Beyond: A Portable Virtual World Simulation Framework

Jason L. Asbahr
ASBAHR.COM
jason@asbahr.com

Abstract

This paper presents a global survey of current work on a commercial system, the Beyond Simulation Framework. Research and development of world simulation control code for real-time 3D environments is described. General approach, motivations, architecture, benefits, and the lessons learned are described, as well as future direction for work in this area. The emphasis is that the dynamic scripting approach described here has merit applied to the construction of entertainment and educational virtual environments.

1. Background and Motivation

The advent of powerful workstation-class 3D graphics accelerator hardware for personal computers has expanded the potential user base and audience for advanced real-time simulated environments for visualization, training, and entertainment. Video games in particular have experienced an enormous leap in visual realism, with games such as Quake, Unreal, and Jedi Knight pushing the graphics envelope. The popularity of these games in turn drives the purchase of faster and more feature rich graphics accelerators. However, the cost of development of these advanced titles is also growing. One consequence of this trend is the identification and development of *3D graphics engines*. These engines are components of code from a development project that can be reused both internally and licensed to third parties, reducing the time requirements and cost of development of additional projects.

While some of these engines are monolithic in construction and "tend to be tuned to a particular content style" [4], some are evolving to resemble and extend the component model of scene graph libraries like SGI's OpenInventor and Sense8's WorldToolKit [20,17]. Libraries such as NDL's NetImmerse, Hybrid Holding's SurRender, and Microsoft/SGI's proposed Fahrenheit system speed up the development of real-time 3D titles by presenting a consistent, high level application programmer interface to advanced 3D animation and rendering functionality [12, 8, 18].

Despite the existence of these scene graph libraries, the rapid development of new games and other 3D simulations remains a significant challenge. Various development groups often reengineers similar game functionality beyond the graphics library, indeed sometimes this development effort is repeated by the same group for subsequent projects. No corresponding reusable library exists for this "game play" logic and the simulation of characters, dialog, weapons, treasure, and other story-like elements of the user experience. Further, current batch-oriented development approaches are generally not well suited to the challenge of tuning for realism and fun.

One solution is to develop a reusable object framework for *game play* programming that is domain independent and highly tunable. This is the area of research focused on in the Beyond Simulation Framework described in this paper. This work carries forward the spirit of the Swarm project goals of identifying reusable "scientific components" in the specific realms of 3D gaming and virtual environments. Beyond's goal is to create a robust flexible system that enables the rapid development of engaging virtual worlds.

2. General Approach

Previous development of real-time 3D simulation software highlights the challenge of manipulating and tuning the user experience. The standard code-compile-test loop breaks down when developing logic and behavioral effects unrelated to syntax and type checking. It is therefore considered essential for tuning to be able to access and modify the state of the simulation *at runtime*.

As control code typically does not represent the largest consumer of processor time, a scripting approach represents a reasonable trade-off of execution time for runtime flexibility. Several recent commercial games implement various proprietary scripting languages for controlling game play, establishing a precedent for the value of scripting [14, 7, 22]. For example, QuakeC and UnrealScript are extension languages written for

Quake and Unreal, respectively. While these languages are particularly suited for the Unreal and Quake games, they are not general purpose languages and are not available independent from their graphic engines.

Important criteria for consideration of a scripting solution includes adherence to standards, support for debugging, portability, and object orientation. Object Oriented programming in particular is an essential technology for building extensible and reusable software components for game mechanics.

Dynamic access to the simulation allows for run-time integration and testing of new modules, which in addition to speeding the development cycle, also allows for the loading of characters, weapons, and environments that extend the base classes currently running. For simulations such as this, object oriented features like runtime dynamic binding are also extremely important.

While many approaches are possible, the Python language satisfies the majority of the practical criteria for this project. The object orientated nature of Python, as well as its elegant syntax and support for runtime dynamic binding, recommend it for a rapid production environment. The existence of cross-platform support on Unix, Macintosh, and Wintel allow for an amazing level of simulation portability, which is of key importance for the reduction of development time and cost necessary to reach the broadest audience. Debuggers and profilers are already available for Python. Further, the high level of extensibility of Python allows for quick integration of third party libraries for rendering, sound, and network functionality.

The applicability of Python as a control layer in large-scale projects is well documented [2, 6]. In addition, projects such as Alice and PUB demonstrate the applicability of Python to gaming tasks such as virtual reality, interactive fiction, and MUD systems [15, 21]. Alice in particular is pivotal in demonstrating both the power of Python in a 3D animation environment and the ease of scripting it makes possible.

3. Architectural Overview

The current simulation design consists of an object model that abstracts world elements into reusable high level components. The simulation is updated in an asynchronous event-driven manner. Underneath the simulation layer and encapsulated in the high level components are sound and graphic subsystems. The Beyond object framework complements a scene graph library's functionality such as mesh loading, animation interpolation, and collision detection by emphasizing object communication, character control, collision response, activity initiation, and abstract logical interaction.

3.1. Design Philosophy

A primary design goal is to develop classes that are interactive fiction analogs (or simulation analogs) of real world objects. Achieving this goal requires a feature rich and reusable protocol of methods that classes implement to create interchangeable and polymorphic objects, resulting in an highly flexible toolkit for building interesting and dynamic virtual worlds.

The objects collected into the Beyond framework are similar in approach to Apple's Application Kit class framework and Swarm's multi-agent discrete event simulation frameworks [1, 9]. In both the Application Kit and Swarm, objects written in Objective-C are employ runtime dynamic binding to achieve flexibility in object design and implementation. Beyond is particularly influenced by the delegation model of the Application Kit. From Swarm, Beyond derives direction for the object-as-agent and scheduling approaches.

Beyond is designed as a set of integrated modules extending the functionality of the core simulation module. Simulations are modeled as a set of interacting entities that represent both components of the world and actors inside it. Communication between these entities is modeled as discrete events at the behavioral level instead of the presentational level, similar to the client-server approach taken by the multi-user Habitat system [10].

In Habitat, the presentation client running on a user's personal computer communicates with a remote server over very low bandwidth lines. This arrangement precludes transmission of the large amount of presentational data required to fully describe a scene. Instead, behavioral data of a higher level of scene abstraction is transmitted and the client performs the work of assembling a scene locally. Beyond also employs communication of a high level of behavioral abstraction between objects, but encapsulates the logic for calculation of presentation within each SimObject. This mechanism proves appropriate both for local object interaction and for remote communication with SimObjects running on a peer computer.

3.2. SimObjects

The requirements for the framework begin with a good root object that defines a number of essential methods and attributes for subclasses to describe and extend simulated object state and behavior. The superclass and subclass objects comprising the Beyond system are named *SimObjects*, following the convention established by the Object Management Group's Business Object Management SIG for "business objects" [13]. Business objects are a useful naming convention for identifying those components of a program that capture the logic of entities and processes in the business domain that are independent from system specific or interface issues. Correspondingly, in Beyond a SimObject is a representation of a thing active in the *simulation* domain.

As in the Swarm system, simulation elements are modeled directly as objects, with instance variables representing the state and methods implementing the behavior of the simulated elements [9]. Subclasses define major game-oriented abstraction types, such as Room, Character, or Weapon, which are further subclassed to refine and specify functionality for a given simulation's requirements, as shown in Figure 1. SimObjects connect up to each other in the simulation and their communication embodies both the behavior and the state of the simulation.

Simulations are created by hierarchically organizing SimObjects in containment graphs, where logically general instances contain more logically specific instances. For example, a given Room contains a number of Characters, each of which in turn contains a number of inventory items, such as a Backpack which in turns contains a Lamp, a small Sack, and a Bottle of water. Simulation events are passed from the top level SimObject in a depth-first fashion to contained SimObjects, each level calling appropriate methods on the next as per the logic of the given container object. It's interesting to note that a "closed" object in the Beyond system (a Room, Backpack, pirate's Chest, etc) will generally not pass events to items contained within it. This serves as a way of partitioning off sections of the simulation, allowing objects that have geometric representations to effectively "disappear" from the scene graph as they cease to receive updates to draw themselves.

In addition to the general containment capability, SimObjects also have specific "arbitrary subdivision" object containers that are used to define important areas of a given SimObject, such as a character's WieldedWeapon, an machinegun's Handle, the mounting point for a Missile, and so forth. Messages can then be targeted as the specific slot and be received by whatever SimObject instance is contained in that slot. In these cases, SimObjects register themselves as delegates of the containing object for those slots, or are registered as delegates by the containing object when containment is assumed, as when a character picks up the item in question. The slots have 3D geometric positions as well as logical positions, so that any geometry represented by the contained object is appropriately parented to the containing object's geometry based on that position.

3.3. SimObject Actions

SimObjects perform actions by calling and executing methods on themselves and on other SimObjects. Actions involve the manipulation of objects in the simulation, picking up or dropping, throwing, using, eating, and destroying. Doors have open, close, lock, and unlock functionality. Keys transmit unique identifiers to locks. Projectile Weapons spawn bullets which are in turn SimObjects, and bladed Weapons directly *causeDamage()*. General behaviors are extended by subclassing to add new behaviors or special effects.

More complicated SimObjects such as *player characters* (those controlled by the user) and *non-player characters* (those controlled by the simulation) have correspondingly more complicated actions. The taxonomy of character actions is a significant object oriented design issue, which will be only briefly touched on here. Character classes are generally concerned with locomotion (walk forward, backward, left, right, run, climb), object interaction (attack, use, wield, pickup, drop), character communication (say, shout, whisper), object exchange (give, take), and combat (melee, missile, instant-hit).

The bridge between logical interaction and graphical representation in the virtual environment is constructed by the encapsulation of *scene graph subsets.* A scene graph subset contains connected and malleable pieces of 3D geometry that are placed, rotated, and animated by SimObject instances. SimObjects initiate the loading of geometry from disk or control its dynamic construction by calling methods of the underlying scene graph API. They activate, deactivate, and set floating values of various features of the scene graph API for the particular scene graph subset they control. Some examples of scene graph features controlled this way include texture mapping, alpha-blending, scaling, and other visually interesting effects.

This interconnection of logical and geometry representations enables logical actions such as a door opening or a character walking to have and to activate a corresponding animation in its scene graph subset. These animations form the basis of world motion for characters and objects in three dimensions, and are typically artist-generated, in contrast with complete physical simulation in Karl Sim's creature system [19].

3.4. Simulation as Scheduling

An important concept in the Beyond system is the difference between *actions* and *activities*. Actions as described above are instantaneous events, while activities are collections of actions executing over time.

An activity is a collection of events represented as Python method objects and arguments to those methods which are called on the owner SimObject or on other objects associated with the owner. Activity events, or actions, executed as method calls on SimObjects preserve encapsulation and thus work in conjunction with method-bound logic. As in the Swarm system, these structures are partial orders, time dependencies between two events sorted by timestamp, or in the Beyond case, by keyframe [9].

In Beyond, activities are grouped hierarchically in an activities set, which allows higher priority activities to subsume others lower priority activities. SimObjects track the progress of their internal activities and receive *activityFinished()* event notification when they have completely executed all actions in an activity. This allows the SimObject to change states, if necessary, and to initiate new activities based on its state. As SimObjects proceed through complex activities, they may initiate a number of sub-activities to accomplish a particular user-defined goal.

A special case of a activities is an activity associated on a one-to-one (isomorphic) basis with particular animations of the scene graph subset geometry. On initiation of this activity, an associated animation is begun in the scene graph subset. Actions are stored at appropriate keyframes in the activity, which allows animators to cause specific events in the simulation to occur at exact times during the course of the animation. This mechanism provides a precise way to indicate that a sound such as a footstep should be played when a character's geometric foot touches the ground, or that a missile should spawn and leave behind a trail of smoke particles as it travels through space.

3.5. Autonomous SimObjects

While logically abstract, SimObjects connected with animatable scene graph subsets become, in effect, virtual brains for the physical body represented by the particular scene graph subset. The logic of the SimObject determines the behavior of the body, whether magic sword, giant robot, or transient explosion effect. This approach is inspired by the brain-body control systems of Karl Sims' genetically evolved creatures, using programmer-generated behavioral logic instead of automatically generated state machines for creature action [19].

SimObjects exhibit autonomous behavior by connecting sensory inputs and effector outputs with an internal state machine. A SimObject can have a simple or complex behavior model. In the case of creatures and other artificially intelligent agents with complex behaviors, internal state is modeled around the purpose or role of the character in the simulation.

Currently these "brains" are implemented as sets of hierarchical finite state machines. At the lowest level a state machine concerned with motility controls the selection, initiation, and transition between motion activities and corresponding animation sets. Above the motility state machine is the behavior state machine, which controls overarching character-world response activities. This includes interacting with other SimObjects, for example taking damage and dying.

Internal state can be represented directly as instance variables for emotional qualities such as fear (flee from other SimObjects), anger (charge or attack other SimObjects), desire (pursue other SimObjects), and can be modulated by a representation of energy (the SimObject can be highly energetic or very tired).

The perception model for autonomous characters is similar to that of the artificial fishes of Demetri Terzopoulos, currently emphasizing distance, direction, and collision with other SimObjects [23]. Sensors are implemented as concentric bounding volumes centered on the SimObject's geometry. The bounding volumes are submitted to the scene graph library's collision detection facility and the resulting colliding geometry is assembled into one or more *collision lists* in the originating SimObject. SimObject references collected this way are considered perceived by the sensor in question. Collision lists are sorted by type and a priority assigned by the distance from the sensing object.

One example is a simple auto-tracking missile object. At each step fo the simulation, the missile travels forward in the direction it was launched until it comes into proximity with an object it can destroy (or at least damage a bit). At a medium range distance, the missile arcs in the general direction of the target, allowing the target a chance to escape. Once the missile has closed with the target to the near range distance, it arcs much more sharply toward the target and increases speed, making it far less likely for the target to escape.

Autonomous SimObjects need not be explicit characters in the simulation, but instead can imbue intelligent behavior into ordinarily simple simulation elements. For example, a “smart camera”, a camera with behavior logic, can identify and track the main character moving through the simulated environment under direct user control. Currently the Beyond virtual camera can set a view target, a view POV, and perform various interpolations to track motion between those two points. Future versions of the camera could refocus on other non-player characters in the scene if their action is deemed of sufficient dramatic importance, similar to Noma and Okada’s calculation of *view direction unsuitability function* for virtual camera animation [11].

As in Noma and Okada's system, the camera calculates an optimal point of view location and direction of view orientation by solving simple functions for weighted values applied to objects in the view space. For example, if a character in the view is talking, her view importance grows and the function shifts the camera to bring her into focus. When her companion begins speaking, the weights shift and the camera moves to correspondingly focus on him.

3.6. Simulation Management

A SimObject subclass known as Application acts as the “simulation manager”. The Application object (AppObject) tracks major game states and can shift the simulation from one state to another. Each loop, the AppObject receives a process tick from the OS Shell, which it uses to advance the simulation through time. The AppObject messages its collection of top-level SimObjects with process messages, iterating over sets of contained high-level SimObject instances. The AppObject maintains multiple lists of varying priority, and SimObjects can add and remove themselves from different priority list for processing.

The AppObject loads an end-user modifiable "boot" module which defines the domain-specific responses to perform. The boot module contains, among other things, application defaults such as the resolution of the graphics window and the default mode in which to operate. It also loads user input configuration files for translating/processing keyboard, mouse, and joystick input for each of these operational modes.

Key-to-object method event translation is handled in the AppObject via dictionaries for mapping specific inputs to specific event outputs. User input such as key presses, mouse, joystick, and glove movements are resolved as requests to perform actions on SimObjects. SimObjects messaged by the AppObject are required to support the Beyond protocol and respond to events in a polymorphic manner. The key-to-event translation and domain-polymorphism is largely related to the type of simulation loaded.

The AppObject also tracks the current event focus SimObject and translates events for the focus based on game state (and, of course, the configuration/translation tables for that state). The focus can shift to be any SimObject in the simulation, but is most often probably either a Console, GUI, Camera, or main Character element. It is via this mechanism that user events are distributed to the appropriate receiver for the given simulation state.

3.7. Simulation Console

Another useful SimObject subclass is the Console object, which provides access to the simulation runtime for tuning and rapid development. The console is a simulation object given a time slice of computation along with the other SimObjects, through timed event messaging. The console polls for inputted text characters and assembled strings. When an end of line (a return) is reached, the console gives the string to the Python interpreter to execute.

The Network Console subclass extends the basic capabilities of the console by opening and maintaining communication over a socket. This enables users to *telnet* into the application from a local or remote machine and observe and manipulate the state of the running simulation.

Because the console is a SimObject like any other and processed in the same time-sliced way, users are able to give commands without stopping the simulation, similar to lightweight threading. The console enables the addressing, observing, and messaging of any object in the simulation. Users are able to load characters, spells, and environment components by messaging SimObjects directly, a very powerful tool.

4. Embedding Python

The Python interpreter is embedded in a thin OS-specific executable which loads the Beyond core and any proprietary libraries wrapped as extensions. Platform dependent code is concentrated in this "OS-Shell" executable to ease porting efforts between divergent platforms.

On application initialization, the OS Shell imports the Beyond modules, gets the Application class, and then calls PyObject_CallObject to instantiate an Application instance. All communication between the thin executable and the Beyond layer is handled through PyObject_CallMethod calls on the Application object instance.

The main loop is in the thin executable. At runtime, the OS Shell collects input from various devices attached to the system, including the keyboard, mouse, and joystick. It passes these input events to the Beyond Application Object as *acceptInput()* messages. The OS Shell makes multiple calls to the Beyond layer on each pass through the loop, one for keyboard input, one for mouse input, one for joystick input, and one for time-delta input, the time passed between this loop "tick" and the previous. Certain operating-system events are handled directly by the OS Shell, such as handling whether the application is "in focus" or not. The Beyond layer calls back serious errors to the OS Shell for user notification. The system also takes advantage of Python's capability to redirect *stdout* to a file object and also proxies the logging data to the optional network console object.

5. Subsystem Integration

One of the critical components of the Beyond system is a "platform" approach to game development by remaining independent from the scene graph and other subsystems. Beyond simulation code is written to execute ideally with or without a graphical representation, either on the client side or server side of a distributed game environment. Well written Beyond code can conceivably execute in a game server for computing logical object interactions between multiple graphical clients, as a multi-user dungeon, or in a client program in text-only debugging mode.

Subsystems such as the scene graph library, sound library, and network communication library are loaded into Beyond as Python-wrapped extensions. The process of wrapping and embedding is greatly enhanced by the use of SWIG [3]. Functions in C are exposed as Python function objects, and compiled objects in C++ are exposed as Python shadow objects. Various subsystems can be given a common interface, enabling independence from a particular subsystem implementation. Subsystems can thus be mixed and matched for optimal solutions for particular deployment situations. A diagram for data flow through such an arrangement is shown in Figure 2.

Once mesh, animation, and texture data have been loaded from disk or over the network, the largest portion of CPU time is devoted to geometry processing and rendering. For performance reasons the majority of geometry-related code needs to be implemented in a compiled language, and is typically part of the C/C++ scene graph library.

Subclasses that deal with geometry explicitly import scene graph modules which contain Python classes and shadow classes of the native scene graph objects. Support code in the scene graph module wraps scene graph function calls and presents a consistent method call API to the containing SimObject. These geometry objects are registered to SimObject subclasses as delegates, and thereafter receive those method calls intended for geometry manipulation.

6. Conclusions and Future Work

Beyond has been in development for about a year from various designs created over the last four years. The original inspiration for Beyond derives from frustration with the inappropriateness of the traditional code-compile-test loop for virtual world creation and the need for a highly interactive means of "reaching in" to the simulation. The system enables the development of complicated real-time environments that have abstract logical as well as graphical interactions.

The current implementation of Beyond consists of 25 core classes and a collection of about 15 game-specific derived subclasses. Simulations of relatively high polygon-count scenes with many independent characters have achieved run time frame rates of 30fps and higher on PC-class hardware. Game simulation code using the alpha-level Beyond system consumes approximately 10% of CPU time running on a Pentium 200 with a first generation 3Dfx Voodoo accelerator. The Beyond system has already been integrated with two unique scene graph libraries, two sound libraries, and one networking library.

Features such as the preliminary remote console has proven very useful for run-time exploration and debugging. There is room for improvement in handling exceptions in the system, and the need for tight integration with a GUI tool for non-technical users is very clear. Additional work remains to be done in determining the ideal balance between interpreted and compiled simulation code, though initial results are very promising.

In addition to GUI support, future versions of the system will include support for object archiving and restoring, allowing every aspect of the simulation state to be saved, complete with object attributes, and links between objects. An improved Medusa-like network console is also planned [16]. Further enhancement to the console could also include a more sophisticated interactive-fiction parser like that in the PUB system [21]. Finally, there is potential for integration with a free scene graph library, perhaps one based around enhancements to the Python VRML module [5].

Development of Beyond continues today as the simulation layer for a science fiction adventure game set in near future Japan. The public will have access to the Beyond system as part of the release of the game, anticipated in Q4 1999.

Acknowledgements

First, thanks to Guido and the rest of the Python devotees for providing such an excellent set of tools and a dynamic, supportive intellectual community. Personal thanks to Mario Vassaux at Ionos for funding the early prototype of this approach. Thanks to Benjamin Koo, Edward Robinson, Michael Crawford, and John Edwards for providing invaluable feedback and suggestions. And special thanks to Tina LeBlanc.

References

1. Apple Computer, Inc., *Application Kit Reference*, http://gemma.apple.com/techpubs/rhapsody/.

2. Beazley, D.M., Lomdahl, P.S., "Feeding a Large Scale Physics Application to Python", *Proceedings of the 6th International Python Conference*, San Jose, California, October 14-17, 1997.

3. Beazley, D.M., "Using SWIG to Control, Prototype, and Debug C Programs with Python", *Proceedings of the 4th International Python Conference*, Lawrence Livermore National Laboratory, June 3-6, 1996.

4. Bishop, L., Eberly, D., Whitted, T., Finch, M., Shantz, M., "Designing a PC Game Engine", *IEEE Computer Graphics and Applications*, January/February, 1998, pp.46-53.

5. Fletcher, M., *mcf VRML Modules*, http://starship.skyport.net/crew/mcfletch/, July 1998.

6. Hinsen, K., "The Molecular Modeling Toolkit: a Case Study of a Large Scientific Application in Python", *Proceedings of the 6th International Python Conference*, San Jose, California, October 14-17, 1997.

7. Huebner, R., "Adding Languages to Game Engines", *Game Developer*, September 1997, http://www.gamasutra.com/features/programming/100397/languages1.htm.

8. Hybrid Holding Ltd., *SurRender Programmers Manual*, http://www.hybrid.org/surrender.

9. Minar, N., Burkhart, R., Langton, C., Askenazi, M., "The Swarm Simulation System, A Toolkit for Building Multi-Agent Simulations", June 1996, http://www.santafe.edu/projects/swarm/overview/overview.html.

10. Morningstar, C., Farmer, R., "The Lessons of Lucasfilm's Habitat", *Cyberspace: First Steps*, ed. by Michael Benedikt, MIT Press, 1992, pp. 274-301.

11. Noma, T., Okada, N., "Automating Virtual Camera Control for Computer Animation", *Creating and Animating the Virtual World*, ed. by Thalmann & Thalmann, Springer-Verlag, 1992, pp.177-187.

12. Numerical Design Ltd., *NetImmerse Programmers Manual*, http://www.ndl.com.

13. Object Management Group Business Object Management Special Interest Group, *Glossary of Terms*, http://www.omg.org/docs/1995/95-09-02.txt, September 1995.

14. Olivier Montanuy, *Unofficial Quake-C Specification*, http://www.gamers.org/dEngine/quake/spec/quake-spec34/index1.htm.

15. Pausch, R., et al., "A Brief Architectural Overview of Alice, a Rapid Prototyping System for Virtual Reality", *IEEE Computer Graphics and Applications*, May 1995.

16. Rushing, S., *Medusa: A High Performance Internet Server Architecture*, http://www.nightmare.com/medusa/.

17. Sense8 Corporatation, WorldToolKit Virtual Reality Support Software, http://www.sense8.com.

18. Silicon Graphics, Inc., "Fahrenheit: Defining the Future of Graphics", http://www.sgi.com/fahrenheit/home.html.

19. Sims, K., "Evolving Virtual Creatures," *Computer Graphics*, Annual Conference Series, 1994, pp. 15-22.

20. Strauss, P., Carey, R., "An Object-Oriented 3D Graphics Toolkit", *Computer Graphics,* Annual Conference Series, 1992, pp 341-350.

21. Strout, J., *Python Universe Builder* , http://www.strout.net/python/, June, 1996.

22. Sweeney, T., *Unreal Script Language Reference*, http://unreal.epicgames.com/UnrealScript.htm, July 1998.

23. Terzopoulos, D., Tu, X., Grzeszczuk, R., "Artificial Fishes with Autonomous Locomotion, Perception, and Learning in a Simulated Physical World," *Artificial Life IV*, ed. by Brooks & Maes, MIT Press, 1994, pp 16-27.

Figures

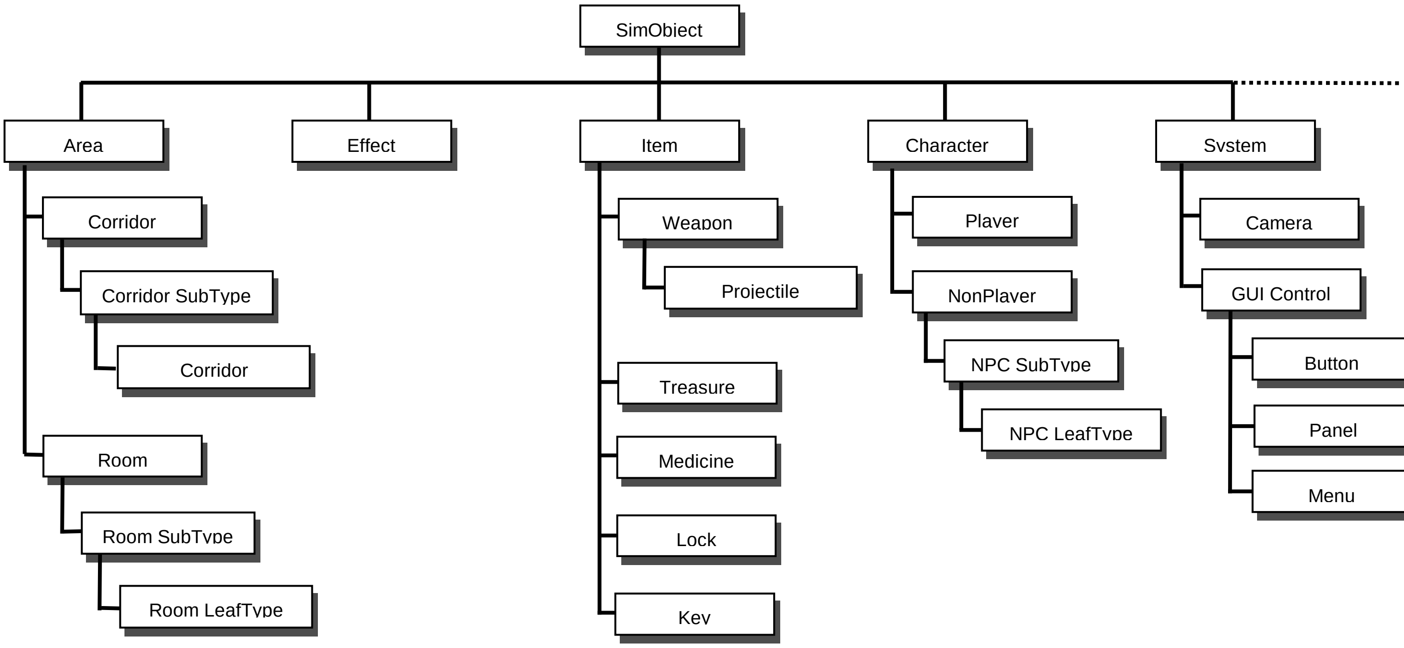

Figure 1: Beyond System Hierarchy Subset

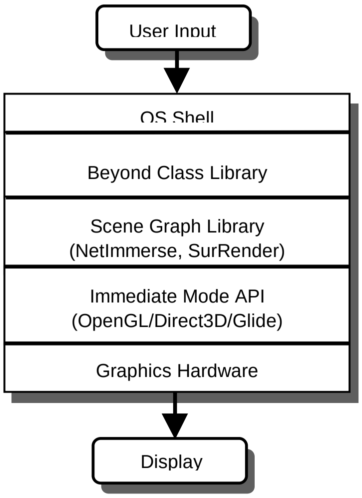

Figure 2: Beyond System Data Flow