Master's thesis

GamePro

A high performance visual profiler for real-time applications

by

Michiel Roza

Supervisors:

Dr. ir. H.M.M. van de Wetering (TU/e)
Ir. M.F.A. Schroders (Eximion BV)

Eindhoven, August 14, 2008
Abstract

Video games are software products which purpose is to entertain its players. Video games sometimes exhibit a phenomenon called frame drops, where the performance of a game is temporarily greatly lowered, cause the amount of fun experienced by players to drop dramatically. We thus want to find bottlenecks and causes of frame drops in video games. So called performance profilers inspect performance of software. However, current performance profilers are often slow while collecting data, so the interactive element of video games is lost and recreating events that cause frame drops is hard. Furthermore, temporary drops in performance are invisible and their causes are difficult to find in current performance profilers, because they accumulate information over relatively large periods of time.

This thesis describes the design and implementation of a tool called GamePro. GamePro is a performance profiler which consists of two elements a data logger, and a data presenter part. The data logger is fast during run-time, has a powerful snapshot feature that collects timed data and can inspect native and scripting functions. The presenter part is able to show causes of sudden drops in performance and overall bottlenecks in software to the developers of this software. Because we want the developer to find performance issues quickly, visualization is used.

To determine the usefulness of GamePro it was tested by developers of the Kalydo engine at Eximion BV, by inspecting the performance of their games, and to find causes of frame drops using GamePro. This resulted in games with smoother game play, and thus a better experience for players.
Chapter 1

Introduction

The purpose of video games is to entertain people. The amount of fun experienced by the player is usually lowered whenever the performance of a game fluctuates, because changes in performance can cause the player to lose control over the game. This is especially a problem if the performance suddenly drops dramatically. Developers of video games should therefore write efficient software, especially since video games usually use a lot of resources and touch the limits of what computers can do.

Numerous tools exist for developers to measure performance of software, called performance profilers. These tools show a developer which parts of their software limit the performance of the entire application, also known as bottlenecks. However, no tools exist that can tell a developer what happens during a sudden drop in performance. A developer may want to know what the cause is of such a drop. Furthermore, as video games are usually large projects, many profilers do not clearly show what happens in the software because they do not make extensive use of visualization, but rather very simple techniques such as pie charts.

For convenience, performance of a video game will be calculated in frames per second; which is the amount of consecutive images shown per second. A frame drop is defined as a sudden drop in number of frames per second below a certain threshold. Note that a low frame rate isn’t as bad as a frame drop, as long as it stays above a certain threshold. This threshold varies from person to person, but in general a frame rate of at least 30 frames per second is acceptable, as long as the frame rate does not vary too much.

This master’s thesis describes the creation of a tool which we call GamePro, which enables developers to learn what causes these performance issues. We use visualization to enable the developers to find performance issues efficiently.

1.1 The company

Eximion BV is a company that focuses on the development of online 3D video gaming technology. Online is, in this context, defined as playable inside a web browser; for that reason, a virtual console is being developed called Kalydo, which can run high quality games inside a web browser. One of the company’s goals is to enable older and slower systems to play 3D games in their browser; enhancing performance of their games is therefore important.

Their technology contains tools for rapidly developing new video games, as well as converting existing games to online games. This technology allows their developers to create online games. It will not only be available to the game developers of Eximion BV, but will also be available to external developers of online video games.
Chapter 1. Introduction

Eximion BV also offers services that make it possible for other developers to offer the games online, including payment and administration. The players have an online profile including their achievements (e.g. high scores and the number of completed levels) in the games. This way, they can continue playing their game anywhere. This concept is called persistent web gaming.

1.1.1 The technology

To enable 3D technology in web browsers, Eximion BV is developing its own Kalydo plug-in for most currently available browsers. It is specifically designed to not only provide a link between the web browser and the Kalydo engine created by Eximion BV, but also other engines such as Torque [Tor] or OGRE 3D [OGR].

The Kalydo engine is designed as a versatile 3D game engine. It is written in C++ and its design is based on software agents and event handlers. This design enables developers working on this engine to quickly add their own agents and event handlers. This engine is thus very versatile, where additions and alteration to it can be realized quickly; a concept called rapid development.

In order to enable the concept of rapid development for not only engine developers, but also for game developers, Eximion BV uses the scripting language Lua [Lua] in their engine for game logic. This way, the game developers can test their changes in a matter of seconds, because changes in game logic do not require a time-consuming compilation.

1.2 User questions

Our target users are developers that want to improve the performance of their real-time software, as well as find causes of frame drops. During the course of this project, the main application used to test the tool is the Kalydo engine.

We assume that our target users are experts in understanding the source code of the software that is inspected, so they know how to resolve issues that our tool points out to them.

To determine what the tool should be able to do, developers were asked what they wanted to know about their software, and what was missing from other profilers. The following user questions about real-time software were formulated based on that survey:

1. What are the biggest bottlenecks in the software?
2. What are causes of sudden drops in performance?
3. What parts of the software use large amounts of memory?
4. Does the software contain memory leaks, and if so, what are causes of those leaks?

1.3 Requirements

The goal of this project is to implement a tool that enables developers of software with real-time behavior, and video games in particular, to gain insight on causes of low performance and frame drops.

This tool, which we call GamePro, should be able to collect and present data on program execution. This data should be collected during run-time, and contain enough information to find bottlenecks and causes of frame drops. Furthermore, the performance of the game is not lowered to such an extent, that the game becomes unplayable. We want the developer to quickly learn what parts of their software to inspect and improve in order to eliminate frame drops and general bottlenecks.
1.4. Document structure

We therefore want to use visualization for effective and efficient presentation of the data. We define the requirements of the tool as follows:

1. The process of collecting data on the software should not affect the software’s performance up to a point that the software loses its usefulness. If the inspected software is a video game, its frame rate should not be below one-third\(^1\) of its usual frame rate during collection of data.

2. The tool should be able to at least inspect software that is compilable using Microsoft Visual C++ 2005 [MSV].

3. The developer should not need to alter the source code of its software to collect data from it.

4. The developer should be able to inspect native functions as well as scripting functions.

5. The developer should be able to find bottlenecks in the software using visualization.

6. The developer should be able to find causes of sudden drops in performance using visualization.

7. The developer should be able to find sources of memory leaks using visualization.

8. The developer should be able to selectively analyze parts of the software.

9. The developer should be able to selectively analyze different aspects of the software, namely memory usage and leaks, performance as a whole and fluctuations in performance.

10. The developer should be able to compare part of a run of software in order to e.g. find differences between an interval exhibiting a frame drop, and an interval that does not, as well as two different runs in order to check e.g. whether optimizations work.

1.4 Document structure

The remainder of this document is structured in the following way: chapter 2 to 6 concern the tool, where chapter 2 describes the design of the tool, chapter 3 describes the data model used in detail, chapter 4 describes modifications defined on that data model and chapter 5 describes the visualizations in the tool. Chapter 6 evaluates the tool, and chapter 7 contains a project evaluation and conclusions.

\(^1\)This value was determined together with the Kalydo engine developers. The frame rate of a game engine compiled with all optimizations (the so-called release build) is approximately three times as fast as an engine compiled without these optimizations and incorporating a lot of extra debugging checks (the so-called debug build). The performance of profiling a release build should not be worse than the performance of a debug build.
1.5 List of definitions

**bottleneck**
A part of an application that limits the performance of the entire application.

**child function**
Function \( f \) is a child function of \( g \) if \( f \) gets called by \( g \).

**event**
An event of a run of an application, such as entry of a function, or memory allocation.

**exclusive value**
A value that applies to a vertex (in the context of values in a tree).

**frame drop**
The sudden decrease in performance of a video game.

**inclusive value**
A value that applies to a subtree (in the context of values in a tree).

**machine cycles**
The basic operation performed by a CPU.

**parent function**
Function \( f \) is a parent function of \( g \) if \( g \) gets called by \( f \).

**performance**
The amount of useful work done in a period of time.

**preloading**
The retrieval of data before it is actually needed; as opposed to just-in-time loading.

**real-time software**
Software that is subject to operational deadlines from event to response.

**recursive function**
a function that calls itself, or is eventually called by a child function.

**resource**
An entity of limited availability.

**run**
An execution of an application.

**sampling**
The reduction of a continuous signal to a discrete signal.

**snapshot**
A set of values at a point in time.

**trace**
A list of events representing a thread of a run of an application.

**video game**
Software that involves interaction with a user interface to generate visual feedback.
Chapter 2

Design

This chapter describes several design aspects of the tool regarding

- the type of performance profiling in GamePro and a comparison to other types;
- the global structure of the tool, and
- the design of the separate parts of the tool.

2.1 Performance profilers

Two types of tools exist that inspect software; namely tools that inspect static aspects, and tools that inspect dynamic aspects. Tools that inspect static software aspects collect information by inspecting the source code of software. An example of such a tool is Doxygen [Dox], a tool that automatically documents classes, their functions and their comments. Tools that collect data on run-time behavior of software are called profilers. An example of a profiler is TraceVis by Pieter Deelen [Dee06], where the dynamic interaction between classes is visualized.

Performance profilers are tools that measure performance of software. Developers of the Kalydo game engine determined that current performance profilers are not suitable for analyzing the performance of a video game, because

- they use a lot of resources. As a result, the profiled video game becomes too slow to retain the interactive element, violating requirement 1, and
- they only show the overall average or total duration of parts of software, but not the behavior over time. As a result, finding causes of frame drops is next to impossible, violating requirement 6.

GamePro is a performance profiler, and as such, it is a tool that inspects dynamic software aspects. We also inspect memory usage and memory leaks, although this is not the main focus of this project.

The following sections describe different design decisions for the design of performance profilers in general, and GamePro in particular.

2.1.1 Online vs. offline

Profilers can basically be subdivided into
Chapter 2. Design

- online profilers, that present collected data during run-time, and
- offline profilers, that consists of a part that collects data, and a part that presents data. The data is not presented during run-time, but communicated between these parts through memory or disk.

Both methods have their advantages and disadvantages. An online profiler can help the user to associate behavior with profiling information; such as the pressing of a button that triggers method calls the user may or may not expect. Examples of online profilers are:

- GlowCode [GC], and
- rtprof [rtp].

An offline profiler may help to improve performance during run-time of the profiled software because no resources are needed to present the data at run-time. It also enables the user to profile full-screen applications: no screen space is needed for presenting the data during run-time. Furthermore, it allows the user to study the data in their own time, without the need for an ‘extra eye’ on the presented data. An example of an offline profilers is AutomatedQA AQtime [AQt].

Whether a profiler is online or offline is not mutually exclusive. A profiler that is both online and offline has the advantage of the user being able to associate behavior with profiling information, and allowing the user to study data in their own time. Its largest disadvantage is the fact that the presenting of data during run-time costs performance. An example of a profiler that are both online and offline is Performance Validator [PV].

GamePro is an offline profiler for reasons of

- performance; an offline tool is overall faster during run-time than an online tool, because online tools force the inspected software and profile tool to share resources. This results in lower performance of both, which could endanger requirement 1;
- ease of design and implementation; this way, it is possible to separate the tool in two separate parts, where the only interface between the two parts is a data structure in memory or on disk;
- the possibility to analyze data for software in any other language, on any other operating system or using another compiler, without rewriting the complete tool. Only the data collecting part needs to be altered, and
- no need for an ‘extra eye’, so the user can focus on the inspected software, and the analysis of the profile data afterwards. Also, it enables the possibility of profiling full-screen applications, which video games often are.

The user running the software (e.g. the tester) can be different than the user analyzing the data. It is even possible to analyze the data on another computer, making it possible to e.g. collect data on a slow computer to check the bottlenecks on slower computers, while analyzing the data on a fast PC for better response of the analysis tool.

2.1.2 Sampling vs. instrumenting

Profilers work by collecting dynamic data about the inspected software during run-time. This can be done using two different methods:

- sampling: sampling profilers check at a predefined time interval which function or stack is active, and
2.2. Global structure

- *instrumenting*: instrumenting profilers alter the machine code in such a way, that the profiler gains control over the software at certain *events*.

Sampling profilers are overall faster due to the fact that overhead needed for collecting data is much less than when using an instrumenting profiler. However, they are less precise than instrumenting profilers because they only check the software at predefined times, meaning that they can e.g. miss function calls that occur in between samples. Instrumenting profilers gain control over the software at all events as entry and exit of a function.

GamePro is an instrumenting profiler because

- sampling profilers are less precise than instrumenting profilers as functions can be called in between samples;
- instrumenting profilers have the possibility to collect and calculate extra information, such as the average duration of a function, standard deviation of the list of durations of calls, number of calls to a function;
- a sampling profiler can not profile memory usage, so user questions 3 and 4 can not be answered.

2.2 Global structure

This section discusses the global design of GamePro.

GamePro is designed to be an offline profiler, meaning that the data collecting part, which we call GamePro Data Logger, and the data presenting part, which we call GamePro Analysis, are separate tools, as can be seen in figure 2.1.

![Figure 2.1: The global design of GamePro including its data flow.](image)

An execution of an application is called a run. A run consists of one or more threads. Inside each thread the following events occur sequentially:

- Function entries;
- Function exits;
- Memory allocation;
- Memory deallocation, and
• Memory reallocation.

A list of these events in sequence is called a trace. GamePro Data Logger is an instrumenting profiler, and as such is designed to collect traces. Note that GamePro Data Logger collects the function entry and exit events for not only native functions, but also include script functions, as per requirement 4.

2.2.1 Data logger

The purpose of GamePro Data Logger is to collect the data needed for the visualization in GamePro Analysis to show valid and useful information.

The data logger is able to capture traces per thread, given that the machine code of the inspected software is modified in a way that allows the data logger to gain control during these events. Note that it is not necessarily so that all events occurring within a run are handled by the data logger; e.g. whenever the developer decides that some part does not need to be instrumented, this is possible, and those events will not be handled as per requirement 8.

For implementation reasons, GamePro Data Logger modifies traces to another data model called profile trees. The trace data model and profile tree data model are presented in chapter 3. Further implementation details about the data logger, including the methods of capturing traces can be found in Appendix A.

2.2.2 Analysis

The purpose of GamePro Analysis is to show the data collected from the inspected software in such a way that the developer of the inspected software quickly finds causes of low performance or frame drops.

The analysis part is designed using the Observer design pattern, sometimes called publish/subscribe. A description of this pattern can be found in [GHJV94].

The principle of the Observer design pattern is to define a one-to-many relationship between objects, where a clear distinction is made between a ‘subject’ object, and observer objects. Whenever the subject object changes, all observers are notified of the change. This is realized by ensuring that the subject keeps track of its observers, and notifies them whenever its state has changed.

In our system, the subject contains profile data of one or more runs, possibly modified in order to simplify visualization, including a selection to focus on parts of software. If the selection changes, our subject object changes, and the observers are thus notified. The observers are visualizations showing different aspects of the data. Due to the design pattern used, visualizations can be easily added or removed, and thus experimentation with several visualizations was easily realized.

In order to enable the developer to selectively inspect different parts of software (as per requirement 8 and 9), GamePro Analysis incorporates a data filter and modifier system, consisting of a pipeline of different modifiers, where the filters and modifiers in the pipeline are selectable by the user. The subject can be altered so the data sent to the observers contain e.g. only elements that contain a memory leak, or filter out all function calls that account for less than a certain percentage of time. This is so the visualizations only show the data the developer desires to see. The way data is modified in GamePro Analysis is presented in chapter 4.

Because we want to create an overall view on parts of software, the subject also contains a global context, selection and focus; so that the selected parts of software, and the visualized data (e.g. memory usage, or duration of parts of software) are the same for every observer. This way, the developer can use different observers to check different aspects of the same data.
Chapter 3

Data model

This chapter describes the data model used by GamePro. Section 3.1 describes the data model of traces, and section 3.2 describes the data model of profile trees, including mathematical definitions of these models. Our data logger uses traces as input, and generates profile trees.

Section 3.3 compares the traces and profile tree model, and section 3.4 contains statistical information about real-life data represented in this model.

3.1 Traces

A list of events captured by the data logger is called a trace. Every thread in a run corresponds to exactly one trace.

We define traces in the following way: A trace $t$ is defined as a list of events $e$. Every event $e$ contains the following information:

- $e.time$, the time stamp of the event in machine cycles, where 0 corresponds with the start of the application.
- $e.type$, the type of the event. Its value is one of the following:
  - $entry$, which represents a call to a function,
  - $exit$, which represents a return from a function,
  - $alloc$, which represents an allocation of memory, and
  - $free$, which represents a deallocation of memory.
- If $e.type$ is entry or exit, it also contains the following property:
  - $e.funcID$, a function identifier (e.g. the address of the function).
- If $e.type$ is alloc or free, it also contains the following properties:
  - $e.allocID$, an identifier, unique for the run of the software which generated this event except for its matching free or alloc event (if it exists).
  - $e.size$, the size of the memory allocated or deallocated.

Note that a reallocation event is operationally the same as an allocation event of the new memory followed by a deallocation event of the old memory; this event will thus be regarded as such.
3.1.1 Well-formedness

We assume that traces are well-formed. The following example illustrates what well-formed means: Suppose, a trace is represented in XML, where each event is represented by a tag. The definition of well-formedness regarding XML can be found in [BPSM+06]. Suppose that entry events are opening tags and exit events are closing tags. In this example, memory events are not taken into account, as the well-formedness of a trace is not affected by them.

Then take for instance a trace of a run of a very simple application containing one thread, where function main is called, which calls the function doStuff, which in term calls sqrt twice.

Converting this trace to well-formed XML gives:

```xml
<main>
  <doStuff>
    <sqrt>
    </sqrt>
    <sqrt>
    </sqrt>
  </doStuff>
</main>
```

The following XML is not well-formed, and thus its corresponding trace is also not well-formed:

```xml
<doStuff><sqrt></doStuff></sqrt>
```

Note that in the remainder of this document, we assume that α and β are traces, and e and e’ are events. Furthermore, we define ε as the empty list, and a concatenation operator α · β. Also, every event e is unique, and is contained in exactly one trace.

A trace α is called well-formed if \(WF(α) = true\) (where we assume that \(e↑.type = entry\) and \(e↓.type = exit\)):

\[
WF(α) = \begin{cases} 
  WF(α_0 \cdot α_1) & \text{if } α = α_0 \cdot e \cdot α_1 \land e.type \notin \{entry, exit\} \\
  WF(α_0 \cdot α_1) & \text{if } α = α_0 \cdot e↑ \cdot e↓ \cdot α_1 \land e↑.funcID = e↓.funcID \\
  true & \text{if } α = ε \\
  false & \text{otherwise}
\end{cases}
\] (3.1)

Considering the way we obtain traces, they are always well-formed except in the case of exceptions, where a function can be exited in an abnormal way, bypassing the instrumentation. GamePro Data Logger takes exceptions into account by adding exactly the function exit events needed for the trace to be well-formed again. In the remainder of this document, all traces are regarded as being well-formed, unless stated otherwise.

3.1.2 Performance measurement

Two values are interesting for the inspection of performance of software:

- the duration of a function call including the duration of all subsequent calls from that function (which is, conveniently, the time between an entry and an exit of the same function) called inclusive time. These values can be used to learn how expensive function calls are, and
3.1. Traces

- the duration of a function call excluding the duration of all subsequent calls from that function, called exclusive time. These values can be used to learn where there is room for improvement when inspecting expensive calls.

To illustrate the meaning of these values, an UML sequence diagram is given in figure 3.1, where a function doStuff is called, which in term calls sqrt twice.

![Figure 3.1: A UML sequence diagram, showing an example of calling of functions. In this example, doStuff is called by main, and calls sqrt twice. This example illustrates the meaning of inclusive and exclusive time.](image)

Our data model should enable the developer to inspect these values for function calls where the developer is able to improve performance.

We define $fd^+(e, \alpha)$ as the (inclusive) time a function call initiated by $e$ takes. If $e$ is not an entry event, $fd^+(e, \alpha) = 0$.

$$fd^+(e, \alpha) = \begin{cases} 
\text{exit}(e, \alpha).time - e.time & \text{if } e.type = \text{entry} \\
0 & \text{otherwise}
\end{cases} \tag{3.2}$$

where $\text{exit}(e, \alpha)$ is the exit event corresponding to the entered function of entry event $e$:

$$\text{exit}(e, \alpha_0 \cdot e \cdot \alpha_1 \cdot e' \cdot \alpha_2) = e' \text{ if } e'.type = \text{exit} \land WF(\alpha_1) \tag{3.3}$$

Note that $e.id = e'.id$ since we assume that $WF(\alpha)$ holds.

3.1.3 Memory measurement

As it is possible to capture memory allocation and deallocation events, we want to use these for detecting memory leaks: parts of memory that are not referenced anymore, but are not deleted.
As the memory allocation and deallocation events only contain a size and allocation identifier, and not e.g. an object type, it is not possible to know what kind of objects are allocated or freed. We thus focus only on the size of the memory allocated and deallocated in order find the increase in memory; e.g. the amount of memory leaked.

We use the following scheme: memory allocation events are matched by memory deallocation events by their unique allocation identifier. If such a deallocation event does not exist, this means that the memory usage is increased.

This scheme translates to the function \( \text{mem}(e) \), which is the memory increase caused by event \( e \):

\[
\text{mem}(e) = \begin{cases} 
    e.\text{size} & \text{if } e.\text{type} = \text{alloc} \land \neg\text{dealloc}(e) \\
    0 & \text{otherwise}
\end{cases}
\]  

(3.4)

where \( \text{dealloc}(e) \) is true if and only if a deallocation event exists for event \( e \), given that \( T \) is the set of events of all threads of the run, where \( e \in T \):

\[
\text{dealloc}(e) = \exists e' \in T \text{. } e'.\text{type} = \text{free} \land e'.\text{allocID} = e.\text{allocID}
\]  

(3.5)

Note that there is no constraint that states that \( e' \) should be an element of \( \alpha \); this is done because the deallocation may be done in another thread than the thread represented by \( \alpha \).

### 3.1.4 Call stacks

Call stacks are defined as a list of function identifiers that represent the entered, but not exited functions of a trace, or an event of a trace. Determining the call stack of an event is important, as the call stack contains information on call history. For instance, if the function \( \text{sqrt} \) takes a lot of time, the developer may be interested in which functions call it.

We define the call stack \( CS(\alpha) \) of trace \( \alpha \):

\[
CS(\epsilon) = \epsilon
\]

(3.6)

\[
CS(\alpha \cdot e) = CS(e, \alpha \cdot e)
\]  

(3.7)

where \( CS(e, \alpha) \) is the call stack of event \( e \), provided that \( e \) occurs in \( \alpha \):

\[
CS(e, \alpha_0 \cdot e \cdot \alpha_1) = \begin{cases} 
    CS(\alpha_0) \cdot e.\text{funcID} & \text{if } e.\text{type} = \text{entry} \\
    \text{pop}(CS(\alpha_0)) & \text{if } e.\text{type} = \text{exit} \\
    CS(\alpha_0) & \text{otherwise}
\end{cases}
\]  

(3.8)

where \( \text{pop}(\alpha) \) is the list \( \alpha \) excluding its last element, if \( \alpha \) is not empty:

\[
\text{pop}(\alpha \cdot e) = \alpha
\]  

(3.9)

In the remainder of this document, \( \sigma \) and \( \tau \) are considered to be call stacks.

### 3.2 Profile trees

A game created by Eximion BV typically generates 5 to 10 million events per second on average, depending on the game and the processor used. Due to this amount, storing the complete trace
is not convenient using today’s computers. Traces are therefore not stored; instead, we convert traces into what we call profile trees, which are stored during data logging.

A profile tree is defined as $G_{\alpha} = (V_{\alpha}, E_{\alpha})$ where $\alpha$ is the trace represented by the tree. Every vertex in $V_{\alpha}$ represents a unique call stack of any event $e$ in $\alpha$, including the empty stack:

$$V_{\alpha} = \{CS(e, \alpha) \mid e \in \alpha\} \cup \{\epsilon\}$$  \hspace{1cm} (3.10)

where $e \in \alpha$ is defined as:

$$e \in \alpha = \begin{cases} true & \text{if } \alpha = \alpha_0 \cdot e \cdot \alpha_1 \\ false & \text{otherwise} \end{cases}$$  \hspace{1cm} (3.11)

An edge $(\sigma, \tau)$ in $E_{\alpha}$ represents all entry events of the call stack $\sigma$, leading to the call stack $\tau$.

$$E_{\alpha} = \{(\sigma, \tau) \in V_{\alpha} \times V_{\alpha} \mid \sigma = \text{pop}(\tau)\}$$  \hspace{1cm} (3.12)

We define that $\sigma$ is a parent of $\tau$ and $\tau$ a child of $\sigma$ if an edge $(\sigma, \tau)$ exists.

Figure 3.2 shows a node link diagram of an example of a profile tree. Note that the label on every edge $(\sigma, \tau)$ is the function $f$ where $\sigma \cdot f = \tau$. In this example, we see that a function can occur on multiple edges. The value $\sigma$ of any vertex can be found by traversing from the root of the tree to the vertex, where $\sigma$ is the list of function identifiers shown in every encountered edge.

Figure 3.2: A node link diagram of an example of a profile tree, where the labels on the edges represent the function augmented by the child vertex.

events($\sigma, \alpha$) is a set containing all events of trace $\alpha$ that concerns vertex $\sigma$. We say that event $e$ is an event of a vertex $\sigma$ if $e \in \text{events}(\sigma, \alpha)$.

$$\text{events}(\sigma, \alpha) = \{e \in \alpha \mid CS(e, \alpha) = \sigma\}$$  \hspace{1cm} (3.13)
3.2.1 Additionally stored information

Traces contain the full profiling information. Since we store only the profile tree, we need to store additional information to analyze the software. We thus store additional data per vertex of a profile tree. Note that we make a distinction between inclusive and exclusive values, where inclusive values concern a complete subtree, and exclusive values concern a vertex.

The values of the following functions are stored for each vertex $\sigma$ of tree $G_\alpha$:

- the number of function calls, or function entry events, leading to call stack $\sigma$:

$$\text{calls}(\sigma, \alpha) = \#\{e \in \text{events}(\sigma, \alpha) \mid e.\text{type} = \text{entry}\}$$  \hspace{1cm} (3.14)

- the total inclusive time spent in call stack $\sigma$:

$$d^+(\sigma, \alpha) = \sum_{e \in \text{events}(\sigma, \alpha)} fd^+(e, \alpha)$$  \hspace{1cm} (3.15)

- the squared inclusive time of $\sigma$:

$$D^+(\sigma, \alpha) = \sum_{e \in \text{events}(\sigma, \alpha)} fd^+(e, \alpha)^2$$  \hspace{1cm} (3.16)

- the amount of memory allocated inside $\sigma$ which is not deallocated:

$$\text{mem}^- (\sigma, \alpha) = \sum_{e \in \text{events}(\sigma, \alpha)} \text{mem}(e)$$  \hspace{1cm} (3.17)

3.2.2 Computed profiling values

The following values are useful, but easily calculatable from the stored information, so they do not need to be stored:

- the total exclusive time of $\sigma$:

$$d^- (\sigma, \alpha) = d^+(\sigma, \alpha) - \sum_{(\sigma, \tau) \in E_\alpha} d^+(\tau, \alpha)$$  \hspace{1cm} (3.18)

- the average inclusive time per function call to $\sigma$:

$$\overline{d}^+ (\sigma, \alpha) = \begin{cases} \frac{d^+(\sigma, \alpha)}{\text{calls}(\sigma, \alpha)} & \text{if } \text{calls}(\sigma, \alpha) \neq 0 \\ 0 & \text{if } \text{calls}(\sigma, \alpha) = 0 \end{cases}$$  \hspace{1cm} (3.19)

- the average exclusive time per function call to $\sigma$:

$$\overline{d}^- (\sigma, \alpha) = \begin{cases} \frac{d^- (\sigma, \alpha)}{\text{calls}(\sigma, \alpha)} & \text{if } \text{calls}(\sigma, \alpha) \neq 0 \\ 0 & \text{if } \text{calls}(\sigma, \alpha) = 0 \end{cases}$$  \hspace{1cm} (3.20)
3.2. Profile trees

- the standard deviation of the duration of calls:

\[ SD(\sigma, \alpha) = SD(d^+(\sigma, \alpha), D^+(\sigma, \alpha), \text{calls}(\sigma, \alpha)) \] (3.21)

where \( SD(s, S, n) \) is defined as the standard deviation of a list \( \alpha \) where \( s \) is defined as the sum of all elements in \( \alpha \), \( S \) as the sum of the squares of all elements of the list, and \( n \) as the number of elements in \( \alpha \):

\[ SD(s, S, n) = \sqrt{\frac{S - \frac{s^2}{n}}{n - 1}} \] (3.22)

- the total inclusive memory increase of \( \sigma \):

\[ \text{mem}^+(\sigma, \alpha) = \sum_{\sigma \tau \in V_\alpha} \text{mem}^{-}(\sigma \cdot \tau, \alpha) \] (3.23)

3.2.3 Snapshots

Using profile trees described in this section, finding causes of frame drops are hard to detect because the time aspect of traces is lost. The standard deviation of calls to vertices may be an indication that something is happening, but gives no information about when the duration of a call is deviant from the normal values.

To regain the time aspects of the trace, we define snapshots, where a snapshot is a tree constructed from only a prefix of a trace. A snapshot thus contains the cumulative information of a run up to a certain point.

We define the function \( SS(t, \alpha) \), which is defined as the prefix of a trace \( \alpha \), containing only events which time stamp is at most \( t \):

\[
SS(t, \epsilon) = \epsilon \] (3.24)
\[
SS(t, e \cdot \alpha) = \begin{cases} e \cdot SS(t, \alpha) & \text{if } e.\text{time} < t \\ \epsilon & \text{otherwise} \end{cases} \] (3.25)

In words, a snapshot represents only the part of the trace that occurs before or on time \( t \). It is possible to create a call tree from this snapshot; however, it is possible (and even likely) that the snapshot is not well-formed, because some functions are entered, but exited after the snapshot (for example the function \texttt{main}). For that matter, the function \( \text{closure}(t, \alpha) \) is introduced, which is the trace \( \alpha \) augmented with exit events for every function still on the call stack:

\[
\text{closure}(t, \alpha) = \begin{cases} \epsilon & \text{if } \text{CS}(\alpha) = \epsilon \\ \text{closure}(t, \alpha \cdot e) & \text{otherwise, with } e.\text{type} = \text{exit} \land e.\text{time} = t \land e.\text{funcID} = \text{last}(\text{CS}(\alpha)) \end{cases} \] (3.26)

where \( \text{last}(\alpha) \) is the last element of \( \alpha \):

\[- \text{last}(\alpha \cdot e) = e \] (3.27)

For convenience, we define \( \text{closedSS}(t, \alpha) \) as the closed snapshot of trace \( \alpha \) with timestamp \( t \):

\[ \text{closedSS}(t, \alpha) = \text{closure}(t, SS(t, \alpha)) \] (3.28)
Using these definitions, a tree of a snapshot is defined as \( G_{\text{closedSS}}(t, \alpha) \) where \( t \) is the time stamp of the snapshot, and \( \alpha \) the trace of which a snapshot is created.

Implementation-wise, the data logger stores snapshots of a profile tree during run-time at pre-defined moments. The user can choose to take snapshots at a predefined interval, e.g. every second.

Note that the memory usage of GamePro during run-time is related to the frequency of the snapshots. Experimentation has shown that one snapshot per second is sufficient for finding causes of not only major, but also minor frame drops. The memory usage is acceptable at a frequency of once per second, and allows for multiple minutes of logging before the memory usage causes a problem. A run of a minute often contains enough information to find performance issues. Once per second is therefore the default.

### 3.2.4 Pivot function

To be able to find causes of frame drops, we first need to know when they occur. If we know the course of the frame rate per second, we know when frame drops occur.

We therefore appoint a single function \( f \) of the inspected software as the pivot function: the function that determines the performance of the software. If we e.g. want to know the frame rate, we use the function that renders an image to the screen. If we e.g. want to know the performance of an algorithm which purpose is to generate a number of elements, we use the function that generates an element. The developer has to provide the name of the pivot function.

\[
Pivots(\alpha, f) = \{ e.\text{time} | e \in \alpha \land e.\text{funcID} = f \land e.\text{type} = \text{exit} \}\]

The data logger stores \( Pivots(\alpha, f) \) for the pivot function \( f \) specified by the developer. To know the performance (e.g. the frame rate) on a time interval \((t, t')\) for the set of pivot timestamps \( P \) we use the following formula:

\[
fps(t, t', P) = \frac{\#\{p \in Pivots | t \leq p < t'\}}{t' - t}
\]

Pivot functions can also be used to determine the frequency of snapshots.

### 3.3 Profile trees vs. traces

Our data model has multiple advantages:

- it enables fast logging, because of the relatively small amount of children of vertices, so searching for any child of a vertex during run-time is fast due to the fact that there are usually not many possible edges to vertices representing different call stack traces. This facilitates requirement 1;

- many interesting values do not need to be stored separately, because they can be calculated afterwards. This saves memory and time to store them. This also facilitates requirement 1;

- compared to a flat profiler, which is a profiler that does not show hierarchical information but only a list of functions, we can trace back functions that are expensive in some cases, but not in others to the call stack that caused the expensive call or calls. This is due to the fact that all separate call stacks are stored, and
3.4 Statistics

- profile trees are easier to present using visualization than traces. Numerous visualizations for trees are available.

The following disadvantages apply:

- The largest disadvantage is the memory usage which still is substantial when profiling a large application, although it is massively smaller than storing the trace. This is especially important when using recursion with many iterations (called deep recursion), which can lead to massive call trees. Massive call trees use a substantial amount of memory which can cause problems when profiling on a system that does not have a lot of memory. A solution to the problem of memory usage when using recursion can be found in appendix A.

- We lose the information on duration of individual calls. This can cause a loss of information whenever for some reason one individual call is much more expensive than other calls to the same function. This is not a large problem, as this type of fluctuations is often noise, caused by e.g. the operating system swapping to another process.

- We lose the information about the sequence of the software occurring between two snapshots. This is not a big problem, as this information is not relevant to inspecting performance of software. However, this does mean that snapshot frequency that is too low can cause the user to miss frame drops. So, requirement 6 can only be fulfilled by a careful choice of snapshots.

- The call stacks or functions where deallocations are done are not visible. This is usually not a problem, as these deallocations would only show up as a part of software ‘doing its job’. Memory leaks often occur because developers forget to create a matching destruct command whenever they are creating an object, and in such a case the call stack of functions that allocate memory that is not deallocated gives enough information for the programmer to find the cause of the memory leak.

3.4 Statistics

The following tables show statistical data about the result of profiling games created by Eximion BV, in order to give the reader an idea about the size of a typical data set, which consists of the snapshots of a profile tree.

The following table is created from data generated when using GamePro on the game Oggo’s Odyssey (for a screenshot, see figure 3.3):

<table>
<thead>
<tr>
<th>Number of vertices</th>
<th>165,110</th>
</tr>
</thead>
<tbody>
<tr>
<td>Average events handled per second</td>
<td>5,845,614</td>
</tr>
<tr>
<td>Maximum depth of the call tree</td>
<td>57</td>
</tr>
<tr>
<td>Number of unique functions</td>
<td>5,190</td>
</tr>
</tbody>
</table>

The following table is created from data generated when using GamePro on the game Konstrukt Invasion (for a screenshot, see figure 3.4):

<table>
<thead>
<tr>
<th>Number of vertices</th>
<th>173,082</th>
</tr>
</thead>
<tbody>
<tr>
<td>Average events handled per second</td>
<td>5,184,945</td>
</tr>
<tr>
<td>Maximum depth of the call tree</td>
<td>40</td>
</tr>
<tr>
<td>Number of unique functions</td>
<td>7,033</td>
</tr>
</tbody>
</table>
Figure 3.3: A screenshot of the game Oggo’s Oddysey by Eximion BV.

Figure 3.4: A screenshot of the game Konstrukt Invasion by Eximion BV.
Chapter 4

Data manipulation

The amount of collected data is vast as can be seen in section 3.4, and unless it is filtered, visualized, or both, it contains too much information for the developer to quickly comprehend.

GamePro Analysis is able to filter, modify, and visualize the data described in section 3.2. This chapter discusses the ways the data can be adapted for analysis by the developer. Section 4.1 describes the way data sets and snapshots can be compared, and section 4.2 describes the different ways the data can be arranged.

4.1 Comparing

Suppose, we have a data set of a run that contains a frame drop. The developer may want to know what happens during an interval which contains that frame drop. We therefore want to compare a snapshot before, and a snapshot after that interval to determine what happens during that interval. This comparison is described in section 4.1.1.

Comparing snapshots gives some information, but no information about why the interval between these two snapshots contains a frame drop, while others do not. Therefore, we also want to compare two intervals, in order to e.g. see differences between an interval containing a frame drop and an interval that does not.

The user may also want to see differences between two traces of different runs in order to compare two runs of the same application, but e.g. altered by incorporating an optimization, or adding features that may hinder performance. These comparisons are described in section 4.1.2.

4.1.1 Comparing snapshots

For convenience, we define trace intervals where every trace interval $i$ contains:

- $i.start$ is the time stamp of the start of the interval,
- $i.end$ is the time stamp of the end of the interval. Note that we enforce $i.end > i.start$, and
- $i.trace$ is the trace for which the interval applies.

The following functions apply to trace intervals $i$:

- $\text{length}(i)$ is the length of the interval:
\textit{length}(i) = i.end - i.begin \hspace{1cm} (4.1)

- \textit{startTrace}(i) is the closed snapshot of the trace \(i.trace\) at \(i.start\):
  \[\text{startTrace}(i) = \text{closedSS}(i.start, i.trace)\] \hspace{1cm} (4.2)

- \textit{endTrace}(i) is the closed snapshot of the trace \(i.trace\) at \(i.end\):
  \[\text{endTrace}(i) = \text{closedSS}(i.end, i.trace)\] \hspace{1cm} (4.3)

Any function \(f(\sigma, \alpha)\) concerning traces such as \textit{calls} or \(d^+\) can be used to get the delta value of a trace interval \(i\) using the following function template \(\Delta f\):

\[\Delta f(\sigma, i) = f(\sigma, \text{endTrace}(i)) - f(\sigma, \text{beginTrace}(i))\] \hspace{1cm} (4.4)

This applies for \(f \in \{\text{calls}, d^+, D^+, d^-, D^-, \text{mem}^+, \text{mem}^-\}\). For example, if we want to know the inclusive duration of a vertex \(\sigma\) during a certain interval we use \(\Delta d^+\):

\[\Delta d^+(\sigma, i) = d^+(\sigma, \text{endTrace}(i)) - d^+(\sigma, \text{beginTrace}(i))\] \hspace{1cm} (4.5)

The average inclusive duration \(\Delta \overline{d^+}\) is calculated using the following formula:

\[\Delta \overline{d^+}(\sigma, i) = \frac{\Delta d^+(\sigma, i)}{\Delta \text{calls}(\sigma, i)}\] \hspace{1cm} (4.6)

Similar for \(\Delta \overline{d^-}\). Note that whenever a vertex \(\sigma\) is non-existent in a tree representing a trace \(\alpha\), the values of the functions \textit{calls}(\sigma, \alpha), d^+(\sigma, \alpha), d^-(\sigma, \alpha), D^+(\sigma, \alpha), \text{mem}^+(\sigma, \alpha)\) and \(\text{mem}^-(\sigma, \alpha)\) are zero by definition. Therefore, comparing a vertex that exists in one trace interval, but not in the other using the above functions, simply returns the value of the trace interval that does contain the vertex.

### 4.1.2 Comparing traces and intervals

Using the \(\Delta f\) functions of the previous section, it is possible to learn what happens during intervals. However, what we can not learn is why one interval is different from another, e.g. why one interval contains a frame drop, and why another does not. We therefore want to compare two different intervals.

The user may also want to know what the differences are between two traces of different runs, e.g. to compare two runs of the same application which is e.g. altered by incorporating an optimization, or by adding features that may lower performance.

If we simply subtract the values from two traces or trace intervals with different lengths, the shorter trace or interval will per definition contain overall lower durations. This is undesirable, because we want to compare the performance of data sets, which is in our case the frame rate. We therefore compare the information relative to the length of the intervals selected.

We define a function template \(\Delta^2 f(\sigma, i, j)\), where \(i\) and \(j\) are trace intervals and \(i.trace = \alpha\) and \(j.trace = \beta\). Note that if \(\alpha = \beta\), we are comparing two intervals of the same trace. If we want to compare complete traces, we define the start and end time of both intervals to the time stamp of respectively the first and last event of the corresponding trace.
4.2. Modifying profile trees

\[ \Delta^2 f(\sigma, i, j) = \frac{\Delta f(\sigma, i) \text{ length}(i)}{\text{ length}(j)} - \frac{\Delta f(\sigma, j) \text{ length}(j)}{\text{ length}(i)} \]  (4.7)

This applies to \( f \in \{ \text{calls}, d^+, D^+, d^-, D^-, \text{mem}^+, \text{mem}^- \} \). If we e.g. want to compare the inclusive time of \( \sigma \) between interval \( i \) and \( j \), we use:

\[ \Delta^2 d^+(\sigma, i, j) = \frac{\Delta d^+(\sigma, i) \text{ length}(i)}{\text{ length}(j)} - \frac{\Delta d^+(\sigma, j) \text{ length}(j)}{\text{ length}(i)} \]  (4.8)

The average inclusive difference in number of calls is calculated using the following formula:

\[ \Delta^2 d^+(\sigma, i, j) = \frac{\Delta^2 d^+(\sigma, i, j)}{\Delta^2 \text{calls}(\sigma, i, j)} \]  (4.9)

Similar for the average exclusive difference \( \Delta^2 d^- \).

### 4.2 Modifying profile trees

This section describes the ways the data model described in section 3.2 is modified before presenting. This is done so that less visual noise is generated, e.g. from insignificant functions, and to help developers understand the data better.

#### 4.2.1 Modifier pipeline

The filtering and modification system in GamePro for modifying call trees is called the modifier pipeline. This system enables the user to focus on certain areas, e.g. areas that the user suspects to be causing performance issues. Furthermore, if insignificant data is filtered, a profile tree can be much smaller, which results in enhanced performance of the tool and it leads to a focus which makes visualization easier.

The modifier pipeline consists of a list of modifiers, which transforms profile trees \( G_\alpha \) to \( G'_\alpha \). Examples of modifiers include e.g. a modifier that alters data sets in such a way, that e.g. all functions of classes are combined together, to create a class-level view of the data. An example of this modifier can be seen in figure 4.1. Another example for a modifier is to remove vertices from the tree, such as vertices that are take an insignificant amount of time (e.g. 1 promille of the total time of the run), or e.g. to remove all vertices, except the ones that cause memory leaks.

The user can choose the modifiers contained in this list. Every data set that is inspected goes through this list of modifiers in the given order before it is presented.

#### 4.2.2 Flattening

One of the methods used to enable the user to interpret the data better is to flatten the profile tree. This flattened profile tree contains a set of all distinct functions \( F \) occurring inside the profile tree.

An element \( f \) in \( F \) represents the vertices where \( f \) is the last element of the call stack represented by that vertex. This way, we can calculate profile information such as time spent inside a function. \( F(\alpha) \) is the flattened tree of \( G_\alpha \):

\[ F(\alpha) = \{ e.\text{funcID} \mid e \in \alpha \} \]  (4.10)
Chapter 4. Data manipulation

Figure 4.1: An example of a profile tree modifier. This example illustrates how the class modifier works. Every vertex is combined with its siblings and children that have the same class.

$P(f, \alpha)$ is the set of parents of function $f$:

$$P(f, \alpha) = \{f' \mid \exists \sigma \in V_\alpha \; \sigma \cdot f' \cdot f \in V_\alpha\} \quad (4.11)$$

$C(f, \alpha)$ is the set of children of function $f$:

$$C(f, \alpha) = \{f' \mid \exists \sigma \in V_\alpha \; \sigma \cdot f \cdot f' \in V_\alpha\} \quad (4.12)$$

This flattened call tree is used for the data for most visualizations described in chapter 5. It is easily comprehensible by the developer as the information from this list resembles information returned by many other profilers. Furthermore, the information has a one-on-one relationship with the code, as every function in this set corresponds to a part of the source code. This relationship helps developers to connect profiling data with the source code of the analyzed software.

All functions in section 3.2 and 4.1 concern vertices. For all previously described functions there is a corresponding function for flattened trees, which can be constructed using the following function template $t$:

$$t_F(f, \alpha) = \sum_{\sigma \cdot f \in V_\alpha} t(\sigma \cdot f, \alpha) \quad (4.13)$$

Whenever we e.g. want to know how large the memory increase is of a certain function, regardless of its place in the call stack, we use:

$$\text{mem}_F(f, \alpha) = \sum_{\sigma \cdot f \in V_\alpha} \text{mem}^{-}(\sigma \cdot f, \alpha) \quad (4.14)$$

This holds for all functions, including the functions in section 4.1.1 and section 4.1.2. One exception occurs, namely when calculating inclusive values of recursive functions. When we e.g. calculate the inclusive time of a function $f$, the time spent inside the recursively called function is added more than once. To illustrate this, see the profile tree shown in figure 4.2.
4.2. Modifying profile trees

Figure 4.2: An example of a profile tree where the inclusive duration in seconds is shown in the vertices. When flattened, we get the value 5 for \texttt{funcA}, the value 2 for \texttt{funcB} and the value 1 for \texttt{funcC}. Note that we ignore the recursive calls of \texttt{funcA} because its time is already contained in the original call.

Suppose, \texttt{funcA} represents the complete run. The run thus took 5 seconds. Should we calculate the inclusive duration of \texttt{funcA} naively, we would get 7, which is larger than the run time of the software. This would confuse the developer into thinking recursive functions are more expensive than they really are. We therefore ignore the functions that are called recursively while calculating inclusive duration of a function:

\[
 d^+_F(f, \alpha) = \sum_{\sigma \cdot f \in V, \lambda / \notin \sigma} d^+(\sigma \cdot f, \alpha)
\] (4.15)

The same goes for inclusive memory usage:

\[
 mem^+_F(f, \alpha) = \sum_{\sigma \cdot f \in V, \lambda / \notin \sigma} mem^+(\sigma \cdot f, \alpha)
\] (4.16)

4.2.3 Multiplication factors

The values that concern time, namely the inclusive duration $d^+$, exclusive duration $d^-$, average inclusive duration $\overline{d^+}$, average exclusive duration $\overline{d^-}$ and the standard deviation $SD$ are values that are represented in machine cycles. This is not very convenient, as the quantity of machine cycles is \texttt{clockspeed}/s where \texttt{clockspeed} is the number of machine cycles the processor executes per second.

With modern processor speeds reaching multi billion machine cycles per second, it is hard to get meaningful results from these values because it is not easy to determine the difference in order of magnitude between e.g. 39916800 machine ticks and 479001600 machine ticks on a 1.7 GHz processor. It is much more convenient if we e.g. use seconds, or percentages.

We therefore define a multiplication factor $m$ per trace that is multiplied by any value that are represented in machine cycles. $m$ is not used for multiplication with e.g. the number of calls \texttt{calls} or the exclusive memory usage $mem^-$, because that would not make sense.

For example, $m = 1/clockspeed$ if the user desires to have all values in seconds. To get the percentage of the value relative to the total duration of all inspected functions, $m = 100/d^+(\epsilon, \alpha)$.
where $\alpha$ is the trace inspected. This results in the fact that $m \times \sum_{\sigma \in V_\alpha} d^-(\sigma, \alpha) = 100$, making it easy for the user to comprehend the durations. This is chosen as the default value for $m$.

In the remainder of this text, we talk about values concerning time, e.g. duration, average or standard deviation, without mentioning the fact that they are multiplied by $m$ to obtain human-readable values.
Chapter 5

Visualization

In the previous chapters, we defined the data model. Profile data stored using this model needs to be presented in a way that enables the developer to gain insight in the performance of the software.

The purpose of the visualization described in this chapter is to present the data described in chapter 4.2 to the developer in such a way, that the user questions in section 1.2 can be easily answered. This chapter describes the different views that are available in GamePro Analysis.

Our aim is to draw the attention of the developer to the vertices or functions that are significant in one way or another, e.g. vertices or functions that are possible causes of a frame drop, or consume a lot of time.

The data has two major aspects:

- A structural aspect, which consists of the profile tree, and
- A time aspect, which consists of the snapshots.

A tree is commonly visualized in two dimensions, as one dimension usually is not sufficient for visualizing complex trees. In order to visualize the time aspect, we could extend one of the existing visualizations to a 3D visualization, where the third dimension represents the time aspect. This results in a number of views stacked on top of each other, one view for every snapshot. We expect that such a view would not be very helpful, as it would probably be very cluttered, as some views will obstruct other views. This was also suggested by Pieter Deelen in [Dec06], and dismissed by him too. Colin Ware also confirms this by performing an analysis of human space perception[War01], where he states that much less information is storable in the direction that points away from the viewer.

We therefore decided that we should create multiple two-dimensional views instead of a two- or three-dimensional view. We need at least a view showing the structural information, and a view showing the time aspect.

The rest of this chapter is structured as follows: Section 5.1 describes the requirements of the visualization, section 5.2 describes the techniques used and section 5.4 describe the views implemented in GamePro Analysis. Section 5.6 concludes this chapter.

5.1 Requirements

We aim to use visualization to gain insight on causes of low performance and frame drops, and enable developers to find performance issues. Furthermore, the visualization should enable the
user to answer the questions described in 1.2.

We thus specify the following requirements regarding the visualization:

1. the structure, and the values contained within the profile tree should be visible, in order to find large bottlenecks and thus answer user question 1. We want this visualization to draw the attention of the user to performance issues caused by expensive call stacks;

2. frame drops should be visible, as well as their causes, as per overall requirement 6;

3. data of functions on the flattened tree described in 4.2.2 including some hierarchical information should be visible, because functions can take a large amount of time while being called from different call stacks, and

4. precise numerical values should be visible, because the developer may want to inspect these after finding memory or performance issues.

We want our tool to be dynamic, so developers can focus on parts of the data. We therefore specify the following requirements regarding the interaction:

5. the developer should be able to select between performance and memory analysis, because we want to find performance issues as well as memory issues as per overall requirement 7 and 9;

6. the developer should be able to select parts, and inspect these in detail, as per overall requirement 8, and

7. the developer should be able to compare intervals (to find differences between a frame drop and a normal execution) and data sets (to find differences when altering code) as per overall requirement 10.

5.2 Techniques

To be able to present the data in the ways described in the previous section, we need to have a number of techniques of visualizing data. This section discusses the non-trivial visualization techniques used in GamePro, namely the delta stack graph described in section 5.2.1 and the treemaps for inconsistent trees with negative values, described in 5.2.2.

5.2.1 Delta stack graph

In this section the delta stack graph is described, which is a modification of the standard stack graph, which in term is derived from the bar graph.

A bar graph is a graph where a finite set of input values \( V \) and a function \( f : V \rightarrow \mathbb{R} \) is represented. For each value in \( V \), a rectangle is shown with a fixed width, and a height linearly proportional to the value. If \( f(v) \) is negative for any \( v \in V \), the bar is drawn completely below the horizontal axis or baseline, and the height of the bar is \( |f(v)| \). In order to determine the color of the bars, we use a function \( f_c : V \rightarrow C \) where \( C \) is the set of colors. An example can be seen in the left graph shown in figure 5.1.

A variable width bar graph is a bar graph, where the widths of the bars are not fixed, but depend on \( V \). In order to draw such a graph, we use the following scheme: let \( V \) consist of the sorted elements \( v_0 < v_1 < \ldots < v_N \). For every consecutive pair of values \((v_{i-1}, v_i)\) we draw a vertical bar on the interval \((v_{i-1}, v_i)\) on the baseline, with height \( |f(i)| \), and color \( f_c(i) \). An example of the
5.2. Techniques

Figure 5.1: An example of a bar graph (on the left), and a variable width bar graph (on the right). We use a discrete set of values $V$, and a function $f : V \rightarrow \mathbb{R}$. The width of bars is discrete in the bar graph, and determined by the difference in consecutive values of $V$ in the variable width bar graph. Note that negative values are also possible, and are shown below the baseline.

variable width bar graph can be seen in the right graph of figure 5.1. Note that this results in the fact that $f(v_0)$ is not shown.

A stack graph [Wat05] is a modification of the variable width bar graph. Using a stack graph, multiple functions $f : V \rightarrow \mathbb{R}$ are literally stacked vertically per input value $v$. See also figure 5.2. Colors are used to identify the functions, so we use a color $c(f)$ for every shown function.

We notice that the rectangles corresponding to a single function are not always connected using this scheme, which gets worse as stacked functions are further away from the baseline. The order in which the functions are drawn is therefore important. If we use a large amount of functions, or functions that fluctuate largely, the stack graph becomes unclear; otherwise, the results are acceptable.

Figure 5.2: This figure shows an example of the stack graph. Note that functions that are drawn further away from the baseline have a larger chance of being disconnected, and thus harder to interpret than functions that are closer.

A stack graph can only visualize positive values. Taking into account that it is possible for our data to contain negative values (e.g. when comparing data sets), we can not use a stack graph without modification. We therefore introduce a technique we call the delta stack graph, where we show
two stack graphs: one for all positive values, and one for all negative values. The negative-valued stack graph is shown upside down directly beneath the positive stack graph. An example of a delta stack graph can be seen in figure 5.3. This graph is drawn using rectangles; we experimented with smoothing the graph which enhanced the aesthetics of the graph and partially solved the connectivity problem, but not its clarity. An example of how this works with real data is shown in figure 5.4.

![Image of two stack graphs]

Figure 5.3: This figure shows the way functions including negative values are stacked using the delta stack graph. Note that functions that are drawn further away from the baseline have a larger chance of being disconnected, and thus harder to interpret than functions that are closer.

![Image of timeline view]

Figure 5.4: This figure shows the timeline view implemented using the delta stack graph. The graph shows the difference from the average for multiple functions.

5.2.2 Treemaps for inconsistent trees with negative values

This section describes an adaptation of the Squarified Cushion Treemaps described in [BHvW00]. An example of a treemap can be seen in the left image in figure 5.5. Treemapping [JS91] is a space-filling method for displaying hierarchical data using nested rectangles, where each rectangle represents a vertex in the hierarchy. The size of each rectangle is proportionate to the size of the vertex it represents. As a result, the structure of the tree is reflected in the treemap. A treemap is a compact visualization particularly tuned to visualize the size of the leaves in a tree, because leaves that are relatively large draw the attention. Drawbacks of treemaps include that the hierarchical
5.2. Techniques

Figure 5.5: Treemaps of all files belonging to the Kalydo Engine, displayed using SequoiaView [SV]. The size of the leaves are relative to the size of the files. On the left, regular treemaps are used. On the right, cushion treemaps are used.

structure is partly lost, because all display space is used for showing the size of the vertices. It is hard to discern e.g. the depth of a vertex.

Cushion treemapping [vWvdW99] uses shading for the shown rectangles to address some of the problems inherent to treemaps, such as the loss of depth information of the tree. This shading results in a ‘cushion-like effect’, hence cushion treemapping. An example of a cushion treemap can be seen in the right image in figure 5.5.

The standard treemapping method often gives thin, long rectangles. As a result, rectangles are often difficult to compare and select. In order to resolve that problem, squarified treemaps are introduced in [BHvW00], where the layout is generated in such a way, that the aspect ratio of rectangles is relatively small. An example of a squarified cushion treemap can be seen in figure 5.6.

Figure 5.6: Similar to figure 5.5, Treemaps of all files belonging to the Kalydo Engine are displayed, but we are now using squarified cushion treemaps. The long thin rectangles are rearranged to approximate squares.

The input of our implementation of the treemap consists of a tree where every vertex has a color, text and size. The text of a vertex and the value in textual form of the same vertex is shown inside the area represented by vertex if the text fits in that area. If not, a title tip is shown containing the text and the value when hovering with the mouse over the area represented by the vertex. We can identify the following problems regarding our data and treemaps:

- Treemaps use the assumption that trees are consistent: the size of vertices in trees are equal to the sum of their children’s sizes. The size of all vertices that do not contain children, also
known as *leafs*, determine the size of the complete tree. This is not true in our case, because if we e.g. use inclusive duration for the size of vertices, the vertices are exactly its exclusive duration larger than the sum of the children’s sizes.

- Another problem encountered when using (cushion) tree maps is the fact that they, like stack graphs, are only suitable for vertices of positive size, because they use areas to visualize sizes. We thus want negative values to be treated differently. We also want to quickly see what the ratio is between positive and negative values.

In the remainder of this section, we address these problems.

**Consistent trees**

We want our trees to be consistent. This is done by modifying the tree by creating a leaf for every vertex that has children. The generated leaf is a child of the vertex, and has the value and color of the vertex. Now, all information is contained in leafs. Note that there is no distinction made between generated leafs and other children of the vertex, so e.g. in a treemap the leaf is drawn in between the children of the vertex.

We now say that the size of a leaf is equal to its absolute value. The size of all other vertices is simply the sum of the size of all its children. This way we can show e.g. inclusive and exclusive duration simultaneously: the exclusive duration of a vertex is represented by the size of the generated leaf, and the inclusive duration is represented by the size of the vertex. An example of a profile tree which is made consistent can be seen in figure 5.7. In the remainder of this document, we assume that the trees visualized by treemaps are consistent.

![Figure 5.7: An example of a profile tree, where the inclusive duration is shown in the vertices. After altering the tree to make it consistent, every leaf contains exclusive duration, and all other vertices contain inclusive duration. Note that if a function calls itself, this leads to two edges containing the same function.](image)

**Visualizing negative values**

In order to visualize the distinction between negative and positive vertices, we introduce cushion tree maps with *inverted cushions*, where positive values are drawn using cushions, and negative values are drawn using inverted cushions. In order to draw inverted cushions, we use the default cushion treemap algorithm which determines an intensity $0 \leq i \leq 1$ per pixel. We replace this intensity for positive values by $\frac{1}{2} + \frac{i}{2}$, and for negative values by $\frac{1}{2} - \frac{i}{2}$. Drawbacks of treemaps with inverted cushions include a loss of contrast resolution because we need half of the possible contrast to show the inverted cushions, and the fact that inverted cushions are less aesthetically pleasing than their original counterpart. An example of a cushion treemap with inverted cushions can be seen in figure 5.8.
5.2. Techniques

Splitting negative and positive leafs

In the case that our data contains both positive and negative values, we want to focus on sizes rather than structure of the tree because profiling is much more about finding significant functions rather than the exact structure of the profile tree. We want to know how ‘positive’ or ‘negative’ the complete tree is, and this information is hard to determine from a treemap of a tree containing a mix of positive and negative values. We therefore create two trees, which both contain all vertices and edges of the original tree. We remove all leafs with a positive value from one tree, and all leafs with a negative value from the other leaf. To illustrate this operation, an example is presented in figure 5.9. When showing such a tree in a treemap, we see clearly how ‘positive’ or ‘negative’ a tree is. A drawback of this method is that we lose structure, as leafs that structurally lie close to each other can now be shown far away from each other. In order to visualize these trees, we simply combine these trees using a root vertex where the children of this vertex are the two trees. Note that using this, the ratio between positive and negative trees is now clear.

Figure 5.8: An example of a tree containing both positive and negative leafs. The size of leafs is their absolute value, and the size of non-leaf vertices are simple the sum of their children’s sizes.

Figure 5.9: An example of how a tree containing both positive and negative leafs are split into two trees, and then combined by a root vertex and two edges. Note that sizes of vertices that originally had children are the sum of its children; if such a vertex becomes a leaf and thus contains no children anymore, its size is zero.
5.3 Data and selection

As stated in section 2.2.2 GamePro Analysis is implemented using the observer design pattern. Our subject consists of a context, described in section 5.3.1 and a selection, described in section 5.3.2.

We use data according to the data model in chapter 3 and 4. We want to enable the developer to focus on certain aspects of their data; it is therefore possible to manipulate the data contained in the subject by selecting functions and data types, and modification of the data by e.g. filtering. This selection is contained in the subject because we have multiple views, and want the developer to get a consistent view of the selection. This way the same data is presented using different visualizations.

5.3.1 Context

The context is defined as a set of data sets $D$, each containing:

- a set of timestamps for every snapshot $T$;
- all snapshots of a run $\{closedSS(V_\alpha, t) | t \in T\}$ where $\alpha$ is the original trace of the thread of the run represented by this data set;
- a set of pivot timestamps $Pivots(\alpha, f)$.

Every function has its own global color $c(f)$ to identify functions in the different views. This color is currently randomly picked from a list of acceptable colors to avoid large differences in contrast.

5.3.2 Selection

The following is contained in the subject, and regarded as selection:

- a data set $d \in D$;
5.4 Views

- a set of functions occurring in this set $SF \subseteq F(\alpha)$ where $\alpha$ is the trace represented by the data set, as per requirement 6. Note that these represent functions occurring in the flattened tree;
- an interval $(t, t')$ which defaults to the complete run, and
- the data type that the user wants to inspect, e.g. memory usage, inclusive time, or average exclusive time, as per visualization requirement 5. Every data type corresponds to a function $\Delta f$ or $\Delta^2 f$ in respectively sections 4.1.1 or 4.1.2, or the functions for the flat tree described in 4.2.2, depending on what the observers show.

If the developer wants to compare intervals, the selection can also contain an second interval $(t_2, t'_2)$ which is used to compare data in this interval to data of $(t, t')$. If the developer wants to compare data sets as per visualization requirement 7, the selection can also contain a second data set $d_2 \in D$.

5.4 Views

![Figure 5.11: A screenshot of GamePro Analysis. Every window inside the main window displays one of the views described in this section.](image)

This section describes the observers used in GamePro, where each observer is a view that visualizes the data described in section 5.3. Each view shows different aspects of the profile data and contains their own focus. Figure 5.11 displays a screenshot of GamePro Analysis showing all views. On the left, we see (from top to bottom):

- the data sets view, described in section 5.4.1,
- the function/children view, described in section 5.4.4;
- the parents view, also described in section 5.4.4, and
- the time line view, described in section 5.4.3.

On the upper right, we see the tabular function view, described in section 5.4.2. On the lower right we see the profile treemap view, described in section 5.4.5.
5.4.1 Data sets view

This section describes the data sets view. The data sets view is implemented using a tree view similar to popular file explorers. The purpose of this view is to enable the user to select data sets and view some properties of data sets. Note that in case the shown data sets are filtered, these properties apply to the original, unfiltered data set. Every data set in \( D \) is shown in a list. The selected data set \( d \) is colored green in this list. The second data set \( d_2 \) is colored red. A screen shot of the data sets view is shown in figure 5.12.

5.4.2 Tabular function view

This section describes the tabular function view. The purpose of this view is to enable the developer to get numerical values of their data sets, as per visualization requirement 4. The flat data set is shown in this view, as per requirement 3. Also, the data type can be selected, facilitating visualization requirement 5 which states that we should be able to select between performance and memory analysis.

The view shows a table containing a number of properties of the complete set of functions of the selected data set \( F(\alpha) \) where \( \alpha \) is the trace represented by the data set is. Every row represents a function and contains the properties of that function. Every column represents one property, where the head of the column contains a descriptive name of its property. The following columns exist for every function \( f \), where the head of each column is shown in italic:

- \textit{Class name}, the class name of \( f \),
- \textit{Function name}, the function name of \( f \) minus the class name,
5.4. Views

- **Full function name**, the complete function name with class name of $f$,
- **Time with children**, the inclusive time $d^{-}$ of $f$,
- **Time**, the exclusive time $d^{+}$ of $f$,
- **Number of calls**, the number of calls noCalls to $f$,
- **Average with children**, the average inclusive time per call $d^{-}$ of $f$,
- **Average**, the average exclusive time per call $d^{+}$ of $f$,
- **Standard deviation**, the inclusive standard deviation of all calls SD to $f$,
- **Memory usage**, the exclusive amount of increase in memory $mem^{-}$, and
- **Memory usage with children**, the inclusive amount of increase in memory $mem^{+}$.

Note that the values shown in the table are dependent on the selection of the intervals as stated in section 5.3.1. The focus of this view consists of a range of cells that are shown. Cells are colored blue if the property represented by its column is the selected data type, and the function represented by its row is in SF. An example of the tabular function view can be seen in figure 5.13, showing a set of functions from the Kalydo engine.

5.4.3 Time line view

This section describes the time line view. The time line view is implemented using the delta stack graph described in section 5.2.1, where time is always represented horizontally. The purpose of the time line view is to:

- see the frame rate of a run in order to find frame drops as per visualization requirement 2;
- show the course of profile values of functions in time so causes of frame drops can be identified, also as per visualization requirement 2, and
- select interesting time intervals to view or compare, as per visualization requirement 7.

Representing the data of snapshots directly would lead to a non-decreasing graph (except when showing memory usage) because snapshots are cumulative. This makes it hard for the user to e.g. see whether functions suddenly use a significant amount of time. We therefore show, for each snapshot $s$ with time $t$, the difference between $s$ and its predecessor $s'$ with time $t'$ (except for inclusive and exclusive memory usage ($mem^{+}$ and $mem^{-}$), where simply the value of snapshot $s$ is shown). This way, the derivative $\Delta f$ is shown for every consecutive interval $(v_{i-1}, v_{i})$. This way, an interval in which e.g. a function uses a lot more time than usual is easily identifiable. The set of snapshot timestamps contained in $d$ is used as the domain $V$ of the graph.

The view has four draw methods:

- **pivot drawing**, where the time line view uses a bar graph to represent the list of pivot frequencies of $d$ (e.g. the frames per second) vertically. The color of the bars are a gradient from blue to red to green, representing low, medium and high values, relative to its maximum. This draw method is used for finding frame drops, so the user quickly knows where the interesting intervals are; these are colored red or possibly blue. A screen shot of the time line view showing the frame rate of a game can be seen in figure 5.14;
Chapter 5. Visualization

Figure 5.14: The time line view, showing the frame rate of a game. Such a variance in frame rate is an indication that the inspected software has performance issues, because under normal conditions, the frame rate should usually be constant.

- **single function**, where the time line view uses a bar graph to show vertically the selected data type of the selected function. Note that the color scheme used in this draw method is different than the color scheme used in the draw method **pivot drawing**: the color of the bars are a gradient from green to blue to red, representing low, medium and high values, relative to its maximum. This is done because a high frame rate is desirable, while a high value here is often undesirable, because it often means that a function is expensive or uses a lot of memory. A screen shot of the time line view showing a single function can be seen in figure 5.15:

Figure 5.15: The time line view, showing memory usage (in bytes) of a single function. Whenever memory usage is shown, a non-decreasing graph such as this one is an indication that a memory leak occurs.

- **single function with children**, where the time line view shows the selected function $f$ and its children $C(f, \alpha)$. The colors of the bars are the global colors of the functions represented. Data types are represented vertically. A screen shot of the time line view showing a single function with its children can be seen in the left image of figure 5.16.

Note that if the time line view is showing a data type that can be either inclusive or exclusive (e.g. duration), a function and its children are treated differently: for $f$, we show its exclusive value, and for all children, we show their inclusive value. This results in that, in case we show durations, not only the exclusive duration of a function is represented, but also its inclusive duration as the complete height of the graph. We can thus show two different data types in one image by making use of the properties of a stack graph. A screen shot of the time line view showing duration of a function and its children can be seen in the right image of figure 5.16;
5.4. Views

Figure 5.16: On the left, we see the time line view showing the number of calls to a function, and its calls to its children. On the right, we see the time line view showing the exclusive duration of a function, and inclusive duration of all its children; note that the total height of the graph is the inclusive duration of the function.

- multiple functions, where all selected functions are represented by their colors, where the selected data type is represented vertically and stacked.

The draw method chosen is dependent on the selection: if $SF = \emptyset$, the draw method is pivot drawing; if $SF$ contains two or more functions, the draw method is multiple functions, and if $SF$ contains exactly one function, the developer can choose between single function and single function with children.

We assume that developers want to focus on functions that are the most time consuming. Therefore, functions are stacked in order of inclusive duration in the final snapshot. The more time a function takes, the closer it is drawn to the baseline in the stack graph, to account for the fact that functions drawn further away from the line are more disjunct and thus less clear. Note that when we use the draw method single function with children, the selected function is always shown at the bottom due to the sorting, because the inclusive value of a parent is always larger than its

Figure 5.17: The time line view showing multiple functions, namely all event handlers occurring in a game using the Kalydo engine. Orange is the render event, and purple is the scripting event. The render event handler seems to fluctuate a lot, indicating that frame drops may occur.
Chapter 5. Visualization

The selected interval \((t, t')\) is shown as a green box, where the width and location of the rectangle depends on the selected interval, and the height of the box is equal to the height of the graph. Similar for the second interval \((t_2, t'_2)\), except that the color of the box is red.

If the draw method is anything other than pivot drawing, the set of pivot timestamps is represented as a set of red vertical bars shown below the delta stack graph, where the horizontal position of the bars is relative to the timestamp. This helps the developer to keep in mind where frame drops occur in the current data set by giving an indication of the performance during the run. An example of this can be seen in figure 5.18.

![Figure 5.18: The time line view, showing duration of a function. Below the graph, the pivot timestamps are represented as a set of vertical bars.](image)

5.4.4 Functions/children and parents view

This section describes the functions/children view and the parents view. These views are implemented using the treemap described in section 5.2.2. Figure 5.19 shows an example of these views. These views both show a treemap with depth at most one. The color of shown leaves is the color of the function it represents, and the size is dependent on the data type selected. Their purpose is to gain insight in hierarchy, and acting as a navigational tool.

In the remainder of this sections the two views are discussed separately.

**Function/children view**

The purpose of the function/children view is twofold:

- to show the user an overall overview of all functions in a data set, and
- to gain insight in call hierarchy of specific functions, as per visualization requirement 3.

The behavior of the function/children view is dependent on the selection:
5.4. Views

Figure 5.19: From left to right, we see the parents view, and function/children view. The used data type is duration. The parents view shows the inclusive duration of the selected function per parent. The function/children view show the distribution of the inclusive duration by showing the inclusive duration per child, and the exclusive duration for the function itself.

- if \( SF = \emptyset \), all functions are shown, where the currently selected data type is used as size. One exception is that the exclusive variant of the data type is used whenever an inclusive data type is selected. The purpose of this is to get an overview of all functions, and to quickly see differences in intervals. A screenshot of this view showing all functions is shown in the left image of figure 5.20;

- if \( SF \) contains exactly one function, it shows a tree map of one iteration deep of the function itself and its children. The child view works in the same way as the time line view using this selection: if applicable to the currently selected value, it shows the exclusive value of the selected function, and inclusive values of children. In the child view, it is possible to see how much time each child accounts for e.g. the total time spent inside this function. A screenshot of this view showing a functions and its children is shown in the right image of figure 5.19;

- if \( SF \) contains two or more functions, the child view shows the selection data type of the selected functions, so the developer can see the relative difference between functions. Their children are not shown to avoid clutter. A screenshot of this view showing multiple functions is shown in the right image of figure 5.20.

Figure 5.20: These images show the function/children view in two different draw methods. On the left, we see the exclusive duration of all functions, where we quickly see which functions are time-consuming. On the right, we see the distribution of the inclusive durations of multiple functions.
Parents view

The purpose of the parents view is to show the user what functions call the currently selected function, as per visualization requirement 3. Furthermore, it acts as a navigational tool, enabling the user to select parent functions of a selected function.

The behavior of the parents view depends on the set of selected functions:

- if $SF = \emptyset$, this view shows nothing;
- if $SF$ contains exactly one function, this view shows a tree map of one iteration deep of the parents of the selected function. The size of the vertices are relative to the contribution of the parent to the value of the data type of the selected function. A screenshot of this view using this draw method is shown in the left image of figure 5.19;
- if $SF$ contains two or more functions, this view shows nothing, because showing the parents of multiple functions becomes hard to understand, as it is not clear which functions correspond to the shown parents.

5.4.5 Profile tree view

Figure 5.21: The profile tree view, showing inclusive duration of the profile tree. The selected parts are brighter than others. A vertex containing the name and a percentage is shown, and a title tip is shown containing the name and value of the vertex.

The profile tree view shows, using a treemap described in section 5.2.2, the call tree. The purpose of the profile tree view is to quickly see call stacks that e.g. use a significant amount of time, as per visualization requirement 1. In the case when two trees are subtracted from each other, it is possible to quickly see the differences between the two data sets or between two intervals, as per visualization requirement 7.

Its input consists of the profile tree of data set $d$. The sizes of leafs in the profile tree are dependent on the data type and the interval contained in the subject. If the data type is an inclusive value, the corresponding exclusive value is used to ensure that the size of the leaf representing the vertex is proportionate to the exclusive value, and the size of the subtree representing that vertex is proportionate to the inclusive value.

Color is used twofold in this view:
5.5 Interaction

- to identify functions, and
- to identify the set of selected functions.

The colors of vertices default to a grayed out version of the color of the function of the vertex by taking the mean of a grayscale and the color of the vertex \( c(\text{last}(\sigma)) \). If the data type is inclusive, every vertex containing a selected function \( \{ \sigma \in V_\alpha \mid f \in \sigma \land f \in SF \} \) is highlighted by omitting the gray-out, and simply use its color \( c(\text{last}(\sigma)) \). If the data type is not inclusive, only leaves representing vertices which last element is a selected function \( \{ \sigma \cdot f \in V_\alpha \mid f \in SF \} \) are highlighted.

A screenshot of the profile tree view can be seen in figure 5.21, where we see a profile tree where one function and inclusive duration is selected as data type.

### 5.5 Interaction

This section describes how the user can manipulate the selection and the focus of the different views.

In the data sets view, it is possible to select a data set, and select or deselect the second data set. It is possible to expand the list of properties of a data set.

In the tabular function view, the user can:

- sort any column, so the attention of the user gets drawn to e.g. functions that have a large variation, or have the largest number of calls;
- select the data type by selecting a column;
- select functions \( SF \) by selecting rows, and
- resize columns by dragging the line between two columns. To hide a column, the user can simple drag a column until it is invisible.

The user can select the global selection interval \((t, t')\) and the second interval \((t_2, t'_2)\) by dragging over the interval that he or she wants to select in the time line view. During selection, the selected area is inverted, so it is clear what is selected.

In the profile tree view, function are selected by selecting vertices; in which case the function of the vertex \( \text{last}(\sigma) \) is selected. To select children of the currently selected function, functions in the functions/children view can be selected. To select its parents, functions in the parents view can be selected.

The focus of the tabular view can be altered by using the scroll bars. If the user desires to focus on a part of the shown interval, it is possible to zoom and pan using the time line view to alter the shown interval. The location of the mouse cursor is used as the center of zooming. If the user desires to know the name of a function that is shown in any treemap view, a title tip can be shown.

### 5.6 Conclusions

In section 5.1 we stated several requirements for our visualization. Using the six views in GamePro Analysis, all of these requirements are met. An evaluation of the resulting tool is given in chapter 6.
5.7 Related work

This section discusses some related visualizations that we could have used to show profiling information. This section also contains explanations why these visualizations were not used.

5.7.1 TraceVis graph view

One of the visualizations that is possibly interesting for showing profile information is the graph view contained in the tool TraceVis by Pieter Deelen [Dee06]. TraceVis is a tool that inspects dynamic aspects of software and focuses on class interaction.

The data format of TraceVis is explained in detail in Pieter Deelen’s master’s thesis [Dee06]. In order to determine the usefulness of this visualization, we created a data logger that generated output in this format.

The data format of TraceVis is based on traces. Every event is stored in a compressed file. A problem which we encountered is that the Kalydo engine generated a vast amount of events, which results in very large files when inspecting it for even a short amount of time. When loading such files, TraceVis generated out-of-memory exceptions.

We generated a trace for 5 seconds of a simple application which is used for testing animation. Figure 5.22 shows the result of this trace. Longer time spans, or collecting data from a real game resulted in data sets which were too big for TraceVis to read.

What we see in this figure is the interaction between classes (around 800 in the Kalydo engine) of the Kalydo engine. Note that we reduced the number of vertices by ignoring the template variables of classes. Clearly, this view is too cluttered to contain useful information. Drawing all vertices (e.g. more than 100,000 in the Kalydo engine) in our data set is not feasible using this visualization. Even drawing all functions (around 6000 in the Kalydo engine) would probably clutter the view, although this is not tested because this is not possible in TraceVis. We therefore did not choose this visualization.

5.7.2 Beamtrees

Beamtrees are a generalization of treemaps developed by Frank van Ham [vH03], where the leaves are scaled in order to view the hierarchical information behind the leaves using depth, thus addressing the problem of the lack of hierarchy visibility, inherent to treemaps. Vertices are shown as stacked circular beams, so that both the hierarchical structure, as well as the size of vertices are depicted. Beamtrees can also be rendered in 3d, where every beam shown in 2D An example of a 3D rendering of a beamtree can be seen in figure 5.23.

Beamtrees are very effective for extracting global hierarchical information of trees. We could e.g. replace the treemap in our profile tree for a beamtree. This would result in a beamtree of around 100,000 elements. We determined that our main focus, namely the sizes of the leaves, are somewhat less clear. We feel that treemapping is a more effective way of emphasizing on sizes of vertices, and less on hierarchy. The use of shading in the form of cushion treemapping results in enough hierarchical information, in combination with the other implemented views, to extract profiling information.

We have unfortunately not tested this method with our own data, because there was no example software available which could import a tree.
5.7. Related work

Figure 5.22: The Kalydo engine inspected by TraceVis, where its input is generated by our own data logger. This visualization generates a lot of clutter because there are so many classes in the Kalydo engine.

Figure 5.23: An example of the beamtree visualization. We can see the hierarchy of leafs clearly; however, the size of leafs are more difficult to determine than treemaps.
Chapter 6

Tool evaluation

In order to determine the usefulness of GamePro, GamePro was tested by Kalydo engine developers at Eximion BV numerous times during the course of this project. This resulted in a large amount of feedback, including feature requests and bug reports.

This chapter contains an evaluation of the final version GamePro.

6.1 Test case

Figure 6.1: A screenshot of GamePro Analysis showing a data set generated from the game RoboBrickz. No functions are selected and no intervals are selected.

This section describes a real-world example of solving a bottleneck and a frame drop in a game using GamePro.

The working title of the game in question is RoboBrickz. Robobrickz is a game where the player controls a spherical robot, which is able to roll around in a level, avoiding obstacles, crossing...
6.1. Test case

checkpoints and gaining points by picking up objects. This game suffered from low performance and major frame drops, and a minor memory leak. Before profiling, the game ran at about 35 frames per second at 100% cpu usage. We therefore used GamePro to profile the application.

The first screen seen after opening the data set in GamePro is shown in figure 6.1. A low variance in frame rate is desirable, and results in a completely green time line view. By looking at the frame rate shown in the time line view, however, we can clearly see that the performance of RoboBrickx fluctuates a lot.

6.1.1 Finding causes of a frame drop

![Figure 6.2: A screenshot of GamePro Analysis showing the time line view and the profile tree view.](image)

We selected an interval containing the largest frame drop, and compare this to the complete run in order to find out what functions are temporarily more expensive.

We first inspect the largest frame drop between 0:28 and 0:30 and determine the cause of it. We thus select the interval containing the frame drop, and compare it to the complete run.

Figure 6.2 shows the time line view, and the profile tree view after the selection of intervals. One function clearly takes way more time than average during this interval, namely the function called TODEApi::collideBetweenSpaces. This function is part of the physics system of the Kalydo engine. The function matrix view shows that the amount of calls to this function is large.

The developer investigated this, and found the reason of this massive amount of calls. The problem that caused the frame drop was as follows: whenever the player picks up an object, the object disappears, and star-shaped objects appear. These 'stars' follow the player until they reach the player; then they disappear. Due to a programming error, these objects were given physical features; which meant that in the physics system treated these objects as having a physical presence. The physics systems was therefore checking whether they all collide with each other. Removing the (unnecessary) physical features of particles solved the problem.

6.1.2 Finding causes of bad performance

Next we inspect the overall low and fluctuating performance. The first thing that we notice in the function matrix view (looking again at figure 6.1) is that the number of calls to several mathematical functions is huge, and much larger than they should be. Clicking on any of these functions, we determine the source of these calls, by finding the most significant parents of these functions (namely, the functions that have the largest size in the parents view) and clicking on these in the parents view. This reveals that they originate from the intersection between an Axis-aligned box and a geometrical pyramid.

This intersection is called by TSceneGraph::processInsideBoundingVolume, which calls a given event handler inside a given bounding volume. How often this function is called, depends on
how the world is divided: the Kalydo engine uses grid cells which subdivide the world in grids, where every grid cell has a tree containing the objects inside the bounding box of that grid cell. Whenever an object occupies a space that spans multiple grid cells, only one grid cell is the owner of such an object, and others link to it. Whenever an object leaves a grid cell, its ownership is transferred to another grid cell. This operation is relatively expensive and should thus be avoided by e.g. enlarging the grid size.

It turned out that the programmer made an error in the creation of that grid by making them far too small. This resulted in many objects being linked in multiple grid cells. The fact that objects were contained in the trees of multiple grid cells resulted in larger trees. Every operation on the trees is thus more expensive than it needed to be, because the cost of such a function is related to the size of the trees.

6.1.3 Finding causes of memory leaks

![Figure 6.3: The function/children view showing elements that leaked memory.](image)

Robobricks also exhibited a memory leak, resulting in a noticeable increase in memory every minute. To find this leak, we filter the complete profile tree, except the vertices containing memory leaks using the modifier system. Figure 6.3 shows the function/children view showing all vertices that allocated memory that was never deallocated. This looks like quite a lot of vertices, but in practice, this is not so bad: all leaks shown here are simply global values that are used for the complete run, so it is not necessary to delete them; except for one, namely the function `TArray<TGameObject *>::operator new`. This function exhibits an ever-increasing memory usage, as can be seen in figure 6.4, which shows the time line view using the draw method single function. This was found by simply clicking on all the functions shown to contain a memory leak, one by one. We determine that this is the cause of the memory leak.

In order to remove the cause, we need to know which function called this function, and be sure that the arrays are destroyed whenever they are not needed anymore. We click on the only function shown by the parents view, namely `tolua_EngineBinding_TArray_TGameObject__new`. This is a function generated by the scripting language Lua, which means that it is called from a script function. After selecting this function, we see that its parent function is `script::createParticleSystem`. This function turned out to create a temporary array of game objects, which was not deleted afterwards. Fixing this memory leak turned out to be trivial by deleting the temporary array afterwards.

6.1.4 Results

The developer fixed the two performance issues and the memory leak described above, and we created a new data set for the game. Figure 6.5 shows GamePro presenting that data set.
6.1. Test case

Figure 6.4: The time line view, showing memory usage of `TArray<TGameObject *>::operator new`. Notice that it is ever-increasing.

The game ran at 65 frames per second at 14% cpu usage. Comparing this to the performance before profiling, the performance is almost doubled, and the efficiency is increased by almost a factor 13. We also notice that the frame rate is very stable, with only a small peak, and a drop at the very beginning. Note that it is acceptable for a few exceptions to exist, e.g. whenever the game is loaded, or a transition to another level takes place. Also, the memory leak is removed.

The cause of the frame drop was found in about ten minutes. The cause of the bad performance was also found in about ten minutes. We improved the performance of the game greatly by using GamePro for only twenty minutes. Further optimizations to this game are not necessary at this point. The cause of the memory leak was found in about five minutes. We conclude that using GamePro, we can efficiently find causes of performance and memory issues.

Figure 6.5: A screenshot of GamePro Analysis showing the a data set created from the optimized version of RoboBrickz.
6.2 Comparison to related work

This section compares GamePro to other existing profilers by performance during run-time, usability, and ease of use. We select a few tools to compare to our own tool. We focus on performance profiling, because that is the main focus of this project, and some tools do not contain memory profiling capabilities. To compare results, we compare the largest exclusive times.

6.2.1 Benchmark values

We use the game called Oggo’s Odyssey which contains a minor frame drop at the start of the level Tutorial 3. We play that level for about one minute while profiling the game.

The frame rate drop and its cause is found quickly using GamePro as can be seen in figure 6.6: we click on the drop shown in the frame rate view, and look at the overall tree map view. Two functions clearly show a large amount of CPU usage during the drop, namely TImageIOSystem::load and TOpenGLApi::loadTexture, which are using respectively 18% and 16% of CPU time during the drop. These functions are called whenever an image is used which is not yet in memory. A solution to this frame drop is to preload the textures.

The overall performance of this game is considered to be good. The functions with the largest exclusive time measured are:

- TWin32OS::sleep, we see that overall, the process sleeps for about 70% of the time (consistent with the 30% cpu usage of the original);
- TOpenGLApi::drawElements, which exclusive time is 2.7% of the total time and is used for drawing objects to the screen, and
- TWin32OpenGLApi::swapBuffers, which exclusive time is 1.7% of the total time.

These values tell us that the bottleneck is at the rendering of the screen. This is exactly what the developer would expect.

The following sections describe commercial profiling tools that were tested for performance and usability. Note that no screenshots are included in this chapter due to copyright reasons.

6.2.2 AutomatedQA AQtime

AutomatedQA AQtime [AQt] is a commercial software inspection tool, and the performance profiler used at Eximion BV before this project started. The largest complaints from the developers were the lack of timed information (thus making the search for causes of frame drops very hard) and low performance during data logging. This section describes a concise evaluation of the tool, and compares it to GamePro.
6.2. Comparison to related work

AQtime is a complete software analysis tool, consisting of multiple tools that collect static and dynamic software information, such as an allocation profiler, performance profiler and a platform compliance check.

We focus on the performance profiler, which is an offline, instrumenting profiler like GamePro. AQtime uses bytecode instrumentation for instrumenting applications, unlike GamePro, which uses compile-time instrumentation. This is an advantage over GamePro, because it removes the need for a recompile. AQtime is also able to instrument code written in .NET, also known as managed code, which GamePro does not. An extension to our data logger is needed for this to be possible.

The main view of AQtime is similar to the function matrix in GamePro. AQtime contains some hierarchical information similar to the information found in GamePro’s children and parents views. Expensive functions can be usually traced back to their parents.

The collected data is shown quickly, because the interaction between the data logger and the presenter is done using system memory. GamePro uses the hard drive for this.

It is however hard to find frame drops and their causes in AQtime. It is possible to take snapshots during run-time by clicking on 'Get Results', but it is not possible to compare these other than showing them side-by-side.

According to AQTime, the functions with the largest exclusive time measured are:

- TWin32OS::sleep; we see that the process is sleeping for about 27%, which is probably mostly done in the menu prior to playing the game, but this is not verifiable;
- TColladaParser::parseFloatArray; which is a function heavily called during preloading, and probably not during running of the game;
- TRenderAgent::update; which exclusive time is 4.31% of the total time. This is a bit odd because this function usually is pretty fast because not much work is done inside it. Further investigation reveals that it calls TWin32OpenGLApi::swapBuffers which is expensive. For some reason, this function is invisible in AQtime and treated as inline, hence the expensiveness of the parent function, and
- TImageIOSystem::load; which is also done during preloading, if the game developer did his work correctly.

The frame drop isn’t visible at all as there is no view present containing timed information.

6.2.3 GlowCode

This section describes GlowCode from Electric Software. Their website [GC] promisingly states that GlowCode is the world’s fastest profiler. GlowCode is an instrumenting, online profiler that, like AQtime, uses bytecode instrumentation. Their statement that it is the world’s fastest profiler is a bold one due to the fact that the data is presented during run-time.

As with AQtime, its main view is a function matrix view. Because it is an online profiler, the values contained in it are constantly shifting. It is possible to sort by any column, but only descending. An annoying feature is that whenever a function is newly called, it is added at the top of the matrix, so the sorting is invalid. This is confusing to the developer. It also contains a complete call tree view; where a tree similar to our profile tree can be seen. This is however shown similar to a file explorer, which is not very convenient: it only shows numerical values for every function, making it hard to compare costs of functions.

The most expensive functions shown are:
• TWin32OS::sleep: we see that the process is sleeping for about 53%, which is consistent with the cpu usage;

• TRenderAgent::update; which exclusive time is 12.92% of the total time. This is again probably including the time spent in the function swapBuffers, and

• TOpenGLApi::drawDisplayList: which exclusive time is 4.40%. This is probably mostly spent in the renderMesh function.

We notice that many functions are not shown. We assume that many events are not handled. This is probably why the profiler is so fast. Also, GlowCode exhibited some performance glitches; the inspected game stalled about once per two seconds for a short while which may have affected profiling values.

As with AQtime, the frame drop is invisible as well as its cause. There is a trace buffer viewer, that shows the latest events, but this goes way too fast to get any useful information from it.

6.2.4 Performance Validator

The company Software Verification creates a wide range of products for a wide range of languages. One of their products, Performance Validator [PV] is a performance profiler which we inspect in this section.

Performance Validator is an instrumenting performance profiler which is both online and offline. It too uses bytecode instrumentation.

We notice that there are many columns in Performance Validator’s main (function matrix) view, which clutter the view a lot. For each value shown, there is a counterpart in percentage, which is not always a useful value (there is e.g. a call amount percentage view, which has no apparent use). A complete profile tree is stored and can be shown in a ‘Call tree view’. This view is very similar to the one in GlowCode, and thus very cluttered and not very useful. The tool is not very responsive during run-time. When used as an offline profiler, it performs well.

According to Performance Validator, the most expensive functions shown are:

• WinMain; which exclusive time is 41.45%. This strikes us as completely odd;

• TRangeChecker::visitMetaLayer; which exclusive time is 3.92% of the total time;

• TWin32OpenGLApi::swapBuffers; which exclusive time is 3.55% of the total time, and

• TNodeUpdater::updateNodes; which exclusive time is 2.79% of the total time.

Note that these values are wildly different from the values gathered from other profilers. We therefore assume that these values are incorrect. This is probably due to the fact that this software slows down the software to such an extent, that the timing of events handles in the Kalydo engine is totally different. We therefore claim that this profiler is not suitable for profiling real-time applications.

Causes of large frame drops can be found using Performance Validator, because it is possible to view the call stack at real-time. A prerequisite is that the drop is caused by a hotspot: a subtree of the profile tree where a large amount of time is spent. The finding of the cause of the frame drop in Oggo’s Oddyssee is infeasible using this, because the user has to recognize the frame drop, and read the call stack in around 0.3 second.
6.3. Other success stories

6.2.5 Performance and memory comparison

For comparing performance during run-time, we use the frame rate as a reference. The application Fraps [Fra] is used for counting the frame rate. We also take CPU usage into account because the frame rate is restricted to 40 fps. If the game reaches that point, the software idles until it needs to do something again. We thus compare frame rate per CPU usage. We also compare memory usage, where the original memory usage is used as a benchmark to get the total amount of memory added by the profiler. The following table shows, per profiler, their memory usage, and the factor of the decrease in performance (e.g. a 2 means it was twice as slow) of the inspected software during profiling.

<table>
<thead>
<tr>
<th>Profiler</th>
<th>Performance penalty</th>
<th>Memory usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>GamePro</td>
<td>1.33</td>
<td>174 MB</td>
</tr>
<tr>
<td>AQtime</td>
<td>5.3</td>
<td>28 MB</td>
</tr>
<tr>
<td>GlowCode</td>
<td>1.33</td>
<td>147 MB</td>
</tr>
<tr>
<td>Performance Validator</td>
<td>19</td>
<td>94 MB</td>
</tr>
</tbody>
</table>

From this table, we can learn that GamePro and GlowCode are equal in performance. We thus see that requirement 1 that states that performance should not drop below one-third is met in this case. Note that we do not falsify the claim that GlowCode is the fastest profiler available. AQtime is slow, but may be acceptable in our case. The performance of Performance Validator is totally unacceptable for profiling real-time applications.

Note that GamePro uses far more memory than other profilers. This is probably due to the storage of the snapshots of the profile tree. This may be a drawback if we collect data on memory intensive software, or whenever the host computer does not have much memory. In practice, this has not been a problem.

6.2.6 Usability comparison

The performance profilers we tested contain no timed information. It is therefore very hard to find causes of frame drops, because even their occurrences are not visible in these tools. It may be possible to see them in online tools, but only if the tester is very quick, which is not very convenient.

Some profilers do not show some functions that have significant cost, or values that seem inaccurate or incorrect, while other insignificant functions are shown to be significant. GamePro seems to detect all significant functions. The accuracy of the values shown in GamePro are hard to verify, but because we are able to find bottlenecks and causes of frame drops, we state that they are accurate enough to contain useful information.

6.3 Other success stories

This section briefly discusses other success stories of GamePro.

The following performance issues could also be found using other performance profilers:

- a heavily used array retrieval operator was declared virtual. This meant that:
  - the function did not get inlined. Inlining this function was desirable, because performance could be improved due to the massive amount of calls to this function, and small body of this function, and
  - every time the function was called, its address was looked up in a table, which resulted in even more slowdowns.
The performance of Konstrukt Invasion was considered to be poor; GamePro showed a massive amount of calls to the collision detection function, resulting in large CPU usage. After investigating this, it turned out that every enemy had multiple range checkers which could easily be replaced with a lot less range checkers that originated from the player. This change resulted in a large performance increase.

The following performance issues were hard, or even impossible to find using other performance profilers:

- in the game Oggo’s Odyssey, some textures and meshes were not preloaded. Whenever an affected object was shown for the first time, the IO system needed to load that file, and the player would experience a frame drop.
- A scripting function in the game Oggo’s Odyssey, which converted parts of the level to transparent meshes whenever they would stand between the camera and the player would occasionally cost thrice its normal time. This also resulted in the player experiencing frame drops.
- Konstrukt Invasion’s game developer forgot to preload its sounds, therefore, whenever a sound was played for the first time, a frame drop would occur because of the disk activity.

6.4 Conclusions

GamePro Data logger is a fast, powerful collector of profile data. It can inspect native and script functions, with only minimal changes in source code. Its snapshot functionality is a powerful way of collecting information on frame drops.

In GamePro Analysis, developers can quickly find bottlenecks, frame drops, and their causes. Locations of memory leaks are easily detectable. Data sets are easily comparable so changes in performance between runs are easily detectable. Other profilers turned out not to be very useful when profiling games.

GamePro is more suitable for the performance profiling of real-time applications than other tools we tested. A notable difference between GamePro and other tools is the fact that frame drops can be seen, including their causes. We conclude that we fulfilled all requirements stated in section 1.3.
Chapter 7

Project evaluation

This chapter contains an evaluation of the entire project. This project had the following main activities:

- researching what developers want to know about their software. The results were used for the design of the data logger and the visualizations;

- visualization research. During the course of this project, different ways of visualizing profiling data were researched to accommodate the desires of the users;

- profiler research. Many different profilers were inspected to get an idea of what was useful, and what was missing in current profilers and how to fill in these missing parts;

- collecting data. A lot of time was used to develop the data logger (described in detail in appendix A), until we settled for the current solution of creating profile trees and snapshots during run-time, and

- implementation of the tool. The tool was used to experiment with different ways of visualizing the data. We had some ideas about what we wanted to visualize, but we did not have a detailed design beforehand of how to visualize profiling data at the start of the project. During the course of this project we experimented with visualizations and evaluated their usefulness.

During the course of this project, the following lessons were learned:

- the creation of a fast and useful data logger is far from trivial. We eventually succeeded in creating a logger that was fast, collected a lot of useful data about structure and time, and had acceptable memory usage. Unfortunately, this took a lot of time, and alterations to the data logger were done throughout the entire project;

- the storage and manipulation of large data sets can be complicated, because large data sets can consume a lot of disk space and memory. Furthermore, it is hard to keep the responsiveness of a tool acceptable while handling large data sets. Filtering, and profiling the tool itself proved to be an effective way of keeping the responsiveness of the tool acceptable;

- a good planning is important. The data logger took a long time to create, time that could also be used for researching visualizations or user wishes;

- treemaps are an excellent way of visualizing performance and memory profiling information, in combination with a profile tree and filtering. A small amount of screen space is needed to show the complete profile tree, and most users quickly understand the meaning of them;
Chapter 7. Project evaluation

- profiling your own software is fun, and optimizing it is addictive. One should take care not to spend too much time optimizing parts of code that are already efficient enough, and
- pointing out performance issues is one thing, but fixing them is another. The developer needs to have a good understanding of what the functions do, and needs to know why they are expensive before fixing them. Profilers cannot help the developer with that.

7.1 Future work

Some suggestions for future work are:

- Experiment with bordered treemaps, so the hierarchy of trees is returned. The size of the border can be e.g. an exclusive values of the vertex itself.
- researching the use of textures in the current visualizations. Danny Holten wrote an excellent report on the use of textures in visualizations in [Hol05]. We could use textures in our visualization to show an extra data type, and thus create a broader view of the data;
- extending our data logger to include managed code, also known as .NET code;
- make a better use of the natural hierarchy of classes in C++. We could e.g. use different coloring schemes, to enable the developer to find subsystems that can perhaps be refactored to improve performance;
- research of other visualizations that can show perhaps a combination of functions and time in a concise way. For example, our (delta) stack graph is not convenient for showing more than about ten functions, vertices or classes;
- show a screenshot of the inspected application for every snapshot, so the developer knows what was happening in the application at the time we are taking snapshots. A frame drop when e.g. loading a level is not a large problem, and
- enable the user to select vertices instead of functions. Our efforts to do this resulted in confused developers, that had trouble understanding why they were only seeing parts of the functions they were inspecting.

7.2 Conclusions

In section 1.3 we defined the goal of this project to be the development of a tool that allows developers of real-time applications to gain insight on causes of low performance and frame drops. Chapters 2 to 5 describe the development of the tool. Chapter 6 contains an evaluation of the tool. We introduced the use of negative values in stack graphs by defining the delta stack graph, and in treemaps by introducing negative highlights. Also, Kalydo engine developers are pleased, because the engine’s performance and memory issues can be found in a short period of time.

The conclusion is that GamePro is a useful tool and offers many improvements over existing profilers when profiling video games. A notable difference between GamePro and related work is that GamePro enables developers to find causes of frame drops by incorporating the concepts of snapshots.

Because the data logger provides enough meaningful information, and GamePro Analysis is well structured, it is relatively straightforward to improve existing views or add new views.
Appendix A

Data logger

This appendix discusses the design of the data logger, including design decisions and encountered problems. Its purpose is to act as a reference for people who want to create a data logger themselves.

A.1 Design

A.1.1 Global design

The data logger can be treated as the following distinct parts:

- debug hooking part, which captures control from the profiled program and retrieves the address of the current function;
- local stack creating part, to enable the logger to know which function called the currently entered or exited function;
- the tree generation part, and
- the storage part.

Hooking

In order to instrument any piece of software, the data logger should be able to gain control of the program at entry and exit of procedure in order to determine the time spent in such a function. There are several ways to enable the data logger to gain control over the profiled software:

- altering of source code, so that every entry and exit is manually added by the user.
- byte code instrumentation, which alters the compiled code;
- using debug hooks from the compiler, which alters the code during compilation.

The first alternative is undesirable, because requirement 3 states that source code should not be altered would be violated this way. The second alternative is good; however, binary alteration of compiled software is complicated and not portable. Since the focus of this project is with visualization, this method was not chosen.
GamePro uses debug hooks, where its implementation is based on the method described in [Pan99]. The method describes a binary alteration of the inspected software by recompiling and linking using a compiler flag that allows external software to gain control over software without altering the source code. This method was thus chosen as it matches the requirements very neatly. The only drawbacks are that the compiler needs to have the functionality of debug hooks, and a recompile is necessary.

A major difference between the debug hooks of GamePro and the method described in [Pan99] is that GamePro uses the `pexit()` debug hook, while the original method did not; due to the fact that the original document was written for an older compiler that included the compiler flag `/Gh` (creating the `penter()` hook at entry of functions), but lacked the compiler flag `/GH` (creating the `pexit()` hook at exit of functions). Using both compiler flags, the hook functions can be much cleaner and simpler.

Using these compiler flags, all functions are compiled as if they have the following form:

```c
{ 
  _penter();
  ..original function..
  _pexit();
}
```

Furthermore, all return commands within the original function are also augmented with `pexit()` right before the actual return.

We use the return address contained in the stack to determine the address of the function. To determine the name of functions, we retrieve all function names and address combinations known by the process before the collection of data starts.

In order to know which thread contained any event, we use a concept called thread local storage, where data is stored local to a thread. We store a pointer to our logger class, so every thread gets their own logger class. This way, a profile tree is generated for each thread.

**Script hooking**

The hooking of Lua functions is done using the debug hooking mechanism incorporated within the Lua interpreter. Sadly, the implementation of these functions require that the code of the profiled software should be changed, while requirement 3 states that the source code should not be changed. This was however accepted by the client, as the change in code is very minimal, and consists of a `lua_sethook` function call, and a hook handler function that calls the `penterScript()` and `pexitScript()` functions implemented in GamePro. Requirement 3 mainly focuses on the fact that we do not want to manually add profile hooks for every function.

These scripting functions are then treated the same way as the native functions, and stored in the profile tree the same way.

**Memory hooking**

In order to know when memory is allocated or deallocated, we need to hook allocations and deallocations. Because the Kalydo engine contains its own memory manager called the memory pool for fast allocation and deallocation, we created hooks manually in this manager, calling our own `addMemUsage` and `remMemUsage` functions. This, again, violates requirement 3, but again, this was not a problem in practice.

If we do not have a memory manager, adding hooks manually is not accepted. We thus use the debugging hooks in `malloc`, which is able to call a user-defined function. In GamePro Data Logger
A.1. Design

there is a wrapper called _crtAllocHook which handles these hooks. A problem with these hooks is that the debug version of the Microsoft Visual C Run-Time Library is needed for the hooks to work. Using the debug version of this library affects profiling values; so this makes it harder to profile memory usage and performance at the same time.

A.1.2 Stack creating part

We need to know the stack of the entered or exited functions, in order to know where in the profile tree the information should be stored. We therefore store a list of functions that represents the stack of the software that is being profiled.

The method used is very similar to the stack creator of John Panzer [Pan99]. However, this implementation uses the STL stack [STL94], which is powerful and bug-free. Two drawbacks occur when using the STL, namely that most functions are inlined, which usually results in larger code, and it is somewhat slower than a dedicated implementation. We therefore chose not to use anything from the STL to enhance performance.

Also note that whenever the function encounters an exit event, it is not checked whether the function that is removed from the list is actually the one being exited. This means that this operation is only valid if the trace is well-formed.

A.1.3 Tree generation

As performance is preferred over memory requirements, the complete call tree is stored. This means that whenever a function is called, only the called functions of the current parent function need to be checked whether they match the current address. It also means that it is possible that functions can be found on multiple places in the tree.

The children in the tree are stored using hash tables, again, as performance is preferred over memory requirements. The hash table is the size of a prime, and the hash function is the address of the function modulo the size of the hash table. This results in the fact that the lookup of a function only takes up one iteration in most cases. Also the previously mentioned function identification array is stored using hash tables.

We do not store complete snapshots, we store them as a list of profile information per vertex. Whenever we decide to take a snapshot (e.g. every second, or at the pivot function; the user can decide), we traverse the complete call tree in order to check whether any values have changed since the last snapshots. If so, a snapshot of the corresponding vertex is stored.

A.1.4 Storage on disk

The profile data is stored on disk using XML [BPSM+06]. This was chosen because:

- the data format is readable by anyone, without using special tools, and
- the availability of an XML serializer inside the Eximion BV code base.

Both of these decisions were made for speeding up the development. When something was wrong with profile data, it was easy to check the file in order to find errors.

The serializer of Eximion BV unfortunately did not meet the scalability factors needed for our data due to its massive size. Therefore, a rewrite was done to make the files smaller by removing redundant information, and ensuring that it created a minimum of temporary objects to enable large data structures to be stored and loaded. This, however, took longer than expected, and eventually probably slowed down the project.
XML is a widespread format nowadays, partly because of its flexibility and because it is easy to understand and use. Many tools exist to work with it, and the structure of any XML file is usually clear.

The structure of the data format is dictated by the Eximion BV serializer. It works in the following way when storing an object:

- A RTTI (Real-Time Type Information) system is used to determine the properties of the object which is to be stored.
- For every property, do:
  - if the value of the property is 0 or NULL or equivalent, skip.
  - otherwise, open an XML tag, store its value, and add a closing tag.
- If the property is a base type (e.g. an integer or floating point), store its numeric value in plain text.
- Otherwise, the property is treated as an object, so its value is stored recursively in the same way.

There is obviously more to the serializer, such as exceptions for certain objects such as strings, bags and matrices, or whenever a property contains an object other than the one expected, but this is beyond the scope of this document. Whenever an object is read from a file by the serializer, the same scheme is used, but instead of storing the value, it is read and parsed to recover the original value.

The advantage of this scheme is that it can store any object for which the RTTI system can determine its properties. The disadvantage of this scheme is the fact that this is heavily dependent on the implementation of the data structure. If the structure changes (e.g. the developer decides to rename a property), all previously serialized objects are unreadable by the serializer. This was a problem encountered whenever we decided to alter the data structure; therefore discarding all previously stored profiles.

Another disadvantage is that the XML files can become pretty large when storing a lot of data. This can be solved if we compressed the file in some way. However, the storing and loading of profile data takes a long time, and this is not solved by compressing it.

In hindsight, we probably should have created our own, binary format in order to generate smaller files, less dependency on implementation and faster stores and loads.

The exact format of the file that stored on disk is described in appendix B.

### A.2 Timing

For timing purposes, GamePro uses the Read Time Stamp Counter (RDTSC) instruction, unique for Intel Pentium or compatible processors. This instruction returns the amount of machine cycles executed since bootup of the processor core or cores.

The RDTSC function can return false or imprecise results in the following cases:

- Whenever a thread jumps processor cores, the result of RDTSC can jump forwards or backwards. This can result in GamePro showing a function taking up negative time, or too much time. A workaround is to make sure threads remain on the same processor core, which GamePro does.
A.3. Portability

- If the processor has throttling capabilities, results can be different on different clock speeds. The user of GamePro should make sure all throttling capabilities are turned off before using this software.

- Some processors (like the Pentium II and Pentium Pro) may perform their instructions out-of-order. This can result in strange measurements whenever e.g. the RDTSC function is called before (part of) the function that is being measured. In practice, this has not been a problem; however future processors can not be tested. Whenever this problem occurs, the RDTSC function should be preceded by a serializing instruction, such as CPUID. This does however slow down the profiled application.

Alternatives were:

- `QueryPerformanceCounter`, a windows API that returns a clock which is guaranteed to never change frequency. In some cases, QPC uses the RDTSC instruction. QPC is a good alternative, however, windows API calls are relatively slow. As the timer function will get called literally millions of times per second, the RDTSC is preferred over this one.

- `TimeGetTime`, which retrieves the system time in milliseconds (at best). Milliseconds are really not precise enough for purposes of profiling; therefore, this timer also is not a viable option.

- `QueryThreadCycleTime` or `QueryProcessCycleTime`, a windows API returning the number of machine cycles a certain thread or process is actively running. However, it is only available using Windows Vista or Server 2008. This violates the user requirement that it should work on Windows XP. Furthermore, as windows API's are slow, this is not done.

A.3 Portability

GamePro incorporates some non-portable code. Although portability was not a goal of GamePro, some effort is put into isolating the non-portable parts.

The following parts of the software are specific to the MSVC2005 compiler:

- the memory and function debug hooks, and
- the retrieval of function names.

The following parts of the software are specific to Intel or compatible hardware:

- the retrieval of machine ticks through the RDTSC instruction;

The following parts of the software are specific to Lua:

- the usage of Lua debug hooks through the C command `lua_sethook` and a hook handler function.

A.4 Problems

This section describes some problems encountered when creating the data logger.
A.4.1 Recursion

A recursive function is defined to be a function that calls itself, or is eventually called by a child function. Such a call will be called a recursive call. Profiling these calls can pose some problems. Because we store information for each possible call stack, every iteration of recursion is stored separately. This results in that profile trees can be large. If memory requirements are tight, this could pose a problem when profiling software which has at least one of the following characteristics:

- there are one or more occurrences of recursion, where a lot of different functions are called per recursion step;
- it contains deep recursion (e.g. recursion consisting of more than 1000 iterations), and
- it contains recursion at lots of places in the call tree.

Whenever recursion occurs (which means that functions with the same address occur more than once in the same call stack), the function that was called recursively is not stored individually; rather, all its information is stored in the original called function. This reduces the call tree size (and thus memory usage) at the expense of speed.

The following advantages apply to this method:

- the memory usage when profiling recursive functions will be extremely low, as the whole function will only be stored once in memory, and
- the problem of wrongly calculated time with children of the recursive function when flattening the tree is non-existent, as recursive functions do not exist as such anymore in the stored call tree.

The following disadvantages apply to this method:

- information about the recursion is lost; such as the structure and depth of the recursion;
- recursive functions are not detectable as such when examining the call tree, and
- the time with children of functions in between recursive calls will be wrong, as they do not contain the time of the second recursive call.

A.4.2 Exceptions

Whenever an exception occurs, it clears the stack up to a certain, predefined point. This means that for all functions that were called in the cleared portion of the stack, no instrumentation is done and we miss one or more function exit events. Our trace therefore loses its well-formedness.

Because we use the program counter for identification of functions, it is hard to check whether the exit event encountered is actually the event we encountered. We therefore store the parent frame in our own call stack, which is retrievable from the ‘real’ call stack [Pan99]. We then check at an exit event whether this parent frame matches by checking it against the final function on our call stack. Note that parent frames are not unique and we can get false positives when an exception has occurred. In practice, these false positives are not a problem, because eventually the function entries and exits are matched again. Note that the profile values gathered between the exception and the eventual match are wrong, so exceptions should thus be avoided.
Appendix B

Data format

This appendix describes the data format on disk.
A profile file contains a tree of objects representing the vertices described in chapter 4. Because the structure of snapshots are all roughly the same, we chose not to store a tree for every snapshot, but rather only one tree, containing a list of the changed data per snapshot, including a time stamp.

B.1 Prelude

The file always starts with the following header (note that every line of text in XML format described below is stored on a separate line, unless otherwise stated):

```xml
<?xml version="1.0" encoding="utf8" standalone="yes" ?>
```

After this header, it is always augmented by a simple tag on a separate line:

```xml
<xml>
```

Then, the identifier of the actual serialized object is stored:

```xml
<ROOTCLASS type="TFuncInfoTree">
```

After this, all function identifiers are stored in a bag:

```xml
<FuncIDs>
```

The number of function identifiers (in the example 481) is stored in the next line:

```xml
<FuncIDs size="481">
```

Then, for each function that is profiled, the following scheme is used, where

- i0 is the index number of the value and should be increased for every stored function identifier,
- 4286481 is the address of a function in decimals, and
- TClass::funcName<templateParam> is the name of the function number of function:
After the last function identifier, the following closing tags are added:

```xml
</FuncIDs>
</FuncIDs>
```

The amount of machine ticks per second executed by the computer on which the run is stored next in the following way, where, in this example, our processor is a Pentium M running at 1.7 GHz:

```xml
<TicksPerSec>1694173157</TicksPerSec>
```

Next, another bag is stored containing the time stamps of the snapshots stored, in this example 69 snapshots:

```xml
<Timestamps size="69">

where every snapshot is stored consecutively, in machine ticks, where 0 is regarded as the start of the software:

```xml
<i1>169417469</i1>
<i2>338834809</i2>
<i3>508252022</i3>
```

Note that this bag starts at 1, however, 0 is regarded as the first snapshot in this file format: it represents the snapshot at the start of the application. However, this time stamp is 0 and thus is implicitly stored due to conventions used by the serializer.

After all timestamps, we need a closing tag:

```xml
</Timestamps>
```

Next we store the timestamps of all exits of the pivot function. This is done similar to the timestamps bag, although the tags are renamed from Timestamps to PivotTimestamps.

After the pivot function, we store the date and time of this data set in plain text:

```xml
<DateTime>Tue Jul 01 20:14:30 2008</DateTime>
```

Next, the actual vertices are stored.

### B.2 Vertices

All vertices are stored using the following properties:

Address, which contains the address of the last function of the vertex, matching the function identifier stored earlier:

```xml
<Address>4286481</Address>
```
B.2. Vertices

TimedLogs, a bag of time stamped logs, which are stored in the following form:

<TimedLogs>

Then, store the bag:

<Array size="69">

Every element has a property Key and Elt, where Key consists of the time stamp of the snapshot (stored previously in the 'Timestamps' bag), and Elt consists of a function log (note that in the first iteration, the key may be 0 as that is the first snapshot, and thus is not stored explicitly; also note that whenever all values of a log are zero, the same applies):

<i0>
    <Elt />
</i0>

<i1>
    <Key>169417469</Key>
    <Elt>
        <NoCalls>1</NoCalls>
        <Duration>160359157</Duration>
        <DurationSquared>25715059233750649</DurationSquared>
        <AllocatedBytes>16</AllocatedBytes>
    </Elt>
</i1>

where every element consists of the number of calls:

<NoCalls>1</NoCalls>

and total duration stored:

<Duration>160359157</Duration>

also the squared duration, in order to calculate the standard deviation:

<DurationSquared>25715059233750649</DurationSquared>

and then the allocated bytes, if applicable:

<AllocatedBytes>16</AllocatedBytes>

finally the log is closed:

</Elt>

and the element too:

</i1>

Note that if a vertex has not changed since the last snapshot, its values are not stored.

and after the last element, the bag is closed:

</Array>
</TimedLogs>
After that, a bag containing every child of the corresponding vertex is stored:

\[<\text{Children size="12"}>\]

where each child is stored as

\[<\text{i0}>\text{vertex}\</\text{i0}>\]

where vertex is replaced by the format of this section.

After all children, we need to close the tag again:

\[</\text{Children}>\]

and then the vertex is stored completely.

### B.3 Closure

Finally, the file contains the following at the end, in order to close all open XML tags so the file contains well-formed XML:

\[</\text{ROOTCLASS}>\]
\[</\text{xml}>\]
Bibliography


[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Professional, 1994.


