module, or through direct channels for low frequency and high frequency spatial information (e.g., shape is low frequency while texture is high frequency).

Some level of spatial processing has recently been incorporated into multiple cognitive architectures including ACT-R (Schunn & Harrison, 2001), Soar (Chandrasekaran, 2002; Chong & Wray, 2002; Wintermute & Laird, 2007), and ICARUS (Langley & Choi, 2006). The CASEMIL spatial module provides fundamental low-level routines that often simplify the cognition necessary to interact in a virtual world, but does so in a way that is architecture agnostic to the extent that it is possible, allowing for integration of neural network architectures capable of visual and spatial processing (e.g., O'Reilly & Munakata, 2000).

To achieve this, we focused on aspects of spatial cognition, rather than on the aspects of existing architectures. For example, the fundamental organization is along the basic division of “what” and “where” pathways. Using this division, key properties of human spatial cognition we used to guide the development of the CASEMIL Middleware include the following:

Spatial Representation (“where” pathway):

- The visual scene is represented hierarchically
- The egocentric frame of reference is primary (obtained for “free” from the environment)
- The exocentric frame of reference is secondary (derived with effort from the environment)
- Multiple representations are simultaneously active (e.g., exocentric and egocentric)
- Spatial representation is multi-modal (vision, audition, and other senses are combined)

Object Representation (“what” pathway):

- Category membership
- Specific identity
- Location: Centroid, Extent, Axes
- Perceptual Attributes: Motion, color, texture, structure

Many of these divisions have been recognized before (e.g., Schunn & Harrison, 2001), and we will not attempt to review this literature here. However, we will point out that CASEMIL makes particular contributions in combining mechanisms for determining object identity (through leveraging the OpenCyc ontology) and maintaining object permanence (described below) with an interface that allows the agent to interact directly with the virtual environment without using the API provided by the environment developer – this will be explained in detail below. CASEMIL also differs from prior approaches where they have emphasized theoretical issues, while CASEMIL is focused on the practicalities involved in implementing a running system based on the application of cognitive principles.

Currently, we have augmented CASEMIL with a hierarchical grouping algorithm which enables it to take a visual scene composed of individual elements, and group those elements into a tree, either in a purely bottom-up fashion, or with a top-down influence. This grouping, or clustering, is an agglomerative clustering method that recursively combines the two 'nearest' clusters based on a distance metric. Bottom-up grouping is achieved by using a pure Euclidean distance metric. However, top-down influences on grouping may also be biased by semantic properties (e.g., group membership), or by other attributes (e.g., color). Thus, a particular object type can be given as an input to the clustering algorithm, producing a hierarchical tree representing the spatial configuration of the objects that match that type.

In CASEMIL, spatial processing can be viewed as a pipeline running from the simulation environment to the extracted representation. Top-down influences are produced through interaction with the cognitive architecture itself, and largely driven by the cognitive architecture's own theory of attention. Thus, CASEMIL must provide an interface to both the superset of available visual elements, and an interface to a filtered subset of those visual elements where the filtering is driven by the cognitive architecture. The CASEMIL processing pipeline depicts unfiltered content with the brown “Lo-F Spatial” and “Hi-F Spatial” boxes, while the “Spatial Buffer” provides access to hierarchical representations and the attention filtering selection processes.

2.1 Filling in Missing Object Features: Identity and Velocity

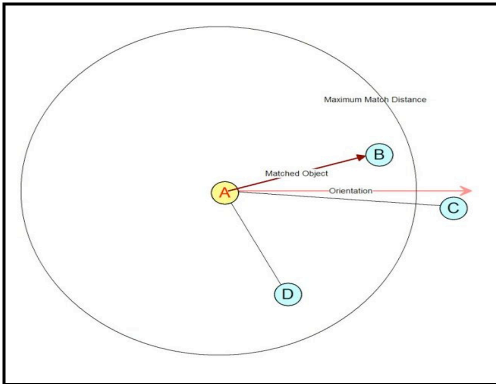
The general object representation for the CASEMIL middleware layer includes features that are widely accepted to be fundamental: these are features that “pop out” of visual displays and depend more heavily on parallel processing. These basic feature commitments include location, shape, color, orientation, category, and motion. The CASEMIL layer commits to making this information available, even if the virtual environment does not provide it.

In addition, objects have identity: it is easy to visually track an individual item among a group of identical items. However, if a virtual environment does not provide identity information (many do not), and provides discrete updates (all of the systems we examined do this), it is extremely difficult at the cognitive level to maintain a focus of attention on a moving object.

In our explorations with the dTank environment (Ritter, Kase, Bhandarkar, Lewis & Cohen, 2007) we discovered that even though information for agents within the field of view was passed along, this information included neither

the identity of individual agents nor their direction and speed (velocity vector). Further, the dTank version we used implements a (deliberately) noisy visibility determination, which means even stationary objects might not be visible on subsequent update cycles. We viewed this as an opportunity to explore solutions to the problem of missing fundamental object information within a simple environment, and describe the approach we implemented here.

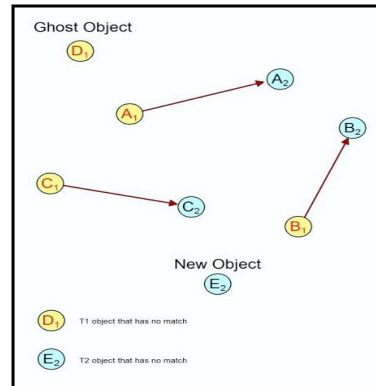
The fundamental problem we are addressing here is reconciling a list of objects and locations at an initial time (T1) with a list of objects and their locations at (T2) in the absence of identity information. Thus, a perceptual object identity needs to be established. Once this is accomplished, the change in location from T1 to T2 for an object can be divided by the time between updates to deduce the velocity of that object. First, however, the correspondence problem needs to be addressed. This requires developing an algorithm for matching objects at T1 and T2. Graphically, the problem is represented below:



Here, an object A at time 1 (T1) is being considered as a potential match against objects B, C, and D, all viewed at time 2 (T2). The number of possible matches is constrained by the maximum distance object A might travel (this value is likely domain dependent), and the previous velocity vector (if it is known), or the orientation of the object (if the velocity is not known). In the example above, this heuristic is suggesting A is much less likely to move sideways than it is to move forward (so that it does not match D), and that it could not get as far as C, leaving B as the most plausible match at T2. However, this gets somewhat more complicated when there are multiple moving objects in a local area, and they might not be visible on every cycle, which is exactly the situation presented by the dTank virtual environment.

The figure below presents an example of a typical situation in the dTank environment. The subscripts of the items represent whether these objects are visible at the first (1) or second (2) cycle. Object A, B, and C are all successfully matched. However, object D appears only at T1, while object E appears only at T2. The task of the perceptual layer is to avoid matching D and E, and instead present a new object, E, at T2, while maintaining a representation for the missing object D at T2.

The spatial module code we have developed accomplishes exactly this: objects have perceptual permanence over a short timescale, where the duration of persistence without perceptually refreshing the object is configurable by the cognitive architecture. A default of ~1 second has proven quite successful in the dTank environment, allowing agents to persist a perceptual representation over a short timescale while preventing them from having perfect situational awareness in the absence of scanning.



It is difficult to overestimate the importance of this simple mechanism. Without some form of object persistence from one update to the next, agents based on cognitive architectures are forced to treat every moment as if it is a new situation, with very little overlap to the previous situation. The persistence mechanism, however, allows generally for processing based on motion where the virtual environment does not directly support it, and further supports a short-term perceptual prediction of an object if it is temporarily occluded.

While this might be considered to be a process that should be implemented by the cognitive architecture, this perhaps forces the cognitive system to do something that the perceptual system can provide with very little effort, and this perceptual filling in is something that is likely to confound cognitive architectures simply because they are not in general designed to handle problems like this that are more naturally parallel. Whether this functionality belongs in the spatial module or the core CASEMIL layer, however, is less obvious. We now turn to the system architecture itself.

3. Hub Protocol Overview

Our hub is designed to be the middleware that provides data transfer between cognitive models and virtual environments. It communicates through sockets and can

be operating on a computer that is independent of both the cognitive model and the virtual environment. The communication is asynchronous; each direction of data flow is handled separately and does not employ any sort of time-synchronization methodology, other than time-stamping each message. The data is sent in each direction as fast as the hub receives it, without waiting for a response. This commitment to real-time communication is necessary because of the nature of many target environments such flight simulators and first-person shooter games that do not provide time-synchronization capabilities. Avoiding such overhead as time synchronization and response acknowledgment allows the communication between simulation environment and cognitive model to be as efficient as possible to allow the model(s) the best chance to keep up with the simulation.

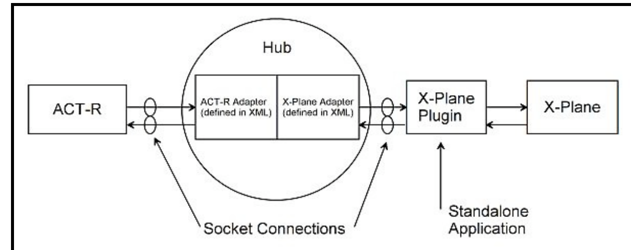
The hub also allows us to employ the principle of information hiding, where the details of the environment are hidden from the model (and vice versa). Information hiding is important because the model doesn't necessarily need to know the implementation details of the virtual environment. From a principled cognitive point of view, this separates knowledge into multiple layers, in particular differentiating general domain skills portable across environments and interface-specific skills that are likely to change across and even within an environment. The former must be represented while the latter is optional. This allows for lightweight as well as high-fidelity models. Only the model hub adapter will have to change to reflect the difference in levels of abstraction.

The hub design includes adapter software that can be used to translate from one language to another, such as going from specific cognitive commands to the more general hub language. Adapter software used by the hub reads the mappings from an XML document. The adapter software itself need not be specific to a model or environment, it simply reads in data from the XML document specific for that application and instantiates the software objects that are necessary to translate from one language to another. The ability to use a generic adapter object will allow a greater user base; the user will simply have to create an XML document as opposed to having to write a new program to be added to the hub framework. Additionally, by not requiring the user to extend the hub code itself we are making the system more secure and robust.

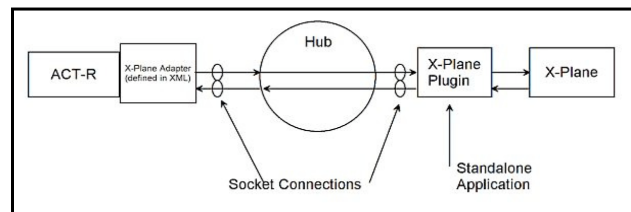
3.1 Hub Flexibility

The adapters are intended for use within the hub but their mappings could also be used in other applications. For example, the model could parse the adapter XML and use the environment's API directly. In this case, the hub simply forwards the communications and provides logging and debugging functionality. If neither of these functions is needed, the hub could be bypassed completely allowing for improved efficiency.

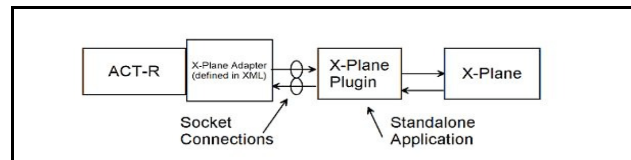
The flexibility is diagrammed in the following figures, which show an ACT-R (Adaptive Control of Thought-Rational) (Anderson, et al, 2004)) model communicating with X-Plane, either using the hub and the adapter, or using the hub as a pass-through, or bypassing the hub and using the adapter directly.



1 The hub's adapters handle the language translation and the X-Plane plugin provides the interface for communication with the simulation environment



*2*The hub simply serves as a server that forwards communications and provides logging functionality



*3*The hub can be completely bypassed

3.2 Hub Language

The hub language needs to be general enough to represent any data that may need to be passed between a model and an environment without becoming redundant or confusing. To keep the agent/hub/environment mappings as comprehensive and non-arbitrary as possible and to improve model generality and portability, we are using the OpenCyc ontology to validate the hub language (http://www.cyc.com/cyc/opencyc/overview). OpenCyc is the open source version of the Cyc (Lenat 2001) ontology. The hub will refer to concepts, which can include actions performed by the model and information passed by the environment, by using the appropriate OpenCyc entry name. OpenCyc also specifies the structure of these concepts, such as the arguments used (e.g. the degree of roll) and their semantics.

To begin building our hub language, we can specify a small set of concepts that are required for a model to interact in a basic way with the environment. These

concepts can be divided into two types: actions (the things a player can do) and percepts (the things a player can perceive). Actions represent the data that a model can send to an environment, via the hub, while percepts represent the data that an environment sends to the model, via the hub. Since all of the data is translated into the hub language, models can operate in any number of environments. Relying on a general domain ontology provides both an opportunity and a requirement for more general, less environment-specific models.

Each adapter assumes a certain compatible syntax from the environment and model, respectively, to be able to perform the translation automatically. If the model or environment expresses commands or information at a different level of abstraction, translation to such syntax is necessary. For instance, the model could issue commands directly at the level above, in which case the adapter could directly translate them into the hub language. Or the model could perform lower-level actions, such as key presses, mouse movements, joystick commands, etc. In that case, those commands should first be translated into the upper-level syntax, at which point the adapter could again translate them automatically into the hub language. The knowledge used for the translation of model commands across levels is exactly the knowledge formalized by a user's manual, i.e. how specific interface actions translate into abstract, higher-level functions and vice-versa. This translation process might itself be formalized and automated in a similar way to the hub translation process. A similar analysis applies in the other direction regarding the information communicated to the model by the environment. These two translation processes could be combined or they could be left separate. While the former is possible and more compact, the latter is probably preferable because:

- the two translation functions are orthogonal. One translates between levels while the other translates between languages.
- they are modular. While the language translation adapter is essential to our approach, the level translation is only required if the model operates at the lowest, keystroke level.
- bridging the gap between levels might make the XML specification of the hub translator too complex. There might be a one-to-one mapping between keystrokes at the lowest level and commands at the higher level, but that is usually not the case

3.3 Hub Coordinates

The final translation problem to overcome is one concerning coordinates. It is difficult to find a general approach to the problem of establishing the position of a player in a virtual environment. The general position approach must be scalable enough to describe locations on a desk as well as another planet. To that end, we've adopted the Military Grid Reference System (MGRS). MGRS is a coordinate system that is used to describe locations by specifying a series of grids-with-grids that each refine to a smaller area. It is designed to be used on a planetary body, but the concept of refining the position using grids can be applied to other environments. Latitude and longitude information from a flight simulator can be directly mapped to MGRS coordinates. World-centric X,Y, and Z coordinates from a first-person shooter can be mapped to MGRS coordinates in a system created for that specific environment. This environment-specific MGRS system will also help to standardize the processes required for mapping from world-centric coordinates to the model's egocentric coordinates.

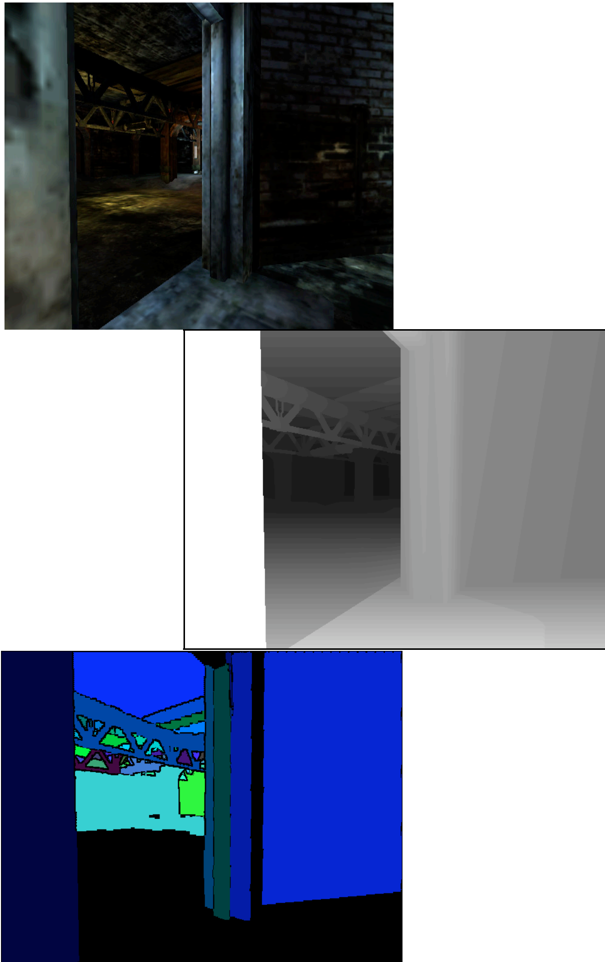
3.4 Leveraging the OpenCyc Ontology

A key problem in providing a general-purpose interface between simulation environments and cognitive architectures is the resolution of representational differences between the terms used by the simulation, especially those exposed to external agents by its API protocols, and those used by the cognitive model. It is often the case that the cognitive model is designed specifically for a given simulation and just ends up using its terms, but that is an undesirable state of affairs. Primarily, it prevents reuse of models across even very similar domains that often differ even slightly in their terms. Moreover, it prevents the development of general-purpose infrastructure such as modules and interfaces that support the development and reuse of general models. As suggested by Ball, Rogers, and Gluck (2004), our solution to that problem has been to commit to adopting a general ontologies, OpenCyc, the open-source version of the Cyc general knowledge base (Lenat, 1990), to translate from simulation-specific terms to domain-general terms that would in turn enable model reuse across simulations, a key component of making high-fidelity cognitive models more broadly applicable and affordable.

However, a key problem lies in matching terms used by the simulation, the cognitive model and the general ontology (or ontologies since despite our use of Opencyc we want to remain uncommitted to any specific ontology). While we have successfully hand-mapped specific cases to the OpenCyc ontology, that methodology needs to be automated and systematized to avoid the same kind of ad hoc process that led to the overly specialized nature of cognitive models and their connection to simulation environments in the first place and resulted in the need for general ontologies in the first place. Fortunately, spurred on by similar incompatibilities over broad enterprises such as the semantic web, a field called ontology matching has emerged to tackle the basic problem. See Euzenat and Shvaiko (2007) for an overview of the problem and its solutions.

4. Extracting Depth Information from Virtual Environments

Although some environment information is easy to come by using the provided interface with the environment, other data is much more difficult to obtain. The rest of this paper focuses on the challenges that extracting depth information from various three-dimensional environments presents. Despite these challenges, we have found that depth extraction is possible for the vast majority of games, without reliance on specific hardware. The following images show a raw image from Unreal Tournament, an extracted depth map, and a segmented image as processed by the CASEMIL system:



Extracting depth information in a practical way requires working on the level where 3D applications interface with hardware. Depending on the game, this level uses either OpenGL, the platform-independent API for hardware acceleration, or DirectX, Microsoft's Windows-only alternative to OpenGL. Our discussion of the development of a general purpose depth extraction method will include both of these APIs.

4.1 Direct vs Indirect Approach

We began by looking for the simplest possibly way to extract depth information. OpenGL provides the function glReadPixels that allows for direct depth extraction. Unfortunately, this function is very slow for the following reasons:

- The function implicitly locks the depth buffer in order to ensure a consistent state.
- The depth buffer information must be transferred from GPU memory to system memory.
- The native format of the depth buffer is converted into a 32-bit floating-point format using the conversion equation: D' = clamp((D - Dmin) / (Dmax - Dmin) * GL_DEPTH_SCALE + GL_DEPTH_BIAS), where **clamp** denotes a function that clamps its input to the range [0, 1], **Dmin** and **Dmax** denote the minimum and maximum depth values, respectively, and **GL_DEPTH_SCALE** and **GL_DEPTH_BIAS** are constants

Unlike OpenGL, Direct3D does not allow for straightforward universal depth extraction, although it is possible to create a depth buffer surface using this API. However it is not possible to read the values without first performing a lock. Unfortunately, the vast majority of video cards do not support locking the depth buffer. Therefore, we are required to take the indirect approach when using Direct3D. Although this is considerably more complicated than the direct approach, it should work on the vast majority of graphics cards, and with some effort, could be made to perform faster than any alternative.

The indirect approach utilizes pixel shaders, which are essentially snippets of code that are compiled and run on the graphics card. Pixel shaders can access the depth value at each rendered pixel and write to textures, which may be locked on all graphics cards; these are both important for our purposes.

By accessing depth information, it is possible for a pixel shader to produce an "image" of the depth buffer. This is exactly the approach used for shadow mapping (http://www.gamedev.net/reference/articles/article2457.asp), a technique in games where depth information in a scene is rendered from the point of view of a light source, and the resulting render information is used to display shadows.

Typically, rendering depth information using a pixel shader is accomplished by creating separate render targets and then rendering the scene using an appropriate pixel shader. However, at least with an application so designed, it would be possible to render the scene only once, using the custom pixel shader, producing only depth information as output. In this scenario, assuming enough of the work is done on the GPU, the indirect approach will perform much faster than the direct approach.

4.2 The Indirect Approach

The most effective approach involves intercepting OpenGL/DirectX calls with the aid of a *proxy DLL*. A proxy DLL is essentially a wrapper for the original DLL, which delegates most of the functionality to that DLL. At any point when the proxy DLL has control of a thread (i.e. whenever it is invoked by the callee), the proxy DLL can execute arbitrary code prior or subsequent to delegation. The proxy DLL is used to create a depth rendering pass that must consist of a complete rendering of the scene from the point of view of the camera representing the player or agent. The rendering must be performed using specific vertex and pixel shaders. A special vertex shader is necessary to compute depth information, and pass the depth along to the pixel shader (pixel shaders do not otherwise have access to depth values). Meanwhile, the pixel shader merely outputs depth information to the target texture. The depth render pass can either capture the various calls to the primitive rendering methods, and repeat them later in a separate rendering pass or somehow override the game's rendering pass so the output is depth information rather than color information.

4.3 Separate Rendering Pass

A separate rendering pass is the most straightforward solution. This would assure that no assumptions of a given game are broken and would give us precise control over the format and size of the ouput depth buffer. Additionally, this approach does not require knowledge of what the game is trying to do with its own vertex and pixel shaders. Unfortunately, a separate rendering pass would be incredibly slow. All information passed to Direct3D during the rendering of the scene must be captured by *deep cloning*, because the game may be storing information in dynamically allocated memory, or may be modifying geometry during the rendering of the scene. In addition to the overhead of deep cloning the entire scene, there is the overhead associated with a completely separate rendering pass. Only taking into consideration the second half, performance of rendering would drop by at least a factor of 2, while taking into consideration the cost of dynamic memory allocation, the performance could easily be an order of magnitude slower than the unmodified game.

4.4 Replacement Rendering Pass Overview

This leaves us with a replacement rendering pass which would subvert the game's primary rendering pass. Since this approach does not call an additional rendering pass, the performance should not be significantly degraded, although there are several other disadvantages that must be overcome:

- The depth information must be encoded in the same format as the game's main render target.
- There is a possibility that subverting the main rendering pass may lead to unforeseen side-effects.
- Naively, color information is not available. However, this limitation could be circumvented by only doing depth renderings every other frame.
- It requires dynamic rewriting of the vertex shaders, which in turns involves parsing and code generation.

Despite these drawbacks, this solution is promising because of its high performance and likely robustness.

4.5 Replacement Rendering Pass Implementation

The depth extraction algorithm involves first identifying the vertex shaders used to transform points into projection space and than rewriting them to store the homogenized depth coordinate in an output register. Finally, the application's pixel shader needs to be replaced with one that grabs the depth value and encodes it in the output format expected by the render target. Although this algorithm is straightforward, the vertex and pixel shaders present challenges that must be overcome.

4.6 Vertex Shaders

Vertex shaders may be used for many purposes. In fact, general-purpose computation (http://www.gpgpu.org) is possible on the GPU. However, the majority of games use them only to transform, animate, and "light" vertices (here, *light* has a very broad definition that refers to the creation or propagation of any per-vertex data which needs to be interpolated across pixels.

Vertex shaders that transform vertices are required to output a point. However, they may also output other data that will be used by the pixel shader, by storing the data in one of twelve so-called *output registers*. In DirectX 9.0c, an application may use any register to hold the output position, but before it can do so, it must declare to DirectX which register that will be. Thus, by parsing the sequence of tokens that make up a vertex shader and looking for such a declaration, it's possible to identify shaders used to transform points. Vertex shader rewriting consists of parsing the tokens that make up a vertex shader to identify any and all places where the output position register is set. Following these places, instructions must be inserted to store the depth rendering (z/w) in one of the output registers. It's possible that some other instructions may need modification as a result of the newly-inserted instructions.

The final step involves replacing the application's pixel shader with a custom one, which reads the z/w value stored in an output register, and encodes this depth information in the format of the application's main render target (typically 24 or 32 bits per pixel, but possibly as low as 16 bits per pixel). Although somewhat tedious because of the need to support multiple output formats, there is nothing particularly challenging in this step

5. Summary

This research establishes that robust depth extraction is feasible for a large majority of 3D applications. The general approach involves creating a proxy library for a specific version of either OpenGL or DirectX. The proxy library merely passes through to the underlying library in the majority of cases. However, in a few key cases, the proxy library makes alternate or additional calls to the underlying graphics library, in order to extract depth information.

6. Conclusion

Our work to-date has demonstrated great potential for supporting the interaction of cognitive agents with virtual environments through the extraction of depth information. This information not only gives veridical depth to the surfaces and objects in the environment, but also provides a basis for localizing objects through figure-ground separation which can potentially aid image-based recognition processes. Further, the combination of platform-agnostic extraction of depth information with ontologically-based representations of categorical memberships and attributes enables the creation of true general-purpose cognitive agents that are able to interact in a variety of environments without modification.

7. Acknowledgments

The work reported here was partially supported through the Air Force Office of Scientific Research by an STTR contract for topic AF07-T020. We would also like to thank Dave Jilk from eCortex Inc. and John de Goes from N-Brain Inc. for their contributions to the software architecture and engineering presented in this paper.

8. References

Anderson, J. R. (2007). *How Can the Human Mind Occur in the Physical Universe*? NY, NY: Oxford University Press.

Anderson, J. R., Bothell, D., Byrne, M. D., Douglass, S., Lebiere, C., & Qin, Y . (2004). An integrated theory of the mind. *Psychological Review* 111, (4). 1036-1060.

Anderson, J. R., & Lebiere, C. (1998). *The Atomic Components of Thought*. Mahwah, NJ: Lawrence Erlbaum Associates.

Ball, J., Rodgers, S., & Gluck, K. (2004). Integrating ACT-R and Cyc in a large-scale model of language comprehension for use in intelligent agents. In *Papers from the AAAI Workshop*, Technical Report WS-04-07, pp. 19-25. Menlo Park, CA: AAAI Press

Best, B. J. & Lebiere, C. (2006). Cognitive agents interacting in real and virtual worlds. In R. Sun (ed.), *Cognition and Multi-Agent Interaction: From Cognitive Modeling to Social Simulation*. Cambridge University Press; New York, NY, 186-218.

Chandrasekaran, B. (2002). Multimodal Representations as Basis for Cognitive Architecture: Making Perception More Central to Intelligent Behavior. In *Intelligent Information Processing*, Musen, M., Neumann, B. & Studer, R. (Eds.), IFIP International Federation for Information Processing Series, Vol. 93. Dordrecht: Kluwer Academic Publishers.

Chong, R.S., & Wray, R. E. (2002). An EPIC-Soar model of concurrent performance on a category learning and a simplified air traffic control task. In *Proceedings of the 20th Annual Conference of the Cognitive Science Society*. Lawrence Erlbaum Associates.

Euzenat, J., & Shvaiko, P. (2007). *Ontology Matching*. Springer.

Langley, P., & Choi, D. (2006). A unified cognitive architecture for physical agents. *Proceedings of the Twenty-First National Conference on Artificial Intelligence*. Boston: AAAI Press.

Lenat, Douglas. Hal's Legacy: *2001's Computer as Dream and Reality.* "From 2001 to 2001: Common Sense and the Mind of Hal"

Newell, A. (1990). *Unified theories of cognition*. Cambridge, MA: Harvard Univ. Press.

O'Reilly, R.C. & Munakata, Y. (2000). *Computational Explorations in Cognitive Neuroscience: Understanding the Mind by Simulating the Brain*. Cambridge, MA: MIT Press.

Ritter, F. E., Kase, S. E., Bhandarkar, D., Lewis, B., & Cohen, M. A. (2007). dTank updated: Steps towards exploring moderator-influenced behavior in a light-weight synthetic environment. *Proceedings of the 16th Conference on Behavior Representation in Modeling and Simulation*.

Ritter, F. E., Van Rooy, D., St. Amant, R., & Simpson, K. (2006). Providing user models direct access to interfaces: An exploratory study of a simple interface with implications for HRI and HCI. *IEEE Transactions on Systems, Man, and Cybernetics, Part A: Systems and Humans*. 36(3). 592-601.

Schunn, C. D. & Harrison, A. M. (2001). ACT-RS: A neuropsychologically inspired module for spatial reasoning. In *Proceedings of the Fourth International Conference on Cognitive Modeling*, pp. 267-268. Mahwah, NJ: Lawrence Erlbaum Associates.

Wintermute, S., and Laird, J. E. (2007). Predicate Projection in a Bimodal Spatial Reasoning System. *Proceedings of the 21st National Conference on Artificial Intelligence (AAAI)*, Vancouver, B.C., Canada.