High-level representation of 3D models of buildings

Sebastien Ouellet (sebouel@gmail.com)

Sterling Somers (sterling@sterlingsomers.com)

Jim Davies (jim@jimdavies.org)

Institute of Cognitive Science, Carleton University
1125 Colonel By Drive, Ottawa, Ontario, K1S 5B6 Canada

Abstract

The goal of the current research was to design and implement a system able to extract high-level representations of 3D building models for the purpose of developing cognitive models. We present an intermediate representation scheme that supports modeling at different levels of detail. A cognitive model could easily use our representation scheme to perceive and navigate a 3D building. The overall goal of the project was to develop an implementation-independent representation scheme, sufficient to support high-level symbolic cognitive models. Our node-based representation supports high-level visual perception (which can be used in a spatial representation internal to the agent) or a complete semantically labelled topological map. Our representation scheme identifies structural features of the 3D building environments: "floor", "ceiling", "wall", "door", and "corner", which are useful for agents programmed in symbolic cognitive architectures and navigating simulated environments. Our evaluation shows that this representation is sufficient for agents to do simple navigation in virtual environments.

Keywords: 3D models; Spatial features detection; Tactical protocols; Spatial reasoning; Agents

Introduction

Symbolic cognitive architectures such as ACT-R (Anderson & Lebiere, 1998) are used to create models of computerized agents: psychologically realistic computer programs whose internal processes and external behaviours (usually within a virtual environment) are constrained by the architecture within which they are created. Often, to validate the models, and the theories they represent, the performance of the models are compared to the performance of human experimental participants doing the same task. In ACT-R, for example, models are often compared with human task-performance measures, reaction time data, and learning rates. While many models are produced for the purpose of presenting and validating cognitive theories, there can often be a secondary use: using the architecture to create realistic agents for use in simulation training tools.

A good example of this secondary use of cognitive architectures came out of the Office of Naval Researchs Virtual Training and Environments (VIRTE) program. Models of Military Operations in Urban Terrain (MOUT) and Close Quarters Combat (CQC) were developed for different cognitive architectures (Wray, Laird, Nuxoll, Stokes, & Kerfoot, 2005; Best & Lebiere, 2003). Soar MOUTbots (Wray et al., 2005) are an extension of Soar Quakebots (Laird, 2001), and are developed for military training simulations. MOUTbots navigate their environments with the use of topological maps built up over time with the use of node-based map markers. The ACT-R MOUT model (Best & Lebiere, 2003) is based on an extended version of ACT-R 5.0 for MOUT operations. In order to navigate, ACT-R MOUT agents use both allocentric and egocentric spatial representations to form a cognitive map. The cognitive map uses a node-like representation scheme to encode the relationship between map areas.

An important commonality between the ACT-R MOUT model and the Soar MOUTbots is the simulation environment. Both the ACT-R MOUT model and the Soar MOUTbots are written to interface with *Unreal Tournament*, a combat simulation game. Much of the aim for the ACT-R MOUT model, Soar MOUTbots, and the related cognitive model SNAP (which also uses *Unreal Tournament* as a simulation environment) (Ting & Zhou, 2009) is to improve the performance of bots in the game. The advantage of using a simulation environment like *Unreal Tournament* is that it has a rich 3D environment, a sophisticated physics engine, and tools which allow virtual agents to interact with the environment and exploit the features of the physics engine (e.g., the firing of a weapon). The advantages gained from using a pre-built engine, especially in using the built-in environment tools, however, limit the degree to which the interface software can be re-used. Different modeling needs may require different features from a simulation environment, especially as environments get more complex in the future.

One of the main cognitive modeling hurdles alleviated by using simulation environments like *Unreal Tournament* is visual perception. Artificial visual perception is a notoriously difficult problem. Importantly, it is not necessary at a certain level of abstraction to model all aspects of the cognitive apparatus. In ACT-R 6.0, for example, visual perception is largely circumvented with the use of visual icons (Fleetwood & Byrne, 2003). Visual icons encode location and semantics of information usually presented on a computer screen. While the ACT-R vision system does not perceive the environment *per se*, it is able to make reaction-time predictions based on the properties of the icons (e.g., where they are located). What is important in cognitive models that use vision systems like these is that they are sufficient for the task being modeled. The visual icon and similar vision modules in ACT-R have been shown to be sufficient for many tasks. Because of the complexity of the simulation environment in

Ouellet, S., Somers, S., & Davies, J. (2013). High-level representation of 3D models of buildings. In R. West & T. Stewart (Eds.), Proceedings of the 12th International Conference on Cognitive Modeling, Ottawa: Carleton University (unnumbered six page online proceedings)

the MOUT/CQC models discussed above, in order to successfully model agents that operate in those environments, a sufficiently complex vision system needs to be developed.

Both the Soar MOUTbots and the ACT-R MOUT model have unique solutions to providing sufficient perceptual information for their agents. MOUTbots, for example, make use of an annotated map, which includes nodes to represent boundaries of rooms, and nodes which link rooms such as doors and windows. The MOUTbots can then use this intermediate representation to develop a topological map which can be used to perform actions within the 3D space. The ACT-R MOUT model uses an automated mapping agent to pre-map the location of walls, corners, rooms, and room openings in the environment using the environment tools in *Unreal Tournament*. In both of these cases, virtual agents use these intermediate representations to perceive their environments. The successful implementation of these models suggest that their perception solutions were sufficient for the task. As discussed in (Best & Lebiere, 2003), however, these solutions are both implementation-dependant in that they rely on the specific tools or pre-processed maps in *Unreal Tournament*.

The problem that this paper deals with is that of detecting structural elements of virtual architecture in 3D environments more generally. Specifically, the detection of floors, ceilings, walls, doors, and corners. 3D environments are commonly presented in scene graphs, which contain geometric representations, which are rendered by the video card to produce graphics. While different simulation engines can use different 3D environments, at a low-level, the geometric representations are often composed of collection of triangles. The aim of our work is to process 3D scene graphs to provide an intermediate representation that can be easily used as perception by virtual agents implemented as cognitive models (hereafter "agents.")

The intermediate representation acts as the agents' direct environment and should be rich enough to allow the agents to act within it. Our intermediate representation is analogous to both the Soar MOUTbot and ACT-R MOUT model solutions in that it uses nodes to represent the environment. These nodes carry semantic information about the structure of the environment, indicating features: floors, ceilings, walls, doors, and corners. Our hypothesis is that the intermediate-level spatial representation created by our detectors is sufficient for agents to do simple navigation in a virtual environment. We tested this with two symbolic-level agent models, which use the representation to navigate to doorways. Although these agents are not implemented in a cognitive architecture, our specific aim is to test whether the representation is rich enough to support this high-level navigation behaviour.

Theory

Our software employs a number of detectors in order to convert the geometric vector representation in the 3D files to the high level representations in our representation scheme. This section provides a high-level overview of the software, describing the detectors we use. Since these detectors are used to create our intermediate representation which, in turn, is meant to be sufficient for high level models, it is important that we detect an appropriate set of structural elements. As already alluded to the structural elements detected by our software are: floors, ceilings, walls, doors, and corners. We first describe 3D scene graphs, followed by our representation scheme. We then describe how the detectors work to convert the 3D scene graphs into our representation scheme.

3D Scene Graph Files

3D scene graph files are hierarchically organized data structures used for displaying 3D graphics. The hierarchical structure of scene graph files increases efficiency of the display process (Wang & Qian, 2010). Leaf nodes are fairly similar across different 3D formats in that they contain graphical matrices which represent the visible surfaces as comprised by the vectors in the matrix.

Importantly for the present work, the organization of the hierarchy is not standard across all 3D file formats. In fact, it is possible, within a single file format, for there to be differences in how low-level nodes are grouped (as was the case in this project). The challenge was to process and group geometric vectors into high-level structures which can be semantically tagged as features of a typical building: floors, ceilings, walls, doors, and corners.

Representation Scheme

The representation scheme we present partitions the environment into topologically linked spatial units. Each spatial unit represents a volume of space, and its properties represent the properties of the original 3D environment (see Figure 1). The precision of spatial units can be set at run-time, though our default setting is such that each node is approximately the width of an agent in the environment. That way, an agent can occupy a single node in the representation scheme. Nodes can represent empty space, structural features like walls, and means of passage, such as doorways. These structural features of architectural spaces are important, for example, in military simulation like CQC room-clearing protocols.

Figure 1 presents a visualization of our representation scheme. The symbols indicate the location of of spatial units. A map is produced for each floor of the scene which helps us confirm that our detectors (described below) capture the features present in the 3D scene graph. We use different coloured symbols to delimit different spaces for a given floor, indicating walls for each space, while the actual symbols (circles and lozenges) are determined by the labels associated with the spatial units. In Figure 1, labels distinguish corners based on whether they are concave (slim lozenges) or convex (regularly-shaped lozenges). To help us further verify that our detectors captured the appropriate features, our software can also output topological maps (Figure 2) and 3D spatial representation (Figure 3).

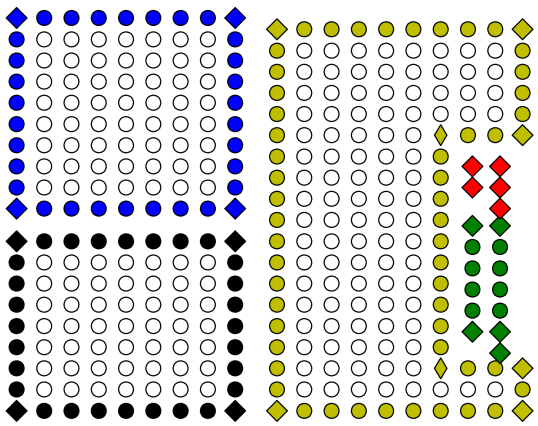


Figure 1: 2D visualization of the spatial units. Three large rooms were found, with two small spaces represented in red and green. Those two spaces are a staircase, which is a type of space that is not currently differentiated by our software.

Detectors

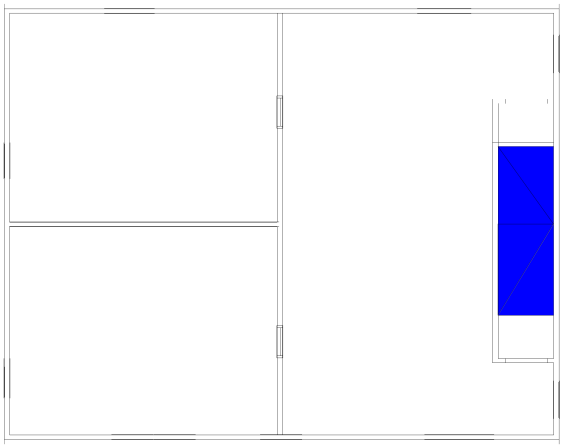
At a high level, transforming 3D scene graph files into our intermediate spatial representation is straightforward. Our software uses the OpenSceneGraph C++ library[1] which we chose because of its support for a wide-range of file formats. As mentioned above, the hierarchy is ignored because it was found to be inconsistent across different files and much of our aim was to make a general-purpose tool.

The file format we used in this project was OpenFlight, and the 3D files we used were acquired from our research partners at CAE Professional Services. The two main files we used for testing were a simple 3D house and a large, multi-building complex. While the multi-building complex made good use of the hierarchical structure of the OpenFlight format, the simple 3D house had oddly grouped vectors (for example, a portion of one wall was grouped with a wall on the opposite side of the house).

The first part of the process involved splitting up each surface in the scene graph into triangular faces. Each new triangular face is a container for the three vectors that define it, the plane equation of each face, as well as their normal vectors. This process assures that the initial stages of processing is the same for most files, by processing at a level common to most 3D files (the level of vectors). While this process increases the computational complexity of our software, it is an essential processes given possible grouping issues described above.

Because 3D files are often inaccurate at *some* degree of granularity, our library also makes use of an adjustable degree of accuracy. The number of significant decimals can be set to

[1] freely available at www.openscenegraph.org



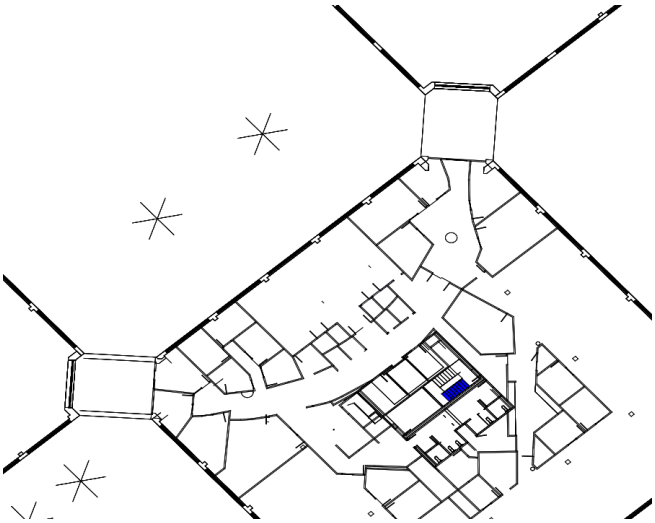


Figure 2: Blueprint-like layout of the simple house (top) and a portion of the multi-building complex (bottom).

the appropriate level for any given scene graph. To avoid data loss, our library does not cut off data points at specified decimals. Instead, the library rounds both coordinates and plane equations during any comparison made during run-time. This is useful, for example, when comparing the planes of two triangle faces. While visually two triangle faces might appear to be on the same plane, in the geometric representation they might, at a fine level of precision, exist on different planes. With the appropriate degree of accuracy set, our library would be able to conclude that the triangular planes in this example exist on the same plane and are, visually, parts of the same wall. This is ideal as our system is meant to represent what an agent experiences in their environment.

The next part of the process involves sorting the triangles according to their plane equation (to the set degree of accuracy).The sorting is done in reverse, as it is most likely the case that each new triangle is part of the most recently created plane. This search is exhaustive across the planes as we

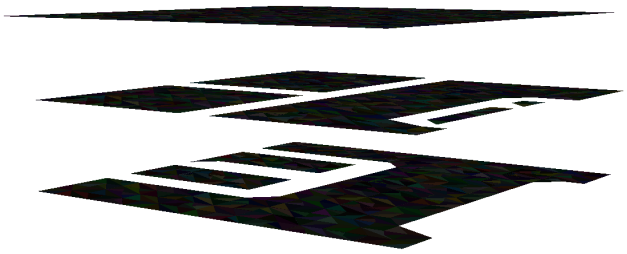


Figure 3: 3D representation of the spatial units. Three floors are shown, the middle one being the floor represented in Figures 1 and 2.

cannot be sure that a new triangle does not belong to a previously made plane (this alleviates the odd grouping problem mentioned above). If there is no matching plane, a new plane is created.

Finally, faces are created from each plane. Since each face is a collection of connected triangles, the software checks to see if each triangle in the face is either directly or distantly connected to every other triangle in the face. Planes with disconnected sets of triangles represent surfaces that exist on the same plane but are detached, as in, for example, two unattached wall segments.

Since the specific aim of our project was to represent structural features of an architectural environment, we used a minimum surface area to filter by size. Large terrain surfaces were also filtered our using a maximum surface area threshold. Areas of the faces are calculated by summing the area of the triangles that make them up. The remaining larger surfaces are candidate surfaces for structural features like walls, doors, floors, and ceilings. To further expedite calculations, we calculate a bounding box for each surface. To produce the bounding boxes, we iterate through all surfaces, finding the maximums and minimums of the coordinates, and calculate an axis-aligned minimum bounding box for each of the surfaces.

Floors and Ceilings In order to detect floors and ceilings of the 3D structure, we filter the list for surfaces in a horizontal orientation. The software then calculates the position of the horizontal faces relative to the boundaries of the entire scene graph. By calculating the relative position we are able to produce a candidate list of faces that represent either floors, ceilings, as well as the roof. A roof, for example, would have a high relative z-axis attribute. Once we determine one floor, we iteratively label all the floors and ceilings according to their z-axis attribute (above a floor is a ceiling, above that another floor, until a roof is reached).

Walls and Doors In order to detect walls and doors, we first gather vertically oriented surfaces. Importantly, for these detectors to work, the floors and ceilings have to already be detected. One assumption we make is that all walls expand from floor to ceiling and that doors do not, as they are embedded into walls. Door detection involves detecting vertically oriented surfaces that have a header above them, i.e. a space between the top of the surface and the ceiling.

Corners Corners are detected post-surface-processing. We use the spatial units to detect both convex and concave corners. When a spatial unit has three 'empty space' neighbours (units which represent empty space), that unit represents a convex corner. When a unit has seven empty space neighbours, that corner is a concave corner (This is discussed in more detail below).

Evaluation

We implemented two forms of evaluation. In the first, visualizations of the 3D scene graph were created and were qualitatively compared to the 3D scene graphs to compare structural elements. A second form of evaluation involved the use of a simulated agent to test our representation scheme in terms of its usefulness in high-level cognitive modeling.

Visualizations

At a high level, the process of conversion in our software can be seen as consisting of two steps. The first step is to sort the vectors into surfaces, and to provide semantic labels. At this point only floors, ceilings, walls, and doors are identified. The second step of the process is to convert that into the final representation scheme, for which a visualization is presented in Figure 2. We use the visualizations to qualitatively assess whether the structural details of the 3D scene graph are captured in the intermediate representation. The creation of the three visualizations (Figures 1, 2, and 3) are briefly described below.

Starting at a point slightly above the floor, the software performs a sweep of one entire floor, creating a new instance of a spatial unit in the surrounding directions: forward, backward, left, and right. Each space has a location that corresponds directly to a location in the original scene graph. Each point in space is determined to either be empty or intersecting with a face from the original scene graph. That is, the software determines which points represent walls or simply empty space in a room based on collisions with the geometry of the surfaces previously analysed (from the first phase of the conversion).

Once all spatial units are determined for a given room, which happens when no unit can expand farther in any direction, the software inspects the neighbours of each unit and tags them depending on the number of units that surround them.

Spatial units are tagged as open space when eight neighbours are found, representing the possibility to move in all directions if an agent stands in that location. They are tagged as walls when six neighbors are found, indicating a boundary in one direction. Finally, spatial units are tagged as convex corners when three neighbors are found or as concave cor-

ners when seven neighbors are found.

The map in Figure 1 is created by drawing symbols to represent each spatial unit of a given floor. Critically, this visualization helps us identify concave and convex corners. The maps in Figure 2 are constructed by going through the faces associated with a given floor and drawing lines between the vertices making up those faces. The precision with which the faces are drawn can be set high enough to be able to distinguish many small individual components, with no associated cost in terms of analysis of the faces. Any filled sections of the map represent angled surfaces, such as the shaded blue rectangles in Figure 2 (top) that show the sloping ceiling of the stairs.

Figure 3 shows a 3D version of our representation scheme, rendered in OpenSceneGraph. Every different space is made up of triangles linking the spatial units shown in Figure 2, displayed at the appropriate height.

These 3 visualizations were used to compare the structural elements in the 3D scene graph to the structural elements in our intermediate representation. We found that all structural elements (floors, ceilings, walls, doors, and corners) were properly identified and represented. Importantly, this corresponded to what was visible in the 3D environment, correcting for any hierarchical organization errors (such as the case of a wall segment being grouped with a wall in a different location).

Simulation

We present two sample simulations to test the sufficiency of our spatial representation scheme. Using one of the main 3D models we worked with during development, a simple house, we developed an agent capable of 1) navigating to all the doors on a floor of the house and 2) navigating to the corners. Our choice of simulation is relevant given our project goal. Navigating to the doors shows the relevance of applying symbols to the elements in a scene graph for use with a symbolic cognitive architecture such as ACT-R. Secondly, navigating to the corners is an example of how to use our representation scheme to reason and make judgements about the space.

Since our simulation is not based in a cognitive architecture, there is no variance in performance from trial to trial. We therefore evaluate our simulation qualitatively in a pass/fail manner.

In case 1), we instruct the agent to walk to every door on the current floor. For this simulation, we assume that the agent has full awareness of the layout of the room (i.e., can directly access the entire map). In this simple simulation, the agent simply looks up the location of each spatial unit our software has labelled as "door." A path finding algorithm produces a path from the agents current location to any given door in the scene. The path finding algorithm uses the spatial units as nodes connected to each other and determines the optimal path by calculating the shortest node-length. While many algorithms can be used for path finding, we used a steepest ascent hill-climbing algorithm for this simple task. This process is iterative, exhaustively bringing the agent to all doors on that floor.

In 2) we had the agent respond to (codified) instructions "Walk to all the corners of the room, in order" (i.e., following the perimeter of the room instead of walking randomly to any corner). This simulation uses a slightly more sophisticated spatial representation. Instead of simply choosing a corner at random, the agent represents the appropriate spatial units (corners) as being either "near" or "far." A similar use of descriptive categories was used in the ACT-R MOUT model, to represent egocentric distance. In our case, these distance terms are represented with overlaps, providing a fuzzy representation of distance for the agent. In this example, the agent can then identify the hard (i.e., near) and the deep (i.e., far) corner of the room.

Unlike doors, which are identified directly from the geometry, corners are identified through reasoning about the already-detected spatial units. Depending on the model, these sorts of calculations can occur automatically (pre-labeled given our system, using various techniques), or, potentially determined during simulation by the agent. Using the node system at run-time, in this way, would be far more efficient than working with the geometry directly. We think this is a good demonstration of potentially using our representation scheme for modeling at different levels. While it is possible to encode the corners in the representation scheme, there is also the option to use a first-person spatial representation and have the agent determine where corners are (presumably in some cognitive manner).

Because there is no source of variance in our two models, we qualitatively evaluated our simulations as either pass/fail. In order to do this, we created a video in which the location of the agent was represented within a topological map of the environment (see Figure 2). These videos showed that the agents in both simulations correctly navigated their environments to complete their given objectives.

Evaluation conclusion

We evaluated or software by first evaluating the detectors. We did this by creating visualizations of our intermediate representation which was created using our structure detectors. Although we tested only two files, a simple house and a multi-building complex, the 3D scene graphs of these two structures were dramatically different. Not only was the multi-building complex much larger, the structure contained a higher level of detail. Importantly for our project, the simple house also contained errors in hierarchical structure. In terms of parameters, the approach also generalizes well, as only the size of the spatial units had to be modified from one file to the other. Although we plan to do more extensive tests in the future, we consider our detectors successful.

We also tested our intermediate representation scheme by developing simple symbolic-level simulations. These simulations were designed to test the sufficiency of our representation scheme for modeling action in 3D environments. By using symbols such as 'door' and 'corner' we were able

to mimic the high level symbology often used by cognitive architectures (such as Soar or ACT-R). The success of our agents to complete their tasks, is good evidence that our intermediate representation is sufficient to support action like navigation.

Discussion

We feel that this representation scheme is both simple to implement and robust enough for a variety of modeling needs. The linked node system is reminiscent of both the Soar MOUTbot and ACT-R MOUT model representation schemes and should be able to support modeling at different levels.

For example, our models have some similarity to the Soar MOUTbots in that they have direct access to a complete map of the environment. Model 2 shows, however, our representation scheme also supports perception to support egocentric representations. This is shown in model 2's ability to represent corners as 'near' or 'far.' Algorithms similar to those used in model 2 could be used to simulate vision. For example, one set a maximum vision distance, and calculate that distance by adding together the widths of the spatial units. A complex vision system would be able to calculate an entire field of view and determine which nodes are visible given the agents current heading. It is even plausible with a small enough grain size for a modeler to restrict vision to nodes within foveal azimuth. Linked with a visual memory system, it is entirely plausible that an agent might have to attend to different portions of the intermediate representation in order to refresh its internal representation of the environment. The linked node design makes it easy to calculate features such as location, size, or distance, depending on the level of detail required by the modeler.

Conclusion and future work

It should be noted that this project represents preliminary work. While our software can be run pre-simulation-runtime, thus limiting the effects of computational complexity, it would be ideal to reduce complexity. It might be fruitful, for instance, to exploit OpenFlight files for hierarchical structure, regardless of any potential inconsistencies between files.

A significant amount of testing both of our software and our representation scheme would also help determine our current pitfalls. With an automated tool like the one presented in this report, an extended analysis of its performance across many different files is required. Doing so will allow us to hone our system so that the representation scheme and proper labels are being generated across many different renderings.

Furthermore we would like to test the range of language-like representations that can be inferred from our representation scheme. While we can infer features such as "corner," we would like to develop more simulations to determine if we can infer more complex features such as "behind the wall."

Also, from our sample simulations we know that our representation scheme is adequate for a simple simulation, in a building without furniture and small objects, but it is currently unclear whether it would scale-up to give the representational granularity required of a more complex simulation. In fact, it seems that a goal of the long-term research would be to determine the optimal level of granularity for such simulations.

Our team would also like to investigate how to apply the output of our software to a simulation within the 3D environment. To this end it might be useful to use a symbolic cognitive architecture to determine the software needs of both the 3D simulation and the architecture. Furthermore this process may illuminate ways to limit complexity or help the system deal with the dynamic nature of the simulation. For instance, the use of a cull visitor (used at run-time to render vectors) to help determine if what the agent is seeing in real-time corresponds with what our system had previously detected. For example, an explosion might have left a hole in a wall and a cull visitor may help us detect such environment changes. Such a system might provide a quick solution, with minimal geometric processing, to update our representation as needed.

Finally, our team would like to investigate how to automatically generate environments from a natural language-like description. Currently, design of realistic virtual environments takes many hours of content creation both in terms of the structural elements of buildings as well as furnishing and other features of large buildings. Automating even parts of this process would help bring down personnel costs associated with content creation.

Acknowledgments

This work was generously funded by an NSERC Engage grant and CAE Professional Services.

References

Anderson, J. R., & Lebiere, C. C. (Eds.). (1998). *The atomic components of thought*. Lawrence Erlbaum Associates. (ACT-R book)

Best, B. J., & Lebiere, C. (2003). Teamwork, communication, and planning in ACT-R agents engaging in urban combat in virtual environments. In *Proceedings of the 2003 ijcai workshop on cognitive modeling of agents and multi-agent interations* (pp. 64–72).

Fleetwood, M. D., & Byrne, M. D. (2003). Modeling the visual search of displays: A revised ACT-R/PM model of icon search based on eye-tracking and experimental data. *Human-Computer Interaction*, *21*(2), 152–197.

Laird, J. (2001). It knows what you're going to do: Adding anticipation to a Quakebot. In *Proceedings of agents* (pp. 385–392).

Ting, B. S., & Zhou, S. (2009). Dealing with dynamic changes in time critical decision-making for MOUT. *Time*(May), 427–436.

Wang, R., & Qian, X. (2010). *Openscenegraph 3.0*. Birmingham, UK: Packt Publishing Ltd.

Wray, R. E., Laird, J. E., Nuxoll, A., Stokes, D., & Kerfoot, A. (2005). Synthetic adversaries for urban combat training. *AI Magazine*, *26*(3), 82–92.