how to make a VEOS entity

by Geoffrey P. Coco

What you'll need

The VEOS kernel docs: talk_doc, nancy_doc and shell_doc.

Know C and your development environment (Unix, Mac OS, et c.) well.

Abstract

VEOS worlds are composed of one or more 'entities'. Each VEOS entity is a stand-alone executable capable of its own data management, inter-entity communication, and process management. These three capabilities are provided respectively by nancy, talk and the VEOS shell. Together nancy, talk and the VEOS shell form the VEOS kernel which is linked-in with every distinct entity.

Entities differ from each other only by their combinations of VEOS 'primitives'. A VEOS primitive is the atomic unit of entity computation. Primitives are functions which perform very general tasks based on a variable parameter list. Primitives can be invoked remotely through inter-entity communication. It is easy to see that with composition of primitives, just a few primitives yield great computational potential.

Each instance of an entity is a completely distinct process from all other entities regardless of duplicates. Accordingly, each instance of an entity can be configured (or programmed) separately. An entity program is in the form of a configuration file (or .fig file).

The .fig file is written in the VEOS generalized data format - 'grouples'. Upon invocation, an entity loads a specified .fig file for VEOS program steps into the entity's 'grouplespace'.

The grouplespace is the entity's central runtime storage (for data, program steps, et c.). This provides the entity with in-line data and runtime program alterability.

Presently, VEOS entity programs consist of one-time or repeated primitive invokations and their respective parameters. A pseudocode example of an entity program might be:

```
    [ (start) (do me once) ]

    [ (process) (do me forever) ]

    [ (DATA) (don't do me at all) ]
```

World Programmers (WP) presently have two avenues to design VEOS worlds. First, by using existing primitives to compile entities. Second, by writing new entity primitives in C to compile entities. Of course, both methods require that the WP customize each entity's .fig file for the particular function the entity will be performing.

Protocol

Presently, each entity shell has the following general primitives.

```
    [ hello (anything) (goes here) (...) ]
      # prints hello, and whatever the parameters are.

    [ clockPrim (time in microsecs) (prim to execute) (prim's params) (...) ]
      # executes the specified primitive every interval of microseconds.

    [ respond (sender's UID) ]
      # sends a respond grouple to the sender, with the local entity's UID.

    [ speakTalkPrim (destination UID) (message grouple words) (...) ]
      # described in talk_doc.

    [ dumpGSpace ]
      # print the entire contents of the grouplespace.

    [ getNextCommand ]
      # interpret the keyboard input as a grouple, post it to the grouplespace.

    [ sleepPrim (time in seconds) ]
      # causes the entity to sleep for specified number of seconds
            (evil primitive).
```

```
    [ spaceRegister (entity type - text) (operation) (entity UID) ]
      # enter (or exit) an existential space's informational domain.
      # then, rebroadcasts:
      #    [ entityRegister (entity type) (operation) (entity UID) ]
            # to all currently registered entities and sends entityRegister
      # grouples to the new registrant - one for each current registrant.
```

The following primitives are not standard issue, but are very common.

All entities should have customized versions of entityRegister...

```
    [ entityRegister    (entity type -    text) \
                  (operation -      text)
                  (sender UID -     text) ]
```

Rendering protocol remains highly general and simple:

```
    [ newStructure      (object name -    integer n)
                  (object type -    integer t)
                  (scale -          float x, y, z)
                  (init pos -       float x, y, z, roll, pitch, yaw)
                  (color -    float r, g, b) ]

    [ alterStructure    (object name -    integer n)
                  (operation -      integer o)     #listed in world.h
                  (more params -    float xd, yd, zd, rolld, pitchd, yawd) ]

    [ alterView   (operation -      integer o)
                  (more params -    float xd, yd, zd, rolld, pitchd, yawd) ]
```

There are other primitive calls, but their protocols are far less stable and so are not worth mentioning.

What to do from C

Setup a .c file according to the requirements below.

1. #include "world.h"

   It contains all common entity constants, externed common primitives, and common data types. Browsing through world.h and it's included files is encouraged for a better understanding of VEOS internals.

2. Design entity code as VEOS primitives.
   Primitives must be of the form:

```
TVeosErr myPrimitive(pGrouple)
  TPGrouple       pGrouple;
{
  TVeosErr        iSuccess;

  /**************/
  /* processing */
  /**************/

  return(iSuccess);

  } /* myPrimitive */
```

   The special error VEOS_EXIT is reserved for terminating the entity shell. If a primitive wishes to bring the entity down, it should return VEOS_EXIT and the entity will terminate gracefully.

   VEOS Primitives enforce their own time-sharing. That means that a primitive should not do extended computations, but rather, break up computation into incremental tasks which take only small amount of processing time per entry into the primitive.

3. WPs are strongly recommended to write special initialization and takedown primitives which will be invoked only once.

   Furthermore, it is important to be aware of the two VEOS functions that should be called in the init primitive.

   A. getNative(entity type text) -

      a VEOS world utility function (not a primitive, it can only be accessed internally) - loads the entity's local cache (see ENTITY_NAME and SPACE_ADDR in world.h) and attempts to register the entity as the given entity type with the existential space entity specified in the entity's .fig file. Take this function's return value very seriously.

   B. setCloseProc(myCloseProc) -

      a VEOS kernel hook (again, not a primitive) - allows the entity to get process control just before the entity terminates. Call setCloseProc() with the address of the entity's takedown function (counterpart to init primitive).

The takedown function (not necessarily a primitive, but could be) should at least call the entity startup's counterpart.

A. getForeign(entity type text) -

   the counterpart to getNative(). getForeign()'s most important function is to attempt to detatch gracefully from the existential space.

4. If the WP intends to follow the current inter-entity communication protocol, provide an 'entityRegister' primitive. This is each VEOS entity's obligation in order to maintain entity compatibility.

   The entityRegister primitive is usually where entity coupling and decoupling occurs. This mechanism is especially important because improper decoupling can cause some or all connected entities to crash. To assure proper entity decoupling, be sure to call speakClose() when entityRegister has determined that a connected entity is about to terminate (see Protocol, above). Pass the UID of the unregistering entity to speakClose() as in this code fragment.

```
TVeosErr entityRegister(pGrouple)
{
    ...

    if (strcmp(pGrouple->pWordList[1], VEOS_KILL_TEXT) == 0) {

        iSuccess = speakClose(pGrouple->pWordList[2]);

        }
    ...

} /* entityRegister */
```

5. Write a C function loadSpecialPrims() in the entity source file in the following form.

   Each entity's VEOS shell contains a local jump table of primitive function addresses. The function loadSpecialPrims() is the WP's mechanism for including custom primitives into the entity shell's jump table so that they are invoked correctly.

```
TVeosErr loadSpecialPrims()
{
  loadFunctionInfo(VEOS_ENTITY_REGISTER_PRIM, entityRegister);
  loadFunctionInfo("myEntityInit", myEntityInit);
  loadFunctionInfo("myPrimitive", myPrimitive);

  ...

  return(VEOS_SUCCESS);

  } /* loadSpecialPrims */
```

NOTE: Every entity must have a loadSpecialPrims() function. If the entity does not contain any non-standard primitives, loadSpecialPrims() must still exist and simply return VEOS_SUCCESS.

6. From any place within entity C code, the WP can check the debugging flag or the timing flag to perform runtime diagnostics. For example:

```
if (BUGS)
  fprintf(stderr, "shell %s: debugging output: ... \n", ENTITY_NAME);

if (TIME)
  fprintf(stderr, "debugging at time: %d\n", time);
```

These debugging flags can be set on the entity shell command line (see Running an Entity).

Compiling an Entity

Link with: shell.o, world_utils.o, <entity code>.o
and options: -ltalk -lnancy -lprims

Programming an Entity

Make a .fig config file in grouple format. A .fig describes the initial steps of entity execution. For example:

```
    [ DATA-ENTITYID (entity UID) ]
        # Mandatory grouple, stores the entity's address.
        # Does not invoke a primitive.

    [ DATA-SPACE-ADDRESS (entity UID of space entity) ]
        # Mandatory grouple, stores the existential space's address, if any.
        # use "nil" for no space.

    [ initMyEntity ]
        # Initialize the custom entity code.

    [ myPrimitive (data) (words) (go here) ]
        # Invoke myPrimitive once with the specified data.

    # After a grouple has invoked a primitive, the entity shell deletes that
    # grouple.  Unless the primitive name begins with '$'.  A '$' prefix to
    # any primitive grouple makes the primitive an ever-repeating primitive
    # (or a daemon).

    [ $listenTime ]
        # Give processing time to the incoming message daemon.

    [ $speakTime ]
        # Give processing time to the outgoing message daemon.

    [ $clockPrim (500000) (dumpGSpace) ]
        # Use clockPrim for timing and pacing.
```

Running an Entity

type the command line:

```
    <entity shell name> <entity config file>.fig [ "bugs" | "time" ]
```

During runtime, an entity shell can be terminated gracefully by typing ^C in the shell's execution window. The shell traps ^C, calls the entity's exit proc, if any, and terminates normally.