

- A base class from which most other classes are derived
- Handles some of the functions common to GUI windows (show, hide, set title) as well as their callbacks (for consistency throughout VolSuite)
- Descendents create, then register, an optional FLTK window that serves as the main configuration window for that object
- Typically not derived from directly
DocFilterModule Class

• The primary base class for manipulating or analyzing Documents in VolSuite

• Descendents usually create, delete, change, or analyze Objects within the document, and may also modify the “meta-data” of the Document or its Objects

• Designed to have a single instantiation

• Example: Channel Chooser, Grayscale Filter
DocViewerModule Class

- Descendent of DocFilterModule, extending its interface to support visualizing Documents
- Designed to have multiple instantiations
- Provides an interface for adding instances of ViewFilterModule
- Also handles User Tracing in version 3.0
ViewFilterModule Class

• A specialized module designed to augment the behavior of a DocViewerModule (i.e. change how the viewer visualizes the document)

• Designed to have its state “captured” and “restored” for synchronization, tracing, or session management

• Currently supports single instantiation; will support multiple instantiation in version 3.0
GLDocViewerModule Class

- Descendent of DocViewerModule that adds support for rendering with OpenGL
- Provides an OpenGL rendering “pipeline” in which parts of each registered GLViewFilterModule execute in “stages”
- Can also “pass-through” user interactions with mouse or keyboard to individual filters for processing
GLViewFilterModule Class

• Descendent of ViewFilterModule designed to be used with GLDocViewerModule
• Provides virtual implementation of each stage in the GLDocViewerModule pipeline
• In a descendent, each stage can then be implemented (i.e. overridden) at the developer’s discretion
Developing a VolSuite Module

- Determine the type of feature to be added
- Choose an appropriate base Module to derive from
- Create your module as a subclass of the base module
- Invoke the macros (usually 2 of them) defined in the .h file of the base module at the correct locations
- Iterate:
  - Add code to implement feature
  - Add GUI to interface with feature (using FLTK)
  - Compile code into a shared object (.so) or dynamic link library (.dll)
  - Test/Debug
Developing a VolSuite Module

• Consult the online class documentation for VolSuite and OSCVR:
  – www.osc.edu/~jbryan/VolSuite/
  – www.osc.edu/~jbryan/OSCVR/

• OSCVR is the foundation library VolSuite uses that supplies all underlying data types, networking, rendering classes, etc.
Introduction to FLTK

• FLTK is a powerful, yet simple, object-oriented GUI toolkit
• It is LGPL open-source and compiles successfully on many flavors of Unix, Linux, OSX, and Windows
• Relatively simple and consistent syntax
• Good online documentation at www.fltk.org
Introduction to FLTK

• GUI example:

```c++
#include <FL/Fl.H>
#include <FL/Fl_Window.H>
#include <FL/Fl_Button.H>
#include <FL/fl_ask.H>

static void buttonCB( Fl_Widget *w, void *arg )
{
    fl_alert( "Pushed!" );
}

int main( int argc, char **argv )
{
    Fl_Window *win = new Fl_Window( 100, 100, 350, 350, "Example Window" );
    Fl_Button *b = new Fl_Button( 10, 10, 80, 30, "Push Me!" );
    b->callback( buttonCB );
    win->end();
    win->show();
    return Fl::run();
}
```
Using FLTK in VolSuite

- GUI is created in the constructor. Usually begins with the creation of an FLTK window.
- Add other GUI components to the window
- Register the window with the parent class by calling `setWindow(window);`
Using FLTK in VolSuite

• Callbacks in a C++ class must be static members
  • In class declaration (.h file):
    
    ```cpp
    inline static void _widgetCB( Fl_Widget *w, void *arg )
    {
      ((MyClass*)arg)->widgetCB( w );
    }
    void widgetCB( Fl_Widget *w );
    ```

  • In class constructor definition (.cpp file):
    
    ```cpp
    Fl_Button *b = new Fl_Button(...);
    b->callback( _widgetCB, this );
    ```

  • Put actual callback code in widgetCB (not _widgetCB). This is easier because you don’t have to constantly dereference the void* argument
Example: Grayscale Filter
Example: GLPointRenderer
Watch out!

• Be sure to call all necessary macros in the proper places when creating your module. For help, consult the .h file of your base module’s class
• Be sure to implement all pure-virtual functions. The compiler will complain if there is a function your class is required to implement but is not present
• Developing modules can be confusing. If you have any suggestions for making things clearer or making the interfaces easier, do not hesitate to contact me!!
Advanced Features

- The network interface (Remote Connections)
- Linking filters / Synchronization
- Session management
- Configuration files
  - .volsuite-conf
  - .volsuite-menu
  - .volsuite-hosts
  - .volsuite-log
Remote Connections

- Here you can establish a connection between your instance of VolSuite and that of a remote instance
- Provides global chatting and basic network statistics
- Necessary for synchronization
Synchronization

- The Link Viewers window allows you to synchronize, or link, view filters from different viewers.
- The viewers can be on your machine or a remote machine.
- Linking two filters causes their states to become synchronized.
- Useful for shared, collaborative sessions.
- Usually, the versions of VolSuite must be the same.
Session Management

- Session management is the ability to save the state of a document “as is” including all windows, viewers, filters and their settings
- When saving a document, be sure the “Save GUI state” option is selected
- Note: this is implemented on a module-by-module basis, and some modules may not have session management implemented (most do however). DocFilterModules do not support the session management interface, but DocViewerModules and ViewFilterModules do
Configuration Files

- `.volsuite-conf`
  - Contains configuration information for volsuite (e.g. ports, limits, default values) in “option = value” pairs
  - Contents are loaded into the GlobalConfig at run-time
  - On Unix/Linux machines, there are two conf files: a user file and a system file. At run-time, both are read and their contents merged. On windows there is just one
  - System file is for system-specific settings, such as device ports or memory settings
  - User file is for user-specific settings, such as default directories
  - User file takes precedence, overwriting values from the system file
Configuration Files

• .volsuite-conf example

# sample .volsuite-conf file
spaceball_port = "/dev/tty1"
total_texture_memory = 64MB
default_compression = 7
pi = 3.1415926
use_spaceball = true
– Comments are on their own line and start with ‘#’
– The “option” is a single token on the left of an ‘=’
– The value is on the right of an ‘=’
– Values can be “strings”, integers or floats (with an optional KB, MB, or GB at the end), or boolean true and false (no quotes)
Configuration Files

• .volsuite-menu
  – Contains menu override information for filters in “filter = menu_location” pairs
  – Filters are put into default menu locations, based on their type. These locations can be overridden using the menu file
  – The filter name is to the left of the ‘=’, the new menu location to the right
  – Example:
    
    LoadMyVolume.so = File/Import/Load My Volume
Configuration Files

• .volsuite-hosts
  – Contains host:port pairs, one per line, of machines to automatically add to the “Remote Hosts” list in the Remote Connections window
  – Example:
    www.google.com:12345
    10.0.0.100:54321
Configuration Files

- `.volsuite-log`
  - Contains errors, warnings, and debug information for the last VolSuite session
  - File is overwritten the next time VolSuite is started
  - File is an identical copy of the contents of File->Log…
What’s new in version 3.0

• Truly arbitrary data types
• User tracing
• Architectural changes to ViewFilterModule (and descendents) and DocViewerModule (and descendents) to support arbitrary data types
• Filter “groups” in DocViewerModule to more easily support the notion of a rendering “pipeline”
• New GLDataViewer class (replaces GLVolumeRenderer) for more generic data visualization
• Drag and drop (DND) interface for more intuitive means of creating a rendering pipeline out of view/render filters