ThreadCity: Combined Visualization of Structure and Activity for the Exploration of Multi-threaded Software Systems

Sebastian Hahn, Matthias Trapp, Nikolai Wuttke, Jürgen Döllner
Hasso Plattner Institute, University of Potsdam, Germany
firstname.lastname@hpi.de

***Abstract*—This paper presents a novel visualization technique for the interactive exploration of multi-threaded software systems. It combines the visualization of static system structure based on the EvoStreets approach with an additional traffic metaphor to communicate the runtime characteristics of multiple threads simultaneously. To improve visual scalability with respect to the visualization of complex software systems, we further present an effective level-of-detail visualization based on hierarchical aggregation of system components by taking viewing parameters into account. We demonstrate our technique by means of a prototypical implementation and compare our result with existing visualization techniques.**

***Keywords*-Visual Software Analytics, Trace-Visualization, Multi-threaded Software Systems**

I. Introduction

Program understanding is a crucial but tedious and time-consuming task within the software development and maintenance process that becomes even more complex with the use of multi-threading. Visual software analytics tools can help to gain insight into the non-trivial processes of such systems. However, most of these tools create depictions that either focus on the runtime behavior or the static structure, not taking into account that for various comprehension tasks the combination of both information bases are required.

Although, previous research in combining static structure of a system with its runtime information for sequential working applications was conducted, there are no suitable approaches for the visualization of systems with concurrent runtime behavior. In this work, we present a visualization technique and prototypical tool (Fig. 1) that allows for the analysis of multi-threaded systems using a combination of both, the organizational structure as well as concurrent runtime-behavior information of a system.

The outline of this paper is as follows. Section II presents an outline of the problem statement and related work. In Section III the visualization approach for combining structural and dynamic information is described. Details about a novel hierarchical aggregation technique are presented in Section IV. Section V gives additional information about the usability of the *ThreadCity* tool, its usage, and the results of a preliminary study. Finally, Section VI concludes this paper and present possible future research directs.

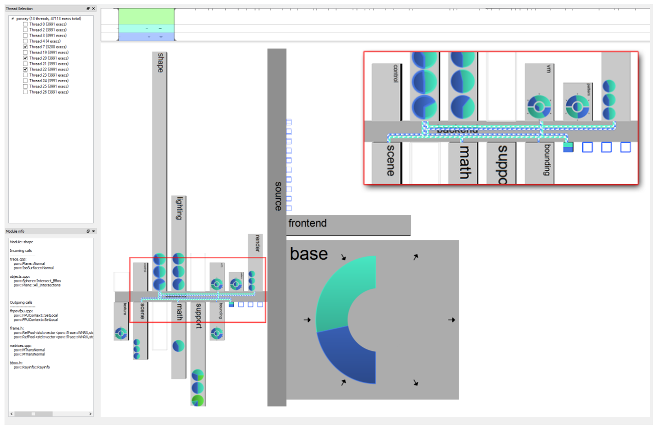


Figure 1. Exemplary screen shot and inset of the presented visualization technique and tool *ThreadCity*.

II. Problem Statement

This section covers fundamental related and previous work with respect to software maintenance, data acquisition, as well as as suitable visualization approaches.

A. Software Maintenance

Software maintenance is the most important stage during the lifecycle of almost all complex software systems [1], [2]. A deep understanding of both, a program's *structure* and *behavior*, is a critical requirement for a successful software maintenance process. According to Fowler, software developers spend about 40% of their time working on program understanding tasks, due to the high complexity of large software systems [3]. Additionally, the use of parallel computation increases the complexity of this tasks [4], [5]. Visual software analytic tools are a common approach to gain insights into software systems. However, most approaches focus on either the depiction of the software static structure or the visualization of its runtime behavior. Since the use of multiple tools would yield a high amount of context switches [6], we argue that a *combined visualization* of both allows for a better understanding of a software system. For it, we propose a novel approach that combines the visualization of structural and dynamic information in a metaphorical way especially suitable for the visualization of multi-threaded behavior.

Since the number of tasks that belong to the field of software maintenance is exhaustive, we focus on two major tasks: *program understanding* and *performance analysis*.

Program Understanding: Program understanding describes the process of gaining knowledge about the software system to "*determine the effort (in terms of resources expended) for corrective, preventive, adaptive, and perfective maintenance*" [7]. For common maintenance tasks, such as (1) enhancing a component's functionality, (2) adapt the existing behavior to new requirements, or (3) generalizing the existing functionality and transfer it into a new component or library, the insight gained from the static structure of a software systems (e.g., from source code analysis) is not sufficient. For it, a software developer also needs knowledge about the runtime behavior of components, e.g., to map execution periods to components.

Performance Analysis: Performance analysis is usually accessed as a non-functional software requirement. In the software maintenance process, the improvement of implementations efficiency represent an important factor of success. For it, profiling tools, working on a high granularity level, are used, creating function list with additional runtime information, such as completion time, caller, and internal function calls.

In addition to the aforementioned tasks, the use of multi-threaded computations within implementations involves particular challenges. A software developer needs to know whether (1) a software system component uses task or data parallel computation, (2) decide which component handles a specific thread, and (3) which component is active during an execution period. Begel et al. underline the importance of this question in their work, presenting that one of the most important questions for software developers is: "What are the common patterns of execution in my application?" [8].

B. Dataset and Acquisition

The input data for the present visualization approach requires for (1) gathering data using *static source code analysis* as well as for (2) a *runtime analysis* of a given software system. For the static source code analysis, we first extract the hierarchical organizational structure of a object-oriented system's artifacts (e.g., package structure in java systems or nested namespaces in C++ systems). Depending on the analysis task, a granularity level is defined for the hierarchy leaf nodes (e.g., classes, functions, or statements). A common approach for visualization of large software systems is mapping classes as leaf nodes [9], [10]. Since we aim at presenting an approach that is independent from a specific object-oriented programming language, we apply the term *"package"* to all non-leaf nodes of a hierarchy. Moreover, all leaf nodes are classes in the system's hierarchy with aggregated information about their functions.

These functions, aggregated on class level, represents the basis for the dynamic information. An instrumentation of the running system is required to extract the dynamic call information:

- Caller: Function that calls another function.
- Callee: Function that is called by caller.
- Timestamp: Point of time a call starts.
- Thread-ID: Unique identifier of a certain thread.

To summarize, the information gathered form static source code analysis and dynamic tracing is combined using the structural organization of a system in a hierarchical way using classes as leaf-nodes and dynamic runtime information (calls), which is acquired during a trace between these classes by aggregating functions (callers and callees).

C. Limitations of EvoStreet Approach

There are various approaches for the visualization of the static structure of a software system. Space-filling approaches, such as treemaps, are unsuitable for the depiction of additional relationship information, since on-top rendered glyphs would lead to occlusion and visual clutter. To counterbalance the occlusion problem, we rely on a non-space-filling approach: *EvoStreet* represented by Steinbrückner et al. [11]. This metaphor-based approach creates a *"software city"* for a given hierarchy, with packages as *non-leaf nodes* and classes as *leaf nodes*. Caserta et al. present an approach to depict relationships between leaf nodes by drawing hierarchical edge bundles on top of the city. This, however, yield the same occlusion problems as in treemaps [12]. Khan et al. use this technique to depict static software related structure and relationship information [13]. SynchroVis, another visualization tool that also depicts multi-threaded systems, presented by Waller et al. is faced with the same problems [14]. With respect to this, we argue, that the fact, that the structural elements have an explicit space (not like with the nesting in treemaps) should be used, to create explicit visual artifacts for relationships on these structural elements. Also, by using a traffic metaphor we do not break the metaphor of a software city but add an easy to understand extension, that is also able to explicitly separate threads by using different traffic lanes.

III. COMBINED VISUALIZATION

Since software systems have no inherent or natural shape, the use of metaphor-based visualization techniques allow for an intuitive understanding of a system's structure, e.g., software cities [11] or software maps [15].

Conceptual Overview: As aforementioned, we show the structure of a software system using an *orthogonal arrangement of streets* for the structural nodes of a hierarchy and *buildings* attached to streets for depiction of the leaf nodes. Additionally, the hierarchy depth of a node is

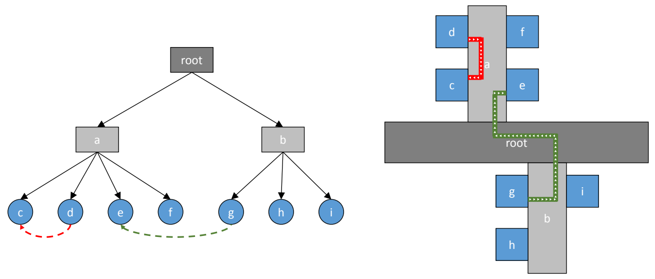


Figure 2. Metaphor-based visualization concept: a hierarchical city layout is extended by a depiction of call dependencies of different threads (different colors used) with a traffic metaphor.

indicated by its width, i.e., wider streets are communicating elements on a higher level in the hierarchy.

While existing approaches focusing on the static system structures and their associated attributes (e.g., using size, height, and color of buildings) only, our approach extends the notion of software cities by visualizing relations between leaf nodes using an explicit *”traffic”* metaphor (in contrast to the implicit depiction by nesting in a treemap-based approach). This explicit separation of structure and leaf nodes allows for the possibility of combining static structural information with dynamic call data, showing *traffic routes* between the buildings. This creates a metaphor for function calls between classes.

Visualization of Function Calls: The routes are implemented using shortest paths between two class representations with respect to the connecting streets and rendered by line segments. The traffic itself is illustrated as right-hand or left-hand traffic. For it, the traffic directions and starting points are set on either the right or left side of a street. As a result, the direction of a call is implicitly visualized by the lane its route is using. In addition to that, a separation of calls from different threads is implemented by using a driving lane separation. That means, the two halves of a street (or building) are subdivided into a number of driving lanes with respect to the number of selected threads. Moreover, calls from different threads are displayed by different colors. Furthermore, we implemented an animated texture with arrow symbols to underline the direction and to achieve a more clear perception of callers and callee's. Finally, by combining the city metaphor with routes between buildings, we create an metaphor-based approach that can be understand easily (Fig. 2).

IV. Level-of-Detail Visualization

In order to achieve visual scalability and enable meaningful insights for *complex* software systems, (e.g., reduction of visual clutter) the presented approach supports level-of-detail visualization as a major functionality. The level-of-detail visualization is based on an aggregation technique for software system information that is controlled based on user navigation (perspective) and selection (user interaction). While Shneiderman's information seeking mantra *”overview first, zoom and filter, then details on demand“* [16] is not sufficient for our use case, we focus on visual analytics mantra: *”Analyze, first, show the important, zoom, filter and analyze further, details on demand“* [17], by providing a multi-scale visualization with respect to three dimensions:

(D1) Number of threads
(D2) Number of structural elements (hierarchy depth)
(D3) Number of function calls (traffic)

While D1 is not main focus of this work (non-massive multi-threading), the concept of hierarchical aggregation is sufficient for D2 and D3.

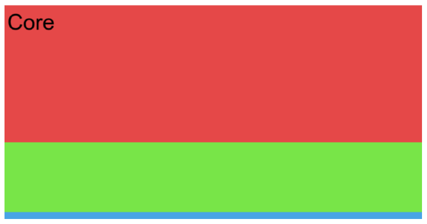
Aggregation Principles: Therefore, the depiction is simplified by aggregating a number of visualization elements into a single visual element according to the hierarchical aggregation guidelines of Elmquist and Fekete [18], that are summarized as follows:

(R1) *Entity budget*: a maximum number of elements is constrained by a ”budget“
(R2) *Visual summary*: aggregates should display condensed information
(R3) *Visual simplicity*: aggregates should be kept simple
(R4) *Discriminability*: should be possible to easily distinguish aggregates from non-aggregates
(R5) *Fidelity*: aggregates should display the condensed information as accurate as possible
(R6) *Interpretability*: just aggregate as much as you need for a correct interpretation, show what you can show

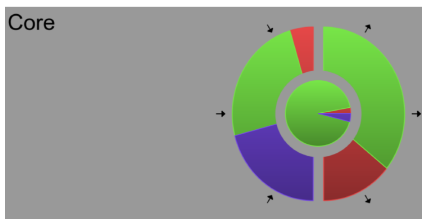
The *ThreadCity* visualization technique basically comprises streets, buildings, and associated traffic.

Aggregation Approach: A street can be aggregated by its bounding box comprising its child elements (serves R3 and R4) yielding a simple box that easy to distinguish. The root node is not aggregated, thus it will not appear as a single street. This approach allows for collapsing and expanding of packages, a standard interaction technique for hierarchies [18]. Requirement R6 is mostly important for visualization techniques for which overlapping causes a problem. However, software cities are free of overlapping. The first requirement R1 is currently not implemented but could be easily implemented in future work. The remaining requirements R2 and R5 are considered by the implementation of specific diagrams that are displayed on aggregates and described in the following.

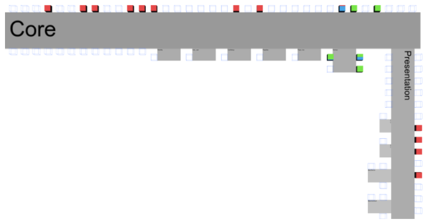
Detail Levels and Aggregation Diagrams: Figure 3 presents an overview of different detail levels, automatically computed by the visualization technique. This LoD concept serves D3. Here, LoD-2 represents non-aggregated view of a street (Fig. 3.c). Aggregating such visual representation of streets requires for an alternative for displaying traffic and information mapped on buildings. To counterbalance this problem, additional diagrams displayed on aggregates



(a) LoD-0: Complete aggregation of thread activities.



(b) LoD-1: Aggregation of thread activities based on incoming calls, outgoing calls and internal calls.



(c) LoD-2: No aggregation of thread activities.

Figure 3. The three levels of detail for a given node. Visualization of thread activities are aggregated based on a users perspective and interaction parameters (e.g., hovering and explicit selection).

are introduced (Fig. 3.b). To generate these diagrams, the following approach is used: (1) find all calls of set of children in aggregate and cluster these into three groups of calls (incoming calls, outgoing calls, and internal calls); and (2) show rate of each thread in each group. We apply *pie charts* for the depiction of a combined chart (Fig. 4) with respect to size of an aggregate in LoD-1. If the remaining size of an aggregate is too small, a *bar chart* depicting only the ratio of each thread without the thread grouping is shown in LoD-0 (Fig. 3.a). We use alpha-blending for the transitions between detail levels that are selected according to the projected item size and the distance to the camera.

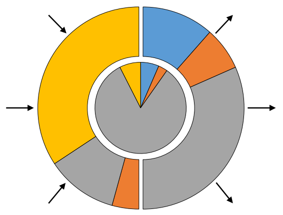


Figure 4. The aggregated diagram shows the amount of incoming (left), outgoing (right), and internal calls (center) per thread with respect to a given aggregate.

V. Application Example

This section describes how a user can interact with the presented system based on an application examples, and discusses the results by means of an expert user evaluation.

A. Overview of User Interface

Fig. 5 shows an overview of the user interface provided by *ThreadCity*. This section briefly describes its components as well as supported navigation and interaction techniques.

User Interface Components and Operation.: The basis for data exploration is temporal filtering of the thread data. Therefore, a thread selection widget and time-line overview are included in the user interface (Fig. 5.A and B). After (multiple) thread selection is performed, the overview widget shows the temporal ordering of all events: a single thread is depicted as icicle plot while remaining threads are shown using a 1D scatter plot. Subsequently, the temporal focus can be changed by user selection. Each focus thread is presented by a unique color. To perform further temporal filtering, a point in time or a time range can be set in the overview. After filtering is completed, the main view then contains the combined visualization of structure and dynamic information with the selected time range. Elements that are not active as a caller or callee within the given time range are shown using a wire-frame mode. All active elements have colored top faces based on the thread they are used in. If this comprises more than one thread, it's a bar diagram is shown (cf. Sec. IV). Additionally, the detail window shows further information about selected elements (Fig. 5.C).

Navigation Techniques.: The thread city is visualized in 2D only by rendering a top view (Fig. 5.D). Therefore, the navigation techniques rely on standard pan and zoom metaphors [19]. This avoids getting-lost situations often occurred when relying on 3D virtual environments [20]. For convenience reasons, zooming is performed with respect to the current cursor position or based on a rectangular on screen selection. As start for interaction, an overview of the complete data shown using the lowest LoD.

Interaction and Selection.: The main purpose of the detail widget (Fig. 5.D) is to display additional information such as function calls on a per-class level. Here, hovering or explicit selection is used to filter the additional information to reduce visual clutter in the display. For example, hovering over a building object just shows the connected traffic is shown while non-connected traffic is hidden. Further, *pinning* a building (using mouse click) marks it as selected and assigns the current focus, i.e., if the mouse cursor leaves the building, the connected traffic is still visualized. A special *zoom-out* mode allows for fast changes of focused classes if pinning is active on a particular class (right-mouse on pinned element) by automatically setting the zoom value in order to show all connected classes.

B. Preliminary System Evaluation

For a qualitative preliminary system evaluation we asked a small group of users to perform three tasks with two different datasets. The users were not involved in the implementation

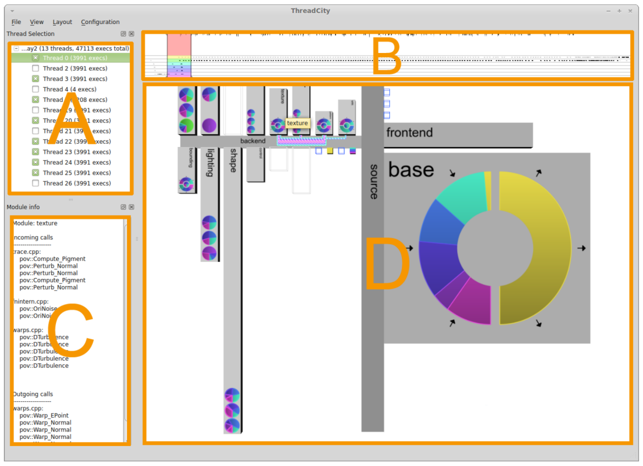


Figure 5. Exemplary screen shots of the user interface provided by the ThreadCity application comprising: (A) thread selection widget, (B) timeline overview, (C) details and meta information, (D) main view.

of neither the visualization tool nor the two systems that were analyzed. To show the usefulness of our system we compared our prototypical implementation with *ViewFusion*, another tool that allows the combination of structure and runtime visualization [6]. The datasets are open-source projects making use of parallel processing:

Chromium: Chromium is an open-source web-browser with about 2.7 million lines-of-code. The tracing was done to investigate the behavior while opening a website by selecting a bookmark. This trace included tasks such as network requests, data parsing and construction and rendering.

Povray: POV-Ray2 is another open-source tool, enabling offline rendering using ray-tracing. The investigated version is 3.7, containing about 180k lines-of-code. We extracted the dynamic data during the rendering of a 3D-scene, which uses data-parallel computation.

The three tasks that each user had to perform were:

1) Categorization of multi-threaded behavior: Can a user determine whether a program's behavior is either data parallel or task parallel.
2) Program behavior overview: The primary goal is to get an overview about the main tasks of a certain thread based on its involved components.
3) Activity or performance analysis: Users should analyze which component in each thread is the most time consuming one and therefore a candidate for a deeper performance analysis.

The qualitative study showed that tasks 1 and 2 could be completed well by all participants. Compared to *ViewFusion*, which does not allow for analyzing multiple thread in a single visualization but rather creates one instance of the visualization for each selected thread, the users stated that the tasks were easier to complete and the aggregation, implemented in *ThreadCity* would help to get a faster overview over the systems behavior. The participants stated that the most benefiting properties of our system were the aggregation and the used LoD concept in combination with the "easy-to-understand" city layout. Further, the users stated that the hovering for additional information in the deeper level was useful. Nevertheless, most users had problems completing task 3 with both systems, which indicates that the used interactions and visualization techniques have a need for improvement to complete such tasks. However, the preliminary study showed that the described concept works good for program understanding tasks of multi-threaded systems.

VI. Conclusions and Future Work

The presented work facilitates analysing and program understanding of tasks within multi-threaded software systems. Especially, the combined visualization of static system structure and threading information, aggregated and presented using additionally diagrams, enables insights in activities of software components during runtime. With respect to existing approaches, the presented concept and implementation shows advantageous characteristics that are experimentally verified by a preliminary user study. However, the presented approach is limited with respect to missing chronological ordering when limited to static non-animated visualizations. Further, the visualization and analysis of particular function calls are limited. Similar, a quantification of calls represented by routes (e.g., a number of calls in a single thread) is currently not supported and requires suitable aggregation techniques.

Despite limitations described previously, there are various aspects for future research. For example, introducing 3D visualization by mapping components attributes to height comprise a possible extension. Further, functionality for aggregation and highlighting, as well as mapping of additional visual variables (e.g., width or shape) to routes can enhance the visualization technique. Finally, one can think of extincting the presented metaphor with respect to underground routes or similar.

Acknowledgments

The authors would like to thank the anonymous reviewers for their valuable comments. This work was funded by the Research School on "Service-Oriented Systems Engineering" of the Hasso-Plattner-Institute and the Federal Ministry of Education and Research (BMBF), Germany, within the InnoProfile Transfer research group "4DnD-Vis" (www.4dndvis.de).

REFERENCES

[1] T. A. Corbi, "Program understanding: Challenge for the 1990s," *IBM Systems Journal*, vol. 28, no. 2, pp. 294–306, 1989.

[2] D. L. Parnas, "Software aging," in *ICSE*. IEEE Computer Society Press, 1994, pp. 279–287.

[3] M. Fowler, "Refactoring: Improving the design of existing code," 1997.

[4] J. Trümper, J. Bohnet, and J. Döllner, "Understanding complex multithreaded software systems by using trace visualization," in *SOFTVIS*. ACM, 2010, pp. 133–142.

[5] B. Karran, J. Trümper, and J. Döllner, "Synctrace: Visual thread-interplay analysis," in *Software Visualization (VISSOFT), 2013 First IEEE Working Conference on*. IEEE, 2013, pp. 1–10.

[6] J. Trümper, A. Telea, and J. Döllner, "Viewfusion: Correlating structure and activity views for execution traces," in *TPCG*. European Association for Computer Graphics, 2012, pp. 45–52.

[7] "ISO/IEC 14764:2006 Software engineering – software life cycle processes – maintenance," Geneva, CH, 2006.

[8] A. Begel and T. Zimmermann, "Analyze this! 145 questions for data scientists in software engineering," in *ICSE*. New York, NY, USA: ACM, 2014, pp. 12–23.

[9] R. Wettel and M. Lanza, "Visualizing software systems as cities," in *VISSOFT*. IEEE, 2007, pp. 92–99.

[10] M. Balzer, O. Deussen, and C. Lewerentz, "Voronoi treemaps for the visualization of software metrics," in *Symposium on Software visualization*. ACM, 2005, pp. 165–172.

[11] F. Steinbrückner and C. Lewerentz, "Representing development history in software cities," in *SOFTVIS*. ACM, 2010, pp. 193–202.

[12] P. Caserta, O. Zendra, and D. Bodénes, "3d hierarchical edge bundles to visualize relations in a software city metaphor," in *VISSOFT*. IEEE, 2011, pp. 1–8.

[13] T. Khan, S. R. Humayoun, K. Amrhein, H. Barthel, A. Ebert, and P. Liggesmeyer, "ecity+: A tool to analyze software architectural relations through interactive visual support," in *Proceedings of the 2014 European Conference on Software Architecture Workshops*. ACM, 2014, p. 36.

[14] J. Waller, C. Wulf, F. Fittkau, P. Dohring, and W. Hasselbring, "Synchrovis: 3d visualization of monitoring traces in the city metaphor for analyzing concurrency," in *VISSOFT*. IEEE, 2013, pp. 1–4.

[15] J. Bohnet and J. Döllner, "Monitoring code quality and development activity by software maps," in *Workshop on Managing Technical Debt*. ACM, 2011, pp. 9–16.

[16] B. Shneiderman, "The eyes have it: A task by data type taxonomy for information visualizations," in *Symposium on Visual Languages*. Washington, DC, USA: IEEE Computer Society, 1996, pp. 336–343.

[17] D. A. Keim, F. Mansmann, J. Schneidewind, and H. Ziegler, "Challenges in visual data analysis," in *IV*. IEEE, 2006, pp. 9–16.

[18] N. Elmqvist and J.-D. Fekete, "Hierarchical aggregation for information visualization: Overview, techniques, and design guidelines," *TVCG*, vol. 16, no. 3, pp. 439–454, 2010.

[19] A. Cockburn, A. Karlson, and B. B. Bederson, "A review of overview+detail, zooming, and focus+context interfaces," *ACM Comput. Surv.*, vol. 41, no. 1, pp. 2:1–2:31, Jan. 2009.

[20] H. Buchholz, J. Bohnet, and J. Döllner, "Smart and physically-based navigation in 3d geovirtual environments," in *IV*, 2005, pp. 629–635.