HEWLETT PACKARD

# Oculus: a visual user interface for the Pantheon storage system simulator

*Uwe Aicheler*

**Storage Systems Program**

**Computer Systems Laboratory**

**Hewlett-Packard Laboratories**

**HPL–SSP–96–1**, **January 8, 1996**

Approved for External Distribution

*Abstract*

*The Pantheon storage system simulator was built to allow exploration of a range of design choices in storage systems. Simulations can take several hours or days to execute and generate a large amount of performance data that is often hard to interpret. In order to gain more insight into the operation of the simulated system, we want to see what is happening inside the simulator.*

*Oculus is an X-window based graphical tool that allows the visualization of the state and performance of the simulated system during the simulation.*

# Contents

## Figures

## Tables

# 1 Introduction

## 1.1 Motivation

The Pantheon storage system simulator was built to allow exploration of a range of design choices in storage systems. Simulations can take several hours or days to execute and generate a large amount of performance data that is often hard to interpret. In order to gain more insight into the operation of the simulated system, we want to see what is happening inside the simulator.

Oculus should be an X-window based graphical tool that allows the visualization of the state and performance of the simulated system during the simulation.

This paper gives an introduction to the Pantheon storage system simulator and describes the Oculus visual user interface.

*Pantheon* is a big building in Rome from which the simulator got its name. The roof of this building, which lets light inside the *Pantheon* building, is called the *Oculus*.

## 1.2 Overview & Goals

The user interface Oculus should be a separate Unix process that is updated by the Pantheon storage system simulator automatically as soon as Pantheon has calculated new data. This data should be processed in Oculus and statistical data should be calculated. The user should be able to choose several viewers like a stripchart[1], a histogram or a simple text window to monitor data. These viewers should then also be updated automatically.

1. diagram with running time axis

## 1.3 The remainder of this paper

Chapter 2 gives an introduction to the Pantheon storage system simulator. Important features to understand the Oculus design are also explained.

If you just want to know how to operate Oculus, read the Short Oculus Manual on page 14 (chapter 3).

Chapters 4, 5 and 6 describe the analysis, specification, design, implementation and test of Oculus and extensions to the Pantheon simulator.

Design changes and alternatives we discussed during the development cycles are described in chapter 7 (development reviews). This chapter also has a section about enhancement possibilities, a reference section that describes how to extend Oculus and a conclusion section.

# 2 The Pantheon storage system simulator

This chapter gives an introduction and an overview of the Pantheon storage system simulator which already existed. Important features to understand the Oculus design like the Tcl configuration interface and measurement points are also explained.

## 2.1 Introduction

The Pantheon simulator was built to allow exploration of a range of design choices in *storage systems*: those hardware and software components of a computer system concerned with I/O to storage devices such as disks. The primary purposes of the simulator are to provide a flexible testbed for speeding up the process of experimenting with different design choices, and to provide performance predictions for designs that have yet to be built.

The simulated storage systems span a wide range of complexity and sophistication. The simplest is a host processor connected to a single string of disks through a channel such as SCSI. At the other, complete parallel system implementations with embedded or separate parallel I/O systems, such as DataMesh [Wilkes92], can be simulated.

The Pantheon simulator was designed to allow exploring changes in several pieces of hardware. We are interested both in possible designs and also in emulating hardware with different performance characteristics. A partial list of hardware elements in the simulator model includes:

- host processors
- host-host interconnects
- host-storage system interconnect
- processors in the storage system
- internal interconnects inside the storage system
- magnetic disks
- (non-volatile) RAM caches
- disk rotation-position-sensing hardware and spin-synchronization.

Pantheon is used to explore a wide range of software designs. Here is a sampling of the kind of algorithms and policies to explore:

- request sequencing or scheduling
- parity calculation for RAIDs
- inter- and intra-request concurrency
- block-placement and file system layouts
- data compression
- storage hierarchies

Pantheon is also used to explore the effects of performing these functions at different places in the storage system — for example, at the host processor or in the I/O system itself.

The system designs made possible by the above components can be exercised in the simulator against real workloads (derived from tracing I/O patterns on existing systems), or against synthetic workloads generated inside the simulator.

## 2.2 Simulator overview

This section provides an introduction to the overall approach and architecture of the simulator.

### Implementation

Pantheon is an event driven simulator. It is written in C++, and uses Tcl as a configuration interface to model a range of possible system configurations. At the time of writing, the simulator contains about 94k lines of C++ code (170 classes) and 12.5k lines of Tcl.

The Raphael C++ multithreading library [Golding 95b] provides coroutines (classes called "threads") that represent independent activity centers in the model—in this case, things like disks and interconnect channels. Virtual (simulated) time in the model is advanced when a thread determines that it needs to perform some activity (e.g., a disk seek), and calls `delay` to indicate that it is busy for the amount of time the activity would take.

The collection of components provided by the simulator is assembled at runtime into a configuration to be modelled. Thus, the system topology is specified as part of the simulator's input (as Tcl script), rather than being completely determined at compile time. One of the simulator's initialization activities is to establish the interconnections between the components, and to set the performance parameters of each of the system components being modelled.

### Development environment

The Pantheon simulator has multiple development trees. There are multiple separate modules with a separate directory per module (such as `Oculus/` ). Libraries are installed in `/lib` and executables in `/bin`.

CVS (Concurrent Version System) is used to allow parallel development.

## Simulator structure

The simulator is composed of a great many different C++ classes and the components (modules) that they go to make up.

About the simplest model possible is shown in Figure 1. Even this simple system shows many of the components of the simulator. Working from the top of the picture downwards, we can identify the following components:

- *Stats (Sensor)*: Overall statistics about the experiment are collected here.
- *Host*: an entity that represents a processor where application and system cycles can be consumed. Workloads run on host; hosts communicate with each other and with the storage subsystem.
- *IOload*: a workload task that generates a sequence of requests representing a single workload (e.g., from a single host process). Each IOload is parameterized by an *IOpattern*, or reads and replays a trace file.
- *Device driver*: represents the portion of the host hardware and software that is used to communicate with a single (virtual) disk in the storage subsystem.
- *Disk channel*: a path by which requests and data are communicated between the device driver and the storage elements (disks, here).
- *Disk*: a rotating magnetic storage device, including hardware and software for connecting to the disk channel, buffering data, and a medium on which data is stored.

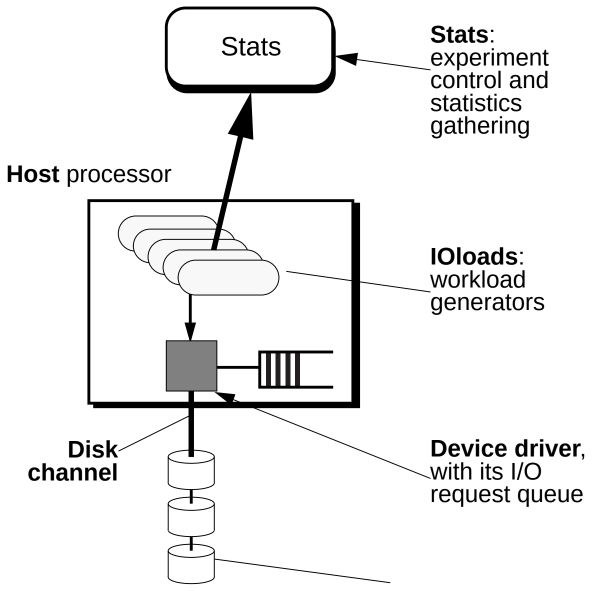

Stats
Stats: experiment control and statistics gathering
Host processor
IOloads: workload generators
Disk channel
Device driver, with its I/O request queue

**Figure 1** A simple Pantheon simulation

## Statistical details

To increase the accuracy of the numbers, the simulator "warms up" the system by running its workload for an initial period before taking any measurements. The number of warm-up requests that must complete before statistics are accumulated can be specified, as can the total number of I/O operations that are to be measured. The simulator keeps the workload at full steam until the last request that is to be measured has completed. Together, these eliminate start-up and termination effects, and the numbers that the simulator produces are thus those for the system's behavior in steady-state.

The simulator initializes all its random number generators from a single seed generator (a random number generator that emits values used as seeds). This mechanism means that all the random number generators don't start out with the same seed value, so (for example) not all the synthetic workload generators attempt to write to exactly the same block on their first request. By default, the seed generator is itself initialized with a seed of zero, so that runs are repeatable. For truly random behavior between runs, the seed generator can optionally be initialized with a seed derived from the low-order bits of the time-of-day clock.

## Disk timing verification

Some experiments were conducted to compare the model against real disk drives. The physical I/O times for a week of I/O traces to a set of disks were captured. Then the same I/O requests were fed into the simulator, and the modelled I/O times were compared against the real ones. The results are shown in Figure 2 as emulative distribution diagrams (x-axis: time/ms, y-axis: fraction of I/Os).

As the graph shows, the agreement is quite good. The marginal discrepancy in the Quantum case is due to the real Quantum disk drive not using immediate reporting for quite as many requests as the description above would suggest. However, this is a small discrepancy, and is unlikely to represent a significant difference in overall performance.

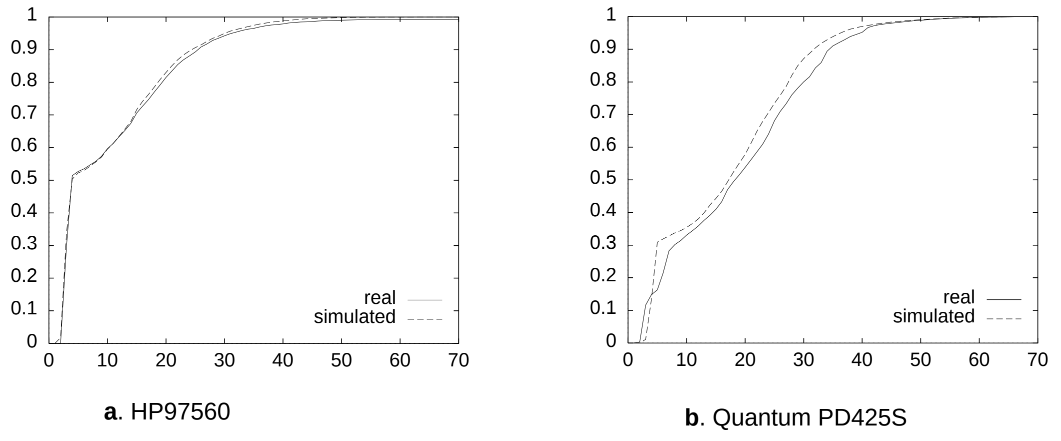


**Figure 2** measured and simulated physical disk I/O times for the simulator's disk models.

## 2.3 Tcl as configuration interface

Tcl is an interpretive language that was developed by John Ousterhout at the University of California at Berkeley. It is implemented as a C library and contains the parser and some general routines. This library can be integrated in existing C programs to get a mighty macro or script language. The interpreter can be easily extended with additional commands to access functions and data of the program.

The following simple Tcl example prints: `name2 contains Uwe`

```
set name1 Uwe
set name2 $name1
set text "name2 contains $name2"
puts $text
```

Pantheon uses Tcl to configure the simulated system [Golding 94]. Each C++ simulation object class has a function (registered with Tcl as a command of the same name as the class) that builds a new instance of the class, constructs a unique name for it, and enters this into a hashed lookup table that maps the text name to a pointer to the new object. (This structure is very similar in intent to the object table used in SmallTalk interpreters [Goldberg 94]).

Thus:

```
set link [Link "SCSI-bus" -b 10e6]
```
constructs an object to represent a SCSI bus with a bandwidth (-b) of 10MB/s, generates a name-string for it, and assigns this string to the Tcl variable link. Now this Tcl variable can be treated just like a C++ pointer to the object itself.

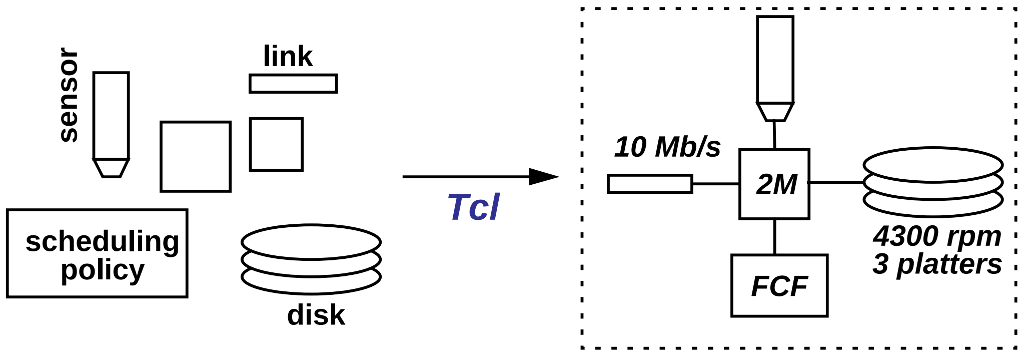


**Figure 3** Tcl as glue language (toolkit approach)

**Execution Flow**

The simulator execution flow is as follows:

**runPantheon** **(shell script)**

- generates main Tcl script
- calls Pantheon executable

**Pantheon** **(C++ program)**

- creates Tcl interpreter
- adds C++ interface functions to Tcl interpreter
- executes configuration scripts on interpreter

**configuration scripts** **(Tcl scripts)**

- configure simulation architecture in Pantheon
- call Oculus (optional)
- start the simulation

## 2.4 Measurement points

Some simulation classes provide measurement points to allow Stats objects to accumulate data and to calculate statistical data. The measurement point can be a pointer to a Stats object or null (not present). The Tcl configuration code decides if the measurement point should be used or not.

The C++ class hierarchy has a number of different Stats objects (Stats, StatsHistogram, StatsHistogramUniform, StatsHistogramLog, ...) which can be used as a sensor and calculate simple statistical values (Stats) as well as large data like histograms (StatsHistogram).

At the end of the simulation, the Stats objects hand their data to a *Reporter* object, which collects their current state and stores the data as Tcl scripts. This data can afterwards be evaluated or used by tools to generate diagrams.

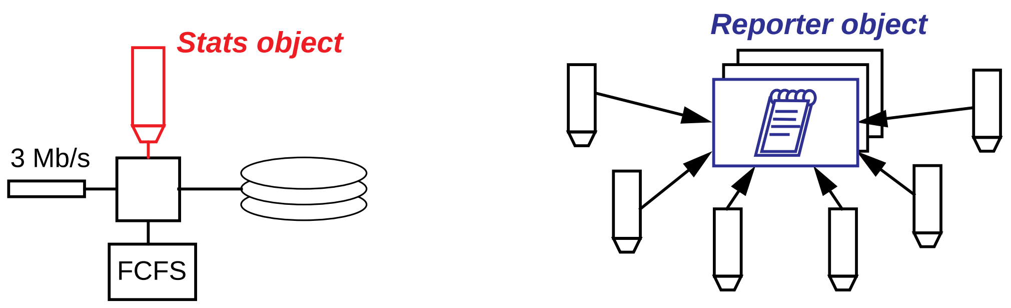


**Figure 4** Stats object as a sensor at a disk; Reporter object converts data

# 3 Short Oculus Manual

In order to better understand the design and implementation details of Oculus, it is useful to first have an overall picture of what Oculus does. In order to provide such an overview, this section presents a short Oculus user manual.

## 3.1 Starting the simulator with Oculus

Start the Pantheon simulator with: `make test`

To start the Pantheon simulator with Oculus, you must specify in the test parameter file (`test.parms`): `-O <OculusLevel>`

where `OculusLevel` is the threshold which decides if an object should be monitored or not. The higher the `OculusLevel`, the more Stats objects will be monitored, -1 means that Oculus should not be invoked. This file will be evaluated before the simulation starts.

The following example will invoke Oculus with `OculusLevel` 1: `-O 1`

Make sure that the procedure `InitOculus` (defined in `Pantheon/oculus_proc.tcl`) is called as one of the first commands of the Tcl scripts.

## 3.2 Selecting Pantheon objects to monitor in Oculus

The objects in Pantheon that should be monitored in Oculus must be specified in a Pantheon Tcl script. To select an object the Tcl procedure `create_OculusCtrl` (defined in `Pantheon/oculus_proc.tcl`) must be called after the object create command in a Tcl script. This function creates a *Ctrl* object in Oculus which controls incoming data and the Pantheon object is connected to a Reporter that updates are reported.

```
proc create_OculusCtrl {ObjName ObjOculusLevel stat_cmd} {
#------------------------------------------------------------------------------
# Function to monitor an object.
# It tries to register the object on Oculus and
# establishes the connection between Stats and OculusReporter in Pantheon.
# Arguments: ObjName:  Pantheon name of the object
#            ObjOculusLevel:Level of the object, if this is higher than the
#                      global OculusLevel the Ctrl is not created and
#                      the object is not monitored!
# Returns:   1 if it is monitored
# Sideeffects:no
#------------------------------------------------------------------------------
```

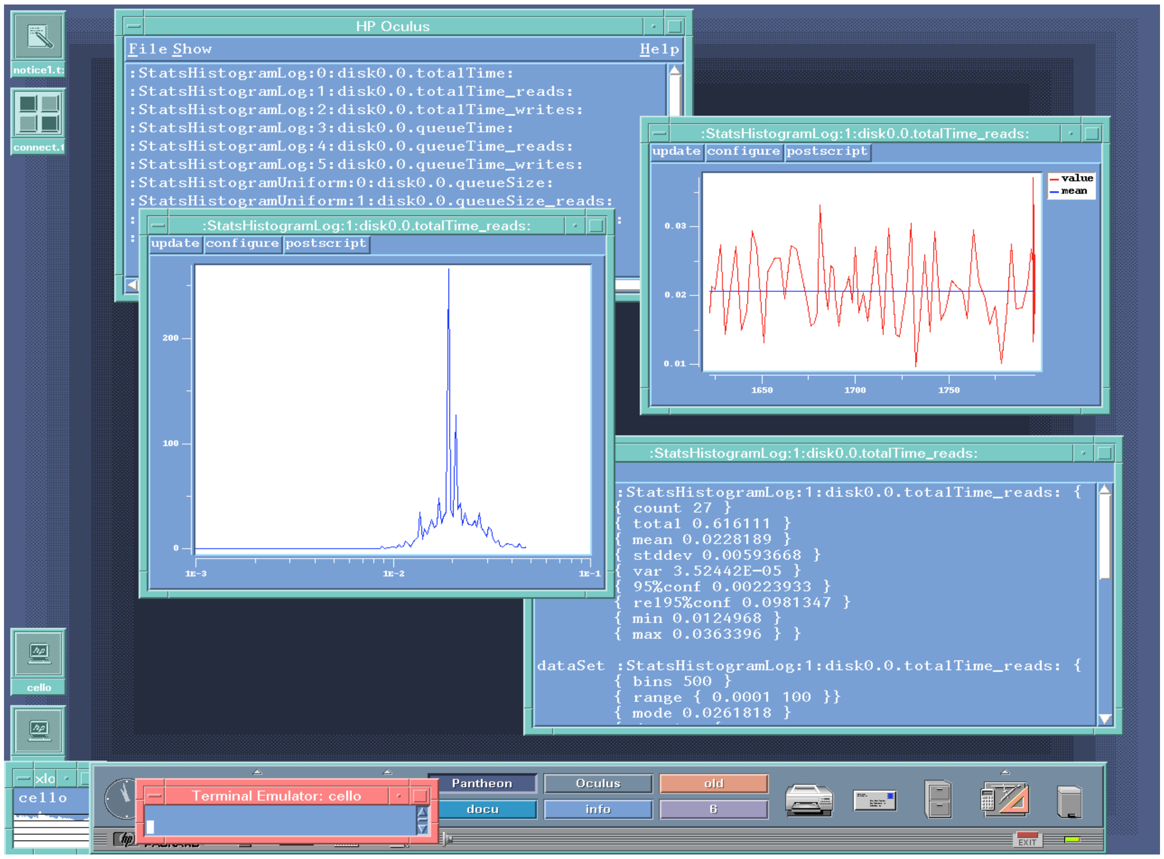
**Figure 5** Function definition create_OculusCtrl

## 3.3 The Oculus Menu

After Oculus is invoked by Pantheon, a list of the Ctrl's will be displayed in the main window. The user can e.g. mark an item with the mouse and select commands from the main menu to show a viewer of this object. A viewer is a window that monitors data of an object.

## Menu commands

- **File - Save Window Positions**
  Opens a file select window to save (viewers) windows positions to.
- **File - Load Window Positions**
  Opens a file select window to load windows positions from.
- **Show - Textwindow**
  Creates a new text window (viewer) with the data of the selected object in the listbox. The data can be updated by pressing the "update" button. The text can be selected in the window and copied to the clipboard with [Ctrl] [Insert].
- **Show - Stripchart**
  Creates a new stripchart viewer with the data of the selected object in the listbox. The display is updated automatically. This command is only available if the selected object supports a histogram.
- **Show - Histogram**
  Creates a new histogram viewer with the data of the selected object in the listbox. The display is updated automatically. This command is only available if the selected object supports a histogram.
- **Help**
  Shows information about Oculus.



```
:StatsHistogramLog:0:disk0.0.totalTime:
:StatsHistogramLog:1:disk0.0.totalTime_reads:
:StatsHistogramLog:2:disk0.0.totalTime_writes:
:StatsHistogramLog:3:disk0.0.queueTime:
:StatsHistogramLog:4:disk0.0.queueTime_reads:
:StatsHistogramLog:5:disk0.0.queueTime_writes:
:StatsHistogramUniform:0:disk0.0.queueSize:
:StatsHistogramUniform:1:disk0.0.queueSize_reads:
```

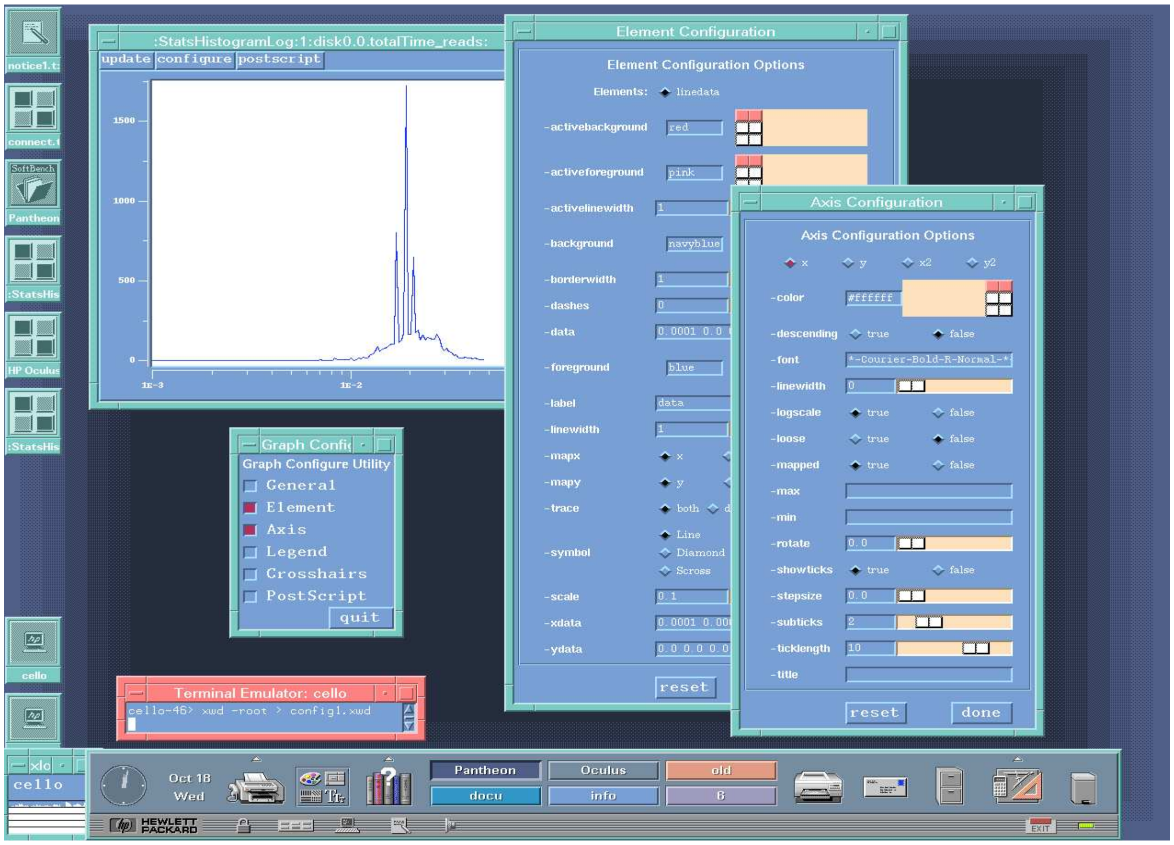
```
         :StatsHistogramLog:1:disk0.0.totalTime_reads: {
         { count 27 }
         { total 0.616111 }
         { mean 0.0228189 }
         { stddev 0.00593668 }
         { var 3.52442E-05 }
         { 95%conf 0.00223933 }
         { rel95%conf 0.0981347 }
         { min 0.0124968 }
         { max 0.0363396 } }

dataSet  :StatsHistogramLog:1:disk0.0.totalTime_reads: {
         { bins 500 }
         { range { 0.0001 100 }}
         { mode 0.0261818 }
```

**Figure 6** screenshot of Oculus with a stripchart, a histogram and a text-viewer

## 3.4 Graph windows

Graph windows like stripchart and histogram provide special features:

- **configure**
  Opens the graph configure application (which was copied from a demo of the BLT widget library). This application lets you configure the diagram, elements, axis, legend, crosshairs and the Postscript options of the graph widget.
- **postscript**
  Generates a postscript file of the diagram.
- **update**
  This button updates the diagram immediately. The diagram is usually updated automatically on every 5 data updates for the stripchart and every 20 data updates for the histogram. (This can be changed by altering the `update_interval` variable of the according class.)
- **zooming**
  You can use your mouse to zoom the diagram by pressing the left mouse button and dragging the mouse over the interesting area. The diagram is repainted with the new scope/scale when you release the mouse button. If you press the left mouse button beside the graph area, the diagram will be repainted with the original scope.
- **active legend**
  When you move your mouse over an item of the legend the according curve in the graph will be highlighted.



```
cello-46> xwd -root > config1.xwd
```

Element Configuration

Element Configuration Options

Elements: [✓ linedata]

| | |
|---|---|
| -activebackground | red |
| -activeforeground | pink |
| -activelinewidth | 1 |
| -background | navyblue |
| -borderwidth | 1 |
| -dashes | 0 |
| -data | 0.0001 0.0 0 |
| -foreground | blue |
| -label | data |
| -linewidth | 1 |
| -mapx | [✓ x] |
| -mapy | [✓ y] |
| -trace | [✓ both] d |
| -symbol | [✓ Line] Diamond Scross |
| -scale | 0.1 |
| -xdata | 0.0001 0.00 |
| -ydata | 0.0 0.0 0.0 |

reset

Axis Configuration

Axis Configuration Options

[✓ x] y x2 y2

| | |
|---|---|
| -color | #ffffff |
| -descending | true [✓ false] |
| -font | *-Courier-Bold-R-Normal-* |
| -linewidth | 0 |
| -logscale | [✓ true] false |
| -loose | true [✓ false] |
| -mapped | [✓ true] false |
| -max | |
| -min | |
| -rotate | 0.0 |
| -showticks | [✓ true] false |
| -stepsize | 0.0 |
| -subticks | 2 |
| -ticklength | 10 |
| -title | |

reset done

**Figure 7** Oculus with the graph configure tool

# 4 Analysis & Specification

This chapter describes the Rapid Prototyping software development process and the analysis and specification of Oculus.

To make the analysis, specification, design, implementation and test of Oculus easy to understand, this and the following chapters describe the last development cycle of the Rapid Prototyping process. If you are interested in important decisions and alternatives we discussed during the development, read section: Important Decisions on page 38.

## 4.1 Development process: Rapid Prototyping

Because Oculus will be used as a tool to help us evaluate I/O experiments, it will evolve to include new functionality as we use it and discover how it can be improved. The Phase Review software process, which is quite rigid, is not well adapted to the development of this kind of tool.

Instead, we use an iterative development process (also called Rapid Prototyping or spiral design model) where some basic features are specified, designed, implemented and tested, and then the design is refined and additional features are added iteratively. This method is very commonly used in the software industry.

The process is like a spiral moving through the 4 fields analysis, design, implementation and test beginning from the inside (analysis) and ending at the outside (test) field in several cycles.

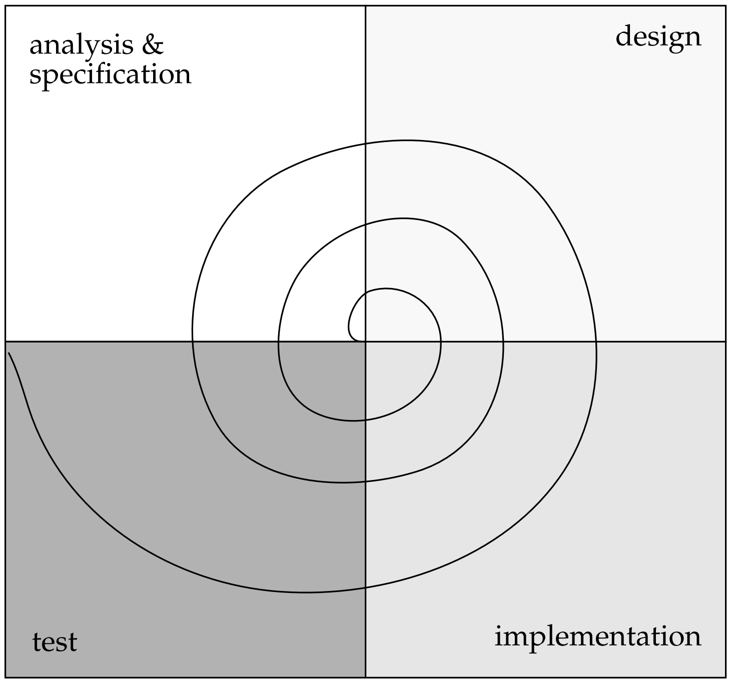


**Figure 8** Rapid Prototyping: several cycles of development

## 4.2 Oculus Goals

The goals for the Oculus user interface are defined as follows:

- **Monitoring of Pantheon Stats objects during simulation**
  The data of statistical objects should be displayed during the simulation and not only after the simulation is finished.
- **Pantheon not slower when Oculus not used**
  Pantheon should not lose any performance because of changes if Oculus is not used.
- **Active user interface**
  Oculus should not be a "dead" application where the user cannot interact during the simulation. Instead, the user should be able to perform actions like changing viewers.
- **Different charts / viewers**
  The user should have a choice of different ways to visualize the data.
- **Flexible and extensible architecture**
  The design of Oculus should be as flexible as possible to make it easy to extend or to make changes to Oculus.
- **Easy to use**
  This is a goal of every user interface.

## 4.3 General Ideas

Oculus is a separate Unix process and receives its data from the Pantheon simulator process.

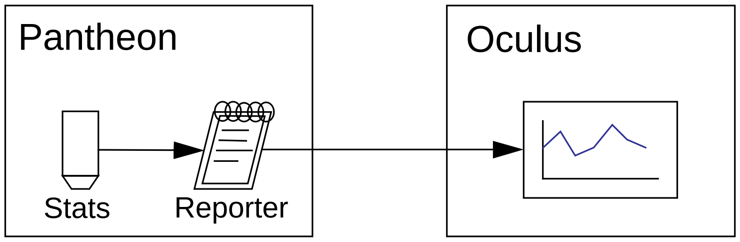


**Figure 9** Dataflow Pantheon - Oculus overview

The basic idea is to use the Stats objects (that are already used in Pantheon as a sensor and for statistical calculation) and connect them to a special Reporter ("`OculusReporter`"). The difference is that they should send updates to the Reporter immediately after they receive an update. The Reporter then sends the data through a communication channel to Oculus.

On the Oculus side one Ctrl object exists for every Stats object on Pantheon that should be monitored. The Ctrl objects process the incoming data and handle the viewer windows. The fact that a Pantheon object "should be monitored" doesn't mean that its data is always displayed in a viewer on the screen. It just means that the corresponding Ctrl object in Oculus is updated and the user can choose a viewer to see the data.

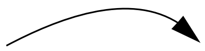
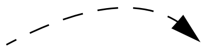
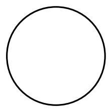
## 4.4 Structured Analysis

This section shows the Oculus analysis with the help of Tom De Macro's Structured Analysis (SA) and Real-Time Modeling of Hetley/Pirbhai. This technique is extended here by the control & data flow, which combines both control and data within one arrow. In the data dictionary the control specification appears as the arrow's name. The data specification appears with the same name plus the suffix "_data". There can be more SA processes (objects) of the same process type (class). Those are indexed with 1, 2, 3 in the figures.

If you don't know Structured Analysis: please note that the Structured Analysis processes don't need to be Unix processes or tasks or threads! SA processes are centers of activity in the system.

This section is also useful because every SA process with a name that ends with "_object" is represented as an object in the design. Those figures show how the objects are designed to glue together.

**Table 1** Description of used signs

| | |
|---|---|
|  | data flow |
|  | control flow |
| | control & data flow |
|  | process, action (please note that a Structured Analysis process doesn't need to be a real Unix process or a separate task) |
| | source or destination for control or data flows |

## Overview

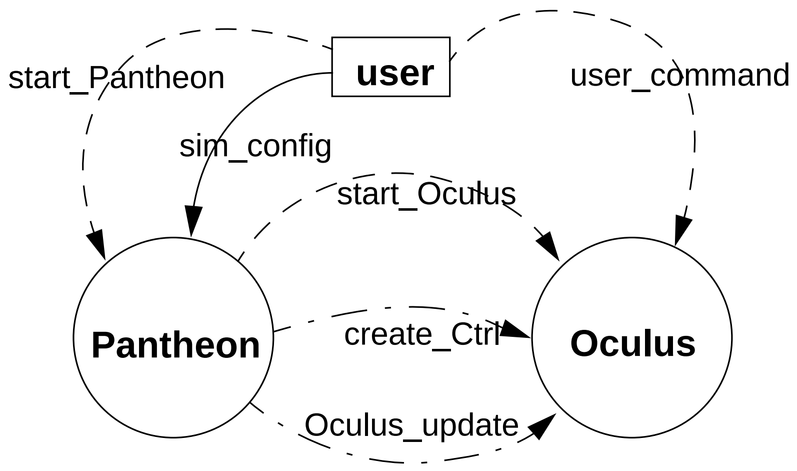


**Figure 10** Structured Analysis -Overview

## Pantheon

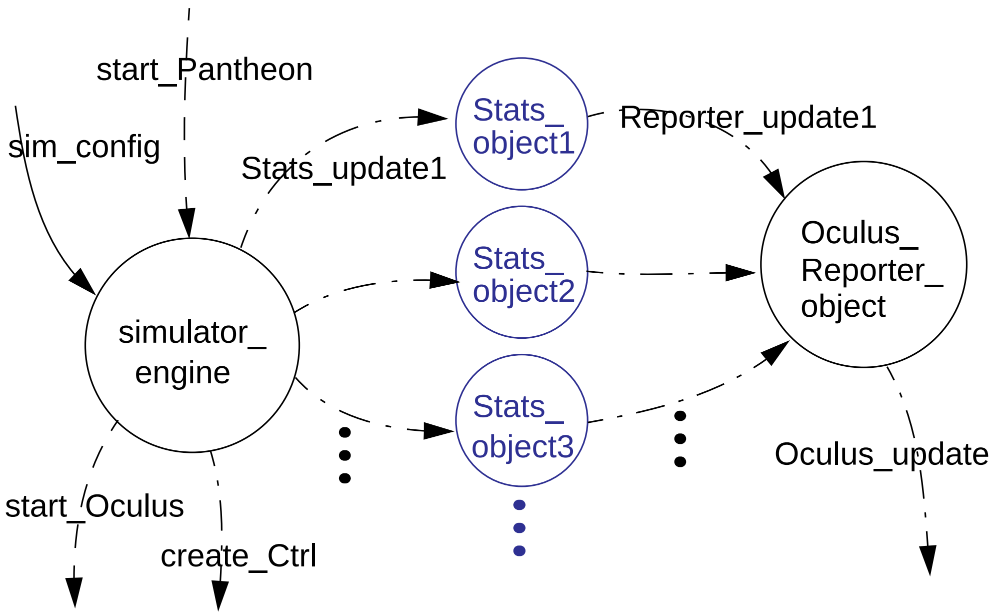


**Figure 11** Structured Analysis - Pantheon

Process specifications:

| | |
|---|---|
| process name: | `simulator_engine` |
| input: | `sim_config` (data), `start_Pantheon` (control) |
| output: | `start_Oculus` (control), `create_Ctrl` (control & data), `Stats_update1` (control & data) |
| body: | |
| The `simulatior_engine` represents the simulator core with the simulated system. It is not detailed further, because the subprocesses are not important to understand Oculus. The `sim_config` input flow specifies the simulation configuration given by the user, who can start the simulation with the control flow `start_Pantheon`. `Start_Pantheon` will activate the necessary subprocesses which send updates through the `Stats_update` control and data flows to the `Stats_object` SA processes. | |

| | |
|---|---|
| process name: | `Stats_object` |
| input: | `Stats_update` (control & data) |
| output: | `Reporter_update` (control & data) |
| body: | |
| This process (object) receives statistical data for one measurement point through the `Stats_update` control & data flow. The data is sent directly to the `Oculus_Reporter_object` through the `Reporter_update` flows. | |

| | |
|---|---|
| process name: | `Oculus_Reporter_object` |
| input: | `Reporter_update` (control & data) |
| output: | `Oculus_update` (control & data) |
| body: | |
| This SA process (object) updates Oculus. It gets the data from the `Reporter_update` flow, converts the format and sends the data through the `Oculus_update` control & data flow to Oculus. | |

**Oculus**

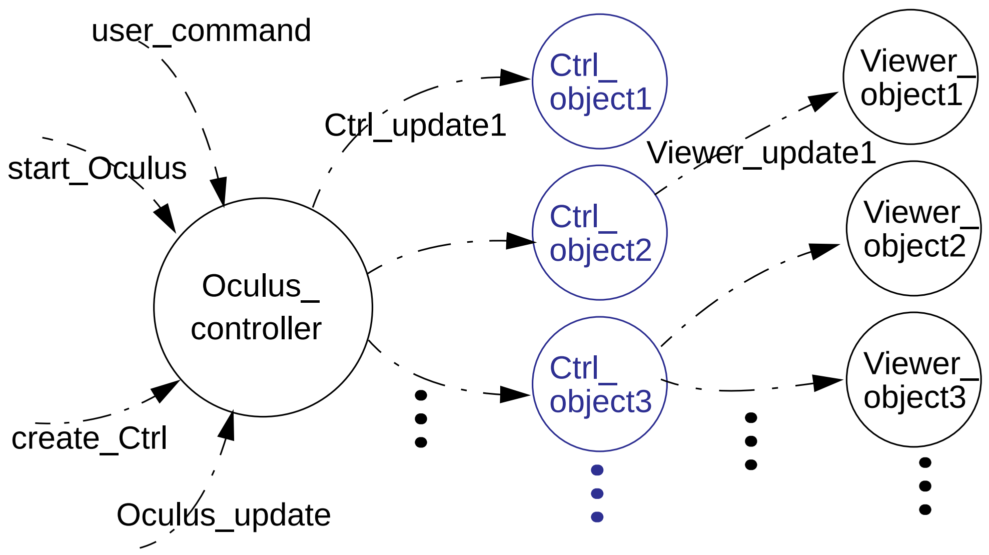


**Figure 12** Structured Analysis - Oculus

Process specifications:

| | |
|---|---|
| process name: | `Oculus_controller` |
| input: | `start_Oculus` (control), `create_Ctrl` (control & data), `Oculus_update` (control & data), `user_command` (control) |
| output: | `Ctrl_update` (control & data) |

body:

`Oculus_controller` represents the Oculus main window and the Oculus core which processes incoming commands from Pantheon and X-Window events (`user_command` control flow). Incoming control & data flows (commands) from Pantheon can be `create_Ctrl` which should create Ctrls in Oculus, or `Oculus_update` with new data. The `start_Oculus` control flow displays the main window and activates all subprocesses.

| | |
|---|---|
| process name: | `Ctrl_object` |
| input: | `Ctrl_update` (control & data) |
| output: | `Viewer_update` (control & data) |
| body: | |
| A `Ctrl_object` can be a parent objects for no, one or more `Viewer_objects`. This process gets new values through the `Ctrl_update` control & data flow to store and to calculate of statistical data. This data will then be sent through the `Viewer_update` control & data flow to existing `Viewer_objects` to be displayed on the screen. | |

| | |
|---|---|
| process name: | `Viewer_object` |
| input: | `Viewer_update` (control & data) |
| output: | |
| body: | |
| This process receives its data from `Viewer_update` control & data flow and visualizes it on the screen in some way. Different viewers exist for different charts. | |

## Data dictionary

| | |
|---|---|
| `Ctrl_update` (control flow) | `update` or `no_update` |
| `Ctrl_update_data` (data flow) | `time` + `value` |
| `Oculus_update` (control flow) | `update` or `no_update` |
| `Oculus_update_data` (data flow) | `ObjName` + `time` + `value` |
| `Reporter_update` (control flow) | `update` or `no_update` |
| `Reporter_update_data` (data flow) | `ObjName` + `time` + `value` |
| `Stats_update` (control flow) | `update` or `no_update` |
| `Stats_update_data` (data flow) | `time` + `value` |
| `Viewer_update` (control flow) | `update` or `no_update` |
| `Viewer_update_data` (data flow) | data format which is accepted by the various viewers (can be a list of data points or histogram data) |
| `create_Ctrl` (control flow) | 1 or 0 |
| `create_Ctrl_data` (data flow) | `ObjName` + `args` (args depend on the class of the object; normally this is the Tcl command to create the same C++ object as the Pantheon object) |
| `sim_config` (data flow) | whole configuration of the experiment (this is too large to explain here) |
| `start_Oculus` (control flow) | 1 or 0 |
| `start_Pantheon` (control flow) | 1 or 0 |
| `user_command` (control flow) | `mark_list` or `show_stripchart` or `show_histogram` or `show_textwindow` or other new commands |

## 4.5 Communication Protocol Specification

Since Oculus is a separate process, a communication protocol needs to be specified so that Pantheon can create Ctrls in Oculus and send updates to Oculus. Table 2 shows the initalization steps.

**Table 2** Pantheon - Oculus initialization protocol

| | |
|---|---|
| Pantheon | invokes Oculus and establishes the communication channel |
| Oculus | creates the main window |
| Pantheon | interprets Tcl scripts and sends OculusCreateCtrl <name> |
| Oculus | creates a Ctrl object and<br>updates the List of Stats in main window |
| Pantheon | starts the simulation and<br>sends data updates to Oculus |

To create a Ctrl on the Oculus side the following command must be sent to Oculus:

```
OculusCreateCtrl <ObjName> <args>
|                 |         |
|                 |         ----e.g. initialization arguments
|                 ----e.g. ":Stats:3:disk0.0.length_reads:"
--- Tcl procedure
```

The ObjName must be a unique name and the type (class) of the object must appear between the first two colons!

To send data to Oculus the following command must be sent to Oculus:

```
dataSet <ObjName> <DataList>
|         |
|         ----e.g. ":Stats:3:disk0.0.length_reads:"
--- Tcl procedure
```

where the following is an example of the format of DataList.
(This format can be evaluated as a Tcl-list on the Oculus interpreter.)

```
{ { value 1537 } { mean 5302.55 } }
```

# 5 Design

This chapter describes the design of the overall project.

## 5.1 Communication Design

Figure 13 shows the asynchronous communication channel from Pantheon to Oculus. To make the channel accessible in Pantheon from the C++ scope as well as from the Tcl scope, a Unix pipe is used where Pantheon can send out commands with data. Another advantage of a pipe is that the Pantheon simulator doesn‘t block because of something like a remote procedure call.

To avoid blocking on the Oculus side the commands that are piped out from Pantheon are invoked with the Tk-send command on the Oculus interpreter. Tk-send can be called within a Tcl/Tk script with a name of another running Tk-interpreter and a command as arguments. The command will then be invoked on the specified interpreter as if this command would be called within the script of that program. Thus, X-events can be processed in Oculus without waiting or polling for an incoming Pantheon command.

### The connect process

To meet both requirements an additional process (Tk-script) handles the communication in a way that both processes can run at the same time. After the simulation is started Pantheon will invoke connect and connect will invoke Oculus. The commands from the pipe are invoked on the Oculus side with the Tk-send command. They should be separated by 2 newlines in the pipe.

As an option Pantheon can send `RPC` as a prefix with the command to wait for a result of Oculus. The result (result of the Tk-send command) will be sent back to the pipe again. Because there‘s no need to get a result yet, this option is not used.

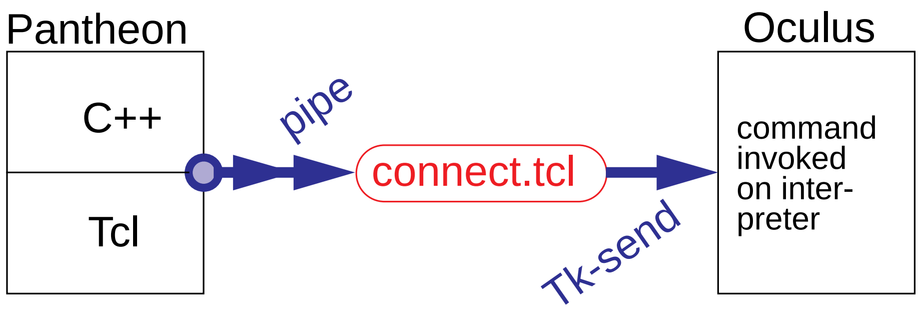


**Figure 13** communication design Pantheon - Oculus

## 5.2 Pantheon Extensions

### The Tcl interface

Some additional Tcl procedures allow Tcl to invoke Oculus, to establish the connection and to send commands (with data) to Oculus. These procedures can be accessed from the Tcl scope as well as from the C++ scope (by invoking `Tcl_Eval(...)` which is a service function of the Tcl C library). For better performance, the Oculus pipe identifier can be handed to the C++ scope to output directly to the pipe.

If the user interface is switched on, a global OculusReporter object is created from the Tcl scope; all Stats objects will be attached to the OculusReporter if they should be monitored (see Structured Analysis on page 19). OculusReporter is an instance of the C++ class ReporterTCL and handles the data conversion and output to the pipe for Oculus.

### C++ extensions

The Stats objects are extended with a pointer to OculusReporter. This pointer also serves as a flag: `NULL` means that the Stats should not be monitored. Whenever the Stats object is updated (when its `add()` method is called) and the object should be monitored, time and value will be passed to the OculusReporter to keep the corresponding Ctrl on the Oculus side up-to-date.

The Stats objects to monitor are selected during the build-up phase of the simulator. The Tcl procedure `create_OculusCtrl {ObjName ObjOculusLevel stat_cmd}` can be used (defined in `Pantheon/oculus_proc.tcl`). Actions of `create_OculusCtrl` are:

- If the `ObjOculusLevel` (given as parameter) is higher than the global `OculusLevel`[1] the object will not be monitored and the function returns with 0. If not the function ...
- sends the command `OculusCreateCtrl` to the Oculus pipe and ...
- establishes the link between Stats and `OculusReporter` in Pantheon.

The sequence of a data update is the following:

- If the Stats object is updated during the simulation and the `OculusReporter` pointer of the object is not `NULL`, time and value will be handed to the `OculusReporter`.
- `OculusReporter` will pipe out the Tcl command with the data to the connect process.
- This process invokes the command via Tk-send on the Oculus interpreter, which causes the update on the Oculus side.

1. described in: Starting the simulator with Oculus on page 14

## 5.3 The Oculus interpreter

The Oculus design should be as flexible as possible to simplify making extensions and the code should be as reusable as possible. Therefore a powerful language is needed to implement Oculus. The following packages are combined to one Oculus Tcl interpreter:

- **Tcl**
  Tcl is an interpretive language which was developed by John Ousterhout at the University of Berkeley. For a short introduction read section: Tcl as configuration interface on page 12.
- **Tk**
  Tk is a standard Tcl extension for X-windows user interfaces.
- **BLT**
  BLT is an extension which adds graph and diagram widgets to Tk.
- **[incr Tcl]**
  A better name would be Tcl++. This package adds object oriented features to the Tcl language. Figure 14 shows a simple example.
- **Pantheon Stats objects**
  Since the Pantheon statistical objects (Stats objects) already have a Tcl interface, they are also added to the interpreter to use them for the calculation of statistical data and to store data.

```
itcl_class Ctrl_Base_abstract {
   protected name ""
   protected TextWin

   constructor {name_in args config} {
     set name $name_in
     set TextWin [Viewer_TextWin #auto $name]        # create text window
   }

   method update_TextWin {} {
     $TextWin update [eval virtual GetData]          # show data in text window
   }

   method GetData {} {
      puts stderr "abstract function called!"        # abstract function
   }
}
```

**Figure 14** object oriented [incr Tcl] class definition (simplified)

## How to integrate new libraries to the Tcl interpreter

The interpreter is implemented in the `Oculus/oculus_interp.C` file. If you want to add a new package to the interpreter, follow the instructions of its documentation. Usually you have to include a header file and add a call to the `XXX_Init` function of this package in the function `Tcl_AppInit(Tcl_Interp *interp)` in file `Oculus/oculus_interp.C`.

To get news about Tcl/Tk packages look at the `comp.lang.tcl` Internet newsgroup.

The creator of the [incr Tcl] package is Michael J. McLennan[1] who is expected to release a new version soon. He will be collaborating in future with Dean Sheehan who offers another object oriented Tcl extension called ObjectTcl[2].

## Exchanging the Tk package

If you want to exchange the Tk package please note that the `main()` function is within this library. This insures that the initialization of X-windows and the processing of the arguments is done right. Within the `main()` function of Tk the external function `Tcl_AppInit(Tcl_Interp *interp)` is called. This function is defined in our file `Oculus/oculus_interp.C` and extends the interpreter with our packages.

Since the `main()` function is in another library a call to `_main()` has to be placed as the first command of the `main()` function in the Tk library. This requires the C++ compiler.

Therefore the `tk-3.6.1` directory was modified and copied to:
`/usr/local/src/tk/tk-3.6.1_C++/`

This directory contains the following changes:

- The command `_main(argc, argv)` was added as the first statement of the `main()` function in file `tkMain.c` to make the library accessible for C++.
- The Makefile was modified that the new library is called `libtk_C++.a` and that only the library is installed.

1. email: mmc@mhcnet.att.com
2. Object Tcl home page: http://www.sco.com/Products/vtcl/objectcl/cover.html

## 5.4 The Viewer hierarchy

Figure 15 shows a hierarchy diagram of all *Viewer* and all *Ctrl* classes.

Viewers are classes which represent windows on the screen to visualize data in some way. Each Viewer object is has a Ctrl as a parent object which controls it.

- **Viewer_abstract** is the abstract base class for all Viewer classes.
  Functionality:
  - building, packing and binding the toplevel window
  - providing a toolbar for optional buttons
  - ability to save and load window positions
  > If you want to implement a new Viewer class, inherit from this base class.
- **Viewer_TextWin** represents a window to display data as text. The data can be updated by pressing the "update" button.
- **Viewer_Graph_abstract** is an abstract class that contains and handles a graph widget from the BLT library. Buttons exist to update and to configure the graph and to generate a Postscript file. Generating a Postscript file and zooming with the mouse are features of the graph widget so that there is no need to do this work from scratch. To configure the graph the Graph Config[1] application is invoked with the names of the interpreter and the graph widget as parameters. With the help of this information Graph Config can send commands directly to the widget to configure it.
  > Inherit from this class if your new Viewer class should contain a graph widget.
- **Viewer_Stripchart** adds a stripchart to the father class to monitor value and mean over time.
- **Viewer_Histogram** adds a histogram chart to the father class.

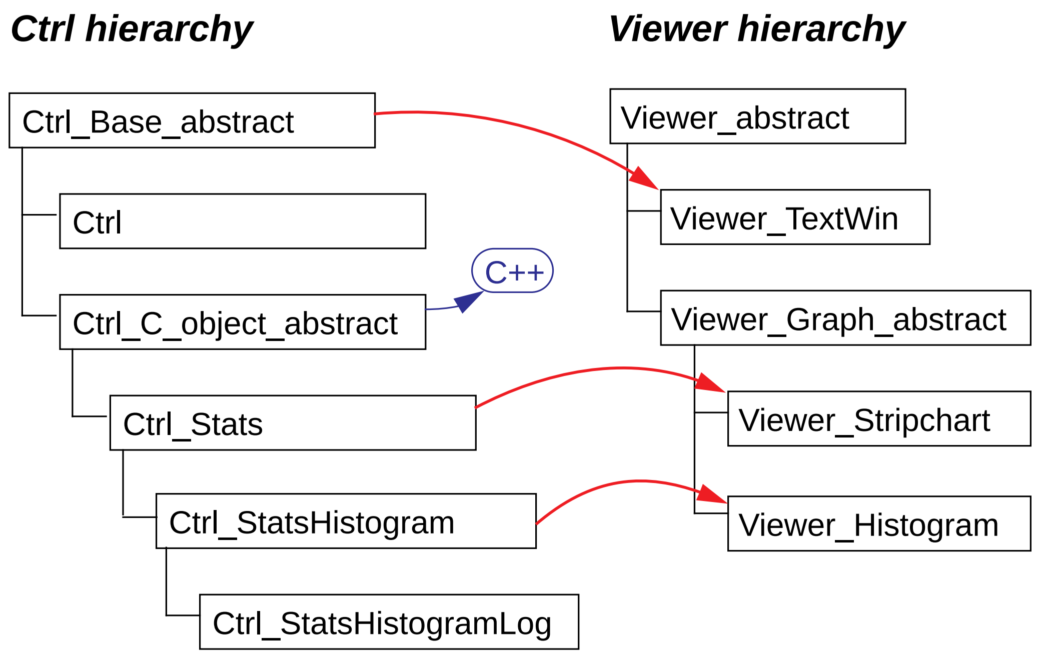


**Figure 15** Hierarchy diagram of all Ctrl and Viewer classes

1. Graph Config was copied from a demonstration application of the BLT package

## 5.5 The Ctrl hierarchy

Objects of this type are used to control data and Viewer objects. Each Ctrl handles data from an object on the Pantheon side that should be monitored. A Ctrl may contain several pointers to Viewer objects where its data can be visualized. Those pointers are represented as arrows in Figure 15 and they are also inherited. If you want to see how Ctrls and Viewers are glued together, look at Figure 12 on page 22.

Description of the Ctrl classes (see hierarchy diagram on the previous page):

- **Ctrl_Base_abstract** is the abstract base class for all Ctrl classes. It provides general functionality to receive data and to show it in a Viewer_TextWin window.
  > If you want to implement a new Ctrl class, inherit from this class.
- **Ctrl** is a simple usable class to display data in a text window. It is used whenever Oculus doesn't find a more specific class.
- **Ctrl_C_object_abstract** has basic functionality to use a C++ class for processing data.It simply "duplicates" the same class existing on Pantheon. This reduces the data transfer, because only data without a lot of statistical data (like e.g. histogram data) is transferred. That’s why the system is faster even if some statistical data is calculated twice.
  > Inherit from this class if your new class should have a C++ object (like a Stats object) to calculate or to store data.
- **Ctrl_Stats** uses a Stats object to calculate and store statistical values that can be displayed in a text window. It also keeps data lists of the data with a constant number of entries. This Ctrl can handle a text window (derived from the base class) and a stripchart window.
  > If the new Ctrl should monitor one of the Pantheon Stats hierarchy classes, inherit from the equivalent Ctrl_Stats class of this hierarchy.
- **Ctrl_StatsHistogram** uses a StatsHistogram C++ object which keeps histogram data inside. This data will be displayed in a Viewer_Histogram window. This Ctrl can handle a text window, a stripchart window and a histogram window.
- **Ctrl_StatsHistogramLog** uses a StatsHistogramLog C++ object (created by a father class) which keeps a histogram with logarithmically scaled bins inside. This data will be displayed in a Viewer_HistogramLog window with a logarithmic scale.

## 5.6 Names of Ctrls / Ctrl selection

Ctrl classes to monitor the data of a Pantheon object have the same name as the Pantheon object with "Ctrl_" as suffix. This makes the selection of the right Ctrl class a lot more easier when a `OculusCreateCtrl <ObjName> <args>` command arrives because the `ObjName` parameter contains the name of the Pantheon class.

The Ctrl selection algorithm can now simply add "Ctrl_" to the beginning of the ObjName and cut characters from the end until a class with the same name is found ([incr Tcl] has a function to find out if a class exists). This class will then be the most specific class and that's what we want. If a special Ctrl for a Pantheon object doesn't exist, the simplest class "Ctrl" will be selected to show at least a text window of the incoming data.

```
Pantheon object name:StatsHistogramUniform          XXX

selection algorithm:StatsHistogramUniform           XXX
               Ctrl_StatsHistogramUniform           Ctrl_XXX
               Ctrl_StatsHistogramUnifor            Ctrl_XX
               Ctrl_StatsHistogramUnifo             Ctrl_X
               Ctrl_StatsHistogramUnif              Ctrl_
               Ctrl_StatsHistogramUni               Ctrl
               Ctrl_StatsHistogramUn
               Ctrl_StatsHistogramU
               Ctrl_StatsHistogram

Oculus Ctrl name:Ctrl_StatsHistogram                Ctrl
```

**Figure 16** Example for 2 Ctrl selections

## 5.7 Ctrls with C++ Stats objects in Oculus

All classes derived from Ctrl_C_object_abstract have a pointer to a C++ object (usually an equivalent to the Pantheon object where the data is coming from).

When the object in Pantheon is updated, the corresponding object in Oculus will be updated as well. Due to that architecture, Oculus doesn't have to request a lot of data from Pantheon for example for updating a histogram viewer. (A histogram can contain thousands of data bins!)

### The data transfer between Ctrls and their C++ classes (Stats) in Oculus

Adding data to a Stats objects is very simple. All Stats objects have the same add() function where one value can be added and the Stats object will calculate the statistical values.

To get the statistical data out of the various Stats objects is a little bit trickier, because each class calculates (a lot) of different data. This problem was solved connecting the Stats with a global object of class ReporterTCL (`GlobalReporter`) which writes the converted data to a global pipe (`ReporterPipe`) **in Oculus**. Please note that this Reporter and this pipe is not the same that are used within Pantheon, although they are used for the same purpose.

The member function `Stats::report(Reporter *gr, const int detail)` hands statistical data to a Reporter specified as a parameter. The detail parameter describes the level of detail and can be used to add or leave out more detailed data.

The `GlobalReporter` handles the data conversion and output from a Stats object to `ReporterPipe`. The incoming data (like string, integers, reals, arrays, ...) are converted to a uniform Tcl command with a structured Tcl list holding the data. This command can then be read from `ReporterPipe` and the data string can be evaluated.

```
dataSet :Stats:2:disk0.0.totalTime_writes: {
       { count 4 }
       { total 0.0966072 }
       { mean 0.0241518 }
       { stddev 0.000405041 }
       { var 1.64058E-07 }
       { 95%conf 0.00039694 }
       { rel95%conf 0.0164352 }
       { min 0.0237106 }
       { max 0.024556 } }
```

**Figure 17** Example for a dataSet list from Stats objects

A smarter way to evaluate the data string is to directly invoke the command on the interpreter and let it do the work for us. The only thing to do is to provide a local virtual `dataSet` method for each class with a data list as an argument. Thus, the virtual `dataSet` method of the class (or any of a more specific class) will be invoked to process the data. This method can then simply access the variables in the data list without having to parse the hole string.

```
method dataSet {ObjNameDummy data} {
#--------------------------------------------------------------------------
# This method within the method update_Histogram (even if you see no call)
# The trick is explained within the method update_Histogram.
# Arguments:ObjNameDummynot used
#         data     data to evaluate
# Returns:no
# Sideeffects:no
#--------------------------------------------------------------------------
  # generates variables included in the list
  foreach element $data {
    set [lindex $element 0] [lindex $element 1]
    # e.g. set mean 0.0241518
  }
  ...
# now we can access the list elements as local variables!
```

**Figure 18** head of method Ctrl_StatsHistogram::dataSet

Data flow from a Stats object to its parent Ctrl within Oculus:

Stats C++ object --> `GlobalReporter`--> `ReporterPipe`--> `Ctrl_XXX.dataSet` proc.

## 5.8 Test Design

### Oculus interpreter tests

To test the interpreter there is a file `test_interp.tcl` which tests some Tk commands and the interface to the C++ Stats classes. During the test the file `test_interp.tcl` is opened and used for data input for the Stats classes.

### Oculus functional, stress and regression tests

The Tcl script `test_oculus.tcl` can "simulate the simulator" to test Oculus. It contains procedures for:

- **functional tests** (There are functional tests for each Ctrl class which create a new Ctrl within Oculus and update it.)
- **stress tests** (This test creates a large number of Ctrls and performs a lot of updates.)
- **regression tests** (This test calls all functional tests to verify that everything still works after a change or extension.)

The tests have to be defined in the selection section of the same file. You can run all tests with the communication channel or as a stand-alone version by setting the variable `OclusStandalone` in file `test_oculus.tcl`. If it is 0, Oculus will be tested under real conditions with the connect process. If it is 1, the Oculus files will be sourced and the commands (to send) will just be invoked on the interpreter. This is also a way to test the communication channel.

## 5.9 File Structure

### The Oculus interpreter

- **Oculus/oculus_interp.C**
  The Oculus interpreter C++ program
- **Oculus/Makefile**
  Makefile for oculs_interp.C

### Oculus interpreter tests

- **Oculus/test_interp.tcl**
  This script tests several features of the interpreter
- **Oculus/test_interp.dat**
  This file contains data to be evaluated by the test_interp.tcl script

### Oculus program

- **Oculus/oculus.tcl**
  The main file of the Oculus program. This file sources (includes) the following modules:
- **Oculus/mainwin**
  Procedures for the Oculus main window.
- **Oculus/interface.tcl**
  Interface functions to Pantheon: Those functions will be invoked from outside!
- **Oculus/misc.tcl**
  Miscellaneous procedures for Oculus.
- **Oculus/ctrl.tcl**
  This file contains the Ctrl class hierarchy. Objects of this types are used to control data and Viewer objects. Each object represents an object on the Pantheon side that should be monitored. An object may contain several pointers to Viewer objects where its data is visualized. The Ctrl classes are explained in section Design - The Ctrl hierarchy.
- **Oculus/viewer.tcl**
  This file contains the Viewer hierarchy. Viewers visualize data in a window in some way. The Viewer classes are explained in section Design - The Viewer hierarchy.
- **Pantheon/oculus_share.tcl**
  This is a "shared library" used by Pantheon **and** Oculus. Please be very careful when modifying this file because both applications use it!
- **Oculus/features.tcl**
  These procedures provide additional functionality for graph widgets such as zooming, active legend, etc.
- **Oculus/xftools.tcl**
  Other procedures copied from the XF Tcl package.

**Oculus tests**

- **Oculus/test_oculus.tcl**
  This file contains functional, stress and regression tests for Oculus. It can "simulate the simulator".
- **Oculus/test_interface.tcl**
  Interface procedures to test Oculus with the communication channel or without (as a stand-alone version).

**The connect process**

- **Oculus/connect.tcl**
  Script for the communication channel between Pantheon and Oculus.

**Pantheon extensions**

- **Pantheon/oculus_proc.tcl**
  Service procedures that have to do with the Oculus interface.
- **Pantheon/oculus_share.tcl**
  This is a "shared library" used by Pantheon **and** Oculus. Please be very careful when modifying this file because both applications use it!
- **Lintel/Stats... (all Stats C++ hierarchy files)**
  This files contain the whole Stats hierarchy with classes like Stats (calculates simple statistical values), StatsHistogram (calculates also histogram data, abstract), StatsHistogramUniform (with uniform scaled bins), StatsHistogramLog (with logarithmically scaled bins), etc. Those classes were extended by the ability to send immediate updates and by a Tcl interface for some methods.

**other files:**

- **Oculus/pipe.tcl**
  This script is used by the procedure `CreatePipe` to emulate an internal pipe. This procedure should once be replaced by a Tcl interface to the C command to create a real and faster pipe.

# 6 Implementation & Validation

## 6.1 Source Code reference

Each module has a module header with information about:

- the file name
- RCS (Revision Control System) information
- the contents / description
- the author
- the creation data and the last modification date
- the language of the module
- the package and
- the status

Each function has a comment with:

- the description
- the arguments (if exists)
- the return value (if exists)
- side effects

Each class has a comment with:

- the functionality
- abstract procedures

## 6.2 Test Results

Oculus passed all functional and regression tests. In a stress test 8000 Ctrls were created with altogether 394940 data updates without problems.

I performed one performance test where the simulation was 15 times slower with 10 Ctrls and another with 50 Ctrls where it took 33 times longer as opposed to a simulation without Oculus. This shows that the performance overhead of Oculus is very high, but can be limited by displaying only a small number of Stats. Note, however, that the simulator is not slowed down when Oculus is not used, which was one of the goals. Time was too short to make a good time analysis but the fact that every data update of Stats which should be monitored is reported to Oculus is certainly a reason. Other critical parts might be the connect process and the data lists that are kept within Ctrl_Stats because this code is interpreted.

# 7 Development Reviews

## 7.1 Important Decisions

This section describes alternatives we discussed and important decisions.

Oculus was developed in 3 Rapid Prototyping[1] development cycles. Each design phase was followed by a design review meeting and each implementation phase was followed by a code review meeting.

The first experiments were done with a synchronous communication channel. Pantheon would send data by invoking a Tcl procedure which would wait until the data was processed by Oculus. This could have been a way to pass back results or even commands to invoke on the Pantheon interpreter from Oculus, but it turned out to be to slow.

Another idea was to use a separate back channel from Oculus to Pantheon where data requests or other commands could be submitted. Since we didn't see an absolute need to pass back commands for the first version of Oculus this was considered low priority.

Since Ctrl_Stats need to receive each data point (in case that the user wants to see a stripchart) it did not make sense to transfer all histogram data for histogram viewers on every display update. Instead, the Ctrl_Histogram (derived from Ctrl_Stats) can be intelligent and calculate the histogram itself using the equivalent StatsHistogram C++ object existing on the Pantheon side. The advantages are less data transfer because just data points are submitted and immediate data availability on Oculus. The communication protocol is also simpler. Thus, we decided to link the Stats C++ interface to the Tcl interpreter.

What makes the Oculus language very powerful is the object oriented extension package [incr Tcl]. With the help of this extension the global name space is avoided and the code is much more modular and flexible.

1. see: Development process: Rapid Prototyping on page 17

## 7.2 Enhancement Possibilities

The best way to find out what should be changed or added is to use the simulator with Oculus for a longer period. I couldn’t do that, but anyway here is a list of possible improvements from my point of view:

- **More Viewer and Ctrl classes**
  Add additional Viewer and Ctrl classes for other objects like the rest of the Stats objects or for example a viewer to show a cache map.
- **Back channel from Pantheon to Oculus**
  As Oculus becomes more and more active, a back channel to Pantheon will be useful. There are a lot of possible features which could be added when such a channel is established. For example one could select an object to monitor which was not intended to be monitored before. Or other commands could be invoked on the Pantheon interpreter during simulation.
- **Replace the `CreatePipe` command**
  In file `misc.tcl` there is a command "CreatePipe" that returns an emulated internal pipe by invoking the pipe.tcl script which simply gets and puts strings from `stdin` to `stdout`. This command is used to read the Stats object data from the `GlobalReporter` within Oculus. However this is no real Unix pipe and it might be much faster replace the CreatePipe procedure by a Tcl interface to the real C pipe command. Unfortunately time was too short to implement and test this.
- **Stop/continue button**
  A nice feature would be a button to stop and continue the simulation.
- **New menu commands**
  Additional menu commands might be comfortable, for example a menu item to remove all viewer windows.

## 7.3 How to Extend Oculus

If you want to implement a new Viewer, read section: The Viewer hierarchy on page 30.

If you want to create a new Ctrl class, read section: The Ctrl hierarchy on page 28

To integrate new libraries to the Oculus interpreter or to exchange packages read section: The Oculus interpreter on page 28.

## 7.4 Conclusions

The design meets all of the requirements, and the implementation is easy to extend. Due to the object oriented architecture it was possible to add some fancy features very quickly, like saving and loading windows positions or having a comfortable toolbar for each Viewer class. I could reuse a lot of code from the Stats / Reporter classes and some code from demonstration files. I think that Oculus is a good basis for future development.

The Oculus project was also very valuable for me, because I learned a lot about storage system techniques as well as simulation techniques. I enjoyed being a member of a sophisticated research team at HP Laboratories.

It turned out that the Rapid Prototyping software development process is very powerful especially when you don’t have an exact idea of the result or if design changes are likely to happen during development.

I am happy that the Oculus development was successful.

## Acknowledgments

I have designed and implemented Oculus at Hewlett-Packard Laboratories, Palo Alto. I would like to thank Professor Wiese for his support and John Wilkes for his efforts to make my visit possible. I am also grateful to the members of my team: Richard Golding, John Wilkes, Carl Staelin, and my advisor Tim Sullivan for being motivating and enthusiastic.

## References

[Goldberg83] Adele Goldberg and David Robson. Smalltalk-80: the language and its implementation. Addison-Wesley, Reading, Mass, May 1983.

[Golding94] Richard Golding, Carl Staelin, Tim Sullivan, and John Wilkes. "Tcl cures 98.3% of all known simulation configuration problems" claims astonished researcher! Proceedings of Tcl/Tk Workshop, New Orleans, LA, June 1994, Available as Technical report HPL-CCD-94-11, Concurrent Computing Department, Hewlett-Packard Laboratories, Palo Alto, CA.

[Golding95a] Richard Golding, Peter Bosch, Carl Staelin, Tim Sullivan, and John Wilkes. Idleness is not sloth. *Conference Proceedings of Winter USENIX 1995 Technical Conference* (New Orleans, LA), pages 201–12. Usenix Association, Berkeley, CA, 16–20 January 1995.

[Golding95b] Richard Golding, Uwe Aicheler, Tim Sullivan, and John Wilkes. The Raphael threads library.

[Wilkes92] John Wilkes. DataMesh research project, phase 1. *USENIX Workshop on File Systems* (Ann Arbor, MI), pages 63–9, 21–22 May 1992.

[Wilkes95] John Wilkes, Richard Golding, Carl Staelin, and Tim Sullivan. The HP AutoRAID hierarchical storage system technology. *Proc 15th ACM Symposium on Operating Systems Principles* (Copper Mountain Resort, CO). Published as *Operating Systems Review*, 29(4), 3–6 December 1995. An earlier version was published as HP Labs Technical Report HPL–CCD–95–7 (April 1995).

## Author information

Uwe Aicheler is a Computer Science student of Esslingen University (Fachhochschule fuer Technik Esslingen). He has spent most of the past years developing a visual mathematical application called Math Designer for Windows. He has also worked for the Software Quality Engineering group of HP in Boeblingen in winter 1994/95.

Uwe Aicheler can be reached by electronic mail as "644888@rz.fht-esslingen.de".