An Architecture for Multi-User First-Person Software Visualization

Clinton L. Jeffery
Department of Computer Science
University of Idaho
Moscow, Idaho
jeffery@uidaho.edu

Abstract—The city metaphor has become a popular way to visualize aspects of large software systems. Research using such visualizations has focused on showing static relationships between software components, changes over time within software repositories, or aspects of navigating such systems with cave or other virtual reality hardware. This paper explores the integration of dynamic program execution behavior within a software city visualization. In order to share the visualization across multiple users and sites, visualization is performed within a simple collaborative virtual environment. The virtual environment’s client-server architecture is extended to support the execution monitoring requirements of the software visualization function.

Index Terms—software visualization, software city, dynamic analysis

I. INTRODUCTION

The visualization of software in 3D by means of a city metaphor was inspired by science fiction classics such as TRON and Neuromancer. By the late 1990’s, advances in 3D graphics, VR, and networking capabilities enabled research that began to resemble those works of fiction. Since then over a dozen independent implementations of a software city metaphor have appeared in the research literature.

Most existing software city visualizations map static and semi-static information, such as changes to a software repository over time, onto a city’s buildings, streets, and other infrastructure. Some dynamic execution behavior has been portrayed, such as function-call activity, profile timing, and object instantiation; these will be discussed in the section on related work. However, existing software city visualizations have not attempted to portray a program’s dynamic state information by populating a software city with inhabitants in the form of anthropomorphized humans, robots, animals or machines. A diverse plethora of such inhabitants can be obtained, given a high performance means of obtaining fine-grained program execution behavior.

Likewise, existing software city visualizations tend to be single user applications. In order to maximize their benefit, such tools need to be easily shared, even when developers are not colocated. One way to meet this requirement is to adapt an existing multi-user collaborative virtual environment for software visualization purposes.

This paper’s research question is: is it possible to merge an execution monitoring architecture into a collaborative virtual environment with enough performance to portray fine-grained dynamic information, such as memory allocations and data structure accesses, in a useful way to multiple users. The question is tested by construction, by building a modified virtual environment client that executes programs and injects fine-grained dynamic information into the virtual environment as the program runs.

After a section describing related work on software city visualizations, the paper’s third section describes the new multi-user software visualization architecture called Legion, which is a hybrid of an execution monitoring architecture and a client-server collaborative virtual environment architecture. Section 4 provides details of the static- and semi-static information used to generate the city visualization. Section 5 describes the new execution monitoring client and revised network protocol that were developed in order to visualize dynamic program behavior by means of a set of animated entities that move and interact within the visualization. Section 6 concludes the paper.

II. RELATED WORK

Software city visualizations have been appearing with increasing frequency in recent years. A literature survey describes many software city research efforts [Jeff19]. This section summarizes the works that are most related to this research.

Knight and Munro used the Maverik VR toolkit [Hubb99] to develop a tool called Software World [Knig00] that is an early single-user example of the city metaphor for software visualization. Their layout maps Java classes to districts, within which methods are buildings whose height is one floor per 10 LOC. Parameters are depicted by exterior doors. Lighter and darker building colors indicate public and private. The mapping of one building onto one function seems suited to visualization of finer-grained details at the expense of scalability to larger systems. The work of Knight and Munro was later subsumed by a tool called Component City, which depicts components as buildings, a coarser granularity than that used in Software World [Char02]. Component City uses XML and XSLT to generate VRML that can be viewed in browsers with a suitable plug-in.

Lanza et al developed Code City, a visualization system in which a building represents a class, with a height proportional to the number of methods [Wett07]. Building length and
width correspond to the number of class attributes. The layout of buildings is performed using a treemap, with elevation indicating the nesting level of packages. Code City has been used to visualize larger object-oriented systems consisting of many thousands of classes. The tool is used in the study of code evolution over time in software repositories.

Panas et al developed a city visualization for C++ programs using a package called Vizz3D [Pana07]. Like Software World, their visualization depicts methods as buildings; a blue platform serves as a pad for all the functions within a class. The number of lines of code in a method is indicated by the texture of its exterior walls. The Vizz3D based city visualization depicts dependencies between functions and classes, for example drawing “water towers” that connect header files and the classes defined within them. The system depicts timing information produced by gprof, to augment the vast amount of static information visualized.

Steinbrückner and Lewerentz created a system called EvoStreets [Stei10]. They adapt the primary, secondary, and tertiary data models from cartography: models that respectively distinguish raw data, a detailed internal model, and a view-specific model that is render-friendly. EvoStreets lays out classes, depicted as cylinders, around streets, corresponding to directories or packages. The origin date of the various software components is indicated by their elevation given on a topographic map. EvoStreets has been used to study spatial orientation on VR head-mounted displays [Rude18].

Waller et al invented SynchroVis in which classes are buildings [Wall13]. Class instances are visualized as floors of the building, allowing per-instance depiction of runtime dependencies and behavior. SynchroVis depicts threads using colored arrows, allowing the visualization of concurrent behavior, communication and synchronization issues such as deadlock. Icons depict reasons why code may be suspended in a given instance, such as a method call to another object, or a wait on a semaphore.

Merino et al created a VR application for the HTC Vive using the CodeCity system to generate the 3D visualization [Meri17]. The number of lines of code in a class is indicated via brightness. Source code for any class can be pulled up in a translucent 2D heads-up-display by pointing and clicking on the building.

Vincur et al wrote a VR application for HTC Vive in which building layout is more organic because the amount of surface area contact between buildings is proportional to the amount of coupling between them [Vinc17]. For highly-coupled classes, the visualization may appear somewhat jigsaw-like. Their visualization allows the use of color-coding to highlight revision log information such as author commits, or dynamic information such as execution call traces.

Khaloo et al developed Code Park to utilize the city metaphor for navigation of source code [Khal17]. Building size is proportional to code size. The walls of buildings contain source code with IDE-like code navigation features. Users can toggle between birds-eye-view and first-person view on the ground. Movement between locations such as from variable uses to their definitions is performed by animating up to birds-eye-view and then down to the new location, improving the user’s orientation.

Ogami et al developed a city-metaphor tool specifically to deliver profile timing information in near real-time. The height of buildings is proportional to the amount of time consumed by the class in a fixed time frame such as the last L milliseconds. By watching the profiles of different classes fluctuate over time, developers can notice issues that are not portrayed in conventional profiler output.

III. THE LEGION SOFTWARE VISUALIZATION ARCHITECTURE

This section describes the Legion software architecture introduced in this paper. The Legion architecture supports multi-user collaborative software city visualizations by extending a conventional client-server architecture. Legion supports the visualization of multiple static and dynamic program properties.

An implementation of Legion was produced as a set of extensions to a collaborative virtual environment called CVE, an open source project hosted on SourceForge (cve.sf.net). CVE was originally built for CS distance education, and provides a collaborative IDE and text chat capability within one of two 3D multi-user virtual environments, modeled after the physical CS departments at two public universities. A screenshot of CVE is shown in Figure 1.

The Legion architecture consists of the following modifications to CVE’s client/server architecture in order to adapt it to the purposes of software visualization.

• The CVE client IDE code was modified to procedurally generate 3D models of the programs that users write, using a software city metaphor. Network protocol messages were added to CVE to transmit and store these 3D models as JSON files.

• Monitored program execution is provided by the server launching an external process called the Legion client.
The Legion client is a modified version of the CVE non-player character (NPC) client, which was originally developed for specifying the behavior of computer-controlled entities such as quest-givers and “monsters”. The Legion client monitors the execution of a target program and translates patterns of program execution events into virtual environment messages that instantiate and direct the behavior of multiple entities.

- The CVE virtual environment network protocol was extended to support execution monitoring. The primary extension is to enable clients to manage the behavior of multiple avatars associated with entities in the program execution, in addition to the primary avatar that denotes a human user. The many avatars that are utilized by execution monitor clients are the reason for the name Legion.

The Legion architecture is summarized in Figure 2. Solid arrows are TCP connections between processes. The dashed arrow labeled `EvGet()` is a coroutine request for the next execution event from the target program, while the dot-dashed arrow labeled `event report` is a coroutine initiated from the runtime system when an event of interest occurs. The target program generally is not aware of being under observation. The Legion client and target program are depicted as smaller circles than the others because they do not have to implement the 3D graphics rendering of the regular CVE client.

IV. STATIC AND SEMI-STATIC PROGRAM PROPERTIES

The static aspects of programs are obtained by reading source code in the form of a directory tree or repository checkout from subversion or git. The locations and sizes of functions, classes and methods are stored in a tree structure. In this and the following section, a running example is used, which consists of the visualization of a large open-source software project, the Unicon programming language from unicon.org.

The static information is used to generate a 2D street layout based on source code directory and package structure, as shown in Figure 3. Street lengths are calculated from the sizes of all source files within the corresponding directory and its subdirectories. The street layout has the desirable property that a path between any two buildings is trivially computed from the tree structure reflected by the streets.

The Unicon language distribution includes code written in multiple languages. The layout places the buildings that represent code on two different levels. The surface of the city contains the building layout for code written in Unicon, such as its compiler and class libraries. The black lines in Figure 3 are surface streets containing code written in the Unicon language and part of its public distribution.

A subterranean layout is performed for the lower-level code. In the Unicon example this consists primarily of the virtual machine, runtime system, and external libraries. This layout is performed from the surface plane extending downwards. The brown (or gray) lines in Figure 3 are the subterranean, C language level components such as its virtual machine and runtime system. Since they are below ground, they do not overlap with surface streets at the same location. Blue pixels on the 2D layout mark the locations of buildings denoting classes and functions, whose layout occurs on alternating sides, up and down the streets.

The subterranean level is expected to be useful more to language (or library) maintainers than to applications developers. However, some developers may elect to visualize their entire implementation including code, rather than visualizing just their application source code.

At each level, the widths of streets, ranging from freeway to arterial avenue on down to road, are proportional to the natural log of the amount of code within them. Each street contains
0 or more files, each of which contains 0 or more buildings denoting classes or functions.

After the (multi-layered) 2D city layout is performed, the city generator traverses the tree that was constructed during 2D layout to generate a 3D model of the city, which consists of a JSON file containing 3D coordinates and texture specifications corresponding to the classes and standalone functions in the system. For the Unicon language distribution containing around 600,000 LOC, the generated file uses 5MB to model 9374 classes and functions.

In addition to the static information obtained from the source code, log information from the subversion or git repository is read, if available. This semi-static information accumulates slowly over time. As an example of how this information is used, the origin date when files were introduced into the repository is used to select the exterior texture of the building that represents that file in the visualization. At present, file ages are partitioned into five categories from which classes and functions are assigned one of five exterior wall textures ranging from very old brick buildings to gleaming modern glass and steel structures. Figure 4 shows an example in which classes of several different ages are visible. Double-clicking on a building opens its source code in the IDE.

V. INTEGRATING DYNAMIC EXECUTION BEHAVIOR

Legion obtains execution information from a program as it runs, using a lightweight architecture for program execution monitoring called Alamo [Jeff98]. Similar functionality could be obtained via other debugging API’s such as JVMTI [JVMTI]. Program execution is divided into a sequence of steps; at each step the monitor (in this case the Legion client) specifies what kinds of behavior are of interest as it requests the next step forward in execution, called an event. When an event of interest occurs, control returns to the monitor via an event report that indicates the type of event an one associated program value. The monitor can examine additional program state as needed. The Alamo architecture has been implemented for Icon, Unicon, C and Python to varying extents. For both ease of access and performance reasons, the monitor and program being observed execute within in the same address space. Execution switches between monitor and target program synchronously at each event.

A. Prioritizing Legion NPCs

Legion’s scalability is limited by

- how many entities can be monitored feasibly by the Legion client in real time while the program runs
- how many entities the server can handle, and
- how many entities the human user clients can render and update graphically with acceptable frame rates

After appropriate package aggregation and non-blocking I/O techniques were applied to the network code, the client graphics became the main bottleneck. In 3D graphic applications, there are limits to the number of moving entities that can be rendered simultaneously. In classic MMO games, for example, the limit might mean that frame rates drop off, or rendering starts to fail, when around 200 users (and their avatars) occupy the same scene. The limit might be increased by using higher end hardware, or by selecting lower resolutions or graphics settings, but Legion must prioritize NPC’s and despawn them when other more important behavior emerges.

The Legion priority queue has the following characteristics. All are based on an adjustable time interval that defaults to the preceding three seconds.

- Queue length is adjustable, based on reported frame rates of affected clients. Length increases whenever no client is reporting a frame rate falling below 24fps. It decreases whenever one or more clients are reporting frame rates below 18fps in a given interval.
- Entity size is a log-based function of the amount of information associated or contained with an entity. Size influences queue priority, as larger objects are deemed more important.
- Entity activity is the number of execution monitoring events that have updated the NPC state. An entity’s activity is reflected by both the speed of its animation as well as the lighting effects with which it is rendered.
- Size, activity, and visibility all contribute to an entity’s priority for retention on the queue. Objects are despawned when they fall off the end of the priority queue.
B. Legion NPCs Example

Legion spawns computer-controlled NPCs in response to program execution events. This section provides a short example, told from the point of view of the events delivered to the Legion client, and the resulting virtual environment commands transmitted to a client that is viewing the program from within its main() building. A toy 202 LOC German-English dictionary program utilizes a singleton class named DeenDictionary that contains a number of German-English dictionary entry objects, instances of a class DeenEntry. The actual dictionary data is read from ftp.tu-chemnitz.de/pub/Local/urz/ding/de-en/de-en.txt.gz, a file with 197771 lines and approximately 400,000 dictionary entries/definitions. Pseudocode for the main procedure of the example looks like

```
dd := DeenDictionary() # read dictionary
for every s := word_from_command_line do
  if answer := dd.lookup(s) then
    write(s, " ", answer.definition)
  else write(s, " not in dictionary")
```

The initial creation of the dictionary results in the creation of a single, large (dictionary) object containing many small, dictionary entry objects. The event necessary to recognize an instance being created is an E_Fcall (function call) whose associated value says it is a class constructor. For the dictionary, the event is reported by the Legion client to the CVE server. Because it comes into existence within main(), and hence is visible to a user also standing in main(), a corresponding CVE entity is created and reported by the server to the client, which renders a large cuboid.

A great many DeenEntry dictionary entry objects came into existence during the dictionary constructor code, from a procedure named get_entries(), which returns lists of entries that are placed into the DeenDictionary by its read() method. The Legion client monitors this behavior, but doesn’t report it to the CVE server because it does not reach a potential-visibility threshold, neither due to size, nor due to non-locality of reference.

The Legion client monitors other execution behavior, including method calls and field accesses, which each have their own event types. In this example, the visibility threshold might be reached for some dictionary entries despite their small size, if the command line arguments giving words to translate is long and contains enough duplicates. Repeatedly-accessed dictionary entries that are reported by the Legion client are rendered as small, droid-like entities marching from the dictionary class to the bulding denoting the main() procedure, where they are being used.

VI. CONCLUSIONS

This paper presents Legion, a gamelike multi-user software visualization architecture that was created by modifying a virtual environment. In Legion, static and semi-static program information is first used to generate a software city. When a monitored program execution is requested in the IDE, a separate virtual environment client is launched to perform that execution, resulting in a bounded set of computer-controlled entities introduced into the city, corresponding to program execution artifacts such as activation records and object instances.

REFERENCES


