Clinton Jeffery · Jafar Al-Gharaibeh

Writing Virtual Environments for Software Visualization

Springer

Writing Virtual Environments for Software Visualization

Clinton Jeffery • Jafar Al-Gharaibeh

Writing Virtual Environments for Software Visualization

Springer

Clinton Jeffery
Department of Computer Science
University of Idaho
Moscow
ID
USA

Jafar Al-Gharaibeh
R&D Department
Architecture Technology Corporation
Eden Prairie
MN
USA

ISBN 978-1-4614-1754-5 ISBN 978-1-4614-1755-2 (eBook)
DOI 10.1007/978-1-4614-1755-2

Library of Congress Control Number: 2014957537

Springer New York Heidelberg Dordrecht London

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

Preface

This book combines two emergent research areas of the computer graphics world. On the one hand, there is software visualization. After 30 years, it remains an under-achiever with enormous potential to impact how we develop software. This book looks at how to visualize dynamic program behavior, while it is happening. The techniques are readily applicable to other areas of software visualization. Our ultimate goal is to make visible many practical aspects of program behavior that are currently invisible or difficult to see at best, such as how to find bugs and performance bottlenecks.

A second major focus of this book is on virtual environments. Lots of programmers want to create them, but for most that challenge is just *too* hard; virtual environments integrate advanced 3D graphics, animation and networking in ways that most ordinary developers can't manage in a practical time frame. We are writing this book to do what we can to help conquer that obstacle.

It is obvious that these two genres of computer graphics should be combined. The networking and persistence afforded by virtual environments are exactly what software visualization needs in order to become more collaborative, shared, ubiquitous, and educational. All we have to do is figure out how to build it.

Unicon is an innovative, very high-level programming language hosted at unicon.org and the popular open source site SourceForge. It is descended from Icon, a language developed at the University of Arizona as a successor to the SNOBOL language family from AT&T. Unicon's 3D and networking facilities turn out to be great for virtual environments, and its execution monitoring facilities make it second to none in the area of software visualization. Programs in other languages can often be instrumented to extract similar program behavior information, but in Unicon 120 or so kinds of program execution behavior are continuously available, any time they are of interest.

This book is organized into two parts. Part I is an overview of execution monitoring and program visualization. It presents Unicon's monitor architecture and the framework for monitoring Unicon programs using a series of example visualization tools that observe many kinds of execution. Part II presents virtual environments, including 3D modeling and network communication necessary for multi-user collaboration. It introduces methods of propagating visualization

information and collaborating within a multi-user virtual environment. The ultimate goal is to produce game-like real-time graphical ways of understanding and talking about bugs, bottlenecks, and other aspects of program behavior. Following Part II is a collection of appendices including some detailed program examples and a description of the implementation of the monitoring framework.

Acknowledgments

Intellectual contributions to our work were made in the University of Idaho and New Mexico State University graduate courses on program monitoring and visualization, and on collaborative virtual environments. The authors wish to thank the students from those courses.

This book descends from the 1999 tome *Program Monitoring and Visualization*, whose acknowledgements section recognizes Ralph Griswold, Rick Snodgrass, Mary Bailey, Norm Hutchinson, Wenyi Zhou, Kevin Templer, Gregg Townsend, Ken Walker, Michael Brazell, Darren Merrill, Mary Cameron, Jon Lipp, Nick Kline, Song Liang, Kevin Devries, Steve Wampler, Thanawat Lertpradist, Laura Connor, Anthony Jones, Khan Mai, and Niem Tang.

Brett Kurzman of Springer is responsible for initiating this book. Mary James and Rebecca Hytowitz at Springer provided the kind of patience and support that all authors want.

This work was supported in part by the National Science Foundation under grant DUE-0402572, and by the National Library of Medicine Division of Specialized Information Services.

Moscow, Idaho and Eden Prairie, Minnesota, September 2014

Clinton L. Jeffery and Jafar M. Al-Gharaibeh

Contents

List of Figures

List of Tables

Chapter 1
Introduction

As the demand for computer software grows to include more diverse and larger applications, software developers desperately need better tools to aid in the understanding of dynamic aspects of program behavior during various phases of the software life cycle, including debugging, performance tuning, and maintenance.

Spectacular improvements in graphics, concurrency and networks during the past decade have put us in a position where we ought to be able to see what is going on while our programs are running, in rich, animated 3D, and to easily share and consult with each other while studying such visualizations of program behavior. It ought to be no harder than, say, playing *World of Warcraft*.

This book applies videogame technology to software development by making the vision of multi-user online collaborative 3D software visualization a reality. The topic will be explored by demonstrating an example implementation, getting down into its code.

1.1 Software Visualization and Program Behavior

The reason to want software visualization is to gain a better understanding of a program's behavior. Some program-understanding systems convey very specific information about a small portion of a program, such as the workings of a single algorithm. Others are concerned with explaining the role that a program or a collection of programs plays within a larger computational system. This book addresses a common problem in between these two extremes: understanding the workings of a single (possibly large) program.

People who need to understand a program usually have two alternatives: studying the source code, or running the program to see what it does. Ideally, a program would be understandable using one or the other of these methods; in practice, reading source code is impractically cumbersome for many programs, and construction of test cases to explain program behavior is often a tedious and speculative undertaking. These difficulties motivate the development of special programs that are used to help explain the behavior of other programs.

C. Jeffery, J. Al-Gharaibeh, *Writing Virtual Environments for Software Visualization,* DOI 10.1007/978-1-4614-1755-2_1

Program-understanding systems are used in a variety of applications. The most common motive for program-understanding is *debugging*. Programs that produce incorrect output or fail to complete their execution due to bugs are prime candidates for tools that assist program developers and maintainers in program-understanding tasks. A *debugger* is a program designed specifically to help with the debugging process. General-purpose program-understanding tools are also used to assist in *debugging*.

A second major application of program-understanding systems is *performance tuning* or *performance debugging*. A correct, working program may be of limited usefulness if its performance is poor. Frequently, a program's authors or maintainers can improve execution speed by using different programming techniques or modifying the program's algorithms and data structures. By providing an accounting of which resources the program is using and which sections of code are primarily responsible, performance tuning systems can direct programmers' efforts to where they are most needed.

A third application of program-understanding is *software instruction* and *orientation*. The internal workings of a program may be of special interest to students learning important algorithms, data structures, or programming techniques; this situation frequently arises when learning a new language. People assigned to maintain or improve a program written by someone else similarly need to *orient* themselves as to its general operation. In both of these cases, the people involved may be entirely unfamiliar with the program source code, and can benefit from information provided by program understanding tools before consulting source code, or without referring to it at all.

In addition to these established uses for program-understanding systems, program-understanding tools can provide language implementors with valuable assistance in the task of *language implementation tuning*. Program-understanding tools that provide information about the execution of programs also directly or indirectly provide information about the language's implementation. This information can be used to improve performance or address problems in the implementation.

1.2 Types of Software Visualization Tools

Programs that provide information about other programs can be separated into two main categories based on the kind of information they provide. *Static analysis tools* examine the program text and, in conjunction with knowledge of the language, provide information about a program that is true for all executions of that program independent of its input [1]. Compiler code optimizers, pretty printers, and syntax-directed editors frequently employ static analysis techniques. This book employs static analysis mainly to provide a spatial context, the terrain within which a user interprets dynamic behavior, perceived as animated objects.

Dynamic analysis tools provide information about a specific program execution on a specific set of input data [1]. Since dynamic analysis involves extracting