Behavior-based Graphics Using Standard Components

Wm Leler
Victoria University of Wellington
wm@leler.com

Abstract

This paper describes AppComposer, the first system to combine behavior-based authoring techniques with standard component architectures. AppComposer features a number of novel features, including the use of a compiled, statically typed language as a scripting language, live editing, and visual encapsulation hierarchies. AppComposer has been used not only for traditional graphics, but also for Web applications and large enterprise applications.

Introduction

Behavior-based authoring has been used in a number of computer-graphics systems, including the Artificial Reality Kit [ARK86] and the multimedia authoring tool mTropolis [mTropolis00]. In these systems, the user applies **behaviors** to **actors**. For example, an actor might represent a visual object, such as a spaceship, and a behavior applied to the spaceship actor might cause it to move about the scene. Behavior-based animation systems have a number of advantages over traditional script-based systems, especially for interactive graphics where it is hard to script all possible interactions.

A significant limitation of most behavior-based systems is that the set of actors and behaviors available to the user is limited to those built-in to the system. It is typically difficult (or impossible) for the user to add new actors or behaviors.

At the same time, component architectures such as COM and JavaBeans have become popular for building user interfaces for interactive computer applications. The user interfaces for most computer applications are now built using component architectures.

We recognized the potential synergy between these two technologies – standards-based components can add needed extensibility to behavior-based systems, and behaviors can make it much easier to build rich, multimedia applications using standard components.

AppComposer

We have built a behavior-based visual authoring system called AppComposer. AppComposer has been used in such diverse domains as animation, multimedia, distributed applications, and Web applications.

We made an early decision to build AppComposer in Java and use Java components, but the same ideas would work equally well in similar languages and component architectures (such as C# and COM/DCOM). Even within Java, there are many different kinds of things that can be considered to be components. We wanted to be as inclusive as possible, so AppComposer is able to use just about anything as a component, including JavaBeans, Enterprise JavaBeans (EJB), databases, Java Server Pages (JSP), Web services, and even regular Java classes.

Figure 1 shows an animation created in AppComposer. This view is called the **outline view**. The rabbit and happy actors are both the same JavaBean component that animates through a sequence of images. Notice the two behaviors called Jump and Drag, which have been applied to the rabbit actor.

Figure 1: Animation with 3 actors and 2 behaviors

A visual editing (WYSIWYG) view of the animation is shown in Figure 2. Each actor has an editing handle, which can be turned off to see the finished animation.

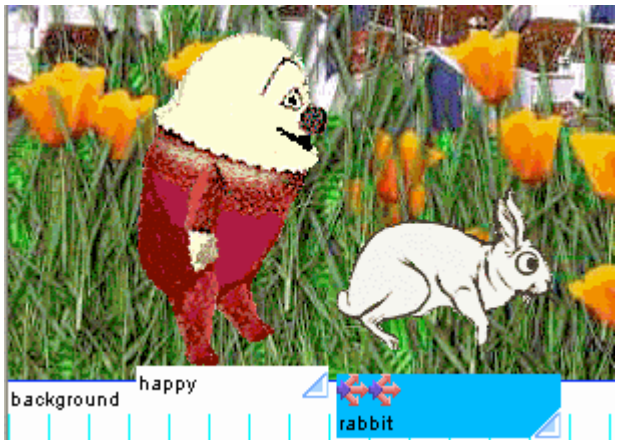

Figure 2: AppComposer visual editor

The Drag behavior enables the user to drag the rabbit around the scene with the mouse, and the Jump behavior causes the rabbit to jump in the air when the user clicks on it. The same behaviors can be applied to the other actors (the happy character, or even the background image) simply by moving (or copying) the behaviors from the rabbit actor to the other actor. We can also use the same behaviors in other animations. A full view of this animation being built is shown in Figure 18.

A key innovation of AppComposer is that it uses behaviors not only to control a single actor, but also as connectors between actors. This is more clearly shown in Figure 3, where the behavior named "start" connects the StartButton actor to the Juggler actor so that when the StartButton is pressed the Juggler starts juggling.

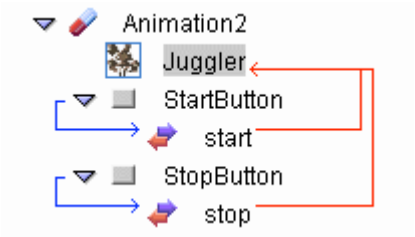

Figure 3: Behaviors as connections

Figure 4 shows the resulting animating juggler, captured in mid-juggle.

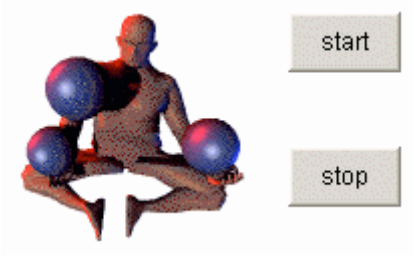

Figure 4: Juggler and buttons

AppComposer allows any component to be used as an actor, and (as mentioned above) AppComposer allows pretty much anything to be used as a component. In Figure 3, the StartButton and StopButton actors are regular Java AWT buttons. Nothing needed to be done to them to allow them to be used in AppComposer, even though Sun did not implement them as JavaBeans (or any other kind of standard component). At the end of this paper are screenshots of a number of applications built using components from a number of sources.

A behavior always has a **stimulus** and a **response**. The stimulus is an event, such as a mouse click or a button press. In Figure 3, AppComposer highlights the stimulus of a behavior using blue arrows, and the response using red arrows.

AppComposer includes seven primitive behaviors, and the user can define new reusable behaviors (in particular, the Drag and Jump behaviors from Figure 1 are user-defined). Primitive behaviors, like the "start" behavior from Figure 3, are defined using a dialog, as shown in Figure 5.

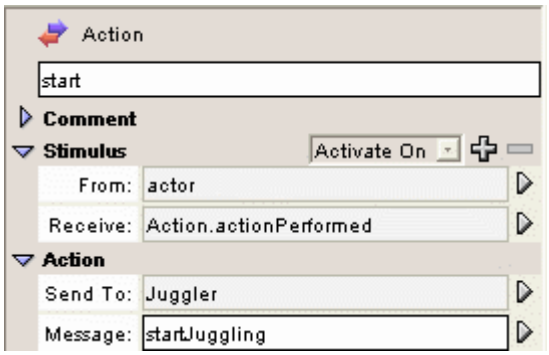

Figure 5: The start behavior

For the "start" behavior, the stimulus is an event from the StartButton that says the button has been pressed, and the response is a method call (startJuggling) on the Juggler.

These events and method calls are those that were defined by the author of the AWT button and the Juggler component. This allows behaviors to be used on any component.

The response of a behavior can be a single method call, as in Figure 5, or it can be more complicated. The response of the Jump behavior (from Figure 1) is a sequence of method calls over time to move the position of the actor (rabbit) up and down. Behaviors execute asynchronously in parallel.

A behavior *always* represents a connection, even when it is connecting an actor to itself. The Jump behavior (in Figure 1) is connecting the rabbit actor to itself by wiring an output on the rabbit (mouse clicked) to an input (position).

AppComposer also allows the stimulus of a behavior to be a Java exception, which AppComposer treats like an event. This provides a convenient way to visually deal with program exceptions using the same behavior mechanism.

Compound Components and Capsules

In AppComposer, actors (components) and behaviors (connections) are assembled inside of **capsules**. For example, in Figure 3 the capsule is called "Animation2". Capsules themselves are also components, and can be used as actors in other capsules. This allows applications to be built up hierarchically. The name "capsule" comes from the fact that capsules provide encapsulation (data hiding). Internal details of the capsule are hidden, and can only be accessed through the capsule's interface.

A capsule has a **type**. In the previous examples, the capsules have all been of type Application, but a capsule can be of a number of types, including a servlet, applet, JavaBean, or even an EJB (we will see how AppComposer can be used with EJBs in a later section). By creating capsules of type JavaBean or EJB, AppComposer can not only assemble applications out of existing components, but can build new components. These new components can not

only be used in AppComposer, but also in applications created using normal programming tools.

Web Applications

Not only can AppComposer be used to build animations and other client-based applications, it can be used to build distributed Web applications. A **servlet capsule** is a Java servlet that receives an HTTP request and sends back an HTML response. Figure 6 shows a servlet capsule for a Web application named TimeServlet that receives an HTTP GET request and responds by sending back the current time as an HTML page.

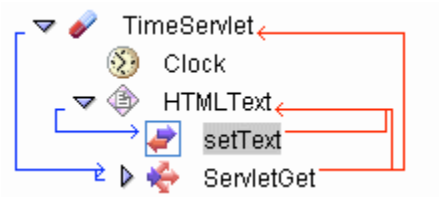

Figure 6: Servlet capsule (Web application)

The ServletGet behavior is activated when it receives a doGet event from the servlet capsule (indicating that an HTML GET request has been received). It responds by outputting the value of its actor (HTMLText). The setText behavior is activated when its actor (HTMLText) is about to be output, and sets the text to be the current time.

AppComposer contains a built-in Web server and servlet container for building and debugging servlets. The servlet defined in Figure 6 can be accessed from any Web browser using the URL http://localhost/servlet/TimeServlet. You can even access the servlet from another machine by replacing "localhost" with the domain name (or IP address) of the machine running AppComposer.

Live Editing and Dynamic Compilation

A powerful feature of many animation systems is the ability to edit an application even when it is running. We wanted to provide the same **live editing** ability in AppComposer, but we realized this would be difficult since Java is a compiled, statically typed language. Because components are normally written in compiled languages like Java, and we wanted the execution speed and type safety of compilation, we didn't want to resort to interpretive techniques. Our solution is **dynamic compilation**.

When part of an application has been changed, AppComposer saves the execution state of that part. This is done using serialization (although AppComposer does not require components to implement the Java serialization interface, since we want to be able to use just about anything as a component). The changed part is then recompiled, dynamically linked back in, and deserialized. AppComposer uses lazy compilation so that a changed part is only recompiled just before it is executed.

Live editing is one of the most powerful and popular features of AppComposer. Dynamic compilation provides the advantages of compiled code while also providing all the flexibility and responsiveness of an interpreted system. And because the output is 100% Java, no runtime system is required (beyond the normal Java runtime).

If there is a programming error, the error message is displayed along with the object that caused the error so it can be fixed. Compiled languages allow many errors to be caught and fixed at compile time (especially type mismatches) that would cause runtime errors in interpreted languages.

Java as a Scripting Language

Most animation environments provide a scripting language for those tasks that are more easily specified textually than visually. Usually such environments create a new (proprietary) scripting language, such as the Lingo scripting language, used by Macromedia Director.

AppComposer already implements dynamic compilation, so we were able to use standard Java as AppComposer's scripting language. A huge benefit of this is that in any place where a value needs to be specified, an arbitrary Java expression can be used instead. For example, Figure 7 contains the definition of the setText behavior from Figure 6.

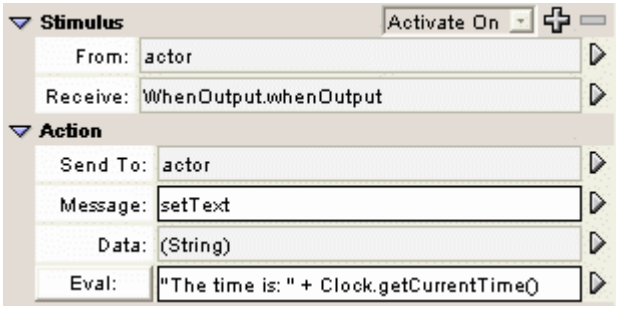

Figure 7: Java expressions in behaviors

This behavior sets the value of the HTMLText actor when it is about to be output (as part of a Web page). The value of the text is set to an evaluated expression – the constant string "The Time is: " concatenated with the value of the current time returned by the Clock actor.

Note that while this example uses a snippet of Java code, the same result could have been obtained visually by using two HTMLText actors (one containing the constant string and the other set to the current time). But in this case it was easier to just use a little Java code.

In addition to using Java expressions in action behaviors, AppComposer provides a **script behavior**, which makes it easier to include extended pieces of Java code, if desired. Note that using Java as a scripting language has nothing to do with the JavaScript language.

Scripting Support

In order to use make Java easier to use as a scripting language, AppComposer does two things:

1. Automatic boxing and unboxing of primitives. In AppComposer, primitive Java types (e.g., int) can be used interchangeably with their object equivalents (e.g., Integer). AppComposer inserts code to do any necessary conversions. This ability was subsequently added to the Java language.
2. Automatic method and class scaffolding. In Java, code must be defined inside a method, and methods must be defined as part of a class. Java expressions used in behaviors are not part of any method or class, so AppComposer wraps scaffolding around them so they can be compiled and executed, and obey scoping rules.

These features make Java much easier to use as a scripting language (and apparently as a programming language – we received an email from a teacher who told us he was using AppComposer for his introductory programming course).

Distributed Applications

The advantages of component-based design are especially apparent in distributed applications – applications that involve multiple processes that do not share memory. AppComposer supports the construction of several different kinds of distributed applications:

- Applications that communicate with a database manager, using SQL.
- Applications that use Enterprise JavaBeans (EJB).
- Applications that communicate with services, such as Web services using SOAP protocols.

A common design pattern in distributed applications is the use of proxies (or stubs) to communicate with remote components. Whether the database, EJB, or Web service is running on a computer half way around the world or just in a separate process on the same computer, the technique is the same. To communicate with the remote component, a proxy component is created. The proxy component handles all communication with the remote component, as shown in Figure 8.

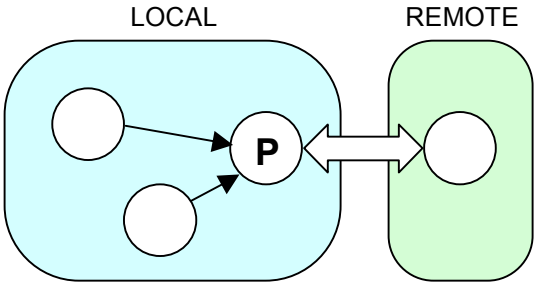

Figure 8: Proxy Component

AppComposer creates and manages proxy components automatically:

- For databases, AppComposer includes a set of components to talk to database managers. These components handle the connection to the database, and even generate SQL automatically so that most database applications can be written without any knowledge of SQL.
- For EJBs, AppComposer creates a proxy component that manages both the home and the remote interface to the EJB.
- For Web services, AppComposer creates a proxy component (using a standard WSDL description file) to communicate with the Web service.

AppComposer includes a built-in database manger (HSQL) and an EJB container (the JBoss application server), which makes it much easier to create and debug applications using databases and EJBs. The resulting distributed applications will run on any Java application server and any SQL database manager. Because there is no standard execution environment for Web services (there is no way to even tell what language a Web service is written in), so you must either access an existing Web service (or set up the service manually) to use it in AppComposer.

AppComposer comes with a number of distributed applications, including a full e-commerce Web application for renting DVDs that uses EJBs (shown in Figure 17), a Web application for buying and selling mutual funds that uses a Web service for accessing the current price of the fund (shown in Figure 16), and a Web application for an online bug tracking system that accesses a database.

Multi-tiered Web Applications

Web applications are typically built using a multi-tiered approach that separates the presentation layer from the program logic layer. In the Java world, the logic layer is implemented using components (typically EJBs), and the presentation layer is implemented using Java Server Pages (JSP), which are HTML pages containing embedded Java code [J2EE02]. Typically, the logic layer is built by programmers using traditional programming tools, and the presentation layer is built by Web designers using HTML layout tools.

As with any application that is separated into components, glue code is required. Usually, one of two things happens:

1. In theory, the program logic components should be independent of the presentation, so the glue code should go in the presentation layer (i.e., the JSPs). The big problem with this is that it requires the designers to write glue code (even though they are not programmers). Even worse, they are writing code using a web page layout tool, not a programming tool.

2. In practice, the programmers end up writing the glue code and put it in their logic layer. This makes the program logic totally dependent on the presentation. Even simple changes in the presentation require the programmers to modify their components. The resulting components are not reusable.

Glue code must be separate both from the presentation layer and the logic layer. AppComposer does this by treating JSP files as components, and uses behaviors to implement the glue code between the presentation layer and the business logic layer, as shown in Figure 9.

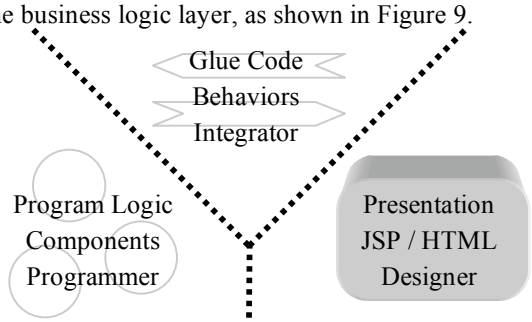

Figure 9: 3-tier Business Application

In this approach, the logic layer is programmed using normal components, which are written to be independent of the presentation layer (and thus are reusable). The presentation layer is made up of JSP files containing only simple method calls. These calls are easy to create using standard HTML layout tools. AppComposer comes with a JSP tag library to make this even easier and with a set of extensions to Macromedia Dreamweaver to automate access to this tag library, so the designer never actually has to write any Java code at all. The glue code is created using AppComposer, either by the programmers, the designers, or by separate application assemblers.

The AppComposer approach to multi-tier applications is similar to the Model-View-Controller (MVC) pattern. The business logic layer is the model, the presentation layer is the view, and the AppComposer behaviors are the controller.

At the end of this paper are a number of screenshots of Web applications built using AppComposer

Results

The concept for AppComposer was conceived at a workshop at the Banff Centre for the Arts in the mid-90s, and we have always been interested in the use of AppComposer by artists. AppComposer has been used to build a wide variety of such projects, including generating graphics for a major motion picture ("All the Rage"). We also found that artists have a keen interest in building Web applications. They have learned how to build static web pages using HTML, and now want an equally easy way to build things that are dynamic or interactive.

AppComposer was originally targeted at non-programmers, such as artists, animators, graphic designers, and Web designers. We were surprised to find that programmers like to use it just as much. Features such as live editing, which have been common in animation systems for years are not so common when dealing with compiled languages. Programmers like the ability to see changes immediately.

AppComposer is also useful for doing application integration [Leler05]. Programmers have been trying to build large applications for years out of reusable components, but with limited success [Arch93]. We found that a visual tool like AppComposer can make it much easier to integrate components. We think application integration is similar to building a financial model using a spreadsheet tool – you can do it using a text-based programming tool, but even programmers prefer using a visual tool. Not only is application integration a primarily visual task, there are huge benefits from being able to test the resulting system as you build it.

AppComposer has been used in several dozen enterprise software projects, including a major Web application built by several governmental bodies, a Web application that uses a spatial database to modify results based on the user's location, and an online bug tracking system.

Our experience over a range of projects is that the right tools and methodologies for performing component integration, along with reusable components, typically decreases application development time by a factor of four to ten times. Our experience confirms that application assembly benefits greatly from visual programming techniques. Application assembly tasks can be tedious and error prone, and are easier to perform visually than using a text-based solution. We also found that the ability to show the customer what the application will look like during development – and make changes based on their feedback – greatly increased the chances that the final application would meet their expectations.

AppComposer helps storyboard applications, and gives project teams that are often composed of people with diverse skills a common language to talk to each other. For example, since graphic designers and programmers can both use AppComposer, they use it as a way to toss ideas back and forth. AppComposer works especially well in projects that use agile (extreme) programming techniques.

And lastly, AppComposer is lots of fun to use!

References

[Actor82] Roy J. Byrd, Stephen E. Smith, S. Peter deJong, "An actor-based programming system," *ACM SIGOA Newsletter*, June, 1982.

[Arch93] David Garlan, Mary Shaw, "An Introduction to Software Architecture," *Advances in Software Engineering*

and Knowledge Engineering, I, Ambriola V, Tortora G, (eds.), World Scientific Publishing Company, 1993

[ArchJava02] Jonathan Aldrich, Craig Chambers, David Notkin, "ArchJava: connecting software architecture to implementation," *Proceedings of the 24th International Conference on Software Engineering*, May, 2002.

[ARK86] Smith, R.B., "The Alternate Reality Kit: an animated environment for creating interactive simulations," Proc. 1986 IEEE Comp. Soc. Workshop on Visual Languages, Dallas, Texas, June 1986, CS-IEEE, Los Alamitos, Calif., pp 99-106. Also reprinted in Computers and Learning, Boyd-Barret, E. Scanlon, (eds.), Addison Wesley.

[J2EE02] *Designing Enterprise Applications with the J2EE™ Platform, Second Edition*, Sun Microsystems, 2002. (http://java.sun.com/blueprints/guidelines/designing_enterprise_applications_2e/)

[Jiazzi01] Sean McDirmid, Matthew Flatt, Wilson C. Hsieh, "Jiazzi: new-age components for old-fashioned Java," *Proceedings of the 16th ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, October 2001.

[Leler05] Wm Leler, "Visual Assembly of Software Components," International workshop on Visual Languages and Computing 2005.

[mTropolis00] Glen Hunter, "mTropolis, the Greatest CBD Tool that Never Was," Mojo Productions, Inc., 2000. (http://www.cbd-hq.com/articles/2000/000501gh_mtropolis.asp)

Screenshots

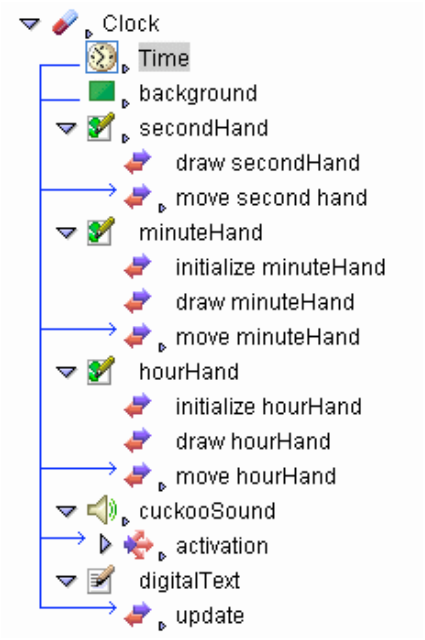

Figure 10: Outline view of clock

Figure 11: Clock with analog and digital displays, which even cuckoos on the hour

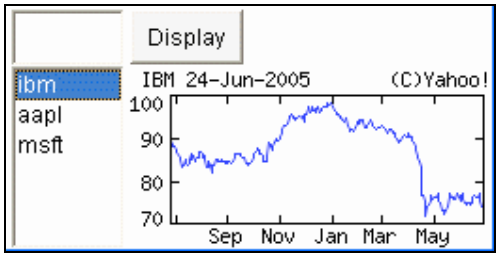

Figure 12: Client app displaying stock information from Yahoo!

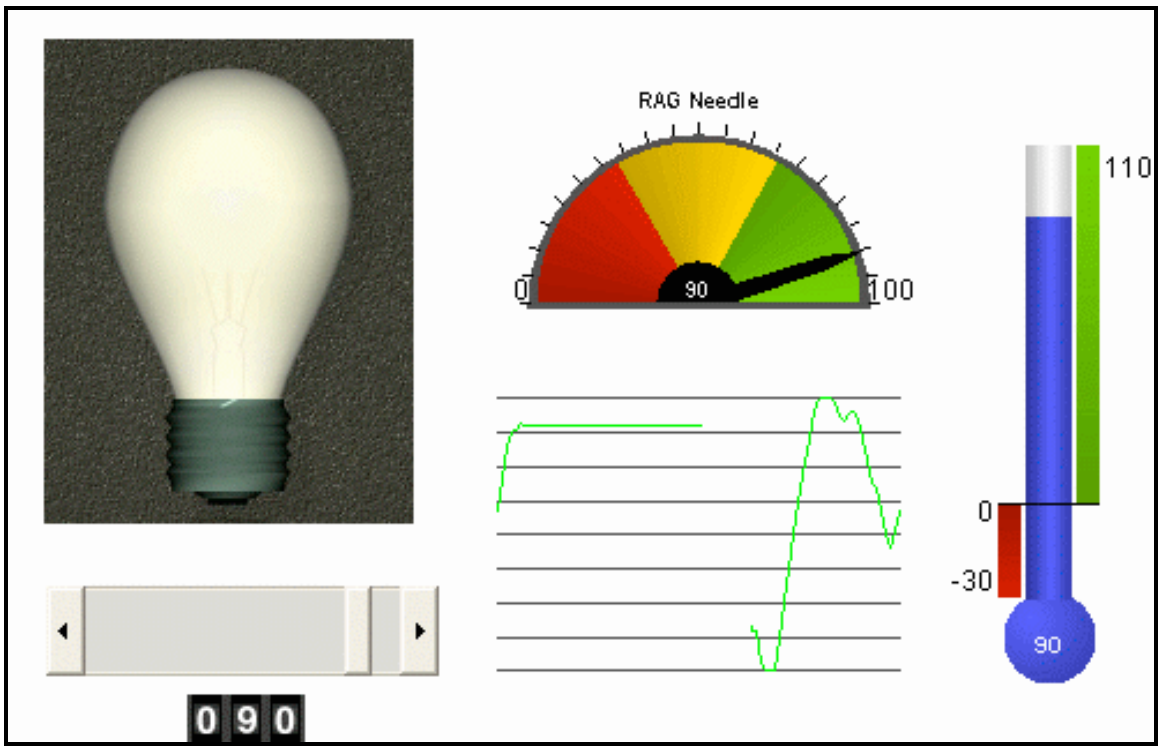

Figure 13: Gauge widgets from IBM

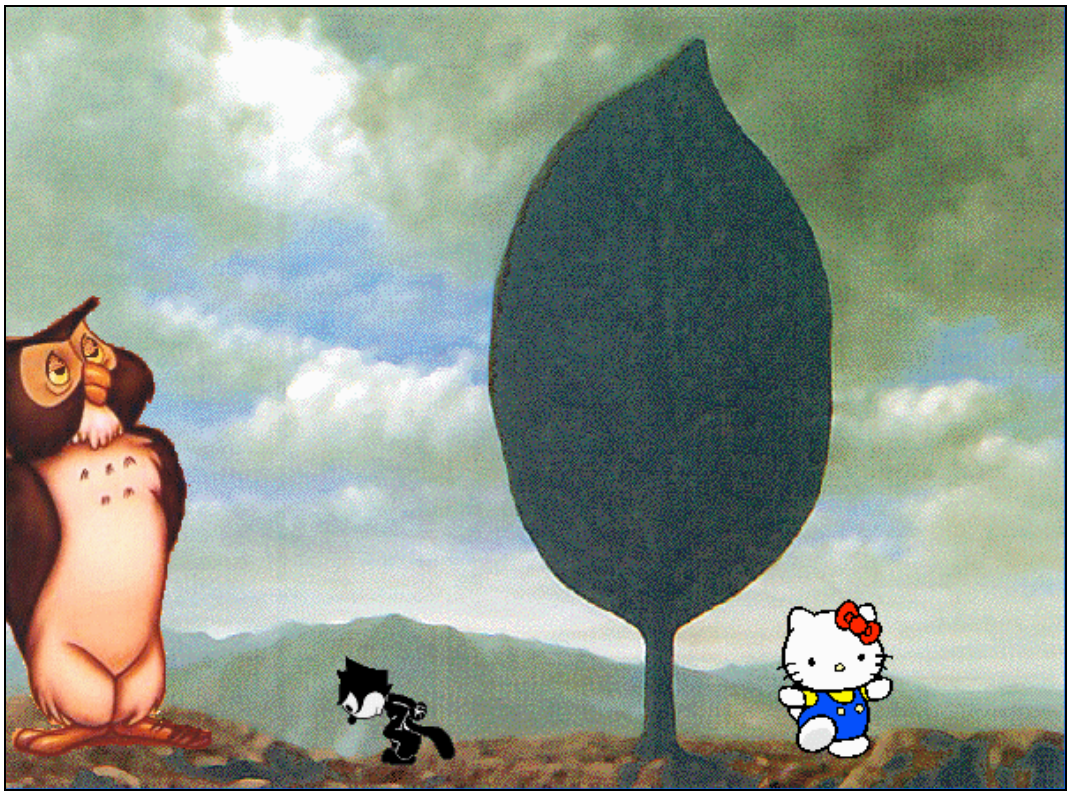

Figure 14: Interactive animation

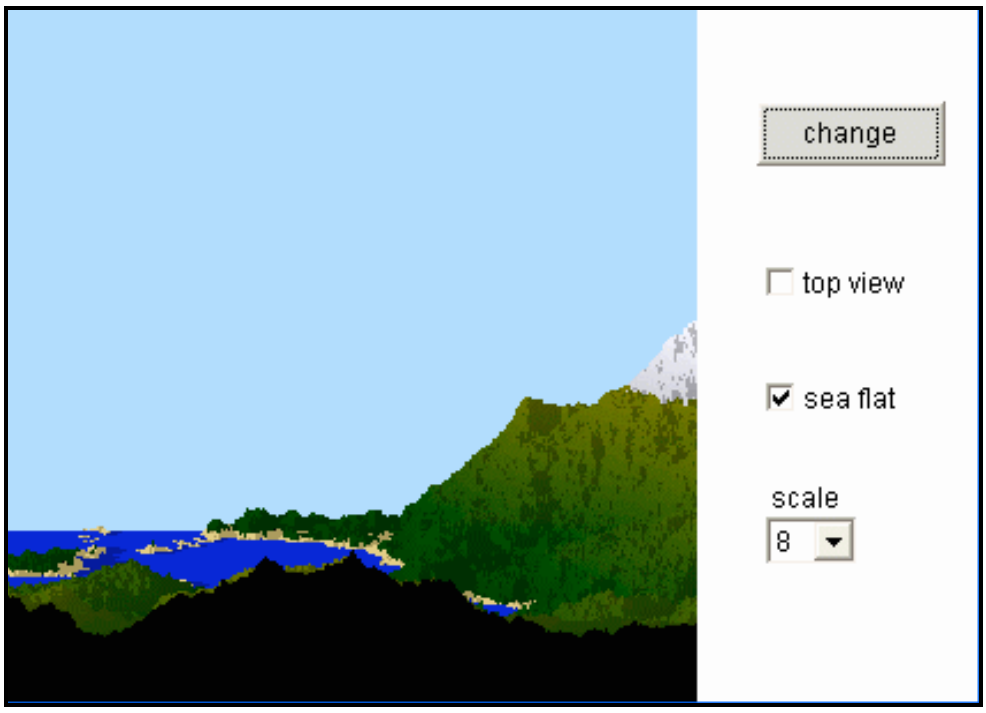

Figure 15: Fractile landscapes

Figure 16: Mutual Fund Web application using Web services

Figure 17: DVD rental Web application using Enterprise JavaBeans

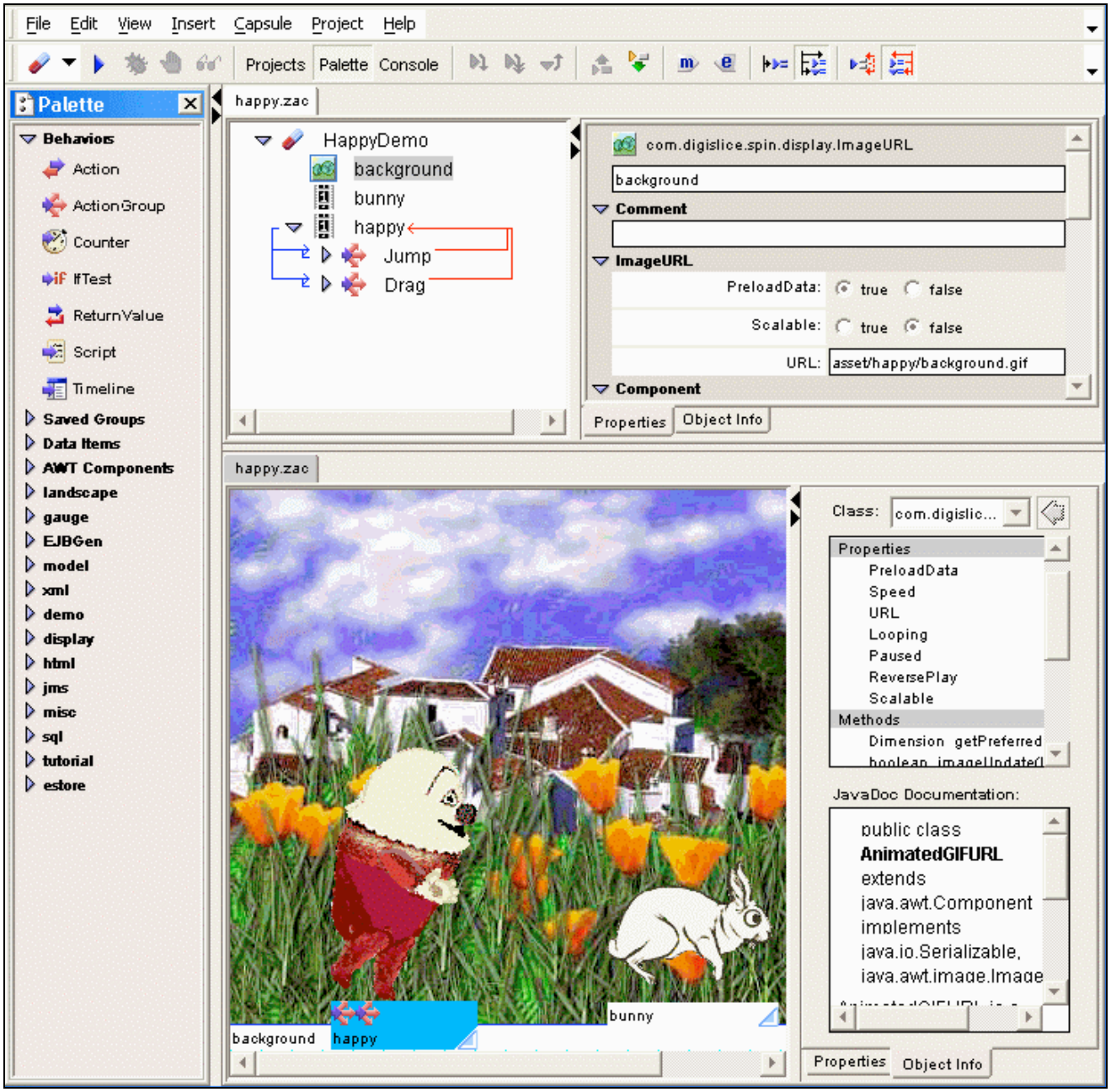

Figure 18: Screenshot of AppComposer building an animation, showing various editor views, palettes, and inspectors