PVoT: An Interactive Authoring Tool for Virtual Reality

Jinseok Seo and Sei-woong Oh,

Dept. of Game Engineering, Dong-eui Univ., Busan, Korea

Summary
The major problems that put virtual environment developers in difficult situations are: (1) the need to satisfy the three important requirements of virtual environments, namely, performance, presence, and usability, (2) complexity of VR objects that have three distinct aspects such as form, function, and behavior, and (3) the physical/logical gap between development and execution environments. To solve the problems, this paper proposes a set of computer-aided tools called "PVoT (Portable Virtual reality systems development Tool)." PVoT is designed based on the methodology, called "CLEVR (Con-current and LEvel by Level Development of VR systems)," a comprehensive collection of conventional and new concepts for building VR systems with three major philosophies [1]. With PVoT, one can design and immediately validate various aspects of the virtual reality systems in design reducing the temporal gap. This results in a highly interactive and seamless development environment. PVoT also collects performance data of all high-level elements in a given VR application and provides developers with a basis for performance prediction and tuning. The effectiveness of CLEVR/PVoT is demonstrated with case studies and an analysis of the development process.

Key words:
Virtual reality, Authoring Tool, Methodology

1. Introduction

Developing, validating, and maintaining VR applications is still a very difficult process. It is because VR applications have their inherent difficulties and complexities in comparison with other application software. In addition, there is practically no methodologies and widely accepted tools that are specifically designed for the need of VR application development. Most virtual environments are still implemented using procedural programming languages and tools like compilers and debuggers. Though recent object-oriented applications programming interfaces (API) [2][3][4][5][6][7] provide abstractions for the system functionalities (scene graph, device handling, display, etc.), artists and content developers (vs. programmers) would like to work with "concrete" VR objects, through reuse and composition.

In this paper, we first clarify the problems that underlie the difficulties and complexities that cause in building VR systems in Section 2. Section 3 reviews work related to our research. Then, in Section 4 and 5, we introduce a methodology and tools that can help solve these problems and promote higher efficiency in VR content development. In Section 6, the effectiveness of our work is illustrated with a brief analysis of the development process. Section 7 demonstrates our work by illustrating three applications, which have been built and/or supported using our approach and tools. Finally, in Section 8, we conclude this paper with a summary and discussion.

2. Major Problems in Developing VR Systems

In this session, we discuss three major characteristics that distinguish VR applications from other types of S/W. These characteristics are also the causes that put VR application developers in difficult situations, and thus constitute the target problems this research is trying to address.

2.1 Satisfaction of Presence, Usability, and Performance

The first problem is that we have to satisfy the three important requirements of virtual environments, namely, presence, usability, and performance. It is so difficult and complex to concurrently maintain acceptable levels of these three requirements, which are often conflicting with each other.

There are many elements that influence presence and usability. They include sensory realism, input/output multi-modalities, degrees of interactivity, simulation fidelity, employing special effects, input devices, device-handling routines, interaction models, etc. Many of the elements also affect performance. We should choose an appropriate combination of the various elements in order to achieve satisfactory levels of presence and usability while maintaining acceptable performance. However, it takes too much effort and time, because many trial and error iterations are needed to determine the proper combination of the elements and minimize the performance degradation caused by cost of newly enhanced elements. In addition, each element has its threshold (that is, limitations) at which higher level of the

Manuscript received April 5, 2007
Manuscript revised April 25, 2007

element does not contribute to presence or usability any more.

2.2 Three Aspects of VR Objects: Form, Function, and Behavior

The next problem is the complexity that VR objects inherently possess. VR objects are abstractions that represent the object exist in virtual environments. Unlike objects in other software systems, VR objects have three orthogonal aspects such as form, function, and behavior [8].

Form refers to the outer appearance of VR objects, their structure (for composite objects), other physical properties, and the scene structure of the virtual world. Function basically refers to encoding what VR objects do (i.e. primitive tasks) to accomplish their behavior, whether autonomously, or in response to some external stimuli or events, while behavior refers to how individual VR objects dynamically change and carry out different functions over a period of time, usually expressed through states, exchange of events, and inter-state transitions.

Each of these three aspects is deeply interrelated with and dependent on others. Moreover, in many cases, the behavior part of an object can easily become complex. Consequently, when a virtual world has many objects and their behaviors have many concurrent flows, not only designing and implementing them but also verifying and validating them would be much harder.

However, most VR application development platforms are just ordinary imperative programming development environments that are not suited for handling such interrelationships and dependencies and properly managing the complexities of objects such as concurrency and synchronization.

2.3 The Gap between Development and Execution

What makes virtual environment construction more difficult is that, on top of having to tackle the traditional computational and logical errors, it is also an exploration task. Developers must find the right combination of various types of constituents of the virtual environment like object, display and simulation details, interaction and modalities, etc. In addition, as stated in the two earlier subsections, the developing process of a VR application should be iterative and incremental in order to not only concurrently satisfy presence, usability, and performance but also handle the complexity of the three aspects of VR objects.

However, unlike ordinary programming tasks, for VR, the execution and development environments are difference, not just in the temporal sense, but also in the physical sense. Many VR systems are unusable simply because the developers are literally tiered of switching back and forth between the development (e.g. desktop) and execution environments (e.g. immersive setup with HMD, glove, trackers, CAVE-like systems, etc.), and fails to fully set and configure the system for usability. Therefore, we need methods to reduce the gap. Reducing the gap may spare us the effort of switching back and forth between environments and help us develop VR content of high quality.

3. Related Work

3.1 Previous Work and Software Engineering for VR

ASADAL/PROTO[8] (which is the predecessor of CLEVR/PVoT) seamlessly integrates constructs to specify form, function, and behavior, and can be used for real time 3D graphics or VR objects. The specification results (which are composed of “Statecharts”, “DFD”, and “VOS”) are simulated and translated into programming languages. However, the semantics of the translated code cannot coincide with the ones of the specifications, because conventional programming languages do not directly support the functionalities such as concurrency and hierarchy, which are the major features of “Statecharts.” Therefore, it is not easy task to combine the translated code with widely used commercial/academic VR APIs.

The Marigold toolset [9] was developed to fuse abstract modeling into the development of virtual environment dynamics. The abstract models are constructed using an hybrid specification formalism (an extended Petri-net). Using the Marigold toolset, users supplement the specification with code segments, and implementation can be produced in C/Maverik [10]. However, these specification formalisms lack the notion for depth (e.g. hierarchy) or modularity, and so the specification can quickly become too complex to handle. The semantics of the Statecharts used in CLEVR/PVoT can handle both hierarchical abstraction and synchronized behavior.

3.2 Presence and Performance Maintenance

Most approaches to dealing with the real time performance requirements of VR systems have focused on reducing the number of objects/polygons that need to be processed by the graphics pipeline in any given frame [11][12][13][14]. Many researchers considered the problem of computing for the part of the virtual world which is directly visible from a given viewpoint (in order to reduce the number of objects or polygons to process) exploiting the model structure of figuring out an occluder and their occludee objects/ polygons [15]. These approaches require

assumptions which may not be applicable to the construction of general VR worlds.

This exploitation of simulation levels of detail has been suggested first by [16]. They used three levels of simulation detail for one-legged virtual robots. They demonstrated how the overall performance varied by dynamically switching the simulation LODs. However, their work did not elaborate on the engineering side of the approach. The models and LOD switching techniques were handcrafted for the particular example.

Several researchers have studied the elusive notion of presence. Most people in the VR community seem to agree with the definition of presence as the feeling of being in the VR world and on the importance of provision of presence as a defining quality of VR. There have been many studies on elements that influence presence [17][18][19][20]. They may include sensory realism, input/output multi-modalities, degree of interactivity, simulation fidelity, employing special effects, etc. Shim and Kim [21] proposed a concept of level of presence (LOP), in which we should select a set of elements and decide proper levels of them to maximize the overall presence while maintaining acceptable performance. However, the cost and benefit model of the elements for presence should be rebuilt according to VR contents.

3.3 Interaction and Interface Design

Another important aspect of VR is its usability. Usability means that the people who use the product can do so quickly and easily to accomplish their own tasks [22].

While VR devices must be designed for maximal efficiency, most developers are constrained to use certain hardware and focus on finding the most natural and/or "performing" interaction methods using that hardware. Interaction design typically starts with task analysis, breaking down a high level task into number of subtasks until they are "primitive" enough for a simple mapping to a metaphoric object or physical hardware, or "generic" enough to apply to established interaction techniques.

Jacob et al. [23] proposed a fine grained software model and a language for describing and programming non-WIMP (Window, Icon, Menu, Pointer) style user interactions. Jocob's model expresses user interactions as a combination of a graph of functional relationships among continuous variables (e.g. input devices) and a set of discrete event handlers. Marigold's specification formalism is also designed with the notion of combining the continuous data stream (from devices) and generating meaningful discrete events for user interaction.

Major obstacles in efficient VR system design include such factors as rapid changes in hardware capabilities, availability, cost, and the absence of a mature methodology in interaction design. Until an acceptable interaction design methodology emerges in the near future, a VR system design tool must consider and support a trial-and-error style of engineering.

3.4 APIs and Tools for Virtual Reality

In most cases, developers of real-time graphics or animation programs, for instance, still proceed by creating visual objects on computer-aided design (CAD) systems, then using low-level simulation programming constructs or libraries to add behavior [2][3][4][5][6][7]. Such software packages allow relative simple encoding of the functional/behavioral aspect of the virtual environment, by hiding and abstracting out low level details and providing easy-to-use API's, and usually support object-oriented programming, communication with popular VR devices, and the importation of various model file formats.

Authoring tools for VR systems (such as the World-Up from Sense8 [6], Lynx from Paradigm [5], Alice from Carnegie Mellon University [24]) generally lack behavior modeling flexibility. For example, though Alice supports behavior specification using Python script and pre-defined functions and World-Up and Lynx also provide functionalities of adding simple behaviors, developers want more powerful and flexible modeling constructs. In addition, because the major objective of Alice is to design interactive 3D graphics, it lacks support for VR specific features (3D multimodal interaction, rigorous performance issues and presence). Above all, because authoring tools tend to abstract too many details, performance tuning and addressing presence is very difficult with these tools. Many developers still prefer to merely use the API in a creative manner, or even use very low level graphic system package (e.g. OpenGL, DirectX) in order to apply various optimization tricks.

In the area of entertainment, many 3D gaming engines have also adopted the scripting approach [25][26]. Scripts are usually associated with an interactive environment in with it is executed. Such interactive programming saves a lot of time and labor. We can confirm the potential utility of such an interactive kernel approach from the great debate about LISP vs. C as a prototyping language [27], but more so from the proliferation of many visual RAD (Rapid Application Development) tools for 2D WIMP user-interfaces and game prototyping examples. Similar approaches in the area of virtual reality application have also been tried. In [28][29], users can model 3D geometric objects and arrange objects in virtual environments using 3D interaction techniques. PIP system [30] used "programming-by-demonstration" approach [31] for modeling behaviors of VR objects within the virtual environment. However, the PIP system lack constructs for

expressing more complex behavior and VR specific functionalities.

4. CLEVE Methodology

The work described in this paper originated from a effort to establish a systematic engineering approach for VR systems called CLEVER (Concurrent and LEvel by level development of VR system) [1], that tackles the problems stated in Section 2.

Our methodology includes a comprehensive collection of conventional and new concepts. For instance, we employ concepts such as the simultaneous consideration of form, function, and behavior, hierarchical modeling and top-down creation of LOD (Level of Detail) [32], incremental execution and performance tuning, user task and interaction modeling, and compositional reuse of VR objects [33]. The basic modeling approach is to design VR objects (and the scenes they compose) hierarchically and incrementally, considering their realism, presence, behavioral correctness, performance, and even usability in a spiral manner.

The three philosophies that underlie the CLEVR are (1) concurrent consideration of form, function, and behavior, (2) hierarchical and incremental development (exploration), and (3) the spiral development process that addresses performance, interaction modeling, and presence enhancements in turn.

One of the main difficulties in VR system development lies in the management of complexity. For instance, the application developer must "design" three things that are interrelated, namely "form," "function," and "behavior" at the same time. Most VR applications are developed in the sequence of form design, followed by function/behavior programming. The result is an unstructured code in which information regarding function, behavior, and constraints among them are all mixed in together, not readily visible for easy future maintenance and reuse. Furthermore, construction of a virtual world often requires many revisions, and changing one aspect of the world will undoubtedly affect other aspects. Such a development cycle is difficult to handle when working in a single level of abstraction and considering these design spaces in isolation. A clear conceptual and computational separation among form, function, and behavior helps the user explore and evolve an object by considering them in a concurrent fashion.

Since virtual environment construction is a design and explorative task, the usual hierarchical and incremental approach is even more fitting than for the conventional software that are more algorithmic in nature. Moreover, employing the hierarchical style in the incremental development process promotes a performance conscious design by forcing the developer to focus on the more critical features in the form, function/behavior in a top-down manner. Intermediate models obtained from the hierarchical modeling approach can be used for LODs as well. At each abstraction level, in order to validate the models (e.g. behavioral correctness, visual effect, performance) and carry out further refinements, they must be simulated or executed.

5. An Interactive Authoring Tool: PVoT

We have developed a computer-aided tool called "PVoT (Portable Virtual reality system development Tool)" to solve the problems stated in Section 2 and support the modeling strategy presented in Section 4 (see Fig. 1).

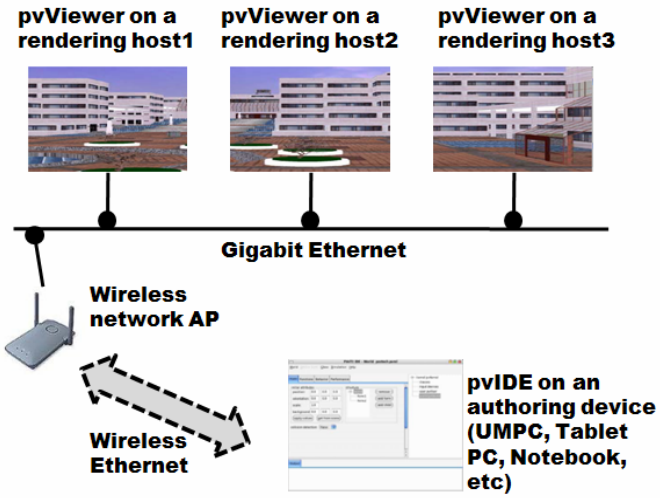

Fig. 1 PVoT in use. (Three pvViewers are running on an immersive VR platform composed of three separate rendering hosts while pvIDE is running on a portable device.)

The major required functionalities of the tools are: ways to construct and represent information regarding the form, function, and behavior of VR objects including input/output device configuration, user-interactions, and coalescing the information into an interactively executable form. The next step is to execute these evolving specifications and collect performance data for validating the behavioral correctness and tuning the resulting presence, usability, and performance, and to save and organize different alternative model configurations for experimentations.

To enable the above functionalities, we proposed an interactive kernel approach and PVoT object model and implemented them into an integrated development environment (IDE) with additional capabilities such as specification constructs, a performance monitor, and

markup languages for VR contents and device configurations.

5.1 The Overview of PVoT

PVoT is comprised of three programs (which are pvIDE, pvDaemon, and pvViewer) and many components (See Figure 2) and developed using two programming languages (C++, Python) and many open-source libraries such as OpenSceneGraph [7], VRPN [34], Open Dynamics Engine (ODE) [35], and Boost.Python [36].

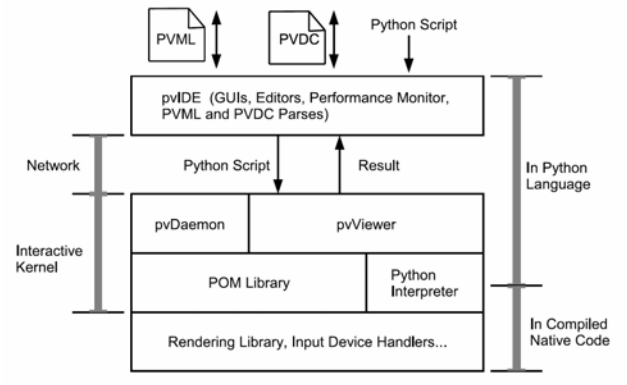

Fig. 2 Overall structure of PVoT.

pvIDE is an integrated development environment including easy-to-use graphical user interfaces (GUIs), script editors, a performance monitor, and parsers for two XML-based markup languages (PVML and PVDC). We design and validate VR contents by using these tools or by typing up a Python script and loading it. The resulting contents and device configurations use PVML and PVDC as their file formats respectively. PVML (PVoT Markup Language) expresses only VR contents themselves independent of hardware-related resources. PVDC (PVoT Device Configuration) conveys configuration information about display systems (e.g. HMDs, CAVE-like systems, Immersive displays) and various input devices including trackers, controllers, and joysticks.

pvDaemon is a separate program, which receives a request to start pvViewer and information about display configuration (such as the type of stereo, the position and size of window, and the parameters for camera position and orientation) from pvIDE and invokes pvViewer. From then, pvIDE and pvViewer directly communicate with each other. pvIDE translates user's commands and contents of PVML and PVDC files into Python script and sends it to pvViewer through the network. Note that the above tools are not just a development environment, but also an execution environment of the same virtual world.

5.2 Interactive VR Kernel

Through the interactive kernel, developers can interactively build virtual objects and deploy them in the execution environment directly in an interpretive way using a script language (we chose "Python") without compilation and optimize the content along the dimensions of performance, interaction usability, realism, and presence.

Although this interpretation method is slower than compilation methods, the overall performance drop is minimized by various optimization techniques [37], one of the well-known rules about this approach is the "90/10 rule in standard code profiling" [38], which means 90% of the execution time is spent by 10% of the code. As an analogy, contrary to many people's belief, LISP programs (that are usually developed and run in an interpretive environment) can run as fast as their C counterparts.

Thus, for instance, we wrote the most time critical part of the kernel such as the scene traversal, behavior simulation, and collision detection in C++ and provided abstraction through the Python language. In addition, the performance drop attributed to using languages is no longer significant with the multi-Giga-hertz processors available nowadays.

5.3 PVoT Object Model: POM

The core of the PVoT's interactive kernel is a flexible VR object model called "POM (PVoT Object Model)," that clearly segregates the aspects of form, function, and behavior. As stated in the previous subsection, POM library is implemented using two programming languages (C++ and Python). The C++ layer implements the most critical part of POM. In other words, most of the form hierarchy, basic functions, and Statecharts simulator of POM classes are coded in C++. The Python layers only wraps the interfaces (such as class name, member functions, and data members) of these classes written in C++.

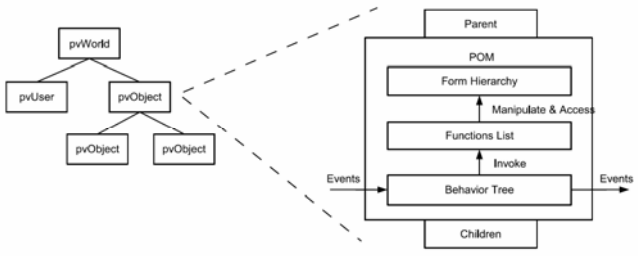

Fig. 3 An example scene graph and the structure of a POM object.

In the interactive kernel, all the objects are POM objects and even the overall world is also a POM object. All VR objects are instances of *pvObject* which is the

topmost class among the POM classes, and the world is an instance of *pvWorld*, which is a subclass of *pvObject* (see Fig. 3). Users can create (or reuse) new (or old) POM object by instantiating this topmost class *pvObject* (or extending it).

As you can see in Fig. 3, a POM object has its own form, function, and behavior and exchanges events with other objects. Each function does its own primitive tasks by manipulating and accessing the attributes of the same object's form. Behavior invokes functions of the same object according to its active states and transitions. As a result, the specification of each aspect (form, function, and behavior) of an object is dependent on others (their representations are clearly segregated, however). These clear representations of dependent relationships among the three aspects enable developers to easily detect any inconsistency.

Moreover, this high-level abstraction of VR objects leads to higher reusability in comparison to the conventional programming method. When reusing existing POM object, we just care about what events the objects need or generate. Currently, in PVoT, we can export and reuse a POM object or a subtree (a hierarchy consisting of multiple POM objects).

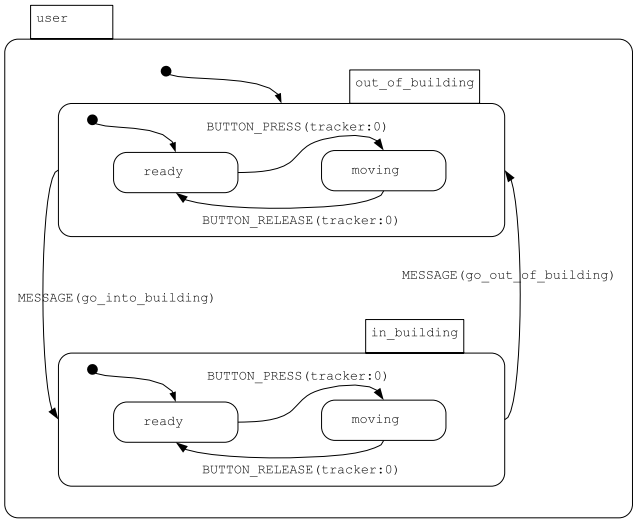

Fig. 4 An example of Statecharts. (This diagram shows an example modeling of an interaction technique.)

5.3.1 Form

The current version of PVoT does not include geometric modeling capability. This is because, as for form construction, many VR-based CAD systems have been developed [28][29] which are equivalent to addressing interactive form construction. In this current version, several primitive geometries (such as cube, sphere, cone, and cylinder) are available for direct creation, and a variety of geometry file formats can be imported.

5.3.2 Function

The *pvObject* (the topmost class of POM class hierarchy) provide predefined primitive and useful functions (or methods) to save development time (similarly to Alice [24]). These primitive functions include manipulation of the form attributes of the object, basic animation routines (e.g. move and turn), construction of a form hierarchy, a scene graph, and a behavior true, and generation of events.

We specify the behavior part of objects using the pre-defined functions and properties. However, before inserting scripting code into state specifications of behavior, we can immediately see the execution results of functions and effects of modified properties within the virtual world in construction.

Thanks to the dynamic binding feature of the Python language, developers can write their own functions by composing these primitive functions (or from scratch). When writing new functions, it is also possible to utilize rich standard libraries that Python already provides. These user-defined functions can be added to the existing objects or classes, and also replace existing functions.

5.3.3 Behavior

The behavior part of an object can be specified using Statecharts formalism (see Fig. 4), and it manages the life-cycle of an object from its construction to destruction. For a more detailed explanation of their exact formalisms, please refer to [39].

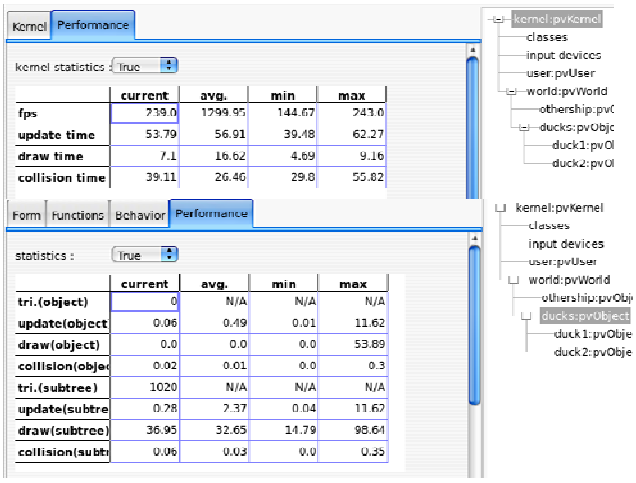

Fig.5 Performance statistics.

5.4 Performance Monitor

The "Performance" tab of pvIDE displays various performance statistics according to the currently selected object (see Fig. 5). By inspecting this report, the source of bottleneck problems can be identified. Then, the overall performance can be improved by solving the problems. Besides, we can find out which component of the system might be more refined to increase presence or usability, not bringing about performance degradation.

In PVoT, the interactive kernel controls the overall system. The kernel consists of "classes" (*pvClasses*, which are reusable POM objects), "user" (*pvUser*), and "world" (*pvWorld*, that is, the root node of the scene graph). When the kernel is selected, the performance tab displays the overall performance statistics which include frames per seconds, update (simulation) time, draw time, and collision detection time. When the others are selected, on the other hand, more detailed high-level statistics are given in the performance tab.

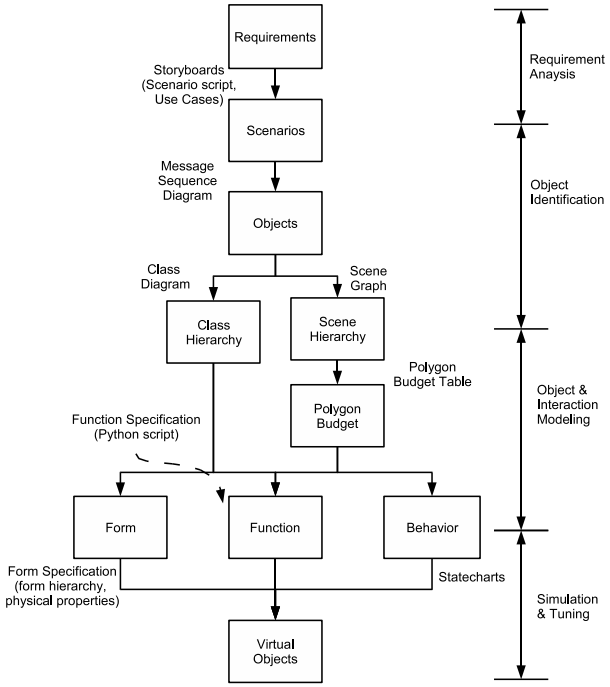

Fig.6 The overall development process in CLEVR/PVoT. (From the beginning or mid-point of this process is iterated at each stage in the spiral process, which is the third philosophy of CLEVR.

For example, as you can see in the lower table in Fig. 5, when an object is selected, statistics about 8 items are displayed. Through the above statistics and the execution results, the levels of the three major requirements (presence, usability, and performance) could be estimated. Then, we can proceed to tuning the three requirements. When we have to improve performance (that is, the current level of performance is not satisfactory), we can decide which should be considered to be simplified between the current node and its child nodes by comparing two groups of data.

Though there have been some research studies quantifying the benefits of objects [11][12], these benefit models are not appropriate to apply to our research, because a complete benefit model should consider many subjective factors that presence and usability have.

6. Development Process in CLEVR/PVoT

Fig. 6 shows the whole development process applying our methodology and tools to the design of VR content. The major part of system specification in CLEVR/PVoT consists of scenarios, sequence diagrams, scene graphs, class diagrams, and explicit representation constructs of form, function, and behavior of objects that appear in the requirements.

7. Example Applications

This section demonstrates our work by illustrating three example applications, which have been built using our approach and tools. They show the effectiveness of our method in terms of the quality of the resulting VR contents and reduced effort in their development and maintenance.

7.1 Ship Simulator

Fig.7 A snapshot of the ship simulator built and tested using PVoT.

The major purpose of the "Ship Simulator" example application is to assist trainee to navigate in and out of the pier and come to an anchorage without colliding with other vessels or the coast.

The specification of the Ship Simulator, according to the spiral process of CLEVR, will proceed first by listing the basic requirements, sketching the rough scenario, identifying the important objects and their relationships, and designing their basic functionalities (first stage). Then additional requirements such as interaction design, presence enhancement, and performance tuning are dealt with during the next stages of specification.

For the example of an object "*Autoship*" (which is navigating automatically and change their orientation and speed when some conditions are met), at the first stage, we concentrated on the basic and essential functionalities of objects. Fig. 8 shows the behavior specification of the automatically navigating *AutoShips*.

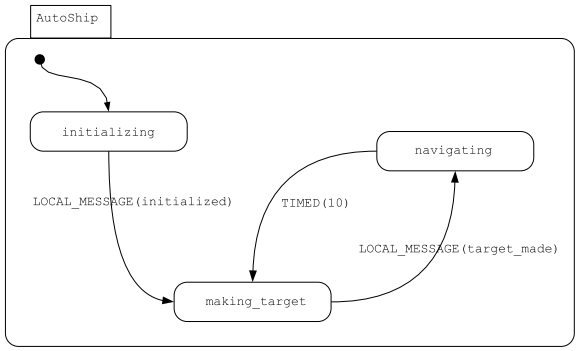

Fig.8 The first Statecharts of *AutoShip*.

Table 1: Simulation results with the 4 first-level *AutoShips*.

frame rate	63.07 frames/sec
updating time	26.21 %
Drawing time	72.69 %
Collision checking time	1,1 %

At this stage, we can run a simple simulation to estimate the performance of the system and validate the basic requirements. Before fixing the distribution and the complexity of *Autoships*, we arbitrary choose four as the number of *Autoships* in this estimation. Table 1 shows the four performance results in terms of the frame rate and the ratio of each of updating time, drawing time and collision checking time in a frame time. Though the drawing time takes up the most portions (72.69%) in a frame time, the overall performance is very fast (63.07 frame/sec). We can now decide whether to refine the behavior or form of objects or increase the number of *Autoships* in order to enhance presence (or usability), because performance degradation caused by refinement or changes of distribution of objects would be permissible to some extent. In the next stage, we decide to refine the behavior of *AutoShip* before adjusting the number of them.

Now, we are ready to further refine the behavior of *Autoships* interactively as we observer the overall scene. The major new features added are more detailed movement (pitching and rolling) of vessels and collision detection. In this refined specification (Fig. 9), *AutoShips* moves back for 5 seconds when they go out of the boundaries of the world model or when they collide with other vessels.

At this stage, we can run another simulation to estimate the performance of the system using various distributions of *AutoShips*. (We can postulate that, because there are relatively many *AutoShips*, it might dominate the load on the processor and the graphics board.) Table 2 shows the performance results of three cases (the number of *AutoShips* are 4, 8, 12). Based on these results, we can decide the proper number of *Autoships* by considering the requirements.

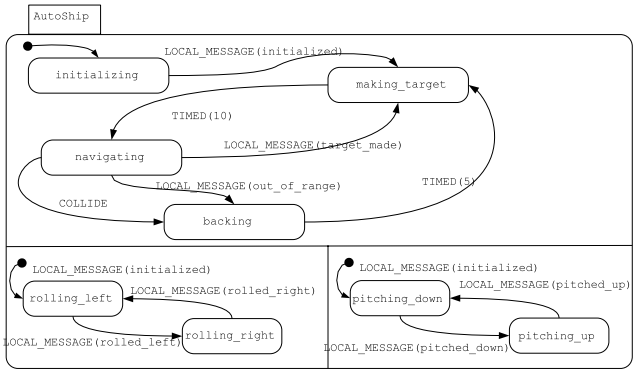

Fig.9 The second (refined) Statecharts of *AutoShips*.

Table 2: Simulation results with various distributions of the second-level *AutoShips*.

	4 ships	8 ships	12 ships
frame rate (frames/sec)	43.88	29.9	19.24
updating time (%)	49.43	71.69	56.05
drawing time (%)	50.41	7.04	8.95
collision checking (%)	0.16	21.26	35.0

7.2 Ubiquitous Computing Environments Simulator

In this application, we proposed to use PVoT to quickly build a virtual ubiquitous computing environment [40]. We have created four sets of components in PVoT for this purpose: (1) sensors, (2) displays, (3) processing objects, and (4) data collection objects. The first three types of objects are used for quickly prototyping and planting ubiquitous computing subsystem in the virtual environment. They are easily reusable within PVoT and can be adapted as needed; its function or behavior can be altered easily at different levels of abstraction. The data collection objects are used for collecting usability data such as performance measurements and survey answers.

Such a virtual ubiquitous computing environment can be effectively used for system architects to walk through and present various scenarios, to validate interaction usability and simply as software testing platforms.

7.3 Learning VR

In this application, PVoT has been used in the "Introduction to Virtual Reality" course three times since spring 2005. Most students had basic understanding of

how to program, but had little or no experience in computer graphics, 3D programming API's, dealing with various sensors, devices, and displays.

PVoT helped students interactively try out and explore different virtual object/scene configurations and immediately see their impact with respect to system performance, interaction usability, realism, and presence. PVoT is designed at an abstraction level appropriate for even non-computer science major students to quickly learn and understand the need of a structured development approach. Having learned the merits of the structured approach firsthand, the students effectively put it to use in the second stage of the course for implementing a more sophisticated class project. For a more detailed report of our experience, please refer to [41].

8. Conclusion

The objective of this research was to solve the three major problems inherent to developing VR content. Which are:

1. The difficulty to efficiently maintain the three important requirements of virtual environments, presence, usability, and performance.
2. It is complex to design, implement, and validate VR object while considering their interrelated three aspects form, function, and behavior.
3. The physical/logical gap between development environments and execution environments bring about significant time and effort.

To solve the above problems, we have presented a comprehensive structured methodology for building VR systems, called "CLEVR", and an integrated development tool, called "PVoT." The underlying modeling philosophies of CLEVR are the concurrent consideration of form, function, and behavior, hierarchical and incremental development and simulation, and spiral modeling process. The major feature of PVoT was the interactive kernel for explorative virtual world construction. Developers can try out and explore different configurations of the constituents of the virtual environment and immediately see their impact (within the virtual world in construction) and optimize the content along the dimensions of performance, interaction usability, realism, and presence. The kernel is based on a flexible object model (POM), and when used appropriately with off-line mark up specifications like PVML and PVDC, it can help fine tune the virtual world in construction fast. The POM reduces the inherent complexity of VR objects by providing an integrated and high-level object model for VR objects and formalizing behavior through the use of Statecharts. The POM also provides high reusability of VR objects and each aspect of an object.

It would be quite difficult to quantitatively show the utility or effectiveness of the proposed methodology and tools in terms of reduced cost in development and maintenance, due to the lack of objective metrics and the qualitative and even fuzzy nature of most software engineering studies. In addition, applying a structured design method makes sense when the target system is relatively large and highly complex. Enriching VR contents often amounts to increasing the number of objects, and detailing their behavior and interactions. Accurate (computational, storage and graphics subsystem) cost models would be required to predict how the system will behave when a system scale up occurs.

We claim that a structured and interactive approach such as CLEVR/PVoT enables and effective exploration of various alternatives for a domain with fairly large design possibilities such as VR systems (e.g. with respect to performance, usability, and presence). Without the exact model of what constitutes a "highly present" VR content, choosing the final system configuration is basically a trial-and-error process, dependent on the perception of the system developer, thus always subject to change and further tuning. It goes without saying that simply leaving at least rough specifications of the VR objects and their organizations, documenting the functional requirements and rationale of the design choices would prove to be beneficial in later maintenance. Moreover, the incremental and interactive validation of the model is likely to reduce the cycles of the usual trial-and-error process of system tuning for the required performance and usability and change in content and interaction methods.

References

[1] Seo, J., Kim, G. J., "Design for presence: a structured approach to virtual reality system design," Presence: Teleoperators and Virtual Environemnts, vol 11, no 4, 378-403, 2002.
[2] Wernecke, J., The Inventor Mentor: Programming Object-Oriented 3D Grahpics with Open Inventor, Addison-Wesley, 1994.
[3] Silicon Graphics, Inc, SGI – OpenGL Performer, http://www.sgi.com/products/software/performer
[4] Sowizral, H., Rushforth, K., Deering, M., The Java 3D API Specification, Addison Wesley, 2000.
[5] MultiGen-Paradigm, Vega Prime, http://www.multigen.com/products/runtime/vega_prime
[6] Sense8: SENSE8 Virtual Reality 3D Software, http://sense8.sierraweb.net
[7] OpenSceneGraph 3D Graphics Toolkit, http://www.openscenegraph.org
[8] Kim, G. J., Kang, K. C., Kim, H., Lee, J., "Software engineering of virtual worlds," Proceedings of the ACM Symposium on Virtual Reality Software and Technology, 131-139, 1998.
[9] Willians, J., Harrison, M., "A toolset supported approach for designing and testing virtual environment interaction techniques," International Journal of Human Computer Studies, vol 55, no 2, 145-165, 2000.

[10] Hubbold, R., Cook, J., Keates, M., Gibson, S., Howard, T., Murta, A., West, A., Pettifer, S., "GNU/MAVERIK: A micro-kernel for largescale virtual environments," Proceedings of ACM Symposium on Virtual Reality Software and Technology, 66-73, 1999.
[11] Funkhouser, T. A., Séquin, C. H., "Adaptive display algorithm for interactive frame rates during visualization of complex virtual environments," Proceedings of the 20th annual conference on computer graphics and interactive techniques, 247-254, 1993.
[12] Hoppe, H., "Progressive meshes," Proceedings of the 23rd annual conference on computer graphics and interactive techniques, 99-108, 1996.
[13] Luebke, D., Erikson, C., "View-dependent simplification of arbitrary polygonal environments," Proceedings of the 24th annual conference on computer graphics and interactive techniques, 199-208, 1997.
[14] Garland, M., "Quadric-based polygonal surface simplification," PhD Thesis, Carnegie Mellon University, 1999.
[15] Cohen-Or, D., Chrysanthou, Y., Silvia, C. T., "A survey of visibility for walkthrough applications," SIGGRAPH 2001 Course Notes on Visibility, problems, techniques and applications, 44-65, 2001.
[16] Carlson, D. A., Hodgins, J. K., "Simulation levels of detail for real-time animations," Proceedings of Graphics Interface, 1-8, 1997.
[17] Slater, M., Usoh, M., Steed, A., "Depth of presence in virtual environments," Presence: Teleoperators and Virtual Environments, vol 3, no 2, 130-144, 1994.
[18] Hendrix, C., Barfield, W., "Presence within virtual environments as a function of visual display parameters," Presence: Teleoperators and Virtual Environments, vol 5, no 3, 274-289, 1996.
[19] Hendrix, C., Barfield, W., "The sense of presence within auditory virtual environments," Presence: Teleoperators and Virtual Environments, vol 5, no 3, 290-301, 1996.
[20] Welch, R., Blackmon, T. T., Liu, A., Mellers, B. A., Stark, L. W., "The effects of pictorial realism, delay of visual feedback, and observer interactivity on the subjective sense of presence," Presence: Teleoperators and Virtual Environments, vol 3, no 2, 263-273, 1996.
[21] Shim, W., Kim, G. J., "Designing for presence: the case of the virtual fish tank," Presence: Teleoperators and Virtual Environments, vol 12, no 4, 374-386, 2003.
[22] Dumas, J. S., Redish, J. C., "A practical guides to usability testing," Intellect, UK, 1999.
[23] Jacob, R., Deligiannidis, L., Morrison, S., "A software model and specification language for Non-WIMP user interface," ACM Transactions on Computer-Human Interaction, vol 6, no 1, 1-46, 1999.
[24] Carnegie Mellon University, Alice: Free, Easy, Interactive 3D Graphics for the WWW, http://www.alice.org
[25] Conitec Datensysteme GmbH: 3D GameStudio /A6, http://conitec.net/a4info.htm
[26] Tyberghein, J., Crystal Space 3D, http://crystal.sourceforge.net
[27] Gat, E., "Lisp as an alternative to Java," Intelligence, vol 11, no 4, 21-24, 2000.
[28] Bowman, D. A., Hodges, L., "User interface constraints for immersive virtual environment applications," Graphics, Visualization, and Usability Center, Georgia Institute of Technology, GIT-GVU-95-26, 1995.
[29] Mine, M. R., "A virtual environment for the interactive construction of virtual worlds," Department of Computer Science, UNC Chapel Hill, CSTR95-020, 1995.
[30] Lee, G. A., "Modeling virtual object behavior within virtual environments, M.S. Thesis, Pohang University of Science and Technology, 2002.
[31] Cypher, A., Watch what I do: programming by demonstration, MIT Press, 1993.
[32] Seo, J., "Top-down specification of LOD(Levels-of-Detail) models l for virtual objects," M.S. Thesis, Pohang University of Science and Technology, 2000.
[33] Seo. J., Kim, D., Kim, G. J., "VR object reuse through component combination," Proceedings Structured Design of Virtual Environments and 3D-Components, 2002.
[34] Taylor II, R. M., Hudson, T. C., Seeger, A., Weber, H., Juliano, J., Helser, A. T., "VRPN: A device-independent, network-transparent VR peripheral system," Proceedings of the ACM Symposium on Virtual Reality Software and Technology, 2001.
[35] Smith, R., Open Dynamics Engine, http://www.ode.org
[36] Abrahams, D., Building Hybrid Systems with Boost.Python, http://www.boost-consulting.com/writing/bpl.html
[37] Dawson, B., "Game Scripting in Python," Game Developers Conference Proceedings, 2002.
[38] Patterson, D. A., Hennessy, J. L., Computer Architecture a Quantitative Approach, Morgan Kaufmann Publishers, 1996.
[39] Harel, D., "STATEMATE: A working environment for the development of complex reactive systems," IEEE Transactions on Software Engineering, vol 16, no 4, 403-414, 1900.
[40] Seo, J., Goh, G., Kim, G. J., "Creating ubiquitous computing simulators using P-VoT," Proceedings of the Fourth International Conference on Mobile and Ubiquitous Multimedia (MUM2005), 123-126, 2005.
[41] Seo, J., Kim, G. J., "Teaching structured development of virtual reality systems using P-VoT," Edutainment 2007 (to appear), 2007.

Jinseok Seo received the BS degree in Computer Science and Eng. from Konkuk Univ. in 1998. He received the MS and PhD degrees in Computer Science and Eng. from POSTECH in 2000 and 2005, respectively. He is now a faculty member at Dong-eui Univ., Busan, Korea. His research interests include game software, entertainment computing, e-learning, virtual reality, augmented reality, and authoring methodology/tools.

Sei-woong Oh received the BS and MS degrees in Electronics Eng. from Hanyang Univ. in 1985 and 1987, respectively, and received the PhD degree in Information Eng. from Osaka Univ. in 1998. He is now a faculty member at Dong-eui Univ., Busan, Korea. His research interests include networked virtual reality, network security, on-line game, home network, embedded software, and ubiquitous computing.