KUNGL TEKNISKA HÖGSKOLAN
Royal Institute of Technology
Numerical Analysis and Computing Science

TRITA-NA-D0002 • CID-60, KTH, Stockholm, Sweden 2000

# A Practician's Guide to DIVE

Gustav Taxén

**Gustav Taxén**

A Practician's Guide to DIVE

**Report number:** TRITA-NA-D0002, CID-60

**ISSN number:** ISSN 1403-073X

**Publication date:** January 2000

**E-mail of author:** gustavt@nada.kth.se

**URL of author:** http://www.nada.kth.se/~gustavt

# Contents

# A Practitian's Guide to DIVE

## Introduction

This report presents solutions to practical problems often encountered when building interactive virtual environments in DIVE [3][4][8]. It is intended for people that have tried navigating with the vishnu browser in single-user mode and want to know how to set up DIVE for multi-user environments or want to create their own DIVE worlds. To create DIVE worlds, a basic knowledge of Tcl/Tk [9] is required. There are plenty of Tcl/Tk tutorials available on the WWW; good place to start is the Scriptics website [10].

DIVE should not be thought of as an application or a program. It is more correct to say that DIVE is a protocol for manipulating and viewing virtual worlds. The protocol is implemented in a set of programming libraries that can be used to create applications. In this text, a DIVE application is defined as an application that "knows" about the DIVE protocol and can talk to other DIVE applications.

There are four main reasons why DIVE is at least as interesting as similar products such as ActiveWorlds [1]:

- DIVE is free for non-commercial use.
- DIVE runs on many platforms, including SGI, Solaris, WindowsNT and Linux.
- If you know Tcl it is relatively easy to assign dynamic behaviour and interactivity to any DIVE object. If you know Tk, you can write GUIs for any DIVE object.
- You can develop new DIVE applications. There are three main ways to do this: Create a new GUI for the `diva` program with Tcl/Tk, link a C program with the DIVE libraries, or communicate with DIVE through TCP/IP.

## Relationship between DIVE binaries

The main binaries that are distributed with DIVE are:

- `diva` - A DIVE application that implements a 3D graphics and sound interface. `diva` is basically an advanced browser for DIVE worlds.
- `diveserver` - A server that contains a list of DIVE applications and the VR worlds they are viewing.
- `proxyserver` - The proxyserver relays messages between networks that support IP multicast and networks that don't. DIVE applications use a *peer-to-peer* communication model, which means that all DIVE applications keeps a copy of the environment and the state of its objects. IP multicast is used to send messages between the applications. If a

network doesn't support multicast, the proxyserver can be used to relay messages to one that does.

These and other DIVE binaries are described in detail in [6]. To set up a multi-user DIVE session, you typically have to do the following:

- Start `diveserver`. The `diveserver` keeps a list of DIVE worlds and IP multicast groups, as illustrated in figure 1. When a DIVE application connects to the server, it tells the server which world it is viewing. If the world is among the worlds in the server list, the server responds with a message containing an IP multicast group. The group contains all DIVE applications that views the world. When started, `diveserver` will respond with something like

  ```
  *** The Dive name server ***
  Olof Hagsand - olof@sics.se
  Emmanuel Frécon - emmanuel@sics.se

  Own address: 130.237.228.72:3177
  ```

  You'll need the server's "own address" when you start the DIVE application. In the example above, 130.237.228.72 is the address and 3177 is the port.
- If your computer is on a network that doesn't support IP multicast, you must use the `proxyserver`. The `proxyserver` acts as a broker between multicast networks and non-multicast networks, as illustrated in figure 2. It connects to the `diveserver` as a DIVE application and forwards any messages it receives to the clients on the non-multicast network. This means that the `proxyserver` must be started on a computer that is connected to a multicast network.
- Start a DIVE application, in this case `diva`. When `diva` is started, it executes a Tcl/Tk script once. The script is responsible for, among other things, creating a GUI. You can specify which script to run in two ways:
  - Change the name of the `diva` executable or create a symbolic link to it under a different name. In Unix, for example, you could say

    ```
    ln -s ./diva ./foo
    ```

    to create a link to `diva` called `foo`. When you start `foo`, it will load the script called `foo_init.tcl`. Generally, if the `diva` executable is named *name*, the Tcl/Tk script *`name`*`_init.tcl` will be loaded. When no script is specified, `diva` loads a script called `blind_init.tcl` that creates a non-graphical browser with audio capabilities. There is a script called `vishnu_init.tcl` that creates a graphical browser with an advanced user interface. Normally a symbolic link to `diva` named `vishnu` is created when DIVE is installed.

- Use the `-init-script` command line parameter, i.e.

  ```
  diva -init-script myscript.tcl
  ```

Since we want to use the DIVE application in a multi-user environment, we must let it know where the `diveserver` is. This is done through the `-nameserver_addr` and `-nameserver_port` command line parameters. To connect to a `diveserver` with "own address" 130.237.228.72:3177, you say

```
vishnu -nameserver_addr 130.237.228.72 -
nameserver_port 3177
```

From now on, it is assumed that `vishnu` is used to view DIVE worlds.

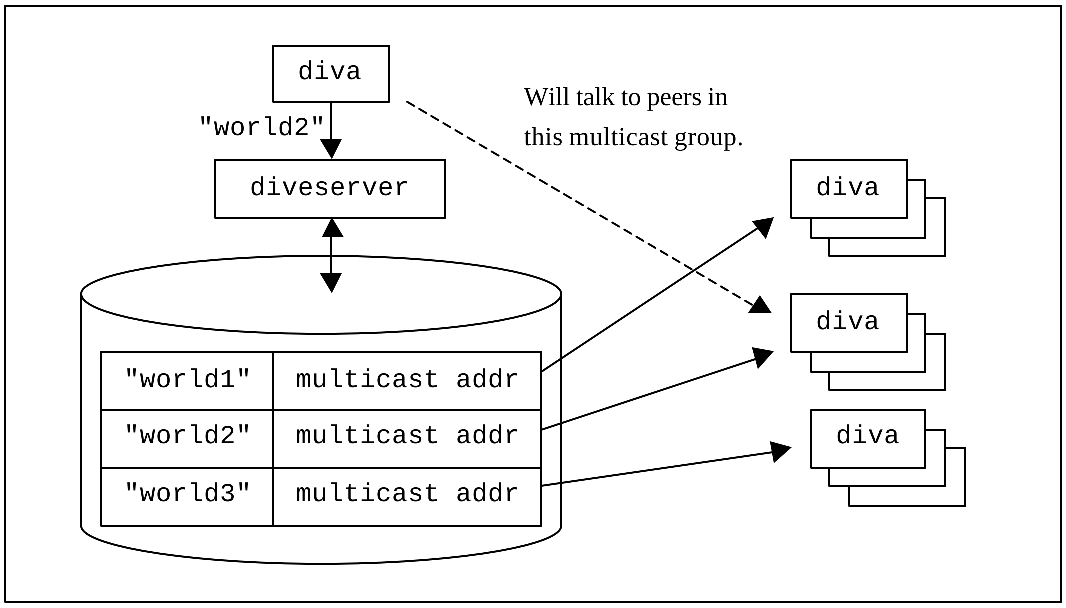


*Figure 1. Organization of the diveserver and DIVE applications in multi-user mode.*

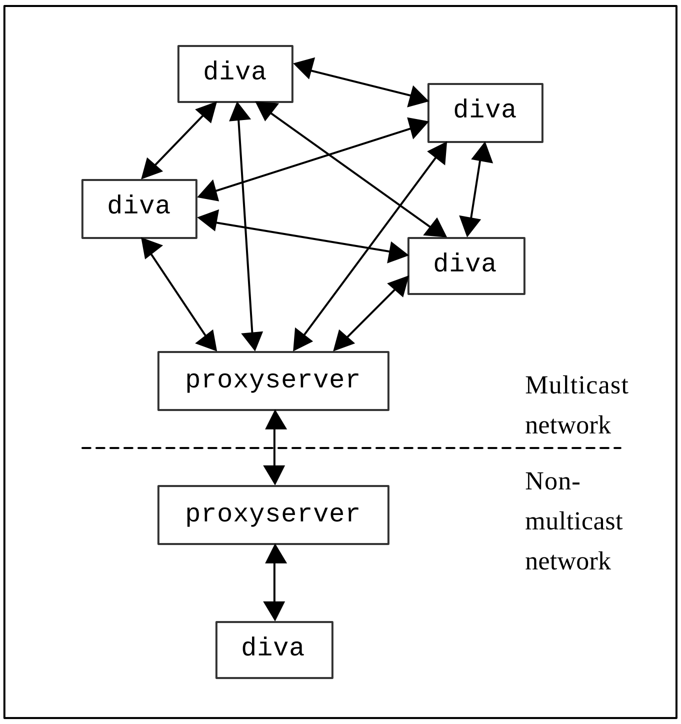


*Figure 2. The DIVE proxyserver acts as a broker for networks without multicast.*

## DIVE worlds

When you start `vishnu`, you need to specify which world to view. Worlds are described in *object definition files* (in DIVE, every world is also a DIVE object). All object definition files have a `.vr` extension. To specify a world to view in `vishnu`, you use the `-world` command line option. If you don't specify a world, `vishnu` will view a world called `tutorial.vr`. To view a world called `foo.vr` you say

```
vishnu -world foo.vr
```

In a multi-user environment, `vishnu` checks with the `diveserver` if the world is viewed in a `vishnu` browser somewhere else. If it is, the world description is fetched from that browser, not from the object definition file.

The object definition file must reside in the current DIVE directory search path. See [7] for more information on how to set the search path. Note that you need to include `./` explicitly in the search path if you want `vishnu` to look for files in the current directory. Also note that `vishnu` searches the directories in the same order as they appear in the DIVE search path. If the object definition file resides on a WWW server, you can specify it using its URL. For example,

```
vishnu -world
http://www.nada.kth.se/~gustavt/dive/foo.vr
```

Whenever something in a world changes, the change is distributed to all DIVE applications that are viewing the world. This means that the world is "alive" as long as there is at least one DIVE application that views it. When all DIVE applications have exited the world, its current state is lost. In other words, DIVE worlds are not persistent. It is possible, however, to save the current world state to disc as an object definition file that could be loaded at a later time.

## Object definition files

In DIVE, everything is an object, including the world itself. The difference between an ordinary object and a world object is that the world object contains a set of special parameters. These includes the color of the background and the point where users enter the world. The DIVE world database is hierarchial: the entire world is described through child objects of a special root node. Every node has one parent (except for the root node) and every node can have an unlimited number of children. The object definition file format is described in detail in [2].

All object definition files are run through a C preprocessor before they are interpreted by `vishnu`. That means that you can include files and use text macros in exactly the same way as in C and that standard C comments can be used. There is a file called `dive.vh` that you should `#include` in all your world and object definition files, since it defines some important text constants. The object definition file syntax will not be repeated here. Instead, this text concentrates on specific issues that the official DIVE documentation omits.

### Scale of DIVE worlds

One unit in DIVE corresponds to approximately one meter. For example, the height of the "Blockie '95" user representation at the podium in the `tutorial.vr` world, is 2 meters.

### Newlines at end of files

All object definition files must end with a newline, otherwise the following error is generated:

```
cpp: No pp_newline at end of file
```

It is unclear whether it really matters if all files have a newline at the end, but it seems as if DIVE applications are more stable if they do.

### Whitespaces after linebreaks

If you use the `\` (backslash) character to indicate a line break in an object definition file or a Tcl file, you must make sure that there are no whitespace characters after it. If there is, the following error is generated:

```
Skipping illegal input '\'.
```

### Vector, texture coordinate and normal specification

Vectors / points, texture coordinates and normals are specified by adding `v`, `t` and `n` in front of the elements, respectively, as in

```
view {
   RBOX
   v -1 -1 -1
   v 1 1 1
}
```

Note, however, that this syntax is not used consistently throughout the DIVE object file format. In the world definition, for example, the start point of the user is defined by

start v *x y z*

while the position of the global light source is defined by

position *x y z*

Note that for stand-alone `N-POLY` polygons, it is allowed to include texture coordinates for every vertex, but incorrect to include any normal information whatsoever. Thus, it is correct to write

```
view {
   N_POLY 3

   v 1 0 0    t 0.0 0.0
   v 0 1 0    t 0.5 1.0
   v 0 0 1    t 1.0 0.0
}
```

but incorrect to write

```
view {
   N_POLY 3

   v 1 0 0    t 0.0 0.0   n 0 1 0
   v 0 1 0    t 0.5 1.0   n 0 0 1
   v 0 0 1    t 1.0 0.0   n 1 0 0
}
```

If you want to specify normals, you must use `N_M_POLY`, i.e.,

```
view {
   N_M_POLY 1 3 (T_PER_VERTEX + N_PER_VERTEX)

   N_POLY 3

   v 1 0 0    t 0.0 0.0    n 0 1 0
   v 0 1 0    t 0.5 1.0    n 0 0 1
   v 0 0 1    t 1.0 0.0    n 1 0 0
}
```

### "object" vs. "subs object"

The correct way to declare a sub-object is to write

```
object {
   object {
   }
}
```

The following syntax is obsolete, even though it is used in some places in the available documentation and example code:

```
object {
   subs object {
   }
}
```

Using `subs object` doesn't generate an error, but the file may be incompatible with later releases of DIVE.

### World declaration

If the world name contains too many characters, the following error message is generated:

```
DIVE warning:
    TCL error: can't read "endofwname": no such variable
    while executing: world_url_msg ACTOR_MIGRATE_EVENT
...
```

The global light source position is a positional light source, not a directional light source. Thus, using the default location (-3 2 -1) will often result in strange-looking lighting of objects.

### Backface culling

The backface culling flag syntax is somewhat ambigous. To *enable* backface culling, you write

```
nobackface off
```

To *disable* backface culling, i.e. to always render polygons no matter which way they are facing, you write

```
nobackface on
```

**Graspable objects**

It is not possible to make a sub-object ungraspable. Thus, the following object will be graspable, even though `nograsp on` is used:

```
object {
   object {
      nograsp on
      view {
         RBOX v 0 0 0 v 1 1 1
      }
   }
   view {
      RBOX v -1 -1 -1 v 0 0 0
   }
}
```

To make the object ungraspable, you can write

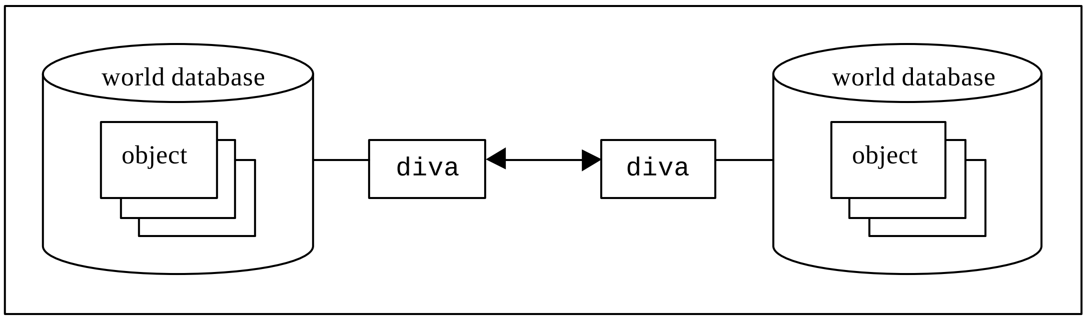
```
object {
   nograsp on
   object {
      view {
         RBOX v 0 0 0 v 1 1 1
      }
   }
   view {
      RBOX v -1 -1 -1 v 0 0 0
   }
}
```

## The DIVE/Tcl interface

In DIVE, you can define the behaviour of an object by including a Tcl script in its object definition file. When the object definition file is read, the Tcl script is interpreted once (although, as we shall see, DIVE events can cause procedures defined in the script to be invoked repeatedly). In multi-user situations, the Tcl files assigned to objects that are sent across the network are interpreted once in each DIVE application. In addition to the standard Tcl command set, a number of DIVE-specific commands have been added. These are described in [5].

It is important to understand how DIVE works in multi-user situations. The distribution model of DIVE is *peer-to-peer,* which means that every DIVE application keeps its own copy of the world database, as illustrated in figure 3. When the state of an object changes in one DIVE application, the change is distributed to all connected applications.

*Figure 3. DIVE uses a peer-to-peer network distribution model.*



## Properties, variable scope and events

The Tcl code of each DIVE object (including sub-objects) runs in its own Tcl interpreter. This means that global Tcl variables cannot be seen outside the scope of the object's Tcl code. Global Tcl varaiables are not distributed among DIVE applications. Therefore, the contents of a global variable can differ between distributed copies of the same object, as illustrated in figure 4.

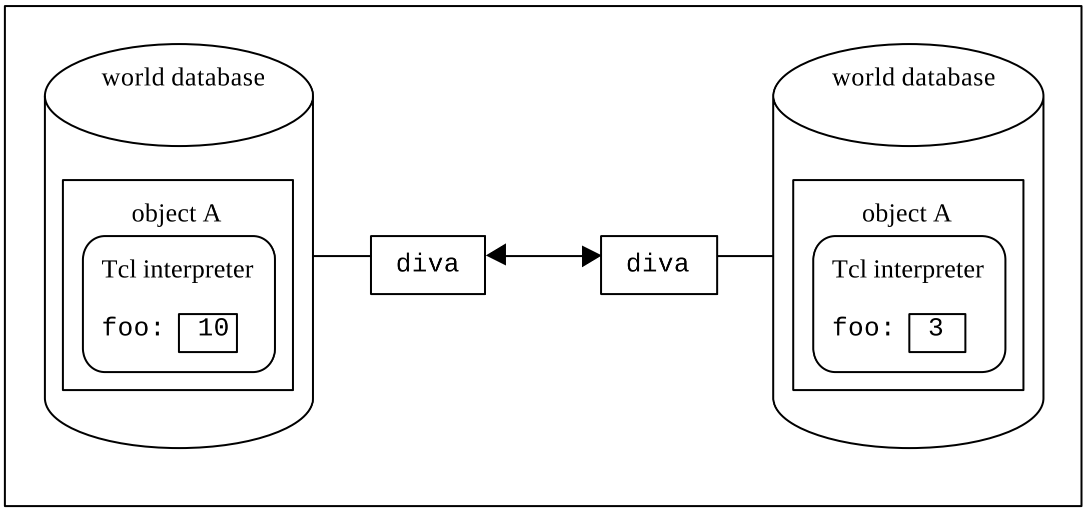


*Figure 4. Global Tcl variables can have different values in different DIVE applications.*

In order to share variable information between objects, *properties* must be used. A property is a distributed DIVE variable that is associated with a DIVE object. When a property has been created in one DIVE application, it is automatically created in all connected applications. When the value of a property changes, the change is distributed to all connected applications. This  is illustrated in figure 5.

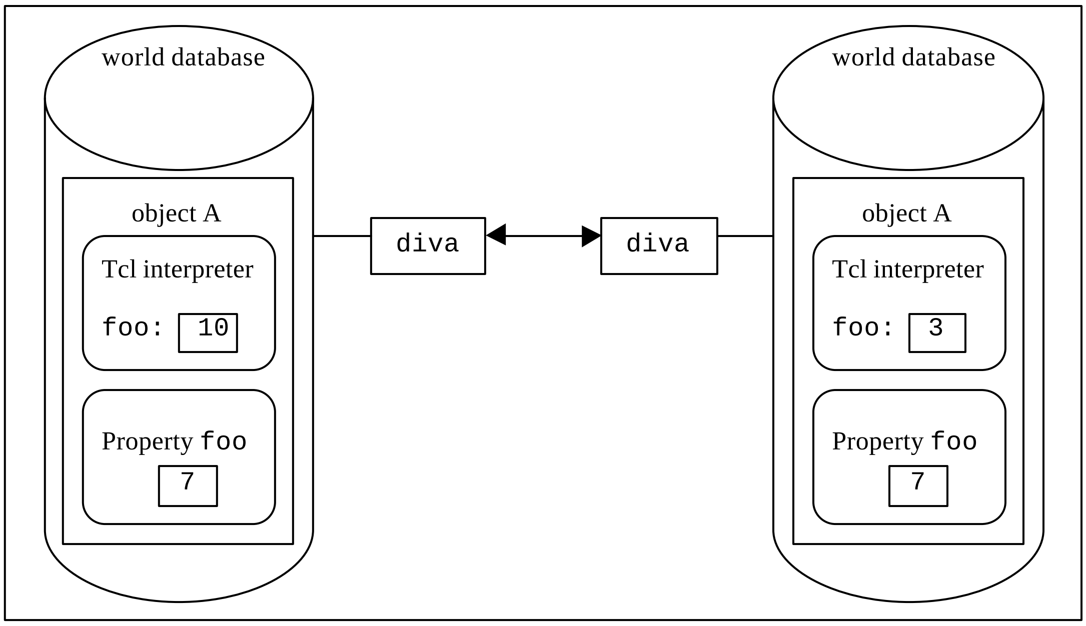


*Figure 5. The value of properties are synchronized across all communicating DIVE applications.*

As an example, the source code for a digital stopwatch is presented in Appendix A. The stopwatch can be started and stopped by clicking on it. When running, it continously polls the system time and notes when the value of the seconds change. When this happens, one second is added to the stopwatch time and the display is updated. The stopwatch uses four properties:

- `active` - 1 if the stopwatch is running, 0 otherwise.
- `hrs` - The number of hours the stopwatch has been running.
- `min` - The number of minutes the stopwatch has been running.
- `sec` - The number of seconds the stopwatch has been running.

Note that the `active` flag must be a property rather than a Tcl/Tk global variable - if not, the result might be that the same stopwatch is running in one DIVE application and is idle in another. The same applies to the other properties as well.

When there is a change of state in the database belonging to a DIVE application, an *event* is generated. Objects can register an *event callback*, a function that is invoked when a specified event is generated. When an object registers a callback, the callback is automatically registered in all connected DIVE applications. However, when an event is generated in an application, only the callbacks of the objects in that application will be invoked. In other words, events are *not* distributed. This means, for example, that clicking on an object in a `vishnu` browser will generate an event in that browser only, not in any other connected applications. This behavior can be overridden for an object by using the `proc_bound` command in its definition file. If an object is `proc_bound`, all Tcl code that is associated with that object runs in the DIVE application in which the object was originally created. This means, for example, that if a `proc_bound` object has registered a callback for interaction events and a user clicks on the object, the callback will be executed in the appropriate Tcl interpreter in the application that created the object, not necessarily in the application from which the object was clicked on.

A *timer* is an event factory that generates events at regular intervals. Timers are special in the sense that when they are created, it is unspecified in which DIVE application they will execute. Also, when a timer is running in an application and that application disconnects from the network, the timer is transferred to one of the remaining applications. The stopwatch needs callbacks for two event types: the interaction event and a timer event. The interaction event is generated when the user clicks with the left mouse button on an object.

Since each DIVE application keeps its own copy of the world database and DIVE/Tcl scripts are executed locally, there will - inevitably - be times when the contents of the local databases differ. That is, you can't assume that a new value of a property is updated instantly in all connected databases. Therefore, you should take care when writing applications that changes the contents of the database often. If you're changing the database faster than the changes can be distributed you'll probably end up with a paradox situation sooner or later. This is a problem that's inherent in all programs that use the peer-to-peer communication model and cannot really be avoided. Therefore, if you need to make frequent changes to the contents of the world database (or if you're on a slow network), DIVE may not be the appropriate platform to use.

A final word on properties: there is a DIVE/Tcl command, `dive_property_link`, that links a global Tcl variable to a property. When the contents of the variable is changed, the property is updated automatically. Since the contents of a global Tcl variable can differ between DIVE applications, you should use this command with care. Also note that you need to create the link in all the connected applications.

### "dive_send" vs. "dive_call"

When you invoke a number of consecutive `dive_send` commands in a behaviour script, there is no guarantee that the function calls will be executed in the same order as they were given. If you want to make sure that the calls arrive in the correct order, use `dive_call`.

### Creating new objects on-the-fly

There are two different ways to create new objects from the behaviour code of an object. The simplest way is to parse a URL by using the `dive_readURL` commands. The drawback of this method is that the specifications of the new object is defined by the contents of the file that is read. The alternative is to use `dive_parse_string`. The following code fragment will create a new sphere with variable radius:

```
proc add_new_sphere {parent x y z} {
   set string "object { view { SPH $x $y $z } }"
   set sphere [dive_parse_string $parent \
      "x-world/x-dive" "$string"]
}
```

Note, though, that the sphere isn't created immediately. Therefore, it is incorrect to write

```
proc add_new_sphere {parent x y z} {
   set string "object { view { SPH $x $y $z } }"
   set sphere [dive_parse_string $parent \
      "x-world/x-dive" "$string"]
   dive_abs_move $sphere 1 0 0 WORLD_C
}
```

The correct way to handle this situation is to let the sphere create an `ENTITY_NEW` event callback for itself. Note though, that DIVE doesn't support the `begin.tcl` and `end.tcl` tags when creating new objects with `dive_parse_string`, so you have to use a separate Tcl file for the code. For example, if the file `foo.tcl` contains

```
on_new {t i o} {
   dive_abs_move [dive_self] 1 0 0 WORLD_C
}
dive_register ENTITY_NEW_EVENT NULL [dive_self] ""
on_new
```

you can say

```
proc add_new_sphere {parent x y z} {
   set tcl_code "inline.tcl \"foo.tcl\""
   set string "object { view { SPH $x $y $z } $tcl_code
}"
   set sphere [dive_parse_string $parent \
      "x-world/x-dive" "$string"]
}
```

**Level-of-detail**

Be careful when you assign level-of-detail information to objects. For example, when the following object is inserted into a DIVE world, only the first sub-object responds to mouse clicks:

```
object {
   lod {
      range { 20 }
      object {
         view {
            RBOX v 0 0 0 v 1 1 1
         }
      }
      object {
         view {
            RBOX v -1 -1 -1 v 0 0 0
         }
      }
   }

   begin.tcl
   proc on_click {event vid type origin src_id x y z} {
      puts "click"
   }
   dive_register INTERACTION_SIGNAL DIVE_IA_SELECT \
         [dive_self] "" on_click
   end.tcl
}
```

Also note that Tcl global variables declared in one LOD sub-object isn't visible in other LOD sub-objects.

**Database navigation**

When you navigate through the DIVE world database using `dive_find` commands, you should be aware that searching from the root node can give unexpected results. The reason is that the user may create several copies of the same object. If, for example, several copies of an object named `foo` exists in the database, it is unspecified which of them that is returned if you start searching for `foo` from the root. So avoid writing

```
}
```

```
object {
   name "my_object"

   object {
     name "sub_object1"
     begin.tcl
     proc my_proc {} {
       set root [dive_find_root]
      set my_object [dive_find_sub_byname $root
"my_object"]
      set sub_object2 [dive_find_sub_byname \
         $my_object "sub_object2"]
     }
     end.tcl
   }

   object {
     name "sub_object2"
   }
}
```

since if there are several objects in the database named `my_object`, the code may fail. A more robust way to do this is

```
object {
  name "my_object"

  object {
    name "sub_object1"
    begin.tcl
    proc my_proc {} {
      set parent [dive_find_super [dive_self]]
     set sub_object2 [dive_find_sub_byname \
        $parent "sub_object2"]
    }
    end.tcl
  }

  object {
    name "sub_object2"
  }
}
```

### Relationship between transformations and object info

When you apply translation or rotation to DIVE objects, the corresponding transformation matrices are multiplied together incrementally and stored internally in the object in two special matrices, a translation matrix and a rotation matrix. There are two versions of each of these matrices:

- The *world coordinate system* version that describes how points given in the object's coordinate system are transformed into the world coordinate system.
- The *local coordinate system* version that describes how points given in the object's coordinate system are transformed into its parent's coordinate system.

The `dive_entity_info` command can be used to retreive these matrices from an object. The translation matrix is given as a (*x y z*) value (i.e. the translation distance along each of the three coordinate axes). The rotation matrix is given as a 3x3 matrix, stored in *column order* (Fortran order). For example, the lists

```
{a b c} {d e f} {g h i}
```

corresponds to the matrix

```
a d g
b e h
c f i
```

There is no built-in way to obtain the inverse to the matrices. Calculating them is not difficult, however. Calculating the inverse of a translation is trivial and since rotation matrices are orthogonal, the inverse of a rotation matrix is simply its transpose. A Tcl function, `untransform`, that calculates the inverse of a DIVE transformation is listed in Appendix A.

Note that there is no scaling matrix. The command `dive_scale` can be used to scale an object, but only relative to its current size. Absolute scaling of objects is not supported, nor is there a way of obtaining the scaling information of a DIVE object.

### Adding objects to the visor

In DIVE version 3.3x, you can add objects to the visor of an actor. It is important to remember, however, that once an object has been added to the visor, it only exists in the DIVE application that added it to its visor. Changes made to visor objects are not distributed. If you want to change a visor object that exists in a remote DIVE application, you have to add a function to the actor that is associated with the remote application and use `dive_call`.

### Keyboard repeat

When a DIVE application starts, the keyboard repeat is deactivated. If the application crashes, the keyboard repeat is sometimes not restored. When this happens, start `vishnu` with the tutorial world and exit immediately to restore the keyboard repeat.

## References

[1] ActiveWorlds. `http://www.activeworlds.com/`

[2] Avatare, Frécon, Hagsand, Jää-Aro, Simsarian, Stenius, Ståhl. *DIVE - The Distributed Interactive Virtual Environment - DIVE Files Description for DIVE version 3.3x*. `http://www.sics.se/dive/manual/dive_file_format.html`

[3] Carlsson & Hagsand (1993) "DIVE - A Multi User Virtual Reality System". *IEEE VRAIS*, September 1993.

[4] Carlsson & Hagsand (1993) "DIVE - A Platform for Multi-User Virtual Environments". *Computers and Graphics*, 17(6), 1993.

[5] Frécon, Hagsand, Hansson, Stenius, Ståhl, Wallberg. *Dive/Tcl Reference Manual*. `http://www.sics.se/dive/manual/tclref.html`

[6] Frécon & Hagsand (1997) *Dive Applications*. `http://www.sics.se/dive/manual/application.html`

[7] Frécon & Hagsand (1997) *Dive Installation*. Technical Memo. `http://www.sics.se/dive/manual/install.html`

[8] Hagsand (1996) "Interactive Multi-User VEs in the DIVE System". *IEEE Multimedia Magazine*, 3(1), 1996.

[9] Osterhaut (1994) *Tcl and the Tk Toolkit*. Addison Wesley.

[10] Tcl/Tk distribution and documentation. `http://www.scriptics.com/`

# Appendix A - Source code

## Stopwatch

```
#include "dive.vh"

#define STOPWATCH_HEIGHT  0.1

object {
    name "stopwatch"

    /* The view sub-objects define the appearance of the stopwatch
     * in the world. The first sub-object is the stopwatch text. It
     * is set to "00:00:00" initially. Note that the view is named.
     * This is because we want to change the text string later. The
     * second sub-object is the stopwatch case.
     */

    object {
        material "white"
        view {
            name "text"
            CTEXT STOPWATCH_HEIGHT
            "00:00:00" "default"
        }
    }

    object {
        material "red"
        view {
            RBOX
            v (STOPWATCH_HEIGHT * -3.5) (-STOPWATCH_HEIGHT) -0.01
            v (STOPWATCH_HEIGHT * 3.5) (1.5 * STOPWATCH_HEIGHT) -0.1
        }
    }

    /* Here is the behaviour for the stopwatch: */

    begin.tcl

    /* This function updates the time display on the stopwatch. */

    proc display_time {hrs min sec} {
        if {$hrs < 10} {
            set hrs "0$hrs"
        }
        if {$min < 10} {
            set min "0$min"
        }
        if {$sec < 10} {
            set sec "0$sec"
        }
        set view [dive_find_sub_byname [dive_self] "text"]
        dive_text $view "$hrs:$min:$sec"
    }

    /* This function removes initial zeros in numbers. Example:
     * 08 -> 8. It is used to convert the output from dive_date. If
     * the initial zero isn't removed, 01-09 are interpreted as
     * octal numbers and syntax errors are genereted whenever Tcl sees
     * 08 and 09 are seen (since they are invalid octal numbers).
     */

    proc stripzeros {value} {
        regsub ^0+(.+) $value \\1 retval
        return $retval
    }

    /* A DIVE timer is set up to invoke this function every 100
     * milliseconds if the stopwatch is running. If the system
     * time has changed since the last time the function was
     * invoked, the stopwatch time is updated.
     */

    proc on_timer {} {
        global last_sec

        set sw_hrs [dive_property_get [dive_self] "hrs"]
        set sw_min [dive_property_get [dive_self] "min"]
        set sw_sec [dive_property_get [dive_self] "sec"]

        set sec [stripzeros [dive_date %S]]

        if {$sec != $last_sec} {
            set sw_sec [expr $sw_sec + 1]
            if {$sw_sec == 60} {
                set sw_sec 0
                set sw_min [expr $sw_min + 1]
            }
            if {$sw_min == 60} {
                set sw_min 0
                set sw_hrs [expr $sw_hrs + 1]
            }
            display_time $sw_hrs $sw_min $sw_sec
        }

        dive_property_put [dive_self] "hrs" $sw_hrs
        dive_property_put [dive_self] "min" $sw_min
        dive_property_put [dive_self] "sec" $sw_sec

        set last_sec $sec
    }

    /* This function is the callback for the interaction event, i.e.
     * the event that is generated when the user clicks on the
     * stopwatch. It toggles the "active" property. If the
     * stopwatch is started, a timer is registered, otherwise
     * the timer is deregistered.
     */
```

```
    proc on_click {event vid type origin src_id x y z} {
        global last_sec

        set active [dive_property_get [dive_self] "active"]
        set active [expr 1 - $active]
        dive_property_put [dive_self] "active" $active

        if {$active == 1} {
            dive_property_put [dive_self] "hrs" 0
            dive_property_put [dive_self] "min" 0
            dive_property_put [dive_self] "sec" 0
            set last_sec [stripzeros [dive_date %S]]
            dive_timer [dive_self] 100 "on_timer"
        } else {
            dive_time_deregister [dive_self] "on_timer"
        }
    }


    /* This function is the callback for the ENTITY_NEW event. It
     * creates the properties needed for the stopwatch. It also
     * registers a callback for interaction events, i.e. when the
     * user clicks on the stopwatch.
     */

    proc on_new {type id origin} {
        dive_property_create [dive_self] "active" string 0
        dive_property_create [dive_self] "hrs" string 0
        dive_property_create [dive_self] "min" string 0
        dive_property_create [dive_self] "sec" string 0

        dive_register INTERACTION_SIGNAL DIVE_IA_SELECT [dive_self] \
            "" on_click
    }


    /* This code is invoked by all DIVE applications that loads the
     * stopwatch. We cannot create properties here since that would
     * result in multiple copies of the same property. The correct
     * way to solve this problem is to register a callback procedure
     * for the ENTITY_NEW event and create the properties there.
     * The ENTITY_NEW event is generated when an object has been
     * created. When the object has been created it is distributed to
     * all connected applications. The global Tcl variable "last_sec"
     * must be created here since it has to exist in the interpreter
     * of all distributed copies of the stopwatch object.
     */

    global last_sec
    set last_sec 0

    dive_register ENTITY_NEW_EVENT NULL [dive_self] "" on_new

    end.tcl
}
```

## Untransform

```
#
# untransform
#
# Transform a point from the world coordinate system into
# the local coordinate system of an entity.
#
# The result is stored in the global varaibles
# untransform_x, untransform_y and untransform_z.
#

proc untransform {entity x y z} {
    global untransform_x
    global untransform_y
    global untransform_z

    dive_entity_info $entity info

    scan $info(T0) "%f %f %f" tx ty tz

    set a [lindex [lindex $info(R0) 0] 0]
    set b [lindex [lindex $info(R0) 0] 1]
    set c [lindex [lindex $info(R0) 0] 2]
    set d [lindex [lindex $info(R0) 1] 0]
    set e [lindex [lindex $info(R0) 1] 1]
    set f [lindex [lindex $info(R0) 1] 2]
    set g [lindex [lindex $info(R0) 2] 0]
    set h [lindex [lindex $info(R0) 2] 1]
    set i [lindex [lindex $info(R0) 2] 2]

    set u_x [expr ($a * $x) + ($b * $y) + ($c * $z)]
    set u_y [expr ($d * $x) + ($e * $y) + ($f * $z)]
    set u_z [expr ($g * $x) + ($h * $y) + ($i * $z)]

    set tmp_x [expr ($a * $tx) + ($b * $ty) + ($c * $tz)]
    set tmp_y [expr ($d * $tx) + ($e * $ty) + ($f * $tz)]
    set tmp_z [expr ($g * $tx) + ($h * $ty) + ($i * $tz)]

    set untransform_x [expr $u_x - $tmp_x]
    set untransform_y [expr $u_y - $tmp_y]
    set untransform_z [expr $u_z - $tmp_z]
}
```