DEVSView: A tool for visualizing CD++ simulation models

Wilson Venhola Gabriel Wainer

Dept. of Systems and Computer Engineering
Carleton University
4456 Mackenzie Building
1125 Colonel By Drive
Ottawa, ON. K1S 5B6. Canada.

wvenhola@connect.carleton.ca **gwainer@sce.carleton.ca**

Abstract: We present an application to visualize simulation results executed using the CD++ modeling and simulation toolkit. DEVSView allows users to create visualizations from the simulation log files outputted by the CD++ toolkit. DEVSView has implicit support for Cell models and uses OpenGL and the OpenGL Utility Toolkit for hardware accelerated rendering. DEVSView provides a graphical user interface and a text file format for the creation of visualizations. visualizations, in DEVSView, consist of visual models that translate CD++ log files into animations. Each visual model corresponds directly to an *atomic* or *coupled* model from a CD++ simulation. These visual models contain visual states and event animations which are used to represent the simulation graphically. The user can set up the rules, to trigger state changes and event animations, within the GUI or in the visualization file, and the user can use the GUI to playback the visualization.

Keywords: DEVS, Cell-DEVS, simulation visualization, CD++, OpenGL

1. INTRODUCTION

In recent years, the Discrete Event Systems Specification (DEVS) formalism [1] has gained popularity to model a variety of problems. DEVS is a framework for the construction of discrete-event hierarchical modular models, in which behavioral models (**atomic**) can be integrated together forming a hierarchical structural model (**coupled**). The Cell-DEVS formalism [2] extended the DEVS formalism allowing the simulation of discrete-event cellular models. The CD++ toolkit [3] allows implementing DEVS and Cell-DEVS models while providing remote access to a high performance DEVS simulation server [4]. The end user tools were organized as a simulation client applied to the CD++ simulator. Using these facilities, the users can now develop and test their models in local workstations, and submit them to be simulated in a remote CD++ server executing in a high performance platform. Then, they can receive, visualize and analyze the results on a local computer, improving model definition and execution. These simulations produce complicated results, and can depict interactions that occur in three dimensions. These results sometimes require extensive interpretation and reconstruction to clearly see what is occurring during the simulation.

Originally, CD++ only provided results on text files, making it difficult to study execution results of the model. visualization tools are crucial in helping to understand better the behavior of the system of interest, thus, different visualization facilities were incorporated [4]. A 3D GUI built using VRML enabled sophisticated visualization of Cell-DEVS models. Unfortunately, VRML technology became obsolete, and applets use Java3D libraries are no longer actively developed. Hence, we have recently focused on new extensions that can be applied to both DEVS and Cell-DEVS, and which are able to run on OpenGL-based environments [5], including a new interface [6], and based on Maya [7]. Alias Maya is an excellent tool for creating environments and objects to visualize simulations; however the installation size, workstation requirements, and licensing issues of the Maya software prevent it from being the optimal viewer.

Here, we describe the desing and implementation of DEVSview, a visualization tool developed to improve the available options for visualizing DEVS simulations executed in the CD++ toolkit environment. Although all CD++ simulations conform to the DEVS specifications, the results they produce often require different interpretation. For example, some simulations output values over a continuous range, while others may output a sequence of discrete states. Therefore visualizing simulation results requires a tool which provides a flexible methodology to visualize the various simulations appropriately. The proposed solution, the DEVSView visualization tool, provides several constructs to enable visualizing the results of DEVS simulations. The models from the simulation are directly translated to visual models. These visual models each contain a visual state transition system and an event animation creation system that allow the simulation to be visualized appropriately. DEVSView provides the graphical user interface to define and playback visualizations in three dimensions.

The DEVSView visualization tool provides basic services that enable simple visualizations. The following lists the services the tool provides:

1) Graphical user interface based on the OpenGL Utility Toolkit [8]: The windowing system provides buttons, text fields, list boxes, resizable windows, and other controls necessary for a GUI. The rendering of the controls is accelerated by OpenGL [9].
2) Visual state transition, and event animation systems: The visual state transition system is a collection of visual states and transition rules defining what simulation events trigger state changes. The event animation system is a collection of rules to define which events trigger certain animations.
3) Design and Implementation of an octtree scene database to enable efficient view culling: The visual models are stored in an octary space partitioning tree. This data structure recursively divides the scene extents into eight regions, which enables efficient algorithms for rendering scenes, object selection, and other frequently used scene operations.

The tool was implemented using C++ and OpenGL. OpenGL is supported by many platforms, and is actively developed and extended to accommodate the advancing field of computer graphics. GLUT provides simple windowing services, and does not reduce OpenGL rendering performance. This approach also produces a small installation size, and no licensing issues.

2. VISUALIZATION METHODOLOGY

Each DEVS simulation result consists of several atomic and/or coupled models communicating with each other over ports using messages, which represent events in the simulated system. The DEVSView tool provides a general method of mapping simulation results to a visual representation. The method and data used to map the results are called a *visualization* in DEVSView, which consists of a set of visual models, and a set of events that manipulate them. The set of events used in the visualization corresponds directly to the external and output events from a CD++ simulation log file. A visualization progresses by sending these events to the visualization models for processing. Events are sent to both the source model and destination model for this processing. The visual model's transition rules specify how an event affects the visual representation of the model, and the event animation creation rules specify whether an event produces certain event animations.

Cell visualization models extend the regular models by adding a three dimensional array of cells. The cells store their own current visual state, position, orientation and size; but they all use a common set of visual states, visual state transition rules, and event animation rules.

Both the visual state transition system and the event animation system described following operate on the events passed to visual models as the visualization progresses through simulation time. When the visualization reaches the time an event occurred during the simulation, it is processed by both models involved in the exchange. Each event contains information about the source/destination visual model name, the time of the event, the port the value is sent through, and the value of the event. The source and destination visual models use this information to process the event. Typically, this involves comparing the port and value with behavioural rules such as transition rules or event animation rules. These rules use the concept of a DEVSView Value rule to operate. A Value rule is a procedure which accepts a real value, typically the event value, and returns a Boolean indicating whether the value passes the rule or whether it fails.

The visual state transition system of the DEVSView tool assigns a simple state machine to each visualization model. The state machine consists of visual states, and transitions between these states, which are triggered by events in the simulation. The current state defines the visual appearance of the model in three dimensions. Transition rules specify when the model changes state during the visualization. A transition between states occurs when a transition rule is invoked. When an event is processed by the visual model, each of the transition rules for the current state are evaluated to check if any transitions should be invoked. As well as transition rules for the current state a separate list of transition rules which apply for all states, are checked. A transition rule is invoked when the transition rule port name and direction match the event port name and direction, and the value rule passes given the event value as input. The event animation system allows visual models to create animations which visualize the processing of certain events. Event animations can produce any sort of visual effect, and are triggered to occur when specific events arrive at a visual model. The only event animation currently provided by the tool is the text animation. A text animation is a three dimensional piece of text which travels from one location to another.

When an event is processed by the visual model, the event animation rules are evaluated to check if any event animations should be created. An animation is created if the current visual state equals the rule source state, the rule port name and direction match the event port name and direction, and the value rule passes given the event value as input. Event animation rules create animations and specify their properties based on the event value and other variables internal to the visual model.

3. OVERVIEW OF DEVSVIEW SYSTEM DESIGN

DEVSView was developed considering the visualization requirements briefly discussed in Section 1, providing the ability to display the output of a DEVS simulation in an appropriate graphical format. To achieve the goals presented earlier, DEVSView functionality was organized as two separate entities:

- A parser for CD++ log files: extracts DEVS models which may require visual models, and any events associated with the extracted visual models
- The User Interface: Specifies the graphical representation of the visual models, Provides controls for starting, stopping, and pausing the visualization. Provides speed control and seeking to a specific time in the visualization, and for loading and saving visualizations

A *Scene database* provides structure for organizing visual models efficiently in three dimensions. These components were organized following the Use Cases in Figure 1.

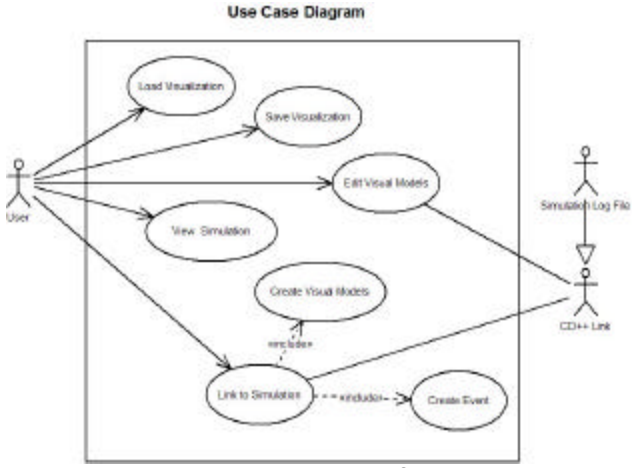

Figure 1: Use case diagram for DEVSView

The actors involved in the use cases are the *User*, and the *CD++ Link*. The *Simulation Log File* actor is a generalization of a *CD++ Link*. *CD++ Link* represents a general link to a CD++ simulation through whichever interfaces CD++ supports. The use cases demonstrate the capabilities of the user to initiate a link to the simulation through a log file, as well as view, edit, load, and save the visualizations.

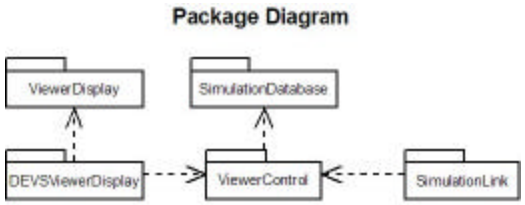

Figure 2: DEVSView package diagram

The *User*'s ability to manipulate the visual representation of the simulation is represented by the *Edit Visual Models* use case. The *CD++ Link* can interact with the visual models in the same manner when the *User* links to a simulation. Consequently, the system has been divided into the four conceptual packages shown in figure 2. The *DEVSViewerDisplay* package is responsible for processing input and converting it to commands for the *ViewerControl* package. The *DEVSViewDisplay* package also controls output to the application window for rendering the user interface and three dimensional graphics. This package depends on the services of the *ViewerDisplay* package for the event driven GUI functionality. The *SimulationLink* package is responsible for interacting with simulation results and reporting the necessary results to the *ViewerControl* package. The responsibilities of the *SimulationLink* include parsing simulation log files and notifying the *ViewerControl* package about new events and new visual models. The *ViewerControl* package processes the requests from both the *DEVSViewerDisplay* and *SimulationLink* packages. The Viewer Control interprets simple commands from both of these packages and then translates them into the proper sequence of interactions with the *SimulationDatabase* package. The SimulationDatabase package stores the information necessary for the visualization. The events, visual models, and all corresponding data are stored in the SimulationDatabase. The SimulationDatabase package also stores the visual models in an octtree data structure to cull objects efficiently. The packages communicate with each other by passing data types which are in the set of common interface types. The interface types allowed include 1) the standard C++ types 2) several basic structures for position and time information and 3) property sets which contain variables of any interface type, including other property sets. A detailed specification of these data structures can be found in the Appendix. The following sections will describe the internal design of each package. The following sections present a generic design of the application packages; detailed information about the design and the implementation of each of the packages can be found in [11].

4. PACKAGES DESIGN

The *ViewerDisplay* package provides a framework for developing graphical user interfaces on top of GLUT. The package has support for event driven programming using commonly required interface controls, such as buttons, text boxes, scroll bars, list boxes, message boxes, etc. Although this package has been developed solely for DEVSView, it is not dependant on the other packages and can be easily incorporated into another application which uses GLUT. The controls supplied by this package are all subclasses of the *VDWindow* class. Figure 3 contains an UML diagram, showing the several controls that subclass the *VDWindow* class and implement specific functionality. Note that any

VDWindow subclass can contain any number of other child windows. User interfaces are created by nesting controls, such as *VDButton*, *VDLabel*, *VDListBox*, etc, in a *VDPanel* class.

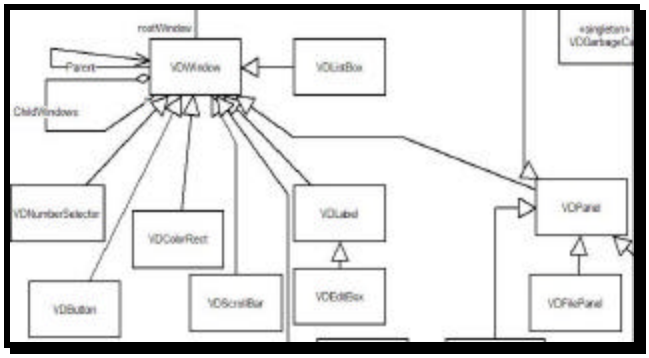

Figure 3: VDWindow class and various controls which subclass it.

Figure 4 shows the implementation of this class, which is represented as a panel with nested window controls, including text buttons, a number selector, and a scrollbar.

Figure 5: Visualization playback options panel from the DEVSView tool.

Each subclass of *VDWindow* is required to process GUI events. For example, the *VDTextButton* class processes button press events to determine when the button has been pressed. Figure 6 shows the class structure of the DEVSView event handling mechanism.

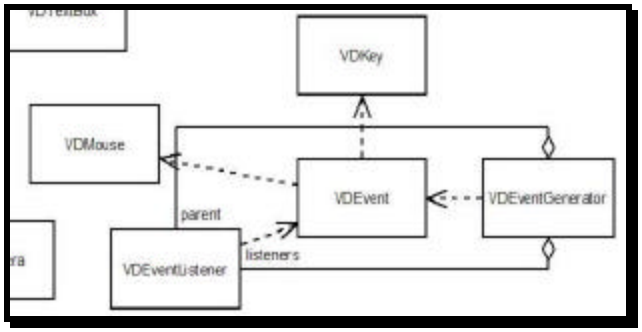

Figure 6: The DEVSView event classes.

As seen in Figure 6, *VDEventGenerator* objects send events up the hierarchy to the parent listener or down the hierarchy to the child listeners. *VDEvent* objects may hold mouse, key or other necessary information. There are many types of events which subclass *VDEvent* but are not shown here for space reasons (details can be found in [11]), including *VDMouseDownEvent*, *VDMouseUpEvent*, *VDMouseMoveEvent*, *VKeyDownEvent*, etc. An event is passed to the *ViewerDisplay* package by GLUT, and then it is processed by the root *VDWindow* object. The *VDWindow* class inherits from *VDEventListener* and *VDEventGenerator* so that any *VDWindow* object can receive and send events. When an event listener receives an event, it can forward the event to the child windows, process the event type and contents, and send a new event in response. Depending on the type of event created, it may travel up the window hierarchy to the parent window or travel down to the child windows. After processing the event, the *VDWindow* object returns a Boolean value to indicate whether the event has been consumed or not. If the event is consumed, the parent window will not pass the event to any other child windows.

DEVSViewerDisplay implements the user interface of DEVSView. This package contains several *ViewerDisplay* panels which provide the user interfaces required to interact with the user during each use case. The portion of the *DEVSViewerDisplay* class diagram shown in Figure 7 displays the structure of the various panels in the user interface.

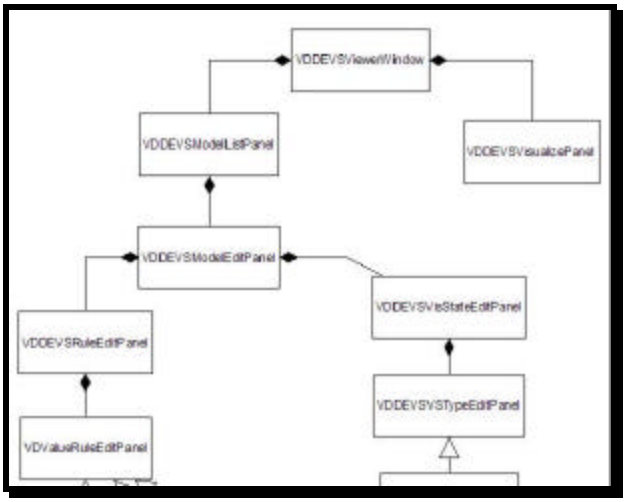

Figure 7: Structure of the DEVSView user interface.

The main viewer window contains the panel for controlling the visualization as well as the panel for editing the visual models (i.e. the VDDEVSModelListPanel). The *VDDEVSViewerWindow* contains the model list panel and the visualize panel. The *VDDEVSViewerWindow* also contains a panel with various controls to start various common tasks, and a command console. The console is used for logging errors, viewing debug/command information, and entering simple commands. The implementation of the toolbar panel and the console are shown in figure 8. As we can see in Figure 8, the main DEVSView window contains two panels.

The toolbar panel has commands for saving/loading/creating a visualization and showing/hiding other panels. The console can execute commands and display the results. The console also displays logging information from DEVSView.

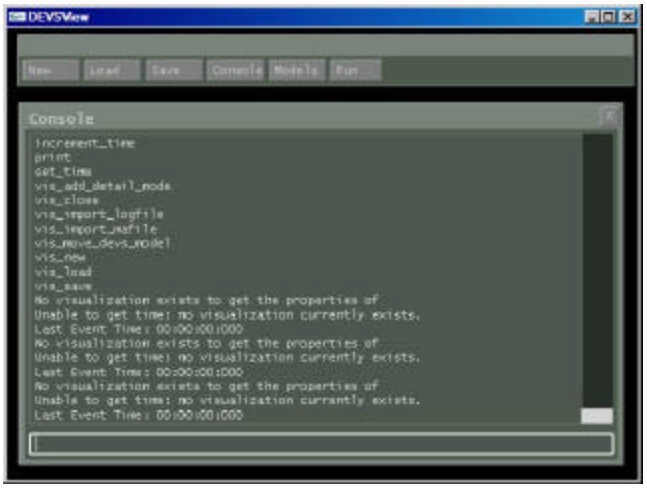

Figure 8: Main DEVSView window.

The *VDDEVSVisualizePanel* is in charge of implementing the controls to enable starting, stopping, pausing, seeking to a specific time (with the scrollbar), and slowing down or speeding up the visualization. *VDDEVSModelListPanel*, and *VDDEVSModelEditPanel* are shown in figure 9.

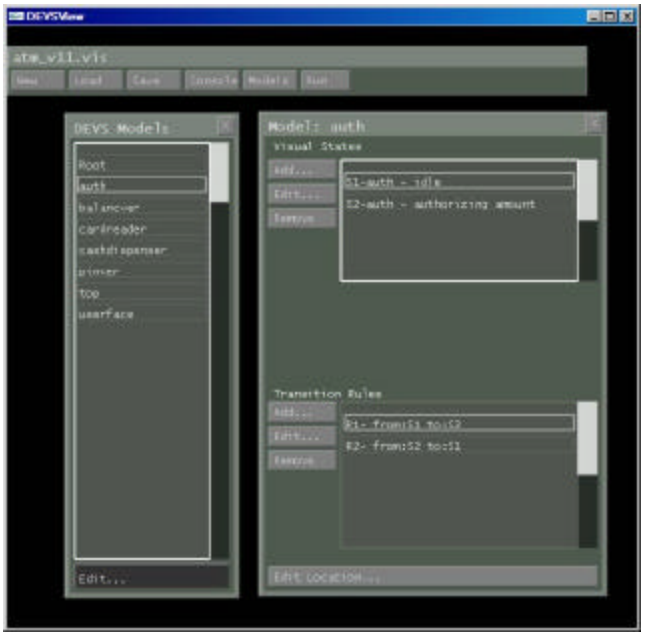

Figure 9: Model list and the model edit panels.

In the example presented on Figure 9, the model list panel shows the visual models of the visualization. The *auth* model is currently being edited. The visual states and the transition rules are shown in the model edit panel. States and rules can be added, edited and removed from the visual model using the model edit panel. Also the location of the model can be edited using the 'Edit Location…' button.

The *VDDEVSModelListPanel* is built from a *VDListBox* and a *VDTextButton*. The list box contains the list of visual models for the current visualization. The text button creates a *VDDEVSModelEditPanel*, which is created from list boxes and text buttons, for editing the currently selected visual model. The visual states and transition rules are managed from the model edit panel. The panel includes three controls 1) A Visual state type list 2) A Label, and 3) A Visual state properties panel. Depending on the type of the visual state selected, the appropriate properties panel will be shown. The transition rules are edited using the *VDDEVSRuleEditPanel*, in which the rule properties are selected from the list boxes shown. Depending on the value rule type selected (i.e. All Values, Equals Value, etc), the appropriate value rule panel will be shown.

For each visual state type and value rule type, there should be a corresponding properties panel that subclasses *VDDEVSVSTypeEditPanel* and *VDValueRuleEditPanel* respectively. These panels provide the required controls to define the properties of their intended objects. The *DEVSViewerDisplay* package eventually translates all requests from the GUI into commands for *ViewerControl*.

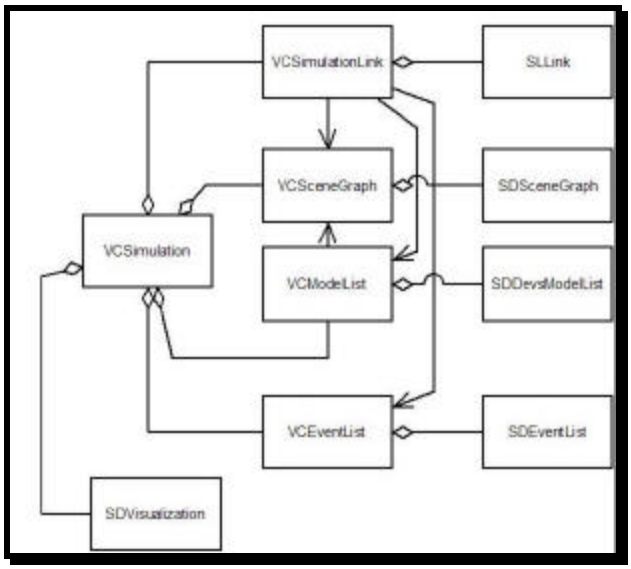

Figure 10: *ViewerControl* and *SimulationDatabase*.

The *ViewerControl* package is divided into classes which wrap a corresponding *SimulationDatabase* class. The requests to modify and/or use the simulation database are channeled through these classes and to the appropriate destination. Each request to the *ViewerControl* package is ini-

tially routed through the *VCSimulation* class, which translates the requests into the more detailed and complicated interactions with the database. The *ViewerControl* classes are shown, along with the *SimulationDatabase* classes they interact with, in figure 10.

As we can see, the *VCSimulation* class encapsulates the *SDvisualization* class as well as the other *ViewerControl* classes.The commands received from the *DEVSViewerDisplay* package are processed by the *VCSimulation* class. The *VCSimulation* class forwards these requests to the appropriate control classes or handles them directly. For example, a request to save the visualization is directly handled by the *VCSimulation* class, while a request to link to a simulation log file is forwarded to the *VCSimulationLink* class. Various navigable associations exist between the *VCSimulationLink* class and other *ViewerControl* classes so that visual models and events can be created and setup without calling the *VCSimulation* class methods. The *VCModelList* class has a navigable association to the *VCSceneGraph* class so that whenever a visual model is modified the appropriate changes to the scene graph can be made efficiently. The main responsibilities of the *ViewerControl* package are decoupling and simplifying the interface between the user interface and the visualization functionality.

The *SimulationLink* package implements the *SLBase* interface. This interface defines the minimum interface required to connect to the *ViewerControl* package and submit data for visualization. This interface requires for each medium that provides information about a DEVS simulation, a separate class which implements it. The classes which have been created for this package are shown in figure 11.

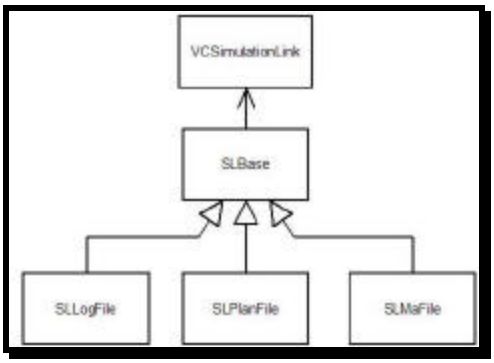

Figure 11: The *SimulationLink* classes.

The *SLBase* in Figure 11 defines the interface over which information is added to the visualization from the various simulation link types. The *SLLogFile* class loads the events and visual models from a CD++ log file, while the *SLMaFile* class loads cell models and their starting states (*SLPlanFile* class is reserved for future functionality).

SLBase notifies *VCSimulationLink* class about new events and new visual models which may be added to the visualization. The *VCSimulationLink* may choose to reject the models or events if they already exist, or if they do not contain valid information. In some instances, such as loading from a log file, model information is identified in pieces. For example, a CD++ log file is a record of events and each event has a source model, a destination model, and a port name, among other things. Any event indicates that the source model has an output port, and the destination model has an input port. The *SLLogFile* object, when reading an event, will notify the *VCSimulationLink* about the new visual models (the source model and the destination model). The *VCSimulationLink* is responsible for merging the provided visual models with the previously existing database. This usually involves adding new input ports and output ports to models identified in events. Other times it requires converting a DEVS model to a Cell-DEVS model or expanding the cell space of a Cell-DEVS model.

The *SimulationDatabase* package is divided into several singleton classes, shown in Figure 12.

Figure 12: Singletons of *SimulationDatabase* package.

These singletons provide the main structure for storing the data necessary for visualization. The *SDvisualization* object holds the information for the visual models. The *SDResourceList* holds the data used for rendering the visual models (fonts, geometry, etc). The *SDAnimationController* holds the current animations. The main singleton is the *SDvisualization* class, which contains the scene graph, visual model list, event list, current time, and scene node list. The octtree data structure is contained in the scene graph, and a visual state transition system is stored in each visual model of the model list. The animations currently active in the visualization are stored and animated by the *SDAnimationController* object. The *SDResourceList* object contains the resources used for rendering the visual models. The list contains 3d fonts, geometry, textures, and other resources which may be used by many different visual models.

The structural properties of the visual model classes are shown in figure 13. The *SDDevsModel* class encapsulates the information for each visual model. *SDCELLDevsModel* subclasses the *SDDevsModel* class to reuse its functionality. The *SDDevsModel* encapsulates the information representing a visual model. The *SDCELLDevsModel* is a subclass of *SDDevsModel* to reuse functionality. Each visual model contains a list of visual states, a list of transition rules, and a

list of event animation creation rules. The scene graph and scene node list implement the octtree database.

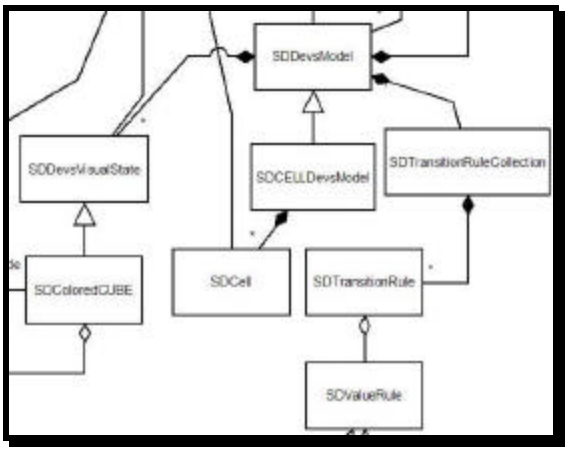

Figure 13: Visual model classes.

The classes corresponding to these components are shown in Figure 14. The *SDSceneGraph* contains the root *SDTreeNode*, which is the root node of the Octtree. The *SDNodeList* class contains a list of *SDSceneNodeInfo* objects which wrap scene node objects and store the locations of the scene node in the Octtree. Each *SDTreeNode* may contain many scene nodes. The *SDNodeList* object contains every *SDSceneNode* object, which is wrapped with a *SDSceneNodeInfo* object. The *SDSceneNodeInfo* class holds the locations of their scene node in the scene graph. When the *SDvisualization* object adds objects to the scene graph, the objects are wrapped with a *SDSceneNodeInfo* object and added to one or more *SDOctTreeNode* objects.

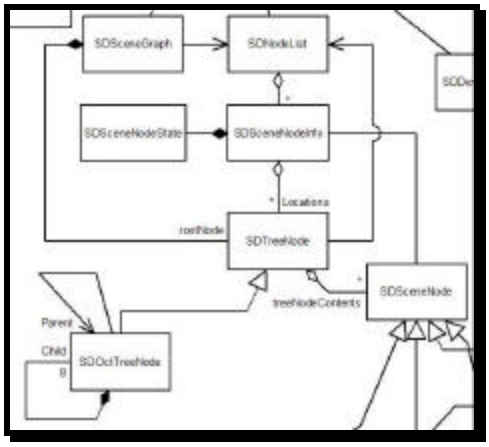

Figure 14: Scene database classes.

4. VISUALIZATION EXAMPLES

We used DEVSView to visualize different existing models. In this case, we show the results obtained when visualizing the results of a simulation of an ATM banking machine [12]. This model represents a simple ATM machine, described in the following figure:

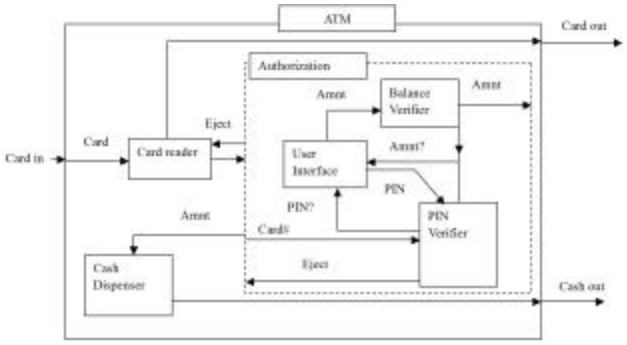

Figure 15: ATM DEVS model.

The components of this ATM and their functions are:

- Card Reader: accepts the input card into the machine and reads customer information. It also returns the card to the customer (end or failed transaction).
- Cash dispenser: responsible to dispense the required and approved amount of money to the customer.
- Authorization and balance verification module: receives a PIN, validates it, receives the amount to be dispensed, checks available funds, and sends approved amount to be dispensed. It is composed of three components:
  - User Interface: an interface with a customer who enters the PIN and amount of money to be withdrawn. These two values are generated with a uniform random variable here to simulate customer input.
  - PIN verifier: verifies the received PIN from the customer. It checks the received PIN and randomly decides if it is valid. If valid it returns to User Interface to get the amount, else it asks again for a new PIN.
  - Balance verifier: verifies that the customer has funds at his/her account to cover the required amount. If not, it would ask the user Interface for a new amount.

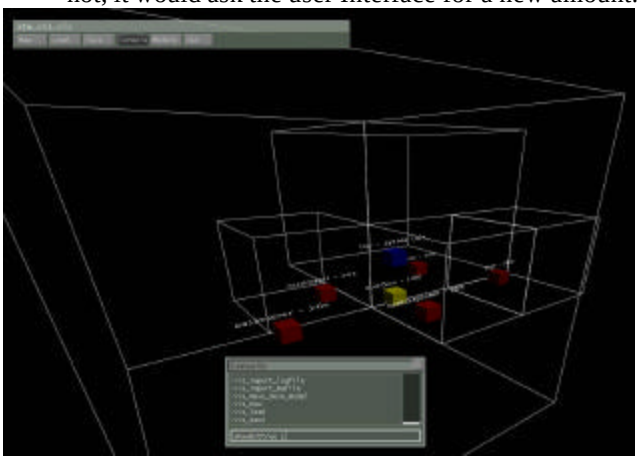

Figure 16: ATM DEVS model.

The ATM simulation also provides a text animation which displays “Card Inserted” whenever a bank card is inserted. The text animation can be seen in Figure 16, which shows a frame from the visualization right after a customer arrives at the ATM. The figure shows the visual models of the ATM visualization and the Octtree regions they were assigned. The following figure shows the simulation results is the specification of a model that represents an object in movement that bounces against the borders of a room. This example is ideal to illustrate the use of a non toroidal cellular model, where the cells of the border have different behavior to the rest of the cells. This visualization shows a simulation of three bouncing balls contained in a 2d grid. Figure 17 shows a composite of the visualization during playback. The image shows the motion of the balls in the 2d grid.

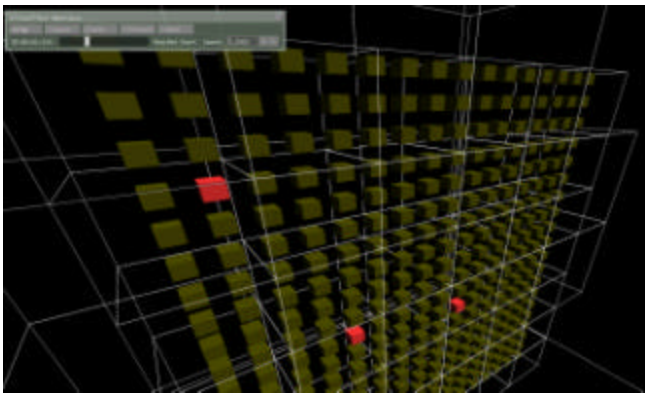

Figure 17: Composite frames. Bouncing ball simulation.

5. CONCLUSION

CD++ allows the simulation of complex physical based on the DEVS and Cell-DEVS. To facilitate the users to use the CD++ simulator, we extended its design to provide a number of services. The 3D visualization GUI enables sophisticated visualization to better understand the results. DEVSView provides facilities for creating visualizations of CD++ simulations, which are based on the Devs formalism. The tool reads CD++ simulation log files to create the visual models needed to visualize the simulation. The visual models have visual state transition systems which define how the simulation models are graphically represented during visualization. The visual models also have event animation rules to create animations when certain events occur. These constructs provide the methodology required to visualize Devs or Cell-DEVS models. The tool provides a user interface and file format to create these constructs, and several visualizations have been successfully created with DEVSView.

The current facilities have highly improved the use of the previously existing tools, thus enhancing the analysis experience of the modelers using the toolkit. The DEVSView tool provides facilities for creating visualizations within an open-source environment. The visual models have visual state transition systems, which define how the simulation models are graphically represented during visualization. The visual models also have event animation rules to create animations when certain events occur. Future work will include loading Maya model files for complex objects, and more advanced model positioning capabilities. DEVSView could also benefit from many user interface improvements. The visualization facilities of the DEVSView tool are quite basic, but provide the beginnings of a powerful tool. The tools are open source and can be found in http://www.sce.carleton.ca/faculty/wainer.

REFERENCES

[1] Bernard Zeigler, Tag G. Kim, Herbert Praehofer. Theory of Modeling and Simulation. *Academic Press.2000*

[2] Gabriel Wainer, Norbert Giambiasi: "Timed Cell-DEVS: modeling and simulation of cell spaces " Discrete Event Modeling & Simulation: Enabling Future Technologies. Springer-Verlag, 2001.

[3] Wainer, G. 2002. CD++: A toolkit to develop DEVS models.*Software,Practice and Experience*.32(3):1261-1306.

[4] Wainer G., Chen W. 2003. A Framework for Remote Execution and visualization of Cell-DEVS Models. *SIMULATION, Vol. 79, Issue 11*. pp. 626-647

[5] Segal, M.; Akeley, K., “Open GL 2.0 spec”, http://www.opengl.org/documentation/specs/version2.0/glspec20.pdf [accessed 2005, Apr. 22]

[6] A. Khan, G. Wainer. “A visualization engine based on Maya for DEVS models”. In *Proceedings of SISO Fall Interoperability Workshop.* San Diego, CA. U.S.A. 2005.

[7] ALIAS Corp. "Maya 6 Features in Detail," [Accessed Oct. 2004],http://www.alias.com/eng/products-services/maya/file/maya6_features_in_detail.pdf.

[8] Mark J. Kilgard. 1996. “The OpenGL Utility Toolkit (GLUT) Programming Interface: API Version 3”. [Accessed Mar 2005]. http://www.opengl.org/resources/libraries/glut/glut-3.spec.pdf

[9] Gil Gribb, and Klaus Hartmann. 2001. “Fast Extraction of Viewing Frustum Planes from the World-View-Projection Matrix”. [Online document; Mar 2005] http://www2.ravensoft.com/users/ggribb/plane%20extraction.pdf

[10] Mark Segal, and Kurt Akeley. 2004. “The OpenGL Graphics System: A Specification”. [Online document accessed Mar 2005]. Available: http://www.opengl.org/documentation/specs/version2.0/glspec20.pdf

[11] Venhola, W.; Wainer, G. “Design and implementation of the DEVSView visualization tool”. Technical Report SCE-05-17. Dept. of Systems and Computer Engineering. Carleton University. 2005.

[12] Saadawi, H. “Implementing a DEVS model of an ATM machine”. Internal Report. Dept. of Systems and Computer Engineering. Carleton University. 2004.