ParCompMark User Manual

Parallel Compositing Benchmark Framework
version 0.6, May, 2007

BME-IT^2 ParCompMark Dev. Team

http://amon.ik.bme.hu/parcompmark/

This manual is for ParCompMark (version 0.6).

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction and Objectives

1.1 About This Document

This document gives an overview of the entire parallel compositing benchmark framework called *ParCompMark* (Parallel Compositing Benchmark Framework) developed by BME[1]. The document provides information on the framework architecture and its functions. This framework allows the simulation of various CPU and GPU tasks, as well as rendering and compositing modes for different software scenarios. It eases setting up the benchmark scenario by a scripting language and generates rich XML output containing all related information of the execution.

1.1.1 Structure of This Document

In the *first chapter* the background of parallel rendering and parallel compositing is presented. The *second chapter* is about the usage of this tool. The *third chapter* explains a real-world example in detail. In the *last chapter* some relevant design and implementation aspects are detailed.

As additional material this document includes five appendices:

- Appendix A is a short squirrel language reference and pointers to the detailed references. Squirrel scripting language is easy to learn and use.
- Appendix B summarizes rendering engine methods. This is a simplified version of the OpenGL API. The functions of this API can be used in the Squirrel scripts that control the operation of an arbitrary benchmark job.
- Appendix C presents the experimental results with the "brute force" triangle rendering benchmark. This appendix contains a set of plots of the measured data on three different SVA clusters.
- Appendix D is a code tutorial how to write a simple renderer plugin that renders a triangle in every frame.

[1] http://www.bme.hu

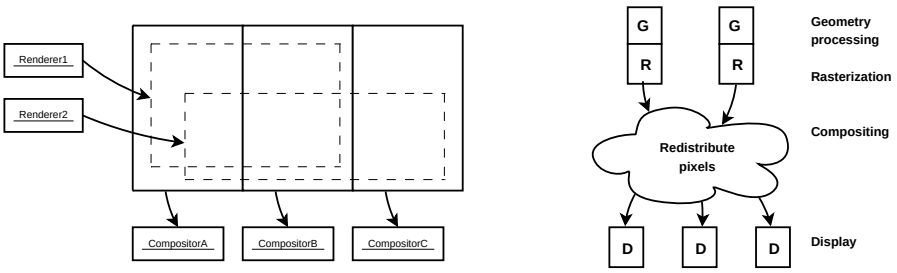


Figure 1.1: Sort-last parallel rendering approach with both screen-space and object-space parallelization. Geometry processing and rasterization are done in the same rendering pipeline, then the renderers transmit the pixels over an interconnection network to compositing processors, which calculate the corresponding segment of the final output [9].

- Appendix E discusses sample post processing scripts that use XSLT[2] and a statistical tool called "R"[3] . Certainly, this is not the "only-solution" to generate expressive results from the gathered XML format results. However, the plots of Appendix C were generated in this way.
- Appendix F contains a look-up table about the ParaComp calls in the code of ParCompMark.

1.2 Parallel Rendering and Parallel Compositing

There are several domains of computer graphics in which so large amount of data has to be handled or so complex rendering model has to be rendered that interactive frame rates cannot be obtained with a single graphics hardware. Therefore, parallelization of the rendering tasks and decomposition of the data are necessary. Several parallel implementations of rendering algorithms have been proposed to overcome this limitation. In order to have quantitative feedback about the operation of parallel rendering systems, test-bench applications are required.

Parallel rendering methods are usually classified based on what kind of data is distributed and where the sorting in graphics pipeline occurs: [9]

1. sort-first: in the beginning of the pipeline,
2. sort-middle: between the geometry processing and the rasterization step,
3. sort-last: at the end of the pipeline, after the rasterization.

The location of the sorting fundamentally determines the required hardware architecture and communication infrastructure.

The *sort-first* approach distributes the primitives at the beginning of the pipeline to the rendering nodes [10] [5]. In case of a *sort-middle* parallelization

[2]XSL Transformations (XSLT): `http://www.w3.org/TR/xslt`
[3]The R Project for Statistical Computing: `http://www.r-project.org/`

the primitives are transformed into screen coordinates, they are clipped, and distributed for rasterization to the corresponding display device [6]. *Sort-last* transmits the primitives through local rendering pipelines and defers sorting after rasterization. In this case, one fraction of processors (*renderers*) are assigned to determined subsets of the primitives and to pixel sets of the final output image, usually to rectangular areas. The other type of processors (*compositors*) are assigned to subsets of pixels in the output image. Note that the partitioning of the pixels does not have to be identical in the two cases (see Figure 1.1).

Another classification scheme is based on the type of entities that are simultaneously processed. Single-threaded software renderers take graphics primitives one after another and the pixels corresponding to these primitives are also processed sequentially. In contrast, recent graphics cards have multiple graphics pipelines, therefore more vertices and pixels can be processed at the same time. This is called *pixel-parallel rendering*. Pixel-based parallelization can also be performed when multiple graphics cards are used for creating tiles of the overall output image and the rendering queue is branched into multiple pipes (*screen-space decomposition*). On the other hand, when the data is divided in an initialization step, multiple subsets of graphics primitives can be processed at the same time. This is called *object-parallel rendering* or *object-space decomposition*.

Object-parallel rendering needs the combination of the subsets of pixels corresponding to different objects, which is called *image compositing*. This is a simple procedure, which involves processing of pixel attributes. Originally alpha colors were introduced as a pixel coverage model for compositing digital images [12]. Besides alpha-based compositing, spatial covering can be also carried out comparing depth values, when a subset of the Z-buffer is transmitted with the color values [2]. These per-pixel calculations are to be achieved for all image elements, therefore compositing can be a bottleneck of the whole rendering system and may make it unsuitable for interactive applications. However, when the compositing is also done in parallel, *interactive compositing* is possible. There are several algorithms providing parallel image compositing on multiprocessor architectures including *direct send* [4, 11], *parallel pipeline* [7], and *binary swap* [8].

The main demands for interactive parallel visualization systems are the auspicious *data scalability* and the *performance scalability*. Data scalability means that adding more computing nodes to the system should enable larger amount of data to render with similar performance, while performance scalability means that the level of performance should increase proportionally to the computing power. Certainly the performance is a compound of several metrics like *frame rate* and *latency*. In this case the frame rate means frequency in which image frames or frame-segments are generated on a host. The generation can mean both rendering and compositing. However, in distributed environment the frame start and frame end times for collective frames are not obviously well-defined. The latency here means the time elapsed between the request to get an image result and when it is actually received. The importance of these factors are often application specific.

Nowadays, there are two significant trends for interactive parallel rendering which satisfies these demands. One of them based on the sort-first approach *virtualizes multiple graphics cards* and provides a *single conceptual graphics pipeline*. In this way the incoming primitives are redirected to the corresponding

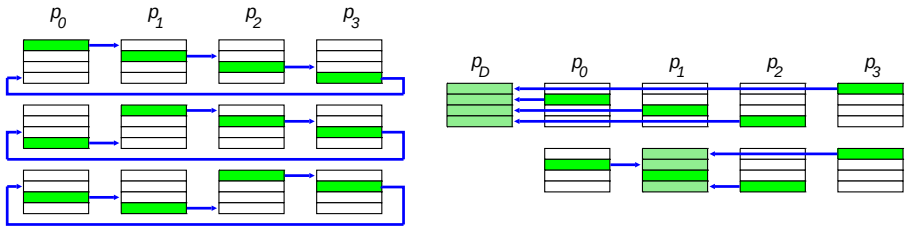


Figure 1.2: Parallel pipeline algorithm for sort-last approach on distributed memory architectures. Left: image area transfer for four compositing processes performed in $N-1$ (number of frame areas) steps. Right: collecting final data for an external or an internal process in one step [7].

rendering node right after their definitions. The benefit of this approach is that applications with originally non-parallel design can be executed in a distributed environment without source code modification or moreover without recompilation. The other solution uses the sort-last method and does not try to modify the local graphics pipeline of the rendering hosts to get better performance, but it operates with object-space data distribution and image compositing. The drawback of this method is that larger modifications or redesign are required for existing applications, but the advantage is that the load balance is more predictable and designable.

The ParaComp library discussed in the next section implements the parallel pipeline compositing algorithm. This algorithm consists of two parts detailed in Figure 1.2. The pseudo code of the method is illustrated in Listing 1.1. The images to be composited are divided into N frame areas, where N is the number of the compositing processes. In the first part of the algorithm these areas flow around through each node in $N-1$ steps, each consisting of a compositing and a communication stage. After $N-1$ steps each processor will have a fully composited portion of the final frame. Then the areas are collected for an external display node or for an internal node in the second part in one step. The clear benefit of this compositing scheme is that the amount of data transferred on the network is independent of the number of compositing processes.

1.3 ParaComp: Parallel Compositing Library

The *Parallel Compositing Library* (ParaComp) was developed by HP and Computational Engineering International (CEI). The addition of this library simplifies the development and use of parallel applications on graphics clusters and allows high performance computing users to interactively render and visualize huge data sets. The HP Parallel Compositing Library does for graphics clusters what MPI did for compute clusters. It enables users to take advantage of the inherent performance scalability of clusters with network-based pixel compositing.

The ParaComp library was designed in order to create a single image from a collection of partial images generated by multiple sources. The sources can be located on one or more machines and can be threads of execution on a single

```
global p_i, N

function parallel_pipeline(i, target, frame) {
  def f_k ← frame.sub_image(0,k*frame.height/N,
        frame.width, (k+1)*frame.height/N-1)
   p_next ← p_{i+1} mod N
   p_prev ← p_{i-1} mod N

  for j ← 0, .., N-2
     f_send ← f_{i-j} mod N
     f_recv ← f_{i-j-1} mod N
     send    f_send →p_next
     receive f_recv← p_prev
     compose f_recvwith f_send

  if i ≠ target
    send f_i  → p_target
  else
    for j ← 0, .., N-1 (j ≠ i)
      receive f_j ←  p_j
}
```

Listing 1.1: Pseudo-code of the parallel pipeline algorithm on distributed memory architectures for process p_i. The variables are coded as follows: $p_0 \dots p_{N-1}$ are the compositing processors, $f_0 \dots f_{N-1}$ are the image framelets, and *target* is the index of the target process.

machine. The library was designed to hide the network layer from the caller and provide a graphics pixel abstraction. For more information see [3].

1.4 ParCompMark

ParCompMark can measure the performance of *sort-last parallel rendering techniques*. The current implementation created a test-bench for the Parallel Compositing Library. This framework allows simulation of various CPU and GPU jobs, rendering and compositing modes, modelling different application scenarios. It eases setting up the benchmark scenario with a scripting language.

The goal of ParCompMark is to gather statistics about the distributed execution and to analyze the performance. This analysis provides a feedback for both parallel rendering and visualization application development. ParCompMark supplies a plugin interface for creating and analyzing prototypes of distributed visualization applications.

The benchmark framework allows the definition and execution of complex parallel rendering tasks in a distributed environment while collecting measurement data from every participating process. The main requirements for such a benchmark program are:

- Provide means for defining the target cluster structure and the CPU/GPU resources to use in a benchmark.
- Define the roles of the participating resources such as rendering and/or compositing.

- Specify the characteristics of the rendering task as shading model, type and number of primitives, specific algorithm, size of the final image, compositing mode (depth, alpha, etc).
- Allow various measurements such as frame rate, latency, and network communication; collect measurement data in a format that is suitable for further processing (e.g. scientific visualization tools).
- Provide means to simulate workload in real environments by specifying multiple processes/threads, extra load on the elements of the cluster, etc.
- Facilitate the definition of time-varying and batch like scenarios for benchmark test suites.

Based on the requirements listed above, a general benchmarking framework was designed and implemented which allows the users to concentrate solely on specifying the cluster parameters, rendering characteristics, scenarios without the need to understand the details behind the operation of the framework. The system exposes its services through an integrated scripting engine which provides all the necessary tools for defining and running the benchmarks.

Chapter 2

Usage of ParCompMark

This chapter provides a detailed description of the usage of ParCompMark. Section 2.1 contains *installation instructions*. The *execution* is presented in Section 2.2. Section 2.3 explains the *scripting fundamentals*. In Section 2.4 the syntax of *user commands* can be found. In Section 2.5 basic *plugin writing hints* are detailed. Section 2.6 presents the *structure of the output document*, which actually contains the results of a benchmark execution.

2.1 Installation

Both prebuilt binary packages and source tarballs can be found on the web site[1] of the project.

2.1.1 RPM Package

ParCompMark can be installed from a prebuilt RPM[2] package in the usual way:

```
rpm -i parcompmark.rpm
```

2.1.2 Building From Sources

ParCompMark can be built from sources using

```
$ ./configure --with-inc-dir=<additional include directory> \
 --with-lib-dir=<additional library directory>
$ cd src
$ make
$ sudo make install
```

On 64-bit HP XC platform the additional path values are the following:

- `<additional include directory>` = `/opt/paracomp/include`

[1] `http://amon.ik.bme.hu/parcompmark`
[2] Red Hat Package Manager

- `<additional library directory>` = `/opt/paracomp/lib64`

If the unit tests are also needed to be compiled, the `cd src` command should be skipped in the previous command list, so thus both the application (directory `src`) and the unit tests (directory `test`) are compiled.

2.1.3 Library dependencies

The following libraries are needed by ParCompMark:

libparacomp : Hewlett Packard implementation of the *Parallel Compositing API*

libreadline : GNU Readline library

libtermcap : GNU termcap library

libhistory : GNU Readline library history extension

libgl : library implementing OpenGL API

libglu : OpenGL Utility Library

libglut : OpenGL Utility Toolkit (GLUT)

libsquirrel : Squirrel language library

libsqstdlib : Standard Libraries implementation

libsqplus : Squirrel–C++ binding library

libdl : Dynamic linking library

libm : C Math Library Functions

There are prebuilt packages for AMD64 HP XC platform on the web site of the project for the Squirrel libraries (libsquirrel, libsqstdlib, libsqplus).

2.2 Execution

After installing ParCompMark, instances can be started on target hosts with the `parcompmark` command. ParCompMark can be started in two different modes:

commander mode instance controls the entire running of the benchmark test and provides the user interface.

soldier mode instance has no user interface, the operation is fully automatic.

There must be *one commander* and *any number of soldier instances* working together. The commander mode instance is designed for head node execution the soldier nodes should run on the render nodes of the cluster.

The ParCompMark instances can be executed both manually using the `parcompmark` and with the aid of special startup script designed for the HP XC platform.

2.2.1 Manual Execution

ParCompMark can be started manually, logging into to all nodes of the cluster using ssh. The syntax of `parcompmark` command is the following:

```
parcompmark [-h] [-v] [-c] [-o <output file>]
 [ -l <low-level script> | -d <dynamic script> |
   -s <scenario script> ]
  [-H <host count>] [-u none|console]
```

where

`-h` show help

`-v` show version of ParCompMark

`-c` indicates commander mode. If `-c` is missing, the instance will be started in soldier mode.

`-l <low-level script>` sets low-level script to execute (see Section 2.3.2 for details).

`-d <dynamic script>` sets dynamic script to execute (see Section 2.3.3 for details).

`-s <scenario script>` sets scenario script to execute (see Section 2.3.4 for details).

`-o <output file>` sets the output file, where the gathered statistics will be stored. When no output is specified the results will be written to the standard output (see Section 2.6 for details).

`-H <host count>` is a useful parameter. Theoretically the commander mode instance can serve any number of soldiers. The startup and login processes of soldier nodes usually take a few seconds but sometimes more. It depends on the current load of the hosts, on the other tasks to do before executing the instance, e.g. starting X server, SLURM[3] job, etc. So thus no exact timeout value exists for the commander to wait for. When setting `-H` flag, the command terminal will be immediately shown when the expected number of login messages has been received.

`-u console` gets the commander mode instance to start with console interface by default. However, in case of batch operation it can be useful to set automatic operation for the commander too. Setting `-u none` will do this.

Here is an example for starting ParCompMark on a cluster containing three hosts (n16, n13, and n14):

```
nohup ssh n13 "export DISPLAY=:0.0; parcompmark" &; \
nohup ssh n14 "export DISPLAY=:0.0; parcompmark" &; \
ssh n16 "export DISPLAY=:0.0; parcompmark -c"
```

[3] A Highly Scalable Resource Manager: `http://www.llnl.gov/linux/slurm/`

2.2.2 Startup Scripts

There are two SLURM startup scripts for easier use of ParCompMark. One is for *HP Remote Graphics Software* and the other one is for *TurboVNC+VirtualGL* combo. Both scripts can be invoked with the following syntax:

```
<command> [-h] [-u] [-v] [-d] [-w <node-list>] [-p <partition>]
         [-g <geometry>] [-l] [-a <parcompmark parameters>]
```

where

`<command>` can be `parcompmark.sh` or `parcompmarkvnc.sh`. The VNC version is designed for VirtualGL+TurboVNC combo, it contains `rrlaunch` calls to redirect the OpenGL calls of the application.

`-h`, `-u`, `-v` show help, usage, and version, respectively.

`-d` turns on script debugging.

`-w` passes a list of nodes to run the program on, e.g. `n[12-14]` or `n[12,14,13]`.

`-p` requests resources from the given SLURM partition (see SLURM documentation for details).

`-g` specifies width×height. The default value for geometry is 800×600, thus when a higher resolution image is needed, it has to be overridden.

`-l` orders the master piece of the SLURM job to run on this node (see SLURM documentation for details).

`-a` indicates that the trailing parameters have to be passed for `parcompmark` binary. The only exceptions are the `-H` and `-c` parameters, which should not be used in this way. These parameters are implicitly set by the startup script.

Here is an example for starting ParCompMark using VNC on an HP XC cluster containing three hosts (n12, n13, and n14) with resolution 1024×768 using SLURM, and with the name of the output xml file `benchtest1.xml`:

```
parcompmarkvnc.sh -w n[12-14] -l -g 1024x768 -a -o benchtest1.xml
```

2.3 Scripting

This section describes the fundamental scripting concept of ParCompMark, which is responsible for the flexible benchmark scenario build-up, and is based on the Squirrel scripting engine. The basic idea is that native application code can execute Squirrel scripts and the Squirrel scripts can call back the native application code. This scheme is favorable because of two reasons. First, the application does not have to be recompiled when a new benchmark test is written. Furthermore, the parallelly running instances do not have to be restarted, the script can be reloaded on the fly. On the other hand, Squirrel scripts are strings for the native application, thus they can be transferred through the network easily.

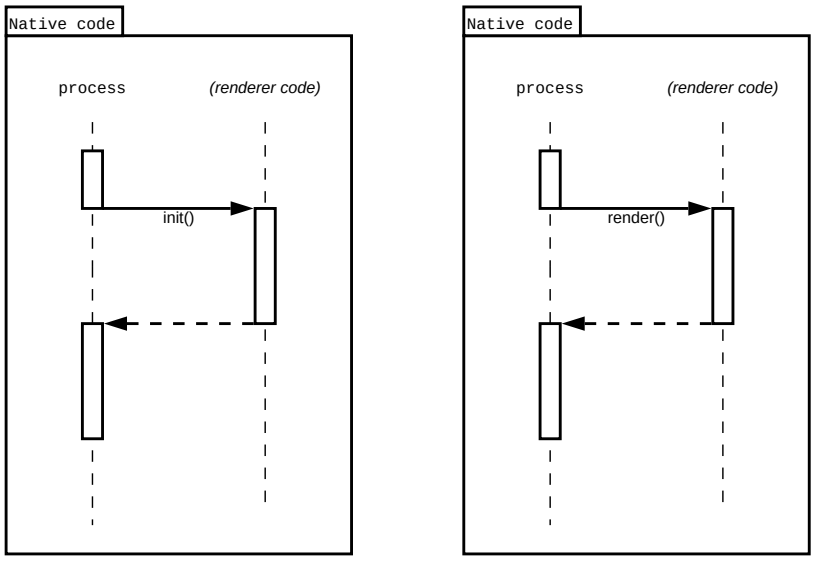
2.3.1 Introduction to ParCompMark Scripting

In this part of the section the ParCompMark scripting concept is summarized briefly. In the next parts scripting levels are detailed.

Classic "Wired-C" Scheme

Without ParCompMark one should write specific C/C++ code for each benchmark test cases. Although, it is possible to create parameterizable renderer classes, it is not the most flexible solution for the problem since the whole benchmark application has to be recompiled on every single modification.

A renderer process usually has two methods which are called by the benchmarking framework: (1) the *initialization* method, which is called once at the beginning of the benchmark execution, and (2) the *operation/running method* which is called at every frame (see Figure 2.1).



(a) "Wired C" initialization (b) "Wired C" rendering

Figure 2.1: "Wired C" scheme: initialization and running methods

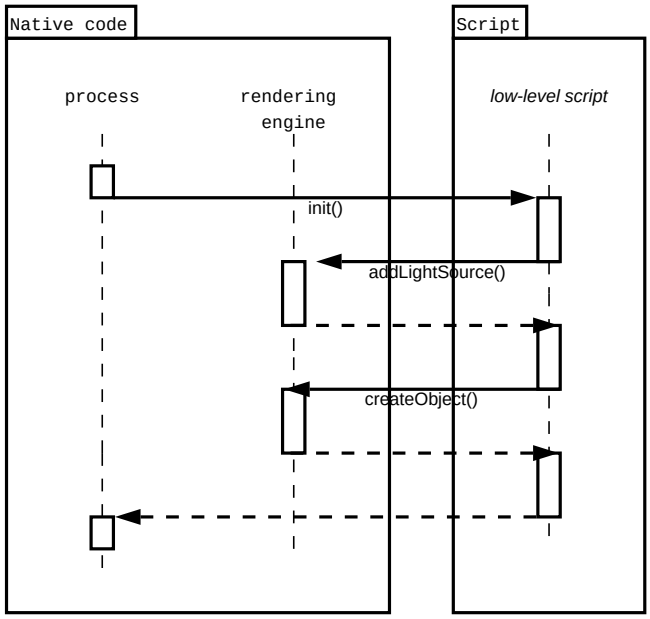
Script-Controlled Scheme

When all of the rendering specific functionality (i.e. simplified OpenGL calls) can be encapsulated in a class, then the control of the specific benchmark test case can be described in a script file (e.g. loops, parameter settings, etc.), where the script can call back the native code during its execution.

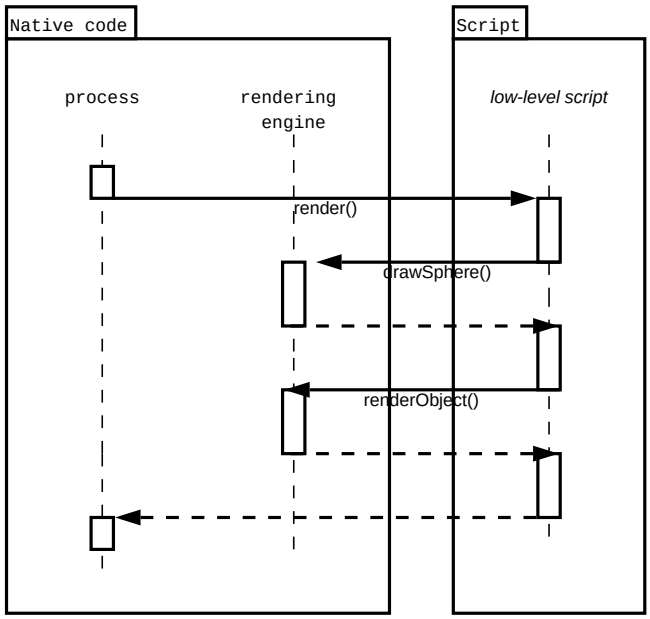
This practically means that the application loads (or generates, see later) the script into a string variable and starts a script virtual machine. This virtual machine compiles the script to bytecode representation which can be executed efficiently when the script is called.

In ParCompMark this control script file is called *"low-level"* script, since it controls the execution of a benchmark at a very low level.

Using Squirrel the interaction between the native code and the script code is very flexible. The native code calls the functions of the script. During its execution the script can create objects defined in the native code and call their methods. Thus, the C++ objects can be reached from both native code and also from the script (see Figure 2.2).



(a) Initialization using low-level script



(b) Rendering using low-level script

Figure 2.2: Low-level script scheme

Renderer Plugin Scheme

The script-conrolled scheme seems to be good for simple benchmark cases. However, complicated rendering jobs cannot be expressed using a predefined set of rendering operations. Moreover, usuallay it is not worth implementing a complicated visualization application in a script. Therefore, ParCompMark tries to give a balanced solution between the flexibility and the usability by introducing the *rendering plugins*.

The rendering plugins are dynamic libraries that can be loaded in run time. The ParCompMark low-level scripts can initiate loading these libraries and making them create renderer objects. These custom render objects have a predefined interface through which both the native code and the script code can call them. This interface exactly means event handlers (onResize, onRender, etc.) and parameter setter methods (e.g. screen-space or object-space decomposition). See Figure 2.3).

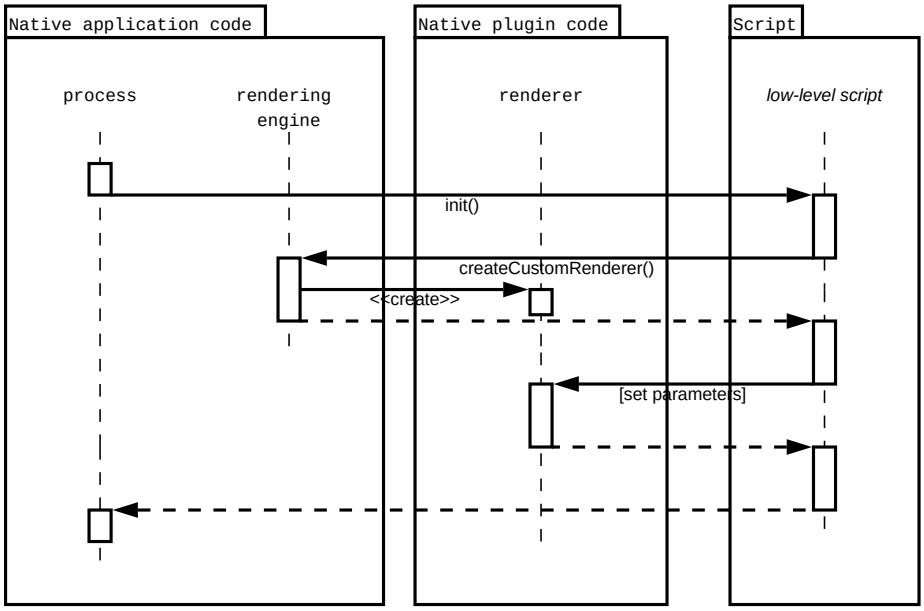
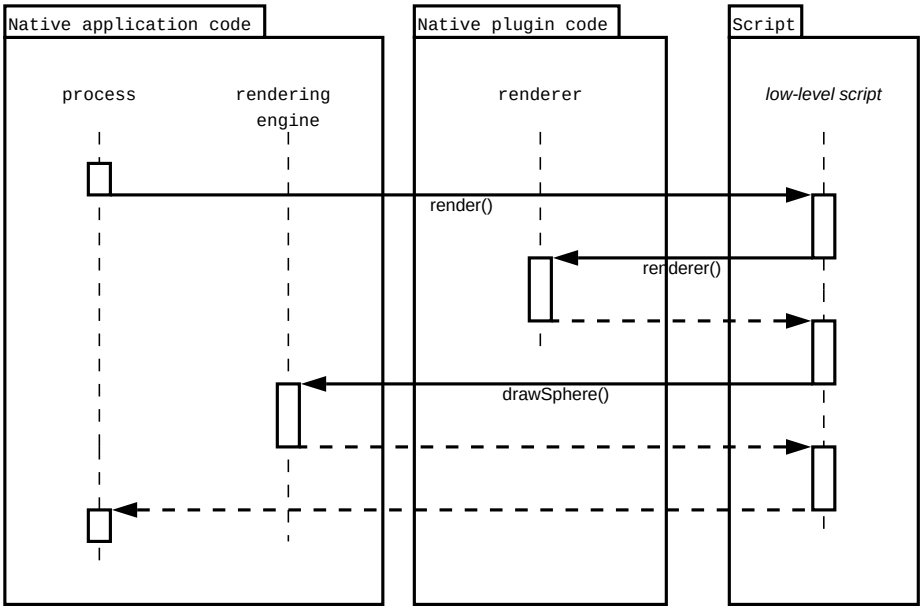
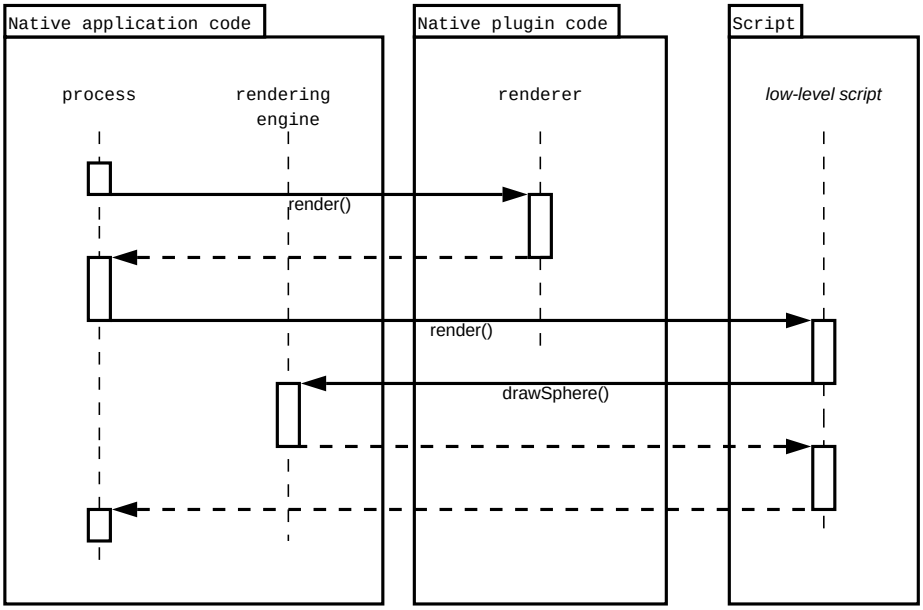


Figure 2.3: Renderer plugin scheme (initialization)

A rendering plugin can be used in so-called *no-autorender* mode, when the low-level script has to call explicitly the render method of the renderer, and in *autorender* mode when the renderers are called in a predefined order by the framework before the execution of the low-level script. When using *autorender* mode the rendering process of the low-level script can be empty, therefore *no additional scripting overhead occurs* (see Figure 2.4).



(a) Rendering using a plugin in *no-auto-render mode*



(b) Rendering using a plugin in *auto-render mode*

Figure 2.4: Renderer plugin scheme (rendering)

Dynamic Script Concept

Despite of its flexibility, writing low-level scripts by hand is cumbersome when using large number of nodes. Therefore, a higher level script is introduced: the cluster-independent *dynamic script*. The input of this script is the *cluster description* and the output is the generated low-level script. Moreover, the dynamic script provides a well-defined interface to the framework for setting customizations parameters (e.g. what to render, number of rendering nodes, sizes of the output, etc) and a function that can create a low-level script that satisfies the demands (see Figure 2.5).

The dynamic scripts are also implemented in Squirrel language. This is favorable because a Squirrel code can efficiently create the objects of the low-level script dynamically. When the low-level script is assembled (ParCompMark calls this dynamic script compilation) it is serialized into a long string which is returned to the native code.

The commander instance transfers this string throught the network to the soldier instances. They also start virtual machines and deserialize the low-level script string into Squirrel language object.

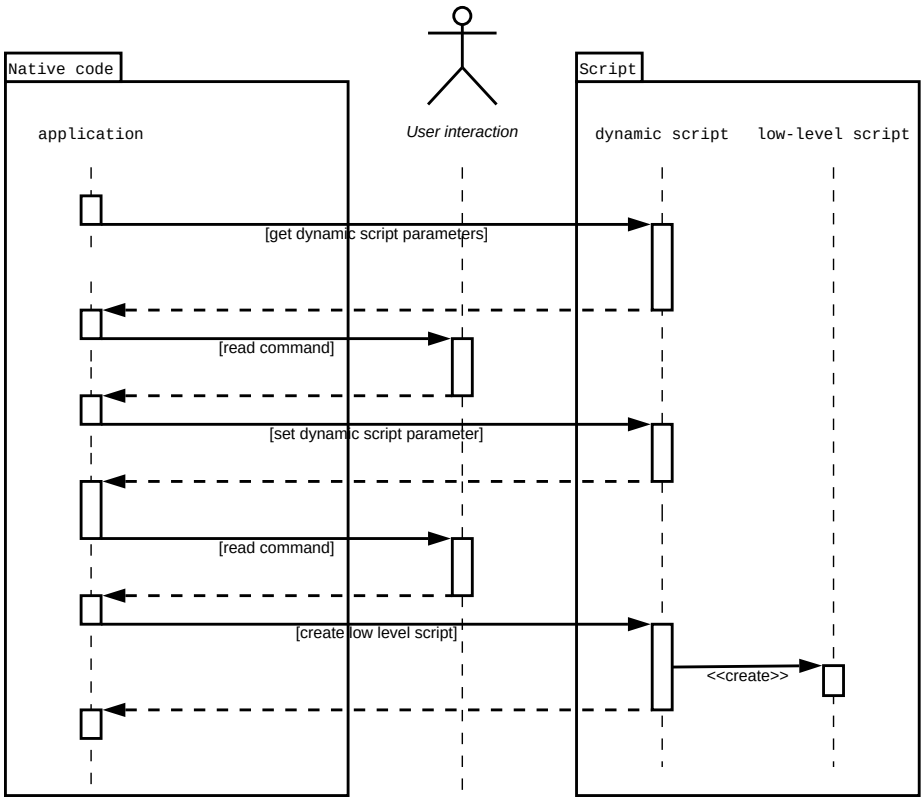


Figure 2.5: Creating low-level script using a dynamic one

Scenario Script Concept

Dynamic scripts are good for doing investigations with user interaction. However, dynamic scripts are not effective when large number of automated benchmark cases have to be executed Therefore, an even higher scripting level is defined above the dynamic script: the level of *scenario script*.

Scenario scripts are also procedural codes generating a command list which acts as if the user would typed them into the command terminal. It comes natural, that this script can also dynamically generate the command list.

This script can load one or more dynamic scripts during its execution and creates test cases based on the actual cluster settings (e.g. number of nodes, amount of graphics memory, etc.) and executes these tests. The concatenated results are gathered in a single XML file with the specified parameters for each test case (see Figure 2.6).

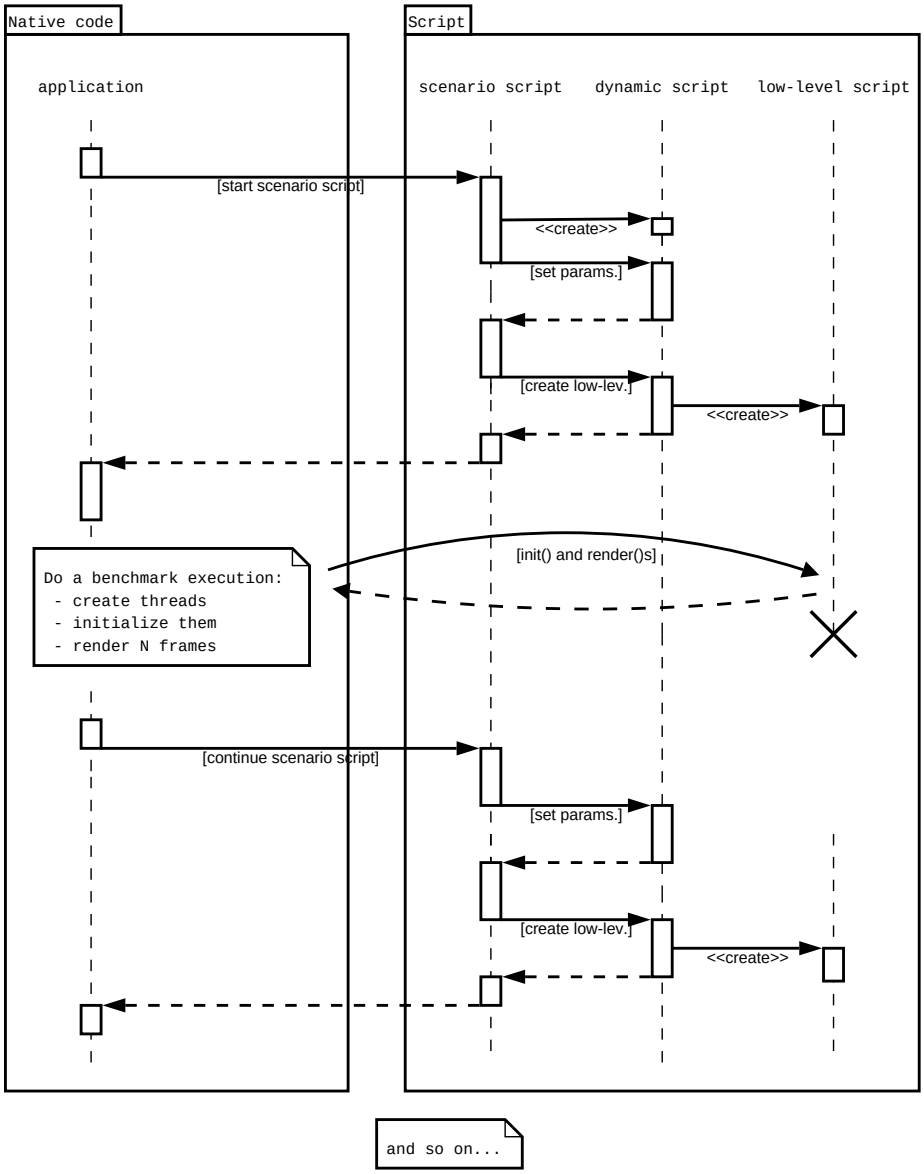


Figure 2.6: Scenario script: creating batched benchmark cases

2.3.2 Level 1: Low-Level Scripts

The declarative *low-level script* contains all information that the current benchmark test needs about the hosts in a given cluster: the list of nodes to create, the processes for each node, and the compositing contexts. The hosts initialize themselves at the benchmark execution time using this script.

Nomenclature of Low-Level Scripts

To understand the structure of low-level scripts and the operation of a host that executes a low-level script the following naming conventions are neccessary to know:

host is a computer within the cluster. A host may create any number of nodes.

node is a logical entity. It contains and encapsulates processes and buffers. The processes of a node can only operate on the buffers of the node. Therefore, the nodes are the logical entities that can be moved from to another host.

process is the atomic element of the execution. A process can do either *rendering* or *compositing* job. Each process must refer exactly one buffer of their node. Each process belongs to exactly one compositing context.

buffer is a system memory buffer. The results of a rendering process is read back to a buffer and the composited pixels are also stored in a buffer.

context is a compositing context having a list of processes that do either rendering or compositing jobs using the global framebuffer of the context. See [3] for more details on Parallel Compositing API.

Structure of Low-Level Scripts

The *low-level script* contains the name of the benchmark and two lists (see Listing 2.1, for Squirrel syntax see Appendix A):

- the *host list* contains the hosts affected in the current benchmark.
- the *context list* contains the parallel compositing contexts.

```
// Low-level script
{
  name = "I'm a script" // Name of the low-level script

  // This is the array of the hosts allocated for benchmarking
  hosts= [
    // . . .
  ]

  // The other part of the low-level script is the context array
  contexts= [
    // . . .
  ]
}
```

Listing 2.1: Structure of the low-level script

Structure of Low-Level Scripts – Host

A *host* in the host list contains the network name of the host (it should be unique) and a *list of the nodes*, see Listing 2.2.

```
// A host
{
  name = "n12" // The (network) name of the host

  // The array of the nodes on "n12"
  nodes= [
    // . . .
  ]
}
```

Listing 2.2: Structure of a host in the low-level script

Structure of Low-Level Scripts – Node

A *node* in the node list contains its logical name (it has to be unique within the host) and two lists for the buffers and the processes (see Listing 2.3):

```
// A node
{
  name = "I'm a node" // The name of the node

  // The array of the buffers on this node
  buffers= [
    // . . .
  ]

  // The array of the processes on this node
  processes= [
    // . . .
  ]
} // end of the node
```

Listing 2.3: Structure of a node in the low-level script

Structure of Low-Level Scripts – Buffer

A *buffer* is a continous segment in the system memory which is logically mapped to the global frame of the context which belongs to the process that refers the buffer. The buffer section of the low-level script describes the parameters of the buffer (see Listing 2.4):

name is the logical name of the buffer, it has to be unique within the node. The process refer to the buffer with this name.

left is the *horizontal offset* in the global buffer of the context.

top is the *vertical offset* in the global buffer of the context.

width is the *horizontal size* of the buffer.

height is the *vertical size* of the buffer.

depth is the type of the depth buffer used with this buffer.

```
// A buffer
{
  name = "Buffer #0" // The name of the buffer

  left = 256
  top = 0
  width = 512
  height = 512

  depth = PC_PF_Z32I // Depth format
}
```

Listing 2.4: Structure of a buffer in the low-level script

Structure of Low-Level Scripts – Process

A *process* section in the process list describes the following attributes:

name is the name of the process (must be unique within the cluster).

type indicates that this process does *rendering* or *compositing*.

buffer the name of the buffer used by this process.

initProc *(optional)* the scriptlet to be executed at the start of the benchmark.

runningProc *(optional)* the scriptlet to be executed at every frame having two parameters used in animations:

- **time** elapsed since the start of the benchmark,
- **frame** the actual frame id (it starts with zero and increments at every frame).

sortOrder *(optional)* is an order used in order dependent compositing operations like alpha compositing.

stopID *(optional)* is the id of the frame when the process stops. Practically it means that **stopID** frames will be processed.

operate *(optional)* indicates that the current process operates or not. The default value is true. Setting to false disables the operation of the **initProc** and the **runningProc** functions.

display *(optional)* indicates that the results of a compositing process is drawn or not. The default value is true.

statistics *(optional)* indicates that gathering statistics is performed for this process. By default, it is true.

exportFrameStep *(optional)* is the stepping between the exported frames. The *exported frames* are written to image files in the current working directory. This feature eases creating movies of the benchmark executions. The default value is zero which means no exporting.

frameFilenamePattern *(optional)* is the filename pattern for frame exporting. The default value is `%06d.png`.

Listing 2.5 presents a sample script. For the reference of the rendering engine (RE) see Appendix B.

```
// A process
{
  name = "Process #0" // The name of the process
  type = Process.RENDER // The type of the process
  buffer = "Buffer #0" // The name of the used buffer
  stopID = 500 // Render 500 frames
  exportFrameStep = 10 // Save every 10th frame
  frameFilenamePattern = "frame-%02d.jpg" // to a jpg file

  initProc = @"function initProc(){
    // Create display list
    displayList<-RE.createDisplayList();

    // Do some rendering
    RE.setColor(1.0, 0.0, 0.0, 1.0);
    RE.drawTriangle();

    // Finish the display list
    RE.finishDisplayList();
  }"

  // Rendering procedure
  runningProc = @"function runningProc(time, frame) {
    time = time.tofloat();

    // Setup rendering
    RE.perspective(90.0, 0.01, 100.0);
    RE.setCameraPosition(1.0, 1.0, 1.0);
    RE.setCameraTarget(0.0, 0.0, 0.0);
    RE.setCameraUpVector(0.0, 0.0, 1.0);
    RE.rotate(time * 10.0, 0.0, 0.0, 1.0);
    RE.translate(0.0, 0.0, 0.1);

    // Execute the display list
    RE.executeDisplayList(displayList);
  }" // end of runningProc
} // end of the process
```

Listing 2.5: Structure of a process in the low-level script

Structure of Low-Level Scripts – Context

A compositing *context* has the following attributes:

compositeType Specifies how the pixels from multiple hosts are composited. Specific values of this property include:

PC_COMP_DEPTH means that the pixel closest to the eye masks the other, more distant pixel.

PC_COMP_ALPHA_SORT indicates that the color of the pixel is a blending of the colors of the 2 pixels. The blending is dictated by the transparency (alpha value) of the 2 pixels.

frameWidth is the overall width of the frame created by the context in pixels.

frameHeight is the overall height of the frame created by the context in pixels.

colourFormat and **depthFormat** specifies the type of the pixels in the frame. Each pixel has an encoding for the color of that pixel and any additional information that may be needed to composite the pixels, such as depth or transparency. See [3] for details.

compressionHint specifies the data compression level (see [3] for details).

retainOutputCount relaxes the *output availability constraint* and gives access to the output for the number of frames designated by this attribute. By default, the output of a frame is only available until the start of the next frame (exactly between pcFrameEnd and the following pcFrameBegin, See [3] for details.)

volatileFrameletCount relaxes the *input copy constraint*, which potentially allows the library to be more efficient. By default, framelets are copied by the Library when passed as a parameter to a function call.

outputDepth indicates that the compositing process needs the depth values for the composited image.

networkID specifies the network to use to transfer images between the hosts when they run on different systems. See [3] for details.

```
// A context
{
  glFrameletEXT = false // Whether to use framelet extension or not

  frameWidth = 512 // Sizes of the global frame
  frameHeight = 512

  colourFormat = PC_PF_BGRA8 // Pixel format
  depthFormat = PC_PF_Z32I

  compositeType = PC_COMP_DEPTH // Compositing operator

  // Image transfer options
  compressionHint = PC_COMPRESSION_NONE
  retainOutputCount = 1
  volatileFrameletCount = 0

  // Whether the depth values of the output are needed
  outputDepth = false

  networkID = PC_ID_DEFAULT // Which network layer is used

  // Array of processes belong to this context
```

```
      processes= [ "Process #0", "Process #1" , "Process #2"]
    } // end of context
```

Listing 2.6: Structure of a compositing context in the low-level script

For a real-world example see Listing 3.4 and Listing 3.6 in Section 3.

2.3.3 Level 2: Dynamic Scripts

Dynamic scripts are cluster-independent procedural scriptlets generating low-level scripts for arbitrary cluster descriptions. With the aid of dynamic scripts the benchmark scenarios are portable; one can copy them from one cluster to another.

The dynamic scripts have two inputs:

- the *cluster description* provided by the framework,
- a set of *customization parameters* which can be modified by the user or the framework.

Structure of Dynamic Scripts

The dynamic script must contain these two functions (see Listing 2.7):

getDynamicScriptParameters() provides the list of customization parameters.

createLowLevelScript(clusterStr) is called after the cluster description has been created by the framework and all of the customization parameters have been set. This function creates the low-level script.

```
// This function returns the array of parameters
function getDynamicScriptParameters() {
  // Return the array of dynamic script parameters
  return [
    // . . .
  ]
}

// This function returns a low-level script structure as a string.
// "clusterStr" is a Squirrel structure representing the scanned cluster.
//            It is passed as a string to the function.
function createLowLevelScript(clusterStr) {
  // . . .

  return low_level_script;
}
```

Listing 2.7: Structure of the dynamic script

Structure of Dynamic Scripts – Customization Parameters

The function `getDynamicScriptParameters()` returns an array of *customization parameters*. Note, that the customization parameters can also be dynamically generated. Each parameter has the following attributes (see Listing 2.8):

name is the name of the parameter. It has to be valid Squirrel identifier (see Appendix A).

type is the type of the parameter. It can have the following values:

`integer`, `float`, `bool`, and `string`

Note, that type checking is not implemented in the current version of ParCompMark.

description is the textual description of the parameter.

possibleValue *(optional)* is the list of values this parameter can have. By default, the parameter can have any value.

defaultValue *(optional)* is the default value of this parameter. By default, the default value is `null` which means the parameter is not defined.

```
// A customization parameter
{
  // The name of the parameter
  name           = "paramname"

  // The type of the parameter
  type           = "paramtype"

  // The textual description of the parameter
  description    = "Description of the parameter"

  // Array of possible values
  possibleValues = [array, of, possible, values]

  // Default value for this parameter
  defaultValue   = default_value
}
```

Listing 2.8: Structure of customization parameters in the dynamic script

Structure of Dynamic Scripts – Low-level Script Generation

The function `createLowLevelScript()` takes the cluster description as its input and generates the low-level script. Both the cluster description and the low-level script are passed as strings from/to the native application (see Listing 2.9).

```
function createLowLevelScript(clusterStr) {

  // First, the cluster description is deserialized to a Squirrel table
  local cluster = stringToObject(clusterStr);
```

```
  // Then the low-level script is created
  local lowLevelScript = {
    // . . .
  }

  // Finally, the ready low-level script structure
  // is serialized to string and returned
  return objectToString(lowLevelScript);
}
```

Listing 2.9: Low-level script generation in the dynamic script

Currently, the structure of the cluster description is simple. It has a `hosts` array with host tables. Each table has an `address` entry that describes the name or IP address of the host (see Listing 2.10).

```
// Sample cluster description
{
  hosts= [
    {
      address = "n12"
    }
    {
      address = "n13"
    }
    {
      address = "n14"
    }
  ]
}
```

Listing 2.10: Structure of the cluster description

2.3.4 Level 3: Scenario Scripts

Scenario scripts must contain these two functions (see Listing 2.11):

prepareScenario() is called when the scenario script is loaded,

getScenarioBatchScript() Generates an array of commands. Then, these commands are executed sequentially by the framework.

```
// This function is called as an initialization step
function prepareScenario() {
  // . . .
}

// This function returns the array of commands
function getScenarioBatchScript() {
  // Return the array of commands
  return [
    // . . .
  ];
}
```

Listing 2.11: Structure of the scenario script

2.4 Commands

The operation of the benchmark can be controlled on the host that executes the commander mode instance of ParCompMark. These commands are also implemented in Squirrel language. Thus, the usability of the command line interface can be easily increased writing more commands.

This section details the syntax of these user commands. To get a list of them type `help` in the ParCompMark terminal. For getting help for a particular command type `help <command name>`.

2.4.1 auto

```
auto - Run automatic cluster detection.

 Usage:
    auto [-n <host count>] [-t <timeout>] [-d <delay>]

 Description:
    Automatically scans computers executing parcompmark
    applications with broadcast messages.
    -n <host count> Wait for specified number of hosts. If
                    not specified, the 'auto' command will
                    scan only once.
    -t <timeout>    Timeout for searching hosts. The default
                    value is 3 seconds.
    -d <delay>      Delay between two searches. The default
                    value is 0.5 second.

 See also:
    lshosts

 Examples:
    Scan once with default settings:
         lshosts

    Try to find 2 hosts in the cluster with timeout value of
    5 seconds and waiting 1 seconds between the searches:
         lshosts -n 2 -t 5 -d 1
```

2.4.2 cleanup

```
cleanup - Cleans up the framework

 Usage:
    cleanup

 Description:
    Removes previously generated low-level script and gathered
    statistics.

 See also:
    start stop
```

2.4.3 compile

```
compile - Compile dynamic script

 Usage:
    compile

 Description:
    Creates low-level script from dynamic script, cluster
    description, and dynamic script parameters. Note that
    the low-level script is cluster specific, but the
    dynamic script can be compiled on any cluster and it
    can also be customized with the parameters.

 See also:
    load param
```

2.4.4 help

```
help - Prints help for a command

 Usage:
    help <command>
```

2.4.5 load

```
load - Loads scenario, dynamic, or low-level script

 Usage:
    load [-s|-d|-l] <script_file>

 Description:
    Loads dynamic or low-level script. If the 'script_file' does
    not have '.nut' extension, it will be automatically added. If none
    of the '-l' and '-d' flags are specified, the script will
    be considered as a scenario script.
    -s   The specified script file name will be considered as a
         scenario script
    -d   The specified script file name will be considered as a
         dynamic script
    -l   The specified script file name will be considered as a
         low-level script

 See also:
    compile param

 Examples:
       Load low-level script file low-level.nut:
         load -l low-level
       Load dynamic script file dynamic.nut:
         load -d dynamic
       Load scenario script file scenario.nut:
         load -s scenario
```

2.4.6 lshosts

```
lshosts - List known hosts in the cluster

 Usage:
   lshosts

 Description:
   Lists known hosts in the cluster. This command prints the
   name of hosts detected by the last auto-detection.

 See also:
   auto
```

2.4.7 param

```
param - Gets and sets dynamic script parameters

 Usage:
   param [-l[d]] | -r | [-g <parameter>] |
         [-s <parameter> <value>]

 Description:

   -l[d]                  List all parameters of the current
                          dynamic script. '-ld' flag gives
                          parameter details.
   -r                     Reset all parameters. The parameters
                          will be changed with the default
                          values.
   -g <parameter>         Get value of the specified parameter.
   -s <parameter> <value> Set value of the specified parameter.

 See also:
   compile load
```

2.4.8 prex

```
prex - Prints value of an expression

 Usage:
   prex <expression1> <expression2> ... <expressionN>

 Description:
   Command prex (print expression) evaluates the expressions
   in its parameter list and prints the values of them.
```

2.4.9 quit

```
quit - Quit from application

 Usage:
    quit|q

 Description:
    Quits from application. If the benchmark is running, this
    command also stops that.

 See also:
    stop
```

2.4.10 start

```
start - Starts the benchmark

 Usage:
    start [-s|-d|-l]

 Description:
    Starts the execution of the benchmark. If a dynamic script if
    specified (instead of low-level script), and the cluster
    description retrieved by the auto-detection is new than the
    compiled script, this command also executes dynamic script
    compilation. If a scenario script file is specified then by
    default the scenario file is executed.

    -s   Starts the execution of the loaded scenario
    -d   Starts the execution of the loaded dynamic script
    -l   Starts the execution of the loaded low-level script

 See also:
    stop compile load

 Examples:
       Starting the loaded highest level script:
         start
       Starting the loaded scenario script:
         start -s
       Starting the loaded dynamic script:
         start -d
       Starting the loaded low-level script:
         start -l
```

2.4.11 stop

```
stop - Stops the benchmark

 Usage:
    stop

 Description:
    Stops the benchmark execution.
```

```
See also:
   start
```

2.5 Renderer Plugins

A renderer plugin must define the following functions. See Appendix D for a plugin example that renders a single triangle per frame.

2.5.1 Plugin Functions

These functions are common for all renderer instances.

```
const char **pcmGetNeededLibs();
```

returns the list of the needed shared libraries.

```
int pcmSetPluginHandle(void *_pluginHandle);
```

sets the plugin handle provided by the framework.

```
typedef void (*logFunType) (const void *, const char *);
int pcmSetLoggerFunction(logFunType _logFun);
```

sets the logger function. This is needed for the plugin to be able to use the logger of the framework.

```
int pcmOnLoad();
```

is called when the renderer defined by the plugin is created. Shared resource management should be placed here.

```
int pcmOnUnload();
```

is called when the framework stops if any renderer was created using this plugin. Shared resource management should be here.

```
const char *pcmGetErrorMsg(const int errorCode);
```

provides an error string for the passed error code. Most of the event handlers return an integer value, which should be zero when no error occurs and non-zero on any error.

2.5.2 Renderer Functions

These functions are renderer instance specific.

```
const char **pcmGetNeededOpenGLExts();
```

returns the list of the needed shared libraries.

```
int pcmSetMiscParam(void *_renderer, const char *name,
                    const char *value);
```

sets miscellaneous parameter for the given renderer. Parameters are passed as string key-value pairs.

```
int pcmSetObjectSpaceBoundingBox(void *_renderer,
    double x0, double y0, double z0,
    double x1, double y1, double z1);
```

sets object space bounding box for the given renderer.

```
int pcmSetObjectId(void *_renderer, unsigned objectId);
```

sets object space distribution with object identifiers for the given renderer.

```
int pcmSetScreenSpaceFramelet(void *_renderer,
    double u0, double v0,
    double u1, double v1);
```

sets screen space bounding framelet for the given renderer.

```
void *pcmOnCreateRenderer(Display *display, Window window,
      XVisualInfo *visualInfo, GLXContext glxContext);
```

is called when a renderer is about to be created. The GLX specific parameters are passed.

```
int pcmOnRender(void *_renderer, double time, unsigned frame);
```

is called when a renderer starts a frame.

```
int pcmOnDestroyRenderer(void *_renderer);
```

is called when a renderer is destroyed.

2.6 XML Output

The commander mode ParCompMark application finally generates an XML output file containing all relevant information on the execution. The philosophy of ParCompMark is "measuring basic information". Parameters like frame rates and latency can also be calculated using the XML output, but they need post-processing (only the frame times are stored for each frame of each process).

2.6.1 Structure of XML Output

The output xml file has two main parts: (1) information on the execution and (2) description of the cluster on which the benchmark was executed.

The statistical information on the execution is separated into more parts according to different executions in a batch processing. After one execution is done hierarchical data collection is performed. The commander mode instance of ParCompMark sends a network broadcast message. The hosts collect these

information from their processes through their nodes and answer the message. These hierarchical structure appears exactly in the xml output. The statistical information is followed by the cluster description.

Two XML namespaces are defined, referring to the local meaning of an element:

info: structural information on a hardware or software element

stat: statistical information measured during the benchmark process

A shortened xml output file can be seen in Listing 2.12. There is one execution and one node with statistics only for the beginning frame.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<info:results xmlns:def="http://www.it2.bme.hu" xmlns:info="http://www.it2.bme.hu
    " xmlns:ref="http://www.it2.bme.hu" xmlns:stat="http://www.it2.bme.hu">
  <info:execution index="0">
  <info:parameters>
    <table>
      <slot name="fullScreen">true</slot>
      <slot name="renderHostCount">1</slot>
      <slot name="exportFrameStep">0</slot>
      <slot name="displayHost">n12</slot>
      <slot name="frameCount">100</slot>
      <slot name="frameWidth">1280</slot>
      <slot name="frameHeight">960</slot>
    </table>
  </info:parameters>
  <info:host name="n12">
    <info:node name="Node #1">
    <info:process name="Process #2">
      <stat:frame average-fps="1.11497" average-triangle-count="0"
        best-fps="-1" best-frame-time="0.016546"
        frame-begin-time="1173278557.169401"
        frame-end-time="1173278557.185947" frame-id="0"
        last-fps="-1" last-frame-time="0.016546"
        last-triangle-count="0" maximal-triangle-count="0"
        minimal-triangle-count="0" read-count="1048576"
        time="0.896889" worst-fps="-1"
        worst-frame-time="0.016546" write-count="0"/>
      </info:process>
    </info:node>
  </info:host>
  </info:execution>
  <info:cluster>
    <info:n12 name="n12">
      <info:param name="pc-number-networks" value="2"/>
        <info:network-ids-names>
          <info:network-id-name id="0" name="Infiniband"/>
          <info:network-id-name id="1" name="TCP/IP"/>
        </info:network-ids-names>
        <info:param name="pc-vendor" value="Hewlett Packard"/>
        <info:param name="pc-extensions"
          value="EXT_IO_count EXT_cur_gfx_ctx HP_frame_output"/>
        <info:param name="pc-volatile-framelet-limit" value="0"/>
        <info:param name="pc-retain-ouput-limit" value="2"/>
        <info:CPUs>
          <info:CPU id="0">
            <info:param name="vendor-id" value="AuthenticAMD"/>
            <info:param name="flags" value="fpu vme de pse tsc
              msr pae mce cx8 apic sep mtrr pge mca cmov pat
```

```
            pse36 clflush mmx fxsr sse sse2 syscall nx
            mmxext lm 3dnowext 3dnow pni ts"/>
          <info:param name="bogomips" value="3981.31"/>
        </info:CPU>
        <info:CPU id="0">
          <info:param name="vendor-id" value="AuthenticAMD"/>
          <info:param name="flags" value="fpu vme de pse tsc
            msr pae mce cx8 apic sep mtrr pge mca cmov pat
            pse36 clflush mmx fxsr sse sse2 syscall nx
            mmxext lm 3dnowext 3dnow pni ts"/>
          <info:param name="bogomips" value="3981.31"/>
        </info:CPU>
          </info:CPUs>
              <info:GPUs>
        <info:GPU/>
      </info:GPUs>
    </info:n12>
  </info:cluster>
</info:results>
```

Listing 2.12: Sample outputfile

The frame has the following statistical data:

frame-id is the id of the frame starting with 0. Frames belonging to the same compositing context have the same id.

average-fps is the average frame rate calculated since the start of the execution.

best-fps is the best frame rate since the start of the execution.

worst-fps is the worst frame rate since the start of the execution.

last-fps is the frame rate for the last frame.

frame-begin-time is the time on the host timer at the start of the frame.

frame-end-time is the time on the host timer at the end of the frame.

best-frame-time is the best frame time the start of the execution.

worst-frame-time is the worst frame time since the start of the execution.

last-frame-time is the frame time of the last frame.

average-triangle-count is the average triangle count calculated since the start of the execution.

maximal-triangle-count is the minimal triangle count since the start of the execution.

minimal-triangle-count is the maximal triangle count since the start of the execution.

last-triangle-count is the triangle count on the last frame.

read-count is the network received data during the last frame.

write-count is the network sent data during the last frame.

The cluster description contains information about the CPUs, GPUs, and ParaComp parameters on the hosts.

2.6.2 Post-processing of the XML Output

From the simple xml output rather complicated statistics can be calculated and presented in a nice output format. Textual output can be generated from the XML output format with an XSLT transformation script. For an example see Appendix E.

Chapter 3

A Detailed Example: Creating *"Brute Force" Triangle Renderer* Benchmark

In this chapter a practical benchmarking example is presented in detail. After setting up the expectations, a solution is introduced from the very basic level to the highest abstraction level (i.e. from creating a low-level, cluster-specific script to the definition of the scenario batch script, which can run on an unidentified hardware and can gather results).

Finally, a renderer plugin is presented with which the rendering job can be customized in an efficient way, using native C/OpenGL code.

3.1 Purpose of Benchmark

It is a common problem that a high resolution polygonal model has to be rendered with interactive frame rates. When the triangle count of the model is so high that one node is unable to visualize it interactively, the rendering job has to be decomposed using data distribution among the nodes.

The aim of this example is simple:

1. render controllable number of triangles,
2. some parameters of the rendering ought to be modifiable (e.g. the number of light sources, immediate or retained mode rendering, depth occlusion or semi-transparent rendering mode),
3. the rendering times should be reported to show how the performance depends on the number of triangles and the allocated rendering nodes.

Using this benchmark one can demonstrate that the number of triangles can be scaled up by adding more nodes. In the next section this informal description is detailed more formally.

3.2 Inputs and Benchmark Requirements

The inputs of the benchmark test are the following:

1. a list containing the names of 1 to n render nodes (assume n to be up to 32).
2. the name of the display node.
3. the number of triangles to be rendered per frame:
   - constant number of triangles renderer per frame,
   - the minimum and maximum number of triangles to render per frame. The actual number of rendered triangles will be a linear function of the frame id.
4. whether immediate mode or retained mode rendering should be used. Retained mode should be performed with display lists and vertex arrays.
5. the frame count and the number of frames to be rendered by a render node.
6. a frame report count.
7. width and height of the image – e.g. 1280×1024.
8. an option that allows for selecting one of several techniques for drawing the triangles.

The benchmark should do the followings:

1. The benchmark should report the *average frames per second* and *latency* achieved using 1 render node, then 2 render nodes, and so on up to n render nodes. Latency is the time from triggering the start of rendering a frame on all the nodes to the actual display of the output to the user.
2. Report the average fps and latency every frame-report-count frames.
3. Render the specified number of triangles per frame (one framelet) on each of the render nodes.
4. Composite the output of the render nodes and display the result on the display node.
5. Allow the user to determine visually from the output display that the test and compositing are running properly.
6. Use either retained mode or immediate mode as specified.

The list of render nodes and the width and height of the image have to be passed to the startup script. Usually, the commander mode instance of ParCompMark is the compositing node too, but it is not necessary. The execution of the benchmark can be controlled from a different node that displays the final output. The number of triangles and frame count can be given in any type of scripts. Using a low-level script everything is wired into the script code. When

using a dynamic script these parameters are interactively set by the user. Since the execution of a scenario script is fully automatic, it contains the possible frame count values. However, it is not necessary to wire the amount of rendering nodes since the scenario script can also use the scanned cluster description. Therefore, it can generate batch jobs on any number of rendering nodes. A default value has to be specified for the resolution of the output, which can be redefined in the startup scripts. ParCompMark saves the starting and ending time of rendering and compositing for every single frame. Frame rates and latency values have to be computed from these values in the post-processing step, and the frame report count should be specified in that step, too.

3.3 Implementation: How to Write Scripts?

The scripting system of ParCompMark is based on Squirrel.

3.3.1 Learning Squirrel in One Minute

The syntax of Squirrel is quite simple, it mostly like C/C++/Java syntax. For more details and for pointers to the Squirrel language, see Appendix A. Here, the most important Squirrel structure to know is the *table*, which can have slots, i.e. key-value pairs. The key should be a valid identifier. The value can be any valid Squirrel object. The other important structure is the *array*, which can be addressed by integer values and can have elements with the same type. See Listing 3.1 for examples.

```
local myTable = {
  // Simple slots (integer, string, array)
  key1 = 1234
  key2 = "Hello World!"
  key3 = [1, 2, 3, 4]

  // A slot can be a table too
  key4 = {
    keyA = 3.14159
    keyB = "It is the PI"
  }

  // An array can have table elements
  key5 = [
    // This is the first element
    {
      keyI = 98765
      keyII = "I'm a value"
    }

    // This is the second one
    {
      key = "That's all"
    }
  ]
}
```

Listing 3.1: Most important Squirrel structures for writing low-level scripts

Handling these structures should be easy for a C++ programmer, only the `rawin()` method is Squirrel-specific. It checks for a key in a table structure. See Listing 3.2 for examples.

```
// This will print 1234
print("myTable.key1 is " + myTable.key1);

// This will print 2
print("myTable.key2[1] is " + myTable.key2[1]);

// Checking for key "key2" in myTable
// If we have we use it (in this case we have it)
if(myTable.rawin("key2"))
  print(myTable.key2)

// This will fail
if(myTable.rawin("we_dont_have_this_key"))
  print(myTable.we_dont_have_this_key)
else
  throw "Hey, we have an error"
```

Listing 3.2: Most important Squirrel functions for handling *table* and *array* structures

3.3.2 Where Should I Place the Script Files?

ParCompMark searches the scripts in the `<dataDirectory>/scripts` directory. The `dataDirectory` is set in the `parcompmark.ini` file which is in the user's HOME directory (`~/.ParCompMark/parcompmark.ini`). Note that this ini file is environment dependent. It is generated at the first execution of ParCompMark. Therefore, if this file is not found, the ParCompMark binary should be executed and this first execution will create one.

Each scripting level has a separate directory inside the `<dataDirectory>/scripts` directory:

- `<dataDirectory>/scripts/low-level/` for low-level scripts,
- `<dataDirectory>/scripts/dynamic/` for dynamic scripts,
- `<dataDirectory>/scripts/scenario/` for scenario scripts.

3.3.3 How Can I Execute My Scripts?

ParCompMark has three different flags for setting what kind of script to execute:

- `-l <scriptname.nut>` for low-level scripts,
- `-d <scriptname.nut>` for dynamic scripts,
- `-s <scriptname.nut>` for scenario scripts.

For example, the following command should be executed in order to run a low-level script called `lltest.nut` using VNC:

```
parcompmarkvnc.sh -g 1024x768 -l -w n[12-13] -a \
    -o lltest-output.xml -l lltest.nut
```

See Section 2.2.1 for more details. After starting the application the following output should be shown:

```
==========================================
==  ParCompMark Terminal
==========================================
Type 'help' for hints!
Scanning cluster...
2 hosts found in the cluster: n12, n13

: _
```

Command `start` starts the benchmark, `stop` stops it, and `quit` or `q` is for quit. Now the final output should look like the following:

```
==========================================
==  ParCompMark Terminal
==========================================
Type 'help' for hints!
Scanning cluster...
2 hosts found in the cluster: n12, n13

: start
Starting benchmark...

: q
Stopping benchmark...
Quiting...
```

3.4 Implementation Level 1: Writing Low-Level Scripts

3.4.1 Structure of Low-Level Scripts

The main structure of the low-level scripts is illustrated in Listing 3.3. See Section 2.3.2 for more details.

```
{
  name = "<The name of the script>" // Name of the low-level script

  // This is the array of the hosts allocated for benchmarking
  hosts =[
    // This is host #0
    {
      name = "<name of host #0>"

      // This is the array of the nodes on host #0
      nodes =[
        // This is node #0
```

```
        {
          name = "<name of node #0>"

          // This is the array of the buffers on node #0
          buffers =[
            // This is buffer #0
            {
              name = "<name of buffer #0>"
            }
          ]

          // This is the array of the processes on node #0
          processes =[
            // This is process #0
            {
              name = "<name of process #0>"
            }
          ]
        }
      ]
    }
  ]

  // The other part of the low-level script is the context array
  contexts =[
    // This is context #0
    {
    }
  ]

} // end of low-level script
```

Listing 3.3: Main structure of low-level scripts

The *low-level script* itself is a Squirrel table with three slots:

- `name` is the name of the script,
- `hosts` is the array of hosts of the cluster allocated for benchmarking,
- `contexts` is the array of compositing contexts that operate in parallel.

Each *host* table has the following slots:

- `name` is the name of the host,
- `nodes` is the array of nodes on this host.

Each *node* table has the following slots:

- `name` is the name of the node.
- `buffers` is the array of buffers on this node,
- `processes`is an array of processes on this node.

After the introduction let us see a real working low-level script!

3.4.2 Low-Level Script Rendering One Triangle

The following highly commented script defines two nodes on hosts `n12` and `n13`. The node on `n12` has one renderer process (`"Process #0"`) while the node on `n13` has a renderer process (`"Process #1"`) and a compositing process (`"Process #2"`). All processes belong to the same context defined in the last part of the script.

See the script in Listing 3.4.

```
{
  name = "Triangle renderer script" // Name of the low-level script

  // This is the array of the hosts allocated for benchmarking
  hosts =[
    // This is the first host
    {
      name = "n12" // The (network) name of the host

      // This is the array of the nodes on "n12"
      nodes =[
        // This host contains one node
        {
          name = "Node #0" // The name of the node

          // The array of the buffers on "Node #0"
          buffers =[
            // we need a buffer for rendering
            {
              name = "Buffer #0" // The name of the buffer

              // Offsets and sizes
              // Note that buffers in this script have
              // no offset values and the same sizes.
              // This example models object-space decomposition
              // With different offset/size values one can specify
              // screen-space decomposition.
              left = 0
              top = 0
              width = 512
              height = 512

              depth = PC_PF_Z32I // Depth format
            }
          ] // end of buffers

          // The array of the processes on "Node #0"
          processes =[
            // This is a renderer process
            {
              name = "Process #0" // The name of the process
              type = Process.RENDER // The type of the process
              buffer = "Buffer #0" // The name of the used buffer

              // The following two procedures
              // describe the operation of the process.
              // The code is given as a string,
              // the addressed host will execute them at every frame
              // after initializing the process.

              // Initialization procedure,
              // does nothing, so it can be
```

```
            // commented out.
            //initProc = @"function initProc(){}"

            // Rendering procedure
            runningProc = @"function runningProc(time, frame) {
              time = time.tofloat();

              RE.perspective(90.0, 0.01, 100.0);
              RE.setCameraPosition(1.0, 1.0, 1.0);
              RE.setCameraTarget(0.0, 0.0, 0.0);
              RE.setCameraUpVector(0.0, 0.0, 1.0);
              RE.rotate(time * 10.0, 0.0, 0.0, 1.0);
              RE.translate(0.0, 0.0, 0.1);

              // Draw a triangle with vertices
              // (1.0, 0.0), (0.0, 0.0) and
              // (0.0, 1.0)
              RE.setColor(1.0, 0.0, 0.0, 1.0);
              RE.drawTriangle();
            }" // end of runningProc
          } // end of the process
        ] // end of processes
      } // end of the node
    ] // end of nodes
  } // end of the first host

  // This is the second host
  {
    name = "n13"

    nodes =[
      // This host also contains one node
      {
        name = "Node #1"

        // This node contains two buffers:
        // one for rendering and one for compositing
        buffers =[
          // a buffer for rendering
          {
            name = "Buffer #1"
            left = 0
            top = 0
            width = 512
            height = 512
            depth = PC_PF_Z32I
          }

          // a buffer for compositing
          {
            name = "Buffer #2"
            left = 0
            top = 0
            width = 512
            height = 512
            depth = PC_PF_Z32I
          }
        ] // end of buffers

        // This node contains two processes:
        // one renders and the other composites the renderings
        processes =[
```

```
          // The rendering process
          {
            name = "Process #1"
            type = Process.RENDER
            buffer = "Buffer #1"

            // Initialization procedure
            //initProc = @"function initProc(){}"

            // Rendering procedure
            runningProc = @"function runningProc(time, frame) {
              time = time.tofloat();

              RE.perspective(90.0, 0.01, 100.0);
              RE.setCameraPosition(1.0, 1.0, 1.0);
              RE.setCameraTarget(0.0, 0.0, 0.0);
              RE.setCameraUpVector(0.0, 0.0, 1.0);
              // This triangle rotates in
              // the opposite direction
              RE.rotate(-time*10.0, 0.0, 0.0, 1.0);

              // Draw a triangle
              RE.setColor(0.0, 1.0, 0.0, 1.0);
              RE.drawTriangle();
            }" // end of runningProc
          } // end of process

          // The compositing process
          {
            name = "Process #2"
            type = Process.COMPOSITE
            buffer = "Buffer #2"

            // Indicate that we would like
            // to display the results
            display = true

            // A simple compositing process
            // does not have to do anything
            //initProc = @"function initProc(){}"
            //runningProc = @"function runningProc(time, frame) {}"
          } // end of process
        ] // end of processes
      } // end of node
    ] // end of nodes
  } // end of the second host
] // end of hosts

// The other part of the low-level script is the context array
contexts =[
  // In this case there is only one context
  // which contains all of the processes
  {
    // Whether use framelet extension or not
    glFrameletEXT = false

    frameWidth = 512 // Sizes of the global frame
    frameHeight = 512

    colourFormat = PC_PF_BGRA8 // Pixel format
    depthFormat = PC_PF_Z32I
```

```
      compositeType = PC_COMP_DEPTH // Compositing operator

      // Image transfer options
      compressionHint = PC_COMPRESSION_NONE
      retainOutputCount = 1
      volatileFrameletCount = 0

      // Whether the depth values of the output are needed
      outputDepth = false

      networkID = PC_ID_DEFAULT // Which network layer is used

      // Array of processes belonging to this context
      processes =[ "Process #0", "Process #1" , "Process #2"]
    } // end of context
  ] // end of contexts
} // end of low-level script
```

Listing 3.4: A low-level script that renders one triangle per rendering process

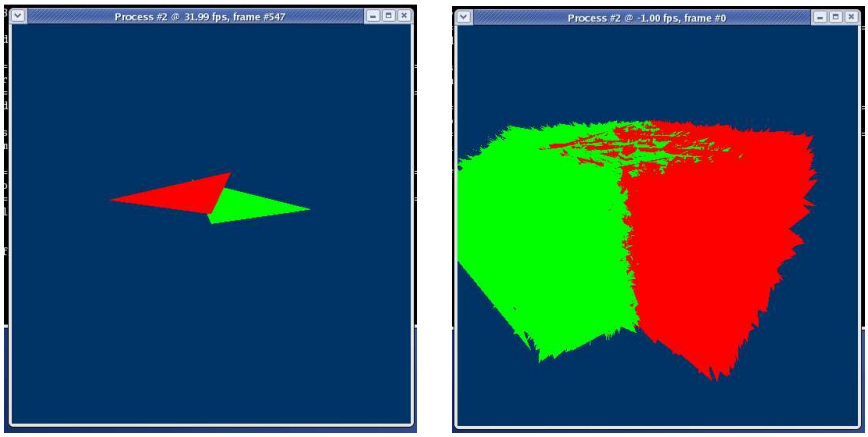
```
runningProc = @"function runningProc(time, frame) {
  time = time.tofloat();

  RE.perspective(90.0, 0.01, 100.0);

  RE.setCameraPosition(1.0, 1.0, 1.0);
  RE.setCameraTarget(0.0, 0.0, 0.0);
  RE.setCameraUpVector(0.0, 0.0, 1.0);
  // This triangle rotates in
  // the opposite direction
  RE.rotate(-time*10.0, 0.0, 0.0, 1.0);

  RE.setColor(0.0, 1.0, 0.0, 1.0); // Draw a triangle
  RE.drawTriangle();
}" // end of runningProc
```

Listing 3.5: Rendering one triangle (cut from Listing 3.4)

The rendering part of `Process #1` is shown in Listing 3.5. The `runningProc` function has two parameters. `time` is used for time-based i.e. real-time renderings and holds the time elapsed since the start of the benchmark in seconds. On the other hand, `frame` contains the actual frame index. This value can be used for "offline" rendering, i.e. creating animation frames with fixed frame rates. These values come from the native application code as strings, therefore they should be converted to float/int values before usage.

Next, the projection and modelview matrices are set. The modelview matrix is rotated with a time varying angle. This introduces animation into the rendering.

Finally, the filling color is set and the triangle is drawn. All of these operations can be accessed using the methods of the rendering engine (`RE` object in the process code) which stands for the rendering engine of the process (member function of the `RE` object are listed in Appendix B). The output display is shown in Figure 3.1 (a).



(a) Rendering one triangle (b) Rendering triangle list

Figure 3.1: Rendering outputs of the one triangle per process and the triangle list cases.

3.4.3 Low-Level Script Rendering Multiple Triangles with Random Vertices

Obviously, rendering one triangle per process can hardly be called a benchmark operation (see Listing 3.5). Rendering more than one triangles can be efficiently done using the capabilities of the rendering engine class (`RE` in the scripts, see Appendix B) instead of using for-loops in the script. The execution time of this latter solution would be dominated by the script-native code callback times which are negligible when rendering a lot of (e.g. at least several tens of thousands) triangles.

Method `RE.generateRandomTriangles(dimension, count)` can be used for generation a triangle list and `RE.renderObject(trilist, useVertexArray)` for displaying it (see Listing 3.6). The display is shown by Figure 3.1 (b).

```
initProc = @"function initProc(){
  trilist<-RE.generateRandomTriangles(3, 10000);
}"

// Rendering procedure
runningProc = @"function runningProc(time, frame) {
  time = time.tofloat();

  RE.perspective(90.0, 0.01, 100.0);
  RE.setCameraPosition(1.2, 1.2, 1.2);
  RE.setCameraTarget(0.0, 0.0, 0.0);
  RE.setCameraUpVector(0.0, 0.0, 1.0);
  RE.rotate(time * 10.0, 0.0, 0.0, 1.0);

  RE.setColor(1.0, 0.0, 0.0, 1.0);
  RE.renderObject(trilist, false);
}" // end of runningProc
```

Listing 3.6: Rendering multiple triangles with random vertices

3.5 Implementation Level 2: Writing Dynamic Scripts

Writing a low-level script is not the most effective solution for creating benchmark jobs. First, when the benchmark job has to be moved to another cluster, the host names has to be changed. On the other hand, these jobs usually measure horizontal scalability – i.e. how the performance goes up when allocating more nodes for the job. In this way, the code snippet in Listing 3.6 would look like the following:

```
initProc = @"function initProc(){
  // local triangleCount = ...
  // local nodeCount = ...

  trilist<-RE.generateRandomTriangles(3, triangleCount/
       nodeCount);
}"
```

To handle these issues, the dynamic scripts are introduced (see Section 2.3.3 for more details).

3.5.1 Structure of Dynamic Scripts

Dynamic script generates the low-level script. Its input is a cluster description and a set of customization parameters. Its output is the low-level script. The main structure of dynamic scripts is illustrated in Listing 3.7. See Section 2.3.3 for more details.

```
// This function returns the array of parameters
function getDynamicScriptParameters() {
  // Return the array of dynamic script parameters
  return [
    // Each parameter is represented by the squirrel table
    {
      name="paramname" // The name of the parameter
      type="paramtype" // The type of the parameter
      // The textual description of the parameter
      description="Description of the parameter"
      // Array of possible values
      possibleValues=[array, of, possible, values]
      // Default value for this parameter
      defaultValue=default_value
    }
  ];
}

// This function returns a low-level script structure as a string.
// "clusterStr" is a Squirrel structure representing the scanned cluster.
//             It is passed as a string to the function.
function createLowLevelScript(clusterStr) {
  // First convert the cluster description to Squirrel table
  local cluster = stringToObject(clusterStr);

  // Create empty low-level script structure
  // createEmptyLowLevelScript is a predefined function that
```

```
  // can be used in dynamic scripts to create an empty
  // low-level structure
  local lowLevelScript =
    createEmptyLowLevelScript("<The name of the benchmark>", cluster);

  //  . . .
  // [Here comes the code that adds nodes, processes,
  //  and contexts to the low-level script structure]
  //  . . .

  // The values of dynamic script parameters can be accessed
  // using the DYNAMICSCRIPTPARAMETERS global table
  // For example: DYNAMICSCRIPTPARAMETERS.paramname
  //
  // The value of the parameter is null when it is not set
  // (this can be useful for optional parameters)

  // Convert the ready low-level script structure to string
  // and return
  return objectToString(lowLevelScript);
}
```

Listing 3.7: Main structure of dynamic scripts

The operation of the dynamic script is simple. Two functions have to be defined:

getDynamicScriptParameters() : This function must return an array of the possible parameters for this script. Thus, each dynamic script defines its own interface and its customization parameters. These parameters can be queried with the `param -l` command in the command line interface (see Section 2.4.7).

createLowLevelScript() : This function generates a low-level script for the given cluster customized with the dynamic script parameters.

Currently, the structure of the cluster description is simple. It has a `hosts` array with host tables. Each table has an `address` entry that describes the name or IP address of the host.

Two global tables can be accessed at any part of the dynamic script:

ENVIRONMENTVARIABLES acts as shell environment variables for the script. Currently the following items are provided for the script:

- `screenWidth`, `screenHeight` are the screen dimensions defined in the startup script (see Section 2.2.2),
- `commanderHost` is the name of the commander host.

DYNAMICSCRIPTPARAMETERS stores the set of dynamic script parameters as slots (the parameter name is the key and the value of the parameter is the value of the Squirrel slot). The value of the parameter can be null when it is not set. This can be useful for optional parameters, which do not have to have a value.

Function `createEmptyLowLevelScript()` creates an empty script with the given name for the given cluster. In Listing 3.8 the empty low-level script is presented for the cluster description illustrated in Listing 2.10.

```
// Sample empty low-level script
{
  name = "<The name of the benchmark>"

  // Host list
  hosts = [
    // Host #0
    {
      name = "n12"
    }

    // Host #1
    {
      name = "n13"
    }

    // Host #2
    {
      name = "n14"
    }
  ]

  // No contexts
  contexts = []
}
```

Listing 3.8: Structure of an empty low-level script

3.5.2 Dynamic Script with Basic Functionality

Now, a real-world script is presented with basic functionality. In the next section this script is further improved.

The basic dynamic script has six parameters:

`renderHostCount` is the number of render hosts. It cannot be higher than the number of the allocated nodes in the cluster.

`displayHost` is the name of the display host. This host will render the composited output.

`triangleCount` is the number of triangles per frame to render. These triangles will be shared among the render nodes, i.e. one node will render `triangleCount`/`renderHostCount` triangles.

`frameCount` is the number of frames to render. Setting the value of zero means infinite rendering.

`frameWidth` and `frameHeight` are the dimensions of the global frame.

The default values and the array of the possible values are calculated using the environment variables.

The summary of the operation of function `getDynamicScriptParameters()` is the following:

1. creating an empty low-level script,
2. creating a compositing context,
3. iterating on the hosts – creating renderer processes,
4. creating a compositing process.

See the code in Listing 3.9.

```
function getDynamicScriptParameters() {

  // Get host list from the native code
  local hostList = stringToObject(Application.getInstance().
       getHostListForSquirrel());

  // Find out which is the default display host
  local displayHost =
    ENVIRONMENTVARIABLES.rawin("commanderHost") ? ENVIRONMENTVARIABLES.
         commanderHost :
    (hostList.len() ? hostList[0] : null);

  // Find out how many hosts can do rendering,
  // assuming that the display host will not render anything.
  local renderHostCount = displayHost ? hostList.len()-1 : hostList.len();

  // Create a list with elements of 1, 2, ..., renderHostCount.
  // This will be needed for "possibleValues" attribute of
  // parameter "renderHostCount".
  local renderHostCountList = [];
    for(local i=1; i<=renderHostCount; i++) renderHostCountList.append(i);

  // Get screen properties
  local screenWidth = ENVIRONMENTVARIABLES.rawin("screenWidth") ?
    ENVIRONMENTVARIABLES.screenWidth.tointeger() : 640;
  local screenHeight = ENVIRONMENTVARIABLES.rawin("screenHeight") ?
    ENVIRONMENTVARIABLES.screenHeight.tointeger() : 480;

  // Now we have enough information to generate the parameter list.
  // Return the array of parameters
  return [
    {
      name="renderHostCount" type="integer"
      description="Number of render hosts"
      possibleValues=renderHostCountList
      defaultValue=(renderHostCount>0 ? renderHostCount : null)
    }
    {
      name="displayHost" type="string" description="Name of display host"
      possibleValues=hostList
      defaultValue=displayHost
    }
    {
      name="triangleCount" type="integer"
      description="Number of triangles to render per frame"
      defaultValue=10000
    }
    {
      name="frameCount" type="integer" description="Number of frames to render"
      defaultValue=100
    }
    {
```

```
      name="frameWidth" type="integer"
      description="Width of the frame in pixels"
      defaultValue=screenWidth
    }
    {
      name="frameHeight" type="integer"
      description="Height of the frame in pixels"
      defaultValue=screenHeight
    }
  ];
}

function createLowLevelScript(clusterStr) {
  local cluster = stringToObject(clusterStr);

  // Create default script
  local lowLevelScript =
    createEmptyLowLevelScript("Triangle renderer Script", cluster);

  // Create a context
  local context = {
    // Whether use framelet extension or not
    glFrameletEXT = false

    // Sizes of the global frame
    frameWidth = DYNAMICSCRIPTPARAMETERS.frameWidth
    frameHeight = DYNAMICSCRIPTPARAMETERS.frameHeight

    colourFormat = PC_PF_BGRA8 // Pixel format
    depthFormat = PC_PF_Z32I

    compositeType = PC_COMP_DEPTH // Compositing operator

    // Image transfer options
    compressionHint = PC_COMPRESSION_NONE
    retainOutputCount = 1
    volatileFrameletCount = 0

    // Whether the depth values of the output are needed
    outputDepth = false

    // Which network layer is used
    networkID = PC_ID_DEFAULT

    // Create empty process list
    processes =[ ]
  };

  // Add this context to the low-level script
  lowLevelScript.contexts.append(context);

  local nodeCount = 0; // Define counters for id generation
  local processCount = 0;
  local bufferCount = 0;

  // Iterate on every host to create rendering processes
  foreach(host in lowLevelScript.hosts) {

    // Skip display hosts
    if(DYNAMICSCRIPTPARAMETERS.rawin("displayHost") &&
      host.name==DYNAMICSCRIPTPARAMETERS.displayHost)
      continue;
```

```
// Skip the hosts above the specified render host count
if(nodeCount >= DYNAMICSCRIPTPARAMETERS.renderHostCount)
  break;

// Create a buffer for the renderings
local buffer = {
  name = "Buffer #" + bufferCount++
  left = 0
  top = 0

  width = context.frameWidth // Use the sizes of the context
  height = context.frameHeight

  depth = PC_PF_Z32I
}

// Create a renderer process
local process = {
  name = "Process #" + processCount++
  type = Process.RENDER
  buffer = buffer.name // This process uses the previously defined buffer

  // Initialization procedure
  initProc = @"function initProc(){
    local renderers = "+DYNAMICSCRIPTPARAMETERS.renderHostCount+@";
    local tris = "+DYNAMICSCRIPTPARAMETERS.triangleCount+@";
    local myIndex = "+processCount+@"-1;

    // Generate triangles
    trilist<-RE.generateRandomTriangles(3, tris/renderers);

    // Set unique color for drawing
    color<-getUniqueColor(myIndex, renderers);
  }"

  // Rendering procedure
  runningProc = @"function runningProc(time, frame) {
    time = time.tofloat();

    // Setup matrices
    RE.perspective(90.0, 0.01, 100.0);
    RE.setCameraPosition(1.2, 1.2, 1.2);
    RE.setCameraTarget(0.0, 0.0, 0.0);
    RE.setCameraUpVector(0.0, 0.0, 1.0);
    RE.rotate(time * 10.0, 0.0, 0.0, 1.0);

    // Render the triangles
    RE.setColor(color[0], color[1], color[2], 1.0);
    RE.renderObject(trilist, false);
  }" // end of runningProc

  // Set stop frame id
  // (0 means endless rendering)
  stopID = DYNAMICSCRIPTPARAMETERS.frameCount
}

// Append this process to the context
context.processes.append(process.name);

// Create a node on the host
host.nodes.append({
```

```
      name = "Node #" + nodeCount++
      buffers =[ buffer ]
      processes =[ process ]
    });
  } // end of foreach

  // Now, create a process for displaying the output
  {
    // Compositing host by default is the first host
    local host = lowLevelScript.hosts[0];

    if(DYNAMICSCRIPTPARAMETERS.rawin("displayHost")) {
      host = findItemByName(lowLevelScript.hosts, DYNAMICSCRIPTPARAMETERS.
          displayHost);
      if(host == null) throw "Host \'" + name + "\' not found."
    }

    // Create a buffer for the compositing
    local buffer = {
      name = "Buffer #" + bufferCount++
      left = 0
      top = 0

      width = context.frameWidth // Use the sizes of the context
      height = context.frameHeight

      depth = PC_PF_Z32I
    }

    // Create process
    local process = {
      // Use the name of the low-level script
      // This will be shown in the title bar
      name = lowLevelScript.name
      type = Process.COMPOSITE
      buffer = buffer.name
      display = true

      // (No initialization or running procedure is needed)

      // Set stop frame ID
      stopID = DYNAMICSCRIPTPARAMETERS.frameCount
    }

    // Append this process to the context
    context.processes.append(process.name);

    // Create a node on the host
    host.nodes.append({
      name = "Node #" + nodeCount++
      buffers =[ buffer ]
      processes =[ process ]
    });
  }

  // Return low-level script as a string
  return objectToString(lowLevelScript);
}
```

Listing 3.9: Script example: Basic functionality

For comparison, the previous dynamic script generates the following low-

level script in case of two nodes (n12 and n13). See Listing 3.10.

```
{
  name="Triangle renderer Script"
  hosts =[
    {
      name="n12"
      nodes =[
        {
          name="Node #0"
          buffers =[
            {
              width=1280
              left=0
              name="Buffer #0"
              depth=196864
              height=960
              top=0
            }
          ]
          processes =[
            {
              type=1
              stopID=100
              name="Process #0"
              buffer="Buffer #0"
              initProc=@"function initProc(){
                local renderers = 1;
                local tris = 10000;
                local myIndex = 1-1;

                // Generate triangles
                trilist<-RE.generateRandomTriangles(3, tris/renderers);

                // Set unique color for drawing
                color<-getUniqueColor(myIndex, renderers);
              }"
              runningProc=@"function runningProc(time, frame) {
                time = time.tofloat();

                // Setup matrices
                RE.perspective(90.0, 0.01, 100.0);
                RE.setCameraPosition(1.2, 1.2, 1.2);
                RE.setCameraTarget(0.0, 0.0, 0.0);
                RE.setCameraUpVector(0.0, 0.0, 1.0);
                RE.rotate(time * 10.0, 0.0, 0.0, 1.0);

                // Render the triangles
                RE.setColor(color[0], color[1], color[2], 1.0);
                RE.renderObject(trilist, false);
              }"
            }
          ]
        }
      ]
    }
    {
      name="n13"
      nodes =[
        {
          name="Node #1"
          buffers =[
```

```
          {
            width=1280
            left=0
            name="Buffer #1"
            depth=196864
            height=960
            top=0
          }
        ]
        processes =[
          {
            type=0
            name="Triangle renderer Script"
            buffer="Buffer #1"
            display=true
            stopID=100
          }
        ]
      }
    ]
  }
  ]
  contexts =[
    {
      compressionHint=0
      glFrameletEXT=false
      frameHeight=960
      depthFormat=196864
      colourFormat=196609
      volatileFrameletCount=0
      processes =[
        "Process #0"
        "Triangle renderer Script"
      ]
      retainOutputCount=1
      outputDepth=false
      networkID=2147483647
      frameWidth=1280
      compositeType=8193
    }
  ]
}
```

Listing 3.10: Generated low-level script for two hosts (n12 and n13)

The script can be tried out in the following way. Execute the ParCompMark application on the desired nodes of the cluster. For example, this can be done using the VNC startup script (see Listing 3.11).

```
parcompmarkvnc.sh -g 1024x768 -l -w n[12-13] -a -o dyntest-output.xml
    -d dyntest.nut
```

Listing 3.11: Starting ParCompMark with the basic dynamic script

The parameters can be checked using the `param -l` or `param -ld` command. Type `help param` for more options. The dynamic script can be compiled to the cluster with the `compile` command. However, this is not necessary since the `start` command will check whether the low-level script exists or it is obsolete

and it needs to be recompiled. See Listing 3.12.

```
=======================================
==  ParCompMark Terminal
=======================================
Type 'help' for hints!
Scanning cluster...
2 hosts found in the cluster: n12, n13

: param -ld
The dynamic script has the following parameters:
integer  renderHostCount   1
  Number of render hosts. Default value: 1. Possible values: [1].
string   displayHost       n13
  Name of display host. Default value: n13. Possible values: [n12, n13].
integer  triangleCount     10000
  Number of triangles to render per frame. Default value: 10000.
integer  frameCount        100
  Number of frames to render. Default value: 100.
integer  frameWidth        1280
  Width of the frame in pixels. Default value: 1280.
integer  frameHeight       960
  Height of the frame in pixels. Default value: 960.

: compile
Compiling dynamic script for current cluster...

: start
Starting benchmark...

: q
Quiting...
```

Listing 3.12: Sample execution of the basic dynamic script

3.5.3 Dynamic Script with Improved Functionality

In the following, an improved version of the previous example is presented. This script is attached to the ParCompMark distribution as an example script. Since this code is rather long, it is presented in fractions. First, the dynamic script parameters are described. The new parameters are:

`minTriangleCount` is the minimum number of triangles to render per frame. When it is set, the dynamic script will generate a code that increments the amount of rendered triangles in every frame started with `minTriangleCount` to `triangleCount`.

`renderMode` is the rendering mode. Possible values are:

- `im`: immediate mode,
- `va`: vertex arrays,
- `dl`: display lists.

`compositeMode` is the compositing mode, with possible values

- `de`: depth compositing,

`al`: alpha compositing.

`lightCount` is the number of point lights.

`glFrameletEXT` is a flag that indicates whether to use OpenGL framelet extension or not.

`exportFrameStep` exports every (exportFrameStep)th frame to a png file. Zero value means no frame export.

See Listing 3.13.

```
function getDynamicScriptParameters() {
  // Information about the hosts
  local hostList = stringToObject(Application.getInstance().
       getHostListForSquirrel());
  local displayHost = ENVIRONMENTVARIABLES.rawin("commanderHost") ?
    ENVIRONMENTVARIABLES.commanderHost : (hostList.len() ? hostList[0] :null);
  local renderHostCountList = [];
  local renderHostCount = displayHost ? hostList.len()-1 : hostList.len();
  for(local i=1; i<=renderHostCount; i++) renderHostCountList.append(i);

  // Screen properties
  local screenWidth = ENVIRONMENTVARIABLES.rawin("screenWidth") ?
    ENVIRONMENTVARIABLES.screenWidth.tointeger() : 640;
  local screenHeight = ENVIRONMENTVARIABLES.rawin("screenHeight") ?
    ENVIRONMENTVARIABLES.screenHeight.tointeger() : 480;

  return [
    {name="renderHostCount" type="integer"
     description="Number of render hosts" possibleValues=renderHostCountList
     defaultValue=(renderHostCount>0 ? renderHostCount : null)}

    {name="displayHost" type="string" description="Name of display host"
     possibleValues=hostList defaultValue=displayHost}

    {name="frameCount" type="integer" description="Number of frames to render"
     defaultValue=100}

    {name="frameWidth" type="integer"
     description="Width of the frame in pixels" defaultValue=screenWidth}

    {name="frameHeight" type="integer"
     description="Height of the frame in pixels" defaultValue=screenHeight}

    {name="triangleCount" type="integer"
     description="Number of triangles to render per frame" defaultValue=1000}

    {name="minTriangleCount" type="integer"
     description="Minimum number of triangles to render per frame"
     defaultValue=null}

    {name="renderMode" type="string" description="Rendering mode: immediate"+
     " mode (im), vertex arrays (va), or display lists (dl)"
     possibleValues=["im", "va", "dl"] defaultValue="im"}

    {name="compositeMode" type="string" description="Compositing mode: depth"+
     " compositing (de) or alpha compositing (al) without sorting"
     possibleValues=["de", "al"] defaultValue="de"}

    {name="lightCount" type="integer" description="Number of point lights"
```

```
    defaultValue=1}

  {name="glFrameletEXT" type="boolean" description="Use OpenGL framelet
      extension"
   defaultValue=false}

  {name="exportFrameStep" type="integer" defaultValue=0
   description="Export every (exportFrameStep)th frame to a png file"}
]; }
```

Listing 3.13: Improved dynamic script, part I.

In the first part of the `createLowLevelScript()` function the correctness of the dynamic script parameters is checked. Then the usual empty script generation is done. See Listing 3.14.

```
function createLowLevelScript(clusterStr) {
  // Check for enough render hosts
  if(!DYNAMICSCRIPTPARAMETERS.rawin("renderHostCount"))
    throw "There are no rendering hosts. Please add more hosts to the"+
    " framework.";

  // Check parameter compatibility
  if(DYNAMICSCRIPTPARAMETERS.rawin("minTriangleCount") &&
    DYNAMICSCRIPTPARAMETERS.renderMode == "dl")
    throw "Display lists are not compatible with incremental triangle"+
    " rendering.";

  // Convert parameter to table
  local cluster = stringToObject(clusterStr);

  // Create empty structure
  local lowLevelScript = createEmptyLowLevelScript("Triangle renderer",
      cluster);
  ...
```

Listing 3.14: Improved dynamic script, part II.

Then the compositing context is created and added to the context list. See Listing 3.15.

```
  ...
  local nodeCount = 0; // For id generation
  local processCount = 0;
  local bufferCount = 0;

  // Benchmark duration (0 means endless rendering)
  local numberOfFrames = DYNAMICSCRIPTPARAMETERS.frameCount;

  // Setup context
  lowLevelScript.contexts.append({
    glFrameletEXT = DYNAMICSCRIPTPARAMETERS.glFrameletEXT

    frameWidth = DYNAMICSCRIPTPARAMETERS.frameWidth
    frameHeight = DYNAMICSCRIPTPARAMETERS.frameHeight

    colourFormat = PC_PF_BGRA8
    depthFormat = PC_PF_Z32I
```

```
  compositeType = (DYNAMICSCRIPTPARAMETERS.compositeMode == "de" ?
      PC_COMP_DEPTH : PC_COMP_ALPHA_SORT)
  compressionHint = PC_COMPRESSION_NONE
  retainOutputCount = 1
  volatileFrameletCount = 0
  outputDepth = false
  networkID = PC_ID_DEFAULT

  processes = [] // No processes by default
});
local context = lowLevelScript.contexts[0];
...
```

Listing 3.15: Improved dynamic script, part III.

Then, an iteration is performed on the hosts, just like in the basic version. However, here the `initProc` and the `runningProc` are more complicated See Listing 3.16.

```
...
// Create rendering process with buffer on each host
foreach(host in lowLevelScript.hosts) {

// Skip display hosts
if(DYNAMICSCRIPTPARAMETERS.rawin("displayHost") && host.name==
     DYNAMICSCRIPTPARAMETERS.displayHost) continue;

// Skip the hosts above the specified render host count
if(nodeCount >= DYNAMICSCRIPTPARAMETERS.renderHostCount) break;

// Create buffer
local buffer = {
  name = "Buffer #" + bufferCount
  left = 0
  top = 0
  width = context.frameWidth
  height = context.frameHeight
  depth = PC_PF_Z32I
}
bufferCount++

// Assemble parameter dependent code fragments
local initCodeFragment = ""
local renderCodeFragment = ""

local thisTriangleCount = DYNAMICSCRIPTPARAMETERS.triangleCount/
     DYNAMICSCRIPTPARAMETERS.renderHostCount
local initJob = @"
  // Generate model
  triangles<-RE.generateRandomTriangles(3, "+thisTriangleCount+@");

  // Setup lighting
  local lightCount = "+DYNAMICSCRIPTPARAMETERS.lightCount+@"
  for(local i=0; i<lightCount; i++) {
  RE.setAmbientLight(0.1, 0.1, 0.1, 1.0);

  local light = RE.addLightSource();
  local color = getUniqueInverseColor(i, lightCount)
  local position = getUniqueColor(i, lightCount)
```

```
  RE.setLightSourcePosition(light, 10.0*position[0], 10.0*position[1], 10.0*
      position[2]);
  RE.setLightSourceDiffuse(light, color[0], color[1], color[2], 1.0);
}"
local setColorJob = @"
  local color = getUniqueColor("+processCount+@", "+DYNAMICSCRIPTPARAMETERS.
      renderHostCount+@")

  RE.setAmbientMaterial(0.1, 0.1, 0.1, 0.0)
  RE.setColor(color[0], color[1], color[2], 0.01)
  RE.setDiffuseMaterial(color[0], color[1], color[2], 0.01)
  RE.setSpecularMaterial(1.0, 1.0, 1.0, 0.01, 100)
  "+(DYNAMICSCRIPTPARAMETERS.compositeMode == "al" ? "RE.setBlending(true);"
      : "")+@"
"

// Rendering jobs
local renderJobImmediate = "";
local renderJobVertexArray = "";

// Usage 1: incremental rendering mode handling
if(DYNAMICSCRIPTPARAMETERS.rawin("minTriangleCount")) {
  // Minimum and maximum number of triangles
  local minTri = (DYNAMICSCRIPTPARAMETERS.minTriangleCount/
      DYNAMICSCRIPTPARAMETERS.renderHostCount).tofloat();
  local maxTri = (thisTriangleCount).tofloat();

  // Triangle count increment
  local frameCount = DYNAMICSCRIPTPARAMETERS.frameCount.tofloat();
  local triInc = (maxTri-minTri)/frameCount;

  // Immediate mode render job
  renderJobImmediate = @"
    "+setColorJob+@"
    RE.renderObjectTriangles(triangles, false, ("+minTri+" + frame * "+triInc
        +@").tointeger());
  "

  // Vertex array render job
  renderJobVertexArray = @"
    "+setColorJob+@"
    RE.renderObjectTriangles(triangles, true, ("+minTri+" + frame * "+triInc+
        @").tointeger());
  "
}

// Usage 2: constant triangle count
else {
  // Immediate mode render job
  renderJobImmediate = @"
    "+setColorJob+@"
    RE.renderObject(triangles, false);
  "

  // Vertex array render job
  renderJobVertexArray = @"
    "+setColorJob+@"
    RE.renderObject(triangles, true);
  "
}

// Decide which render mode to use
```

```
switch(DYNAMICSCRIPTPARAMETERS.renderMode) {
  case "im": // Immediate mode
    initCodeFragment =@"function initProc() {
      "+initJob+@"
    }"
    renderCodeFragment = renderJobImmediate;
  break;

  case "va": // Vertex array
    initCodeFragment =@"function initProc() {
      "+initJob+@"
    }"
    renderCodeFragment = renderJobVertexArray;
  break;

  case "dl": // Display list
    initCodeFragment =@"function initProc() {
      "+initJob+@"
      displayList<-RE.createDisplayList();
      "+renderJobImmediate+@"
      RE.finishDisplayList();
    }"
    renderCodeFragment = "RE.executeDisplayList(displayList);"
  break;
} // end-of switch

// Create process
local process = {
  name = "Process #" + processCount++
  type = Process.RENDER
  buffer = buffer.name

  initProc = initCodeFragment
  runningProc = @"function runningProc(time, frame) {
    time = time.tofloat(); frame = frame.tointeger();
    local size = 16.0;

    RE.perspective(90.0, 0.01, 100.0);
    RE.setCameraPosition(10.0, 10.0, 10.0);
    RE.setCameraTarget(0.0, 0.0, 0.0);
    RE.setCameraUpVector(0.0, 0.0, 1.0);
    RE.rotate(-time * 15.0, 0.0, 0.0, 1.0);
    RE.translate(-0.5*size, -0.5*size, -0.5*size);
    RE.scale(size, size, size);

    // Here comes the rendering job
    "+renderCodeFragment+@"
  }"

  // Set dummy sorting order for alpha compositing
  // process count has a unique value for each process
  sortOrder = processCount

  // Set stop frame ID
  stopID = numberOfFrames
}
context.processes.append(process.name);

// Create node
local node = {
  name = "Node #" + nodeCount++
  buffers = [ buffer ]
```

```
    processes = [ process ]
  };
  host.nodes.append(node);
  } // end-of foreach
  ...
```

Listing 3.16: Improved dynamic script, part IV.

Finally, the compositing process is set up and the low-level script is returned (Listing 3.17).

```
  ...
  // Create compositing process
  {
    // Compositing host by default on the first host
    local host = lowLevelScript.hosts[0];

    if(DYNAMICSCRIPTPARAMETERS.rawin("displayHost")) {
      // If it is possible, try to place it one the commander
      host = findItemByName(lowLevelScript.hosts, DYNAMICSCRIPTPARAMETERS.
          displayHost);
      if(host == null) throw "Host "' + name +"\' not found."
    }

    // Create buffer
    local buffer = {
      name = "Buffer #" + bufferCount++
      left = 0; top = 0
      width = context.frameWidth; height = context.frameHeight
      depth = PC_PF_Z32I
    }

    // Create process
    local process = {
      name = lowLevelScript.name
      type = Process.COMPOSITE
      buffer = buffer.name
      display = true
      stopID = numberOfFrames // Set stop frame ID

      // Exporting frames
      exportFrameStep = DYNAMICSCRIPTPARAMETERS.exportFrameStep
      frameFilenamePattern = "datascal-%06d.png"
    }
    processCount++
    context.processes.append(process.name);

    // Create node
    local node = {
      name = "Node #" + nodeCount++
      buffers = [ buffer ]
      processes = [ process ]
    }
        host.nodes.append(node);
  }

  return objectToString(lowLevelScript);
}
```

Listing 3.17: Improved dynamic script, part V.

3.6 Implementation Level 3: Creating Scenario Scripts for Batched Execution

As presented in Section 2.3.4, the scenario script must have two functions with specific names. In the initialization step the dynamic script is loaded (Listing 3.18).

```
function prepareScenario() {
  // Load dynamic script
  load(["-d", "trirenderer.nut"]);

  print("Scenario has been loaded.\n");
}
```

Listing 3.18: Scenario Script I.

The batch script generator method is executed after the dynamic script has been loaded. Any code can be implemented to generate the command list. For example in Listing 3.19 the number of render hosts is increased.

```
function getScenarioBatchScript() {

  // Create empty command list
  local script = [];

  // Set default parameters
  script.append("param -s frameWidth 1280");
  script.append("param -s frameHeight 1024");
  script.append("param -s lightCount 1");
  script.append("param -s renderMode im");
  script.append("param -s triangleCount 100000");
  script.append("param -s frameCount 500");

  // Get host list from the native application
  local hostList = stringToObject(Application.getInstance().
       getHostListForSquirrel());

  // Iterate on render host count
  for(local rhostCount=1; rhostCount<hostList.len(); rhostCount++) {
    // Set render host count
    script.append("param -s renderHostCount " + renderHostCount);

    script.append("start -d"); // Start dynamic script
  }

  return script; // Return the command list
}
```

Listing 3.19: Scenario Script II.

3.7 Post-Processing the Results

The original XML output can be converted to any format using XML transformation and formatting tools (XSLT, XSL-FO). In Listing 3.20 the number of

rendering hosts, the triangle count, the frame start and finish times, and the average frame rates are exported into CSV[1] format.

```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    xmlns:def="http::/www.it2.bme.hu"
    xmlns:info="http://www.it2.bme.hu"
    xmlns:ref="http://www.it2.bme.hu"
    xmlns:stat="http://www.it2.bme.hu" version="1.0">
<xsl:output method="text" />

    <xsl:template match="info:results">
        <xsl:text>hosts;triangle_count;avg_fps;frame-begin-time;frame-end-time</
            xsl:text>
        <xsl:apply-templates select="info:execution[info:parameters/table/slot[@name
            ='compositeMode']/text() = 'al']"/>
    </xsl:template>

    <xsl:template match="info:execution">
        <xsl:value-of select="info:parameters/table/slot[@name='renderHostCount']/
            text()"/><xsl:text>;</xsl:text>
        <xsl:value-of select="info:parameters/table/slot[@name='triangleCount']/text
            ()"/><xsl:text>;</xsl:text>
        <xsl:value-of select="info:parameters/table/slot[@name='frame-begin-time']/
            text()"/><xsl:text>;</xsl:text>
        <xsl:value-of select="info:parameters/table/slot[@name='frame-end-time']/
            text()"/><xsl:text>;</xsl:text>
        <xsl:value-of select="info:host/info:node/info:process[@name='Triangle
            renderer']/stat:frame[last()]/@average-fps"/><xsl:text>;</xsl:text>
    </xsl:template>
</xsl:stylesheet>
```

Listing 3.20: XSLT for post-processing

In Appendix E there are scripts that convert the CSV data into 2D and 3D plots. These scripts are implemented in the *R statistical tool*. Certainly, this is not the only solution. Plots and charts can be generated using the GNU plot tool, Matlab, MS Excel, etc.

3.8 Creating Renderer Plugins

In Appendix D a simple plugin is presented that renders a triangle. Any rendering method/algorithm can be implemented in the method `Renderer::onRender`, this code will be executed on every process that uses it (see Listing 3.21).

```
  . . .
void Renderer::onRender(double time, unsigned frame) throw (Exception) {
  // Create time based color values
  double s = 0.5*sin(time)+0.5;
  double c = 0.5*cos(time)+0.5;

  if(mMiscParams["invColours"] == "yes") {
       glBegin(GL_TRIANGLES);
         glColor3f(c, 0.0, 0.0);    glVertex2f(5.0, 0.0);
```

[1]Comma Separated Values

```
          glColor3f(0.0, s, 0.0);    glVertex2f(0.0, 0.0);
          glColor3f(0.0, 0.0, c);    glVertex2f(0.0, 5.0);
        glEnd();
  }
  else {
        glBegin(GL_TRIANGLES);
          glColor3f(s, 0.0, 0.0);    glVertex2f(5.0, 0.0);
          glColor3f(0.0, c, 0.0);    glVertex2f(0.0, 0.0);
          glColor3f(0.0, 0.0, s);    glVertex2f(0.0, 5.0);
        glEnd();
  }
}
  . . .
```

Listing 3.21: Renderer code snippet in a plugin code (see full source in Listing D.4)

Appendix A

Squirrel Language Reference

Squirrel is a high level imperative/OO programming language, designed to be a powerful scripting tool that meets size, memory bandwidth, and real-time requirements of applications like games. Benchmarking frameworks has very similar requirements, so this engine was applied in ParCompMark.

Although Squirrel offers a wide range of features like dynamic typing, delegation, higher order functions, generators, tail recursion, exception handling, automatic memory management, both the compiler and the virtual machine fit together in about 6k lines of C++ code. [1]

This appendix is a summarized version of the official Squirrel language reference which can be found on `http://squirrel-lang.org/`. In addition here are some other useful sites related to the language:

- `http://wiki.squirrel-lang.org/` (Wiki site)
- `http://squirrel-lang.org/forums/` (Forums)

A.1 Squirrel Syntax

Identifiers

Identifiers start with an alphabetic character or `_` followed by any number of alphabetic characters, `_` or digits (`[0-9]`). Squirrel is a case sensitive language, this means that the lowercase and uppercase representation of the same alphabetic character are considered different characters.

Keywords

The following words are reserved words by the language and cannot be used as identifiers:

```
break case catch class clone constructor continue default delegate
delete else extends false for function if in instanceof local null
resume return static switch this throw true try typeof while parent
yield vargc vargv
```

Operators

Squirrel recognizes the following operators:

```
!      !=     ||     ==     &&     <=     =>     >
+      +=     -      -=     /      /=     *      *=
%      %=     ++     --     <-     =      &      ^
|      ~      >>     <<     >>>
```

Other tokens

Other used tokens are:

```
{   }   [   ]   .   :   ::   '   ;   "   @"
```

Literals

Squirrel accepts integer numbers, floating point numbers, and stings literals (see Table A.1).

`34`	Integer number
`0xFF00A120`	Integer number
`'a'`	Integer number
`1.52`	Floating point number
`1.e2`	Floating point number
`1.e-2`	Floating point number
`"I'm a string"`	String
`@"I'm a verbatim string"`	String
`@" I'm a multi-line verbatim string "`	String

Table A.1: Squirrel literal samples

Comments

This syntax is the same as in ANSI C.

A.2 Values and Data types

Squirrel is a dynamically typed language so variables do not have a type, although they refer to a value that does have a type.

Integer, Float, String, and Bool

An *Integer* represents a 32-bit (or better) signed number. A *Float* represents a 32-bit (or better) floating point number. *Strings* are immutable sequences of characters; to modify a string it is necessary create a new one. Squirrel's strings, behave like in C or C++, they are delimited by quotation marks(") and can contain escape sequences(`\t`, `\a`, `\b`, `\n`, `\r`, `\v`, `\f`, `\\`, `\"`, `\'`, `\0`, `\xhhhh`). Verbatim string literals begin with `@"` and end with the matching quote. Verbatim string literals also can extend over a line break. If they do, they include any white space characters between the quotes. The *Bool* data type can have only two values, the literals `true` and `false` (see Listing A.1).

```
local a = 123 //decimal Integer
local b = 0x0012 //hexadecimal Integer
local c = 'w' //char Integer

local d = 0.234 // Float

// has a newline at the end of the string
local e = "I'm a wonderful string\n"
// the \n is copied in the string same as \\n in a regular string "I'm a
     verbatim string\n"
local f = @"I'm a verbatim string\n"

local g = true; // Bool
```

Listing A.1: Squirrel types: Integer, Float, String, and Bool

Null

The `null` value is a primitive value that represents the empty or non-existent reference. The type Null has exactly one value, called `null`:

```
local a = null
```

Array and Table

Arrays are simple sequence of objects, their size is dynamic and their index starts always from 0. *Tables* are associative containers implemented as pairs of key/value (called a *slot*). See Listing A.2.

```
// Array
local a = ["I'm","an","array"]
local b = [null]
b[0] = a[2];

// Table
local t = {}
local test =
{
    a = 10
    b = "S"
```

```
    b = function(a) { return a+1; }
}

// Compound example: Array and Table
local table = {
        a = "10"
        subtable = {
                array = [1, 2, 3]
        },
}
```

Listing A.2: Squirrel types: Array and Table

Function

Functions are first class values like integer or strings and can be stored in table slots, local variables, arrays and passed as function parameters. Functions can be implemented in Squirrel or in a native language with calling conventions compatible with ANSI C.

Functions are declared through the function expression:

```
local f = function(a, b, c) {
  return a + b - c;
}
```

or with a syntactic sugar

```
function ciao (a, b, c) {
  return a + b - c;
}
```

The function call is evaluated after the argument list.

The Squirrel language has several other features, language elements allowing the creation of flexible and powerfull scripts. For further information on e.g. Class, Class instance, Generator, Userdata, Thread, and Weak References see the official Squirrel reference. Scripting of ParCompMark does not use these types.

A.3 Statements

A Squirrel program is a simple sequence of statements. The end of a statement is indicated by the semicolon mark or a newline character:

```
program := stats
stats := stat [';'|'\n'] stats
```

Block

A sequence of statements delimited by curly brackets (`{ }`) is called a block; a block is a statement itself:

```
stat := '{' stats '}'
```

Expressions

In Squirrel every expression is also allowed as a statement, if so the result of the expression is thrown away:

```
stat := exp
```

Squirrel implements two kinds of *assignment expression*: the normal assignment (`=`) and the and the "new slot" assignment (`<-`):

```
exp := derefexp '=' exp
exp := derefexp '<-' exp
```

For example:

```
a = 10;
b <- 10;
```

The *new slot* expression allows to add a new slot into a table. If the slot already exists in the table the expression behaves like a *normal* assignment. However, the normal assignment expression will fail if the slot does not exist.

Squirrel implements the operator expressions in a C-like manner. These operators are the `?:` operator, the standard arithmetic operators (`+`, `-`, `*`, `/`, and `%`), the compact arithmetic operators (`+=`, `-=`, `*=`, `/=`, and `%=`), the increment and decrement operators (`++` and `--`), the relations operators (`<`, `<=`, `==`, `!=`, `>=`, and `>`), the logical operators (`&&`, ||, and `!`), the bitwise operators (`&`, |, `^`, `~`, `<<`, `>>`, and `>>>`[1]), etc. For operator precedence see Table A.2.

Control Flow Statements

Squirrel implements the most common control flow statements: `if`, `while`, `do-while`, `switch-case`, `for`, and `foreach`. `foreach` executes a statement for every element contained in an array, table, class, string or generator:

```
'foreach' '(' [index_id',] value_id 'in' exp ')' stat
```

An example is illustrated in Listing A.3.

[1] unsigned right shift operator

-, , !, typeof, ++, –	highest
/, *, % ...	
+, -	
<<, >>, >>>	
<, <=, >, >=	
==, !=	
&	
^	
|	
&&, in	
||	
?:	
+=, =, -= ...	
,(comma operator)	lowest

Table A.2: Operators precedence in Squirrel

```
local a = [10, 23, 33, 41, 589, 56]
foreach(idx,val in a)
  print("index=" + idx + " value=" + val + "\n");
// or
foreach(val in a)
  print("value=" + val + "\n");
```

Listing A.3: For-each loop example

Exception Handling

The `try` statement encloses a block of code in which an exceptional condition can occur, such as a runtime error or a `throw` statement. The `catch` clause provides the exception handling code. When a `catch` clause catches an exception, its id is bound to that exception:

```
stat:= 'try' stat 'catch' '(' id ')' stat
stat:= 'throw' exp
```

Appendix B

Rendering Engine (RE) methods

These rendering engine methods can be called as `RE.methodname(<arguments>)` from `initProc` and `runningProc` methods of the low-level script.

Projection matrix methods

```
void perspective(const double fovy, const double zNear,
                 const double zFar);
```

sets the perspective projection (aspect ratio is automatically calculated).

```
void ortho2D(const double left, const double right,
             const double bottom, const double top);
```

defines a 2D orthographic projection.

Modelview matrix methods

```
void pushModelView();
```

pushes the modelview matrix on the matrix stack.

```
void popModelView();
```

pops the modelview matrix from the matrix stack.

```
void translate(const double x, const double y, const double z);
```

translates the modelview matrix.

```
void rotate(const double angle, const double x, const double y,
            const double z);
```

rotates the modelview matrix.

```
void scale(const double x, const double y, const double z);
```

scales the modelview matrix.

```
void setCameraPosition(const double eyeX, const double eyeY,
                       const double eyeZ);
```

sets camera position.

```
void setCameraTarget(const double centerX, const double centerY,
  const double centerZ);
```

sets the position of camera target.

```
void setCameraUpVector(const double upX, const double upY,
                       const double upZ);
```

sets the up-vector of the camera.

Viewport methods

```
viewport(const double left, const double top,
         const double width, const double height);
```

sets the viewport with window-relative coordinates (0.0 .. 1.0).

Triangle-mesh based render methods

```
void drawTriangle();
```

draws a triangle width (0,1,0), (0,0,0), (1,0,0) vertices, respectively.

```
int generateRandomTriangles(const int dimension, const int count);
```

generates certain number of 2D or 3D triangles with random coordinates. The vertex coordinates are in the [0..1] interval. Use `translate()`, `rotate()`, and `scale()` methods to modify this square (2D) or cube (3D).

```
void renderObject(const unsigned handle, const bool useVertexArrays);
```

renders the object with the specified handle.

```
void renderObjectTriangles(const unsigned handle,
                           const bool useVertexArrays,
                           const unsigned triangleCount);
```

renders the specified number of triangles from an object with the given handle.

GLU primitives

```
void drawSphere(const double radius, const int slices,
                const int stacks);
```

draws a sphere.

```
void drawCylinder(const double baseRadius, const double topRadius,
  const double height, const int slices,
                  const int stacks);
```

draws a cylinder.

```
void drawDisk(const double innerRadius, const double outerRadius,
              const int slices, const int loops);
```

draws a disk.

GLUT primitives

```
void drawTeapot(const double size);
```

draws a teapot.

```
void drawCube(const double size);
```

draws a cube.

```
void drawTorus(const double innerRadius, const double outerRadius,
               const int nsides, const int rings);
```

draws a torus.

```
void drawDodecahedron(const double size);
void drawOctahedron(const double size);
void drawTetrahedron(const double size);
void drawIcosahedron(const double size);
```

draws a dodecahedron, an octahedron, a tetrahedron, or an icosahedron, respectively.

Lighting methods

```
void setAmbientLight(const double red, const double green,
                     const double blue, const double alpha);
```

sets global ambient light parameters.

```
void removeLightSources();
```

removes all light sources.

```
int addLightSource();
```

adds a light source into the scene.

```
void setLightSourcePosition(const int light, const double x,
                            const double y, const double z);
```

sets the position of the specified light source.

```
void setLightSourceDiffuse(const int light, const double red,
                           const double green, const double blue,
                           const double alpha);
```

sets diffuse light parameters for the specified light source.

```
void setLightSourceSpecular(const int light, const double red,
                            const double green, const double blue,
                            const double alpha);
```

sets specular light parameters for the specified light source.

Material methods

```
void setColor(const double red, const double green,
              const double blue, const double alpha);
```

sets drawing color.

```
void setAmbientMaterial(const double red, const double green,
                        const double blue, const double alpha);
```

sets ambient material color.

```
void setDiffuseMaterial(const double red, const double green,
                        const double blue, const double alpha);
```

sets diffuse material color.

```
void setSpecularMaterial(const double red, const double green,
                         const double blue, const double alpha,
                         const int shininess);
```

sets specular material color.

```
void setDrawStyle(const unsigned drawStyle);
```

sets the drawing style. Possible values are:

- `OpenGLRenderingEngine.NONE` draw nothing
- `OpenGLRenderingEngine.POINT` draw points
- `OpenGLRenderingEngine.WIRE` draw lines
- `OpenGLRenderingEngine.FILL` draw polygons

```
void setBackCulling(const bool isBackCulling);
```

specifies whether back-facing facets are culled.

```
void setBlending(const bool isOn);
```

turns on/off blending.

Display list methods

```
unsigned createDisplayList();
```

creates a new display list. This will be the active list.

```
void finishDisplayList();
```

finishes the active display list.

```
void executeDisplayList(const unsigned displayList);
```

executes the specified display list.

Custom renderer handling

```
Renderer * createCustomRenderer(const char *rendererName);
```

creates a custom renderer from a renderer plugin.

Appendix C

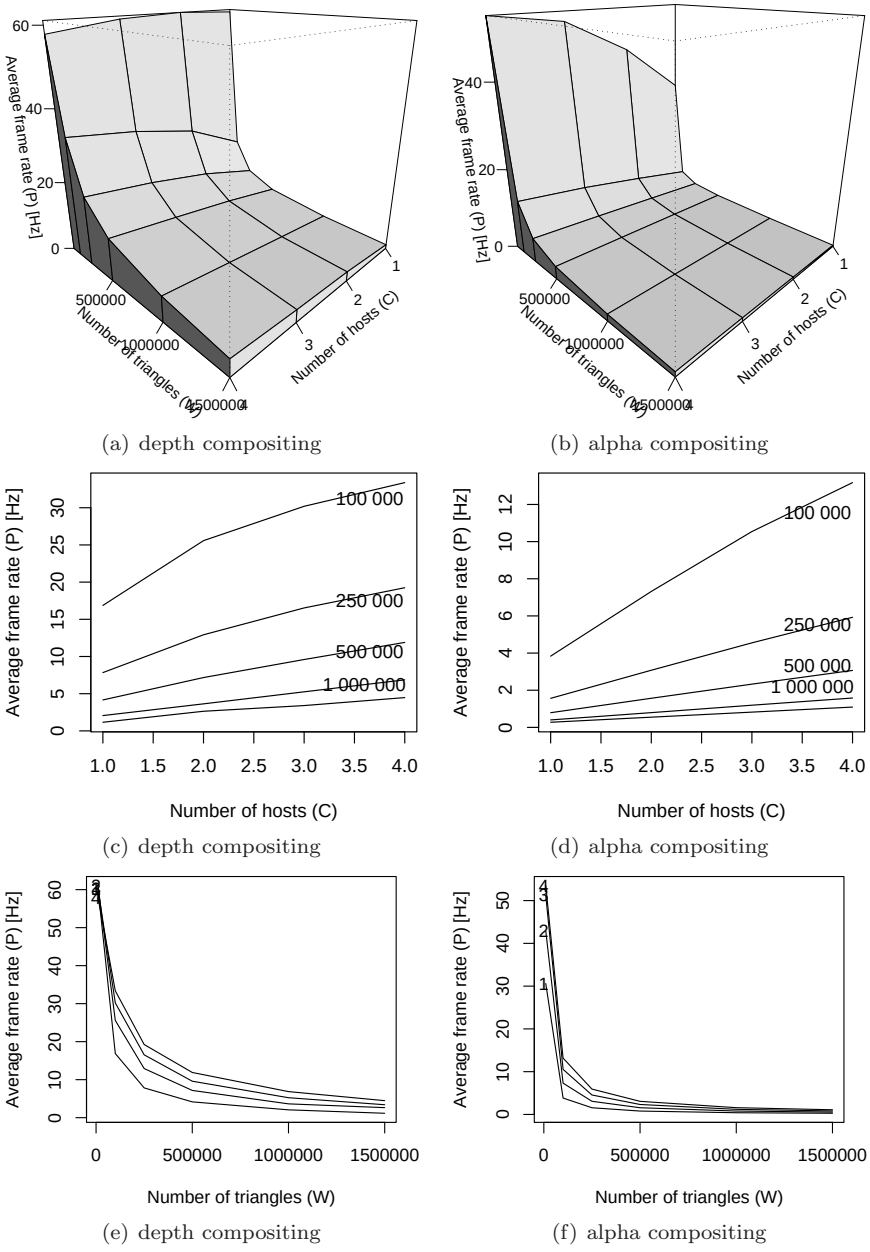
Experimental Results with the "Brute Force" Triangle Rendering Benchmark

This appendix contains results of the sample "brute force" triangle rendering bench test described in chapter 3.

C.1 Measuring with Different Parameter Settings

Figure C.1-C.3 Compositing benchmark, results for the average frame rate as *performance* (P) in the function of the host count (*computing power*, C) and the total number of triangles (*work*, W) (a, b). *Performance scalability* results plotting the frame rate in the function of the host count for different number of triangles (c, d). The frame rate is shown as a function of the triangle count for different number of rendering-compositing nodes (e, f). The performance was measured for *different parameter settings*. The number of rendering-compositing nodes varied from 1 to 4, 1 to 8, and 1 to 16, while the total number of triangles was $10^4, 10^5, 2.5{\cdot}10^5, 5{\cdot}10^5, 10^6, 1.5{\cdot}10^6$, and $1000, 200, 150, 100, 50, 20$ frames were averaged, respectively. The frame sizes were 800×600 and the compositing operators were *depth compositing* (a,c,e) and *alpha compositing* without depth sorting (b,d,f).

Figure C.4 Comparison of *performance scalability* results plotting the frame rate in the function of the host count for different number of triangles. The number of rendering-compositing nodes varied from 1 to 4, 1 to 8, and 1 to 16, while the total number of triangles was $10^4, 10^5, 2.5{\cdot}10^5, 5{\cdot}10^5, 10^6, 1.5{\cdot}10^6$, and $1000, 200, 150, 100, 50, 20$ frames were averaged, respectively. The frame sizes were 800×600 and the compositing operators were *depth compositing* (a) and *alpha compositing* without depth sorting (b).

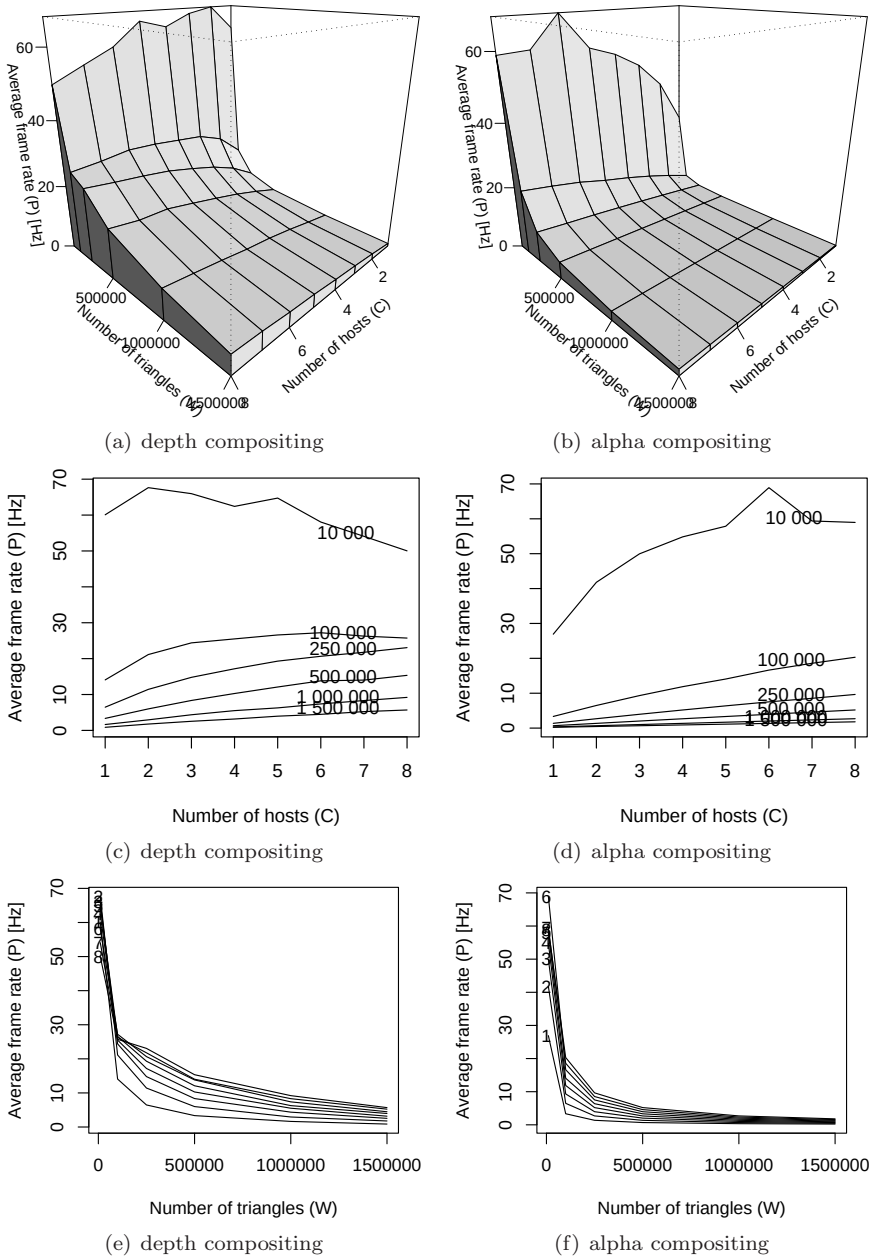


(a) depth compositing (b) alpha compositing

(c) depth compositing (d) alpha compositing

(e) depth compositing (f) alpha compositing

Figure C.1: Compositing benchmark results for the average frame rate as *performance* (P) in the function of the host count (*computing power*, C) and the total number of triangles (*work*, W) (a, b) on a five-node cluster. *Performance scalability* results plotting the frame rate in the function of host count for different number of triangles (c, d). The frame rate in the function of the triangle count for different number of rendering-compositing nodes (e, f).

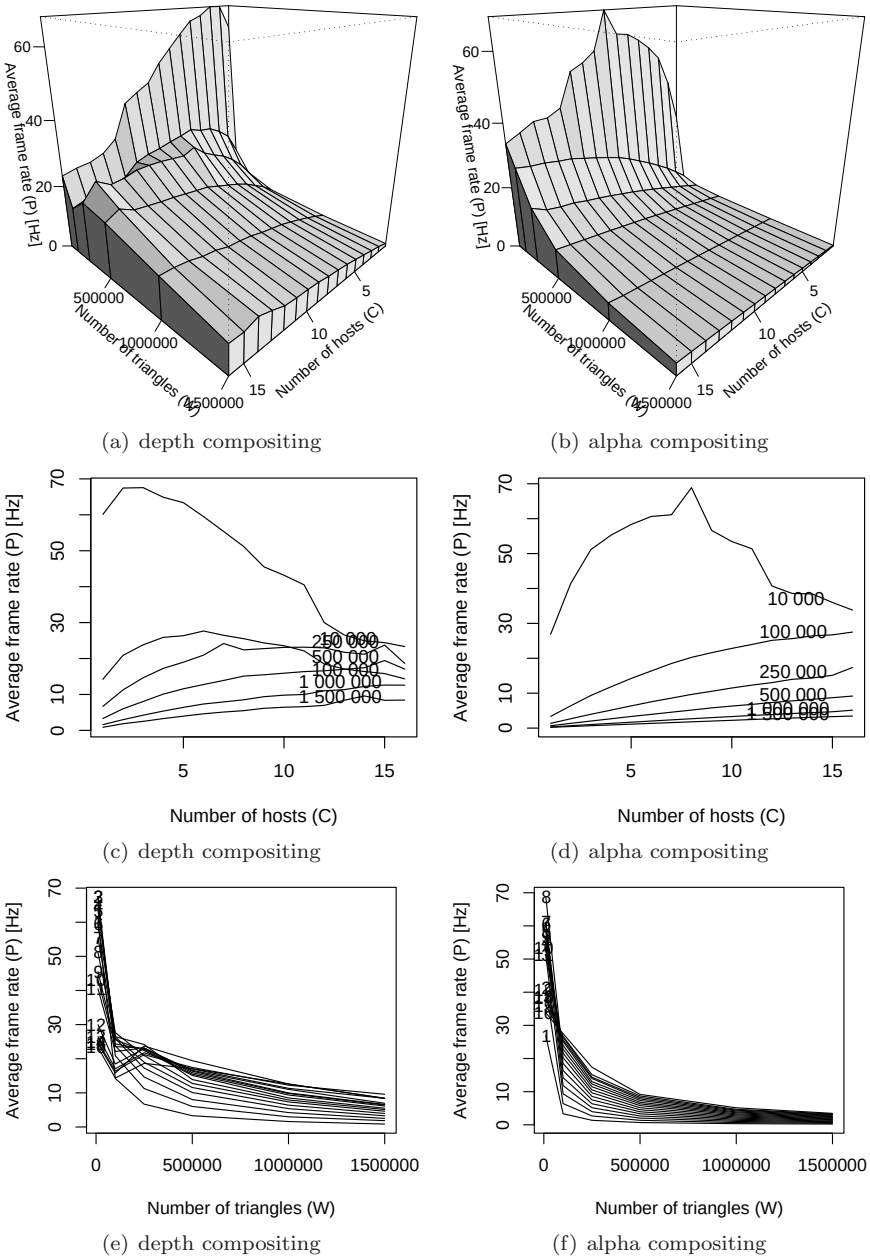


(a) depth compositing (b) alpha compositing

(c) depth compositing (d) alpha compositing

(e) depth compositing (f) alpha compositing

Figure C.2: Compositing benchmark results for the average frame rate as *performance* (P) in the function of the host count (*computing power*, C) and the total number of triangles (*work*, W) (a, b) on a nine-node cluster. *Performance scalability* results plotting the frame rate in the function of the host count for different number of triangles (c, d). The frame rate in the function of the triangle count for different number of rendering-compositing nodes (e, f).

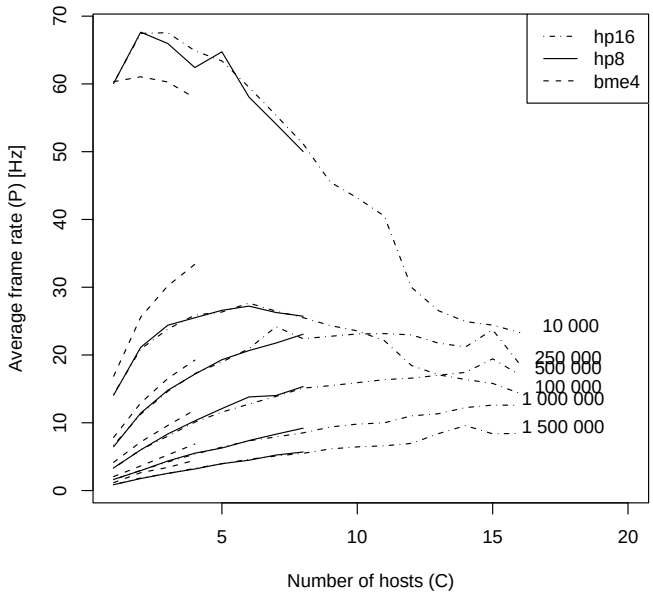


(a) depth compositing (b) alpha compositing

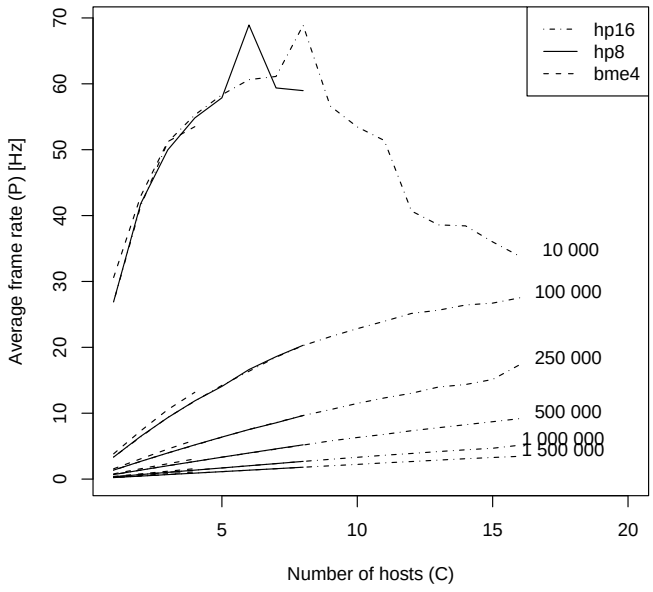
(c) depth compositing (d) alpha compositing

(e) depth compositing (f) alpha compositing

Figure C.3: Compositing benchmark results for the average frame rate as *performance* (P) in the function of the host count (*computing power*, C) and the total number of triangles (*work*, W) (a, b) on a seventeen-node cluster. *Performance scalability* results plotting the frame rate in the function of host count for different number of triangles (c, d). The frame rate in the function of the triangle count for different number of rendering-compositing nodes (e, f).



(a) depth compositing

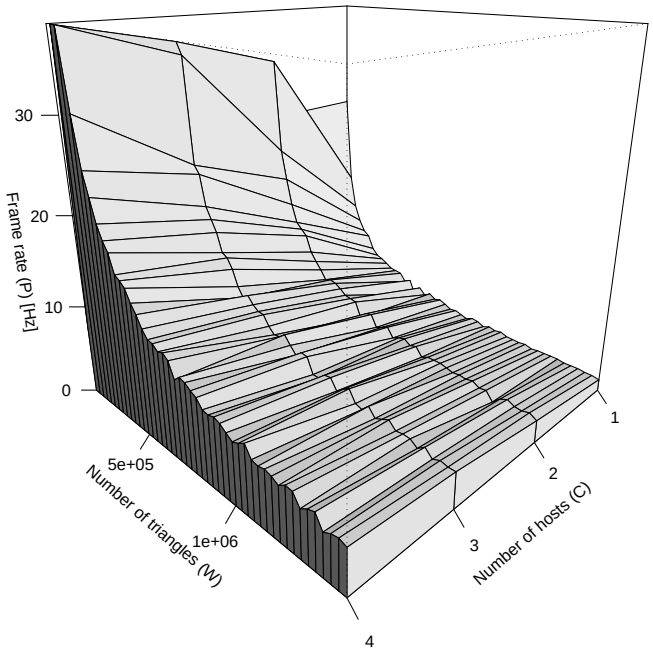


(b) alpha compositing

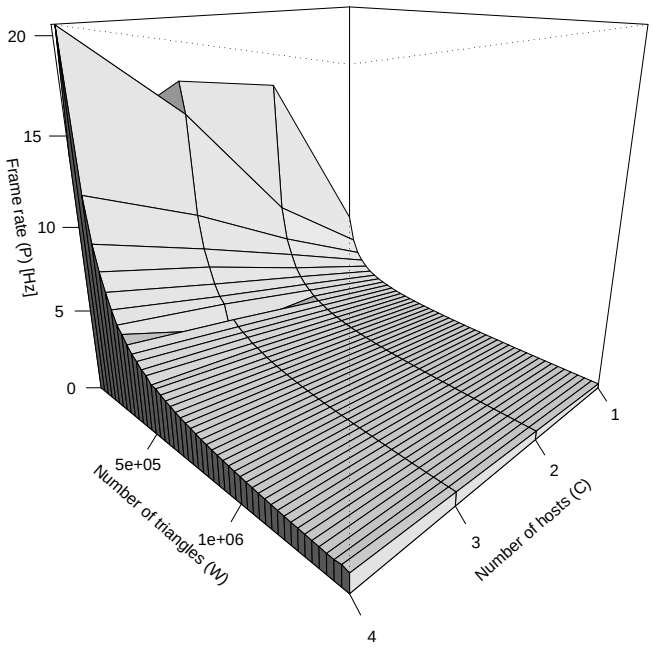
Figure C.4: Comparison of *performance scalability* results plotting frame rate in the function of the host count for different number of triangles. The compositing operators were *depth compositing* (a) and *alpha compositing* without depth sorting (b). The triangle count labels correspond to the curves of a seventeen-node cluster.

C.2 Measuring with Continuous Triangle Count Incrementation

Figure C.5-C.7 Compositing benchmark results for frame rate as *performance* (P) in the function of the host count (*computing power*, C) and the total number of triangles (*work*, W). The performance was measured *continuously increasing the number of triangles* from 10^4 to $1.5 \cdot 10^6$ while rendering 500 frames. 10 consecutive measured values are averaged as plotted values. The number of rendering-compositing nodes varied from 1 to 4, 1 to 8, and 1 to 16, while the compositing operators were *depth compositing* (a) and *alpha compositing* without depth sorting (b).

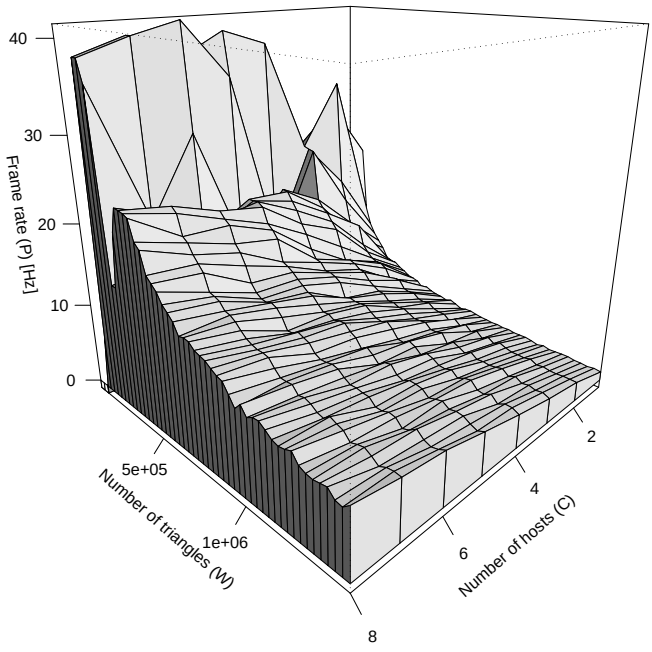


(a) depth compositing

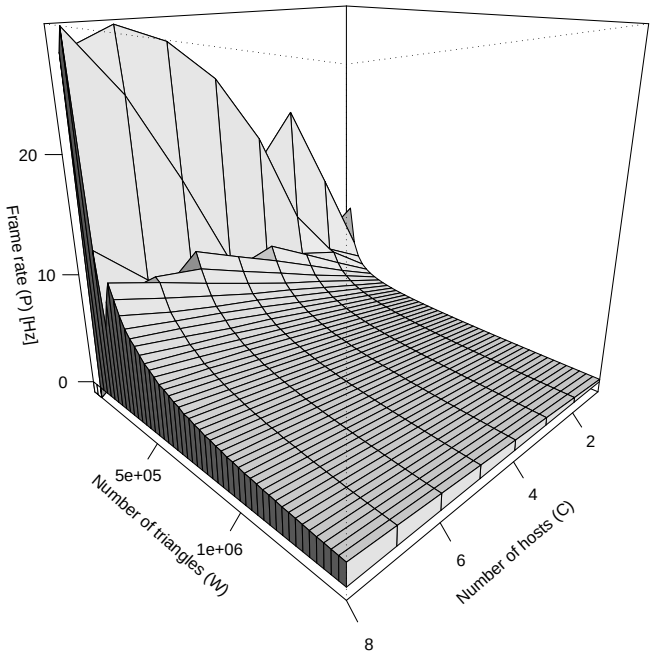


(b) alpha compositing

Figure C.5: Compositing benchmark results for frame rate as *performance* (P) in the function of the host count (*computing power*, C) and the total number of triangles (*work*, W) on a five-node cluster.

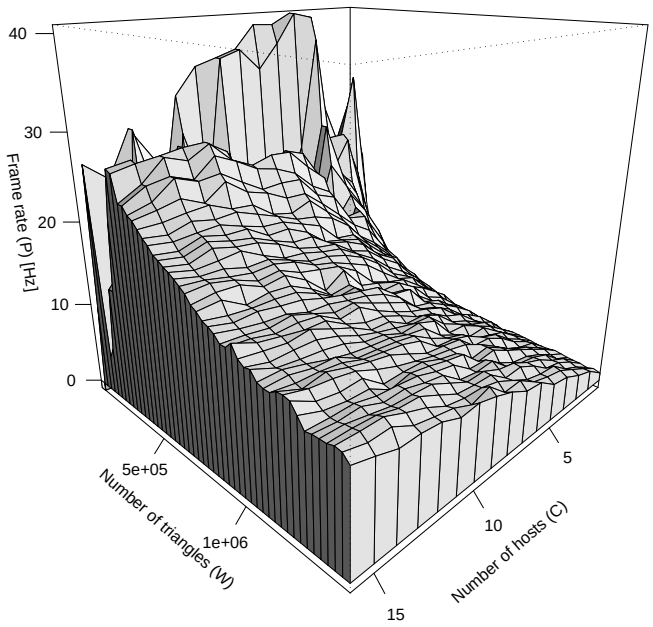


(a) depth compositing

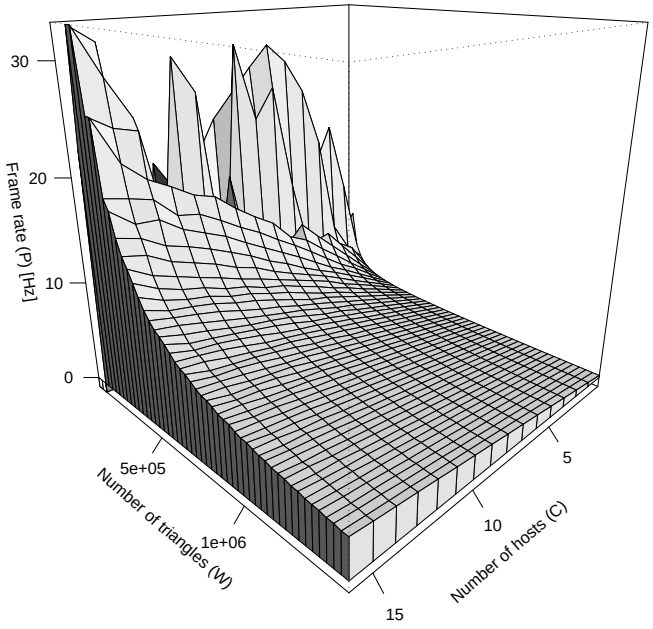


(b) alpha compositing

Figure C.6: Compositing benchmark results for frame rate as *performance* (P) in the function of the host count (*computing power*, C) and the total number of triangles (*work*, W) on a nine-node cluster.



(a) depth compositing

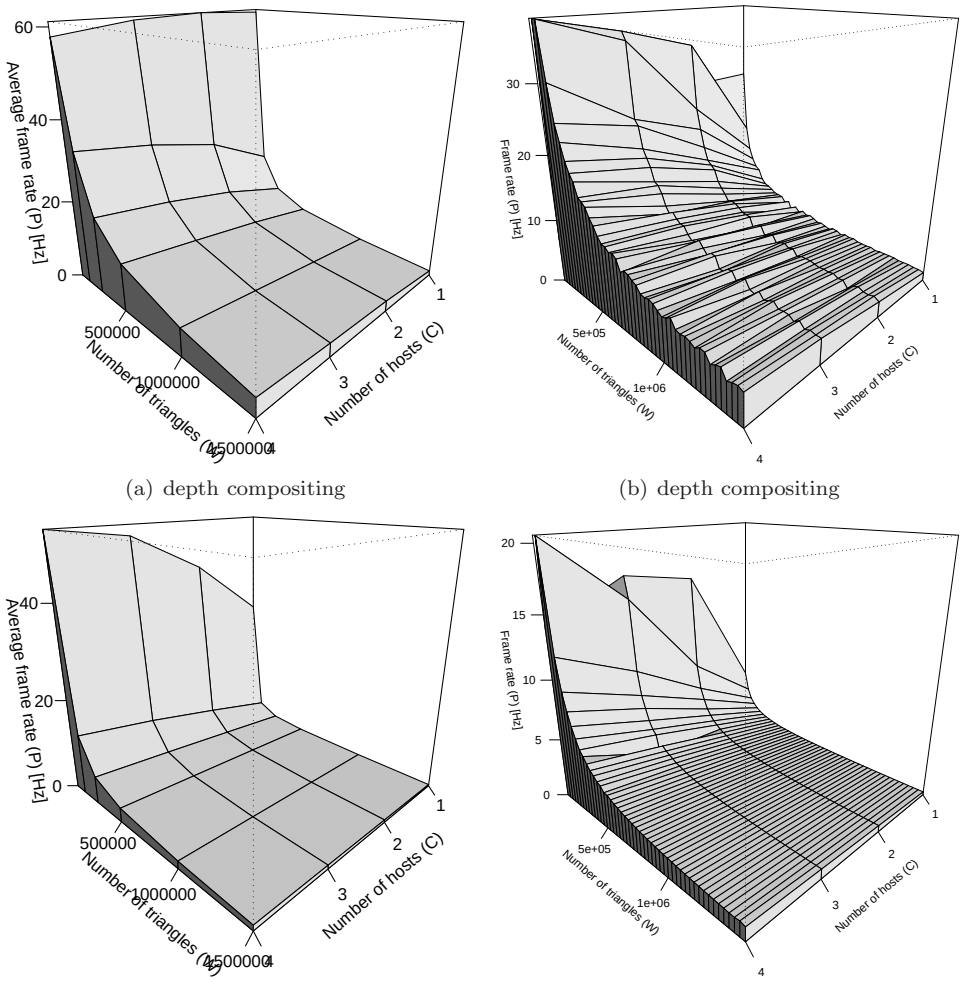


(b) alpha compositing

Figure C.7: Compositing benchmark results for frame rate as *performance* (P) in the function of the host count (*computing power*, C) and the total number of triangles (*work*, W) on a seventeen-node cluster.

C.3 Comparing Different Parameter Settings and Continuous Triangle Count Incrementation

Figure C.8-C.10 Comparing results for *different parameter settings* and for *triangle count incrementation*. The number of rendering-compositing nodes varied from 1 to 4, 1 to 8, and 1 to 16. The total number of triangles was $10^4, 10^5, 2.5 \cdot 10^5, 5 \cdot 10^5, 10^6, 1.5 \cdot 10^6$, and $1000, 200, 150, 100, 50, 20$ frames were averaged respectively in the case of different parameters settings. For the case of continuous triangle count incrementation the total number of triangles was varied from 10^4 to $1.5 \cdot 10^6$ while rendering 500 frames and 10 consecutive measured values are averaged as plotted values. The frame sizes were 800×600 and the compositing operators were *depth compositing* (a, b) and *alpha compositing* without depth sorting (c, d).



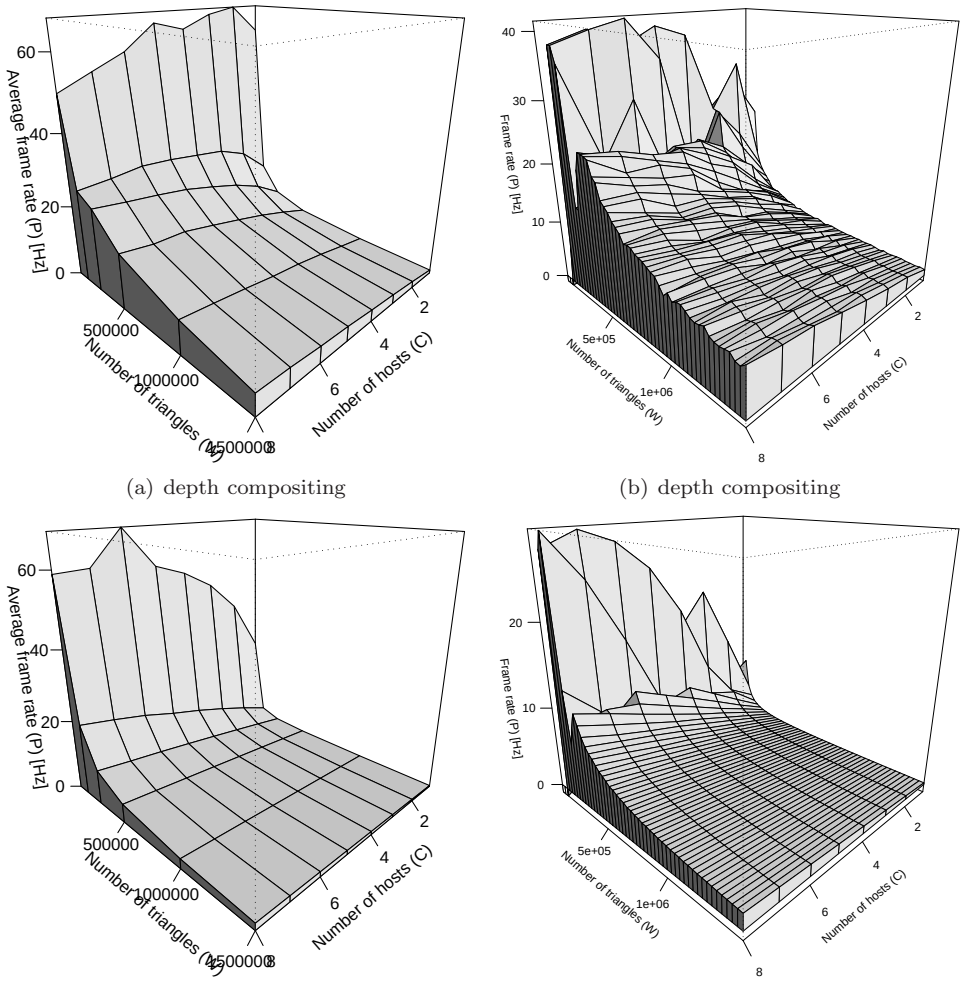
(a) depth compositing

(b) depth compositing

(c) alpha compositing

(d) alpha compositing

Figure C.8: Comparing results for *different parameter settings* (a, c) and for *triangle count incrementation* (b, d) on a four-node cluster. The compositing operators were *depth compositing* (a, b) and *alpha compositing* without depth sorting (c, d).



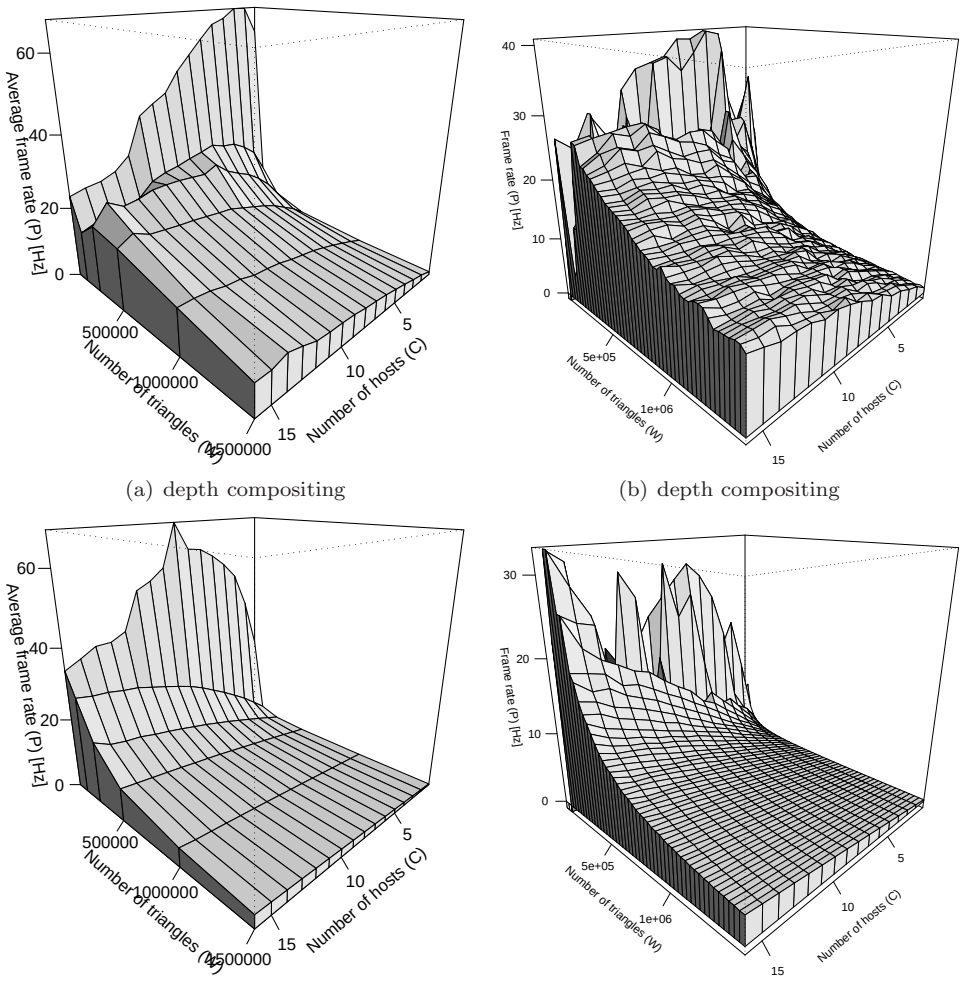
(a) depth compositing

(b) depth compositing

(c) alpha compositing

(d) alpha compositing

Figure C.9: Comparing results for *different parameter settings* (a, c) and for *triangle count incrementation* (b, d) on a nine-node cluster. The compositing operators were *depth compositing* (a, b) and *alpha compositing* without depth sorting (c, d).



(a) depth compositing (b) depth compositing

(c) alpha compositing (d) alpha compositing

Figure C.10: Comparing results for *different parameter settings* (a, c) and for *triangle count incrementation* (b, d) on a seventeen-node cluster. The compositing operators were *depth compositing* (a, b) and *alpha compositing* without depth sorting (c, d).

Appendix D

Renderer Plugin Sample

This is a sample plugin rendering a single triangle per process. For details see Section 2.5.

```
#ifndef __PLUGIN_H__
#define __PLUGIN_H__

#ifndef api_export
#define api_export extern
#endif

// Logging macro for the plugin
// The plugin can connect to the logging system of ParCompMark
#define LOG(msg) logFun(pluginHandle, msg)
typedef void (*logFunType) (const void *, const char *);
api_export logFunType logFun;

// Handle to the plugin object in ParCompMark responsible
// for this plugin instance
// You do not have to deal with it
api_export void *pluginHandle;

#endif // __PLUGIN_H__
```

Listing D.1: plugin.h sample

```
#include <X11/Xlib.h>
#include <GL/glx.h>

#define api_export
#include "plugin.h"
#undef api_export

// Include your renderer header file
// that contains the declaration of the renderer class
#include "renderer.h"

//
// Plugin methods
//

//
```

```
// Plugin methods with int return value should return non-zero on any error.
// The error message can be retrieved using the pcmGetErrorMsg method.
//

// Return the list of the needed shared libraries
extern "C" const char **pcmGetNeededLibs() {
  return Renderer::mNeededLibs;
}

// Setter of plugin handle
extern "C" int pcmSetPluginHandle(void *_pluginHandle) {
  pluginHandle = _pluginHandle;
  return 0;
}

// Setter of logger function
extern "C" int pcmSetLoggerFunction(logFunType _logFun) {
  logFun = _logFun;
  return 0;
}

// Event handler: called when the plugin
// is loaded by ParCompMark
extern "C" int pcmOnLoad() {
  try {
       Renderer::onLoad();
  } catch (const Exception &e) {
       return e.mCode;
  }

  return 0;
}

// Event handler: called when the plugin
// is unloaded by ParCompMark
extern "C" int pcmOnUnload() {
  try {
       Renderer::onUnload();
  } catch (const Exception &e) {
       return e.mCode;
  }

  return 0;
}

// Return the plugin specific error message for the given code
extern "C" const char *pcmGetErrorMsg(const int errorCode) {
  return Exception::getDescription(errorCode);
}

//
// Renderer plugin methods
//

// Return the list of OpenGL externsions to load
extern "C" const char **pcmGetNeededOpenGLExts() {
  return Renderer::mNeededOpenGLExts;
}

// Set misc parameter for the given renderer
extern "C" int pcmSetMiscParam(void *_renderer, const char *name, const char *
     value) {
```

```
  try {
       Renderer *renderer = static_cast<Renderer *>(_renderer);
       renderer->setMiscParam(name, value);
  } catch (const Exception &e) {
       return e.mCode;
  }

  return 0;
}

// Set object space bounding box for the given renderer
extern "C" int pcmSetObjectSpaceBoundingBox(void *_renderer, double x0, double
     y0, double z0, double x1, double y1, double z1) {
  try {
       Renderer *renderer = static_cast<Renderer *>(_renderer);
    renderer->setObjectSpaceBoundingBox(x0, y1, z0, x1, y1, z1);
  } catch (const Exception &e) {
       return e.mCode;
  }

  return 0;
}

// Set object space distribution with object IDs
// for the given renderer
extern "C" int pcmSetObjectId(void *_renderer, unsigned objectId) {
  try {
    Renderer *renderer = static_cast<Renderer *>(_renderer);
       renderer->setObjectId(objectId);
  } catch (const Exception &e) {
       return e.mCode;
  }

  return 0;
}

// Set screen space bounding framelet for the given renderer
extern "C" int pcmSetScreenSpaceFramelet(void *_renderer, double u0, double v0
    , double u1, double v1) {
  try {
       Renderer *renderer = static_cast<Renderer *>(_renderer);
       renderer->setScreenSpaceFramelet(u0, v0, u1, v1);
  } catch (const Exception &e) {
       return e.mCode;
  }

  return 0;
}

// Event handler called when a renderer is about to be created
extern "C" void *pcmOnCreateRenderer(Display *display, Window window,
     XVisualInfo *visualInfo, GLXContext glxContext) {
   return (void *) new Renderer(display, window, visualInfo, glxContext);
}

// Event handler: called when a window of a renderer is resized
extern "C" int pcmOnResize(void *_renderer, unsigned width, unsigned height) {
  try {
       Renderer *renderer = static_cast<Renderer *>(_renderer);
       renderer->onResize(width, height);
  } catch (const Exception &e) {
       return e.mCode;
```

```
  }

  return 0;
}

// Event handler: called when a renderer starts a frame
extern "C" int pcmOnRender(void *_renderer, double time, unsigned frame) {
  try {
        Renderer *renderer = static_cast<Renderer *>(_renderer);
        renderer->onRender(time, frame);
  } catch (const Exception &e) {
        return e.mCode;
  }

  return 0;
}

// Event handler: called when a renderer is destroyed
extern "C" int pcmOnDestroyRenderer(void *_renderer) {
  try {
        Renderer *renderer = static_cast<Renderer *>(_renderer);
        delete renderer;
  } catch (const Exception &e) {
        return e.mCode;
  }

  return 0;
}
```

Listing D.2: plugin.cpp sample

```
#ifndef __RENDERER_H__
#define __RENDERER_H__

#include <X11/Xlib.h>
#include <GL/glx.h>

#include <map>
#include <string>

// Place here your own excpetion handler
#include "exception.h"

using namespace std;

class Renderer {

   // Hashmap for misc parameters
   map<string, string> mMiscParams;

  public:
   // Needed shared libraries and OpenGL extensions
   static const char *mNeededLibs[];
   static const char *mNeededOpenGLExts[];

   // Constructor and destructor
   Renderer(Display *display, Window window, XVisualInfo *visualInfo,
        GLXContext glxContext);
   ~Renderer() throw (Exception);

   // Plugin load/unload handlers
   static void onLoad() throw (Exception);
   static void onUnload() throw (Exception);

   // Setter of misc params
   void setMiscParam(const char *name, const char *value) throw (Exception);

   // Decomposition setters
   void setObjectSpaceBoundingBox(double x0, double y0, double z0, double x1,
         double y1, double z1) throw (Exception);
   void setObjectId(unsigned id) throw (Exception);
   void setScreenSpaceFramelet(double u0, double v0, double u1, double v1)
        throw (Exception);

   // Rendering based handlers
   void onResize(unsigned width, unsigned height) throw (Exception);
   void onRender(double time, unsigned frame) throw (Exception);
};
#endif // __RENDERER_H__
```

Listing D.3: renderer.h sample

```
#include "renderer.h"
#include "plugin.h"

#include <GL/gl.h>
#include <GL/glu.h>

#include <math.h>
```

```
// Define our requirements
const char *Renderer::mNeededLibs[] = {"libm", "libGL", "libGLU", 0};
const char *Renderer::mNeededOpenGLExts[] = {"GL_EXT_texture3D", 0};

Renderer::Renderer(Display *display, Window window, XVisualInfo *visualInfo,
    GLXContext glxContext) {
}

Renderer::~Renderer() throw (Exception) {
}

void Renderer::onLoad() throw (Exception) {
}

void Renderer::onUnload() throw (Exception) {
}

void Renderer::setMiscParam(const char *name, const char *value) throw (
    Exception) {
  // By default, do nothing just push the parameter
  mMiscParams[name] = value;
}

void Renderer::setObjectSpaceBoundingBox(double x0, double y0, double z0,
    double x1, double y1, double z1) throw (Exception) {
}

void Renderer::setObjectId(unsigned id) throw (Exception) {
}

void Renderer::setScreenSpaceFramelet(double u0, double v0, double u1, double
    v1) throw (Exception) {
}

void Renderer::onResize(unsigned width, unsigned height) throw (Exception) {
}

void Renderer::onRender(double time, unsigned frame) throw (Exception) {

  // Create time based color values
  double s = 0.5*sin(time)+0.5;
  double c = 0.5*cos(time)+0.5;

  if(mMiscParams["invColours"] == "yes") {
       glBegin(GL_TRIANGLES);
         glColor3f(c, 0.0, 0.0);    glVertex2f(5.0, 0.0);
         glColor3f(0.0, s, 0.0);    glVertex2f(0.0, 0.0);
         glColor3f(0.0, 0.0, c);    glVertex2f(0.0, 5.0);
       glEnd();
  }
  else {
       glBegin(GL_TRIANGLES);
         glColor3f(s, 0.0, 0.0);    glVertex2f(5.0, 0.0);
         glColor3f(0.0, c, 0.0);    glVertex2f(0.0, 0.0);
         glColor3f(0.0, 0.0, s);    glVertex2f(0.0, 5.0);
       glEnd();
  }
}
```

Listing D.4: renderer.cpp sample

Appendix E

Sample Post-Processing Scripts

These scripts were used to generate plot for Appendix C.

E.1 Bash script

```
#!/bin/bash

function process_impl() {
  prefix=$1  # File prefix
  compop=$2  # Compositing operator

  # Do XSL Transformation
  xalan -in ${prefix}-tri-host.xml -xsl CW-field-${compop}.xslt -out ${prefix
      }-CW-field-${compop}.csv

  # Transform list to matrix form
  ./sortmatrix < ${prefix}-CW-field-${compop}.csv > input.csv

  # Create plots
  R --no-save --slave < CW-field.r
  ps2eps < output.ps > ${prefix}-CW-field-${compop}.eps

  R --no-save --slave < scal-i.r
  ps2eps < output.ps > ${prefix}-scal-i-${compop}.eps

  R --no-save --slave < scal-ii.r
  ps2eps < output.ps > ${prefix}-scal-ii-${compop}.eps
}

function process() {
  # Do for both operators
  process_impl $1 de
  process_impl $1 al
}

function compare_impl() {
  compop=$1   # Compositing operator

  # Transform lists to matrix forms
```

```
  ./sortmatrix < bme-CW-field-${compop}.csv > input-bme.csv
  ./sortmatrix < hp8-CW-field-${compop}.csv > input-hp8.csv
  ./sortmatrix < hp16-CW-field-${compop}.csv > input-hp16.csv

  # Create plot
  R --no-save --slave < scal-i-compare.r
  ps2eps < output.ps > scal-i-compare-${compop}.eps
}

function compare() {
  # Do for both operators
  compare_impl de
  compare_impl al
}

# Create plots for each cluster
process bme
process hp8
process hp16

# Create comparing plot
compare
```

Listing E.1: Example .sh file for postprocessing

E.2 XSLT

```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:def="http://www.it2.bme.hu"
  xmlns:info="http://www.it2.bme.hu"
  xmlns:ref="http://www.it2.bme.hu"
  xmlns:stat="http://www.it2.bme.hu" version="1.0">
<xsl:output method="text" />

  <xsl:template match="info:results">
    <xsl:text>hosts;triangle_count;avg_fps;</xsl:text>
    <xsl:apply-templates select="info:execution[info:parameters/table/slot[@name='
        compositeMode']/text() = 'al']"/>
  </xsl:template>

  <xsl:template match="info:execution">
    <xsl:value-of select="info:parameters/table/slot[@name='renderHostCount']/text()
        "/><xsl:text>;</xsl:text>
    <xsl:value-of select="info:parameters/table/slot[@name='triangleCount']/text()"/
        ><xsl:text>;</xsl:text>
    <xsl:value-of select="info:host/info:node/info:process[@name='Triangle renderer
        ']/stat:frame[last()]/@average-fps"/>
  </xsl:template>
</xsl:stylesheet>
```

Listing E.2: Example .xslt file for postprocessing

E.3 R Plotting Scripts

```
# Load lattice package
require(lattice)

# Set output format
postscript(file="output.ps", bg="transparent",paper="special",width=6.23,
     height=6.23,horizontal=FALSE)

# Input data
d <- read.csv("input.csv", header=FALSE, sep=";", quote="\"", dec=".")
z <- data.matrix(d)

# Get data and axes
x <- z[1,2:ncol(z)]
y <- z[2:nrow(z),1]
z <- z[2:nrow(z),2:ncol(z)]

# Add falldown values
z0 <- 0
z <- rbind(cbind(z, z0), z0)
x <- c(x, max(x) + 1e-9)
y <- c(y, max(y) + 1e-9)

# Create plot
persp(y, x, z, theta = 135, phi = 20, scale = TRUE,
     ltheta = -120, shade = 0.4, border = NULL, box = TRUE,
     xlab = 'Number of hosts (C)', ylab = 'Number of triangles (W)',
     zlab = 'Average frame rate (P) [Hz]',
     ticktype = 'detailed', nticks = 4
)
```

Listing E.3: CW-field.r

```
# Set output format
postscript(file="output.ps", bg="transparent",paper="special",width=4.15,
     height=4.15,horizontal=FALSE)

# Load data
d <- read.csv("input.csv", header=FALSE, sep=";", quote="\"", dec=".")
z <- data.matrix(d)

# Drop first column (10000 triangles)
dropcol<-1

# Get X, Y, and Z vectors from the data
x <- z[1,(2+dropcol):ncol(z)]
y <- z[(2+0):nrow(z),1]
z <- z[(2+0):nrow(z),(2+dropcol):ncol(z)]


# Create plot
matplot(y, z,
       xlab = 'Number of hosts (C)', ylab = 'Average frame rate (P) [Hz]',
       type = 'l', col = "black", lwd = 1.0, lty = 1
)


# Add text
```

```
text(0.9*max(y), z[nrow(z)-1,1:(ncol(z)-1)] +
  min(nrow(z),1:(ncol(z)-1)), format(x[1:4], sci=FALSE, big.mark=' '))
```

Listing E.4: scal-i.r

```
# Set output format
postscript(file="output.ps", bg="transparent",paper="special",width=4.15,
     height=4.15,horizontal=FALSE)

# Load data
d <- read.csv("input.csv", header=FALSE, sep=";", quote="\"", dec=".")
z <- data.matrix(d)

x <- z[1,2:ncol(z)]
y <- z[2:nrow(z),1]
z <- z[2:nrow(z),2:ncol(z)]
z <- aperm(z)

# Create plot
matplot(x, z,
       xlab = 'Number of triangles (W)', ylab = 'Average frame rate (P) [Hz]',
       type = 'l', col = "black", lwd = 1.0, lty = 1
)

# Add text
text(min(y), z[1,1:ncol(z)], format(y, sci=FALSE, big.mark=" "))
```

Listing E.5: scal-ii.r

```
# Set output format
postscript(file="output.ps", bg="transparent",paper="special",width=6.23,
     height=6.23,horizontal=FALSE)

# Load data
loadcsv <- function(filename) {
  d <- read.csv(filename, header=FALSE, sep=";", quote="\"", dec=".");
  data.matrix(d)
}

# First data set
z <- loadcsv("input-hp16.csv"); data <- z[2:nrow(z),2:ncol(z)]; maind <- data
x <- z[1,2:ncol(z)]
y <- z[2:nrow(z),1]
DATA <- data

# Append second data set
z <- loadcsv("input-hp8.csv"); data <- z[2:nrow(z),2:ncol(z)]
DATA <- cbind(DATA, rbind(data, array(NA,
            c(nrow(DATA)-nrow(data),ncol(data)))))

# Append third data set
z <- loadcsv("input-bme.csv"); data <- z[2:nrow(z),2:ncol(z)]
DATA <- cbind(DATA, rbind(data, array(NA,
            c(nrow(DATA)-nrow(data),ncol(data)))))

# Append empty lines set
DATA <- rbind(DATA, array(NA, c(4, ncol(DATA))))
```

```
# Create plot
matplot(c(y,(max(y)+1):(max(y)+4)), DATA,
       xlab = 'Number of hosts (C)', ylab = 'Average frame rate (P) [Hz]',
       type = 'l', col = "black", lwd = 1.0,
       lty = c(4,4,4,4,4,4,1,1,1,1,1,1,2,2,2,2,2,2)
)

# Add text
text(1.1*max(y), maind[nrow(maind)-0,1:ncol(maind)] +
  min(nrow(maind),1:ncol(maind)), format(x, sci=FALSE, big.mark=' '))

# Add Legend
legend("topright", c("hp16", "hp8", "bme4"), bg="white", lty = c(4,1,2), lwd =
      1.0)
```

Listing E.6: scal-i-compare.r

Appendix F

ParaComp Calls in ParCompMark

Table F.1 illustrates the location of the ParaComp API calls in the code of ParCompMark. Note, the `pcGetErrorString` is not in the table. It is called after each ParaComp call to verify the success.

Name	File (.cpp)	Class	Function
pcSystemInitialize()	PCMApplication	Application	initialize()
pcSessionCreate()	PCMApplication	Application	initialize()
pcContextCreateMaster()	PCMContext	Context	initialize()
pcContextCreate()	PCMContext	Context	initialize()
pcContextDestroy()	PCMContext	Context	finalize()
pcQueryExtension()	PCMContext	Context	initialize()
pcContextSync()	PCMContext	Context	initialize()
pcContextSetInteger()	PCMProcess	Process	initPC()
pcFrameBegin()	PCMProcess	Process	task()
pcFrameAddFramelet()	PCMProcess	Process	task()
pcFrameAddGLFrameletEXT()	PCMProcess	Process	task()
pcFrameEnd()	PCMProcess	Process	task()
pcFrameResultChannel()	PCMProcess	Process	task()
pcContextGetInteger()	PCMContext PCMProcess	Context Process	initialize() gatherStatistics()
pcSystemGetInteger()	PCMHostInfo	HostInfo	refreshData()
pcSystemGetString()	PCMHostInfo	HostInfo	refreshData()

Table F.1: ParaComp Calls in ParCompMark

Index

Bibliography

[1] A. Demichelis. Squirrel, The Programming Language, 2003-2006. `http://squirrel-lang.org/`.

[2] T. Duff. Compositing 3-D rendered images. In *SIGGRAPH '85: Proceedings of the 12th annual conference on Computer graphics and interactive techniques*, pages 41–44, New York, NY, USA, 1985. ACM Press.

[3] Hewlett Packard. *HP Scalable Visualization Array Parallel Compositing Library Reference Guide*, 2007.

[4] W. M. Hsu. Segmented ray casting for data parallel volume rendering. In *PRS '93: Proceedings of the 1993 symposium on Parallel rendering*, pages 7–14, New York, NY, USA, 1993. ACM Press.

[5] G. Humphreys, M. Eldridge, I. Buck, G. Stoll, M. Everett, and P. Hanrahan. WireGL: a scalable graphics system for clusters. In *SIGGRAPH '01: Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pages 129–140, New York, NY, USA, 2001. ACM Press.

[6] W. J. L. and H. R. E. A proposal for a sort-middle cluster rendering system. In *Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications, 2003. Proceedings of the Second IEEE International Workshop*, pages 36– 38, 2003.

[7] T.-Y. Lee, C. S. Raghavendra, and J. B. Nicholas. Image Composition Schemes for Sort-Last Polygon Rendering on 2D Mesh Multicomputers. *IEEE Transactions on Visualization and Computer Graphics*, 2(3):202–217, 1996.

[8] K. L. Ma, J. S. Painter, C. D. Hansen, and M. F. Krogh. Parallel Volume Rendering using Binary-Swap Compositing. *IEEE Computer Graphics and Applications*, 14(4):59–68, 1994.

[9] S. Molnar, M. Cox, and D. Ellsworth. A sorting classification of parallel rendering. *IEEE Computer Graphics and Applications*, 14(4):23–32, 1994.

[10] C. Mueller. The sort-first rendering architecture for high-performance graphics. In *Symposium on Interactive 3D Graphics: Proceedings of the 1995 symposium on Interactive 3D graphics*, pages 75 – ff, New York, NY, USA, 1995. ACM Press.

[11] U. Neumann. Parallel volume-rendering algorithm performance on mesh-connected multicomputers. In *PRS '93: Proceedings of the 1993 symposium on Parallel rendering*, pages 97–104, New York, NY, USA, 1993. ACM Press.

[12] T. Porter and T. Duff. Compositing digital images. In *SIGGRAPH '84: Proceedings of the 11th annual conference on Computer graphics and interactive techniques*, pages 253–259, New York, NY, USA, 1984. ACM Press.