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 developed to visualize the results of a simulation created using the CD++ modeling and simulation toolkit. The tool, called DEVSView, allows users to create visualizations from the simulation log files generated by the CD++ toolkit. DEVSView has implicit support for DEVS and Cell-DEVS models, using 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. Future work will include loading Maya model files for complex objects, and more advanced model positioning capabilities.

Keywords: DEVS, Cell-DEVS, Visualization, CD++, OpenGL

1. INTRODUCTION

We will introduce the features of DEVSView, a visualization tool implemented to improve the available options for visualizing Discrete Event Systems Specification (DEVS) simulations [1] executed in the CD++ toolkit environment [2][3]. DEVS provides a framework for the construction of discrete event hierarchical models in a modular manner. A system modeled with DEVS consists of behavioural (called atomic) models and structural (called coupled) models. A structural model is composed of several atomic or coupled sub-models. The coupled models are composed of atomic models connected through input and output ports defined in their interfaces. The Cell-DEVS formalism extends this behaviour to enable defining cellular models to model systems that operate over area of space.

CD++ is a tool to create simulations that follow the DEVS specifications. A new atomic model is generated by designing a new class derived from the *Atomic* class. First, the model must be registered using the method *MainSimulator.registerNewAtomics*(). Then, the following methods should be overloaded:

- ***initFunction***: this method is invoked at the beginning the simulation. It allows to define initial values and to execute initial functions for the model. When this method is executed, the value of *sigma* (next scheduled event) is set to infinite and the model phase to *passive*.
- ***externalFunction***: this method is invoked when an external event arrives from an input port.
- ***internalFunction***: this method is started when the value of *sigma* is zero, since an internal event has occurred.
- ***outputFunction***: this method executes before the internal function, allowing to provide outputs for the model.

These methods have been built following the formal specifications of DEVS models. In addition, several primitives have been defined to allow interacting with the abstract simulator:

- ***holdIn***(state, time): it is used to define that the model will remain in *state* during *time*. When this time is consumed (*sigma* = 0), the model executes an internal transition. This function is devoted to implement the **D** (lifetime) function of the DEVS formal specification.
- ***passivate()***: the model enters in passive mode and it will be reactivated by an external event.
- ***sendOutput***(time, port, value): it sends an output message through the given port.
- ***state***(): it returns the present model phase.
- ***getParameter***(modelName, parameterName): it allows to access to the model state variables.

Coupled models are defined using a specification language specially defined with this purpose. This specification language also follows the formal definitions for DEVS coupled models. Each coupled model is composed by a set of definitions. Optionally, configuration values for the atomic models can be included. Each set indicates the name of the model and its attributes. The **[top]** model defines the coupled model at the top level.

Four properties must be configured: components (using the clause "*components*"), output ports (clause "*out*"), input ports (clause "*in*") and links between models (clause "*link*"). The syntax is:

- ***Components***: It describes the models composing the coupled model. The syntax is: *model_name@class_name*. The name of the model is needed because we can use more than one instance of the same model. The class' name can reference to either atomic or coupled models. The last ones should be defined in the same configuration file as a new group. The order used when the models are set defines the priority for the **select** function (that is, the execution order under simultaneous events).
- ***Out***: It defines the names of output ports.
- ***In***: It defines the names of input ports.
- ***Link***: it describes the internal and external coupling schema. The syntax is: *source_port*[*@model*] *destination_port*[*@model*]. The name of the model is optional since if it is not indicated the coupled model being defined will be used.

Cell-DEVS specifications are completed by adding the following parameters:

- *type*: [cell | flat].
- *width*: INTEGER.
- *height*: INTEGER.
- *link*: in this case it must use the name of the cell space and the corresponding input/output cell (Model(x,y)).
- *border*: [WRAPPED | NOWRAPPED].
- *delay*: [TRASPORT | INERTIAL].
- *neighbors*: Cell-DEVS_name(x_1, y_1), ..., Cell-DEVS_name(x_n, y_n).
- *localTransition*: It defines the description for the behavior specification used for the local computation function.
- *zone*: transitionName {range_1..range_n}. It associates a behavior specification with the cells included into the rage defined by the sentence. In this way, different ranges can provide different behavior.

Simulations in CD++ produce complicated results, and can depict interactions that occur in three dimensions. The results of CD++ are recorded in text based log files. These results sometimes require extensive interpretation and reconstruction to clearly see what is occurring during the simulation.

```
MessageI/0:0:0:00/Root(0) for top(1)
MessageI/0:0:0:00/top(01) for incdec(2)
MessageD/0:0:0:00/incdec(02)/... for top(1)
MessageD/0:0:0:00/top(01)/... for Root(0)
MessageX/0:0:0:00/Root(0)/op0/1 for top(1)
MessageX/0:0:0:00/top(01)/op0/1 for incdec(2)
MessageY/0:0:05:00/incdec(02)/res0/1 for top(1)
MessageY/0:0:05:00/incdec(02)/res1/0 for top(1)
MessageY/0:0:05:00/incdec(02)/res2/0 for top(1)
MessageY/0:0:05:00/incdec(02)/res3/0 for top(1)
MessageY/0:0:05:00/incdec(02)/res4/0 for top(1)
MessageD/0:0:0:00/incdec(02)/... for top(1)
MessageY/0:0:05:00/top(01)/res0/1 for Root(0)
MessageY/0:0:05:00/top(01)/res1/0 for Root(0)
MessageY/0:0:05:00/top(01)/res2/0 for Root(0)
MessageY/0:0:05:00/top(01)/res3/0 for Root(0)
MessageY/0:0:05:00/top(01)/res4/0 for Root(0)
```

Figure 1: DEVS simulation results in CD++.

The purpose of all DEVS visualization tools is to provide the capabilities to accomplish this task. CD++ log files contain an event per each line of the log file. Each event specifies: source model, destination model, time sent, value sent, port over which the value was sent, and event type. The tool uses this information and a couple of visualization techniques to provide the capability of visualizing simulations.

CD++ was provided with different software tools to visualize the results of the simulations:

- Java Applet VRML viewers [3]
- Alias Maya 3D Software [4]

These methods have some limitations. The Java applets use Java3D libraries and the VRML specification, both which are no longer actively developed. The current VRML viewers also lack functionality and ease of use. 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 for every user (moreover considering that CD++ is an open-source tool available for academia [5][6].

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 used to create the simulations, 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 visualizations:

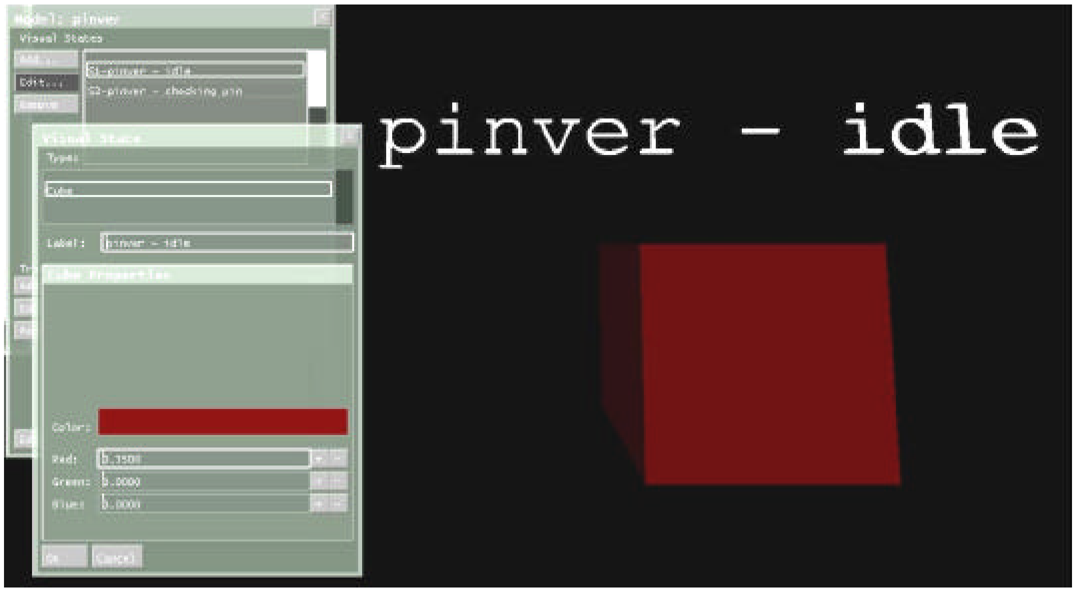
1) Graphical user interface based on the OpenGL Utility Toolkit [7]: it includes a windowing system that provides buttons, text fields, list boxes, resizable windows, and other controls necessary for a GUI. The rendering of the controls is accelerated by OpenGL [8].
2) Visual state transition and event 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 input/output 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 the DEVSView tool. A Visualization 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 and destination models 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.

The tool can create visual representations of systems by parsing the log files of a CD++ simulation and creating visual models for each atomic and coupled model found. Once created, the visual models can be customized to follow a visual state transition system (described in 2.1) and/or produce animations following certain events (described in Section 2.2). Alternatively, the visualization models can be created by editing the visualization file directly. Figure 1 shows an example visual model named *pinver* (Pin Verifier), from an ATM simulation, in its idle state.



S1-pinver - idle
S2-pinver - checking pin

Type: Cube

Label: pinver - idle

Color: 🟥

Red: 0.3500
Green: 0.0000
Blue: 0.0000

Figure 2: A visual model in its 'idle' visual state. This visual state is a cube visual state. The options for selecting the color are provided in the Visual state edit panel shown in the bottom left.

Each visualization model has a:

- Unique name
- List of output ports
- List of input ports
- Information about location, orientation and size
- List of visual states
- List of visual state transition rules
- List of event animation creation rules
- Current visual state

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 each use the same visual states, visual state transition rules, and event animation rules.

Both the visual state transition system and the event animation system described in 2.1 and 2.2 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. For example, an event sent from an ATM model to a Customer model will be processed at the ATM visual model and the Customer visual model.

Each event contains the following information:

- The source visual model name
- The destination visual model name
- The time the event occurs
- The port the value is sent through
- The value sent

The source and destination visual models use this information to process the event. Typically, this involves comparing the port and value with behavioral rules such as transition rules or eve nt 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 DEVSView tool currently provides a couple basic value rule types to enable guard conditions on transition rules and conditions for creating event animations.

These value rule types are:

1) All Values: this rule returns true for all values passed to it.
2) Equals Value: this rule passes if the value passed to it is equal to a predetermined constant.
3) Range of Values: this rule passes if the value passed to it is greater than the lower pre-determined constant and less than the higher predetermined constant.

The pre-determined constants are entered using the user interface. Alternatively, the constants can be edited in the visualization file directly.

2.1 Visual State Transition System

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.

All Visual States have the following properties:

- Unique Id (per visual model)
- Label
- Type

Each visual state also implements entry and exit methods to setup their visual appearance according to various inputs. These inputs can be obtained from the event triggering the transition or from other variables internal to the visual model. The visual model specifies a position, scaling and orientation of the model, which a visual state may choose to use or ignore when rendering. A visual state edit panel provides the services for editing the visual state of the model. Depending on the visual state type, the properties provided for editing may change. Figure 1 showed the cube visual state with the color option it provides. The other components of the visual state system are the transition rules from state to state.

Each of these transition rules has several properties:

- Port name and direction (Output or Input)
- Value rule
- Next state
- Unique Id (per visual model)

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. These special types of transitions are useful for reducing the number of transitions required for certain state machines. 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. When a transition rule is invoked, the visual state of the model changes to the next state specified in the rule. In addition, the state change is recorded in the visual model history, so the transition can be reversed when the playback is reversed.

2.2 Event animation system

The event animation system allows visual models to create animations which visualize the processing of certain events. Event animations provide facilities to visualize the reasons why visual state transitions occur. Consider a secure login visual model which accepts or rejects a password, and then passes this information to a server visual model. Observing the visual state of the server visual model may show the server repeatedly attempting to validate a password with the secure login model but it will not show why the server is doing so. A text animation which displays ‘password rejected’ at the secure login visual model would clearly indicate the problem. Without such an animation it is difficult to determine why the server is repeating the login process. It could be timing out and resending, it could be validating several passwords sequentially, etc. Event animations solve this problem by creating animations when certain events occur. Event animations can be 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.

Each visual model contains a list of event animation rules which contain the following information:

- Port name and direction (Output or Input)
- Value rule
- Source state
- Animation length
- Unique Id (per visual model)

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. After an animation is created, it is guaranteed to last the amount of time specified in the event animation rule.

2.3 Octary Spatial Partitioning Data Structure

Each visual model in the visualization has a current visual state and this visual state defines the graphical representation of the visual model. The octtree data structure provides a data structure for organizing the graphical representations by their location in space. This data structure recursively divides the scene into 8 regions of space and assigns scene nodes into the smallest region that contains them completely. Each region is represented by a node in the octtree. The initial region and one subdivision are shown in Figure 3.

The reason for using an octtree data structure is that it provides efficient view culling and other common scene operations. Complex scenes require significant rendering time and may contain many different objects. Determining what objects to draw is important, since it is inefficient to draw every object every frame. View culling is the process of calculating which objects are in view and therefore require rendering. The octtree data structure can optimize the process of determining what is in view. The process consists of traversing the octtree, and for each node (region) determining whether the node is in view. If the node is out of view, the entire tree extending from that node can be culled, which potentially culls many objects with a single node visibility check. Conversely if the node is completely in view, the entire tree extending from that node is visible and does not require a visibility check.

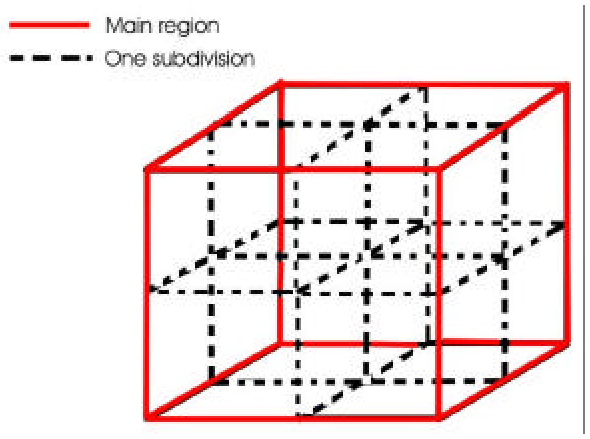


Figure 3: Octtree region division

This view culling algorithm is shown in the following pseudo code:

```
Let ON = the current octtree node
Let ParentON = the parent octtree node of ON
Let VF = the view frustum (i.e. the field of view) (See [4])
If ParentON intersects VF {
    Calculate ON visibility
    If ON is not visible {
        Stop traversing ON
    }
}
Else ParentON is completely visible {
    ON is therefore completely visible
}

If ON is completely visible or intersects VF {
    Draw each scene node contained in ON
}
If ON has children octtree nodes {
        Draw each child octtree node by repeating this
        pseudocode with ParentON = current ON, and new
        ON = child octtree node.
}
```

The octtree also provides the capability for efficient collision detection (Important for object selection), and distance sorting (Important for transparency). Object selection and transparency are features which could be useful for future development but are not currently being used.

The DEVSView implementation of the octtree has the capability of assigning graphical objects to several octtree regions to better define the outline of the object. Consider a small object located at the centre of the root region. This object will only fit inside the root region so it must be assigned to that region. Therefore the small object will only be culled if the root is culled, despite the fact that it may rarely be in view. If the small object were added to the 8 smallest regions that contain it completely, then that object will be culled much more efficiently by the Octtree. Figure 4 shows the principle in two dimensions.

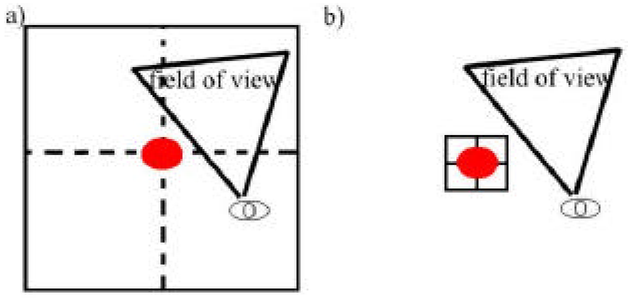


Figure 4: a) the red circle is added to the smallest region that entirely fits it. The field of view does not cull the region so the circle is drawn. b) the red circle is added to the 4 smallest regions that contain it completely. The field of view culls the regions and the circle is not drawn.

3. THE DEVS VISUALIZATION TOOL

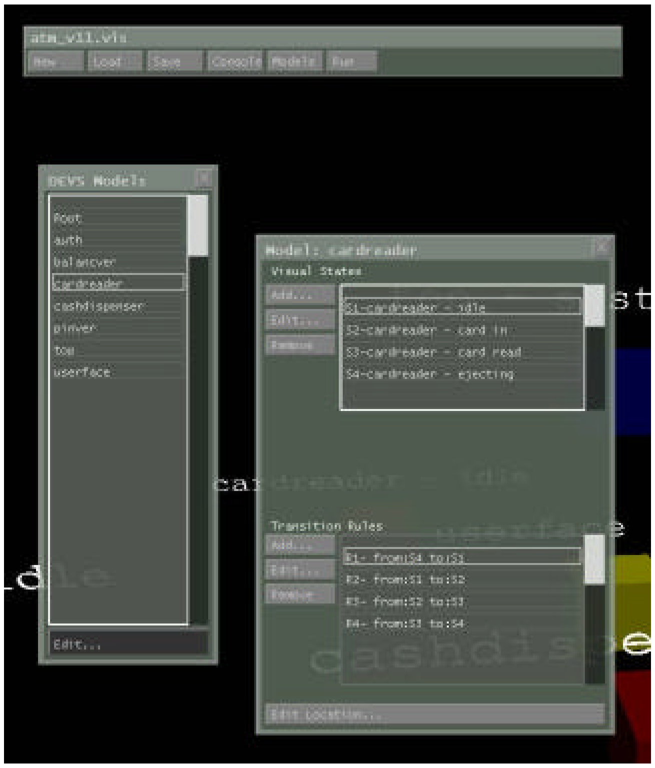
The following sections describe a couple of simulations visualized using the DEVSView tool.

3.1 An ATM Simulation

The ATM simulation consists of several atomic models interacting with each other to approximate the services provided by an ATM machine. The visual models were extracted from the simulation log file and the visual state machines were defined using the DEVSView user interface. Figure 4 shows the visual models, and the visual state machine interface for the *cardreader* model.

The ATM simulation also provides a text animation which displays "Card Inserted" whenever a bank card is inserted. This text animation was added manually to the visualization file. The text animation can be seen in Figure 5, which shows a frame from the visualization right after a customer arrives at the ATM.

The visual model *Root* and *top* are always present in DEVS simulations; however they rarely need a visual state. The ATM has other visual models: *auth* (the authorization component), *balancever* (in charge of verifying customer's balance), *cardreader* (to read the data from customer's bank cards), *cashdispenser* (to deliver cash), *pinver* (the application to verify the correctness of pins typed), and *userface* (the user's interface). The visual states and transition rules for the *cardreader* model are shown in the model edit panel. The *cardreader* model can be in four states: *idle*, *card in*, *card read*, *ejecting*, for each of the states related to the card use. There are four transition rules which travel from *idle* to *card in*, *card in* to *card read*, *card read* to *ejecting*, and from *ejecting* back to *card in*.



Root
auth
balancver
cardreader
cashdispenser
pinver
top
userface

S1-cardreader - idle
S2-cardreader - card in
S3-cardreader - card read
S4-cardreader - ejecting

R1- from:S4 to:S1
R2- from:S1 to:S2
R3- from:S2 to:S3
R4- from:S3 to:S4

Figure 5: ATM visual models and the interface to edit the visual models.

Figure 6 shows text animation for the ATM model, which reads *Card Inserted*, just above the *cardreader* model. Since the card was inserted, the *cardreader* model transitions to the *card in* state and the *top* model transitions to the *customer in system* state.

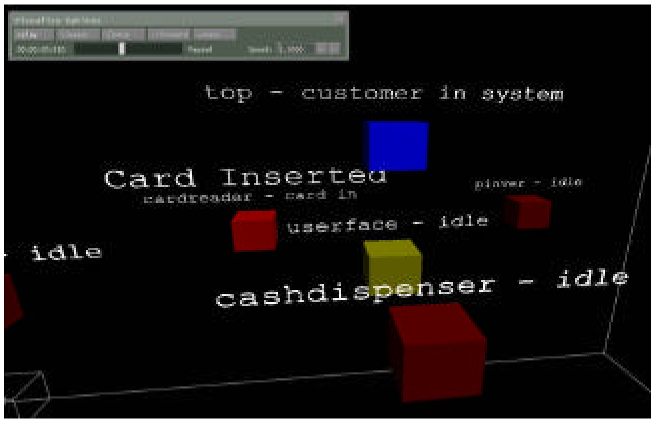


Figure 6: A frame in the ATM visualization after a customer arrives.

3.2 A Bouncing Ball Simulation

This model represents a bouncing ball within a closed area. Each cell is represented by 5 possible states: 0, an empty space; 1, the object moving SW; 2, the object moving SW; 3, the object moving NE; 4, the object moving NW. The following figure presents the model representation in CD++

```
[bouncing]
type : cell  width : 20 height : 15
delay : transport  border : nowrapped
neighbors : (-1,-1) (-1,1)(0,0)(1,-1) (1,1)
localtransition : move
zone : ULcorner{ (0,0) }
zone : URcorner { (0,19) }
zone : BLcorner { (14,0) }
zone : BRcorner { (14,19) }
...

[move]
rule : 1 100 { (-1,-1) = 1 }
rule : 2 100 { (1,-1) = 2 }
rule : 3 100 { (-1,1) = 3 }
rule : 4 100 { (1,1) = 4 }
rule : 0 100 { t }
...

[ULcorner]
rule : 1 100 { (1,1) = 4 }
rule : 0 100 { t }

[URcorner]
rule : 3 100 { (1,-1) = 2 }
rule : 0 100 { t }

[BLcorner]
rule : 2 100 { (-1,1) = 3 }
rule : 0 100 { t }

[esquinaDR-rule]
rule : 4 100 { (-1,-1) = 1 }
rule : 0 100 { t }
```

Figure 7: Cell-DEVS definition for the bouncing ball simulation.

The simulation in Figure 8 shows three balls contained in a 2d grid which bounce of the walls. The figure 6 shows a composite of the visualization during playback. The image shows the motion of the balls in the 2d grid.

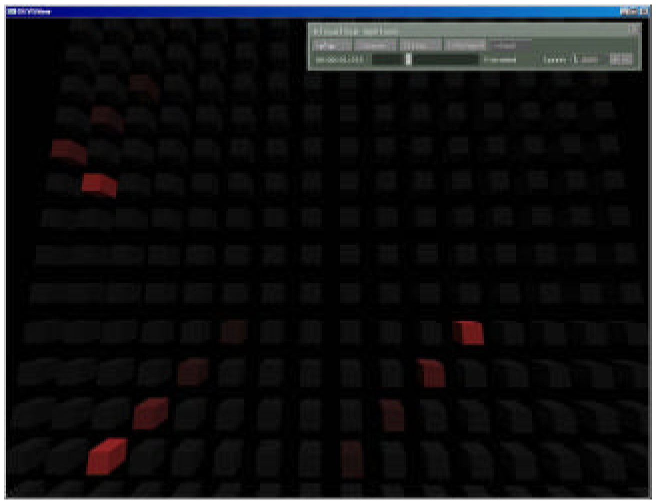


Figure 8: A composite of several frames in the bouncing ball simulation.

4. CONCLUSION

The DEVSView tool 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 the DEVSView tool.

There are several important features that could potentially increase the utility of the DEVSView tool. The following is a list of several improvements to the tool that may prove useful.

- An interface to scripting languages for complicated Value rules, Entry methods, Exit methods, and other state machine operations.
- Maya 3D model loading for complex graphical objects
- Environment detail objects. Terrains, Backdrops, etc
- Cell model alignment and per cell position manipulation. With this feature, one dimensional cell models could be aligned to lines, 2D cell models could be aligned to planes, and 3D cell models could be aligned to containers.
- Multiple camera views and point and click object selection.
- Simulation statistics display
- Graphical visual state machine editing

- Interfacing to a running CD++ simulation for interactive simulations

The visualization facilities of the DEVSView tool provide the beginnings of a powerful tool.

REFERENCES

[1] Zeigler, B; Kim, T; Praehofer, H. "Theory of Modeling and Simulation". Academic Press. New York, 2000.

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

[3] Wainer G, Chen W. 2003. A Framework for Remote Execution and Visualization of Cell-DEVS Models. *SIMULATION, Vol. 79, Issue 11.*

[4] Khan, A.; Wainer, G.; Venhola, W.; Jemtrud, M. "On the use of CD++/Maya for visualization of discrete-event models". In *Proceedings of IMACS 2005*. Paris, France. 2005.

[5] CD++ distribution. http://www.sce.carleton.ca/faculty/wainer/wbgraf

[6] Open Source DEVS http://sourceforge.net/projects/odevspp/

[7] Kilgard, M. J. "The OpenGL Utility Toolkit (GLUT) Programming Interface: API Version 3". Available: http://www.opengl.org/resources/libraries/glut/glut-3.spec.pdf. 1996.

[8] Gribb, G.; Hartmann, K. "Fast Extraction of Viewing Frustum Planes from the World-View-Projection Matrix".Available: http://www2.ravensoft.com/users/ggribb/plane%20extraction.pdf. 2001.

[9] Segal, M.; Akeley, K... "The OpenGL Graphics System: A Specification". Available: http://www.opengl.org/documentation/specs/version2.0/glspec20.pdf. 2004.