Smart, P. R., Scutt, T., Sycara, K., & Shadbolt, N. R. (2016) Integrating ACT-R Cognitive Models with the Unity Game Engine. In J. O. Turner, M. Nixon, U. Bernardet & S. DiPaola (Eds.), Integrating Cognitive Architectures into Virtual Character Design. IGI Global, Hershey, Pennsylvania, USA.

Integrating ACT-R Cognitive Models with the Unity Game Engine

Paul R. Smart[1], Tom Scutt[2], Katia Sycara[3] and Nigel R. Shadbolt[4]

[1]Electronics and Computer Science, University of Southampton, Southampton, SO17 1BJ, UK.

[2]Mudlark, 1 Brown Street, Sheffield, South Yorkshire, S1 2BS, UK.

[3]Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh, PA 15213, USA.

[4]Department of Computer Science, University of Oxford, Oxford, OX1 3QD, UK.

INTRODUCTION

Cognitive architectures are computational frameworks that can be used to develop computational models of human cognitive processes (Langley et al., 2009; Taatgen & Anderson, 2010; Thagard, 2012). Cognitive architectures have been useful in terms of advancing our understanding of human cognition in specific task environments, and they have also been used to support the development of a variety of intelligent systems and agents (e.g., cognitive robots). Although a variety of cognitive architectures are available, such as SOAR (Laird, 2012; Laird et al., 1987), ACT-R (Anderson, 2007; Anderson et al., 2004) and CLARION (Sun, 2006a; Sun, 2007), the focus of the current chapter is on ACT-R. ACT-R is a rule-based system that has been widely used by cognitive scientists to model aspects of human cognitive performance. It is also one of the few cognitive architectures that has an explicit link to research in the neurocognitive domain: the structural elements of the core ACT-R architecture (i.e., its modules and buffers) map onto different regions of the human brain (Anderson, 2007), and this enables cognitive modelers to make predictions about the activity of different brain regions at specific junctures in a cognitive task (see Anderson et al., 2007)[1].

Given their role in the computational modeling of cognitive processes, it is perhaps unsurprising that cognitive architectures have been used in the design of intelligent virtual characters. The SOAR architecture, for example, has been used to control a humanoid character that co-exists in a virtual 3D environment alongside a human-controlled avatar (Rickel & Johnson, 2000). The aim, in this case, is to provide a training environment in which the SOAR-controlled character possesses expertise in a

[1] ACT-R is, in fact, somewhat unique in proposing a neuro-anatomical mapping for its structural elements. The goal module, for example, is deemed to map onto the anterior cingulate cortex, whereas the procedural module is deemed to map onto the caudate nucleus (see Anderson, 2007, pp. 74-86). This neurological mapping enables ACT-R models to be used in conjunction with brain imaging studies (Anderson et al., 2005). For example, the activity in specific modules (e.g., the goal module) can be used to predict activity in specific brain regions (e.g., the anterior cingulate cortex) at specific points in a cognitive task.

particular domain of interest and then mentors human subjects as they progress through the stages of skill acquisition. This is an excellent example of the productive merger of cognitive architectures with virtual environments. As Rickel and Johnson (2000) point out, virtual tutors that cohabit a virtual environment with human players benefit from the ability to communicate nonverbally using gestures, gaze, facial expression and locomotion. In addition, the virtual agents can closely monitor the behavior of human subjects in a way that is not typically possible outside of a virtual environment; for example, a user's actions and field of view can be carefully monitored to determine their likely focus of attention. Rickel and Johnson's (2000) work is also a clear example where a multidisciplinary focus is required to engineer the intelligent virtual agent: the development of an intelligent virtual tutor draws on technical and scientific advances in the fields of knowledge elicitation (Shadbolt & Smart, 2015), knowledge modeling (Schreiber et al., 2000), human expertise development (Chi et al., 1988), and educational psychology.

Another example of cognitive architectures being used in virtual character design is provided by Best and Lebiere (2006). They used ACT-R to control the behavior of synthetic team-mates in a virtual environment as part of military training simulations[2]. As is the case with virtual tutors, the use of a virtual environment is important here because it enables human actions to be closely monitored in a way that is difficult (if not impossible) with real-world environments. As a result of such monitoring, the behavior of ACT-R-controlled virtual agents can be adjusted in ways that respect the norms and conventions of team-based behavior. This is a topic of particular interest in the context of military behavior simulations, where issues of team coordination and the synchronization of team-member responses (often in alignment with doctrinal specifications) are all-important.

In addition to situations where cognitive architectures have been used to control virtual characters as part of training simulations or tutoring applications, there are a number of other research and development contexts where one sees a convergence of issues relating to cognitive architectures, virtual environments and virtual character design. These include the use of cognitive architectures to model the behavior of human game players (Moon & Anderson, 2012), as well as the use of cognitive architectures and virtual environments to study issues in embodied, extended and embedded cognition (Smart & Sycara, 2015b). Other areas that benefit from the integrative use of cognitive architectures and virtual environments include computer simulations of socially-situated behaviors and collaborative problem-solving processes (Smart & Sycara, 2015b; Sun, 2006b), the development of believable game characters (Arrabales et al., 2009), the implementation of virtual coaches and mentors in therapeutic applications (Niehaus, 2013), the creation of game characters with psychologically-realistic properties (Bringsjord et al., 2005), and the digital modeling of human behavior in a variety of occupational and ergonomic settings (Lawson & Burnett, 2015).

No matter what the motivation for integrating cognitive architectures with virtual environments, all integration efforts rely on the existence of mechanisms that support the seamless inter-operation of the cognitive architecture with whatever system is used to implement the virtual environment. In situations where the virtual environment is a 3D environment similar to those encountered in contemporary video games, then the target of the integration solution will typically be a game engine. This can present challenges to integration, since the game engine and the cognitive architecture are typically systems that use different code bases and run in different processes[3]. In addition, both kinds of systems often place

[2] Unlike the solution described here, Best and Lebiere (2006) used ACT-R in conjunction with the Unreal game engine.

[3] The notion of a process, here, refers to a computer process, i.e., an instance of a computer program that is being executed.

significant demands on the computational resources of the host machine. This can make it difficult or impossible to run the cognitive architecture and the game engine on the same machine at the same time.

In the current chapter, we focus on an integration solution for a particular cognitive architecture (ACT-R) and a particular game engine (Unity). We first provide an overview of the ACT-R architecture and the Unity game engine. We describe the key features of both systems and discuss why they provide such a compelling target for integration. We then go on to describe the nature of the integration solution itself. We outline the extensions to the ACT-R architecture that enable ACT-R models to exchange information with Unity. We then go on to present the Unity components that enable virtual characters to be controlled or influenced by ACT-R models. Finally, we provide a concrete example of the use of the integration solution: we show how an ACT-R model can be used to control the behavior of a virtual robotic character that inhabits a Unity-based virtual environment.

THE ACT-R COGNITIVE ARCHITECTURE

ACT-R is one of a number of cognitive architectures that have been used for cognitive modeling (Anderson, 2007; Anderson et al., 2004)[4]. It is primarily a symbolic cognitive architecture in that it features the use of symbolic representations and explicit production rules; however, it also makes use of a number of sub-symbolic processes that contribute to various aspects of performance (Anderson et al., 2004).

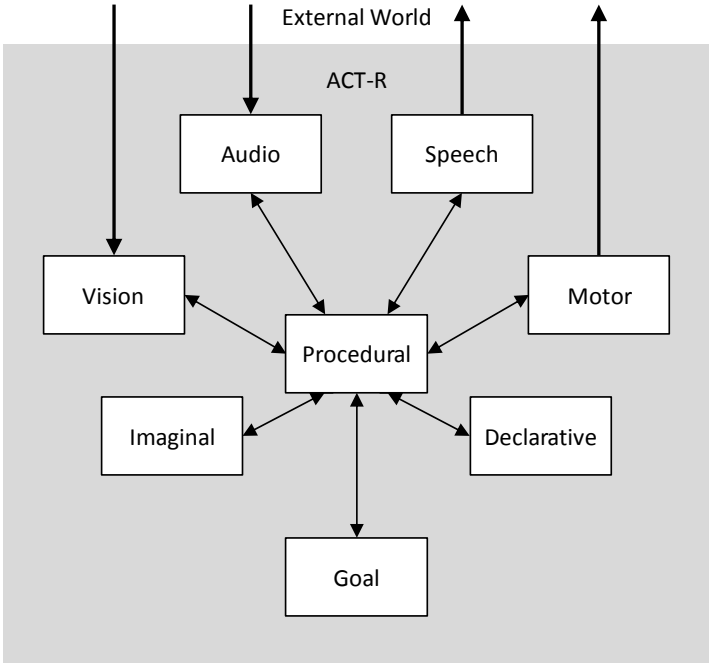

Figure 1: The core modules of the ACT-R v.6 cognitive architecture.

[4] ACT-R is, in fact, only one of a number of different cognitive architectures that can be used to support the design of intelligent virtual characters. The SOAR architecture is, of course, a popular alternative to ACT-R (e.g., Rickel & Johnson, 2000). In addition, a range of other, bespoke architectures, such as the RASCALS architecture (Bringsjord et al., 2005), have been the focus of efforts to combine cognitive architectures with virtual environments.

ACT-R consists of a number of modules (see Figure 1), each of which is devoted to processing a particular kind of information. Each module is associated with one or more capacity-constrained buffers that can contain a single item of information, called a 'chunk'[5]. The modules access and deposit information in the buffers, and coordination between the modules is achieved by a centralized production system module – the procedural module – that can respond to the contents of the buffers and change buffer contents (via the execution of production rules). Importantly, the procedural module can only respond to the contents of the buffers; it cannot participate in the internal encapsulated activity of the modules, although it can influence module-based processes.

As shown in Figure 1, there are eight core modules in the latest version of ACT-R[6]. These modules assume responsibility for the implementation of specific cognitive functions as part of an integrated cognitive system. The goal module, for example, is a specialized form of 'working memory' that maintains information relevant to task goals. It serves to contextualize the activity of other modules. Another important module is the declarative module. This module is responsible for the mnemonic encoding and retrieval of information. It stores information in the form of chunks, each of which is associated with an activation level that determines its probability of recall. The vision, audio, speech and motor modules function as points of perceptuo-motor contact between an ACT-R agent and the external environment. They provide support for the modeling of agent-world interactions.

The ACT-R architecture has been used to model human cognitive performance in a wide variety of experimental settings[7]. It has generated findings of predictive and explanatory relevance to hundreds of phenomena encountered in the cognitive psychology and human factors literature, and this has earned it a reputation as the cognitive architecture that is probably the "best grounded in the experimental research literature" (Morrison, 2003; p. 24). ACT-R has also been used to model behavior in a range of complex task settings, such as driver behavior (Salvucci, 2006) and collaborative problem-solving (Reitter & Lebiere, 2012). These features make ACT-R a compelling target for research that seeks to model cognitive performance and test assumptions regarding the characteristics of the human cognitive system. In addition, the various cognitive capabilities and features of the ACT-R architecture, especially those relating to learning and memory, make it an interesting choice for developers who are not so much interested in the modeling of human cognition as in the development of agents and systems that exhibit signs of human-level intelligence. This has given rise to studies that have attempted to use ACT-R to control the behavior of real-world robotic systems (Best & Lebiere, 2006; Kurup & Lebiere, 2012; Trafton et al., 2013). It is also the basis of recent work concerning the use of ACT-R models to control the behavior of virtual robots that inhabit virtual 3D environments (Smart & Sycara, 2015a, 2015c). Finally, the rich support that ACT-R provides for the computational modeling of cognitive processes supports its use in controlling the behavior of virtual synthetic agents that engage in interaction with human-controlled characters in the context of (e.g.) training simulations (Best & Lebiere, 2006).

---

[5] A 'chunk' in ACT-R is the basic unit of information over which rule-based processes operate.

[6] Although the eight core modules (and their associated buffers) tend to form the basis of most ACT-R models, cognitive modelers are not restricted to the use of these modules. New custom modules can be added to implement additional functionality, as required. The JNI module described by Hope et al. (2014) is one example of such a module (see also below).

[7] The ACT-R website (http://act-r.psy.cmu.edu/) provides access to a broad range of academic publications covering areas such as problem-solving, learning, language processing, decision making, and perceptual processing.

UNITY GAME ENGINE

Unity[8] is a game engine, developed by Unity Technologies, that has been used to create a broad range of interactive 2D and 3D virtual environments. Although, it is most commonly associated with the development of video games, it has also been used to develop other kinds of systems. These include simulation capabilities and visualization environments for use in educational (Christel et al., 2012), medical (Rizzo et al., 2014) and engineering (Mattingly et al., 2012) applications.

The Unity game engine forms part of what is sometimes referred to as the Unity game creation system. This includes, in addition to the game engine, an Integrated Development Environment (IDE) (see Figure 2) and an object-oriented scripting framework that is available in three languages: Boo (a Unity language that resembles Python), JavaScript and C#. These languages can be used to create custom code components that derive from a common class, called `MonoBehaviour`, which provides access to overridable methods that are invoked at different stages of the game's execution, e.g., during the game's update loop. Scripts that contain classes derived from `MonoBehaviour` can be attached to game objects in order to control their behavior at runtime. They can also be used to take actions in response to user input, implement custom user interface overlays, store and load game data, and implement general game mechanics. Importantly, a script component that inherits from `MonoBehaviour` can be attached to several different game objects in order to implement common functionalities, and multiple scripts can be attached to the same object in order to create combined functionalities. Using such features, it is possible to create complex control mechanisms for characters within a game. In the example described later in this chapter, for example, a custom `MonoBehaviour` component was developed to respond to commands from an ACT-R model in order to control the behavior of a virtual robot.

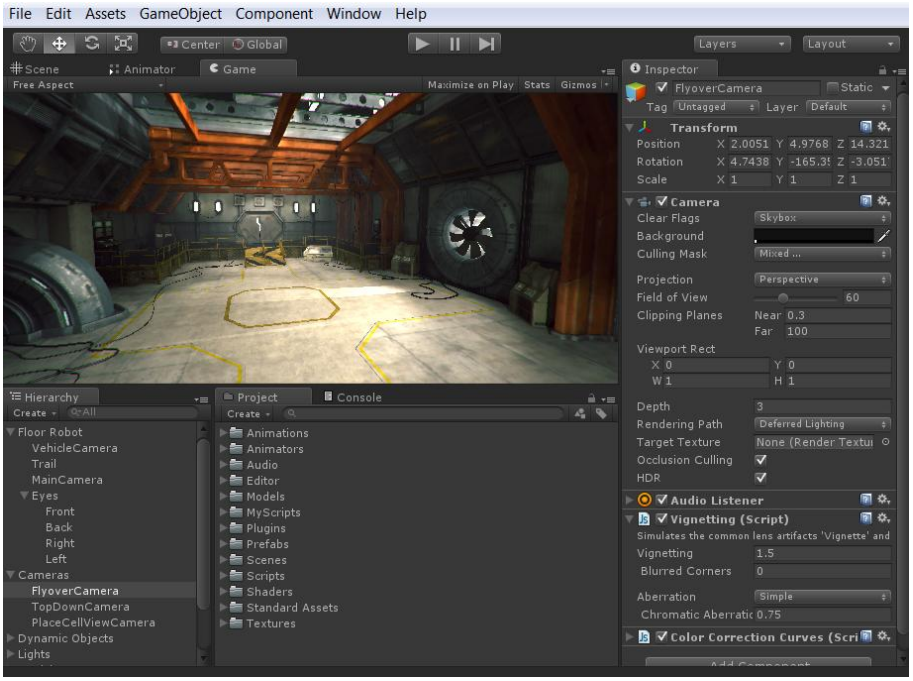

Figure 2: View of the Unity IDE showing the Game, Hierarchy, Project and Inspector windows.

[8] See http://unity3d.com/.

The Unity IDE (see Figure 2) provides extensive support for the development of 2D and 3D virtual environments. It includes a number of windows, such as the Scene, Project, Hierarchy, Game and Inspector windows. These enable a developer to visualize the current scene (Scene window); adjust the properties of game objects (Inspector window); manage game assets (Project window), such as texture, audio, code and model assets; and play test the current state of the game (Game window). The IDE also integrates seamlessly with external code editors, such as MonoDevelop[9] or Microsoft Visual Studio.

A key function of the Unity IDE is to provide easy access to a number of built-in components that form part of the Unity game engine. One component of particular importance is the `Camera` component. This provides a view of a 3D or 2D scene from a particular vantage point. While `Camera` components typically render their view of the scene to the main game window during game play, they can also be used to render a scene to a special type of 2D image asset, called a `RenderTexture`. This can then be processed using standard 2D graphic processing routines in order to retrieve information about luminance levels in the red, green and blue (RGB) color channels of the image's pixel data. In the context of our own work, we rely on this technique to implement virtual 'eyes' that generate a series of 2D images as an agent moves through a 3D landscape. Using image processing techniques, it is possible to extract simple visual features from these images in order to implement various forms of visuo-motor control.

One of the features that makes Unity a compelling target for integration efforts is its popularity. Unity is undoubtedly one of the most popular game creation systems currently in use. Underlying this popularity, particularly with the indie game community, is the support it provides for multi-platform development. Unity can thus be used to create applications that are deployable to the Web, game consoles (e.g., Xbox, Wii, and PS3), mobile devices (e.g., iPhone, iPad, and Android devices), and personal computers (e.g., Windows, Mac OS and Linux). In addition to its deployment features, the Unity game engine boasts extensive technical support, a community-based asset store, an impressive list of game development features, and compatibility with Microsoft's .NET Framework. Moreover, the latest version of the game engine (Version 5) is free to use for research and development purposes[10].

INTEGRATING ACT-R WITH UNITY

Integration Challenges

A number of challenges confront the attempt to develop an integration solution for ACT-R and Unity. One of the issues to address concerns the fact that Unity and ACT-R are implemented using different languages: Unity is implemented in C++, while ACT-R is implemented in Lisp. This complicates the attempt to straightforwardly embed ACT-R within Unity.

A second challenge relates to performance. Many 3D environments depend on computational operations that are highly processor-intensive (e.g., the operations associated with graphics rendering or physics simulation). Similarly, the execution of ACT-R models can consume significant computational resources, particularly when multiple ACT-R models are used to perform multi-agent simulations. If the aim of an

---

[9] http://www.monodevelop.com/

[10] Although we focus on the Unity game engine in the current chapter, the integration solution outlined here could be adapted to work with other game engines, such as Epic's Unreal game engine. All that is required, in this case, are components that replicate the functionality of the ACT-R Unity Interface Framework components (e.g., the `ACTRAgent` component).

integration solution is to use ACT-R in conjunction with a virtual environment and have both cognitive and virtual world simulations rely on the same set of processing resources, steps must be taken to ensure that the complexity of the ACT-R model and virtual world simulation are within the performance capabilities of the target deployment platform. This is particularly so if the aim is to run simulations in real-time.

Finally, it should be noted that ACT-R is typically intended to be used as a stand-alone system. By default, the ACT-R system includes very little support for inter-operation with external systems and applications. This underscores the need to implement extensions to ACT-R that support flexible modes of inter-process communication and systemic integration.

A Network-Based Approach to Integration

One approach to integrating ACT-R with Unity is based on a networking solution that supports bidirectional modes of communication between one or more ACT-R models and a single Unity-based virtual environment. This approach has the advantage of dealing with the aforementioned challenges[11]. The issue regarding different code bases is resolved by enabling ACT-R and Unity to run in different processes and communicate using standard network protocols and messaging systems. In addition, the issue of performance overheads can be tackled by having ACT-R run on a different machine to that hosting the Unity-based environment. This enables the cognitive and virtual world simulations to be handled by dedicated processors that run in parallel and synchronize their activity via the exchange of messages. It should also be clear that the strategy of running ACT-R and Unity on different machines has the added bonus of addressing issues associated with compilation and deployment. As mentioned above, the ability to deploy Unity games and applications to multiple platforms is one of the features that motivates its use above other game engines. Although the ACT-R system can be compiled to run on systems, such as Mac OS, Linux and Windows, the use of a networking solution to integrate ACT-R with Unity simplifies the deployment strategy: a developer can deploy the Unity-based application to a target platform using the features available within the Unity editor and then rely on network connections to communicate with an instance of ACT-R that runs on another (perhaps dedicated) machine within the local network environment or the Internet.

The particular integration solution described here relies on the use of an ACT-R component developed by Hope et al. (2014) called the JSON Network Interface, or JNI. JSON, in this case, is an acronym that stands for JavaScript Object Notation Format. It is a data interchange format that is commonly encountered in the context of Web-based communications. By using the JNI, it is possible to run ACT-R models that communicate with a target external environment by posting JSON messages. Although, in our case, the external environment of interest is an application or simulation that runs on top of the Unity game engine, it is important to recognize that many different systems could play a similar role. The notion of an external environment could thus be applied to a conventional Windows desktop application that communicates with ACT-R as part of an attempt to evaluate the cognitive ergonomic design of different graphical user interface designs. It could also apply to a real-world robotic platform, where ACT-R is used to process sensor information and control the robot's effector systems. What is required, in all these

[11] Note that this solution is preferable to an integration solution that attempts to re-implement ACT-R within the native programmatic environment of the game engine. One issue to consider, here, is that ACT-R is subject to periodic updates and revisions. This makes the maintenance of ported code somewhat problematic. In addition, the scale and complexity of the ACT-R system complicates the attempt to duplicate ACT-R functionality within the native environment of a particular game engine.

cases, is a means by which messages sent by ACT-R can be interpreted by the target external environment. Typically, this involves the development of an Application Programming Interface (API) for the target environment. In the present case, we have developed an API for Unity that supports communication with ACT-R via the JNI. We refer to this as the ACT-R Unity Interface (ACT-R UI) Framework.

JSON Network Interface

In order to support the bidirectional exchange of information between ACT-R models and a range of external environments, Hope et al. (2014) advocate the use of a network-based approach to integration, in which an external system plays the role of a server and individual ACT-R models play the role of clients. Communication between the two systems is then established via a series of client-server interactions mediated by TCP/IP socket connections.

Hope et al.'s (2014) solution is encapsulated in a custom ACT-R module – the JSON Network Interface, or JNI. This module can be installed alongside the core modules that form part of the default ACT-R architecture (see Figure 1). Each JNI module implements the functionality needed to establish connections and interact with the external environment. It comes with parameters that identify the IP address of the external environment as well as the number of the TCP port that the external environment is listening on. This is clearly important in terms of enabling a specific ACT-R model to connect to the environment during the initialization process. The JNI module communicates with the external environment using a set of commands that form part of the JNI API. These commands are transmitted as JSON-formatted messages that either inform the external environment about the state of model execution or which instruct the environment to take particular actions based on model outputs. For example, the `model-run` command is posted by the JNI whenever an ACT-R model has started running (thereby providing state information); the `keypress` command, in contrast, is posted whenever the ACT-R model has decided to initiate a keypress action via the motor module. In all cases, the messages sent to the external environment include the name of the ACT-R model from which the message originates, the name of the command that is being issued, and any data that is relevant to the interpretation or processing of the command named in the message. The inclusion of the name of the originating model in the message is important here because each ACT-R model has its own instance of the JNI module. This means that multiple ACT-R modules can communicate with the same external environment (e.g., Unity) either as part of a multiplayer online game (perhaps featuring a combination of human-controlled and ACT-R-controlled characters) or as part of an experimental simulation into socially-situated or socially-distributed cognition (see Smart & Sycara, 2015b). The ability to identify the originating ACT-R model, in this case, enables the external environment to delegate the processing of received messages to specific virtual agents within the environment.

In addition to the commands posted by an ACT-R model (known as ‘module commands’), the JNI API includes commands that an ACT-R model expects to receive from the external environment. These commands, known as ‘environment commands’, provide information about the state of the external environment, and they are typically used to update the content of ACT-R’s perceptual modules. Figure 3, for example, shows the structure of a sample `display-new` environment command that is used to update the content of the vision module of an ACT-R model called ‘myModel’. Other environment commands enable the external environment to add items to declarative memory, trigger the presentation of rewards and present auditory stimuli.

```
{"model": "myModel",
  "method": "update-display",
  "params": {
    "visual-location-chunks": [
      {"isa": "visual-location",
        "slots": {"screen-x": 100,
                  "screen-y": 200}},
      {"isa": "visual-location",
        "slots": {"screen-x": 300,
                  "screen-y": 400}}],
    "visual-object-chunks": [
      [{"isa": "visual-object",
        "slots": {"value": "hello"}},
      {"isa": "visual-object",
        "slots": {"value": ":world"}}]],
  "clear": true}}
```

Figure 3: An example environment command (formatted using JSON) that is used to update the contents of an ACT-R model's vision module.

The set of module and environment commands that comprise the standard JNI API provide the basis for forms of communication in which an ACT-R model is interacting with a particular kind of environment, namely a display screen. Obviously, this is unlike the kind of environment that is encountered in the context of a video game, where a virtual character is required to respond to the perceptual features of a 3D scene[12]. Fortunately, the set of commands that can be exchanged via the JNI module is easy to extend. In terms of extending the range of module commands that are available, the JNI API provides access to a set of Lisp functions that can be called upon to create and format the new command messages. Extending the range of environment commands (commands originating from the external environment) can be accomplished by relying on a particular environment command called `trigger-event`. This command enables the external environment to trigger the execution of named Lisp functions that execute in the context of the ACT-R environment. Such functions can be used to great effect in situations where the kind of sensory information that needs to be handled by ACT-R is not easily accommodated by the core perceptual modules of the ACT-R architecture. In the case of our own work with virtual robots, for example, we often include tactile information in the package of sensor information that is posted to ACT-R. The core ACT-R architecture has no module that can handle this kind of information, so it is necessary to implement a custom module and rely on the JNI `trigger-event` command to update the contents of the buffers associated with the new module. Figure 4 provides an example of such a function. It shows a Lisp function being used to update a sensor buffer in response to the posting of a `trigger-event` command by an external environment.

[12] Although it might be appropriate for situations in which the ACT-R model is intended to model human player behavior and the display screen corresponds to the human's view of the game.

```
(defun update-tactile-sensor (left-whisker  right-whisker)
  "Create a new chunk representing the state of the robots left and right
  whiskers and place it in the 'robot-sensor' buffer, if the buffer is
  currently empty."
  (let ((buffer 'robot-sensor)
    (left-whisker-val (intern (string-upcase left-whisker)))
    (right-whisker-val (intern (string-upcase right-whisker)))
    (chunk (car (define-chunks (isa robot-sensor-state)))))
  (if (query-buffer buffer '((buffer . empty)))
    (progn
      (set-chunk-slot-value-fct chunk 'left-whisker left-whisker-val)
      (set-chunk-slot-value-fct chunk 'right-whisker right-whisker-val)
      (schedule-event-relative 0 (lambda nil (set-buffer-chunk buffer chunk))))))
)
```

Figure 4: Example of a Lisp function that is used to update the contents of a buffer (called robot-sensor) in response to the receipt of a trigger-event command from an external environment.

ACT-R Unity Interface Framework

Although the JNI provides the basis for ACT-R integration efforts, additional work needs to be done in order to enable the external environment to communicate with ACT-R. In the case of Unity, for example, it is necessary to develop components that can handle incoming connection requests from ACT-R models, interpret the JSON messages received from ACT-R models and implement whatever commands are contained in the messages. Furthermore, as mentioned above, it is often necessary to extend the functionality of the ACT-R environment in ways that enable an ACT-R model to function within the specific sensorimotor niche provided by the external environment. Often this involves the implementation of additional ACT-R modules that can handle specific kinds of sensory or motor information.

In the context of our own work, we have developed a framework, called the ACT-R Unity Interface (ACT-R UI) Framework, which enables ACT-R models to communicate with applications that run on top of the Unity game engine (Smart et al., forthcoming). The ACT-R UI Framework consists of a number of components, all of which are implemented as Unity-compatible C# scripts. The main components of the framework are the following[13]:

- **ACTRNetworkInterface:** The main function of the `ACTRNetworkInterface` component is to handle connection requests from ACT-R models. It also implements the functionality to post messages to ACT-R clients on the network. The component relies on the native support that Unity provides for .NET socket connections.

- **ACTRCommand:** The `ACTRCommand` class is the base class for all the commands that are recognized by ACT-R and Unity in the context of a particular integration solution. There are two main subclasses of the `ACTRCommand` class: `ModuleCommand` and `EnvironmentCommand`. As is suggested by these names, these abstract classes are intended to represent the two categories of commands that form part of the JNI API. Subclasses of these top-level classes represent all the commands that are included in the JNI API. Additional subclasses can be created as required to extend the range of messages that Unity and ACT-R are able to process.

[13] Further information about the ACT-R UI Framework can be found in Smart et al. (forthcoming).

- **ACTRMessageInterface:** This is a component that inherits from the Unity `MonoBehaviour` class. Its primary function is to engage in the preliminary processing of ACT-R messages. It processes the raw JSON content of the message and attempts to create appropriate instances of the `ACTRCommand` class based on the content of the message. Once created, these commands are posted to the relevant `ACTRAgent` component (see below) according to the name of the originating ACT-R model (which is included in the message). The `ACTRMessageInterface` component contains a user-editable field (visible in Unity's Inspector window) that specifies the port number that ACT-R clients should use to connect to Unity. The component also references an instance of the `ACTRNetworkInterface` class, and it initializes this instance whenever the game starts. This results in Unity listening for incoming connection requests from whatever ACT-R models are running on the network.

- **ACTRAgent:** This `MonoBehaviour` component represents the entity in the virtual environment that is controlled by a particular ACT-R model. It is typically attached to a game object representing a non-player character, such as a virtual robot or humanoid avatar. It exposes a property that enables a developer to specify the name of the ACT-R model that will control the entity. At runtime, the main function of the `ACTRAgent` component is to engage in sensor processing and implement the motor instructions received from ACT-R. In general, `ACTRAgent` components can be configured to periodically poll available sensors and post information back to ACT-R as part of what is called the sensor processing cycle. They can also respond to instructions from ACT-R (e.g., to move forward or turn to the right). Unlike the other components that form part of the ACT-R UI Framework, the `ACTRAgent` component is not intended to be used in specific simulations. This is because the nature of the sensorimotor processing routines for ACT-R-controlled agents are likely to be very different depending on the kind of simulation that is being run. A simulation involving virtual robots, for example, is likely to feature a different set of sensor and effector components compared to a simulation involving humanoid characters. For this reason, most simulations will need to create components that derive from the `ACTRAgent` component in order to adapt the capabilities of the virtual character to the specific sensorimotor niche they occupy within the virtual environment.

Using the components of the ACT-R UI Framework, it is possible to create a network-based integration solution in which one or more ACT-R models control the behavior of virtual characters that are embedded within a virtual environment. Given the network-based nature of the integration solution, it is clearly possible to distribute the computational burden associated with the execution of both cognitive models and the virtual environment: multiple cognitive models, each representing a distinct cognitive agent, can be run on different machines and communicate with a Unity-based application that itself runs on a dedicated machine[14]. This solution can be used to integrate ACT-R cognitive models into online multiplayer games that feature the participation of human-controlled characters.

[14] The simulation described in the current chapter focuses on a single ACT-R cognitive model that controls a single virtual character; nevertheless, the extension to the multi-model case (i.e., where multiple ACT-R models control the behaviour of multiple virtual characters) is relatively straightforward. See Smart et al. (forthcoming) for practical steps on how to implement multi-model solutions using the ACT-R UI Framework.

CASE STUDY: MAZE NAVIGATION USING A VIRTUAL ROBOT

In order to demonstrate the use of the ACT-R UI Framework, we describe a simulation capability that focuses on the spatial cognitive capabilities of a virtual robot. The simulation features a robot that is embedded in a virtual maze environment that was created using the Unity IDE. The simulation is organized into two phases. In an initial exploratory phase, the robot is required to explore the maze at random and learn about its structure. Then, in a subsequent navigation phase, a goal location is specified and the robot must use its previous experience of exploring the maze in order to plan a route and navigate to the target location. A video showing the performance of the robot across both phases of the simulation is available for viewing on the YouTube website[15].

Virtual Environment Design

The environment used for the maze navigation task is a simple virtual maze created from a combination of geometric shapes, such as blocks and cylinders. The design of the maze is based on that described by Barrera and Weitzenfeld (2007) who used a similar maze as part of their effort to test bio-inspired spatial cognitive capabilities in a real-world robot. The maze consists of a number of vertically- and horizontally-aligned corridors that are shaped like the letter 'H'. An additional vertically-aligned corridor serves as a common point of departure for the robot during the exploration and navigation phases of the simulation (see Figure 5).

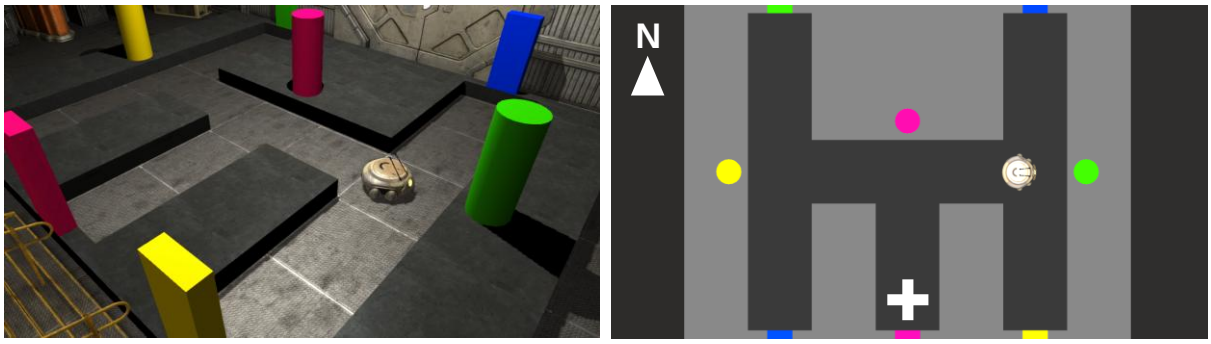

Figure 5: View of the 'H' Maze from a first-person perspective (left) and from a top-down perspective (right). The robot is located on the right hand side of the maze in both images. The camera that renders the top-down view of the maze (on the right) has been configured to simplify the rendering output. This makes it easier to visualize and analyze simulation results. The white cross in the right hand image represents the starting location for the robot on all training and testing trials. The compass indicator (again in the right-hand image) shows the direction of 'north'.

A number of brightly colored blocks and cylinders were placed around the walls of the maze to function as visual landmarks. These objects show up as colored patches in the images that are rendered by each of the robot's eyes, and they can thus be used by the robot to identify its location within the maze.

[15] See http://youtu.be/zolWEO8PRQg

Robot Design

The virtual robot used in the simulation is a disc-shaped 3D model that is capable of linear and rotational movements. The robot comes equipped with a number of virtual sensors that are capable of processing visual, tactile and directional information. These sensors correspond to the robot's eyes, whiskers, and onboard compass, respectively. The eyes are represented by Unity `Camera` components that are positioned around the edge of the robot and oriented slightly upwards at an angle of 15 degrees. For convenience, these cameras are referred to as 'eye cameras'. There are four eye cameras in total. They capture views from the left, right, forward and backward directions of the robot. The output of the eye cameras is captured as a `RenderTexture` asset as described above and is subjected to lightweight image processing techniques in order to extract visual features. The primary aim of the robot's visual processing routines, in the current simulation, is to detect the brightly colored objects arrayed around the walls of the maze. This is accomplished by matching the luminance levels of image pixels in the red, green and blue color channels to a number of target colors corresponding to those of the visual landmarks (i.e., the pink, blue, green and yellow objects). The target colors in this case are specified by the user at design time and can be set to any color. The sensitivity of the robot's eyes is governed by two values: the 'tolerance' and 'threshold' values. The tolerance value represents the range of luminance levels in each color channel that is recognized as a match to the target luminance level. The threshold value, in contrast, specifies the minimum number of matching pixels that must be counted in order for the eye to signal the detection of a particular color. For the purposes of the current study, the tolerance value was set to a value of 0.01 and the threshold value was set to a value of 1500. In addition, each retina was sized to 200 x 200 pixels to give a total of 40,000 pixels per eye camera on each render cycle.

In addition to eyes, the robot comes equipped with 'whiskers'. These function as tactile sensors. Similar to the eye cameras, there are four whiskers situated around the robot's body, and these project outwards in the forward, backward, left and right directions. The whiskers respond to contact with the walls of the maze. This tactile information, in combination with the visual input, serves to identify specific locations in the maze. It also provides affordances for action-related decisions, helping to inform the robot when it needs to turn and what directions it can move in. From an implementation perspective, the whiskers rely on the use of ray casting techniques: each time the robot is required to report sensory information to ACT-R, rays are projected from the vehicle's body and any collisions with the walls of the maze are recorded.

The final sensor used by the robot is a compass. The robot is capable of detecting which direction it is facing based on the compass reading, which is based on the rotation of the vehicle's transform in the world coordinate system. A rotation of zero degrees, for example, corresponds to a heading value of 'NORTH'.

For the purposes of this work, the linear movement of the robot was restricted to the north, south, east and west directions: these are, in fact, the only directions that are needed to fully explore the 'H' Maze environment, and restricting movement in this way reduces the complexity of the simulation. The robot is also capable of making 360 degree turns. This enables the robot to turn around when it reaches the end of one of the arms of the maze. Turning movements are implemented by rotating the robot's transform based on the commands received from ACT-R; linear movements, in contrast, are implemented by specifying the velocity of the robot's `Rigidbody` component, a component that enables the robot to participate in the physics calculations made by Unity's physics engine.

Cognitive Model

The cognitive modeling effort involved the development of an ACT-R model that could support the initial exploration of the maze and also enable the robot to navigate to target locations. The requirements of the model were the following:

- **Motor Control:** The model needs to issue motor instructions to the robot in response to sensory information in order to orient and move the robot within the maze.

- **Maze Learning:** The model needs to detect novel locations within the maze and memorize the sensory information associated with these locations.

- **Route Planning:** The model needs to use the memorized locations in order to construct a route to a target location.

- **Maze Navigation:** The model needs to use route-related information in conjunction with sensory feedback in order to monitor its progress towards a goal location.

The ACT-R model developed to support the target behavior consists of 110 productions in addition to ancillary functions that control the communication with Unity. A key function of the model is to memorize spatial locations that are individuated with respect to their sensory context (i.e., unique combinations of visual and tactile information). These locations are referred to as 'place fields' in the context of the cognitive model. Each place field is created as a chunk in declarative memory and retrieval operations against declarative memory are subsequently used to recall the information encoded by the place field as the robot moves through the maze. The collection of place fields constitutes the robot's 'cognitive map' of the maze. This map is structured as a directed graph in which the place fields act as nodes and the connections between the nodes are established based on the directional information that is recorded by the robot as it explores the maze[16]. Any two place fields that are created in succession will be connected via ACT-R chunks that record the direction the robot was heading in when the connection was made. For example, if the robot creates a place field (PF1) at the start of the simulation and then creates a second place field (PF2) while heading north from the starting location, a connection is established between PF1 and PF2 that records PF1 as the source of the connection, PF2 as the target of the connection and 'NORTH' as the direction of the connection. The collection of place fields and the connections that join them thus serve to record information about the topological relationships between maze locations based on a combination of sensory experiences and the robot's own movements.

The productions of the ACT-R model were used to realize the motor control, maze learning and maze navigation functions mentioned above; the route planning function, however, was implemented using separate Lisp routines. In order to plan a route, the robot first needs to be given a target location. This was specified at the beginning of trials that tested navigational performance. The robot then needs to identify its current location in the maze. This was achieved by the model comparing current sensory information with all the place fields stored in memory. The result (assuming that the robot's initial navigation of the maze is complete) is a place field that corresponds to the robot's actual position in the maze. Finally, the robot needs to compute a sequence of place fields that encode the path from the start location to the target location. This is achieved using a spreading activation solution that operates over the network of place fields contained in memory. The chain of activated place fields from the start location to the target

[16] To simplify the structure of the map, bidirectional connections between adjacent nodes were not permitted.

location specifies the sequence of place fields (identified by combinations of sensory information) that must be detected by the robot as it navigates to the target. In addition, the connections between adjacent place fields along the computed route contain information about the direction the robot needs to move in as each place field is encountered. For example, if the connection between the first and second place fields in the route has an associated value of 'NORTH' and the robot is currently facing north, then the model can simply instruct the robot to move forward. If the robot is facing south, then the robot needs to perform a 180 degree turn before moving forward and the model thus needs to issue instructions for the robot to engage in a turning maneuver.

Sensorimotor Interface Module

The modules that form part of the core ACT-R architecture include modules that deal with the processing of sensory and motor information. The vision module, for example, is designed to handle requests about the presence, location and properties of visual objects within the visual field. These modules were not used as part of the current research effort. Instead, a custom module was used as a single point of sensorimotor contact between ACT-R and Unity. The module was designed to handle all sensory information received from Unity and to also issue motor instructions that controlled the behavior of the virtual robot. One reason why the existing vision module was not used to process visual information relates to the fact that the module was not intended to support the processing of low-level sensor data. While it is not inconceivable that the vision module could be used to handle the kind of visual input seen in the case of the current simulations, we deemed it more straightforward to implement a custom module. Another reason motivating the choice of a custom module to handle sensorimotor processing concerns the fact ACT-R is not designed to accommodate perceptuo-motor processes that derive from the kind of sensory and motor systems seen in the case of virtual robots. The ACT-R motor module, for example, was not used for motor output in the present case because it was originally designed to implement the sort of actions (e.g., keyboard presses and mouse movements) that a human subject might perform while seated in front of a desktop computer. Obviously, the kind of motor control capabilities required in the case of a non-humanoid virtual robot demand a set of specialized motor commands that are best handled by a custom module.

The custom module itself features two buffers: the 'robot-sensor' and 'robot-motor' buffers. The robot-sensor buffer contains information about the current sensory state of the robot within the virtual environment. The content of the buffer is populated automatically based on the messages received from Unity. In particular, when Unity posts a message containing updated sensory information, the message triggers the execution of a Lisp function that creates an ACT-R chunk containing all the sensor information contained in the message. This chunk is subsequently placed in the robot-sensor buffer.

The robot-motor buffer processes requests relating to the movement of the robot within the virtual environment. It accepts information (in the form of ACT-R chunks) about the desired state of a robot's motor system. When a chunk is placed into the buffer (as a result of rule execution), the robot module processes the chunk to extract the relevant motor information and then dispatches a JSON-formatted message containing the motor information to the JNI module for transmission to Unity. Once this message is received by Unity, it is handled by the ACT-R UI Framework: the relevant robot is identified based on the name of the ACT-R model that posted the message, and the message is then sent to the relevant `ACTRAgent` component for further processing. During the course of each update cycle of the game engine, the `ACTRAgent` will attempt to implement whatever motor instructions it receives from ACT-R. Typically, these instructions result in the robot moving forward at a particular velocity or turning to face a particular direction.

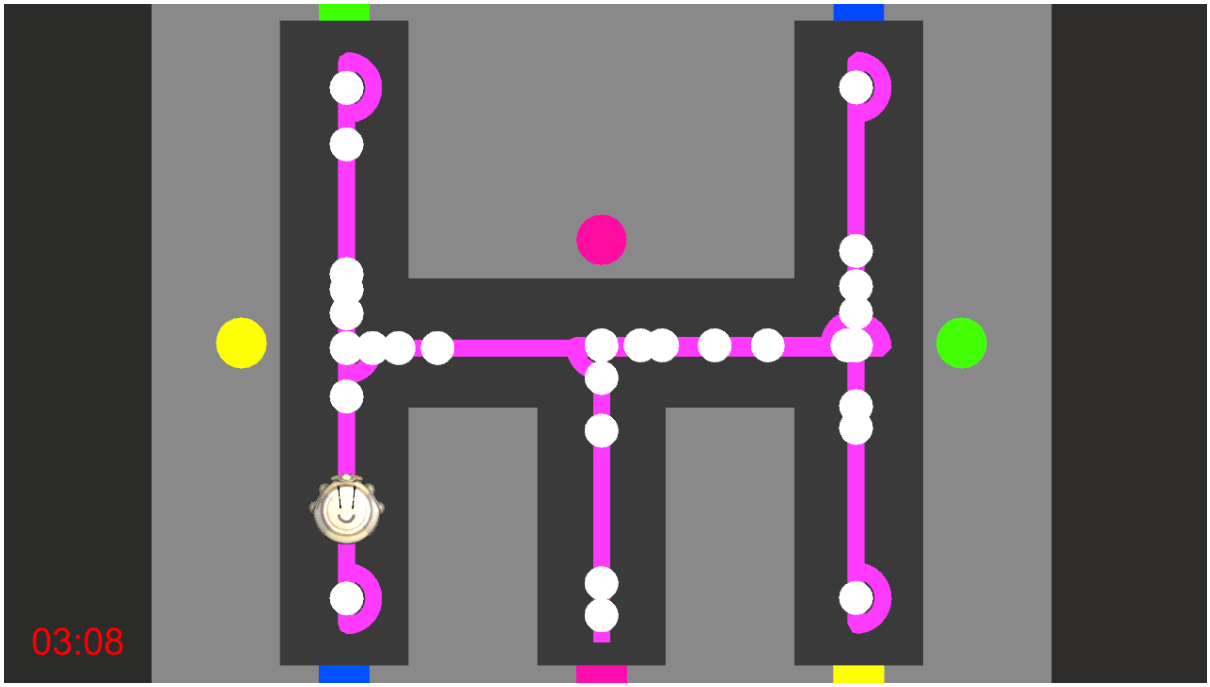

Figure 6: The results of the exploration phase of the experiment. The white circles symbolize the location of place fields that are memorized by the robot as it explores the maze. The magenta trail shows the path taken by the robot.

Phase I: Maze Exploration

In order to test the integrity of the integration solution and the performance of the cognitive model, a simple experiment was performed. The experiment consisted of two phases: an exploration (learning) phase and a navigation (testing) phase. In the exploration phase, the robot was placed at the start location (i.e., the base of the middle vertically-aligned arm), and it was then allowed to wander around the maze and learn about its structure. This resulted in the robot forming a cognitive map of the environment. Once the robot had explored all of the maze, the exploration phase was terminated and the cognitive map was saved to disk for later use.

In the subsequent navigation phase, the previously created cognitive map was loaded into declarative memory, and the robot was given a series of target locations to navigate towards. These target locations were situated at the ends of the vertically-aligned corridors that made up the long arms of the maze. As was the case for the exploration phase, the starting location of the robot for each trial of the navigation phase was the base of the smaller vertically-oriented arm.

The structure of the cognitive map formed by the robot during the exploration phase is shown in Figure 6. The white circles in this figure indicate the position of the place fields that were formed by the robot as it moved around the maze. The magenta trail represents the path of the robot. As can be seen from Figure 6, the robot explored the entire maze in a time of 3 minutes and 8 seconds. A total of 29 place fields were created in memory during the exploration phase. These served as the nodes in the graph that made up the cognitive map. The nodes were connected together into a single component network that consisted of 33 connections.

Phase II: Maze Navigation

The results of the maze navigation phase are shown in Figure 7. This figure shows the path taken by the robot as it navigated to each of the four target locations. It also shows the time taken by the robot to reach each of the target locations. For example, it took 1 minute and 4 seconds to reach the goal located at the top left of the maze. In all trials, the robot was able to successfully reach the target location – the robot identified its location at the start of the trial, plotted a route to the target based on the structure of the cognitive map, and then used this route to guide its movements towards the goal.

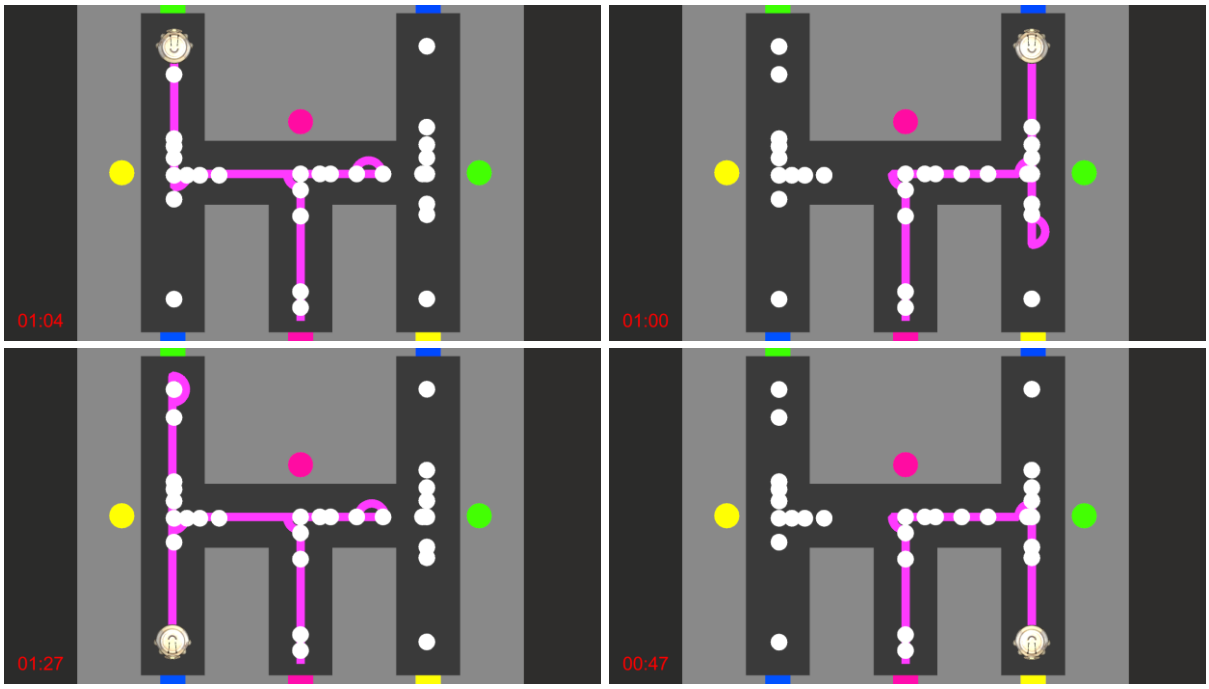

Figure 7: The results of the maze navigation phase of the experiment. The robot successfully navigated to each of the goal locations. The magenta trail indicates the path taken by the robot as it navigated towards the goal location.

In spite of the ability of the robot to ultimately reach the target locations, Figure 7 shows some inefficiencies in the navigational decisions made by the robot. In particular, in three out of the four trials, the robot made an unexpected detour at some point along the route. For example, in the case of the goal at the top left of the maze (see top left image in Figure 7), the robot initially made a turn towards the right when it reached the horizontal corridor of the maze. In fact, the correct turn is towards the left, and the robot seemed to realize it's mistake when it subsequently made a 180 degree turn a few seconds later. The reason for this navigational anomaly is due to the fact that not all locations in the maze are uniquely identified by combinations of visual and tactile input. In Figure 8, for example, we can see that locations 2 and 4 (i.e., L2 and L4) are associated with exactly the same combinations of sensory input (a green cylinder to the east, a yellow cylinder to the west and walls to the north and south). Imagine that a robot has already encoded a link between L1 and L2 so that it knows that L2 is located to the east of L1. If the robot now heads west towards L4, it will pass through L1 *en route* to L4. As the robot encounters the stimuli associated with L1 it will recognize L1 as its current location. When it subsequently encounters L4, it will 'believe' (based on the combination of visual and tactile sensory information that is receives) that it is actually at location L2. The existence of a place field representing L2 in memory will prevent another (identical) place field being created to represent L4. In addition, the fact that the cognitive map

used by the robot is a directed graph with unidirectional connections prevents the creation of a link encoding the otherwise confusing state-of-affairs that L2 is located both to the east and the west of L1. The problem, however, is that the failure to represent L4 with a separate place field results in the robot creating a link between the place fields representing L2 and L5 when it eventually passes L4 and reaches the vicinity of L5. The source of the robot's navigational confusion during testing is now revealed. When a route is planned to a goal location that is to the west of L4, the robot constructs a route that consists of place fields representing L1, L2 and L5. However, the cognitive map tells it that L2 is located to the east of L1, and so that is the direction it heads in when it reaches L1. When it subsequently arrives at L2, it realizes that L5 is located to the west of L2 and that it is therefore heading in the wrong direction. Hence the reason for the sudden about turn.

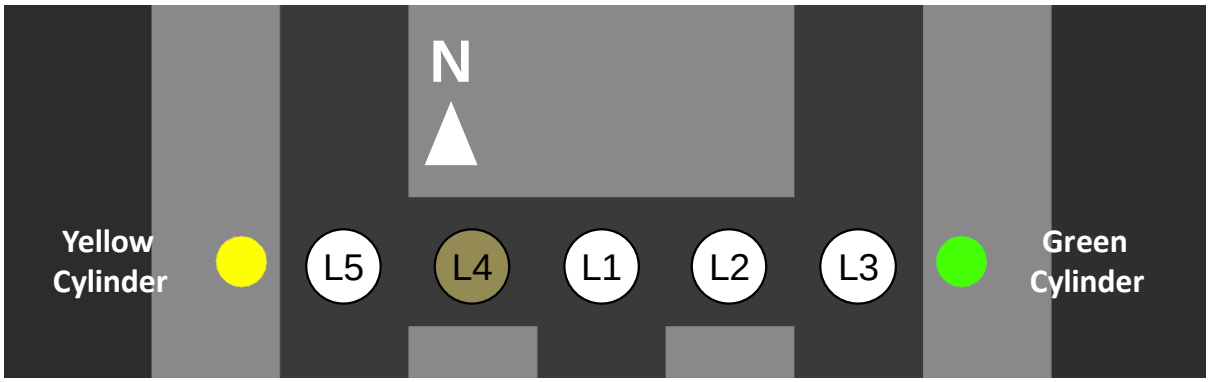

Figure 8: A series of locations in the 'H' Maze, only some of which are uniquely identified by combinations of tactile and visual input. Based on the sensory capabilities of the robot, the locations indicated by L2 and L4 are perceived as identical. This leads to the robot failing to create a separate place field corresponding to L4 (only locations indicated by white circles are associated with place fields). The result is that when a route is planned to L5, the robot will erroneously turn east and head towards L2 when it arrives at L1.

There are a couple of ways in which this peculiar form of navigational behavior could be remedied. Firstly, the spatial reasoning capabilities of the robot could be extended in order to support a sense of the relative positions of distinct locations. For example, with respect to Figure 8, the robot could reason that if it encountered a location that was identical to L2 having already passed through L1 while heading in a westerly direction, and that L2 was to the east of L1, then the new location could not possibly be the same as L2. It could thus create a separate place field to represent L4, albeit with the same configuration of sensory information as that associated with the place field representing L2. An alternative strategy would be to improve the discriminative capabilities of the robot when it comes to an assessment of the stimuli associated with particular locations. The visual system of the robot could thus be enhanced to detect the relative size of visual landmarks within the visual field. This would enable a robot to discriminate between L2 and L4 in Figure 8 on the basis of the fact that the yellow cylinder would appear larger at L4 than it would at L2 (the reverse being true for the green cylinder).

CONCLUSION

ACT-R and Unity are the two most widely used systems in their respective domains of use. ACT-R is one of the most widely used cognitive architectures, with a long history of use within the human factors, cognitive psychology and (more recently) robotics communities. Similarly, Unity is a very popular game engine that has been used to develop a broad range of visualizations, applications, simulations and, of course, video games. Given the range of potential applications and research opportunities enabled by the

ability to merge the capabilities of cognitive architectures with virtual environments, it is surprising that no discernible attempt has been made to develop an integration solution for ACT-R and Unity. As far as we are aware, the solution presented in this chapter is the only attempt thus far to integrate the two systems.

The integration solution described here draws on previous work by Hope et al. (2014); however, it also extends that body of existing work by enabling ACT-R to inter-operate specifically with the Unity game engine. The solution is referred to as the ACT-R UI Framework: a set of components that can be used by Unity developers to link virtual game characters to distinct ACT-R models. The example implementation we have used to demonstrate the integration solution is a maze navigation problem in which a virtual robot is required to learn about the spatial structure of a maze and navigate to specific goal locations. This example is intended to showcase some of the features of the integration solution in a task context that exploits at least some of the cognitive features of the ACT-R architecture. Extensions of this work could obviously seek to improve the sophistication and complexity of the spatial learning and spatial navigation capabilities. For example, by integrating temporal information into the connections between place fields it should be possible to enable a virtual character to choose between multiple routes to the same target based on their relative time costs. In addition, by extending the range of movements of the virtual robot and the directional headings they are able to record, it should be possible to study navigational performance in a variety of other mazes; for example, the kind of mazes that are typically used by behavioral neuroscientists, as well as the more labyrinthine mazes encountered in many video games. Another form of extension relates to the perceptual processing capabilities of the virtual characters that are linked to ACT-R models. Clearly, the kind of low-level sensor processing seen in the case of the current example may not be appropriate for all situations where ACT-R is being used to control virtual characters, especially when the aim is to generate rapid behavioral responses in real-time. The advantage of a virtual environment, in this case, is that it is often possible to determine precisely what the outcome of sensor processing should be based on a knowledge of the properties of the sensor system and the character's location in the game. For example, in the case of visual input, the objects that are visible to a particular character can be determined based on the location of game objects relative to the character's view frustum. This obviates the need to engage in detailed processing of rendered images because high-level visual information regarding the location and properties of visual objects in the visual field can be computed independently of a render cycle. This strategy will no doubt be useful in situations where virtual characters must coordinate their behavioral output with respect to complex 3D scenes, although the use of low-level sensor processing routines may still be appropriate in cases where the primary purpose of the integration effort is to support scientific simulations.

Regardless of the kind of sensorimotor capabilities that are implemented for virtual characters, the ability to integrate ACT-R with Unity is likely to prove useful in a range of situations. The integration solution presented here is particularly useful in contexts where the aim is to use the capabilities of the Unity game engine to run simulations in which ACT-R models are effectively embedded in perceptually complex virtual environments. This is likely to be of particular relevance when it comes to computer simulations that seek to investigate the performance of virtual cognitive robots (as in the current case) or in situations where the aim is to advance our understanding of the role that agent-world interactions play in shaping human cognitive capabilities (Clark, 2008). It is also likely to be relevant to situations involving the interaction of multiple cognitive agents, as is the case in a number of recent scientific studies using the ACT-R architecture (Reitter & Lebiere, 2012; Smart et al., 2014).

Aside from the use of the integration solution to support serious scientific endeavors, there is also the case of cognitive architectures being used to create more human-like virtual characters. Such characters are particularly suited for applications that require synthetic agents to interact with their human counterparts, for pedagogic (e.g., Rickel & Johnson, 2000) or other purposes.

Finally, we encounter the more entertainment-oriented aspect of integration efforts: the use of cognitive architectures to enhance game play experiences by yielding characters with evermore intelligent capabilities (Arrabales et al., 2009). Given ACT-R's history of use by the scientific community specifically for the purposes of cognitive modeling, it is perhaps unsurprising that it is difficult to evaluate this particular use of the ACT-R/Unity integration solution. Further research needs to be done to determine whether ACT-R can produce the kind of intelligent responses that are currently difficult or impossible to produce using conventional game AI techniques. Needless to say, the first step in evaluating this possibility is the availability of a robust integration solution that enables ACT-R to be tested in conjunction with a state-of-the-art game engine. With such a solution now in place, and with Unity being used by an ever-greater number of professional game developers, there has never been a better time for us to explore the possibilities. Let's play!

ACKNOWLEDGEMENTS

This research was sponsored by the U.S. Army Research Laboratory and the U.K. Ministry of Defence and was accomplished under Agreement Number W911NF-06-3-0001. The views and conclusions contained in this document are those of the author(s) and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Army Research Laboratory, the U.S. Government, the U.K. Ministry of Defence or the U.K. Government. The U.S. and U.K. Governments are authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation hereon.

REFERENCES

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

Anderson, J. R., Albert, M. V., & Fincham, J. M. (2005) Tracing problem solving in real time: fMRI analysis of the subject-paced Tower of Hanoi. *Journal Of Cognitive Neuroscience*, 17(8), 1261-1274.

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., Qin, Y., Jung, K.-J., & Carter, C. S. (2007) Information-processing modules and their relative modality specificity. *Cognitive Psychology*, 54(3), 185-217.

Arrabales, R., Ledezma, A., & Sanchis, A. (2009) *Towards conscious-like behavior in computer game characters*. IEEE Symposium on Computational Intelligence and Games, Milan, Italy.

Barrera, A., & Weitzenfeld, A. (2007) *Bio-inspired model of robot spatial cognition: Topological place recognition and target learning*. International Symposium on Computational Intelligence in Robotics and Automation, Jacksonville, Florida, USA.

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 Interaction*. Cambridge University Press, New York, New York, USA.

Bringsjord, S., Khemlani, S., Arkoudas, K., McEvoy, C., Destefano, M., & Daigle, M. (2005) *Advanced synthetic characters, evil, and E*. 6th Annual European Game-On Conference, Leicester, UK.

Chi, M. T. H., Glaser, R., & Farr, M. J. (Eds.) (1988) *The Nature of Expertise*. Erlbaum, Hillsdale, New Jersey, USA.

Christel, M. G., Stevens, S. M., Maher, B. S., Brice, S., Champer, M., Jayapalan, L., Chen, Q., Jin, J., Hausmann, D., & Bastida, N. (2012) *RumbleBlocks: Teaching science concepts to young children through a Unity game*. 17th International Conference on Computer Games, Louisville, Kentucky, USA.

Clark, A. (2008) *Supersizing the Mind: Embodiment, Action, and Cognitive Extension*. Oxford University Press, New York, New York, USA.

Hope, R. M., Schoelles, M. J., & Gray, W. D. (2014) Simplifying the interaction between cognitive models and task environments with the JSON Network Interface. *Behavior Research Methods*, 46(4), 1007-1012.

Kurup, U., & Lebiere, C. (2012) What can cognitive architectures do for robotics? *Biologically Inspired Cognitive Architectures*, 2, 88-99.

Laird, J. E. (2012) *The SOAR Cognitive Architecture*. MIT Press, Cambridge, Massachusetts, USA.

Laird, J. E., Newell, A., & Rosenbloom, P. S. (1987) SOAR: An architecture for general intelligence. *Artificial Intelligence*, 33(1), 1-64.

Langley, P., Laird, J. E., & Rogers, S. (2009) Cognitive architectures: Research issues and challenges. *Cognitive Systems Research*, 10(2), 141-160.

Lawson, G., & Burnett, G. (2015) Simulation and Digital Human Modelling. In J. R. Wilson & S. Sharples (Eds.), *Evaluation of Human Work* (4th ed.). CRC Press, Boca Raton, Florida, USA.

Mattingly, W. A., Chang, D.-j., Paris, R., Smith, N., Blevins, J., & Ouyang, M. (2012) *Robot design using Unity for computer games and robotic simulations*. 17th International Conference on Computer Games, Louisville, Kentucky, USA.

Moon, J., & Anderson, J. R. (2012) *Modeling Millisecond Time Interval Estimation in Space Fortress Game*. 34th Annual Conference of the Cognitive Science Society, Sapporo, Japan.

Morrison, J. E. (2003) *A Review of Computer-Based Human Behavior Representations and Their Relation to Military Simulations.* Institute for Defense Analyses, Alexandria, Virginia, USA. (Ref: IDA Paper P-3845)

Niehaus, J. (2013) *Mobile, Virtual Enhancements for Rabilitation.* Charles River Analytics, Cambridge, Massachusetts, USA. (Ref: R1219802)

Reitter, D., & Lebiere, C. (2012) *Social Cognition: Memory Decay and Adaptive Information Filtering for Robust Information Maintenance*. 26th AAAI Conference on Artificial Intelligence, Toronto, Canada.

Rickel, J., & Johnson, L. W. (2000) Task-Oriented Collaboration with Embodied Agents in Virtual Worlds. In J. Cassell, J. Sullivan & S. Prevost (Eds.), *Embodied Conversational Agents*. MIT Press, Cambridge, Massachusetts, USA.

Rizzo, A., Hartholt, A., Grimani, M., Leeds, A., & Liewer, M. (2014) Virtual Reality Exposure Therapy for Combat-Related Posttraumatic Stress Disorder. *Computer*, 47(7), 31-37.

Salvucci, D. D. (2006) Modeling driver behavior in a cognitive architecture. *Human Factors*, 48(2), 362-380.

Schreiber, G., Akkermans, H., Anjewierden, A., de Hoog, R., Shadbolt, N. R., Van de Velde, W., & Weilinga, B. (2000) *Knowledge Engineering and Management: The CommonKADS Methodology*. MIT Press, Cambridge, Massachusetts, USA.

Shadbolt, N. R., & Smart, P. R. (2015) Knowledge Elicitation: Methods, Tools and Techniques. In J. R. Wilson & S. Sharples (Eds.), *Evaluation of Human Work* (4th ed.). CRC Press, Boca Raton, Florida, USA.

Smart, P. R., & Sycara, K. (2015a) *Place Recognition and Topological Map Learning in a Virtual Cognitive Robot*. 17th International Conference on Artificial Intelligence, Las Vegas, Nevada, USA.

Smart, P. R., & Sycara, K. (2015b) *Situating Cognition in the Virtual World*. 6th International Conference on Applied Human Factors and Ergonomics, Las Vegas, Nevada, USA.

Smart, P. R., & Sycara, K. (2015c) *Using a Cognitive Architecture to Control the Behaviour of Virtual Robots*. EuroAsianPacific Joint Conference on Cognitive Science, Turin, Italy.

Smart, P. R., Sycara, K., & Tang, Y. (2014) *Using Cognitive Architectures to Study Issues in Team Cognition in a Complex Task Environment*. SPIE Defense, Security, and Sensing: Next Generation Analyst II, Baltimore, Maryland, USA.

Smart, P. R., Sycara, K., Tang, Y., & Powell, G. (forthcoming) *The ACT-R Unity Interface: Integrating ACT-R with the Unity Game Engine.* Electronics and Computer Science, University of Southampton, Southampton, England. (Ref: ITA/2015/ACT-R-UI)

Sun, R. (2006a) The CLARION Cognitive Architecture: Extending Cognitive Modeling to Social Simulation. In R. Sun (Ed.), *Cognition and Multi-Agent Interaction: From Cognitive Modeling to Social Interaction* (pp. 79-99). Cambridge University Press, New York, New York, USA.

Sun, R. (2007) Cognitive social simulation incorporating cognitive architectures. *Intelligent Systems*, 22(5), 33-39.

Sun, R. (Ed.). (2006b) *Cognition and Multi-Agent Interaction: From Cognitive Modeling to Social Interaction*. Cambridge University Press, New York, New York, USA.

Taatgen, N., & Anderson, J. R. (2010) The Past, Present, and Future of Cognitive Architectures. *Topics in Cognitive Science*, 2(4), 693-704.

Thagard, P. (2012) Cognitive architectures. In K. Frankish & W. M. Ramsey (Eds.), *The Cambridge Handbook of Cognitive Science* (pp. 50-70). Cambridge University Press, Cambridge, UK.

Trafton, G., Hiatt, L., Harrison, A., Tamborello, F., Khemlani, S., & Schultz, A. (2013) ACT-R/E: An Embodied Cognitive Architecture for Human-Robot Interaction. *Journal of Human-Robot Interaction*, 2(1), 30-54.