AVANGO

Programming introduction

AVANGO

- **av-sh and aview:** the basic shells
- Scheme scripting
- Scheme bindings
- **Tasks**
  - Shell execution 1
  - Shell execution 2
  - Manipulating scene graph geometry
  - Simple data routing
  - Creating new nodes: subclassing AVANGO nodes

The AVANGO shells

- Two shells available:
  - av-sh, run `$AV_HOME/bin/av-sh.sh`
  - aview, run `$AV_HOME/bin/aview.sh`
- Both shells run the same binary
- Shells run a scheme interpreter with AVANGO objects available
- Behavior and appearance is changed via command line options

The AVANGO shells

- Supported architectures:

irix: N32_mips3, N32_mips4, 64_mips3, 64_mips4

linux: IA32_I586, IA32_I686

The AVANGO shells

- Basic command line options

```
-A <arch>    : use binaries and libraries for <arch> or
-D <arch>    : use debuggable binaries and libraries for
               <arch>
-d <string>  : use <string> as command to execute av-
               sh.sh with
-t           : turn on tracing
-v           : turn on verbose operation
-x           : disable screensaver during program
               execution
-z           : enable OpenGL|Performer debug library
               search paths
-h|-?        : help screen
```



The AVANGO shells

- Basic command line options

```
av-sh       [-a <arena-size (mb)>]
            [-H <elk-heap-size (mb)>]
            [-m <multi-processing-mode>]
            [-n <notify-level>]
            [[-p <pipe-display>] ...]
            [-P <preload-type>]
            [-L <preload-file>]
            [-f <scheme-file>]
            [-o <option>:<value>] [-h]
```



The AVANGO shells

- Basic command line options
  - All parameter passed to aview.sh will be passed to av.sh
  - aview is just an av-sh with some preloaded configuration options and scheme scripts that set up the basic viewing window

The AVANGO shells

- aview
  - **Aview can be used to view Avocado scenes on any output device thinkable. It is highly configurable and supports the Cave, theWorkbench and screen output for now**
  - **It sets up the a suitable viewer for the selected output device andenables all known interaction devices**

AVANGO task 1: shell execution

- Run both shells by executing
  - av-sh.sh
  - aview.sh

"Examine the nice difference"

- Run the aview with an arbitrary geometry file by just typing
  - `aview.sh <your desired file>`

AVANGO task 2: shell execution

- aview has by default a somehow strange moving model
- The –o options: All –o <option:value> pairs are accessible from your scheme code
- Change your aview configuration by running aview with the option

`–o device:mon`

The AVANGO shells

- There is one basic shell binary
- The shell implements a basic scheme interpreter
- All configuration is done by
  - command line parameters
  - scheme scripts
- Aview can load arbitrary geometry

AVANGO scheme scripting

- Remember: The shell implements a basic scheme interpreter
- A scheme script can be loaded via the `-f <scriptfile>` option
- There is an approach to implement different scripting languages using a common scripting interface

AVANGO scheme scripting

- Example 1

```
(define myvar 1)

(set! myvar 2)

(define mynode (make-instance-by-name „anode“))

(-> (-> mynode ´Name) ´set-value „myname“)
(fp-set-value mynode ´Name „myname“)

(if (fp-option-value "verbose")
    (begin
      (printf "aview: loading setup ...\n")
      ))
```

AVANGO scheme scripting

- Example 2 aview built-ins

```
(av-help)                 - display this text
(av-reset)                - reset the viewer position
(av-open "filename")      - load geometry from 'filename'
                            and replace old
(av-import "filename")    - load geometry from 'filename'
                            and add to old
(av-load "filename")      - load scheme file with
                            overloaded scene-root
(av-clear)                - delete scene graph
(av-dump <node>)          - dump the fields of a node and
                            their values to stdout
(av-add <node>)           - add graph to root node
```

AVANGO scheme scripting

- Example 2 cont. aview built-ins

```
av-lib-vicinity  - the vicinity in which the aview
                   support files can be found
av-data-vicinity - the vicinity in which the aview data
                   files can be found
av-view-sensor - the main view sensor in use.
av-main-screen - the main screen. this is useful for
                 calibration scripts
av-viewer        - the viewer node. this node is referenced
                 by the av-view-sensor and define the
                 viewer position. by default it is
                 connected via field connection to the
                 Matrix field of a fpFlyer.
```

AVANGO scheme scripting

- Example 2 cont. aview built-ins

```
av-mover         - by default a fpFlyer node connected to
                 av-viewer
av-head          - if we are in a head tracked environment
                 this is a reference to the node which
                 represents the head. in non head
                 tracked environments this is equal to
                 av-viewer.
```

AVANGO scheme scripting

- Example 2 cont. aview built-ins

```
av-right-eye   - in a stereo projection environment these
               are references to the nodes whic represent
               the left and right eyes of the viewer. in
               non stereo projection environments they are
               equal to av-viewer.
av-stylus      - if the output device supports a stylus
               like device this is a reference to a
               fpToolStylus node with a fpPointTool
               connected.
av-stick       - if an input device in form of a joystick
               is supported, this is referenced here.
av-output-device - this variable can be used to make
               scripts output device dependent.
```

AVANGO task 3: the scene graph

- Create a basic scene graph with 4 objects and two layers. Manipulate the approp. matrix fields and alter the position of the objects. Play around with different values and see the results.

```
(define node1 (make-instance-by-name „fpLoadFile“))
(define node2 (make-instance-by-name „fpGroup“))

                          ...
(fp-set-value node1 ´Filename „<my-desired>“)
(av-add node2)
(fp-add-1value node2 ´Children node1)
(fp-set-value node1 ´Matrix (make-trans-mat x y z))

                          ...
```

AVANGO bindings

- There are scheme bindings for all necessary object types.

For example, certain always needed data types like a matrix need a scheme binding that can be manipulated from the scheme side:

`mat?`	is it a matrix?
`make-mat a00 a01…a33`	create a matrix
`make-trans-mat x y z`	create a translation matrix
`mult-mat mat1…matn`	multiply the matrices
	…

AVANGO task 4: simple data routing

- Create a scene with a fpTimeSensor and a fpText node. Connect the time-sensors `time` field to the text-nodes `TimeString` field. Load that scheme file in your preferred aview.

```
(define mytime (make-instance-by-name
                  „fpTimeSensor“))
(define txt (make-instance-by-name „fpText“))
(fp-set-value txt ´Name „MyTextNode“)
(fp-set-value txt ´Color (make-vec4 1 0 0 1))
```

do some more for the txt fields Mat, Justify or DrawStyle as you wish.

```
…
(av-add …)  you already should know what to to here
(-> (-> txt ´TimeString) ´connect-from
                  (-> mytime ´Time))
…
```

AVANGO task 5: Creating your own objects

- AVANGO is extensible.
  - If you don't find a node that suits your needs, just create a new one and make it available to the scripting layer.
  - You have to implement certain features in C++ to enable the AVANGO functionality.

AVANGO task 5: Creating your own objects

- Create a new node type that gets time input and generates a scaling matrix in a certain range. Tip: Use an fpDCS as the basic object to subclass and change its matrix with the newly calculated scaling values.

```
class fpNewClass : public fpDCS {
   FP_FC_DECLARE();
public:
   fpNynewClass();                  //constructor
   virtual ~fpNewClass();           //destructor
   fpSFFloat TimeIn;                // some fields
   fpSFFloat Active;                //   "     "
   virtual void evaluate();
   virtual void evaluateLocalSideEffect();
   virtual void fieldHasChanged(fpField&);
   virtual void fieldHasChangedLocalSideEffect(fpField&);
private:
   static pfType *classType;
};
```

AVANGO task 5: Creating your own objects

```
FP_FC_DEFINE(fpNewClass)
fpType *fpNewClass::classType = 0;
fpNewClass::fpNewClass(): your init values
{
 FP_FC_START_CONSTRUCT(fpDCS);
 FP_FC_ADD_FIELD(TimeIn, 0.0);
 FP_FC_ADD_FIELD(Active, false);
 …
 FP_FC_FINISH_CONSTRUCT();
}
fpNewClass::initClass(){
 if (classType == NULL) {
  fpDCS::initClass();
  classType = new pfType(fpDCS::getClassType(),
„fpNewClass");
  FP_FC_INIT(fpDCS, fpNewClass, TRUE, classType);
}
```

AVANGO task 5: Creating your own objects

```
fpNewClass::evaluateLocalSideEffect(){
 double elapsed = TimeIn.getValue() - _lastTime;
 if ( Active.getValue()){
  // do your stuff here
 }}
fpNewClass::fieldHasChangedLocalSideEffect(field){
 if (&field == &Active)
  // do something

 if (&field == &TimeIn)
  // do something else
 …
}
```

AVANGO task 5: Creating your own objects

- Example c++ header file
- Example c++ source file
- Example Makefile