A GAME ENGINE-BASED SIMULATOR FOR AUTONOMOUS UNDERWATER VEHICLES

Alexander B. Strout & Roy M. Turner*
School of Computing and Information Science
University of Maine, Orono, ME 04469
{alexander.strout,rturner}@maine.edu

Abstract

The ideal development platform for autonomous underwater vehicle control software is an actual physical vehicle. However, limited availability of AUVs and the high costs associated with purchasing and maintaining them often call for the use of computer simulations as a more feasible and flexible development platform. The simulation must accurately emulate real-world conditions and equipment to allow control software development in a virtual space. Unfortunately, simulations tend to be developed from the ground up for particular AUVs and are thus idiosyncratic and difficult to reuse, and development time and cost can be substantial.

We are developing a simulation testbed for intelligent control software for AUVs that is based on using off-the-shelf, sophisticated simulation software: game engines. A game engine provides physics, modeling, and rendering capabilities that can be significantly reduce simulator development time. The particular game engine we are currently using is Unity, which allows network connections to control in-world objects. A translator/network interface then allows existing AUV control software to receive sensor data from and to control simulated AUVs in the world.

Introduction

The design and development of intelligent controllers for autonomous underwater vehicles is a difficult, labor-intensive process that cannot be done in isolation from aspects of the AUV to be controlled. The best development system and testbed for such software is, of course, an actual AUV. However, this may not in most cases be the best *first* development system. Purchasing or (especially) building an AUV is very costly, and there is significant cost involved in maintaining the vehicle and in the support needed for fielding it. In addition, the primary focus of intelligent control researchers is usually not fielding AUVs, and they often lack expertise in vehicle maintenance and operation. They may also be distant from collaborators who do have AUVs and the interest and expertise to operate them. And using an AUV for the early stages of prototyping and debugging control software can take the vehicle away from more productive work it could be doing using its existing software. Consequently, the best first testbed for intelligent control software is usually a simulator.

Unfortunately, simulators have their own problems. Development of intelligent control software is often highly specific to the vehicle itself, and so there is limited availability of off-the-shelf simulations to suit the needs of a particular project. Instead, these simulations are typically build the ground up, with the emulated vehicle systems implemented by the artificial intelligence (AI) researchers themselves and customized to the target AUV. This involves the significant overhead of designing physics engines, model design, and building rendering mechanisms (or interfacing with existing ones, such as OpenGL)—none of which directly contributes to the task of developing the control software itself.

In addition, creation of a new simulator carries with it a significant commitment to maintaining the software and adding new technology and functionality to keep it up-to-date. Software maintenance is often estimated to take up to 90% of the total effort devoted to a piece of software. This was certainly true of our own early simulator for intelligent control research, SMART [Turner et al., 1991], which, though quite simple, still rapidly became obsolete. Even very sophisticated simulators, such as the

*Corresponding author: School of Computing and Information Science, Boardman Hall, University of Maine, Orono, ME 04469, rturner@maine.edu.

DIS–JAVA–VRML simulator [Brutzman, 1995] for the Phoenix AUV [Brutzman et al., 1998] or CADCON [Chappell et al., 1999], are rapidly outpaced by new technology without significant resources to dedicate to the simulators—resources that AUV labs usually can put to better use directed toward vehicle and software development.

Fortunately, there are alternatives. There are numerous off-the-shelf platforms that have capable physics engines, rendering, and modeling facilities that others have already committed to maintaining and extending: Game engines. As the name implies, these software systems are created for the design and development of modern 3D video games. Examples include Unity [Goldstone, 2009], Torque3D [Lloyd, 2004], Polycode [Safrin, 2013], and CryEngine3 [Seeley, 2007].[1] These range from well-maintained rendering and physics libraries to full toolkits that feature GUI-driven game world and object creation editors. Using these, a game designer can concentrate on the game itself and leave the physics and rendering to the engine.

The GEAS[2] (Game Engine-based Agent Simulator) project aims to leverage game engines for AUV simulation development. We are focusing initially on the free Unity engine. In the remainder of the paper, we discuss how such a game engine can be used to simulate AUVs and their environment, and how AUV intelligent control software can interface with such a game engine via middleware so that no modifications are needed to the controller. We discuss the status of GEAS and plans for future work.

Related Work

There have been many simulators developed over the years for AUV development, including those facilitating intelligent control development. Our own early effort, SMART (Simulator for MultiAgent Research and Testing), was a simple Lisp- and C-based simulator for multiagent systems research that simulated the EAVE [Blidberg et al., 1990; Blidberg and Chappell, 1986] AUVs. Sensor (e.g., sonar) and low-level control simulations were reasonable, but vehicle dynamics was crude, graphics was simplistic, and there was little or no real physics engine.

An early example which shared some of GEAS's goals was the DIS-Java-VRML simulator from the Naval Postgraduate School, which had sophisticated physics and rendering capabilities, making use of hardware help from Silicon Graphics workstations. It was one of the first to use both standardized communication (Distributed Interactive Simulation [DIS] protocol [Fullford, 1996]) and a standardized modeling language (Virtual Reality Markup Language [VRML; Hartman and Wernecke, 1996]). While it was a very advanced simulation for its time, it also was written entirely from the ground up and it relied on specialized hardware, and its age clearly shows, at least visually. Its descendant, the AUV Workbench [Davis and Brutzman, 2005], while more modern, will also require substantial resources for maintenance and continuous modernization.

The CADCON (Cooperative AUV Development CONcept) simulator [Chappell et al., 1999] was an ambitious project at the Autonomous Undersea Systems Institute (AUSI) that aimed to provide a community multiagent system simulation facility utilizing a client–server model. The server, a Linux-based system, maintains the shared virtual world, while the Windows-based clients represent the simulated AUVs and provide visualization services. CADCON has been incorporated into a high-level multiagent systems control simulator (CoDA/CADCON [Albert et al., 2003]), and it has been updated and used to support multi-vehicle operations using solar powered AUVs (SAUVs) [Komerska and Chappell, 2006]. Unfortunately, this simulator is also custom-built and is tied to particular operating systems, and its visualization abilities and physics engine are somewhat limited compared to game engines. It is also unclear at the present time what support is available for maintenance and upgrades.

The use of game engines for simulation has been done at least once in the research community, for traffic and land vehicle simulation [Pereira and Rossetti, 2012], and it is done for so-called "serious games" (e.g., for military applications and training) and simulators. However, to our knowledge, they have not been used for simulation for AUV intelligent controller development.

GEAS

GEAS is a simulator for research in intelligent control of autonomous agents, especially AUVs. It is not targeted toward developing simulation methods for low-level vehicle control software development (e.g., control-theoretic or other movement control software), which would require a level of detail, environmental verisimilitude, and speed that it is unclear game engines are appropriate for. Instead, GEAS focuses on simulation at the level appropriate

[1] Some of the names are registered trademarks of the organizations producing the engines.

[2] Pronounced "gesh".

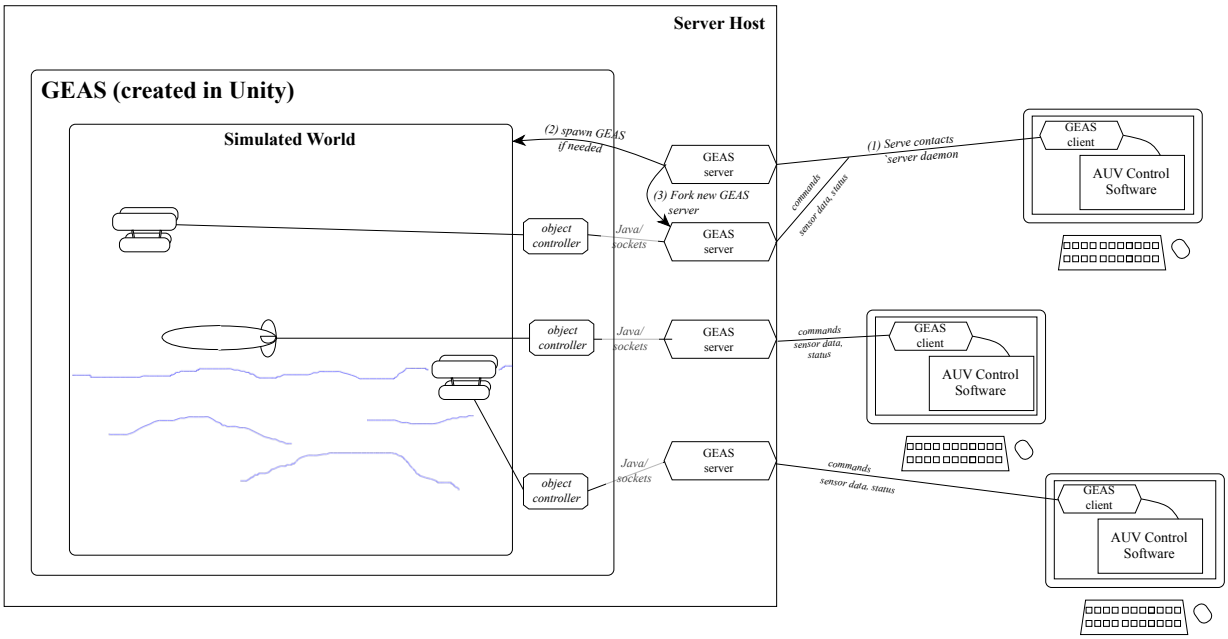

Figure 1: The GEAS simulator.

for high-level mission control software.

Figure 1 shows the structure of GEAS. The basic idea is that researchers develop their intelligent control software on any platform and in any language they wish. GEAS itself runs on a server machine, which may or may not be the same machine as the AUV controller (but usually will not be). Unity supports TCP sockets, and so GEAS can communicate with a server process that listens to a port for incoming connections.

The AUV controller either connects directly to the GEAS server running on the server machine or (the usual case) communicates with a GEAS client running on its own machine, which in turn connects to a server, depending on user preferences and needs (explained below). The server starts GEAS, if necessary, then forks a copy of itself to handle further communication with the AUV controller.

The AUV controller sends commands (e.g., to move the simulated vehicle) and receives sensor and telemetry data in the same way it would if it were controlling a real vehicle. We can think of this as the "language" it uses to control the vehicle. This language could be either idiosyncratic to the vehicle (i.e., the API[3] of the low-level software) or it could be one of the common control languages in use, such as DIS, Player interfaces [Gerkey et al., 2003], the AUV Control Language (AUVCL) [Davis, 2005], or Generic Behaviors [Komerska et al., 1999; Turner et al., 1993], if that is what the vehicle understands. Commands and data in this language are exchanged with the simulated vehicle inside GEAS via the GEAS client and server.

The GEAS simulator is a standalone program (a game) built using Unity and running on a server machine, which can be virtually any common hardware and software combination.[4] The simulated vehicles are contained in a simulated world that GEAS maintains.

As in most modern video games, interactive objects (*actors*) in Unity are controlled by the game players or by scripts run within the game world itself, e.g., to add an AI-based opponent. The code that handles the inputs from the users or from scripts are called *controllers*. In GEAS, controllers and scripts for AUV objects are implemented to accept input from and send data to GEAS servers in order to communicate with their intelligent controllers.

The simulation can be easily viewed using the game engine's built-in GUI functionality. A separate (invisible) object can be created for the user,

[3]Application Programming Interface.

[4]There is even the possibility of the server running on a mobile platform or a dedicated gaming console; see the Unity website (`unity3d.com`) for more information.

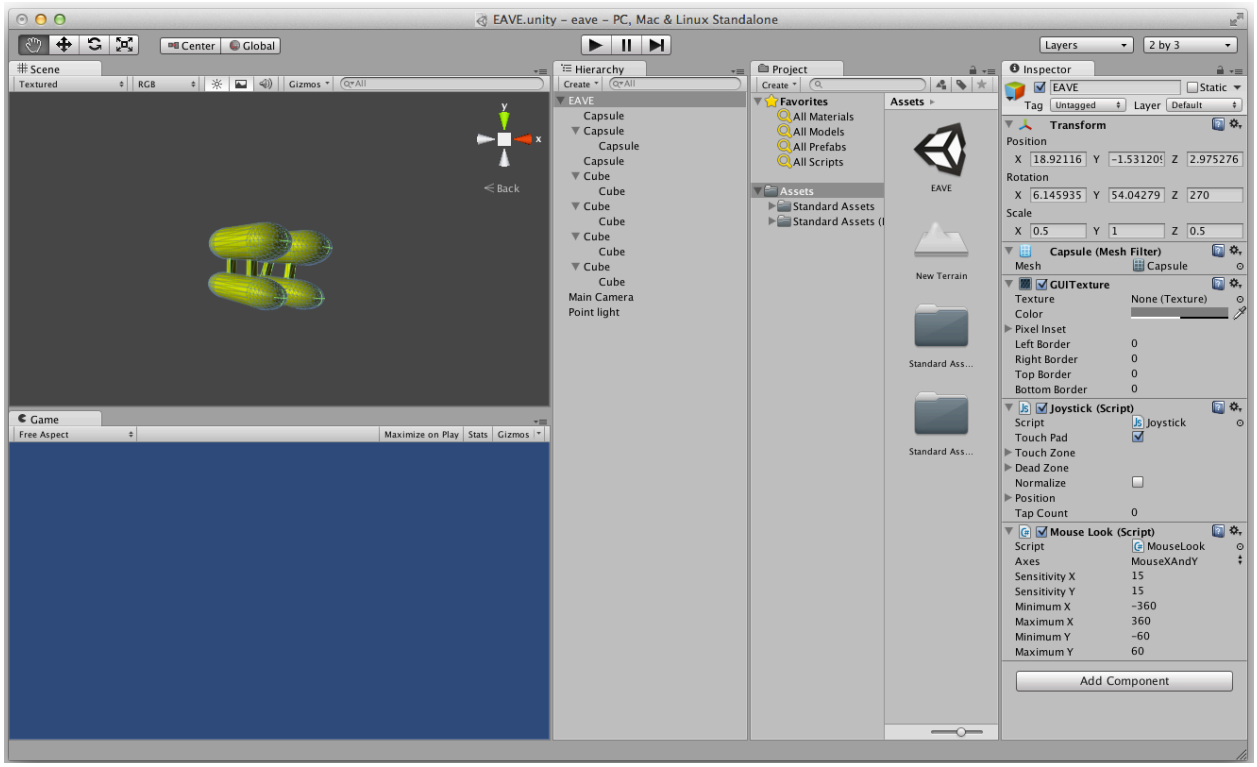

Figure 2: Unity user interface.

and this object can move around the simulated world under the user's command. Alternatively, Unity has the ability to let the user view the simulation from any of the simulated AUVs' perspectives. And, of course, since this is a game engine-based simulator, it is relatively straightforward for a human to control a simulated AUV, either directly (as in a standard video game) or by sending information via the server from another, external interface. Instrumentation of the simulation is possible as well by capturing, for example, the real position of a vehicle at different times.

The simulation can be controlled by connecting to GEAS in the usual way, then sending control commands to the GEAS server. This can be done directly by a controller program or by such a program using a GEAS client.

World and Vehicle Simulation

Unity, like most game engines, provides a rich set of objects and functions with which to design a game world, and it provides a convenient interface for the world builder. Figure 2 shows a model of an AUV being constructed, and Figure 3 shows

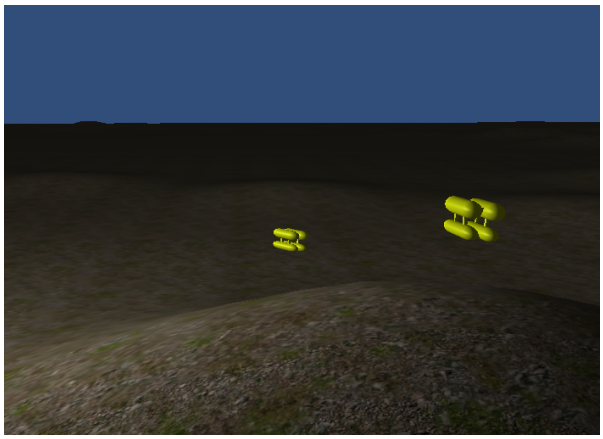

Figure 3: A simulated underwater world.

a simple simulated world. Unity can also import models from common 3D modeling programs, e.g., Blender [Roosendaal and Wartmann, 2003].

Underwater environments can be defined using a seabed terrain mesh, which is done using Unity's terrain tools, or actual terrain data can be imported from external files. Underwater environments in video games are typically simulated using lower movement speeds and reduced gravity, and

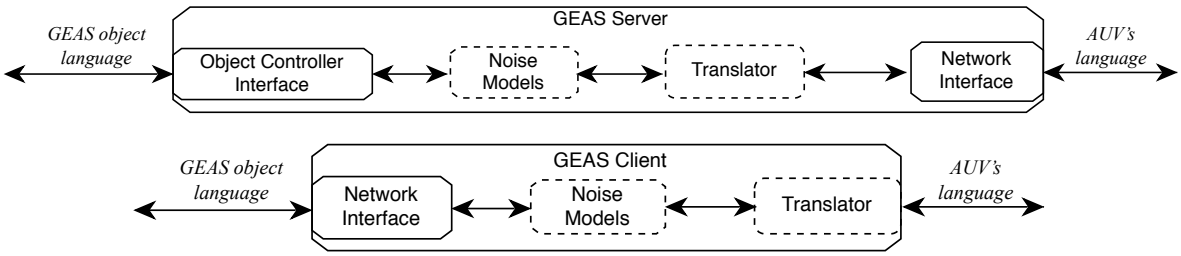

Figure 4: Internal structure of the GEAS server and client. Dashed boxes indicate optional modules.

these techniques will be used along with others (e.g., simulating drag by applying a decay factor to velocities, etc.) to simulate underwater motion, at least as a first approximation to what is needed. Future work will entail investigating how to make the underwater physics more realistic.

In addition, we are interested in simulating other aspects of the underwater environment, including currents, temperatures, and so forth that impact the vehicle and that serve as input for sensors simulation. Currents can be emulated by having the controller (the in-game script) apply the appropriate forces to the simulated vehicle based on its location relative to the currents, for example. Maps for temperature, magnetic fields, etc., can be maintained and sampled at the simulated vehicle's location, and in-world data about the objects and terrain can be used as input for sonar simulations, possibly cross-referenced with maps with information about density gradients (e.g., thermoclines) to simulate such things as sonar refraction and ambient noise.

Simulating Sensors and Effectors

GEAS will provide a suite of simulated sensors for the user that correspond to common vehicle sensors, including various sonars, location sensors (e.g., GPS, long-baseline navigation, dead reckoning), temperature sensors, depth sensors, salinity sensors, magnetometers, and laser rangefinders. Sensors will be parameterized where needed, for example, to control sampling rate or to specify the geometry and range of a sonar. One important parameter for many sensors will be the noise model, if any, to apply to the real sensor data.

Simulated sensors will take data directly from the in-world controller and synthesize the sensor data needed, possibly adding noise. This functionality will usually reside in the GEAS server. However, for flexibility, we will allow the functionality to reside instead on the client machine, in the GEAS client. This allows the user to write his or her own custom sensor simulation and noise models by creating a custom instance of the GEAS client. A special simulated sensor in the GEAS server will be available to return whatever raw data from the simulation necessary to the client in this case. Or, if the user would rather not customize the client, then he or she can write a custom sensor that can receive the raw data from the client itself.

Actuators (effectors) will be simulated in a similar manner. There is a set of behaviors that the simulated objects, via their controllers, can perform directly. These may or may not correspond to the effectors desired. For example, the object may be able to move with a particular velocity, but the AUV controller may need a simulated vehicle that has thrusters, that is, that can be controlled by applying forces along particular axes. The simulated "thruster" effector would have to map between the generated forces and changes in acceleration (and thus, periodic velocity updates).

Actuators can be simulated in either the GEAS server or client as well, as the user needs. A suite of actuators will be provided that can be implemented in the server, such as basic velocity settings, waypoint finding (i.e., move to location), orientation (e.g., heading) changes, and so forth, including some specialized effectors (e.g., lights). If the user needs additional or more specific simulated effectors, then he or she can create them as part of a custom instance of the GEAS client or external effector simulation code. In the latter cases, the client or the custom code will need to produce raw commands that GEAS' object controller can understand. Noise models, to model inexact effector response, can be specified for the GEAS server or built into the client or custom software.

Figure 4 shows where the simulated sensors and

effectors can reside in the GEAS servers and clients.

Translating Between Controller and Vehicle

GEAS also needs translation services, since it is highly unlikely that the AUV controller will use the same input/output language as the simulated object's in-world controller. For example, the AUV controller may use DIS, AUVCL, Generic Behaviors, or a custom command language when communicating with the lower-level software aboard an actual AUV. The Player interfaces/drivers are also a common control "language" for robots (particularly for land robots). GEAS would need to translate these languages to and from its internal object controller language.

Translation can occur in several ways. The default, and usual, way will be to have the GEAS server handle it, as shown in Figure 4. When the AUV controller requests a connection (usually) via a GEAS client, it will specify the translator needed, which will then be part of the server that is forked. In this case, whatever the AUV controller's native language is will be transferred over the network to the server.

Alternatively, if the user is using a language that is not handled by GEAS, for example, an idiosyncratic command language, then he or she can create a custom GEAS client instance that includes a custom translator he or she writes, as shown in Figure 4. In this case, the client will communicate with the AUV controller via the controller's own language, but the object controller language will be sent to the GEAS server. Or, if the user does not want to customize the client, he or she can write his or her own translator that communicates using the object controller's language with a default GEAS client that itself does no translation.

Status and Future Work

GEAS is still being developed and implemented. At the time of writing, the overall design has been developed, and simple versions of a simulated world and the GEAS servers and clients are being implemented. Work on simulated sensors is ongoing, as is implementation of a translator module that can understand DIS.

In the future, additional work will focus on expanding the sensor, actuator, and translation libraries. In addition, work will include developing a library of simulated vehicles and additional ambient in-world agents (e.g., fish, surface traffic, etc.).

We intend to test GEAS first using our own intelligent controllers (Orca [Turner, 1995] and the ACRO planner [Albert et al., 2007]), both to aid in those controllers' development and to gain experience with GEAS. Ultimately, GEAS will be released to the AUV community as a simulation resource.

References

Albert, E., Bilodeau, J., and Turner, R. M. (2003). Interfacing the CoDA and CADCON simulators: A multi-fidelity simulation testbed for autonomous oceanographic sampling networks. In *Proceedings of the Thirteenth International Symposium on Unmanned Untethered Submersible Technology (UUST)*, Durham, NH. The Autonomous Undersea Systems Institute.

Albert, E., Turner, E. H., and Turner, R. M. (2007). Appropriate commitment planning for AUV control. In *Proceedings of the 2007 International Symposium on Unmanned Untethered Submersible Technology (UUST'07)*, Durham, NH.

Blidberg, D. R., Chappell, S., Jalbert, J., Turner, R. M., Sedor, G., and Eaton, P. (1990). The EAVE AUV program at the Marine Systems Engineering Laboratory. In *Proceedings of a workshop at the Monterey Bay Aquarium Research Institute (MBARI)*, Pacific Grove, CA. MBARI.

Blidberg, D. R. and Chappell, S. G. (1986). Guidance and control architecture for the EAVE vehicle. *IEEE Journal of Oceanic Engineering*, OE–11(4):449–461.

Brutzman, D. (1995). Virtual world visualization for an autonomous underwater vehicle. In *Proceedings of the IEEE Oceanic Engineering Society Conference OCEANS 95*, pages 1592–1600, San Diego, CA.

Brutzman, D., Healey, T., Marco, D., and McGhee, B. (1998). The Phoenix autonomous underwater vehicle. In Kortenkamp, D., Bonasso, P., and Murphy, R., editors, *AI-Based Mobile Robots*, chapter 13. MIT/AAAI Press.

Chappell, S. G., Komerska, R. J., Peng, L., and Lu, Y. (1999). Cooperative AUV Development Concept (CADCON) – an environment for high-level multiple AUV simulation. In *Proceedings of the 11th International Symposium on Unmanned Untethered Submersible Technology (UUST99)*, Durham, NH. The Autonomous Undersea Systems Institute, Lee, NH.

Davis, D. (2005). Automated parsing and conversion of vehicle-specific data into autonomous vehicle control language (avcl) using context-free grammars and xml data binding. In *Proceedings of the 14th Interna-*

tional Symposium on Unmanned Untethered Submersible Technology.

Davis, D. T. and Brutzman, D. (2005). The autonomous unmanned vehicle workbench: mission planning, mission rehearsal, and mission replay tool for physics-based x3d visualization. In *Proceedings of the 14th International Symposium on Unmanned Untethered Submersible Technology*.

Fullford, D. A. (1996). Distributed interactive simulation: its past, present, and future. In *Proceedings of the 28th conference on Winter simulation*, pages 179–185. IEEE Computer Society.

Gerkey, B., Vaughan, R. T., and Howard, A. (2003). The player/stage project: Tools for multi-robot and distributed sensor systems. In *Proceedings of the 11th international conference on advanced robotics*, volume 1, pages 317–323.

Goldstone, W. (2009). *Unity Game Development Essentials*. Packt Publishing Ltd.

Hartman, J. and Wernecke, J. (1996). *The VRML 2.0 handbook: building moving worlds on the web*. Addison Wesley Longman Publishing Co., Inc.

Komerska, R., Chappell, S. G., Peng, L., and Blidberg, R. (1999). Generic behaviors as an interface for communication, command and monitoring between AUVs. Technical Report 9904-01, Autonomous Undersea Systems Institute, 86 Old Concord Turnpike, Lee, NH.

Komerska, R. J. and Chappell, S. G. (2006). A simulation environment for testing and evaluating multiple cooperating solar-powered auvs. In *OCEANS 2006*, pages 1–6. IEEE.

Lloyd, J. (2004). The torque game engine. *Game Development Magazine*, 11(8):8–9.

Pereira, J. L. and Rossetti, R. J. (2012). An integrated architecture for autonomous vehicles simulation. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, pages 286–292. ACM.

Roosendaal, T. and Wartmann, C. (2003). *The Official Blender Game Kit: Interactive 3d for Artists*. No Starch Press.

Safrin, I. (2013). Polycode web site. http://polycode.org, accessed 3/28/2013.

Seeley, H. (2007). Game technology 2007: Cryengine2. In *ACM SIGGRAPH 2007 computer animation festival*, page 64. ACM.

Turner, R. M. (1995). Intelligent control of autonomous underwater vehicles: The Orca project. In *Proceedings of the 1995 IEEE International Conference on Systems, Man, and Cybernetics*. Vancouver, Canada.

Turner, R. M., Blidberg, D. R., Chappell, S. G., and Jalbert, J. C. (1993). Generic behaviors: An approach to modularity in intelligent systems control. In *Proceedings of the 8th International Symposium on Unmanned Untethered Submersible Technology (UUST'93)*, Durham, New Hampshire.

Turner, R. M., Fox, J. S., Turner, E. H., and Blidberg, D. R. (1991). Multiple autonomous vehicle imaging system (MAVIS). In *Proceedings of the 7th International Symposium on Unmanned Untethered Underwater Submersible Technology (AUV '91)*.