A Virtual Environment Framework for Embedding Neural Models

Corey. M. Thibeault[1,2,3,*], Narayan Srinivasa[1], and Frederick C. Harris Jr.[3]

[1]Center for Neural and Emergent Systems, Information and Systems Sciences Laboratory, HRL Laboratories LLC., Malibu, CA.
[2]Department of Electrical and Biomedical Engineering, University of Nevada, Reno. Reno, NV.
[3]Department of Computer Science, University of Nevada, Reno. Reno, NV.

Abstract

An important aspect of spiking neural network research is the implementation of environments and scenarios to develop and test novel theories of function and adaptation. In addition to embodying a model in the physical world, there are a number of strategies for virtually embedding neural models. Two popular approaches are using existing games or creating custom environments. Both of these however, can require significant software development in addition to the difficulty in creating compelling visual elements. The visual world can not only be the most time-consuming to develop but to render as well. Furthermore, it is often the most unnecessary aspect of the research. In this work we briefly explore using the entity system paradigm as the foundation of a generic framework for developing virtual environments independent of the visualization. This is contrasted with using an object-oriented approach for the same goal. The resulting framework not only simplifies the development of novel virtual environments but provides a higher level of code reuse.

keywords: Neurorobotics, Entity System Paradigm, Virtual Environments, Computational Neuroscience.

1 Introduction

The concept of neurorobotics is dependent on the ability to immerse an embodied agent into a real or virtual world. Although it has been argued that using a physical environment is vital to creating intelligent systems [8], this is often impractical. Furthermore, this can create unnecessary complications during the development of novel neural theories. Employing a Virtual Environment (VE) however, presents its own unique issues. One that is often a hindrance to rapid model development is performance. This emphasis on creating detailed but slowly executing visual environments runs counter to the high-performance neural models of software simulators and more recently of high-performance neuromorphic processors.

With traditional game and VE software development the focus is on graphical rendering of the environment. Developing new worlds or elements requires creating a complete visual representation of it—introducing both development and performance bottlenecks. The work presented here is aimed at reducing this congestion. By removing the restriction of creating and rendering the visual representation, researchers can focus on how the environment will influence the entity and its sensing systems. This can not only increase performance but reduce development time as well. In addition, a system that can interact with both hardware and software models, without concerns for visualization, can help in the development of neuromorphic models. The features of this approach are:

- Designers can focus on the important interactions rather than the minutia of creating compelling visible environments.
- Results can be plotted roughly during testing and later rendered in more detail for publishing and presentation.
- Neuromorphic hardware can run at different speeds, both faster and slower than real-time—a system that can interact with these without concerns for visualization are important for rapid model development.

There are a few existing virtual environment packages that target neuroscience directly. CASTLE [12] provides environments with capabilities similar to those presented here. However, it lacks mechanisms for running simulations faster than real-time or without rendering the graphics to the screen. These restrict the performance of the neural models and their use in distributed computing. The latter is important not only for parameter searches but for stability analysis of deployable systems. Entities and environments are created in Blender with support for other 3D modeling software planned in the future.

Similarly, Webots [10] offers a comprehensive virtual environment along with a number of different modes, including a headless one. However, Webots was not developed

*Corresponding Author. cmthibeault@hrl.com

specifically for neurorobotics, and adaptation for neural applications requires custom software modules. In addition, the software requires a physical license for each instance—making distributed execution unreasonable.

None of the existing packages satisfied all of our requirements and the decision to design a custom framework was made. The development of a virtual environment can be compared to creating a level in a video game. Because of this similarity we chose to search for an appropriate design pattern from within video game development. There are a remarkably large number of game engine design patterns available and there is no apparent consensus on where each one is appropriate. This made selecting a pattern to apply to this project a surprisingly difficult task. In addition, there were no existing C++ based game engines that met our requirements — these included open source license, options for headless execution, high performance, and both development and execution on Linux platforms. In addition, it was determined that the existing engines lacked the agility required for supporting rapidly changing experiments. Based on this, a custom framework utilizing the entity-system design paradigm (ESP) was selected [13].

The game engine landscape includes a number of commercial and open source ESP based frameworks that were options for this project. The commercial Unity3D engine [1] is on such example. Although the core of Unity3D does use the ESP, it is not freely available and developing in Linux is not supported — there is limited support for previewing games in Linux though. Another ESP option would have been Artemis [2]. However, at the time this project was developed the C++ port of Artemis was still unstable.

In this paper the benefits of the ESP are highlighted by the design of a simple virtual environment—a classic Pong style game. Its implementation using object-oriented design is presented first. As we will show this works well for basic tasks but for larger environments the complexity of the element interactions is undesirable. The resulting lack of scalability and code reuse was the motivation for the ESP that is then presented. Finally, we demonstrate how the Pong environment would be represented using the ESP.

2 Playing games

2.1 Object-oriented design

The object-oriented design pattern (OOP) is based around the concept of reducing a software system into reusable components or classes. This is naturally a data centric approach however, objects are generally coupled with logic for working with the data they encapsulate. Classes define the blueprint for the state and behavior of objects and how they interact with the program.

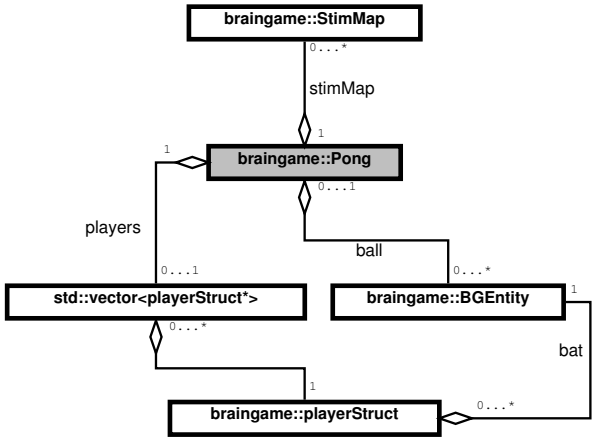

Figure 1: Pong class layout.

An additional layer of complexity that is important for OOP-based software is polymorphism. This allows derivative class objects to extend an existing object by inheriting its data and function interface. Inheritance provides a way for building and storing complex objects that are based on a common object but it can be difficult to control in generic environments where it is not always possible to predict what a user may require. In addition, these elaborate object interactions often lead to hierarchies where objects need to encapsulate any desired functionality or inherit from an object that provides that functionality. The resulting deep dependencies can make extending a project difficult.

A way around deep hierarchies would be to use composition instead of inheritance [4]. This is closely related to the framework proposed here however, composition still results in a common root object. Alternatively, aspect oriented programming can reduce redundant code and add decomposition but it introduces another level of complexity on top of OOP [5, 9]. This can make it difficult to extend and requires that each developer has a complete understanding of the software architecture before making any modifications.

When employed appropriately OOP can work for virtual environments and game engines [1]. To illustrate how to apply OOP to a virtual world we present an example of interfacing neural models to a simple pong style game. In this, a puck is given basic physics where it will move linearly through the game board and bounce off of the left, right and top walls. The player controls a paddle at the bottom of the board and must use that to reflect the puck and keep it on the board. If the puck gets past the paddle the player loses a life. Keep in mind that we are only creating the game elements, controller and interface to the neural model. The visualization is performed after the simulations have been completed by a separate program that simply renders the script output from this environment.

The objects for this system are simple and the class diagram outlining each of the properties and functions is

[1] http://unity3d.com
[2] http://gamadu.com/artemis/

presented in Figure 1. The BGEntity class is used to represent the different elements that make up the game. In this case that is the player's paddle and the puck. The StimMap class is used to encode the game space into neural stimulus based on the position of the puck. Finally, the controller for the game is implemented in the Pong class.

2.2 BrainGames

This is a straightforward environment with a small number of elements interacting. However the Pong class is responsible for much of the heavy lifting, such as collision detection, input and output processing, and game physics. For a small code base this is reasonable but, as the complexity of the task increases this design pattern quickly becomes unmanageable.

To support a reduction in VE development time, we are proposing a unique ESP implementation. Unlike hierarchical design patterns, most notably the object-orient design pattern presented above, ESP provides better code-reuse and extensibility—this is an important requirement in research and development. Traditionally, ESP has been used exclusively in video game development, however, it is an ideal solution for a generic VE engine. The features of this design are:

- A headless framework for creating high-performance virtual-environments capable of interfacing with neural simulations and neuromorphic hardware.
- Designed for C++ and can be embedding into existing neural simulators such as NCS [6] and HRLSim [11].
- Generic reusable components for efficient environment development.
- Data driven design pattern removes much of the complexity for new developers.

2.3 Entity System Design

Component based design offers a way to decompose the different functional domains of the VE entities into their constituent parts. In addition, it provides a means for layered abstraction without the negative impacts on performance and extensibility that many hierarchical object-oriented designs impose.

Entities represent groups of components—here every VE element is an entity. The components themselves do not contain any logic. Instead, a data-driven approach is taken and the components are nothing more than collections of data with exposed getter and setter functions. The control logic is implemented by systems. Systems encapsulate the update functions for each of the components. The systems are responsible for modifying the data contained within the components. The different systems contain references to the components they are interested in. In fact, the systems never have a need to reference the entity

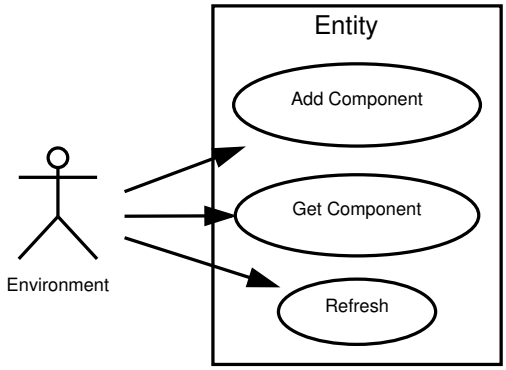

Figure 4: Basic virtual environment use cases with the Entity objects of BrainGames.

object itself. This allows the addition and removal of systems in a clean and unobtrusive way. In addition, the removal of a component from an entity does not result in a broken hierarchy. Instead the systems previously using the components of that entity simply remove the references to it. The addition or modification of systems works exactly the same way—essentially creating a run-time interface.

The design for a C++ realization of this idea is presented in Figure 3. Use cases for this concept are presented in Figures 2, 4 and 5. This represents a minimal architectural design based on the ideas presented in [3] and the framework of [2]. Unlike game engines, this implementation focuses on the specific needs of the neurorobotics community. Mainly, many parallel sensory and motor loops that are the hallmark of most neurally inspired designs.

2.4 Pong using ESP

Once an ESP framework is in place, creating the components for the Pong game becomes relatively simple. Consider this from the point of view of the main code. The world object is instantiated first along with the system objects:

```
// Create the world.
world = new World();
// Create the systems.
world.registerSystem( new MovementSystem() );
world.registerSystem( new CollisionSystem() );
world.registerSystem( new StimOutputSystem() );
world.registerSystem( new InputControlSystem() );
world.registerSystem( new RecordingSystem() );
```

The Puck and the Paddle are then the only entities added to the world:

```
// Create the puck.
Entity puck = world.createEntity();
puck.addComponent( new Position() );
puck.addComponent( new Velocity() );
puck.addComponent( new Collidable() );
puck.addComponent( new Stimulus() );
// Create the paddle.
Entity paddle = world.createEntity();
paddle.addComponent( new Position() );
paddle.addComponent( new Velocity() );
paddle.addComponent( new Controllable() );
```

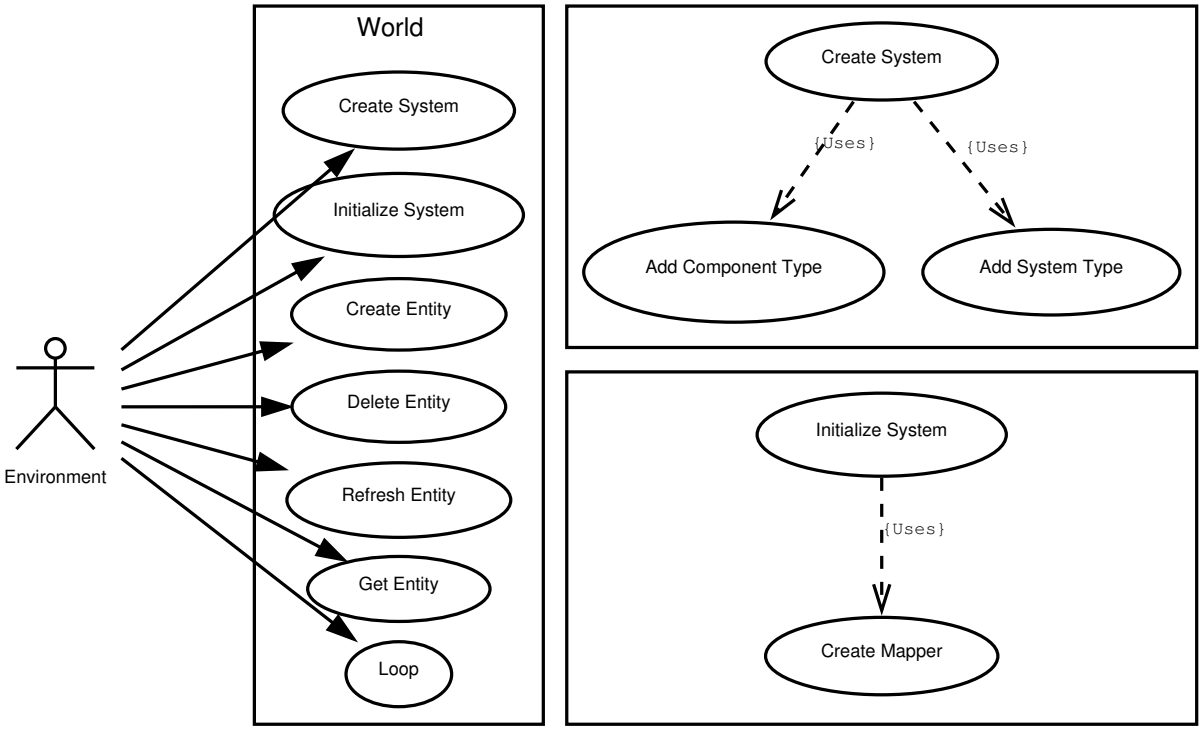

Figure 2: Basic virtual environment use cases with the world class of BrainGames.

The environment then loops until an end condition is reached. The implementation for each of the system objects defines the logic for the world. The systems loop through the entity references that contain all of the required components.

The motion of each of the entities is handled by the movement system. It uses the position and velocity components to determine the change in position. The collision system is then responsible for all objects in the game that move around and can collide with other game elements. In this case that is only the puck. The input system is responsible for gathering the input information from the neural model, processing it and updating the control system for the paddle. This could be further separated into input and control but would require a component to maintain the input information. The stim output system is used to create the neural stimulus based on the position of the puck. Finally, the recording system collects the current state of the environment and saves it for rendering off-line.

In the transition to the BrainGame architecture, the code presented in Figure 1 is moved into the separate system classes. In the ESP however, that code becomes cleaner and is logically abstracted, rather than contained in a single monolithic class. In addition, switching out systems to perform different tasks or adding players and elements no longer requires redesigning and testing the existing codebase.

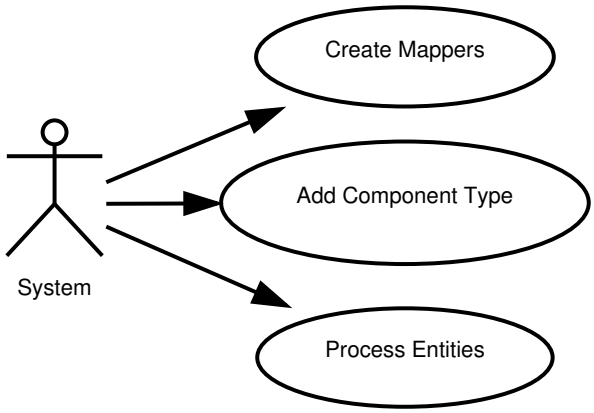

Figure 5: Basic system use cases.

3 Discussion

3.1 Benefits of ESP

A difficult problem to address in the design of these types of systems is the communication between components. One option for addressing this would be to give each entity a reference to those it needs to send messages to. The problem is this creates a tight coupling between components and changing any of those becomes extremely difficult. Another approach has been to create a message passing system. For larger systems this can become complex and again the coupling of components can be an issue. The advantage to ESP is that the systems take care of all communication

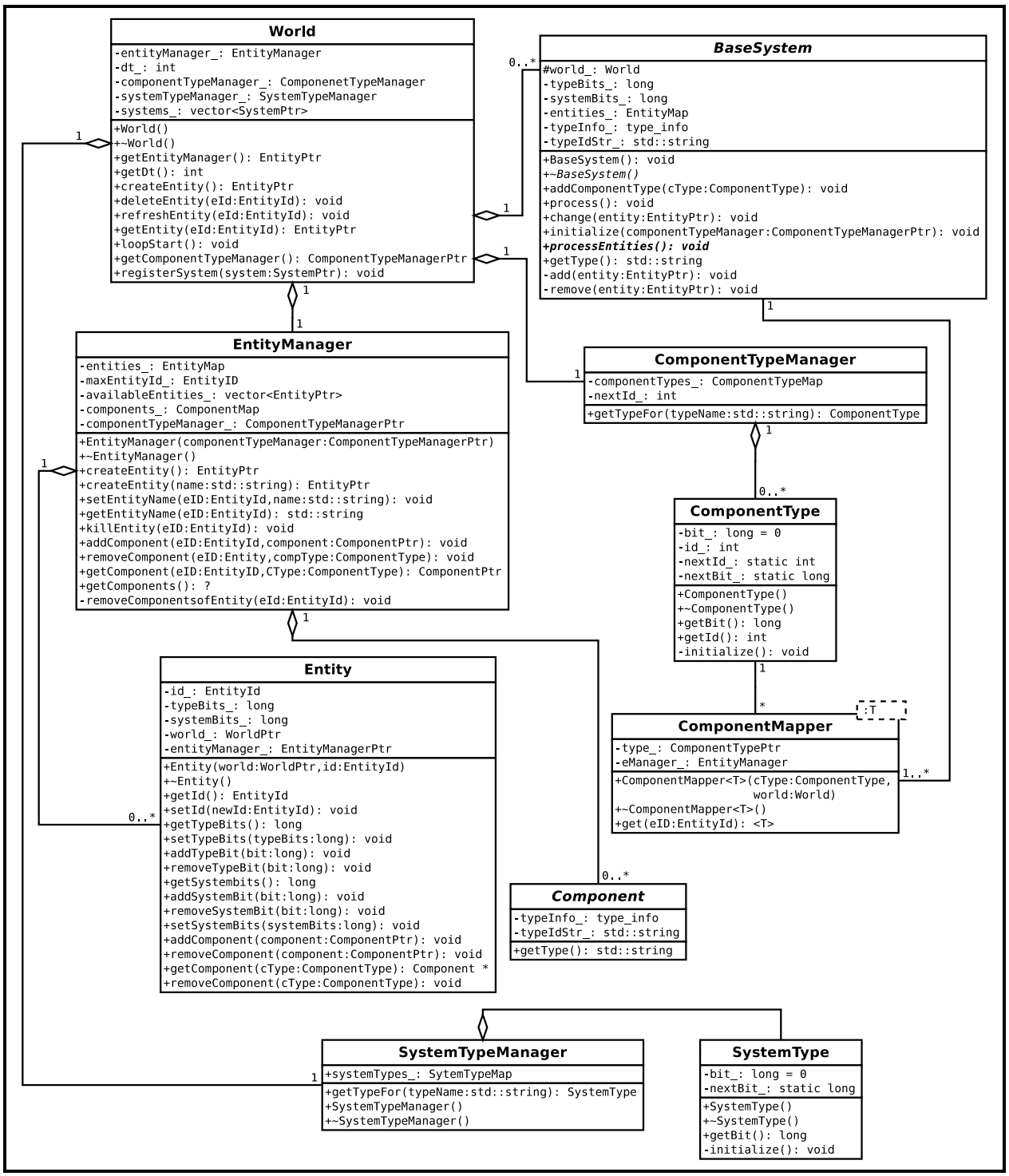

Figure 3: BrainGames Framework.

between objects. They can easily access different classes of components associated with an entity. The system does not need to know everything about the entity, only that it contains the components it requires to do its update. Similarly, a component does not need to know anything about the other components in the entity, or anything about the entity itself for that matter.

With ESP the logic for the environment is contained entirely in the system objects. If designed correctly, those systems will essentially be autonomous units. Changing one will not affect the others. In addition, the components essentially do not change. Combined, these provide a level

of code reuse that is important in the rapid development of neurorobotics.

Development and Testing also becomes more straightforward in ESP systems. In particular, multi-developer projects become more tractable, as developers can work on independent system objects. The common components are the only aspect that need to be enforced.

Finally, unlike the existing virtual environments discussed in Section 1, this design can be compiled directly into the simulation environment or it can be instantiated remotely. Additionally, the inherent support for parallel and distributed hardware further separates it from existing generic virtual environments.

3.2 Real vs. virtual worlds

In Krichmar *et. al* (2005) [7] it is argued that simulated environments introduce unwanted biases to the neural models. These are a product of the artificial nature of the input stimulus. In addition, they contend that simulated environments cannot compete with the noisy stimulus of the real world and cannot support many important emergent properties.

The chaotic nature of the real world is no doubt difficult to simulate however, there are benefits to constraining initial model development to virtual worlds. The first is the time intensive nature of dealing with physical robots. Development of novel neural models becomes extremely time consuming as repeating experiments must be completed in real time. Using a high-performance environment such as the one proposed here, models can be rapidly developed and tested. The tests can be automated and the results can be validated over multiple simulations in a realistic amount of time. The complexity of the worlds can also be increased providing a process for building up and exploring the unique properties of a given model.

Virtual worlds also provide a sense of control to the experiments. Different networks can be immersed in a common environment, allowing for a quantifiable comparison between them. As models are developed and refined, they can then benefit from the type of embodying championed by Krichmar *et. al* (2005) [7]. The virtual environments then become a compliment to their real-world counterparts.

Acknowledgements

The authors gratefully acknowledge the support for this work by Defense Advanced Research Projects Agency (DARPA) SyNAPSE grant HRL0011-09-C-001. This work is approved for public release and distribution is unlimited. The views, opinions, and/or findings contained in this dissertation are those of the author and should not be interpreted as representing the official views or policies, either expressed or implied, of the DARPA or the Department of Defense.

References

[1] Apostolos Ampatzoglou and Alexander Chatzigeorgiou. Evaluation of object-oriented design patterns in game development. *Information and Software Technology*, 49(5):445–454, 2007.

[2] Arni Arent and Tiago Costa. Artemis entity system framework. `http://www.gamadu.com/artemis/`, 2012.

[3] Scott Bilas. A data-driven game object system. `http://scottbilas.com/files/2002/gdc%5Fsan%5Fjose/game_objects_slides.pdf`, 2007.

[4] Siobhán Clarke and Robert J Walker. Composition patterns: An approach to designing reusable aspects. In *Proceedings of the 23rd international conference on Software engineering*, pages 5–14. IEEE Computer Society, 2001.

[5] Eduardo Figueiredo, Nelio Cacho, Claudio Sant'Anna, Mario Monteiro, Uira Kulesza, Alessandro Garcia, Sérgio Soares, Fabiano Ferrari, Safoora Khan, Francisco Dantas, et al. Evolving software product lines with aspects. In *Software Engineering, 2008. ICSE'08. ACM/IEEE 30th International Conference on*, pages 261–270. IEEE, 2008.

[6] Roger V Hoang, Devyani Tanna, Laurence C Jayet Bray, Sergiu M Dascalu, and Frederick C Harris. A novel CPU/GPU simulation environment for large-scale biologically-realistic neural modeling. *Frontiers in Neuroinformatics*, 7:19, 2013.

[7] J. L. Krichmar and G. M. Edelman. Brain-based devices for the study of nervous systems and the development of intelligent machines. *Artif. Life*, 11(1-2):63–78, jan 2005.

[8] Jeff Krichmar. Neurorobotics. `http://www.scholarpedia.org/article/Neurorobotics`, 2008.

[9] Uirá Kulesza, Vander Alves, Alessandro Garcia, Carlos JP De Lucena, and Paulo Borba. Improving extensibility of object-oriented frameworks with aspect-oriented programming. In *Reuse of Off-the-Shelf Components*, pages 231–245. Springer, 2006.

[10] O. Michel. Webots: Professional mobile robot simulation. *International Journal of Advanced Robotic Systems*, 1(1):39–42, 2004.

[11] K. Minkovich, C.M. Thibeault, M.J. O'Brien, A. Nogin, Y. Cho, and N. Srinivasa. HRLSim: A high performance spiking neural network simulator for GPGPU clusters. *Neural Networks and Learning Systems, IEEE Transactions on*, PP(99):1–1, 2013.

[12] SET Corporation. Castle. `https://project.setcorp.com/castle/index.html`, 2012.

[13] Mick West. Evolve your hierarchy. `http://cowboyprogramming.com/2007/01/05/evolve-your-heirachy/`, January 2007.