developer

GUIDE.

mTropolis Developer Guide

Contents

Chapter 6. Messaging

Chapter 7. Tutorial

Chapter 8. mTropolis Examples

Chapter 1. Introduction

Note to the reader: because you are reading this manual, you must have already received sufficient proof of mTropolis' value to you and your work. Just to make our marketing people happy, here is a restatement of what the core value of mTropolis is: it helps you build better titles, faster, propelled by better collaboration between you and your coworkers, that perform superbly on the platforms likely to be in people's homes or offices. mTropolis is also totally extensible, so if you can't do something in mTropolis "out of the box," you can buy or build components that seamlessly extend mTropolis in the manner you desire.

mTropolis is a true multimedia application development environment that produces stand-alone, platform-optimized titles. Depending on the richness of the media you want to incorporate, a mTropolis title can be deployed on CD-ROM, floppy disks, hard-disks, on-line systems, or any combination thereof. Platforms currently supported by mTropolis for title deployment include any Macintosh or Power Macintosh (native) running System 7 or later, and any Windows 3.x or Windows 95 machine. mTropolis, the development environment itself, runs on Macintosh or Power Macintosh (native) machines. Versions for other platforms are on the way.

What distinguishes mTropolis from other development environments is that it compromises neither accessibility nor performance. While it is considerably more approachable than Visual BASIC or Director, mTropolis produces titles with performance that would impress even the hardest-core Visual C++ or MetroWerks user. At the same time, titles can be developed, tested and debugged interactively inside the mTropolis environment. Without lengthy compile times and while still providing complete debugging information about every process occurring within your title, mTropolis offers true incremental development of very sophisticated titles.

This incremental development process has been designed from the ground up to be equally inviting to both dedicated programmers and creative talent. An artist or producer can immediately begin moving media objects around the screen of the mTropolis environment, crafting layout, appearance and relationships without a line of script or code. Programmers can work through complex problems in a framework designed to manage complexity without imposing development metaphors. Many programming problems, ranging from sophisticated collision detection systems to very large state machines, can be created without a single line of code. Programmers and artists can integrate each others' work at any time, independently of each other, in evolutionary steps, without having

to commit to a massive final production process.

The Developer Guide describes the mTropolis environment in relation to other development models, and it provides an understanding of various design approaches to multimedia programming problems. The guide also provides a conceptual overview of the mTropolis framework, which is founded on an object-oriented, message-passing core technology called mFusion™. This guide will focus on the "citizens" of mTropolis — the various objects that mTropolis provides for your use; the hierarchies for objects that provide structure to help manage complexity; and the messages that connect mTropolis objects together into a dynamic, unified title.

The Developer Guide is targeted at multimedia developers, programmers, designers, producers, project managers and creative professionals. We are assuming that readers have had some exposure to interactive multimedia development including media types, media creation and conversion, and a fundamental understanding of the development process.

CONVENTIONS USED IN THIS MANUAL

Illustrations have been included to help you find specific information quickly.

Step-by-step instructions are shown as bullet text. For example:

- Choose **Open** from the File menu.
- Select the graphic tool in the tool palette.

The names of menu items, buttons, check boxes and radio buttons are capitalized and set in bold type. For example:

- …the **Duplicate** menu item…
- …the **Cancel** button…

The names of messages, dialogs and text fields are capitalized. For example:

- …the Modifier's Name field…
- …the Element Info dialog…

The names of commands are capitalized and italicized. For example:

- …sends a *Play Forward* command.

For keyboard shortcuts, the command key is written as "⌘".

Chapter 2. mTropolis Interface

This chapter provides an overview of the mTropolis interface. When you start mTropolis, a window appears with pull-down menus, scroll bars, pop-up menus for quick navigation around a work in progress, and a variety of floating palettes (Figure 2.1). This window represents an untitled project. A *project* is the conceptual framework in which your work on a multimedia title takes place. For all intents and purposes, a *title* and *project* are synonymous. You can work on multiple projects simultaneously: each project occupies a window like the one described above.

OBJECT MANIPULATION

Every action in mTropolis, except very specialized, global operations (like saving a title version of your project) can be accomplished

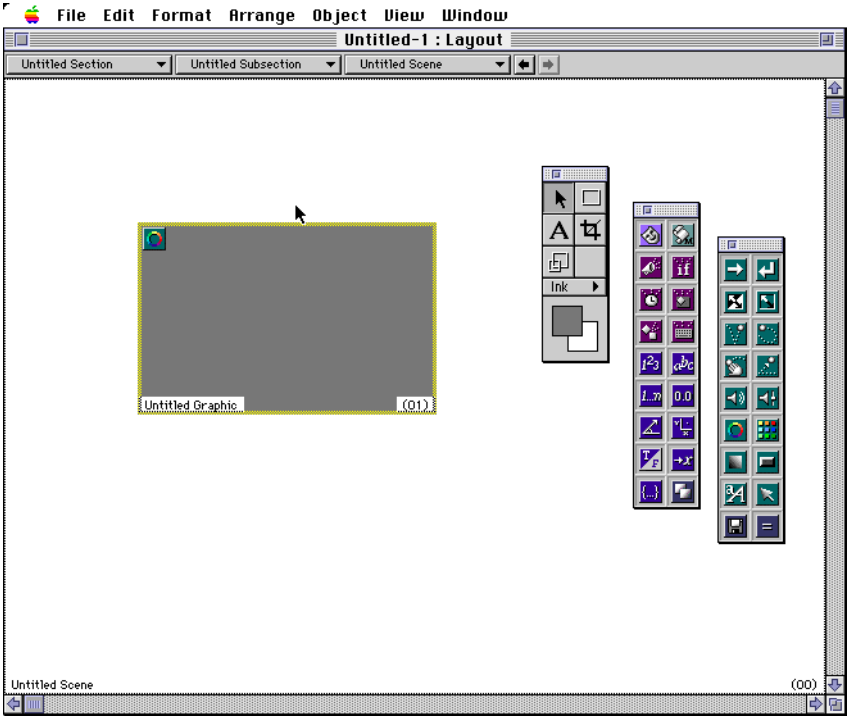


Figure 2.1 The layout window with tool and modifier palettes

with direct manipulation. To illustrate, the internal operations of mTropolis objects can be configured by double-clicking on them. Portions of a mTropolis project can be rearranged, or even moved to a different project, by dragging and dropping. The pull-down menus in mTropolis provide access to specialized operations such as opening files or linking media assets, as well as many of the operations that are available through direct manipulation.

mTropolis has been designed with careful attention to user interface consistency. If an object on-screen can be clicked on, it can be dragged and dropped somewhere meaningful and will provide helpful feedback about its role in its new home. Any object on-screen can be double-clicked, and it will, at the very least, bring up a dialog that conveys useful information about its internal workings. Generally, this dialog can be used to reconfigure an object as well.

This consistency means that you can experiment with mTropolis, incrementally applying knowledge that you gain to new tasks, without frustration. You can concentrate on learning techniques, not interface details, and the consequent short learning curve puts more power into your hands more quickly.

TOOL AND MODIFIER PALETTES

mTropolis provides a variety of palettes to keep development resources readily at hand. The modifier palettes included with mTropolis provide quick access to mTropolis objects that you can drag and drop in the process of building a title (Figure 2.2). A tool palette offers the tools most frequently used in building and manipulating the constituent pieces of a mTropolis title.

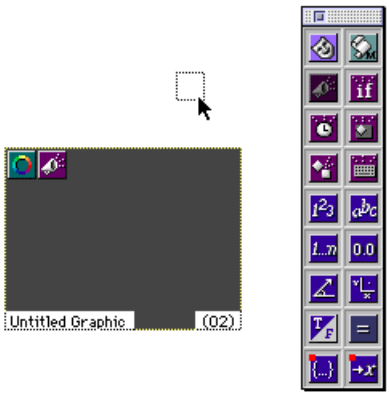


Figure 2.2 Modifiers can be dragged and dropped onto elements

Other palettes provide similar productivity in managing more sophisticated aspects of the mTropolis development process. For example, the asset palette provides a view of all of the media assets used throughout your project. Even if a media asset has already been used in your project, it can simply be dragged off the asset palette to be used again in whatever section of the project you are working on. Also, the asset palette is automatically updated whenever you link new media to your project.

mTROPOLIS VIEWS

mTropolis offers three primary views of your work, each of which allows you to edit and manage your project in different ways. The

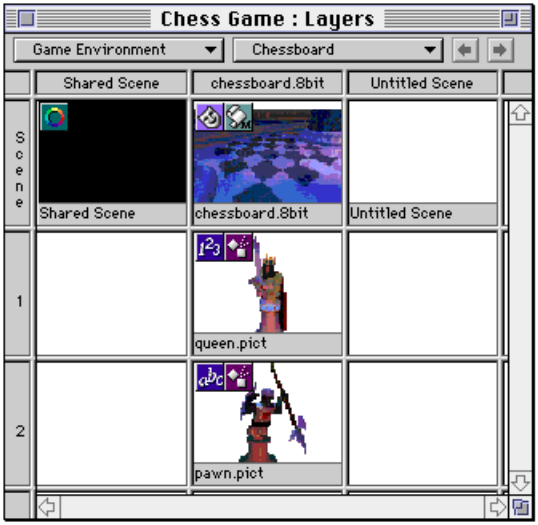


Figure 2.3 The Layers view

layout view (Figure 2.1) allows you to directly manipulate the graphical aspects of your title in WYSIWYG fashion—arranging objects, altering their appearance and so forth. The layout view is described in detail in Chapter 9 of the *mTropolis Reference Guide,* "Layout Window".

The layers view (Figure 2.3) shows you all of the constituent graphical pieces of your project in their spatial order—what is behind, or in front of what else—and allows you to rapidly rearrange this order by dragging and dropping. The layers view is a simple matrix, with each row representing a successive layer in the drawing order for a single subsection. The first column represents the shared scene and the elements it contains. Each additional column represents an individual scene and the elements that it contains. The layers view is described in detail in Chapter 10 of the *mTropolis Reference Guide,* "Layers Window".

The structure view (Figure 2.4) presents an expandable/collapsible outline that mirrors the logical hierarchy of your project. This view shows what objects are contained within others — and allows you to rearrange this hierarchy at will by dragging and dropping. The structure view is described in detail in Chapter 8 of the *mTropolis Reference Guide,* "Structure Window".

LIBRARIES

mTropolis offers a very powerful facility for managing large projects, enabling collaboration

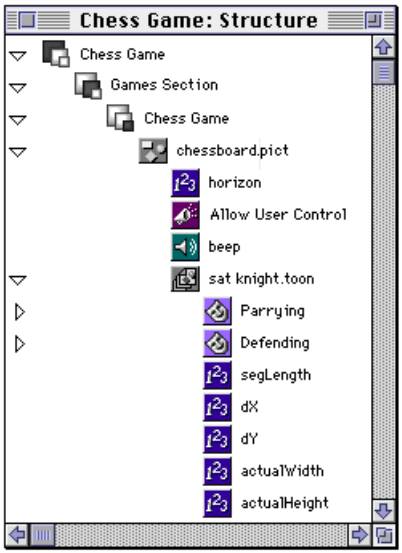


Figure 2.4 The Structure view

and promoting reuse of project work. This facility is called a library. mTropolis libraries can contain any mixture of objects, media assets, or the structure in which objects and media are embedded. Multiple libraries can be open while you are working on a project, and you can freely drag and drop to and from libraries. Libraries are described in detail in "Creating and Modifying mTropolis Libraries" on page 2.3 of the *mTropolis Reference Guide*.

DEBUGGING SUPPORT

Last but not least, mTropolis offers thorough debugging facilities. The primary mTropolis debugging facility is called the Message Log window (Figure 2.5). As mTropolis is an object-oriented system, the flow of a mTropolis title is determined by messages exchanged in conversation between mTropolis objects. The message log provides a complete, contextual history of every message that was exchanged between objects, including those that are user-generated. The message log is described in detail in "Message Log Window" on page 7.3 of the *mTropolis Reference Guide*.

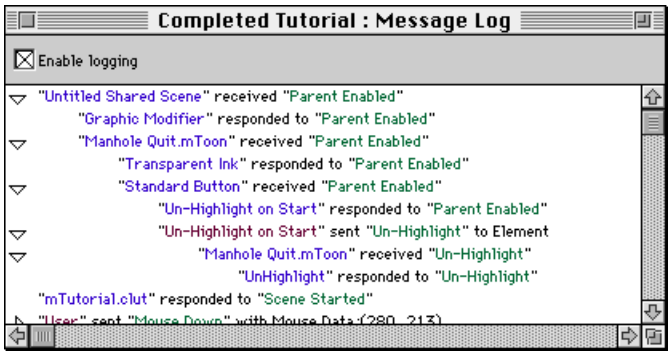


Figure 2.5 The Message Log window

Chapter 3. Object-Oriented Design

The mTropolis development environment is object-oriented. That is, you use mTropolis to create a multimedia title out of a set of cooperating software "objects". Using objects to create a title is like building a race car from Lego blocks; you can build something very sophisticated from simple parts that snap together in a clear and understandable fashion. Conversely, the old ways of creating a title were like trying to write a Shakespearean play in Latin; you had to laboriously script every potential action, step by step, in a language with which you might not feel too comfortable.

mTropolis does not impose any particular development metaphor upon its users. Artists and programmers can create, manipulate and evolve any combination of media and logic without being constrained by an abstract, linear paradigm, such as creating "frames" of a movie. This freedom permits groups of artists and programmers to collaborate without procedural bottlenecks. Once the title is developed, its constituent objects can interact under the control of any combination of time, internal decisions, or external user-generated events. This characteristic of mTropolis titles allows them to more closely model the real world, with many different events of different types driving the title experience in truly novel directions.

This chapter introduces the concept of object-oriented design. Chapter 4, "mTropolis Basics", shows how the design approach is implemented in mTropolis' authoring environment.

Any programmer or author comfortable with object-oriented programming (e.g., experienced with C++, SmallTalk, or ScriptX), can skip this chapter except for "Components and Encapsulation" on page 3.4.

OBJECTS AND MESSAGING: A DEFINITION

mTropolis' design approach requires some background information because it's an entirely new way of working with the content of a multimedia title.

Software objects are a way of structuring computer software to work more like the real world. In the real world, a hammer and a clock are objects. Most everyone knows how to use them, and no one mistakes them for each other (except perhaps Salvador Dali). Software objects act literally as the software counterparts of real-world objects: they model the real-world objects and the interaction of those real-world objects inside the computer.

In the real world, it is clear what you can do with a hammer. For example, you can pick it up or swing it. It is also clear what you can

use a hammer to do. For example, you can use it to drive a nail into a wall. What you can do to or with a real-world object could be called its *capabilities*. A hammer is capable of being swung, and it is capable of driving a nail into a wall. Real-world objects also have *properties*. Properties are intrinsic features of a real-world object, like its height, weight or color. An interesting example is that of a clock, which could be described as having the "property" of time.

A software object models a real-world object by duplicating as many of the real-world object's capabilities and properties as appropriate. A software object modeling a clock might present an image of a clock (with color, height and weight properties), as well as have code to keep the time (a clock's time property). Thus, software objects come much closer to representing the autonomous, rich real-world objects from which compelling experiences are derived.

An important point about software objects is that they can represent capabilities or properties that are abstracted from a complete real-world object. For example, a software object might keep track of time, but not have the other, visual attributes of a clock. Such an object would be a timer software object. By combining such objects that implement abstractions with objects that model more tangible attributes, you can create very sophisticated real-world—and more importantly, unreal but still coherent—models. For example, you could combine many timer software objects with other, more visual objects to create a many-faced clock object. Now, consider that you could add an object with the abstract capability of movement to create a flying, many faced clock.

In the real world, objects interact directly. You grab a hammer, or glance at a clock. In the unreal world of the computer, software objects interact through messages. Messages are the mechanism by which software objects make use of one another's capabilities or discover the state of one another's properties. A software object representing a person would send a message to a hammer software object in order to pick it up. A timer object will divulge the state of its time property when it receives a message telling it to do so. A message sent between two objects is like a very fast, structured e-mail message: it contains information about the sender, the receiver and what the receiver is supposed to do.

Software objects, although they have the same benefits of simplicity and easy interconnection as do Lego blocks, have other advantages. Because they are simply pieces of software living in a computer's memory, they can be arbitrarily created, destroyed, duplicated, renamed, reconfigured or otherwise altered.

Design Example: Objects vs. Procedures

An analogy may help to make the design issue more apparent. Imagine a program designer who wants to model the activities of taxis in a city. Using a procedural programming tool that requires scripting, the designer creates the map of the city and then must define, in

advance, all the possible paths that the taxi cabs will be allowed to follow as well as all the conditions under which the taxi cab stops, starts, breaks down, runs a red light, runs out of gas, takes on passengers and so on. Without programming that challenges even the most seasoned of professionals, it is not possible for the taxis to behave in truly novel, interactive ways.

Using a procedural model, the more dynamic the simulation, the more difficult and time-consuming the program coding becomes. Programs that use the procedural (scripting) method act like a taxi dispatcher who has to tell all the cabs exactly where to go, when to stop for gas and what to do as the simulation unfolds. To illustrate, if there is a request for a cab from a hotel in the city, the dispatcher has to query all the cabs to see if there is one that is available, then must calculate which available cab is the nearest to the hotel. If the cab is just about to run out of gas, it must be "flagged" as unavailable.

In object-oriented programming languages, the designer lays out the city and creates "taxi objects." The taxi objects are embedded with the instructions about where to go and what to do. They have their passenger status, and location, and gas gauges built into them. In this model, if a passenger calls for a taxi, the taxis with full tanks and no passengers listen for the message, measure their distance from the hotel and the taxi closest to the hotel picks up the fare.

Although it is certainly possible to model the complex interactions of natural processes in procedural languages, the more complex the interactions in the program, the more complex and inflexible the structure of the program becomes. Given that complex interactions are the heart of truly compelling new media titles, procedural languages are merely roadblocks to your productivity. The bottom line is that in object-oriented models, objects take care of themselves, leading to not only greater productivity but greater realism as well.

There are other significant productivity advantages to object-oriented approaches, which we'll discuss below.

Design Changes

Although the message-passing that occurs in object-oriented programs can be complex, object-oriented design is much more flexible and responsive to design changes.

For example, if a design change is introduced to the taxi simulation, it has a much greater impact on a procedurally-constructed program than an object-oriented one. If a decision is made to set the city in the 18th century rather than the 20th century, the simulation developed with an object-oriented tool would not have to be radically changed. The image property of the taxi could be changed to a horse. Tweak the gas gauge to be a "hay gauge," set a different value for "horsepower" (literally!) and cars are replaced by horses.

Reusable Code

The ability to take existing intellectual or creative effort and arbitrarily reapply it to some new need is a major productivity benefit of object-oriented systems.

For example, if the designer of the taxi cab simulation decides to include trucks in the simulation, the behavior of the taxi cab can be duplicated, slightly altered to suit that of a truck and added to the simulation. The taxi cab code has literally been reused to create truck objects.

Inheritance and Building Complex Objects

The rapid creation of sophisticated entities from simple constituent objects is another substantial productivity benefit of object- oriented systems. Complex real-world systems can be modeled much more easily than with any other approach.

When discussing this concept previously, the analogy of snapping together simple Lego blocks into a more sophisticated entity (such as a race car) was employed. The race car "inherits" the capability of the motor block to spin a drive-shaft, and it "inherits" the strength properties of each block. Similarly, a clock object in an object-oriented world can inherit its time-keeping capability from an abstract timer object. One of the major benefits of inheritance is that if the inherited object changes, its "heir" acquires that object's capabilities or properties. For example, if the abstract timer object is changed to have millisecond precision, the clock object now can keep time to the nearest millisecond.

Components and Encapsulation

Advanced object-oriented systems such as mTropolis provide more transparent reusability than that described above. This transparency is derived from objects that act as totally autonomous components, truly like Lego blocks. Standard object-oriented systems (such as C++) do not provide such "plug and play" objects; some knowledge of their inner workings is always required to use them.

Autonomous components depend on a feature of object-oriented systems called encapsulation. Objects "publish" what capabilities they will employ on behalf of other objects, receive messages invoking those capabilities and send back the results. Thus, they need not ever have knowledge of each others' internal specifics. Therefore, those internals—both the data that an object operates upon and the code that does the operations—can be "encapsulated," or hidden away. Encapsulation eliminates dependencies between objects that prevent rapid enhancement of individual objects, and allows objects to act as truly independent components.

Components can be reused in any context, without the user having to understand anything about their internals. For example, a crow component used on the bleak Scottish heath of a murder mystery title can be placed in a children's title. It will still caw and flap about its confines without any modification, tweaking or other effort on the part of the user. Thus, artists can use extremely powerful components without any programming knowledge.

There is one important distinction between mTropolis and low-level development environments such as C++. In mTropolis, the author combines components and connects components in conversation, to create so-

phisticated entities that themselves are building blocks for titles. This approach eliminates the tremendous, detail-oriented drudgery of C++ or SmallTalk. However, mTropolis does not allow the direct modification of component internals at the author level.

The good news is that mTropolis provides a comprehensive set of programming interfaces (APIs) called mFactory Object Model (MOM). MOM permits someone like a casual XCMD writer to happily modify or replace any of the mTropolis components, and at the same time enables a serious C++ developer to alter or override all of the core systems in mTropolis itself. See "mTropolis Object Model (MOM) User's Guide" on page B.1 of the *mTropolis Reference Guide* for additional information.

Chapter 4. mTropolis Basics

Chapter 3, "Object-Oriented Design", outlines object-oriented design philosophy. In this chapter, mTropolis' implementation of this design philosophy is presented, along with other, basic information required to understand mTropolis.

OVERVIEW: mTROPOLIS OBJECTS AT WORK

Working with software objects to create sophisticated, dynamic, interactive models of real or imaginary environments is theoretically very easy. There are only two tasks:

- Create and combine visual and abstract software objects.
- Control the interaction of the objects with simple messages.

mTropolis was created to turn this theory into practice. The people at mFactory believe that the sophisticated, "live" models at the heart of a great multimedia title—anything from a haunted house to a human body to a race-track simulator—can be crafted without frustration and tedium.

mTropolis provides a complete collection of components, facilities for creating and altering your own, and a way of connecting the objects together. This allows the author to focus on visually laying out the project and defining the messages that will bind together the project's components, without having to "reinvent the wheel" every time a simple operation, like loading and running an animation, has to occur.

All of your creative work with mTropolis components can be done visually, without intricate and time-consuming scripting. Combining components is done via dragging and dropping— binding components together in a conversation is a point-and-click process. If you want a roomful of clocks on a computer screen, you can cut and paste the clock component repeatedly with no more difficulty than using a word processor. Each of those clocks will tell time without any more work on your part. Creating your own component can be as easy as creating a new circle or square in a drawing program.

A stand-alone title is composed of the same objects you work with in the mTropolis authoring system, along with the instructions you gave them on how to interact. The only difference between what's under the hood during development, and a stand-alone title, is that the stand-alone title doesn't have dialog boxes or other interfaces that would let someone change its fundamental behavior.

ELEMENTS AND MODIFIERS: BUILDING "MEDIA OBJECTS"

The fundamental building blocks in mTropolis are called elements and modifiers. Element

components are essentially vessels for media and have built-in code for maintaining their basic state (e.g., their position). Modifiers are programming components that can be added to elements to endow them with new capabilities (e.g., collision detection). The operation of both elements' and modifiers' code can be configured through dialog boxes.

Combining multiple modifiers with elements, configuring their operation, and creating conversations between these "modified," author-defined elements is at the heart of the authoring process in mTropolis. These author-defined elements are still first-class citizens of mTropolis: they work exactly like built-in mTropolis components, and they can be freely shared and reused without any problems.

A very important aspect of mTropolis is that all of these components are "live." As mentioned previously, titles can be executed and debugged inside the mTropolis development environment. To do so, the author switches from the default editing mode in which components are being manipulated and configured to a runtime mode in which the "live" components carry out their tasks at full performance—the title actually runs inside mTropolis. See "Switching to Runtime Mode" on page 2.13 of the *mTropolis Reference Guide*.

Elements: Creating Media Objects

When the author creates a new element, it does not contain any media. The first step in evolving an element towards its final role in a title is to link it to external media. This external media can range from simple bitmaps, to animations or time-based video/audio.

Figure 4.1 shows a new graphic element as it is first created. Beside it is a graphic element that has been linked to a digital video.

Elements contain snapshots of the external media files. For example, in Figure 4.1 the first frame of a digital video is shown within the boundaries of the element. When the author switches from editing mode to runtime

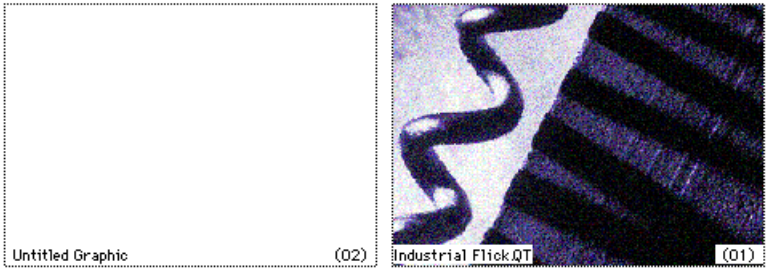


Figure 4.1 A new graphic element, and a graphic element linked to a QuickTime movie

mode, the digital video plays on the screen within the element boundaries.

The author can configure the operation of the element through its Element Info dialog box (Figure 4.2). This dialog is opened by double-clicking on the element.

The Element Info dialog options set the media's initial state: hidden or visible, paused or unpaused, cached, and so on.

The element's configuration does not directly alter the external media file. Instead, it alters the appearance and behavior of the media at runtime, such as its representation on the screen, its volume level, etc.

mTropolis supports three types of elements:

- Graphic elements, including stills, animations, and digital video.
- Sound elements.
- Text elements.

Modifiers: Customizing Media Objects

So far we've explained that element components can inherit the capabilities of modifiers. In the mTropolis environment, this process is simple and direct: an element acquires new capabilities or properties when modifiers are dragged and dropped on them (Figure 4.3).

When an element inherits new capabilities or properties from modifiers its media is not permanently altered. For example, an element containing an image of an orange fish can inherit a modifier with a special inking effect that causes the fish to appear blue. The appearance of the element is changed, but the external media remains unaltered.

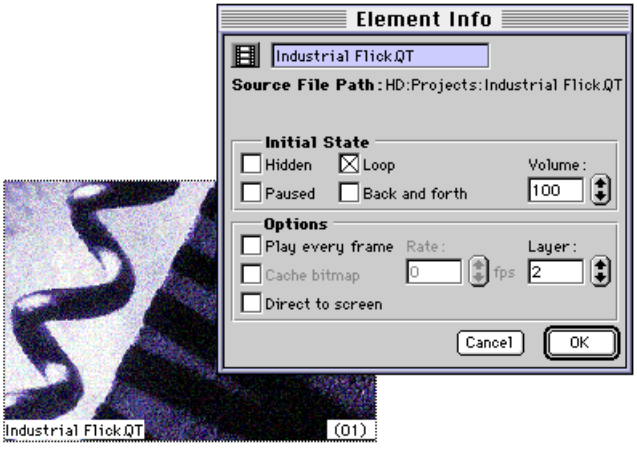


Figure 4.2 The Element Info dialog associated with a video element.

Like elements, modifiers can be configured through a dialog box. This dialog can be displayed by double-clicking the modifier. Each modifier's dialog is specific to its particular capabilities or properties. Figure 4.4 shows the dialog for configuring a graphic modifier. This dialog controls the particular properties that an element would inherit from this modifier—in this case, the appearance of its media. This dialog also contains the message pop-ups used to configure the modifier's messaging operation. These functions are discussed in more detail in Chapter 6, "Messaging".

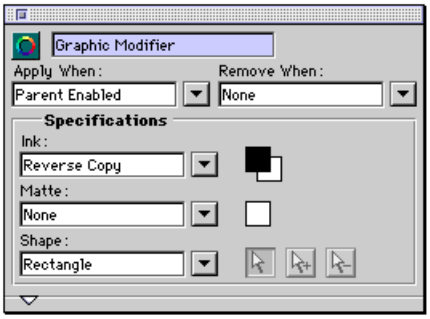


Figure 4.4 The Graphic Modifier dialog.

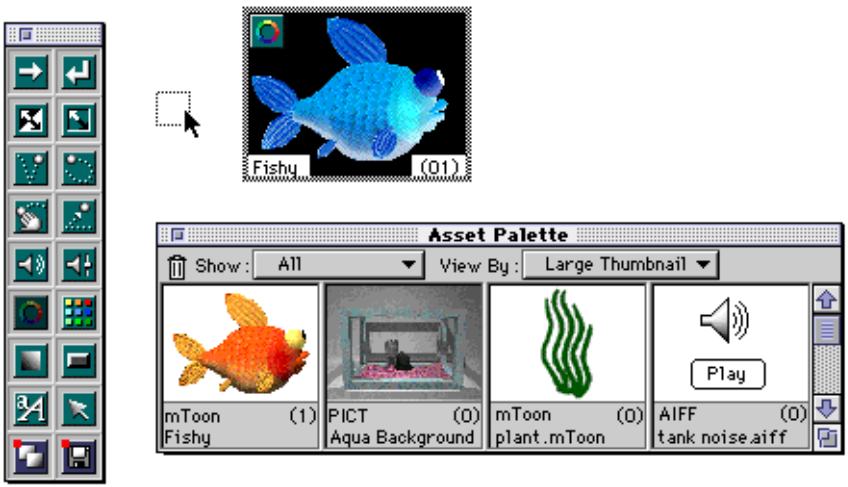


Figure 4.3 Dragging a modifier from a palette and dropping it on a graphic element.

An element combined with one or more modifiers can be thought of as a unique author-defined mTropolis component linked to a specific media file.

mTropolis comes with a very rich set of modifiers that can be combined with elements to build titles with sophisticated features. For example, one modifier provides vector motion control over elements to which it is added. Another class of modifiers, called variables, endows elements with persistent or transient properties, such as the score of a game, or the location of a hidden door. Individual modifiers are described in Chapter 12 of the *mTropolis Reference Guide,* "Modifier Reference".

The ability to easily control author-defined components is extremely valuable. The author no longer needs to think about the state of the media as the title evolves. Instead, the author works with concrete objects that literally "know how to behave." The tedious work of maintaining an object's state or checking on its operation is eliminated.

Again, we want to emphasize that these author-defined components can be freely reused—cut and pasted, duplicated, stored and reapplied—across the breadth of a title or even in entirely new projects.

MESSAGING AND USER INTERACTION

As we have previously explained, messaging is the glue that binds object-oriented systems together. As a fully fledged object-oriented development environment, mTropolis is no different. As you craft your title using mTropolis components, you bind them together with messages. This section provides an overview of how messaging works in the mTropolis environment.

The Conversational Model

The model that best describes the interaction in an object-oriented system like mTropolis is *conversation*. In a conversation, the participants interact dynamically, changing their responses according to feedback from others. More critically, in object-oriented systems the user can be an active participant in the conversation. Conversely, the same interaction in a procedural design requires that all the possible responses be predetermined—including those of the user.

The procedural or scripting model is much like a cocktail party where every single utterance is known in advance to every participant. Just as this party would be boring for you as a participant, new media consumers find titles with this predictability to be equally unappealing. Consider the taxi example from Chapter 3, "Object-Oriented Design", a taxi simulation in which all routes were predetermined would quickly bore even a 2-year-old child.

The real world, or any compelling, simulated world, is both internally consistent and yet unpredictable. The conversational model creates consistency because the format of the conversation (like choosing a language and topic at a party) is determined. The conversational model also accommodates unpredict-

ability because the content of the conversation is not predetermined. Thus, the conversational model truly reflects the real world: every conversation, from particle physics to politics, rests on some form of structured exchange between parties—the content of which is not known in advance.

This faithfulness to real-world systems—or really, any system that is internally consistent—means that you can model them more naturally and much more quickly. A sword fight is both a complex and fluid process with an indeterminate outcome. It can also be elegantly and rapidly modeled by sword and fighter components in conversation. Can mTropolis produce a sword-fight experience that will mesmerize users? We think so.

Messaging Basics

The basis of the conversational model, which mTropolis employs, is messages. A conversation between mTropolis components consists of an exchange of messages, which are like small, very fast, structured e-mail messages. Messages tell a component who is talking to it and what that other entity wants done.

As we have discussed, elements and modifiers have certain capabilities. These capabilities can be configured by double-clicking on a component's on-screen representation to call up its dialog box. Sending a message to a mTropolis component is an alternate, dynamic vehicle for invoking these capabilities during runtime. For example, the graphic modifier component in Figure 4.5 invokes its capability when it receives a message called "The light is on."

Messages are signals, or requests. These signals are acted upon by components configured to respond to them. Components that receive messages, but which are not configured to respond, merely relay the message to components that they contain. For example, an element might contain the graphic modifier depicted above. If the element received a "The light is off" message, it would pass the message to the graphic modifier. Remember, as far as the sender of the message is concerned, the element is a perfectly appropriate recipient because it inherits the capabilities of the graphic modifier.

Messages can be generated by the user (like a mouse click), by system events (like new data becoming available over a network connection), and by components themselves (like notification of collision from a collision modifier). Regardless of the source of a message, any

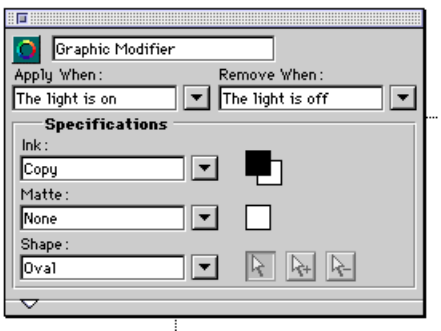


Figure 4.5 A graphic modifier for switching a "light" on and off.

component can be configured to respond to it. Special modifier components, called messenger modifiers, can be used to generate abstract messages (e.g., a "The power is off" message) to control the flow of events in a title.

mTropolis, in general, provides very powerful but uncomplicated facilities for controlling the scope and flow of message conversations. For example, a message can be sent to every single component in a certain portion of a project, or just to a single component. One extremely useful feature of mTropolis is that authors can define messages using any text string for the message name. mTropolis maintains a dictionary of these author-defined messages so that they can be applied at any time.

Benefits of the Messaging System

One of the major benefits of the mTropolis messaging system is that it makes reusability a snap. As an author combines elements and modifiers into sophisticated author-defined components, they are also defining the messages that those components use to communicate. Consider an author-defined balloon element that contains a balloon image, a collision detection modifier and a motion modifier. The motion modifier causes the balloon element to drift around the screen and the collision detection modifier detects any collisions with sharp objects.

The motion modifier might be activated upon receipt of a message called "There is Wind." The collision detection modifier might report a collision with "I'm Popped!" To effectively reuse the balloon component in any new title, someone unfamiliar with the balloon component would only have to understand these messages.

For example, an author could duplicate the balloon component ten times and place the copies in a jet fighter title for comic relief. When a fighter (itself a sophisticated author-defined component) launched, it might generate a "There is Wind" message to simulate jet-wash. Upon receiving the message, the balloons would begin to drift. If any of the balloons encountered the sharp radar probe on another fighter, the balloon in question would generate an "I'm Popped!" message. The author could have this message signal a sound component to play an explosion, or to trigger an animation of the balloon deflating.

Another important benefit of mTropolis' messaging is that user interaction with a title can be extremely rich and dynamic. Components that not only "know" how to interact with each other, but also with the user, can be dynamically introduced under title control or user control. Consider a game like SimCity in which the user is constantly introducing different objects, each with its own rich behaviors. In mTropolis, the author would simply create components with the desired behaviors. At runtime, the user could introduce as many of those components as the game permitted, and they would dynamically interact with each other and with the user without any prior scripting or additional programming.

mTropolis' messaging system also enables rapid prototyping and pain-free design changes for even the most demanding titles. Consider a real-time 3D polygon game with fighter bombers. A bridge could have regions of "intelligent" polygons that animated differently in response to different messages. For example, the steel supports of a bridge might respond to a "Really Big Bomb" message by animating a shattering process.

The author could add a new weapon with its own message, a "Really Hot Bomb." Without disturbing the existing aspects of the bridge's behavior, the author could have the steel supports animate themselves melting/twisting in response to a "Really Hot Bomb" message. Of course, this use of messaging also illuminates the incredible realism and detail that mTropolis makes possible with its intelligent components and messaging system.

mTropolis' object-oriented system allows titles to be analyzed, designed and implemented at the same time, saving enormous development time. Although a complex, event-driven system can be modeled in a procedural or command-based authoring tool, mTropolis' messaging system makes the interactions in the system much easier and faster to prototype and test.

STRUCTURE IN mTROPOLIS: A HIERARCHY

As titles become increasingly sophisticated, the complexity of their internals also increases. mTropolis provides a unique facility for managing complexity in even the largest, most involved titles. This facility is the mTropolis containment hierarchy, the same hierarchy that you can view and rearrange in the mTropolis structure view. This section explains the containment hierarchy and how it helps increase your productivity.

The mTropolis Containment Hierarchy

Containers are mTropolis components that can literally contain other objects within them, just as a paper bag can contain anything inside it from other paper bags to a sandwich. An element is an example of such a component, since it can contain modifiers. When a container object holds another object within it, the container object is called a *parent* and the held object is called a *child*.

Think of a mother kangaroo containing her child within her pouch. If the child component in turn contains another component, the child component is considered the parent of the object it holds. A container ship can be the parent of the containers it holds, which in turn are parents to the boxes that they hold, which in turn are parents to the Energizer Bunnies in the boxes.

This chain of parents and children is called a container hierarchy. We use the word hierarchy to mean one branch of a tree, like only the maternal branch of parents and children in a family tree. In a container hierarchy, just like a family tree, it is possible for a parent to have more than one child. In mTropolis, children of the same parent container are called *siblings*. By the way, one of the best ways to represent one branch of a tree is an outline—which is

why the structure view is presented as an outline (Figure 4.6).

Another important aspect of containers is that they are endowed with all of the capabilities of all of the components in their container hierarchy. In other words, containment is equivalent to inheritance. Consider a truck component that contains an engine component that contains an oil reservoir component. The oil reservoir component has the capability to be filled. Because the truck component is the ultimate parent in the container hierarchy, it can receive a "fill me with oil" message on behalf of the oil reservoir component. As far as a component external to the truck component's container hierarchy is concerned, the truck component has the capability to be filled with oil.

Now that you understand what a container is, you will also understand that a modifier is *not* a container. Modifiers are placed in containers, where they do work on behalf of the container—sending messages to and receiving messages from other modifiers (generally inside other containers). Whatever capabilities a container may have are derived from the various modifiers placed in its container hierarchy.

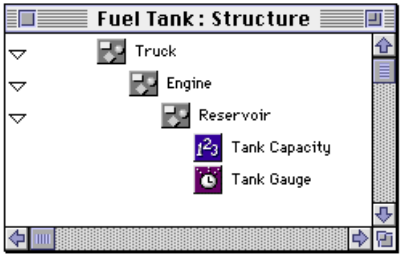


Figure 4.6 A sample Structure View.

If you think back to our author-defined balloon component example earlier, the balloon element contained various modifiers, such as a motion modifier. The capability of the balloon component to float around the screen depended on its containership of the motion modifier, which provided that capability.

Structural containers can be used by the title developer to group the various contents of a title into organized parts, like the chapters of a book or the acts in a play. Some structural containers, such as scenes and elements, can also contain raw media, such as pictures, sounds or animations.

The mTropolis project itself is a structural container that contains section containers, subsection containers, scene containers and element containers. And, of course, all containers can contain modifiers. This hierarchy, beginning with the project, is the complete container hierarchy of a project that you access through the structure view.

Behaviors

A *behavior* modifier is a special container that can hold other modifiers inside it. Behaviors can be used to group collections of modifiers that work in close concert into "supermodifiers" that provide more sophisticated operations than single modifiers—hence the term behavior. Behaviors, like other modifiers, interpret messages. The primary use of messaging a behavior is to collectively enable or

disable the group of modifiers contained within it.

This feature of behaviors is intended to help authors manage complexity by gathering together cooperating modifiers under one roof. Powerful behaviors can be dragged and dropped as needed, either from libraries or from different sections of a project. In general, behaviors promote clean reuse of logic and enable programmers to deliver sophisticated title operations to artists in drag-and-droppable packages.

Another special feature of a behavior is that it can contain another behavior within it, and that child behavior can be parent to another behavior in turn, to an arbitrary depth. This feature permits the creation of an extremely sophisticated behavior at the top of a container hierarchy of behaviors.

Consider the behavior of a variety of pests: they avoid light. But cockroaches also run from light only under certain conditions. A light avoidance behavior could be contained within a cockroach behavior, selectively switched on under certain conditions, to quickly, simply and easily model a cockroach.

It is important to remember that this behavior containment hierarchy is a part of the project containment hierarchy that can be inspected and altered through the structure view. Part of the power of the project containment hierarchy is that is contiguous and all-embracing.

Messaging and the Containment Hierarchy

The containment hierarchy fulfills an important function besides providing a framework to organize the inclusion of components within one another. Each successively higher level of the container hierarchy represents a higher level of abstraction in the project, an arrangement that actually helps you direct the flow of messages quickly and easily, even in complex projects.

Consider a project for a children's edutainment title on physics. Because gravity and friction are constant presences in the physical world, you might place modifiers for those characteristics at the project level. Thus, the project would contain two modifiers and, as well, perhaps sections representing different experiments or rooms in a virtual physics lab.

Any good tutorial should show what happens when constants are changed. If the child clicks an "antigravity" switch in a room of the title, gravity should switch off. Of course, in a language like C++, you would have to send a message directly to the gravity modifier. That also implies that you remember exactly where the modifier is. In mTropolis, you have much more flexibility in messaging, thanks to the containment hierarchy.

First, the containment hierarchy enables message "broadcasting." You could simply search the message list for "gravity," locating the "Turn off Gravity" message. Then you could literally target the project, and mTropolis would "cascade" the message from the project level on down through the entire containment hi-

erarchy. Any component, contained anywhere in the hierarchy, capable of responding to the "Turn off Gravity" message would do its stuff. In this case, only the gravity modifier contained by the project would act on the message.

The advantage of broadcasting is that you can cause global changes without laboriously specifying each and every component that needs to act on a message. In the example above, if gravity modifiers were scattered throughout the containment hierarchy, they would turn themselves off without any further work on your part. On the other hand, you may not want to cause entirely global changes. Fortunately, the containment hierarchy enables more precise targeting of messages.

Broadcasting is simply the most general case of what is called "message targeting" in mTropolis. You can also "target" a message at any level of the containment hierarchy from a single, indivisible modifier at the very bottom to some constrained portion of the containment hierarchy. In the preceding example, you could target the "Turn off Gravity" message to only a section of the project. The message would cascade only through the section, down to scenes representing particular rooms, onto the elements and modifiers in those scenes, and nowhere else in the project as shown in Figure 4.7.

Basic Rules of the mTropolis Containment Hierarchy

Remember the following basic "rules" of the mTropolis containment hierarchy:

- Containment is equivalent to inheritance; as far as any entity outside of a container is concerned, the container has all of the capabilities of whatever it contains.
- All mTropolis components are a container except modifiers.
- All containers can contain modifiers and behaviors.
- All containers can contain other containers; however,
- Only elements and behaviors can contain other containers like themselves.
- All mTropolis objects can be the target of a message, but a modifier (or the system/user) must be the originator of the message and another modifier will actually process

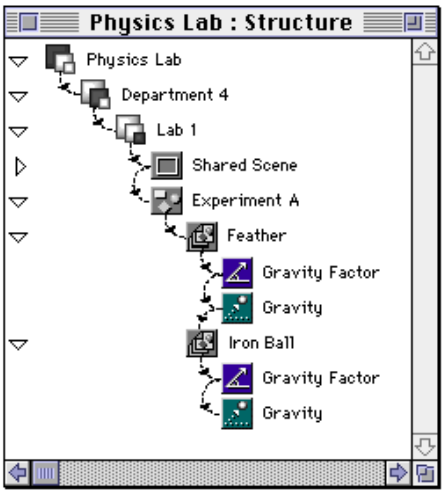


Figure 4.7 Message passing between components.

the message and do the work. The only exception is that elements can directly receive command messages to change their basic appearance. For example, an element containing a QuickTime movie can be directly commanded to play the movie.

Chapter 5. mTropolis Components

This chapter discusses mTropolis components, how they work and their role in the mTropolis containment hierarchy.

THE ELEMENT COMPONENT: PUTTING IT IN CONTEXT

The fundamental building block of a title is an *element*. An element can be linked to an external media file, to display still images or play time-based media. Elements have built-in code for maintaining their basic state (e.g., the element's position) and controlling the appearance of media they contain. This section discusses elements just enough to help place the other mTropolis components in context. We'll discuss elements in more depth later in this chapter.

Elements and External Media

The author creates elements and links media to them. The appearance of an element changes to indicate the media to which it is linked. For example, if an element is linked to a QuickTime movie, a thumbnail from the first frame of the QuickTime movie is shown within the element's boundaries. Elements can be resized and positioned in the layout view.

There are three basic types of media elements in the mTropolis environment:

Graphic Elements

Graphic elements can be linked to images (e.g., PICTs), digital video (e.g., QuickTime movies), and animations (in mTropolis' proprietary animation format, called "mToons"). Figure 5.1 shows three graphic elements linked to different types of media.

Sound Elements

Sound elements can be linked to sound files (e.g., AIFF, AIFC, and snd files).

Text Elements

Text elements cannot be linked to external text files. However, text can be entered into text elements and formatted within mTropolis.

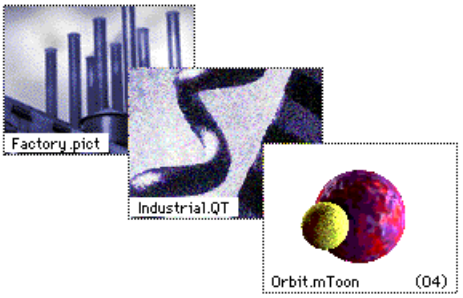


Figure 5.1 Graphic elements linked to a PICT, QuickTime movie, and mToon.

ELEMENTS AND THE CONTAINMENT HIERARCHY

Elements are always contained by other components, either other elements or scene components. When a new project is opened, mTropolis provides a project component (named "Untitled-1"), a section component (named "Untitled Section"), a subsection component (named "Untitled Subsection"), a shared scene component (named "Untitled Shared Scene"), and a scene component (named "Untitled Scene"). These components are described in more detail below. Figure 5.2 shows a structural view of a new mTropolis project.

Project Components

A *project* component is a structural container that holds an entire title within it. The children of a project are *sections*. A project can only contain sections and modifiers. The project's ability to contain modifiers is useful for creating modifiers that modify the actions of the entire title. Projects cannot contain other projects.

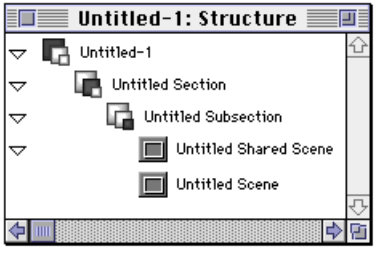


Figure 5.2 Structure view of a new mTropolis project.

Section Components

A *section* component is a structural container that can be used by the title developer to gather different chunks of a title into groups that logically belong together. For example, a title developer for a travel game might put everything to do with Africa under a single section. A section is most closely analogous to an act in a play or a chapter within a book. Sections are parents to *subsections*. A section can only contain subsections and modifiers. Sections cannot contain other sections.

Subsection Components

A *subsection* component is a structural container that can be used by the title developer to divide the content of a section more finely, literally like a subsection in a book. For example, the title developer for a travel game might put everything to do with the country of Kenya in a subsection, the parent of which would be the Africa section. Subsections are parents to *shared scenes* and *scenes*. Subsections can only contain a single shared scene, multiple scenes, and modifiers. Subsections cannot contain other subsections.

Shared Scene Components

The *shared scene* component is a structural container that is a peer of *scene* containers. It is used to contain elements common to all scenes in a subsection. A music or voice track, for example, might be placed in the shared scene. Background images common across scenes in a subsection can also be placed in the shared scene. In our African travel project, a shared scene would maintain

the appearance of the plains across different Kenyan scenes, as well as providing common background music.

Shared scenes can only contain elements and modifiers. Shared scenes cannot contain scenes or other shared scenes. Also, only one shared scene is ever present in a subsection. Shared scenes can also have graphical media assets linked directly to them.

In terms of layer order (as you can inspect with the layer view), shared scenes are drawn by default behind the scenes that are their siblings. This convention stems from the fact that shared scenes are intended to provide common backgrounds for scenes.

Scene Components

A *scene* component is a structural container that is very much like the scene in a play. As a scene in a play contains all the props and actors required to convey some discrete piece of interactivity to an audience, a scene in a mTropolis title contains all the components to do the same for a user. A scene presents a microcosm—like an African watering hole or a room in a haunted house—that contains child components for all of the props and actors in that microcosm.

Scenes can only contain elements and modifiers. Scenes cannot contain other scenes. Note, however, that there is no limit to the number of scenes that can be present in a subsection. Like shared scenes, scenes can have graphical media assets linked directly to them.

Element Components

An element component is a structural container that is the workhorse of mTropolis. Elements can be linked to raw media such as pictures, sounds and animations.

Elements can also contain modifiers, which help to bring the raw media they contain to life. Consider a scene representing an African plain. The title author would use elements containing pictures of dry grass and trees to create a compelling image of the plain. Now consider that the author wants a vulture to fly around the plain.

An element simply containing a picture of a vulture, or even containing various frames to animate the vulture's wings, will not accomplish the objective of making the vulture fly. The vulture element needs to contain a motion modifier component. Once the author drops on that component, the vulture element would be endowed with the motion capabilities of that motion modifier. The vulture could be set to follow a preprogrammed path, or to move about randomly as it iterated through its wing animation. Now consider that the tree elements of the plain could contain collision detection modifiers. The tree elements would then have the ability to detect a collision with the vulture element.

Elements can also be the parents of other elements. Why would an element contain another element? Consider the vulture described above. The wings of the vulture might have been created separately from the vulture's body, so the author might receive

the vulture as three separate elements created by the art department. The vulture's body needs to carry the wings along the same path that it travels. The simple solution is to attach the wings to the body by containing the two wing elements inside the body element. The final, collective vulture element will move along the same path.

HOW GRAPHIC COMPONENTS ARE DRAWN

Only shared scenes, scenes and elements are actually drawn on the screen by mTropolis. This section contains some basic information on how these components are drawn.

Elements contained by a scene draw by default on top of the scene and are contained within its boundaries. Scenes are drawn by default on top of the shared scene that services them.

Elements are automatically assigned a layer order when they are added to a scene. The layer order of new elements increments as they are added to the scene. Each layer contains only a single element. The layer order can be changed dynamically under program control. Note that layer order is independent of the parent/child relationships of elements in a scene. Layer orders are described in more detail in Chapter 10 of the *mTropolis Reference Guide,* "Layers Window".

By default, mTropolis builds the scene off-screen in memory before it is presented to the viewer. The viewer does not see each element added to the scene, but rather views the result of layering one element on top of another.

In this so-called "2.5D" approach, elements always appear above elements with a lower draw order and below elements with a higher draw order when they are redrawn. Because they are properly clipped, this approach cre-

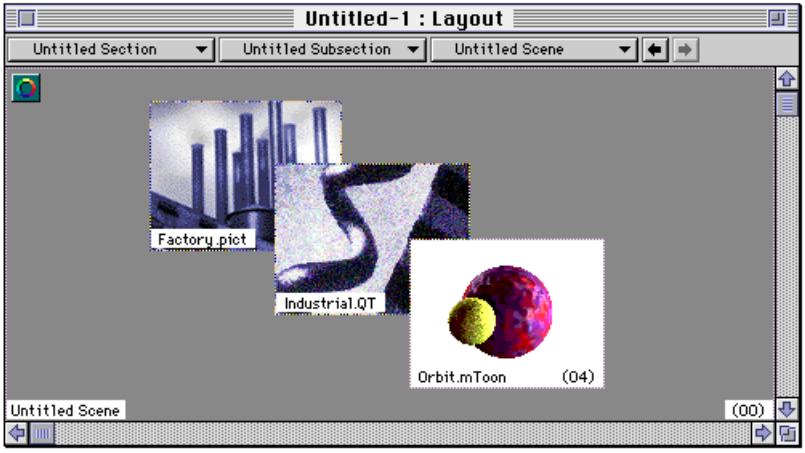


Figure 5.3 The "2.5 D" effect of layer order.

ates the illusion of a 3D perspective. Figure 5.3 illustrates how layer ordering affects graphic elements.

mTropolis provides the option of displaying elements "direct to screen", turning an element's draw order off. This feature is useful when you want an element to be always drawn on top of everything else on screen.

MODIFIERS

Modifiers are special mTropolis components that modify the properties of other components in a project. Some modifiers are built into mTropolis, but new ones can be plugged in so seamlessly that they are indistinguishable from the built-in modifiers.

Modifiers are used by dragging them from one of the modifier palettes and dropping them on the object that they are to modify. Each modifier on the modifier palettes has unique capabilities or properties. When a modifier is dropped onto a component, the component assumes these capabilities or properties.

For example, a gradient modifier has the ability to alter the visual characteristics of graphic elements. When dropped onto a graphic element, the gradient modifier's capabilities are added to the information that makes up that object, as shown in Figure 5.4.

While some modifiers have the ability to change the visible characteristics of the elements onto which they are placed, other modifiers change invisible characteristics, or *properties,* of the element that contains them. For example, when a point variable modifier that contains the value 2.5 is placed on an element, the physical representation of the element does not change, but its content, the value 2.5, becomes an intrinsic part of the element that contains it.

All modifiers can be configured (i.e., their capabilities can be customized) by changing the default settings in their modifier dialogs. In addition, most modifiers can be configured to apply their effects at specific times through a process called messaging.

A message in mTropolis can be as simple as a mouse click, or as complex as an author-defined message that is generated only after specific conditions in the runtime environment have been met. Some messages are generated by mTropolis during runtime and automatically sent to specific components throughout the project, and others can be sent to components from special modifiers called messengers.

Complete information on mTropolis modifiers can be found in Chapter 12 of the *mTropolis Reference Guide,* "Modifier Reference".

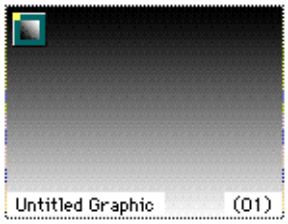


Figure 5.4 The gradient modifier, placed on a graphic element

Behavior Modifiers

A special type of modifier is worth mentioning here. A *behavior* is a special component in the mTropolis environment. It can be used to encapsulate (i.e., contain) groups of modifiers and other behaviors.

Behaviors can be used to hierarchically group collections of modifiers that work in close concert. Each collection can be enabled or disabled with messages, creating "super modifiers" that provide more complex operations than single modifiers alone.

The power of behaviors lies in the fact that they can be made switchable. That is, they can be turned on or off with messages. When a behavior is switched off, all of the modifiers it encloses are disabled. When a behavior is switched on, individual behaviors or modifiers within a behavior can then be activated by incoming messages. This feature allows the author to create and control components with very sophisticated behaviors and capabilities.

A comlete description of the behavior modifier can be found in "Behavior" on page 12.13 of the *mTropolis Reference Guide*.

Aliases

One powerful mTropolis feature is the ability to make a special copy of any modifier (including behavior modifiers), called an *alias*. Creating an alias makes a "master copy" of a modifier and places it on mTropolis' Alias Palette. Modifiers on the alias palette can be dragged and dropped onto any element in a project. All modifiers placed in this way share the same settings and all aliases of a modifier can be updated by editing any instance of that modifier.

This feature is useful in complex projects where a modifier may be employed in an identical fashion in many different sections of the project. For example, in an adventure game, a graphic modifier might apply the same effect to images that represent stone walls throughout the game. During the authoring process, changing the settings of many identical modifiers can be time consuming. Similarly, sending many messages during runtime to identical modifiers could also be time consuming.

Using aliases of the graphic modifier in our example would permit the author to make a change to either the original or any of its aliases, and that change would instantly occur in all copies of the modifier.

Aliases can be very powerful when used with behavior modifiers. Dropping a new modifier into an aliased behavior causes all instances of that behavior to be updated.

For complete information on creating and managing aliases, see "Alias Palette" on page 11.7 of the *mTropolis Reference Guide*.

Chapter 6. Messaging

Chapter 5, "mTropolis Components" outlined the role that components play in the mTropolis authoring environment. This chapter focuses on the messaging relationships between components.

ACTIVATING ELEMENTS AND MODIFIERS

As we discussed previously, messages are sent and received by modifiers at various levels of the project container hierarchy. Modifiers can also receive messages from the mTropolis runtime environment itself, such as network events or user mouse events. Elements do not send messages, but they can receive certain special messages directly from modifiers. Behaviors, while they do not send messages either, can be switched on or off by messages.

Messages are essentially signals that tell elements and modifiers to engage in some operation. Consider the graphic modifier depicted in Figure 6.1, placed on some arbitrary element.

In this example, the graphic modifier listens for a Mouse Down message. When that message is sent to it, it turns the element blue. When it receives the Mouse Up message, it returns the element to its default color (black).

An important point about messages is that they are always available to the author, regardless of whether they are associated with some specific environmental event. In the example above, the Mouse Down message could have been sent to the graphic modifier from the mTropolis environment in response to a user mouse-click. However, it could also have been sent by a messenger modifier configured to send a Mouse Down message in response to some condition. The ability to simulate external events under program control is particularly useful for debugging and testing.

MESSENGER MODIFIERS: BUILDING LOGIC

Messenger modifiers (often referred to simply as "messengers") are dedicated to sending

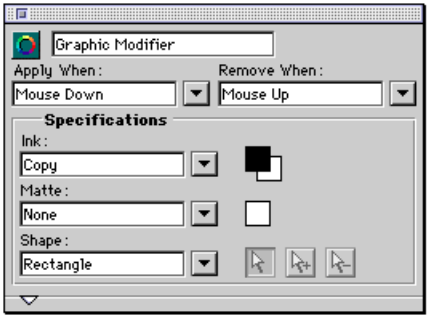


Figure 6.1 Graphic modifier dialog configured to activate on Mouse Down and deactivate on Mouse Up messages.

and receiving messages. Messengers are the glue that implements the abstract logic of a mTropolis title. For example, a timer messenger listens for a message and delays for a selected period of time. Then it sends another message out.

The timer messenger dialog shown below (Figure 6.2) illustrates the full power of messaging in mTropolis. Through their dialogs, messengers can be configured to send specific messages, to specific destinations with any data that the author wants to send and receive. Messaging is a powerful, but simple process. The four "W's" of messaging—when, what, where, and with—are described below.

When
The timer messenger, like most modifiers, has two "When" pop-up menus. When the modifier receives the message selected by the "Execute When" pop-up, the modifier will activate as it has been configured to do so. When the modifier receives the message selected by the "Terminate When" pop-up, it will return to an inactive state.

What
The "Message/Command" pop-up is used to select the specific message to be sent when the messenger is activated.

Where
The "Destination" pop-up is used to select where the selected message will be sent.

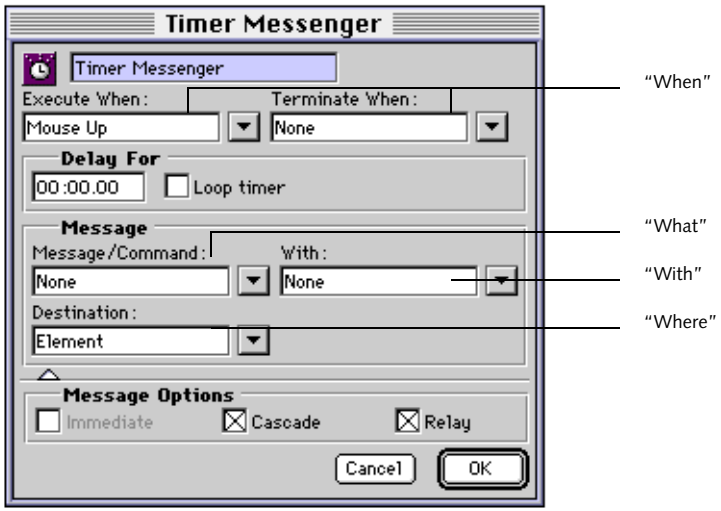


Figure 6.2 A typical messenger dialog.

With

Optionally, data can be sent with a message. The "With" pop-up is used to select a variable or constant value to send.

Each of the pop-ups will be described in turn.

"When" and "What" Message Pop-Ups

The 'When' and 'What' pop-ups deal with the same entity—messages—so we'll describe them together.

Together the "When" pop-ups control the conditions under which the messenger (or any modifier, for that matter) will operate. The first "When" pop-up selects the message that will activate the messenger. It can be any arbitrary message, either author-defined or predefined, just as with any other modifier. The second 'When' pop-up selects the message that will return the messenger to an inactive state, just as with any other modifier.

The "What" pop-up is distinct from the "When" pop-ups in that it determines the *output* of the messenger. The message selected by the "What" pop-up will be sent when the messenger activates. This message can be any message available in the mTropolis environment, either author-defined or predefined. And, as mentioned above, this message could be a simulated environment message, such as a Mouse Down.

See "The 'When' Pop-Up Menu" on page 13.1 of the *mTropolis Reference Guide* and "The Message/Command Pop–Up Menu" on page 13.2 of the *mTropolis Reference Guide* for complete information on these menus.

"Where" Destination Pop-Up

The "Where" pop-up selects the destination, or target, of a messenger's output message. The ultimate target must be another modifier, an element or a behavior; however, these components can be anywhere within a project's containment hierarchy. In the case of a modifier or behavior, they could be nested within a behavior as well.

Container Hierarchy and Messages

As explained above, the destination for a messenger's output message can be either a specific component, or an arbitrary level of a project, or a behavior containment hierarchy. mTropolis will automatically handle cascading the message down through the containment hierarchy to the elements, modifiers or behaviors that might be listening for the message that was sent. Remember, we explained the power of the containment hierarchy for controlling the flow of messages "Messaging and the Containment Hierarchy" on page 4.10.

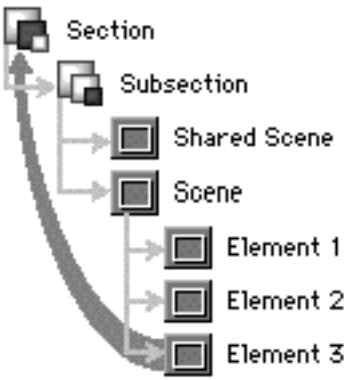
Some examples of possible message destinations in the container hierarchy:

- The element that contains the messenger modifier (i.e., to its immediate parent in the containment hierarchy):

- Another modifier contained by the same element (i.e., to its sibling in the containment hierarchy):

- To another element (i.e., to a sibling of its parent in the containment hierarchy):

The illustration below depicts a messenger sending a message to an arbitrary level in the containment hierarchy and how that message cascades down to the eventual recipients:



In the preceding illustration, the section is the container (and parent) to the subsection, which is in turn the container and parent to the scene, in turn the container and parent to elements 1, 2 and 3. A messenger contained by element 3 could target its message at the section. That message would cascade down to every component in the section's portion of the containment hierarchy.

There are two points to make about this use of the containment hierarchy for messaging. First, the message sent by the messenger goes directly to the section and does not travel up the containment hierarchy. Second, the message, as it cascades down the containment hierarchy, is only acted upon by modifiers that have specified in their "When" menus that they wish to be activated by the message in question. All other components ignore the message.

A very powerful feature of messaging in conjunction with the containment hierarchy is relative message targeting. You have seen that you can specify some abstract level of the containment hierarchy as a target, and that mTropolis will handle all of the details of cascading the message to the possible recipients.

Relative message targeting enables you to target other components by their relation to a modifier, rather than their specific names or positions in the containment hierarchy. For example, you could specify that you want a message sent to a modifier's parent in the containment hierarchy, or its parent's parent, all the way up the hierarchy. Or, you could send a message to its parent's sibling.

Behavior Containment Hierarchy and Messages

As we mentioned in our discussion of behaviors in "Behavior Modifiers" on page 5.6, behaviors can contain modifiers, as well as be nested inside of other behaviors. This behavior containment hierarchy is a part of the project containment hierarchy, inspectable and alterable through the structure view. The following illustration shows the relationships between behaviors, modifiers and an element.

Behavior 1 is contained by the element. Behavior 2 is contained by Behavior 1. Behavior 2 can target the modifiers contained by Behavior 1. In order for the modifier contained by Behavior 2 to send a message to the modifier on the element, it must target the element.

The next illustration shows the way a message is broadcast to an element that contains both a behavior and another element.

Element 2 is contained by Element 1. A message broadcast to Element 1 cascades successively down the containment hierarchy.

Behavior Dialog

Figure 6.3 shows a behavior on an element, with the editing dialog for the behavior open to show the modifiers it contains.

Notice that the behavior dialog box shows the order of modifiers (1, 2, 3) and the particular messages that activate them (Mouse Up, Switch On, and Parent Enabled). mTropolis processes messages in the order that they are received, and looks for the first modifier that will respond to the message currently being processed.

In Figure 6.3, a Light Switch behavior provides a concrete demonstration of how behaviors can cleanly group cooperating modifiers into a high-level behavioral compo-

nent. The behavior in this example contains the modifiers that would be activated when the user clicked on a light switch element. However, once created, the light switch behavior can be dragged onto other switch elements throughout a project. Behaviors truly promote easy and painless reusability.

We have previously mentioned that behaviors, like modifiers, can be activated and deactivated through the receipt of messages. When checked, the Switchable check-box in the behavior dialog box (Figure 6.3) enables a behavior to be controlled in this fashion. The left "When" pop-up specifies the message that will enable the behavior and the right "When" pop-up specifies the message that will disable the behavior.

If a behavior is deactivated, all components within its chunk of the containment hierarchy are effectively "deaf," not listening to any messages.

This "switchability" of behaviors is especially powerful when behaviors are nested within each other. As we discussed previously, behaviors can be nested so that they fire activation and deactivation messages downwards in a sophisticated cascade, almost like a neural network. This "gating" of nested behaviors can be used to model very complex logic in a manageable and reusable manner.

See "The Destination Pop-Up Menu" on page 13.21 of the *mTropolis Reference Guide* for complete information on the destination menu.

"With" Menu: Sending Data

The "With" message pop-up allows a messenger to pass data along with the message out-

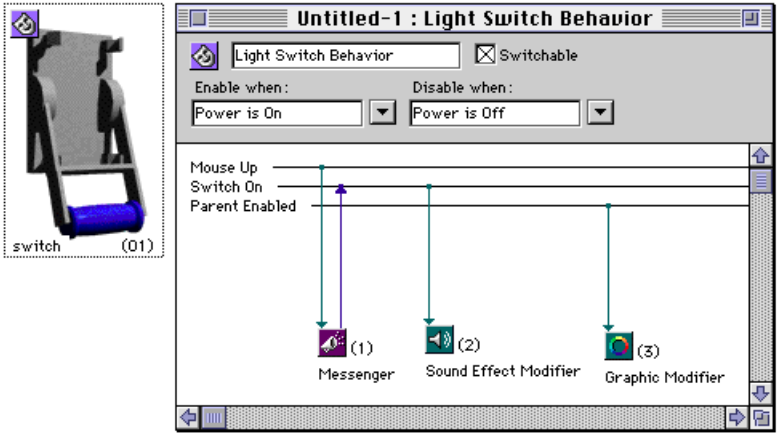


Figure 6.3 A behavior modifier dialog.

put by a messenger. For example, information about the current element's screen position, where the user clicked on the screen, or current values of variables could be sent along with the message.

A messenger can send no data (the default selection); it can send a constant value (entered in a dialog box while configuring the messenger's operation); it can relay the value that it, the messenger, received from the message that activated it (the "incoming data"), and it can also send the contents of any variable modifier accessible to it.

See "The "With" Pop-Up Menu" on page 13.20 of the *mTropolis Reference Guide* for complete information on this menu.

TYPES OF MESSAGES

Messages in mTropolis can be divided into two types. The first type, which includes all author-defined messages and the majority of all built-in mTropolis messages, act as signals or conveyors of information. For example, the Mouse Down message signals the recipient that someone clicked the mouse. These signal messages can be subdivided into two types, "environment messages" and "author messages".

The second major type of message is an imperative, or command. Sending a command message constitutes a demand that the recipient perform some action, and it cannot be ignored. Command messages (called commands) are primarily used to control the behavior of elements; while modifiers can send commands, the command messages have no specific meaning to them. For example, the *Play* command would immediately cause a modifier's element to play any time-sensitive media that it contained; however, the play command is simply another message to which the modifier can be configured to respond.

This section explains the various types of messages available for use in mTropolis. More information on these messages can be found in "Environment Messages" on page 13.2 of the *mTropolis Reference Guide*, "Author Messages" on page 13.4 of the *mTropolis Reference Guide*, and "Commands" on page 13.4 of the *mTropolis Reference Guide*.

Commands: Control Signals

Commands appear in the "What" menus as *italicized* text to distinguish them from other messages. Figure 6.4 shows one of the cascading menus available from the Message/Command pop-up menu.

Commands are different from signals in the following two respects:

- Commands act directly on elements. Elements do not have to be configured to "hear" commands, and they respond to them immediately without interpretation. For example, there are commands that can be sent to a digital video to play or hide, or to a still image to show or hide.

  Since commands act directly on elements, they do not affect modifiers. The author cannot, for example, command a modifier to *Play* or *Pause*. However, modifiers can

receive and interpret commands, or pass them on to elements if directed to do so.

- Commands are like any other message in that they can be targeted to a specific element, to a specific level of the containment hierarchy, or to relatively positioned elements. While the command acts on the recipient element immediately, its result is passed on to modifiers within that element as a signal message. For example, if the command *Pause* is sent to an element, the modifiers within it could listen to and act on the resulting signal, which would be "Paused."

Here are some examples of commands:

- *Play:* Play the animation or digital video from the first cel in its range.
- *Stop:* Stop the animation or digital video and hide it.
- *Close Project:* Quit the title.

Chapter 13 of the *mTropolis Reference Guide,* "Modifier Pop-Up Menus and Message Reference" for a complete list of commands.

Author Messages and Environment Messages

While command messages are imperatives that elements cannot ignore, signal messages inform modifiers that an event has occurred. If the modifier is listening for that information, it acts upon it.

Signal messages fall into two types: author messages and environment messages.

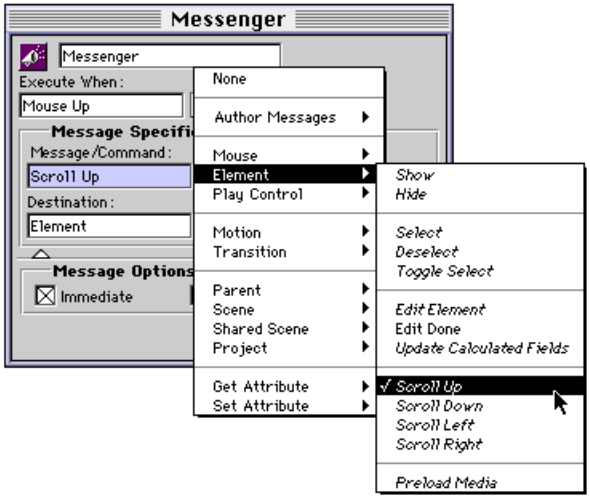


Figure 6.4 Commands in the Message/Command menu's "Element" section.

Author Messages

Author messages are defined by the author by entering the text of a new author message in the "When" pop-up of a modifier. mTropolis will ask the author if he wishes to create a new author message.

Author messages are never sent by the system, they are only sent by messenger modifiers, under the author's control. However, any modifier, through its "When" pop-ups, can be controlled by any author messages.

Environment Messages

Environment messages appear as options on the message menus. Examples include:

- Mouse Down: the mouse button was pressed while the cursor was over an element.
- At First Cel: the first cel in an animation contained by an element has been reached.
- Motion Started: an element has started moving.
- Scene Ended: the scene has ended.

While mTropolis itself sends environment messages to modifiers that are listening for them, mTropolis does not have a monopoly on the use of environment messages. As we mentioned previously in this chapter, the author can send environment messages from a messenger at will. For example, an author wishing to test some user interaction logic could emulate a mouse event by simply sending a Mouse Down message from a particular messenger.

Chapter 7. Tutorial

This tutorial provides a general introduction to mTropolis. It is intended for new users who want to get a feel for what mTropolis can do and how to use the mTropolis authoring environment. In this tutorial, you'll create a multimedia "puzzle". The process of authoring in mTropolis is demonstrated step-by-step, beginning with adding media to the first scene.

WHAT YOU'LL NEED

- mTropolis must be installed on your machine. Installation instructions can be found in the "Read Me First!" file on the mTropolis CD-ROM.
- The tutorial files are installed by default when mTropolis is installed. Tutorial files can be found in the "mTutorial" folder of the mTropolis installation. If the tutorial has not been installed, it can be installed by running the installer from the mTropolis CD-ROM, or by dragging the "mTutorial" folder from the CD to your hard disk.
- For best performance, make sure your monitor is set to display 256 colors.

Tutorial Project Description

Let's begin by looking at the completed puzzle project.

- Open the completed tutorial project into mTropolis by dragging the **Completed Tutorial** icon, found in the **mTutorial** folder, onto the mTropolis icon. If you have multiple versions of mTropolis installed (e.g., both the 68K and PPC versions), be sure to select the correct one for your machine.
- If a dialog appears prompting to search for the media files, click on the Search button. A folder selection dialog appears. Use it to select the "mTutorial" folder on your drive. mTropolis will find the media files it needs in the subdirectories of this folder. The mTropolis interface appears with the completed tutorial project shown in its layout view.
- The project is shown in *edit mode*, where changes could be made to the project. To view the project as a user would see it, press ⌘-T to switch to *runtime mode*. The project will run from its first scene.

The first scene of the project shows a QuickTime movie of mFactory's "M" logo being drawn on a napkin. When the movie is finished playing, a new scene appears.

The second scene (Figure 7.1) shows pieces of a puzzle spread randomly about the screen. The pieces can be dragged around the screen. If a piece is dropped near its correct position on the backdrop, it "snaps" into place with an audible "clang". The pieces form a machine that looks like the mFactory logo. When all the pieces are in their proper

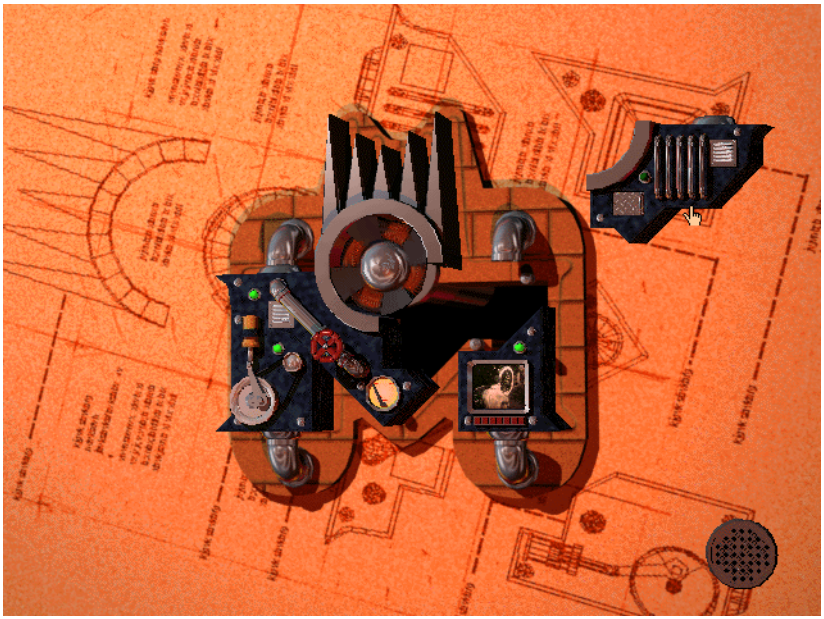


Figure 7.1 Solving the mTutorial puzzle

places, the "M" machine springs to life as a series of animations on different parts of the "M" begin to play.

When finished with the puzzle, click on the "manhole" icon in the lower right corner of the screen to jump to a credits scene. When the credits have finished, the project ends and mTropolis returns to edit mode.

START A NEW PROJECT

When you are finished exploring the completed puzzle, close the finished tutorial project by selecting Close from the mTropolis File menu. If you are prompted to save any changes, click the Don't Save button. The tutorial's layout window will disappear.

Now we'll recreate the puzzle project. Start a new project by selecting **New-Project** from the File menu. A new, empty project appears. This project contains an empty section, subsection, and scene. The empty scene is displayed in the layout view.

CREATE THE FIRST SCENE

In the first part of this tutorial, we'll begin by adding media to the scene.

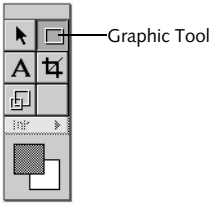
Adding the Media

Let's add the background image for the logo movie that plays when the project is first run.

- Click on the Scene to select it.
- Choose **Link Media-File** from the File menu. A standard file selection dialog appears.
- Choose the image named **Napkin.pict** from the **PICTs** folder, found inside the **mTutorial Media** folder. The **Napkin.pict** image fills the scene.

Now let's put the logo QuickTime movie on top of this background image. First we'll create an element to contain the QuickTime movie.

- Select the graphic element tool (the box-shaped tool) in the tool palette.



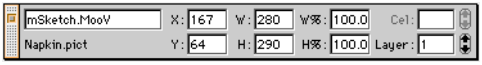
- Drag anywhere on the scene to create an empty graphic element of any size.
- Select the new element and choose **Link Media-File** from the File menu. A standard file selection dialog appears.
- Choose the file named **mSketch.MooV** in the **MOOVs** folder, found inside the **mTutorial Media** folder.

If the Application Preferences (accessed through the Edit menu) are set to their defaults, the element will resize automatically to the size of the QuickTime movie. However, if it doesn't resize automatically, select the element and select **Revert Size** from the Object menu.

Position the Movie

Let's position the QuickTime element (mSketch.MooV) precisely using the Object Info palette.

- If the Object Info palette (shown below) is not already visible, select **Object Info Palette** from the View menu. The Object Info palette appears. This palette shows sizing and position information for the currently selected object.
- Select the mSketch.MooV element, and enter 167 in the "X" field and 64 in the "Y" field. Use the tab key to jump between fields and the enter key to confirm the final data entry.



Using a Custom Color Palette

The project we are creating was designed to run on 256-color displays. The graphics for this project were rendered using a custom color palette. The color palette has been saved as a CLUT file that we can import into our project.

- Ensure that Modifier Palette Group 2 is visible. To do this, select the View menu and look at the **Modifier Palettes** cascading menu item. If there is a checkmark next to **Group 2**, that palette is already visible. If

there is not a checkmark, select the **Group 2** menu item to display the palette.

- Drag a color table modifier from this palette and drop it onto the scene (i.e., the background Napkin.pict element, not the QuickTime video element).

Hot Tip

Hold down the control key while moving the cursor over the modifier palette to view each modifier's name.

- Double-click the color table modifier on the scene to open its dialog box.

The highlighted text at the top of the modifier dialog is the default name of this modifier.

- Rename the modifier "mTutorial.clut." It's a good authoring habit to give your modifiers unique and descriptive names.
- From the dialog's Color Table pop-up menu, choose the **Link file** option. A standard file selection dialog appears. Choose the file **mTutorial.clut** from the **CLUTs** folder found within the **mTutorial Media** folder. Your Color Table Modifier dialog should now look like the one shown Figure 7.2. Click the "OK" button to dismiss the Color Table Modifier dialog.

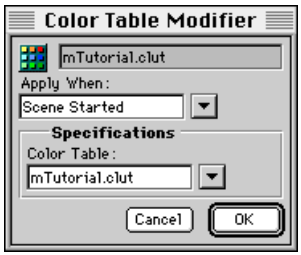


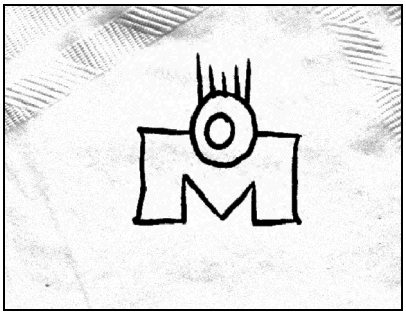
Figure 7.2 A typical modifier configuration dialog.

Hot Tip

To view the effect of a color table that has been applied to a scene while in edit mode, choose the name of the color table from the View menu's Preview Color Table submenu.

Test the Project

So far, we've worked on the project in *edit mode*. We can switch to *runtime mode* to view the project as a user would see it. Press ⌘-T to run the project. The screen should go black, then the napkin picture appears and the QuickTime movie plays over the top of it. To switch back to edit mode, press ⌘-T again.



Changing an Element's Properties

For the most part, the QuickTime movie element behaves just as we want it to—it plays through one time, then stops. The element can be customized in numerous ways through its Element Info dialog. Use the movie's Element Info dialog to adjust the movie's volume.

- Double-click the mSketch.MooV element to open its Element Info dialog, or display the dialog by selecting the mSketch.MooV element and then choosing **Element Info** from the Object menu.
- In the Element Info dialog, change the value of the Volume setting to 80. Now when the project is played, the volume of the movie will be 80% of its maximum volume.
- Click "OK" to accept this change and dismiss the Element Info dialog.

Run the project again by pressing ⌘-T. Press ⌘-T again to return to edit mode.

Saving the Project

Now would be a good time to save your work.

- Choose the **Save** option from the File menu (or press ⌘-S).
- Name and store the project as you would any other application file.
- If, at any point, you want to restore the project to a previously-saved version, select **Open** from the File menu to load the saved file.

Using a Modifier to Change the Scene

When the movie is finished playing, we want our project to continue to the next scene. The Change Scene modifier can be used to add this type of functionality to our project. We'll also configure our introductory scene so that if the user clicks before the movie is done, the scene will change.

- Drag a Change Scene modifier from the Group 2 modifier palette and drop it on the movie element. The modifier icon attaches itself to the upper left corner of the movie element.

- Double-click the modifier icon to display its configuration dialog.
- Change the name of this modifier. Change the text of the modifier's name field (which currently reads "Change Scene Modifier") to **To Next Scene**. When naming modifiers in your own projects, use concise, descriptive names that relate to the function of the modifier.

Now use the Execute When pop-up menu to specify the message that will activate this modifier.

- Open the Execute When pop-up menu and select **Play Control-At Last Cel** as shown

in Figure 7.3. Now when the movie ends, a scene change will occur.

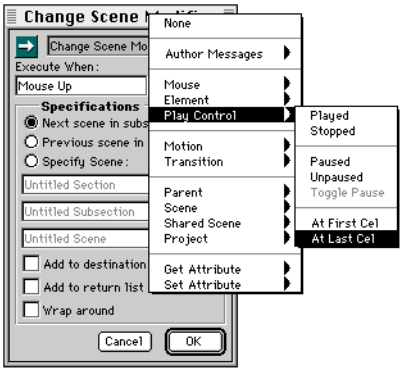


Figure 7.3 Configuring the "Execute When" field of a Change Scene Modifier

The Specifications area of the Change Scene Modifier dialog is used to choose the destination scene to which to change. Since we want to change to the next scene, and this is the default setting, we won't change it.

- Accept the changes to the modifier by clicking the OK button. The Change Scene Modifier dialog disappears.

Now let's create a Change Scene modifier that changes to the next scene if the user clicks on the screen while the introduction is still playing.

- Select the change scene modifier on the mSketch.MooV element and press ⌘-D to duplicate it. Drag the new copy from the movie element and drop it on the scene (i.e., drop the modifier *outside* the bounds of the mSketch.Moov element so that it attaches to the scene element).
- Double-click the new change scene icon to display its configuration dialog.
- Use the Execute When pop-up menu to select **Mouse-Mouse Up**. Now this modifier will activate when the user clicks on the scene.
- Click OK to accept the change. The dialog disappears.

Now we will create the next scene in the project.

Create a New Scene

To create a new scene in the layout view, use the third pop-up on the right at the top of the window (where it now reads "Napkin.pict"). The pop-up lists all scenes in the current subsection and a "New Scene" option that can be used to create new scenes. New Scene always appears as the last item in this pop-up.

- Select New Scene from the scene pop-up. A new, empty scene (named "Untitled Scene") is created. The layout window changes to display this new scene.

Let's link a background image to this new scene.

- Click on the scene to select it.
- Choose the **Link Media-File** option from the File menu. A standard file selection dialog appears.
- Select the **mBackground.pict** file from the **PICTs** folder within the **mTutorial Media**

folder. The mBackground.pict file fills the scene. Note also that the name of our new scene changes to "mBackground.pict".

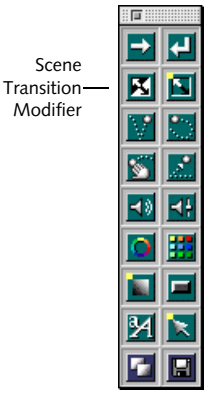
Let's see the effect of adding this scene.

- Return to the previous scene by using the scene pop-up to select "Napkin.pict" or click once on the left scene navigation arrow at the top of the layout window.
- Now run the project (press ⌘-T) and view the changes. Notice that when the movie ends, or when you click on the first scene, the scene changes to the next scene. Press ⌘-T again to return to edit mode.

Adding a Scene Transition

When the scene changes, it simply jumps from the first scene to the next, without any sort of transition effect. Let's create one.

- Drag a Scene Transition modifier from the Group 2 modifier palette and drop it onto the Napkin.pict scene.



- Double-click the modifier to display its configuration dialog.
- Change its name from Scene Transition Modifier to Random Dissolve.
- Select Scene Ended from the Enable When pop-up.
- Select Random Dissolve from the Transition pop-up.
- Set the value of the Rate option to 30.
- Click the OK button to close the Scene Transition Modifier dialog. The dialog disappears.

Press ⌘-T to run the project. Notice that when the scene changes, the first scene appears to "dissolve" into the next. Press ⌘-T again to return to edit mode.

CREATING THE SECOND SCENE

Now let's add elements to the second scene.

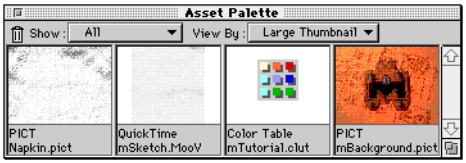
- Navigate to the second scene by selecting mBackground.pict from the scene pop-up (the third pop-up from the right at the top of the layout window) or click the right navigation arrow (also found at the top of the layout window).

In this scene, we will build a reasonably complex "puzzle" using animated elements as the puzzle pieces. The puzzle pieces will be draggable by the user and programmed to "snap" into place if they are within a 15 pixel radius of their destination coordinates. Once they are in place, the pieces are no longer draggable. Once all of the puzzle pieces are in place, they become animated.

Adding a Puzzle Piece to the Scene

In this section, we'll use the Asset Palette to manipulate media that has been linked to the project.

- Select Asset Palette from the View menu. The Asset Palette appears.



This palette shows thumbnail images of all of the media assets currently linked to the project.

Previously, we had linked media directly to graphic elements in the project. Now, however, we will import media without having an element selected. The media will be linked to the project, but won't appear in the layout window—it will only be added to the Asset Palette.

- Link all the media files contained in a directory into the project. Choose the **Link Media-Folder** option from the File menu. A standard file selection dialog appears. Select the **mTOONs** folder found within the **mTutorial Media** folder. Click on the button at the bottom of the folder selection dialog when the name "**mToons**" appears in that button.

Also link one more file into the project.

- Select the graphic tool from the tool palette.
- Create a new graphic element by dragging the tool somewhere on the scene.
- Choose the **Link Media-File** option from the File menu. A standard file selection dialog appears. Select the file **mPiece 1.pict** found in the **PICTs** folder in the **mTutorial Media** folder. The picture appear in the graphic element. Notice that the contents of the Asset Palette are also updated after linking the media. You may need to resize the Asset Palette or use its scroll bar to see the new media icon.
- We only want this new element in the Asset Palette right now—it is not actually needed in the scene. Delete the graphic element from the scene by selecting it (if it is not already selected) and choosing **Cut** from the Edit menu.

The media assets on the Asset Palette can be dragged from the palette and dropped into our project. Let's add one of the puzzle pieces to the project.

- Drag the mToon element mPiece 2.toon from the Asset Palette and drop it onto the mBackground.pict scene.

Programming the First Puzzle Piece

Since all of our puzzle pieces need to behave in a similar fashion, we will program one element first and then create an *alias* of that programming that we can copy onto all the other puzzle pieces.

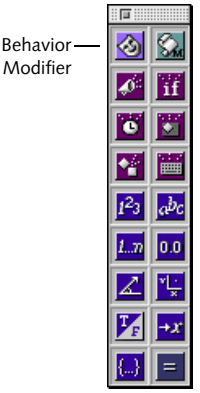
Aliasing requires additional explanation. When programming the puzzle, we are going to create a single behavior that can then be aliased and used on *all* of the puzzle pieces. The alias feature allows you to create copies of programming that can be used on multiple elements in a project. Updates made to one

copy are made to all of the aliased versions simultaneously.

Creating the Puzzle Piece Behavior

Let's begin programming the first puzzle piece by adding modifiers to it. The first type of modifier we'll add is a behavior.

- Ensure that Modifier Palette Group 1 is visible. Select the View menu and look at the **Modifier Palettes** cascading menu item. If there is a checkmark next to **Group 1**, that palette is already visible. If there is not a checkmark, select the **Group 1** menu item to display the palette.
- Drag a behavior modifier from the Group 1 modifier palette and drop it onto the puzzle piece mPiece 2.toon in the layout window. The behavior modifier is a special modifier that can contain other modifiers.

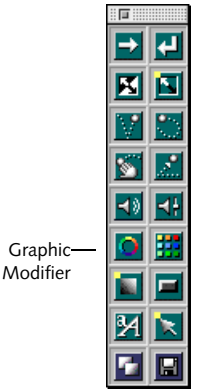


- Double-click the behavior modifier to display its configuration dialog.
- Change the name of the behavior from **Behavior** to **Puzzle Piece**. Do not close the behavior dialog.

Making the Puzzle Piece Transparent

Now let's add the programming that will make the piece transparent to the background.

- Drag a graphic modifier from the Group 2 modifier palette and drop it into the open behavior window.
- Double-click the graphic modifier to open its configuration dialog.



- Change the graphic modifier's name from **Graphic Modifier** to **Transparent Ink**. Its purpose is to apply a background transparent effect to the element with which it is associated.
- Use the Ink pop-up menu to select the Background Transparent option.
- Close the graphic modifier dialog to accept the change.

The behavior dialog should now look like the one shown in Figure 7.4.

Making the Puzzle Piece Draggable

By adding another modifier to the behavior, we can make the puzzle piece draggable by the user.

- Drag a drag motion modifier from the Group 2 modifier palette and drop it into the Puzzle Piece behavior window.

Since we're using just one drag motion modifier, we'll use its default name. No special configuration of this modifier is necessary at this point.

Use ⌘-T to switch to runtime mode and try dragging the puzzle piece around. Note that the animation will be running, but we'll pause it in the next few steps. When finished, press ⌘-T again to return to edit mode.

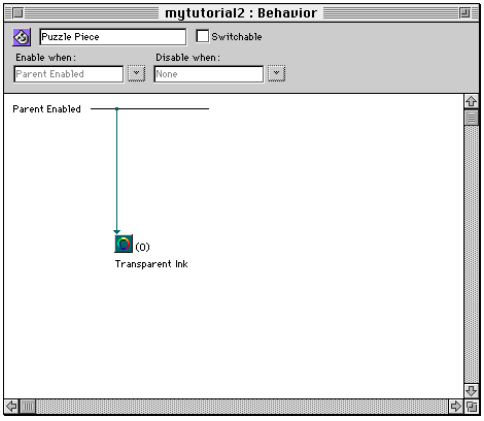
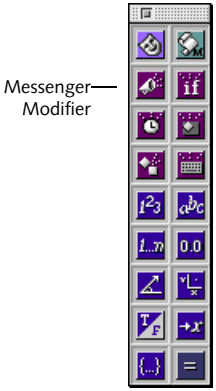


Figure 7.4 The Puzzle Piece behavior window with one modifier

Changing Other Aspects of the Puzzle Piece Behavior

Since these elements are reasonably small animation files, and we want good playback performance, we will preload them into RAM. This can be accomplished by sending the puzzle piece a *Preload* command.

- Re-open the Puzzle Piece behavior by double-clicking its icon.
- Drag a messenger modifier from the Group 1 modifier palette and drop it into the Puzzle Piece behavior window.



- Double-click the messenger modifier icon to display its configuration dialog.
- Change the name of the messenger from **Messenger** to **Preload**.
- Use the messenger's Execute When pop-up menu to select the **Scene-Scene Started** option.
- In the messenger's Message Specifications section, use the Message/Command pop-up to select the **Element–*Preload Media*** option.
- No change needs to be made to the With and Destination menus.
- Click OK to accept the changes. The Messenger dialog disappears.

By default, the puzzle piece animations play when the scene starts. However, we want the animations to be paused until the puzzle is complete. The animation can be paused by sending a *Pause* command to the puzzle piece.

- Drag another messenger modifier from the Group 1 modifier palette and drop it into the Puzzle Piece behavior window.
- Double-click the messenger modifier icon to display its configuration dialog.
- Change its name to Pause.
- Use the messenger's Execute When pop-up menu to select the **Parent-Parent Enabled** option.
- Use the messenger's Message/Command pop-up to select the **Play Control-*Pause*** option.
- Leave the Destination of the message as Element (this is the default destination). The With pop-up should also not be changed.
- Click OK to accept the changes and dismiss the Messenger dialog.
- Click the behavior's close box to dismiss the Puzzle Piece behavior window.

Note that the animation could also have been paused by setting the element's "paused" attribute via the Element Info dialog. However, by using a messenger to pause the animation and placing that messenger in a behavior that will be used on all the other puzzle pieces, we have eliminated the need to individually select the "paused" attribute for each puzzle piece.

Adding the Puzzle Snap-In Function

Let's now program the "snap-in" functionality of the puzzle piece. First, we need a way to store the screen coordinates of the correct position of the puzzle piece. We can use a point variable to store this value. Since every puzzle piece will use the same (aliased) behavior, but will all have different final screen positions, the variable that contains each piece's x and y coordinates must be placed *outside* the behavior.

Positioning variables this way allows variables of the same name to contain different values. The modifiers that access them from within aliased behaviors will use the correct variable for each element.

Let's add a point variable modifier to the mPiece 2.toon puzzle piece.

- Drag a point variable modifier from the Group 1 modifier palette and drop it on the mPiece 2.toon puzzle element.
- Double-click the point variable icon to display its configuration dialog.
- Change the variable's name from **Point Variable** to **piecePosition**.

- Enter 167 into the modifier's X field and 204 into the Y field.
- Click OK to accept the changes and dismiss the Point Variable dialog.

Adding a Miniscript Modifier to the Behavior

The mTropolis Miniscript modifier is a special modifier that can execute commands written in a simple scripting language. This modifier allows you to create modifiers that perform complex or customized tasks.

Here, we'll add a simple script that sets the puzzle piece's position to a random position within the boundary of the screen.

- Double-click the Puzzle Piece behavior icon to open its window.
- Drag a Miniscript modifier from the Group 1 modifier palette and drop it into the Puzzle Piece behavior window.

- Double-click the Miniscript modifier icon to display its configuration dialog.
- Change the modifier's name from **Miniscript Modifier** to **Random Position**.
- Use the Execute When pop-up to select the message **Scene-Scene Started**.
- In the modifier's Script text box, type the following script:

```
-- Set the puzzle piece to a
-- random position:
--
set position to rnd(580), rnd(420)
```

Your modifier dialog should now look like the one shown in Figure 7.5.

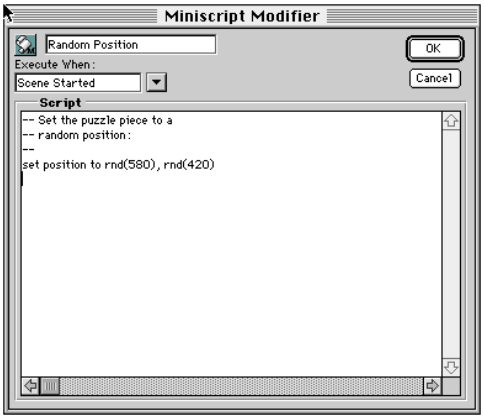


Figure 7.5 The "Random Position" Miniscript modifier dialog

The first three lines of our script are comments—the two dashes that start each line tell mTropolis to ignore any text that follows on that line. Comments are not required for the script to function properly, but help to make your scripts easier to read and debug.

The last line is a Miniscript statement. It uses the Miniscript function "rnd" to generate a random number between 0 and the number in parentheses for the x and y coordinates of the element. When activated,

this script will set the puzzle piece's position to the random values generated by the "rnd" function.

- Click OK to accept these changes and dismiss the Miniscript Modifier dialog.

Encapsulating the Modifiers into a Behavior

As good housekeeping, we'll encapsulate some of the modifiers we have created into a new behavior. Let's group the modifiers that set the initial characteristics of the puzzle piece together.

- Drag a new behavior from the Group 1 modifier palette and drop it into the open Puzzle Piece behavior window.
- This time, instead of opening the behavior to rename it, simply click on the behavior's name shown below its icon, and enter the new name, **Initial Settings**. Click outside the name when you are done editing the name.
- Instead of opening the new behavior's dialog, modifiers can be added to the behavior simply by dragging and dropping their icons onto the new behavior's icon. Drag the following modifiers onto the Initial Settings behavior icon: the graphic modifier named Transparent Ink, the messenger named Preload, the Miniscript named Random Position, and the messenger named "Pause." The modifier icons seem to "disappear" as they are moved into the new behavior.

Configuring an 'If' Messenger to Test for the Puzzle Piece Position

Now let's program an "if" messenger to test for the position of the puzzle piece. This modifier will compare the position of the puzzle piece when the user releases the piece to the value stored in the piecePosition point variable that we previously attached to the element.

- Drag an "if" messenger from the Group 1 modifier palette and drop it into the Puzzle Piece behavior window.

- Double-click the messenger's icon to display its configuration dialog.
- Change the name of the messenger from **If Messenger** to **Dropped in Valid Position**.
- By default, this messenger is configured to act on the Mouse Up message, and this is what we want, so we don't need to change the Execute When pop-up.
- In the "If" text field, enter the following statement:

```
((position.x < (piecePosition.x + 15)) and \
(position.x > (piecePosition.x - 15))) and \
((position.y < (piecePosition.y + 15)) and \
(position.y > (piecePosition.y - 15)))
```

This statement evaluates to true when the puzzle piece is released within 15 pixels of

the value stored in the piecePosition point variable.

- When these conditions are met, we want to send a message to the element that it has been released in the proper location. To do this, we will create a custom message (an author message). Highlight the content of the Message/Command field (do not use the pop-up button). Type **Piece In Place** into the field and click OK. A dialog appears, asking if you want to create this new author message. Click OK on this new dialog.

Programming the Puzzle Piece when it is in Place

Now we are ready to program the actions of the puzzle piece when it is dropped in place. A behavior will be used to store the actions that occur.

- Drag a new behavior modifier from the Group 1 modifier palette and drop it into the Puzzle Piece behavior window.
- Click on the name of the behavior that appears below the icon and enter the new name, **Piece in Place**.

Next we'll configure the previously-created drag motion modifier so that the puzzle piece cannot be dragged once it is in place.

- Drag the drag motion modifier icon from its current position and drop it into the Piece in Place behavior.
- Double-click the Piece in Place behavior icon to open its window.
- In the Piece in Place window, double-click the drag motion icon to display its configuration dialog.
- We want to disable this modifier when the piece is put in its correct place. Use the Disable When pop-up menu to select the option **Author Messages-Piece in Place**. Now the piece will no longer be draggable after this message is received.
- Click OK to confirm your changes and dismiss the Drag Motion Modifier dialog.
- Move the "Dropped in Valid Position" messenger you created previously into the "Piece in Place" behavior.

We now need to add a Miniscript modifier to the Piece in Place behavior that moves the piece to its final position when it is dropped near, but not exactly on, the final position.

- Drag a new Miniscript modifier from the Group 1 palette and drop it into the Piece in Place behavior.
- Double-click the modifier to display its configuration dialog.
- Name the new Miniscript modifier **Piece in Place**.
- Use the Execute When pop-up menu to configure the modifier to activate when the Piece in Place author message is received. Select the **Author Messages-Piece in Place** option from the menu.
- In the Script text field, enter the following script:

```
-- Snap piece into place:
set position to piecePosition

-- Change cel of toon:
set cel to 2
```

- Click "OK" to close the Miniscript dialog. Click on the close boxes of the open behaviors to dismiss their windows.

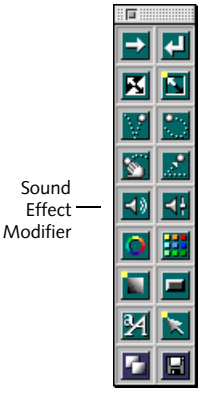
The first line of the script moves the element to its exact final position in the puzzle. The second line changes the currently displayed cel of the element so that it looks different when it is in place in the puzzle.

You may want to test your programming up to this point. Press ⌘-T to switch to runtime mode. When the puzzle appears, drag the puzzle piece and drop it near its correct position (this piece belongs on the left-side slanted line of the "M"). You should notice it snap into place when you release it. Once it does, you will no longer be able to drag the piece around. Press ⌘-T to return to edit mode.

Adding the Snap Sound

A sound effect would be nice feedback to notify the user that the puzzle piece is in place.

- Drag a sound effect modifier from the Group 2 modifier palette and drop it into the Piece In Place behavior window.



- Double-click the sound modifier to display its configuration dialog.
- Name the modifier **Piece in Place Sound**.
- Use the Execute When pop-up to select the message that triggers the sound. Select **Author Messages-Piece in Place**.
- Use the dialog's Sound pop-up menu to select a sound to be played. Select **Link File…** from the menu. A standard file selection dialog appears. Select the sound file **Piece in Place.aiff** located in the **AIFFs** folder found in the **mTutorial Media** folder.
- Click the Preview button to preview the sound. Click OK to accept your changes and dismiss the Sound Effect Modifier dialog.

Disabling the Piece in Place Behavior

One of the most powerful capabilities of behaviors is that they can be made "switchable". That is, they can be turned "on" or "off" by messages, just like any other modifier. When a behavior is deactivated, all of the modifiers inside that behavior are also deactivated.

In our project, it makes sense to deactivate the "Piece in Place" behavior once a piece is actually

in place. By switching this behavior off, we can insure that the project doesn't keep checking for puzzle pieces that are already in their proper places.

Let's add one last modifier to the Piece in Place behavior.

- Drag a new messenger modifier from the Group 1 modifier palette and drop it into the Piece in Place behavior window.
- Double-click the new messenger to display its configuration dialog.
- Rename the messenger to **Disable Checks**.
- Use the dialog's Execute When pop-up to select the **Author Messages-Piece in Place** message.
- Highlight the text in the Message/Command menu and type **Disable Checks**.
- Click OK. A dialog appears, asking if you want to create the new author message. Click OK.
- In the "Piece in Place" behavior window, select the Switchable checkbox found next to the behavior name.
- Two previously inactive pop-up menus become accessible. Now the behavior has Enable When and Disable When pop-up menus that can be used to specify the messages that activate and deactivate this behavior (and all of the modifiers contained within it).
- Verify that the behavior's Enable When message is Parent Enabled, then use the Disable When pop-up menu to select **Author Messages-Disable Checks**. Now the functionality of this behavior will be disabled when the piece is in place.

Your Piece in Place behavior window should now look something like the one shown in Figure 7.6. We are finished modifying the Piece in Place behavior, so close the window by clicking its close button.

Keeping Track of the Puzzle Pieces in Place

Let's program a behavior that keeps track of the number of puzzle pieces in place, so that when the sixth piece is dropped in place, the entire puzzle will come alive.

- Start by dragging a new behavior modifier from the Group 1 modifier palette and dropping it into the "Puzzle Piece" behavior window.
- Double-click the new behavior's icon to display its configuration window.
- Change the name of the behavior to **Puzzle Status**.
- Drag an integer variable from the Group 1 modifier palette and drop it in the Puzzle Status behavior window.
- Double-click the integer variable's icon to display its configuration dialog. Change the name of the variable to **pieceCount**. Leave its value field set to 0 (the default). This variable will be used to store the number of puzzle pieces that are in place.
- Click OK to dismiss the variable's dialog.

- Drag another integer variable into the Puzzle Status behavior and name it **totalPieces**.
- Set the variable's value to 6.

We're going to use copies of these variables in each of the puzzle piece elements, so now we'll make these variables into "aliases".

- Select both variable icons in the Puzzle Status behavior window (using Shift-click or by simply dragging the pointer across the variables to "marquee" select them).
- Choose **Make Alias** from the Object menu. Making these variables aliases means their values will be the same wherever they occur in your project.
- Click outside the variables to deselect them. You will notice a visual change to the icons.

Whenever the puzzle scene is first displayed, we want to make sure that the pieceCount variable is initialized to zero.

- Drag a Miniscript modifier from the Group 1 modifier palette and drop it into the Puzzle Status behavior window.
- Double-click the Miniscript modifier to display its configuration dialog.
- Change the modifier's name to **Reset pieceCount**.
- Use the modifier's Execute When pop-up to select the **Scene-Scene Started** message.
- In the Script text field, enter the following script:

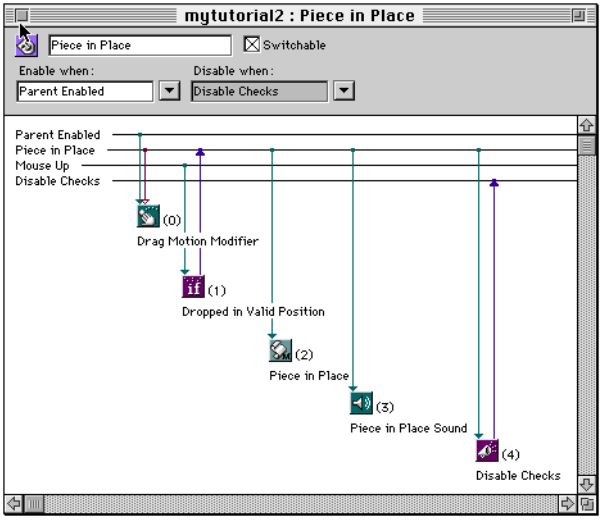


Figure 7.6 The completed Piece in Place behavior

```
-- Reset the count of pieces
-- in place:
set pieceCount to 0
```

- Click OK to confirm the changes and dismiss the dialog. Now each time the user arrives at the puzzle scene, the pieceCount variable is set to 0, meaning that no pieces are currently in place.

Notifying the Environment that the Puzzle Is Complete

Now we need to add functionality that updates the pieceCount counter each time a puzzle piece is put in place. When the counter reaches the total number of pieces, a message will be sent to the environment that the user has finished the puzzle.

Before we add another modifier, let's create a new author message. This time, use the Author Messages window to create the author message.

- Select **Author Messages Window** from the View menu. The Author Messages window appears.
- Click the New Message button in the Author Messages window. An untitled message appears below the two previously-defined messages shown in the window.
- Double-click the untitled author message and change its name to **Puzzle Complete**.
- Close the Author Messages window by clicking its close button.

Now we'll return our attention to the "Puzzle Status" behavior.

- Drag a new Miniscript modifier to the Puzzle Status behavior window.
- Double-click the modifier's icon to display its configuration dialog. Change its name to **Add To Counter** and configure the Execute When field to execute on **Author Messages-Piece in Place**.
- Enter the following script in the Script text field:

```
-- Increase the pieceCount:
set pieceCount to pieceCount + 1

-- Is the puzzle complete?
if pieceCount = totalPieces then
send "Puzzle Complete" to \
     element's parent
end if
```

  The **`set`** statement in the script increases the count of puzzle pieces by one. The rest of the script (the **`if`** statement) is a simple conditional statement. When the number of pieces in place equals the total pieces in the puzzle, the "Puzzle Complete" author message is sent to the element's parent, which is the scene.

- Click OK to dismiss the Miniscript Modifier dialog.
- Close the "Puzzle Status" behavior window by clicking its close box.

Animating the M Machine

Now we'll program the actions that are to take place when all the puzzle pieces have been put in place.

- Add a new behavior modifier to the "Puzzle Piece" behavior.
- Double-click the behavior icon to display its configuration window.
- Change the name of the behavior to **Puzzle Complete**.
- Select the Switchable box at the top of the behavior window. The Enable When and Disable When pop-ups become available.
- Use the Enable When pop-up menu to select **Author Messages-Puzzle Complete**. Use the Disable When pop-up menu to select **Parent-Parent Enabled**.
- Add a Miniscript modifier to the "Puzzle Complete" behavior window.
- Double-click the Miniscript modifier to display its configuration dialog.
- Change the Miniscript modifier's name to **Set Animation Specs**.
- Use the Execute When pop-up menu to select the **Parent-Parent Enabled** message.
- Enter the following script in the Script text field:

```
-- Set the range of cels to play:
set range to 2 thru 9

-- Set the rate of play:
set rate to 15
```

  Now when the animation plays, it will only play cels 2 through 9 at a rate of 15 frames per second.

- Click the OK button to dismiss the Miniscript Modifier dialog.
- Drag a messenger modifier from the Group 1 modifier palette and drop it into the "Puzzle Complete" behavior window. This messenger will be configured to activate the animation.
- Double-click the messenger icon to display its configuration dialog. Change the messenger's name to **Play Animation** and configure it to execute on the **Parent-Parent Enabled** message using the Execute When pop-up.
- Use the Message/Command pop-up menu to select the **Play Control-*Play*** command. When sent to the element, this command will make its animation begin playing.
- Click OK to close the messenger dialog and confirm the changes.
- Close the "Puzzle Complete" behavior window by clicking its close button.
- Close the "Puzzle Piece" behavior window by clicking its close button.

If you haven't saved your project in a while, now is a good time to select **Save** or **Save As** from the File menu.

Adding and Programming the Other Puzzle Pieces

We have just created a reusable software component that we are going to apply to all of the puzzle pieces. Let's add them now.

- If it is not already visible, select **Asset Palette** from the View menu. The Asset Palette appears. Also, select **Alias Palette** from the View menu. The Alias Palette appears. Note that the two aliases we created previously are shown on the Alias Palette.
- Drag the rest of the puzzle pieces (**mPiece 1.pict, mPiece 3.mToon, mPiece 4.mToon, mPiece 5.mToon,** and **mPiece 6.mToon**) from the Asset Palette into the layout window.
- Select (click on) the "Puzzle Piece" behavior icon found on the mPiece 2.toon element.
- Choose **Make Alias** from the Object menu. Notice that the "Puzzle Piece" behavior is added to the Alias Palette.
- Now distribute this modifier to the five other puzzle pieces by dragging the aliased Puzzle Piece behavior icon from the Alias Palette and dropping it onto each piece. As you drop the alias on each piece, notice how each piece's background becomes transparent. Each piece is inheriting the properties defined by the Puzzle Piece behavior.
- Copy the point variable (named piecePosition) from the mPiece 2.mToon element to each of the other puzzle pieces. Don't confuse this variable with the two on the Alias Palette. Option-drag the point variable from mPiece 2.mToon to each piece. Using Option-drag makes a copy of the variable instead of moving the original.

Now we have to configure the point variables on each of the newly-added puzzle pieces with the correct positions.

- For each new puzzle piece, double-click the point variable on that piece to display its configuration dialog. Change the X and Y values to the appropriate value shown in Table 7.1. The value of mPiece 2.toon has already been set, but is shown below for completeness.

Element Name	X	Y
mPiece 1.pict	242	83
mPiece 2.toon	167	204
mPiece 3.toon	294	199
mPiece 4.toon	167	241
mPiece 5.toon	356	237
mPiece 6.toon	257	166

Table 7.1: piecePosition values for each puzzle piece

Changing the Layer Order of Pieces

One final consideration for the integration of these puzzle pieces is their *layer order*. Layer order refers to the order in which elements in a scene are drawn on the screen. Elements will draw on top of one another according to their layer order. Higher layer order numbers draw "in front" of lower numbers.

We can use the Object Info Palette to set each piece's correct layer order.

- If it is not already displayed, open the Object Info Palette by selecting **Object Info Palette** from the View menu.

- Select each puzzle piece, and enter its correct layer order number, as shown in Table 7.2, in the "Layer" field of the Object Info Palette. When the pieces have been assigned the layer orders shown in Table 7.2, the completed "M" machine will look its best.

Asset Name	Layer
mPiece 4.toon	1
mPiece 1.pict	2
mPiece 2.toon	3
mPiece 6.toon	4
mPiece 3.toon	5
mPiece 5.toon	6

Table 7.2: Layer order numbers for each piece

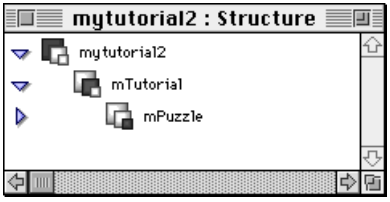
Now press ⌘-T to run the project. Each piece should snap into place and make the "clang" sound when dropped into its proper position. When all the pieces have snapped in, the pieces of the "M" should become animated. Press ⌘-T to return to edit mode.

NAMING STRUCTURAL ELEMENTS

As with any mTropolis element, it is good practice to give descriptive names to all of the sections and subsections of a project. These names will make the project much easier to understand, especially for others. We're going to use the mTropolis structure view to rename the section and subsection used in this project.

- Select Structure Window from the View menu. The Structure window appears.
- In the structure window, click the text label that reads **Untitled Section**. When the field becomes editable, change the name to **mTutorial**.
- Each level of the hierarchy shown in the structure view has an "Open/Close" triangle to its left. If the triangle is pointing to the right, there are move levels in the hierarchy that can be revealed by clicking the triangle. When clicked, the triangle points downward and the next level of the hierarchy is revealed. We want to reveal the level below the mTutorial section, so click the triangle next to it. The Untitled Subsection level is revealed.
- Click the text label that reads **Untitled Subsection**. When the field becomes editable, change the name to **mPuzzle**.

Your Structure window should now look like the one shown below.



ADDING SOUND

One final touch will make our puzzle more satisfying: a sound that plays when the puzzle is complete. Previously, we used a *sound modifier* to play the "Piece in Place" sound. However, sound media can be mTropolis objects, just like graphical media. In this section, we'll add a *sound element* to the puzzle. Sound ele-

ments can only be added to a project in the structure view, as they have no visual representation in the layout view.

- In the Structure window, click on the open/close triangle next to the mPuzzle subsection to reveal our project's scenes.
- Now click on the open/close triangle next to the scene named "mBackground.pict". Icons for the elements of that scene (our puzzle pieces) appear in the list.
- To add a new sound object, select (click on) the mBackground.pict element and choose **New Sound** from the Object menu. A sound element icon appears below the mToon icons.
- Make sure that the sound icon is selected and choose **Link Media-File** from the File menu. A standard file selection dialog appears. Select the file **Puzzle Complete Loop.aiff** found in the **AIFFs** folder within the **mTutorial Media** folder and click the "Link" button. The name of the sound element icon changes to reflect the name of the media. The Structure window should now look like the one shown in Figure 7.7.
- Double-click the sound icon to display its Element Info dialog. In the dialog's Initial State section, select the Paused and Loop options.
- Click OK to confirm the changes and close the dialog.

Now we'll program the sound to start playing when the puzzle is completed.

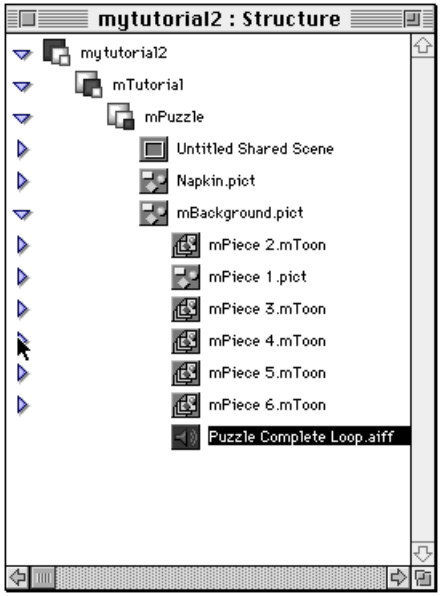


Figure 7.7 Adding a sound element to the structure window

- Drag a messenger onto the sound icon. The icon will seem to "disappear" into the sound icon. Click the sound icon's open/close triangle to reveal the next level of the structure hierarchy—the sound element's modifiers. Now the Messenger icon is visible.
- Double-click the Messenger icon to display its configuration dialog.
- Change the messenger's name to **Play when Puzzle Complete**.
- Use the Execute When pop-up to select **Author Messages-Puzzle Complete**.
- Use the Message/Command pop-up to select the command **Play Control-*Play***. Now

when activated, this modifier will cause the sound element to begin playing.

- Click OK to confirm the changes and dismiss the Messenger dialog.

Press ⌘-T to run the project and hear the difference when the puzzle is completed. Press ⌘-T to return to edit mode.

THE CREDITS SCENE

After all this work, it's time for some self recognition. Making up your own credits screen is a good start. We are going to create a simple credit roll in a new subsection of the project.

- Create a new subsection by choosing the New Subsection option from the Subsection pop-up on the *Layout window*. This menu is the second menu from the left at the top of the layout window (its label currently reads "mPuzzle"). A new "Untitled Subsection" is created. The layout window updates to show the subsection's "Untitled Scene".
- In the *Structure window,* note that a new Untitled Subsection icon appeared at the bottom of the window. Click the name of the new subsection and change it to **Credits**.
- Click on the subsection's open/close triangle to reveal its scenes.
- Highlight the name of the Untitled Scene and change it to **My Credits**.

Now let's return our attention to the Layout window and create some text for our credits.

- Click on the scene in the Layout window.
- Select the text tool (the one that looks like the letter "A") from the tool palette. The cursor changes to an I-beam with a small square next to it.
- Create a text field by dragging in the layout window. Make the text element fairly large, so you can enter a large amount of self-congratulatory text. The outline of the new text element appears in the window.
- Notice that when you move the cursor over the text element, the cursor changes to a simple I-beam. Click inside the text element to put an insertion point in the element. A flashing cursor appears.
- Type the text that you want to appear in the credits. For example:

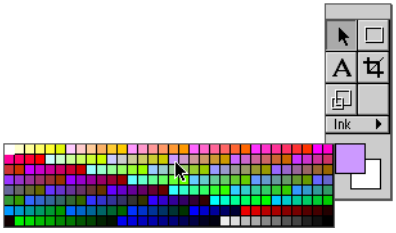
  This Tutorial Created by:
  Happy M. User
  mFactory
  1440 Chapin Ave. #200
  Burlingame, CA 94010

  When you are finished entering your text, select all of the text by dragging over it with the I-beam cursor. Now you can select various text options from the Format menu. Pick a font, size, style, and alignment that appeals to you.
- Return to the tool palette and choose the selection tool (the arrow). The cursor changes back to an arrow.

- Double-click on the new text element to display its Element Info dialog. Change the element's name to **Credits Text**.
- Click OK to confirm your change and dismiss the Element Info dialog.

We should now change the color of our text so that it will show up against the default black background.

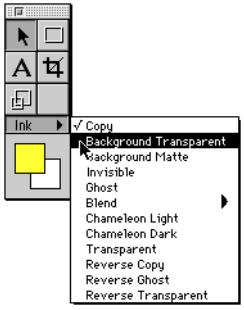
- Make sure that the text element is selected.
- Select a new color for the text by clicking and holding the cursor over the foreground color swatch in the tool palette. A palette appears and the cursor changes to an "eyedropper". Drag the eyedropper to a color in the palette and release the mouse button to select a color as shown below.



Notice that a graphic modifier icon appears on the text element. Selecting colors from the tool menu is equivalent to configuring a graphic modifier.

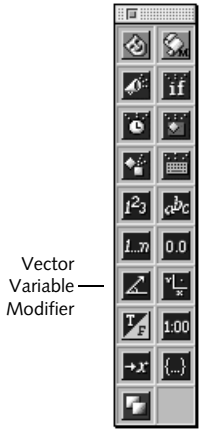
We should also make the background of the text element transparent.

- Make sure that the text element is selected and click on **Ink** in the tool palette. A pop-up menu appears. Select **Background Transparent** as the ink option.



Now let's modify the text element so that it scrolls up and off the screen. To do this we will use a vector motion modifier. This modifier works in conjunction with the vector variable.

- Drag a vector variable from the Group 1 modifier palette and drop it onto the text element.
- Double-click the vector variable icon to display its configuration dialog.
- Change the variable's name to **Up**. Type 90 in the Angle field and 0.8 in the Magnitude field.



- Click OK to confirm the changes and close the dialog.
- Drag a vector motion modifier from the Group 2 modifier palette and drop it onto the text element.

- Double-click the vector motion modifier icon to display its configuration dialog.
- Change the modifier's name to **Move Up**.
- Use the Execute When pop-up to select the Scene-Scene Started message.
- Use the Vector pop-up to select the name of the vector variable associated with this vector motion. Select the **Credits Text-Up** option.
- Click "OK" to dismiss the Vector Motion dialog.

Now press ⌘-Y to run the project from this scene (⌘-T runs the project from the very first scene). The text element should rise from its starting position to the top of the screen. Press ⌘-Y again to return to edit mode.

The effect is interesting, but you might want some action to take place once the text leaves the screen. Let's use a boundary detection messenger.

- Drag a boundary detection messenger from the Group 1 modifier palette and drop it onto the text element.
- Double-click the boundary detection modifier icon to display its configuration dialog.
- Change the name of the modifier to **Detect Leaving Top**.
- In the Detect Boundaries of Element's Parent section, check only the "Top" checkbox. The "Bottom", "Left" and "Right" options should be unchecked.
- In the Detect Element section, choose the "Once Exited" and "On first detection" radio buttons.
- In the Message Specifications section, use the Message/Command pop-up to select the **Project-*Close Project*** command. Use the Destination pop-up to select **Project** as the destination for the command.
- Click OK to confirm the changes and dismiss the Boundary Detection Messenger dialog.

In the layout window, you might want to position the text element slightly below the bottom of the frame of the scene. When the text scrolls up it will look like a credit roll like you might see at the end of a movie.

Another nice thing to do is to give the user the ability to abort the play of the credits.

- Drag a messenger modifier from the Group 1 modifier palette and drop it on the My Credits scene.
- Double-click the messenger icon to display its configuration dialog.
- Change the name of the messenger to **Close Title**.
- Leave the Execute When message set to its default (**Mouse Up**). Use the Message/Command pop-up to select **Project–Close Project**. Use the Destination pop-up to select **Project**. Now, if a user clicks on the title scene before the credits have finished rolling, the project just ends.

- Click OK to dismiss the dialog.

Using the Shared Scene

You might have noticed that even though we have created a fully-functional credits scene, there's no way for it to be activated from earlier scenes in the project!

We'll create a button on the shared scene that activates the title roll. Putting the button on the shared scene will make it available to all scenes within a subsection.

- Use the controls at the top of the layout window to navigate to the shared scene of the "mPuzzle" subsection. To do this, select **mPuzzle** from the subsection pop-up menu (the second pop-up from the left that currently reads "Credits"). The layout window changes to show the mPuzzle subsection's first scene, which just happens to be the Untitled Shared Scene that we're interested in.
- Select the graphic element tool from the tool palette and create a new graphic element by dragging on the shared scene.
- Select the new element, then choose **Link Media-File** from the File menu. A standard file selection dialog appears. Select the file **Manhole Quit.mToon** from the **mTOONs** folder found in the **mTutorial Media** folder. The first cel of the Manhole mToon is shown in the graphic element.
- Reposition the element by dragging it to the lower right area of the shared scene.
- Double-click the element to open its Element Info dialog.
- In the Initial State section of the dialog, ensure that the Paused checkbox is checked.
- Click OK to confirm the change and close the dialog.

Let's make the manhole's background transparent.

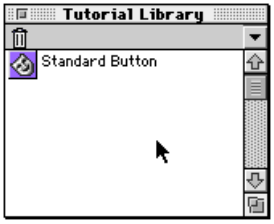
- Drag a graphic modifier from the Group 2 modifier palette and drop it on the Manhole Quit.mToon element.
- Double-click the graphic modifier's icon to display its configuration dialog.
- Change the modifier's name to **Transparent Ink**. Use the Ink Effect pop-up to select **Background Transparent**.
- Click on the dialog's close box to accept the changes and dismiss the dialog.

Using Libraries

Let's apply some functionality to the Manhole Quit.mToon button by adding a behavior from a *library*. Libraries can be used to store project components (e.g., sections, subsections, scenes, elements, behaviors, and modifiers) in a file that is separate from the project.

- Select Open from the File menu and choose the item named "Tutorial Library".

The Tutorial Library appears as its own palette as shown below.



- Drag the behavior named Standard Button from the Tutorial Library palette and drop it on the Manhole Quit.mToon element.

This behavior emulates a button that changes its appearance when it is clicked on. The behavior sends out three author messages: "Highlight," "Un-Highlight," and "Execute." To use this behavior, add it to a graphic that you want to act as a button, configure its two states (highlight/un-highlight) and configure what it does (execute).

In this case, we have an mToon with two cels. One cel shows the highlighted button state, the other shows the un-highlighted state. To program the button to show the correct cel at the correct time, we'll use two Miniscript modifiers to change the cel display of the mToon.

- Drag a Miniscript modifier from the Group 1 modifier palette and drop it on the Manhole Quit.mToon element.
- Double-click the Miniscript icon to display its configuration dialog.
- Change the name of the modifier to **Un-Highlight**.
- Use the Execute When pop-up to select **Author Messages-Un-Highlight**. This author message was automatically created when you added the Standard Button behavior to your project.
- Enter the following script in the Script text field:

```
set cel to 1 -- Show closed manhole
```

- Click OK to dismiss the dialog.
- Now place another Miniscript modifier on the manhole element.
- Double-click the new Miniscript icon to display its configuration dialog.
- Change the name of the modifier to **Highlight**.
- Use the Execute When pop-up to select **Author Messages-Highlight**. This author message was automatically created when you added the Standard Button behavior to your project.
- Enter the following script in the Script text field:

```
set cel to 2 -- Show open manhole
```

- Click OK to dismiss the dialog.

Since messengers in the Standard Button behavior are already programmed to respond to the user's mouse actions by sending the appropriate message, now all we need to do is configure the button's action.

- Drag a change scene modifier from the Group 2 modifier palette and drop it on the Manhole Quit.mToon element.
- Double-click the change scene modifier to display its configuration dialog.
- Change the name of the modifier to **To Credits**.
- Use the Execute When pop-up to select the **Author Messages-Execute** message.
- In the Specifications section of the dialog, click the Specify Scene radio button. Three pop-up menus become active. These menus can be used to select the section, subsection and scene that will be changed to. Select the mTutorial section, Credits subsection, and My Credits scene.
- Click OK to accept the changes and dismiss the Scene Change Modifier dialog.

Your Quit button is now ready to operate. Use ⌘-T to run your project from the beginning. Note that the "manhole" button is always available. Clicking it sends you to the credits page.

Don't forget to save your finished project one last time before quitting mTropolis.

That's it! Congratulations on your completion of the mTropolis tutorial project!

Chapter 8. mTropolis Examples

This chapter describes some of the mTropolis examples included in the "mExamples Project". This mTropolis project is installed in the "mExamples" subfolder of the mTropolis folder by default when mTropolis is installed.

The samples included in this project demonstrate general mTropolis features, such as the use of behaviors to encapsulate programming code, the methods used to structure a project, as well as more specific topics, such as messaging and the use of variables.

Like all mTropolis projects, each sample project contains reusable components. For example, the button logic that is encapsulated into behaviors in the Button Gallery project can be saved in libraries and used when creating buttons in your own projects. Think of the components in each project as "clip programming" that you can use royalty-free. Use the projects as a source of ideas for planning and implementing basic title features.

INSTALLATION

If the mExamples project is not installed on your system, it can be installed as follows:

- Insert the mTropolis CD into your CD-ROM drive.
- Double-click on **Install mTropolis** to launch the installer.
- Use the installation options to perform a custom installation that installs just the "mExamples Project and Media" option.
- Alternatively, simply drag the "mExamples" folder from the CD-ROM to your hard drive.

RUNNING THE mEXAMPLES PROJECT

To run the mExamples Project, double-click the "mExamples Project" icon, found in the "mExamples" folder of your mTropolis installation. mTropolis will start and open the mExamples Project in *edit mode*.

When mTropolis is first started, the program is in edit mode. Edit mode is used to author projects. Runtime mode is used to preview a project. In runtime mode, the project is displayed just as a user would see the completed title.

- *Note: Note that there may also be a "mExamples Title" icon in the mExamples folder. This program is the mExamples Project built into a standalone title. This title can be run, but cannot be edited.*

Running the Project from its First Scene

Select **Run-From Start** from the File menu (or press ⌘-T) to switch to runtime and pre-

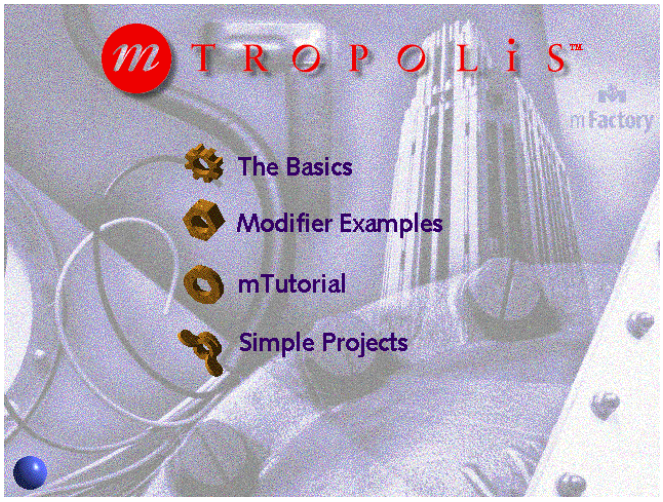


Figure 8.1 The mExample Project's main menu

view the project from its first scene (the first scene in the first subsection in the first section of the project). The mTropolis interface disappears and the project starts. Alternatively, press ⌘-Option-T to view switch to runtime mode but keep the mTropolis interface visible.

Returning to Edit Mode

Press ⌘-T again to return to the scene where ⌘-T was originally pressed. Alternatively, press ⌘-. (Command-period) to return to edit mode at the current runtime scene.

Running the Project from a Selected Scene

Select **Run-From Selection** from the File menu (or use ⌘-Y) to switch to runtime mode and preview the project from the scene currently displayed in edit mode.

To Return to Edit Mode

Use ⌘-Y again to return to the scene where ⌘-Y was originally pressed. Optionally, use ⌘-. (Command-period) to stop at the current runtime scene.

THE mEXAMPLES PROJECT INTERFACE

When run from its first scene, the project plays a short movie (which can be bypassed by clicking). When the movie finishes, the mExample Project's main menu appears (Figure 8.1). There are a number of options available in this menu:

- **The Basics**: Click this option to display a menu of simple projects that illustrate some basic mTropolis programming concepts. This section is described in more detail in "The Basics" on page 8.3.

- **Modifier Examples**: Select this option to display a menu of simple projects that illustrate the use of each mTropolis modifier.
- **mTutorial**: Select this option to run display a project similar to that described in Chapter 7, "Tutorial".
- **Simple Projects**: Select this option to display a menu of projects that illustrate more advanced mTropolis programming concepts. These projects are described in more detail in "Simple Projects" on page 8.35.
- **Quit Button**: Click the blue sphere in the lower left corner of the scene to quit the examples and return to edit mode. Note that, in some scenes, a red sphere is also present. Click the red sphere to return to the previous menu.

THE BASICS

The following options are available in "The Basics" submenu:

- **Button Gallery**: The implementation of user interaction through radio buttons, animated buttons, and buttons with selected/deselected bevels is featured in this project.
- **Calculated Fields**: This project demonstrates text and graphic "running total" displays.
- **Changing Cursors**: To show the library of cursors available in mTropolis' cursors modifiers and how to use them are the goals of this project.
- **Communicating**: This project demonstrates a simple, but robust use of messaging.
- **Controlling Audio**: This project shows how modifiers can be used to control the playback of sounds.
- **Controlling mToons**: The control mToons, mTropolis' proprietary animation format, are featured in this project.
- **Linear Navigation**: This project shows a simple navigational system using arrow buttons.
- **Revealing Objects**: This project shows how to show and hide elements in response to messages.
- **Scene Transitions**: This project shows the use of mTropolis' library of transition effects.
- **Spatial Navigation**: Building a system for navigating a 3D world is featured in this project.

COMMON BASICS PROJECTS FEATURES

If you are a relative newcomer to mTropolis programming, the following section provides a basic description of authoring methods used to create a project's interface screen and navigational buttons. These features are common to all of the section in "The Basics". In particular, the shared scene, the title bar, and the logic of the scene navigational arrows are described.

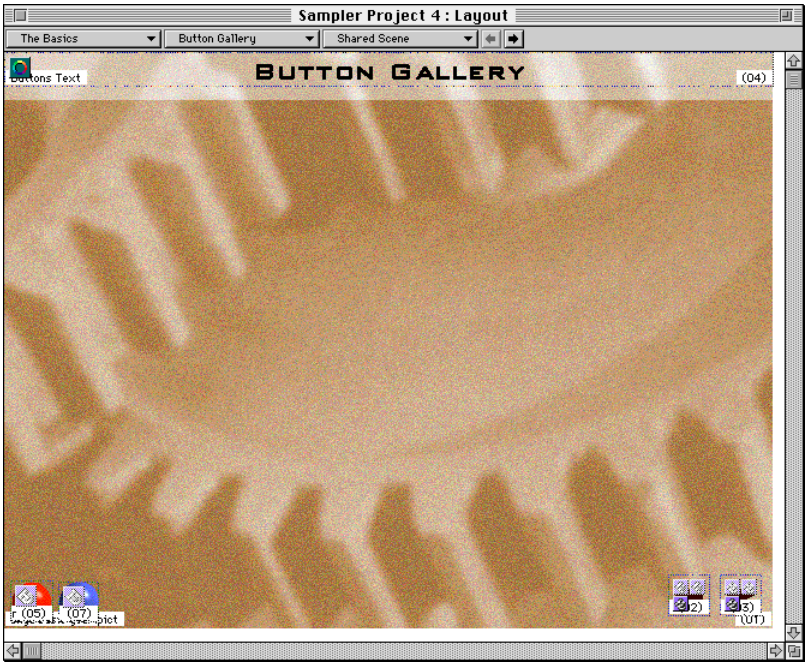


Figure 8.2 The shared scene for the Button Gallery

The Shared Scene

Shared scenes are special elements that come first in a subsection. During runtime, the shared scene is visible behind all other scenes in a subsection, making it a logical place for elements and modifiers that are common to them all. For example, the navigational buttons that are to appear in all scenes in a section can be placed in the shared scene.

In the Basics Projects, the background image, the title bar and the navigational buttons are placed in the shared scene. Figure 8.2 shows the shared scene for the Button Gallery project.

A background image (bkgd basic gear.pict) is linked to the shared scene. This image shows through the transparent parts of the buttons, which are graphic elements that have been laid on top of it.

The Title Bar

In projects that have the same title bar over multiple scenes, the text element that contains the title text is placed in the shared scene. For example, the title bar text element in Figure 8.2 has been placed on the background image, the Button Gallery shared scene. By applying a graphic modifier with a transparent ink effect to the text element, the text's

field is made transparent allowing the background image to show through.

The area of the background image beneath the title bar text element in Figure 8.2 has been lightened to emphasize the title bar's text. This effect was created in a graphics application before the PICT was linked to the project.

Unlike the background image, the text element called "Buttons Text" was not linked to an external file, but created within the mTropolis application with the text tool from the tool palette.

The Navigational Arrow Buttons

The navigational arrow buttons allow the user to move forward or backward in the project, from one scene to another. Since they are to appear in all project scenes, the buttons have also been placed on the shared scene.

Each navigational arrow button consists of a two-celled mToon animation that has been linked to a graphic element. The arrow in the first cel of the animation is plain, and the arrow in the second cel of the animation is dimmed. Miniscript modifiers in the Bevel Button behavior on the mToon animation control which cel of the animation is displayed during runtime, depending on the user's mouse actions.

Like the title bar text element, a graphic modifier configured to apply a Background Transparent effect has been placed on the mToon, allowing the background image to show through. This graphic modifier is inside the Bevel Button behavior.

The figure below shows the three behavior modifiers that have been placed on the right arrow button.

Figure 8.3 Arrow mToon with behaviors

The function of the navigation buttons is determined by the order and configuration of modifiers contained in the three behaviors that are placed on them. The contents of each of these behaviors are described in the following sections.

Standard Button Behavior

The Standard Button behavior contains a collection of messengers that make the button work as a proper button should (i.e., clicking down on the button highlights it, dragging off the button removes the highlight, and releasing the button will initiate its associated activity). This behavior is used on all buttons throughout the projects.

Figure 8.4 shows the modifiers contained in the Standard Button behavior (double-click on a behavior icon to display that behavior's contents)

Three author messages are generated at different times by the six messengers in this behavior. These author messages are "Highlight," "Un-

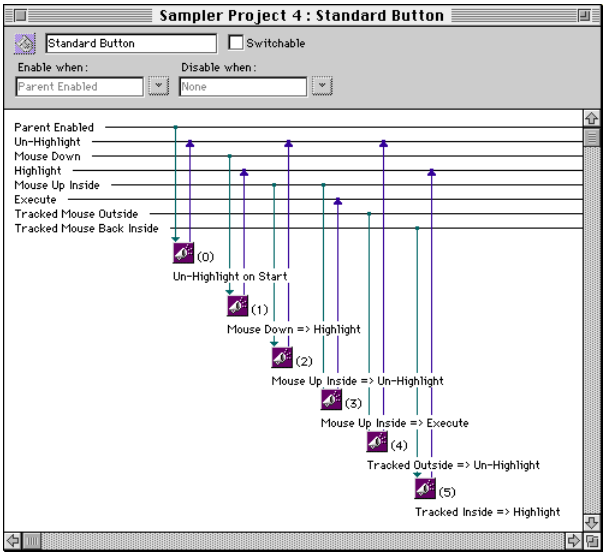


Figure 8.4 Contents of the Standard Button behavior

Highlight" and "Execute." Each of these messages is targeted to the modifier's element, the mToon animation. They will automatically be broadcast to the modifiers in the other behaviors on the mToon (see "Bevel Button Behavior" on page 8.7 and ""The Navigational Arrow Buttons" on page 8.5 for details.)

Un-Highlight on Start
When runtime is first entered, Parent Enabled is automatically sent to components throughout the project. The first messenger in the Standard Button behavior is configured to send an "Un-Highlight" author message to the mToon button.

Mouse Down Highlight
When the user presses the mouse over the button, the message Highlight is sent to the element.

Mouse Up Inside Un-Highlight
When the user releases the mouse over the button, the message Mouse Up is sent to the element.

Mouse Up Inside Execute
When the user releases the mouse over the button, an author message called "Execute" is also sent.

Tracked Outside Un-Highlight
When the user drags the cursor off of the button while the mouse is depressed, an "Un-Highlight" author message is sent to the element.

Tracked Inside Highlight
When the user drags the cursor back onto the button while the mouse is depressed, a

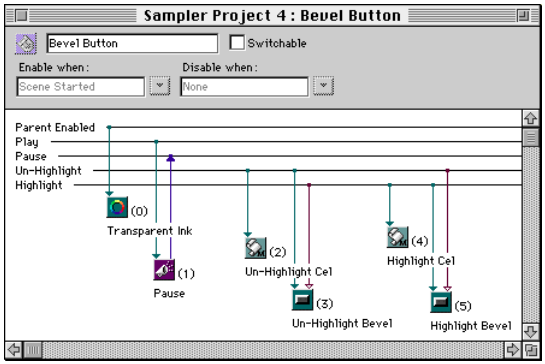


Figure 8.5 Contents of the Bevel Button behavior

"Highlight" author message is sent to the element.

Bevel Button Behavior

The second behavior on the navigational buttons is called "Bevel Button." The modifiers in this behavior respond to the messages sent from the messengers in the Standard Button behavior by displaying the correct cel of the button animation, and by applying graphic beveled edge effects to the mToon navigation buttons. Figure 8.5 shows the contents of Bevel Button behavior.

This behavior is also used throughout the projects to provide a consistent look to the navigation buttons.

Transparent Ink

The first modifier applies a background transparent ink effect to the animation, which allows the background image to show through.

Pause

To prevent the mToon button from playing, this messenger is configured to send a *Pause* command on receipt of a Played message. This insures that the correct cel is displayed on the scene when it is entered.

Un-Highlight Cel

On the "Un-Highlight" author message, the Miniscript modifier called "Un-Highlight Cel" sets the cel of the animation to 1.

Un-Highlight Bevel

On the "Un-Highlight" author message, an image effect modifier applies a Section 1 bevel effect.

Highlight Cel

On the "Highlight" author message, the Miniscript modifier called "Highlight Cel" sets the cel of the animation to 1.

Highlight Bevel

On the "Highlight" author message, an image effect modifier applies a highlight bevel effect.

NAVIGATIONAL BUTTON BEHAVIOR: LEFT AND RIGHT

The third behavior contains the two messengers used to control the visibility of the right and left arrow buttons, and a change scene modifier. Figure 8.6 shows the contents of the "Left Arrow" behavior.

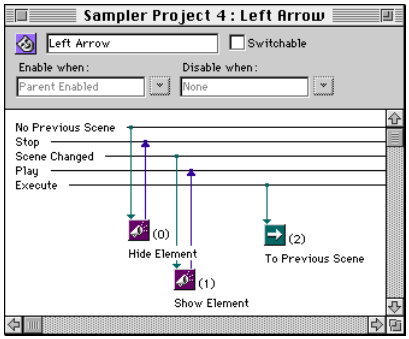


Figure 8.6 Contents of the Left Arrow behavior

Hide Element

When this messenger receives a No Previous Scene message, this messenger sends a *Stop* command to its element to hide the arrow button.

Show Element

When this messenger receives a Scene Changed message, it sends a *Play* command to its element to display the arrow button.

To Previous Scene

A change scene modifier is configured to change to the next or previous scene on receipt of the "Execute" author message.

BUTTON GALLERY

The "Button Gallery" example, available from "The Basics" menu, shows how to create various styles of buttons using elements linked to PICTs or mToons. Button effects are created using graphic modifiers, or a combination of messengers and graphic modifiers.

The "Button Gallery" project uses various mTropolis' modifiers in combination with its messaging system to create of a variety of button states, such as selected/deselected.

This project uses a shared scene, and two other scenes called "Button 1," and "Button 2." In this project, ten button states are shown on the two scenes (excluding the navigational buttons that toggle between them). Each type of button is described by the text label to its right.

Button Scene 1

Highlight on Mouse Over

Effect: The button highlights (flashes) when the mouse passes over the button element.

- The image effect modifier named "Button Initial State" applies the deselected 3D bevel effect to the button element on receipt of a Parent Enabled message. This message is sent by mTropolis when the scene is entered.
- Another image effect modifier named "Button Highlight" applies a Tone Up effect to the Element when it receives a Mouse Over message, that is, when the user's cursor is inside the button's element boundaries. The effect is removed (turned off) when a

Mouse Outside message is received, that is, when the cursor is outside the element's boundaries.

Highlight on Mouse Down
Effect: The button highlights when the button is selected.

- This button acts the same as the previous example, except that the Tone Up effect is applied on receipt of a Mouse Down message and removed on receipt of a Mouse Up message.

Select on Mouse Down
Effect: The button's beveled edges invert when the button is selected.

- The image effect modifier named "Button Initial State" applies the deselected bevel effect to the button element when the scene starts.
- The image effect modifier called "Button Selected" inverts the bevels when the button receives a Mouse Down message.

Select on Enter Key
Effect: Pressing the Enter key simply inverts the beveled edges of the button.

- The keyboard messenger sends an Enter Key Pressed message to the button element when the Enter key is pressed.
- The image effect modifier named "Button Selected" applies a selected bevel to the button when the message is received.

Select/Deselect on Mouse Up/Down
Effect: Clicking on this button causes the beveled edges of the button to flash.

- The image effect modifier named "Button Initial State" applies the deselected bevel effect to the button element on receipt of a Parent Enabled message. This effect is removed on receipt of a Mouse Down message.
- The image effect modifier named "Button Selected" applies a selected bevel effect to the button on receipt of a Mouse Down message. This effect is removed on receipt of a Mouse Up message.
- The image effect modifier named "Button Deselected" resets the button to its initial state on a Mouse Up message.

Button Scene 2

Show PICT on Mouse Down
Effect: A lighter PICT of a gear appears over the gear graphic when the user clicks on the button. The darker gear PICT reappears when the mouse is released.

- The image effect modifier named "Button Initial State" applies the deselected bevel effect to the button element when the scene starts.
- The messenger named "Show Gear Light.pict" sends a *Play* command to the Gear Light.pict element when it receives a Mouse Down message.
- The messenger named "Hide Gear Light.pict" sends a *Stop* command to the Gear Light.pict element when it receives a Mouse Up message.

Radio Buttons (1 and 2)
Effect: Clicking on one radio button inverts the beveled edge (selected), and causes the other radio button to toggle to the opposite state (deselected).

Radio buttons are paired (1 and 2) in this project. The method used here can be applied to any number of buttons acting as radio buttons.

- The image effect modifier named "Button Initial State" applies the deselected bevel effect to the button element when the scene starts. The effect is disabled on a Mouse Down message.
- The image effect modifier named "Button Selected" applies the selected bevel effect on a Mouse Down message. This effect is removed on the "Enable Other Button" author message that is sent when the other radio button is selected.
- The messenger named "Deselect Radio Button" sends the "Enable Other Button" author message to the other radio button on a Mouse Down message.
- When it receives an "Enable Other Button" author message, the image effect modifier named "Button Deselected" applies a deselected bevel effect to its element. This effect is removed on receipt of a Mouse Down message.

Toggle Select Behavior Button
Effect: The button continuously alternates between inverted and plain beveled edges, creating a blinking button effect. This example also demonstrates how a behavior may be used as a container for other modifiers, making it easy to copy and transfer programming to other elements.

- The messenger modifier named "Button Initial State" applies the deselected bevel effect to the button element when the scene starts by sending a Deselect message to the element.
- The timer messenger named "Continuous Select/Deselect Toggle" causes the repeated flashing of the button. When the scene starts, it repeatedly sends a "Toggle Select" author message in 0.5 second intervals. The Toggle Select message is interpreted by the modifiers listening for the author messages "Select" or "Deselect." If the element is currently "Select," it toggles to "Deselect" and vice versa.
- The image effect modifier named "Button Selected" applies the selected bevel effect to the button element on receiving a Mouse Down message. This effect is disabled on receipt of a Mouse Up message.
- The image effect modifier named "Button Deselected" applies the selected bevels effect to the button element on receiving a Mouse Up message. The effect is disabled on receipt of a Mouse Down message.

Animated Button
Effect: When clicked, the gear image on the button continuously rotates. In this example, the media linked to the button element is a mTropolis mToon animation.

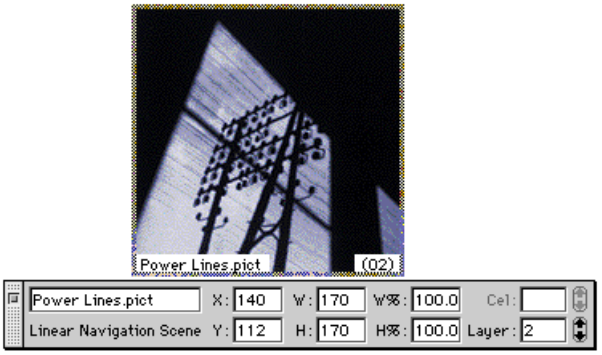


Figure 8.7 Power Lines.pict and the Object Info palette

- The image effect modifier named "Button Initial State" applies the deselected bevel effect to the button element when the scene starts.
- The messenger named "Play Gear.Toon" sends a *Play* command to Gear.Toon on a Mouse Down message.
- In the Gear.toon element, the messenger named "Preload Element" sends a *Preload Media* command to its mToon element on receipt of the Parent Enabled message. The *Preload Media* command loads the mToon animation into memory before the scene starts. Animations played from memory, rather than from disk, play more quickly and more smoothly.

LINEAR NAVIGATION

The "Linear Navigation" example, available from "The Basics" menu, is a simple slide show that illustrates how to change from one scene to the next. For information regarding navigation through a complex 3D world, see "Spatial Navigation" on page 8.12.

Description

This project uses the navigational arrows described in "Common Basics Projects Features" on page 8.3. The right and left arrows navigate through a series of pictures. When there is no "Previous Scene," the left arrow is hidden. When there is no "Next Scene," the right arrow is hidden.

Adding the Images

In this project three scenes were created, one for each of the PICT elements.

The pictures linked to the elements were created outside the mTropolis environment. They are the same size. After linking the pictures, the object info palette (Figure 8.7) was used to position each PICT element in the same location in each scene.

All three pictures were placed at screen coordinates 140,112.

The navigation buttons allow the user to flip through the PICTs linked to each element in each scene.

SPATIAL NAVIGATION

The "Spatial Navigation" example, available from "The Basics" menu, shows how to navigate through a 3D world. Aliased cursor modifiers, transition modifiers and scene change modifiers are also discussed. For information regarding simple scene-to-scene navigation, see the Linear Navigation project that immediately precedes this project.

Description

The user clicks on a door in the opening scene, enters a 3D room, and explores it, eventually leaving by the door again. The room has twelve different points-of-view, one in each lateral direction (four walls), four up (ceiling) and four down (floor). Navigating up to the ceiling or down to the floor produces a different view, depending on from which wall the user navigates.

Movement is selectively allowed in right and left directions, and up and down directions, depending on where the user is located. For example, the user can look up to the ceiling from any wall, but can only look down to the same wall again. To see a different view of the ceiling, the user must navigate to a different wall. Although navigation is simplified, all possible types of movement in a 3D world are illustrated.

- *Note: Use the structure window (select **Structure Window** from the View menu) to review the hierarchical structure of components in this project.*

Navigating the Scenes

Each of the points-of-view in the 3D room is a separate scene that has a PICT, such as Ceiling/Lamp.pict, linked to the background. A second, "invisible" graphic element called "Down" is placed on the scene. This element provides a large hot spot that is configured with modifiers to respond to the user's mouse.

Figure 8.8 shows a PICT of the ceiling that is linked to the example scene. The transparent element called "Down" contains a cursor, scene transition, and scene change modifier.

- *Note: The way that scenes in this project are navigated is common to all points of view, so only one will be discussed.*

The cursor modifier named "Down When Mouse Over" changes the default arrow pointer to the Hand Down cursor when the pointer passes over the area covered by the element called "Down."

The transition modifier named "Push Up" applies a Push transition on a Mouse Up message. This transition will take effect when the user clicks on the scene.

The change scene modifier named "Go To Wall" is the last modifier on the Down element. On receipt of a Mouse Up message the scene will change to Wall/Lamp, the scene specified in this modifier (Figure 8.9).

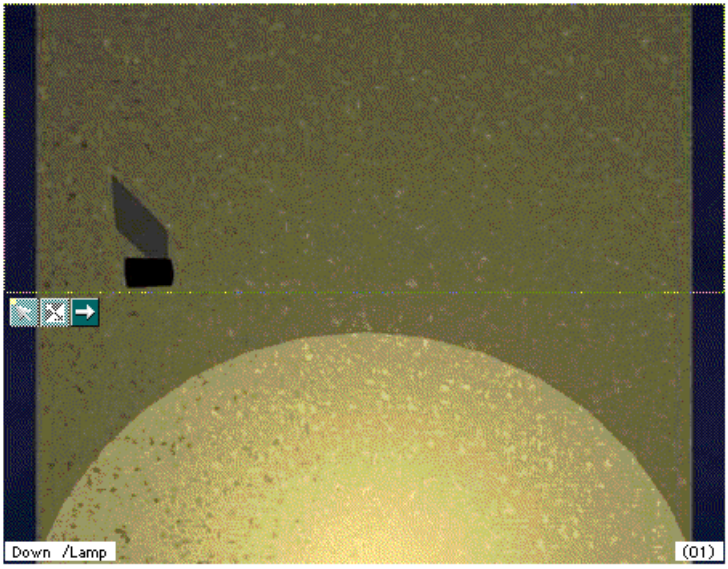


Figure 8.8 Layout of the example scene, Ceiling/Lamp

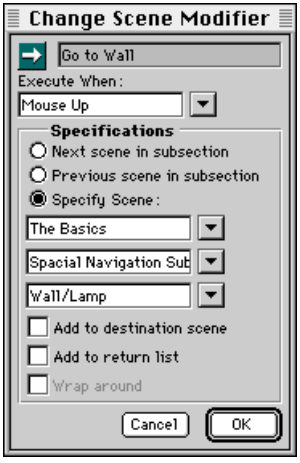


Figure 8.9 The "Go to Wall" Scene Transition

Aliasing Cursor and Transition Modifiers

The Spatial Navigation project illustrates the use of aliases to maximize authoring productivity. An alias is a special copy of a modifier; it takes its functionality from the modifier from which it was made. In this project, directional cursors and scene transitions have been aliased and used throughout the scenes. Aliasing the modifiers not only prevents having to program each individually, but it also saves time if the settings of the modifiers need to be changed. For example, all aliased cursor modifiers called "Up When Mouse Over" use the Hand Up cursor. If the author wanted to change this cursor to a pointer, simply changing the settings in one alias would update all other aliases of the same modifier (Figure 8.10).

Figure 8.10 Alias Palette showing just a few of the aliases used in the mExamples project

The transition modifiers have also been aliased, meaning that in order to change the rate of all aliased Push Down transition modifiers, only the rate settings of one alias would have to be changed.

See "Alias Palette" on page 11.7 of the *mTropolis Reference Guide* for details regarding aliases.

SCENE TRANSITIONS

The "Scene Transitions" example, available from "The Basics" menu, shows how mTropolis' scene change modifier is used in combination with the scene transition modifier to create transitions between scenes.

Description

This project contains two scenes. "Transitions 1" is the main interface and "Transitions 2" is a background image used to demonstrate the visual effect of each transition. Clicking on one of the gear buttons causes one of a variety of transitions to a full screen image.

Each button in this project is configured the same way, except that the settings in the transition modifiers vary (i.e., type, direction, step and rate settings). The Slide Button transition is documented below.

Slide Button

The element called "Slide Button" uses the Standard Button behavior as well as a behavior called "Bevel Button & Next Scene." (A change scene modifier configured to go to the next scene on the "Execute" author message has been included in the Bevel Button & Next Scene behavior.)

Each button is configured to function the same way, allowing the button behaviors to be aliased and used on each. Since the scene transition modifiers are used to apply unique transition effects, one has been placed on the button element beside each aliased behavior.

The change scene modifier named "Slide Transition" on the first button applies the scene transition on an "Execute" author message. The Slide Down effect is used, set at 64 steps and at the maximum rate.

Double-click on the transition modifier to open its dialog. Experiment with different step and rate settings for each of the transitions.

CONTROLLING mTOONS

The "Controlling mToons" example, available from "The Basics" menu, shows how to use modifiers to control mToons, animations created in mTropolis' proprietary format.

mTropolis' mToon animation format creates animations composed of individual cels. Unlike time-based formats like QuickTime which play frames over time, individual cels or a range of cels can be named and specified for playback. The speed of the animation can also be precisely controlled.

Description

In this project, the user toggles between two scenes. In the first scene, three buttons allow the user to play different ranges of cels in the scene's orbiting planet animation. In the second scene, three buttons allow the user to play the scene's orbiting planet animation at different speeds. In either scene, the animation can be dragged about the scene within the constraints of an invisible boundary while it continues to play.

Hot Tip

Use the structure window to view the modifiers and elements in this project.

Providing Frame-by-Frame Control of mToons

The animation included with this project contains three logical cel ranges. These ranges were created in the mToon editor. The first scene of the project called "mToon Ranges" illustrates precise control of the playback of these ranges.

Messenger modifiers on the buttons are used in combination with integer range variables on the scene to control the playback of a specified range of cels. Depending on which button is clicked, a specific range of cels is played.

Range Variables

The three integer range variables that define the range of cels played by each button are placed on the scene. By placing the variables on the scene, their values are accessible by all modifiers on the elements in the scene (see "Variable Scopes" on page 13.25 of the *mTropolis Reference Guide* for details).

Gear Buttons

The range of cels the animation will play is determined when the user clicks on one of the gear buttons.

Each of the buttons, labeled Slow Button, Medium Button, and Fast Button, sends an author message named "Set Orbit Speed 1," "Set Orbit Speed 2," or "Set Orbit Speed 3" with an integer variable called "Orbit Speed 1," "Orbit Speed 2," or "Orbit Speed 3" to the scene when it is clicked. The variables to which each of these messengers refer have been placed on the scene.

A Miniscript modifier can also be used to set the range the animation is to play. See Chapter 14 of the *mTropolis Reference Guide,* "Miniscript Modifier", for details.

Radio Buttons

The Radio Button behavior is used in place of the Standard Button behavior. According to the programming of this behavior, only one button in the group may be active at any one

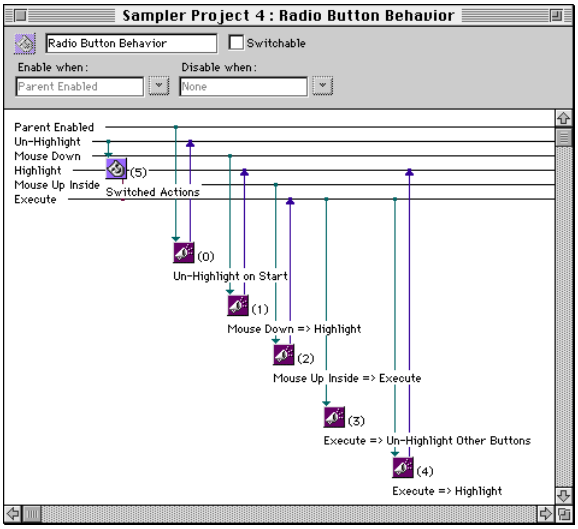


Figure 8.11 Contents of the Radio Button behavior

time, and this button remains highlighted once clicked on. Figure 8.11 shows the contents of the Radio Button behavior.

The contents of the Radio Button behavior is similar to that of the Standard Button behavior, except this behavior contains two additional messengers and another behavior. The first additional messenger, named "Execute => Un-Highlight Other Buttons," sends an "Un-Highlight Other Buttons" author message to the element's parent. This message is then broadcast to all of the buttons in the scene, and to their modifiers which are configured to respond by un-highlighting their elements. The next additional messenger, called "Execute => Highlight," sends the opposite message to the scene, causing the buttons in the scene to highlight.

The additional behavior (Figure 8.12) contains two messengers that highlight and un-highlight the button when the mouse is tracked inside and outside the boundaries of the button element. These actions are switched off on receipt of the "Execute" author message so that when the button is pressed down, tracking over it again won't un-highlight it.

Animation Modifiers

The orbit mToon on the mToon Ranges scene contains a behavior and two messengers. Since the functionality of the behavior is also required in the next scene, mToon Speed, the behavior has been aliased.

The alias "Demo mToon" behavior contains the following modifiers:

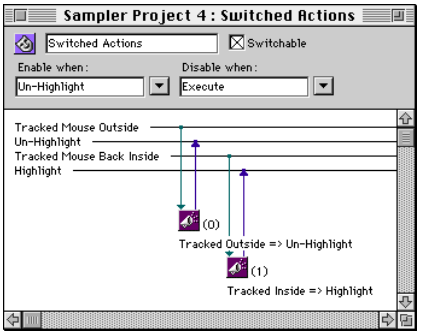


Figure 8.12 Contents of the Switched Actions behavior

- A graphic modifier that applies a transparent background ink effect.
- A drag motion modifier that allows the user to drag the animation around the screen within a limited area. To define the boundaries within which the animation can be dragged, the animation is made a child of an invisible element with the parent/child tool. The drag motion modifier on the animation is then configured to confine the user's dragging of the element to the boundaries of its parent. For more about parent/child relationships, see "Effects of Parent/Child Relationships" on page 8.7 of the *mTropolis Reference Guide*).
- A behavior that contains three cursor modifiers used to simulate a hand picking up the animation. These modifiers are activated on Mouse Over, Mouse Down and Mouse Up messages respectively.
- Two Miniscript modifiers that reset the position of the animation on Scene Started and Scene Ended messages. By using these Miniscripts, if the user moves the element during runtime, its position will be reset when the scene ends.

Changing the Range of Animations

The messenger on the first button, called "Set Orbit Range 1," acts on receipt of the Execute message. It sends "Set Orbit Range" with the variable "Orbit Range 1" to the scene. The Miniscript on the Orbit.toon animation, called "Set Range," responds to this message by setting the range of the Orbit.toon animation to the value of the incoming integer range variable.

The messenger on the Orbit.toon animation, called "Play Animation," is also activated on a "Set Orbit Range" author message. Upon receipt of this message, it sends a *Play* command to the mToon, causing the animation to play.

Except for the value of the variables that they send, the programming of each of the buttons is the same.

- *Note: Try duplicating the animation object (Edit menu–Duplicate, or ⌘-D), then run the project. Now drag the animation elements away from each other, and click one of the buttons.*
- The "Demo mToon" behavior has been aliased, because it is used in the next scene, called "mToon Speed," as well.

Changing the Rate of Animations

The scene called "mToon Rate" functions similarly to the mToon Ranges scene, except that the messengers on the three gear buttons

set the speed of the animation rather than the range of cels to play. The variables to which they refer have been placed on the scene.

The Orbit.toon animation has been configured to play Range 2, cels 10–19 at various speeds when one of the three buttons is pressed.

The Gear Buttons

The speed of the animation is set when the user clicks on one of the gear buttons.

Each of the buttons, labeled Slow Button, Medium Button, and Fast Button, sends an author message named "Set Orbit Speed 1," "Set Orbit Speed 2," or "Set Orbit Speed 3" with a integer variable called "Orbit Rate 1," "Orbit Rate 2," or "Orbit Rate 3" to the scene when it is clicked. The variables to which each of these messengers refer have been placed on the scene.

- *Note: A Miniscript modifier could also be used to set the animation's speed. See Chapter 14 of the mTropolis Reference Guide, "Miniscript Modifier", for details.*

The Animation

The mToon on this scene has an aliased behavior, two messengers and one Miniscript.

For details regarding the functioning of the alias "Demo mToon" behavior, see its documentation earlier in this section entitled "Animation Modifiers."

The first messenger called "Set Default Range" sets the range of the animation with the integer variable named "Orbit Range 2" that is located on the scene.

On receipt of the author message "Set Orbit Speed," the Miniscript modifier named "Set Speed" sets the rate of the mToon to the value of the incoming integer variable.

The second messenger, called "Play Animation," sends a *Play* command on receipt of the "Set Orbit Speed" author message.

COMMUNICATING

The "Communicating" example, available from "The Basics" menu, shows how to use mTropolis' messaging system to switch the states of objects in a lighting system. Also featured is the use of aliased behaviors to associate identical copies of the behavior with various elements. Finally, this project also shows how to control mToons (mTropolis' proprietary animation format) with modifiers, especially 'if' and Miniscript modifiers.

Description

In this project, a small lighting system is modeled, comprised of a main power switch, indicator lights, and two light bulbs with their own power switches (Figure 8.13).

When the main power switch is on, it supplies electricity to the light bulb switches. Indicator lights on the light switch plates turn green to show power has reached the light switches. Clicking on the light bulb switches turns the light bulbs on and off. The bulbs can be dragged from their sockets by the user. If they are lit, removing them from the socket causes them to go out. The bulbs are also interchangeable between the two sockets.

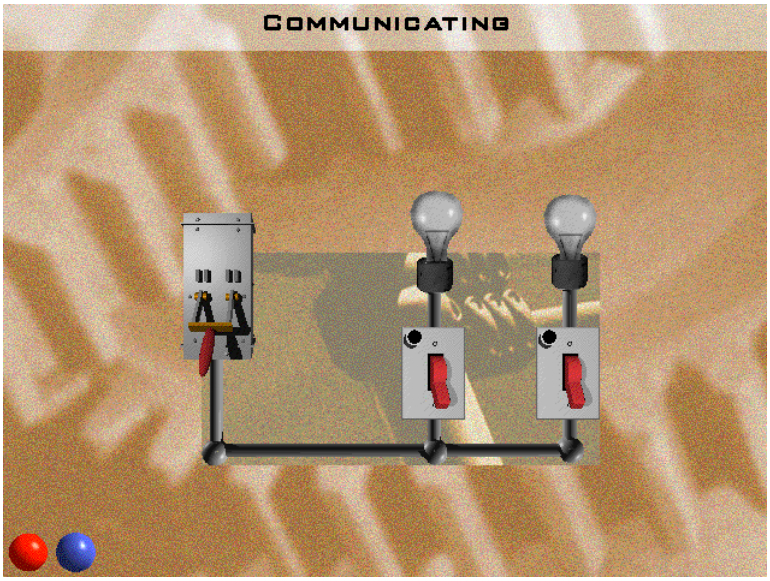


Figure 8.13 The Communicating example lighting system

Since both bulbs behave in the same way, the behaviors on the light bulbs have been aliased.

How the Lighting System is Programmed

The project uses mToons, mTropolis' proprietary animation format, to represent the on/off states of switches and lights. mToons were chosen because the display of the cels of the animation can be controlled by modifiers. Because the basic states of switches are up or down, and lights are on or off, all mToons consist of two cels. For example, if the main switch is off, the cel showing the switch in its down position is displayed. If the switch is on, the cel displaying the switch in its up position is displayed.

The messaging system uses a combination of 'if', Miniscript, and variable modifiers to manage the state of the system. The authoring is designed to implement the following logical construction:

- If the main power switch is on, and
- if the light bulb switch is on, and
- if the light bulb is in the socket,
- then display the light bulb in its on state.

The on/off states of switches are stored in Boolean variables placed on the scene. This position makes them accessible to all elements in the scene.

- *Note: See "Variable Scopes" on page 13.25 of the mTropolis Reference Guide for details regarding the placement of variables.*

Collision messengers are used to determine if the bulb is in the socket. These reside on an "empty" graphic element locked into position over the bulb stem.

Author messages are used to signal other elements as to the presence of the bulb in the socket and the power state of switches. For example, the Socket Mask that hides the bulb stem receives author messages.

Since the light bulbs use the same logic to determine if they are on (lit) or off (unlit), an alias of the "Bulb Behavior" is used.

User interaction with the switches and bulbs are handled by modifiers residing on the respective elements. For example, modifiers on the main power switch are configured to respond to a mouse click and to notify other modifiers to initiate new actions.

Each of the parts of the lighting system will be described in turn.

The Scene

The scene component contains the background image.

The three Boolean variables on the scene store the current state of the switches, and, therefore, the flow of electricity through the system. These are checked by modifiers attached to the elements in the scene. The variables are as follows:

- **switchAon**: the light switch for the left light bulb
- **switchBon**: the light switch for the right light bulb
- **thePowerIsOn**: the state of the main power switch

When the scene starts, the Miniscript modifier named "Reset variables" sets the variables to "false" ("off"). The lighting system is off.

Main Power Switch

The element called "Main Power Switch" turns power on and off for the entire lighting system.

The Miniscript modifier named "Set Handle Position and Power" on the main switch element controls the state of the power and the position of the power handle. It tells the switch animation which frame to play and sets the value of the Boolean variables on the scene. The Miniscript also sends an author message "PowerON" or "PowerOFF" to the scene to notify all elements of the state of the power supply. Since author messages are broadcast from the scene to its elements and modifiers, the modifiers are automatically notified when the user has turned the main power on or off.

Indicator Lights

The indicator lights respond to PowerON and PowerOFF messages sent by the main power switch.

The Indicator Light elements contain an aliased "SwitchLight" behavior. Two Miniscript modifiers encapsulated in the "SwitchLight" behavior listen for the on/off messages from

the main power switch. When they receive a "PowerON" author message they play the cel that shows a green light in the Indicator animation. A "PowerOFF" author message plays the other cel (the black light).

Light Bulb Switches

The light bulb switches (Switch A and Switch B) control the flow of electricity to the light bulb from the main power supply.

Each switch element contains a Miniscript modifier called "Toggle Switch State" that toggles the switch animation between the up and down frames. It also toggles the switchAon or switchBon variables on the scene between true or false. That is, if the light bulb switch is currently on, and the user clicks on the switch, the modifier sets the respective variable to false, and vice versa.

Power Source A and Power Source B Elements

Empty graphic elements called "Power Source A" and "Power Source B" are located within the boundaries of the light bulb sockets. They contain the most important modifiers for implementing the logic between the bulbs and the power sources. Specifically, they turn the associated light bulb on by sending a "Turn On" author message to the light bulb element. Their associated behaviors are similar, although their state depends upon the state of the associated power switch and the state of the main power switch.

Each Power Source element behavior encapsulates three other behaviors. These divide programming into three logical sections: User Drags Bulb, Power Up Checks, and Power Down Checks.

- The behavior called "User Drags Bulb" simply turns the bulb on and off as it is dragged over the light bulb socket. It encapsulates two modifiers, one that sends a "Turn ON" author message to the bulb if the power is on to the socket and the light bulb switch is on, and one that sends a "Turn OFF" author message to the bulb if the bulb is out of the socket.
- The behavior called "Power Up Checks" uses collision detection to test for the existence of a bulb in the socket when the main power is turned on, or alternately when the light bulb switch is turned on. The two 'if' modifiers contained within this behavior decide whether to test for a bulb in the first place.

  The first 'if' modifier, named "Check for bulb on PowerON," is executed upon receipt of a "PowerON" author message, that is sent from the main power switch. Then the modifier checks if the appropriate power switch is on by checking the state of the associated variable (e.g., SwitchAon). If the variable is true, then a check is made for a light bulb with a collision modifier named "Check for a bulb over me (ON)." If this is true, the "Turn ON" author message is sent to the message sender (the light bulb).

  The second 'if' modifier is named "Check for bulb on Switch A ON," and executes its test upon receiving the "Switch A ON" author message from the power switch. It

checks the value of the variable called "thePowerIsOn." If this value is true, then a check is made for a light bulb with the collision modifier named "Check for a bulb over me (ON)." If this is true, the "Turn ON" author message is sent to the message sender (the light bulb).

The collision modifier in this behavior named "Check for a bulb over me (ON)" tests for a light bulb over the Power Source element when it is sent a "Check for a bulb (ON)" author message from either of the above 'if' modifiers. Notice that the modifier enables and disables itself on the same message. This implements a quick collision test. The "Which bulb is here? (ON)" author message is sent once to all elements colliding with the Power Source element at the instant the "Check for a bulb (ON)" author message is received.

If a light bulb was detected, the "Which bulb is here? (ON)" author message is sent to the light bulb (the message sender). If the bulb is detected, it responds by sending back the author message "A bulb is here (ON)." Upon receipt of this message, the last messenger in this behavior sends back to the light bulb a "Turn ON" author message.

- The behavior called "Power down checks" looks for a bulb over a power source. If a bulb is present, it turns it off when the associated power switch turns off.

  On receipt of a "Switch A OFF" author message, the first messenger named "Check for bulb on Switch A OFF" sends a "Check for a bulb (OFF)" author message to itself.

  The collision messenger named "Check for a bulb over me (OFF)" functions the same as in the above behavior.

  The final messenger in this behavior sends a "Turn OFF" author message to the sender of the "A bulb is here (OFF)" author message.

Light Bulbs

The Light Bulb elements are programmed identically, that is they use an aliased behavior called "Bulb Behavior." Inside the behavior lies a graphic modifier named "Transparent Black" that makes the background of the bulb transparent, and a drag motion modifier named "Drag" to enable dragging of the bulb. The last item in the behavior is a messenger that sends the author message "Turn OFF" to the Miniscript on the bulb. Two Miniscript modifiers set the cel of the bulb animation to 1 or 2 on the "Turn OFF" or "Turn ON" author message respectively. There is also a behavior within the "Bulb Behavior" called "Cursors." It contains three cursors that change the Hand cursor to a Hand Clenched cursor.

The bulb elements each have a small child element named "Bulb Stem."

Bulb Stems

The presence of the light bulb in the light bulb socket is tested by a small empty graphic element named "Bulb Stem." It is attached to the light bulb as a child element.

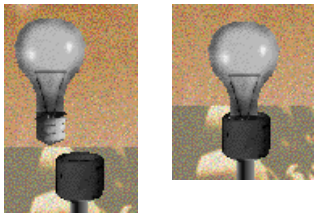


Figure 8.14 The Socket Mask effect

To study the "Bulb Stem" behavior, open the structure window. Then click on the expand triangle beside one of the Light Bulbs. Select the Bulb Stem element. Then open the layout window to see the Bulb Stem element highlighted with a thick dotted line.

The purpose of the Bulb Stem elements is to limit the collision region of the bulb. If the entire bulb were an active collision region, touching a power source with the glass part of the bulb would turn on the bulb!

Notice that the Bulb elements used an aliased behavior ("Bulb Stem" behavior). This allows the developer to apply the programming of one bulb to the other.

The two collision modifiers within the Bulb Stem behavior send the author messages "Bulb is here" and "Bulb is gone" to all objects colliding or "uncolliding" respectively with the bulb stem.

The messengers named "Bounce 'Turn ON' to parent" and "Bounce 'Turn OFF' to parent" simply tell the light bulb to turn itself on or off when the stem gets these messages. These modifiers are used in the conversation between the bulb stem and a power source.

The other pair of modifiers, named "Bounce back answer (ON)" and "Bounce back answer (OFF)," send a message back to the message sender (a Power Source element) that says a bulb is here.

Sockets and Socket Masks

Socket A and Socket B are empty elements containing messengers that send messages to their respective Socket Masks.

The Socket Mask is used to hide the bulb's stem when it is placed inside the socket (Figure 8.14).

The mask, which is shown dark in the figure, is invisible to the user.

Upon receipt of a "Bulb is here" author message, the Socket sends a *Play* command to its respective, initially hidden, Socket Mask element. The Socket Mask element then appears over the light bulb stem and the background. This is because it has a higher draw order number than both the background it is placed over, and the bulbs.

When the Socket receives a "Bulb is gone" author message, a *Stop* command is sent to the Socket Mask element, making it disappear from view.

The Bulb is here and "Bulb is gone" author messages are sent by collision modifiers on the Light Bulbs.

CHANGING CURSORS

The "Changing Cursors" example, available from "The Basics" menu, shows how to use mTropolis' cursor modifier to dynamically change the cursor icon as it passes over selected elements.

Description

mTropolis uses a system pointer as its default cursor. It also uses a Hand Pointing Up when the mouse is over an element that has been configured with a modifier to respond to a mouse message. However, several other types of mouse cursors are stored by the cursor modifier. Figure 8.15 shows the list of cursors available on the pop-up menu in the cursor modifier.

The first four examples in the project show four of these cursors (Hand Up, Hand Right, Hand Left, Hand Down). In each case, the cursor is changed to the one selected in the cursor modifier pop-up when the mouse cursor passes over the gear button element.

The fifth example, named "Draggable Object," uses multiple modifiers to change the cursor and make the gear element draggable.

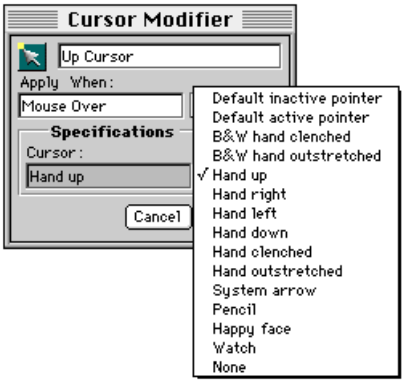


Figure 8.15 Cursors available in the cursor modifier

Draggable Object

The fifth gear element can be dragged and dropped during runtime within the boundaries of the scene.

At scene start, a drag modifier named "Allow Dragging" sends a message to the element to allow dragging. The cursor modifier, named "Cursor Initial State," changes the cursor to the Hand Outstretched on a Mouse Over message.

The cursor modifier named "Clenched Cursor" changes the cursor to the Hand Clenched on a Mouse Down message.

The cursor modifier named "Outstretched Cursor" changes the cursor to the Hand Outstretched on a Mouse Up message.

The Miniscript modifier named "Reset Element Position" resets the element to its initial position when the scene ends.

REVEALING OBJECTS

The "Revealing Objects" example, available from "The Basics" menu, shows how to display images from another scene, the current scene, or a shared scene.

Description

In this example, clicking on buttons in the scenes allow the user to see the current image in another scene; hidden images in the current scene, or, images from other scenes in the current scene.

The following section outlines the methods used to program messengers and change scene modifiers to:

- Show images from another scene.
- Show hidden images in a current scene.
- Show images from the shared scene.

Button Behavior

The gear button uses two behaviors: Bevel Button behavior and the Radio Button behavior. The first behavior provides typical highlight and un-highlight button behavior, and the second behavior makes the button continue to appear depressed while the image is displayed on the scene.

Showing Images from Another Scene

The gear button in the first scene contains a change scene modifier that is configured to show the current scene under another specified scene when the user clicks on it.

When the user clicks on the button and the change scene occurs, the element called "Layered Image" appears over the scene called "Show Image From Another Scene." That is, the first scene is layered under the new scene.

Layered Image Scene

The scene called "Layered Image" contains two transition modifiers. One applies a random dissolve effect on receipt of the "Scene Started" message, the other applies the same effect on the "Scene Ended" message. As a result, when the user clicks on the button, the image appears to dissolve into and out of the background image.

The user may only change scenes from this scene by clicking on the image. This is accomplished by layering the elements in this scene to confine the active area of the screen.

In order to disable invalid user choices, that is, the navigation buttons at the bottom right of the scene and the top left of the scene, a transparent element called "Disable User Actions Button" has been placed over the scene, but under the element called "Smokestacks.pict."

- *Note: A messenger called "Absorb Mouse Messages" configured to receive (and absorb) the mouse messages generated by user's mouse clicks has been placed on the invisible element in the scene. This keeps any additional modifiers on other elements from inadvertently being triggered when the user clicks on the screen.*

To view the layer order of these two elements, switch to the layout window. The elements are visible at positions 7 & 8.

The layered element called "Smoke Stack" is also modified with a return modifier that is activated on a "Mouse Up" message. Clicking on the smoke stack image returns the user to the first scene.

There is one disadvantage to layering scenes and that is slightly slower performance, since more than one scene needs to be "processed."

Showing a Hidden Image on a Scene

The next example uses simple messaging to show and hide an image called "Man In Rain.pict." This method is used for images to be shown on a single scene, called "Show from Scene."

Gear Button

Both the Standard Button and the Bevel Button behaviors are used to create the functionality of this button. (*See* "Common Basics Projects Features" earlier in this section for information on the use of button behaviors.) A single messenger, called "Show Scene Image," also is placed on the element. On receipt of the author message "Execute," this messenger sends a *Play* command to the Man in Rain.pict, causing the image to appear on the scene.

The Man in Rain

Two messengers have been placed on The Man in Rain.pict element. The first, called "Hide on Mouse Up," sends a *Stop* command to its element on a receipt of a "Mouse Up" author message. On receipt of the message "Stop-Hidden," the second messenger on the PICT, called "Un-Highlight Button," sends the author message "Un-Highlight" to the scene. This message activates the programming of the Bevel Button behavior, causing it to un-highlight the gear button.

Showing Images in a Shared Scene

The next example shows an image from the shared scene. During runtime the shared scene appears behind all other scenes in a subsection.

The Gear Button contains a messenger called Show Shared Scene Image. When the user clicks on the gear button, this messenger sends the message "Show Shared Scene Image" to the shared scene.

The element named Power Lines.pict in the shared scene has four messengers:

- The messenger called "Hide on Scene Changed" sends a *Stop* command to the element on receipt of a "Scene Changed" author message. This causes the image to disappear when a scene change occurs (i.e., when the user clicks on the button).
- The messenger called "Show Element" sends a *Play* command to the element on receipt of the "Show Shared Scene Image" author message.
- The messenger called "Hide on Mouse Up" sends a *Stop* command to the element on a receipt of a "Mouse Up" author message.
- On receipt of the message Stop Hidden, the messenger called "Un-Highlight Button" sends the author message "Un-Highlight" to the active scene. This message activates the programming of the Bevel Button be-

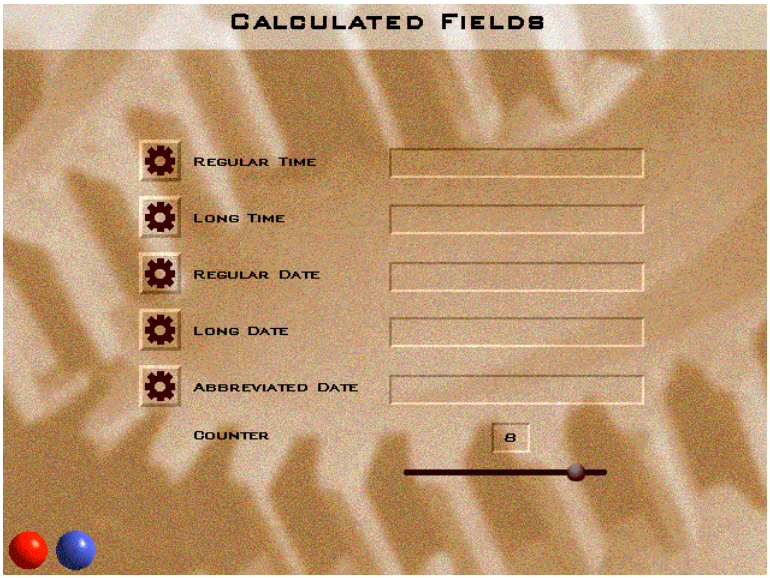


Figure 8.16 The Calculated Fields example

havior, causing it to un-highlight the gear button.

CALCULATED FIELDS

The "Calculated Fields" example, available from "The Basics" menu, shows the methods for displaying variable data in text elements, such as the display of the current system date, or counters that display values dynamically.

Description

The Calculated Fields project shows how to display values in text elements using a variable or a reserved word. The current value of the variable or reserved word is substituted at runtime.

The first five examples in this project use the reserved words: "time," "long time," "date," "long date," and "abbrev date." These are replaced with the computer's current date and time at runtime.

The last example uses a counter that changes interactively during runtime as the user drags on a slider.

Showing the Date and Time

Figure 8.16 shows the buttons, display boxes, and date and time text displays in the scene.

The five date and time displays have the following in common:

- The buttons along the left side of the screen are created using the methods described in "Common Basics Projects Features" earlier in this section.
- Between the buttons and the display boxes are text labels. These are standard text elements containing graphic modifiers.
- The Date and Time display boxes are created with graphic elements and image effect modifiers. The modifiers are configured to give the display box an inverted beveled edge.
- A text element is added on top of the display box. The text element contains the reserved word that is updated at runtime. It is hidden when the scene starts.
- When the user clicks on a button, a messenger sends a *Play* command to the text element on the display box.
- A messenger on the text element is configured to respond to the *Play* command by updating the reserved word. The current date or time is then displayed.

The displays are created using similar methods, with the following exceptions and notes.

Regular Time Display

The timer modifier named "Update Text Parameters" starts a looping timer on the Played message. On each iteration of the loop, the modifier sends an *Update Calculated Fields* command to the element. This updates the time value in the text field.

Long Time Display

The Long Time display is configured the same as Regular Time, but appends seconds to the display as well.

Regular Date Display

The Regular Date contains the system's current Short Date, as defined in the system's Date & Time control panel (e.g., 12/24/95).

Long Date Display

This field contains the long version of the system date (e.g., December 24, 1995).

Abbreviated Date Display

This field shows yet another view of the system date (e.g., Dec. 24, 1995).

Showing a Graphical Counter

The Calculated Fields example includes a graphical counter (labeled "Counter") and its accompanying slider (Figure 8.16).

How the Counter Works

The user slides a knob along a slider, changing the value in a display box. Moving the slider to the right changes the value in one unit increments from 0 to 9 (ten units). Sliding the knob to the left reduces the display in one unit decrements.

The Parts of the Counter

The counter has the following parts:

- The label "Counter" is a standard text element called "Counter Text."
- The slider is an element called "Counter Slider" linked to an image of a slider.
- The knob is an element called "Knob.pict" linked to an image of a knob.

- The counter display box is created in the same way as the display boxes in the date and time examples. It is an "empty" element to which an image effect modifier has been added. The modifier creates bevels along the boundaries of the element.
- A text element in the display box contains a word surrounded by angle brackets: <numHolder>. The angle brackets prompt mTropolis to look for an Integer variable, called "numHolder."

How the Counter was Programmed

As the knob is moved from one position along the slider to the next, its current "counter value" is calculated and sent to the counter display. Most of the functionality of the graphic counter resides in modifiers on the knob element. The knob element is a child element of the Counter Slider element.

Finding the Width of the Slider

The width of the Counter Slider element is calculated at runtime by a Miniscript modifier named "Store width of parent" (the slider is the parent). The current width of the slider is stored in a variable called "pWidth."

Dividing the Width into Counter Units

The width of the slider is divided into ten units displayed by the counter. For example, if the slider is 200 pixels wide, then a movement of 20 pixels changes the count by 1.

A value of 0 represents the leftmost extent of the Slider element and is stored in a variable named "sliderMin." A value of 9 represents the rightmost extent of the element and is stored in a variable named "sliderMax."

Since the counter values are relative to the width of the slider element (it's divided into units of ten), changing the width of the slider element does not affect the range of values displayed.

The values given to the sliderMin and sliderMax variables can be changed in edit mode, allowing the author to change the relative range displayed by the counter. For example, changing sliderMax to 100 creates a counter that tracks the movement of the knob in units of 1 between 0 and 100.

Moving the Knob

The knob has a drag motion modifier named "Drag Horizontal Only" that has been set to constrain the movement of the knob to the boundaries of the parent Counter Slider.

Calculating the Knob's Relative Position on the Slider

When the user moves the knob, its messenger called "Update Slider on Mouse Tracking" sends the author message "Update Slider Value" to the knob element.

A Miniscript modifier, called "Calculate (sliderValue)" listens for this message and calculates the knob's current screen position relative to the Slider (Figure 8.17).

This figure is converted into a value between 0 and 9. The value is stored in the "sliderValue" Integer variable.

Updating the Counter Display

A messenger on the knob called "Update Number Field" hears the same "Update Slider Value" author message as the Miniscript. It sends a "New Number" author message along with the new value for sliderValue to the scene.

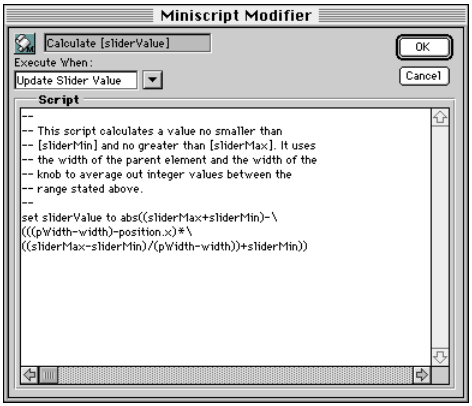


Figure 8.17 The Calculate [sliderValue] Miniscript modifier

A messenger named "Update Text Parameters" on the display box hears the "New Number" author message and sends an *Update Calculated Fields* command to the Counter Text element.

The Miniscript called "Update Number" on the Counter Text element also acts on the "Update Number" author message by updating the numHolder Integer variable. This causes the text element to display the new Counter Display value.

CONTROLLING AUDIO

The "Controlling Audio" example, available from "The Basics" menu, shows how mTropolis' sound effect modifier and sound fade modifier can be used to control the playback of sounds.

Description

This project uses a shared scene and two other scenes called "Controlling Audio 1" and "Controlling Audio 2."

The first scene illustrates different uses of both sound elements and the sound effect modifier. It contains a single button on the left, and three sets of buttons on the right.

Two sound effect modifiers have been placed on the single button, a graphic element. When this button is clicked upon, its sound effect modifiers are activated, causing two on/off aiff sounds linked to them to play.

Each pair of buttons controls the on/off states of three sound elements called "Water.aiff," "Frogs.aiff," and "Birds.aiff." By clicking on the buttons, each of the aiff files linked to the sound elements can be played or stopped, or they can be played or stopped while the other sounds play.

The second scene illustrates the programming of four more buttons used to control a single sound element. Clicking on one of each pair of buttons fades the sound in or out, or increases or decreases its volume.

Linking Sounds to a Project

In the mTropolis environment, sounds can be linked to a project in two ways. Individual sound files can be linked directly to sound elements, or indirectly to other elements via sound effect modifiers. As sound elements have no graphic component, they cannot be created in the layout window.

To create a sound element, switch to the structure window. Select an element and choose **New Sound** from the Object menu. The new sound element, represented by a speaker icon, will appear in the selected location. Snd or aiff sound files may now be linked to the selected element by choosing the **Link Media** option from the File menu.

To link sound files using the sound effect modifier, drag this modifier from modifier palette group 2 to any element, in any view. Double-click its icon to open its dialog. Select **Link File** from its Sound pop-up menu. Select a snd or aiff file from the navigation dialog that appears.

Controlling Audio 1

Two sound effect modifiers were placed on the left button, an mToon graphic element. Individual sound files were then linked to these sound effect modifiers. Each modifier was then configured to respond to author messages generated by the Standard Button behavior in response to the user's mouse actions.

Each on/off button element in the scene is also an mToon graphic element. For each pair of buttons, a sound element was created as its sibling in the structure window. Each sound element was then linked to a sound file: Water.aiff, Frogs.aiff, and Birds.aiff.

Instead of the Standard Button behavior, Radio Button behaviors have been used to switch one button on when the other is off. As this behavior's programming requires each pair of buttons share the same parent, they have been made children of the text element that describes them. This allows author messages generated by the Radio Button behavior that are targeted to a parent (in this case, the text element) to be simultaneously received by all child elements and their modifiers (in this case, the on/off button elements, the sound element, and all their modifiers).

With the exception of messengers that send *Play* or *Stop* commands to each sound element, the programming of all "off" buttons and all "on" buttons is the same, allowing aliases to be used on each.

Single Sound Button

Like the navigation buttons documented in "Common Basics Projects Features" on page 8.3, the Single Sound button is a two-celled mToon animation that uses the programming of the Standard Button behavior and the Bevel Button behavior.

These behaviors control the display of the highlighted or un-highlighted cel of the gear mToon animation, and they function to apply highlighted and un-highlighted effects to the button element's beveled edges. They also send out messages in response to the user's mouse actions, and these are used to activate the element's two sound effect modifiers. These modifiers are described below.

Click In on Highlight

The Highlight message generated in response to a mouse down on the button activates the sound effect modifier called "Click In on Highlight."

Click Out on Execute

The Execute message generated in response to a mouse up on the button activates the sound effect modifier called "Click Out on Execute."

The Water, Bird and Frog on/off buttons in Scene 1 are all programmed alike. The programming of the first set of buttons is documented below:

Water Off Button

The Water Off button is a two-celled mToon element configured with two behaviors and three messengers.

The aliased messengers used to send Execute and Highlight messages activate the modifiers in the Radio Button and Bevel Button behaviors when the scene starts. These behaviors function together to show the highlighted cel of the mToon animation, and to simultaneously apply the highlight bevel effects to the button's beveled edges. When the user clicks on the Water On button, modifiers in the Radio Button behavior are again activated, causing the On/Off buttons to switch states.

The messenger called "Stop Sound Execute" is used to control the sound file. The author message "Execute" is generated when the scene starts and again when the user clicks on the button. This triggers the messenger which sends a *Stop* command to the Water.aiff.

Water On Button

The Water On button is also a two-celled mToon element. With the following exceptions, the programming of the Water On and Water Off buttons is the same.

- The messengers "Highlight on Start" and "Execute on Start" are not used on the Water On button. This is because the Water On button does not appear selected when the scene starts.

- The messenger "Stop Sound Execute" is replaced by the messenger called "Play Water Sound." This messenger sends a *Play* command to the "Water.aiff" when the user clicks on the button.

Controlling Audio 2

In this scene, four buttons called "Fade In," "Fade Out," "Volume Increase" and "Volume Decrease" are used to control a single sound element.

Fade In/Fade Out Buttons

Since the first pair of buttons are radio buttons that may be on or off, they are configured with the Radio Button behavior as well as the Bevel Button behavior. The programming of the Radio Button behavior used to switch the button's states is designed to send messages to the button's parent. As a result, the buttons have both been made children of "Fade In Button Label." This allows them to receive the messages that will activate/deactivate them simultaneously.

Fade In Button

The messenger called "Fade In" on the Fade In button is configured to send the author message "fadeIn" to the scene on receipt of the "Execute" author message. (This message is generated by a messenger in the Radio Button behavior in response to the user clicking on the button.)

The "fadeIn" author message does two things. It activates the messenger on the sound element, which sends the command *Unpause* to the Festival.aiff, and it activates the sound fade modifier called "Fade to Full" on the sound element. This modifier causes the sound to increase, that is, to fade in.

- *Note: The initial state of the sound element is set to Pause in the sound element's element info dialog.*

Sound Fade Modifier

The percentage of volume to which the original sound file is to fade, and the duration of the sound's fade is set from the dialog of the Fade to Full sound fade modifier. To see this dialog (Figure 8.18), Double-click its icon on the Fade to Full sound element.

Figure 8.18 The Fade to Full sound fade modifier

Fade Out Button

The programming of the Fade Out button is the same as that of the Fade In button, with the exception of the messenger that sends the author message "fadeOut."

The messenger called "Fade Out" on the Fade Out button is configured to send the message "fadeOut" to the scene on receipt of the Execute message. (Again, this message is generated by modifiers in the Radio Button behavior in response to the user clicking on the button.) When received by the sound fade modifier called "Fade to Silence" on the sound element Festival.aiff, the sound fades away.

To cause the sound to fade, the dialog of the Fade to Silence sound effect modifier has been configured to fade the sound out to 0, over a duration of 5 seconds.

Volume Buttons

Unlike the Fade In and Fade Out buttons, the Volume buttons are not programmed with a Radio Button behavior. Since clicking on one or the other increases or decreases the volume of the sound incrementally, neither button is simply off or on. As a result, the Standard Button behavior is used.

In addition to controlling the selected or deselected appearance of the volume button, the Execute message generated by the Standard Button behavior in response to mouse actions triggers the messenger on the Volume Up or Volume Down button. Depending on the button selected, "increaseVol" or "decreaseVol" is sent to the Festival.aiff, which in turn activates either the Increase Volume or Decrease Volume Miniscript. Figure 8.19 shows the script of the Increase Volume Miniscript.

Volume is an inherent property of any sound file. Since the sound fade modifiers on the sound element may have altered this property, this script is designed to assesses its current value. If the percentage of the current volume of the sound is less than 90, it will be

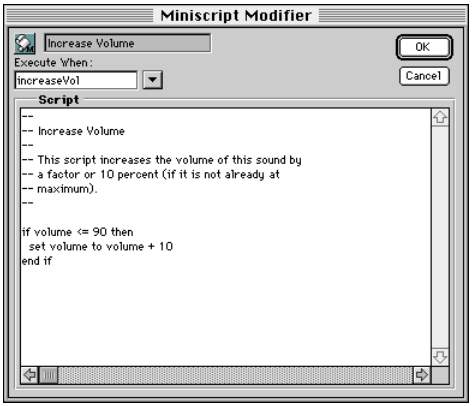


Figure 8.19 The Increase Volume Miniscript

increased by 10, bringing the volume of the sound file to its maximum possible value (100%).

The Decrease Volume Miniscript on the Decrease Volume button also checks the volume of the sound. If the percentage of the current volume of the sound is greater than 10, it will be decreased by 10, bringing the volume of the sound file to its minimum possible value (0%).

MODIFIER EXAMPLES

Select "Modifier Examples" from the mExamples main menu to display a submenu (Figure 8.20) that gives access to many simple examples that demonstrate the use of each mTropolis modifier.

Click on an item in the list to display the example. Each example has three buttons at the bottom of its scene:

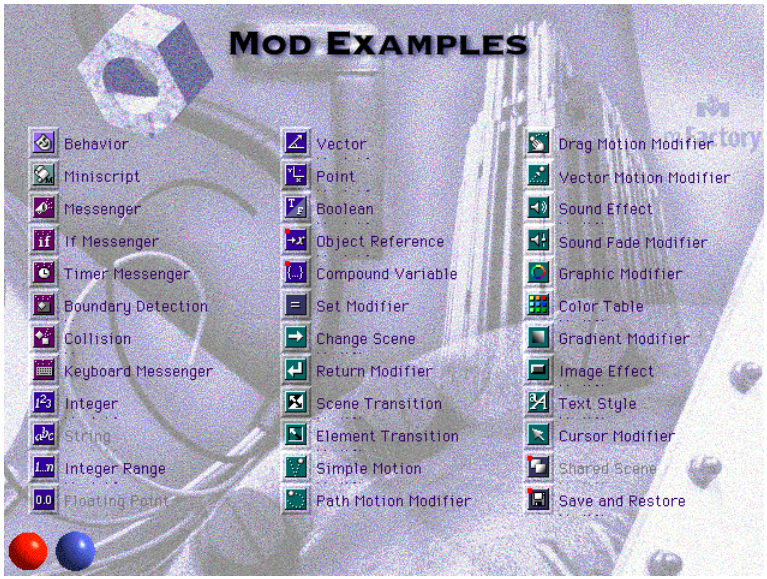


Figure 8.20 The Modifier Examples submenu

- **About this Modifier**: Click and hold on this button to see a short description of the modifier being demonstrated.
- **About this Example**: Click and hold on this button to see a short description of the current example.
- **Modifier Icon**: Click and hold on the modifier icon to see the configuration dialog for this modifier.

SIMPLE PROJECTS

Select "Simple Projects" from the mExamples main menu to display a submenu that gives access to some advanced mTropolis projects. The "Autonomous Behaviors" example uses behaviors to create portable programming that can be reapplied in the same project or saved in libraries for use in other projects. The "Character Interaction" example features user-controlled animations using pre-set motion paths in "2.5D" space. These examples are described below.

AUTONOMOUS BEHAVIORS

This example is designed to illustrate authoring that can be used and re-used on various media elements. Once configured, the functionality of the behavior can be copied and reused in other locations in a project, or saved in a library for use in future projects.

Description

A butterfly and a bug appear between the pillars in a temple scene. The bug has been programmed to crawl over the floor, and to squish when the user clicks on it. The butterfly is programmed to flutter in random directions in the air. It has also been programmed to fly away when it detects the proximity of the user's mouse cursor. To see the programming in action, run the project and try clicking on the butterfly or the bug.

The programming of these animations has been modularized so that the behavior of each may be easily copied and pasted onto other objects. In edit mode drag and drop the Bug behavior from the bug element to the butterfly, and the Butterfly behavior from the butterfly to the bug. Run the project again and try clicking on either animation to see the effect. The bug will evade the user's cursor, and when clicked on, the butterfly will squish.

The Temple Scene

Custom palettes created in other applications and saved as CLUT files may be linked to a mTropolis project and used in any scene via the color table modifier. In this project, the color table on the scene has been configured to display its graphic elements using Temple CLUT v.2.0 when the scene starts.

Blue Bug.toon

A behavior called "Bug" and a variable called "actualSize" on the bug animation determine the bug's functionality.

actualSize

The point variable called "actualSize" on the bug animation contains the actual pixel size of the bug media file. This variable is used to restore the dimensions of the element if the bug is squished during runtime.

Bug
Within the behavior called "Bug" is an aliased graphic modifier called "Transparent Ink." This modifier is used in both the bug and butterfly animations to make the background transparent.

Next in the Bug behavior are the following variables:

changeOdds
This integer variable defines the odds of the butterfly changing direction.

directions
This integer variable specifies the number of possible directions in which the bug may move.

direction
This integer variable contains the direction (in degrees) in which the bug currently moves.

normalSpeed
This integer variable contains the value that defines the normal speed of the bug's movement.

motion
This Vector variable contains the angle and magnitude of motion of the bug.

There is another behavior named "Bug Motion" in the Bug behavior.

Bug Motion
This behavior contains all the programming pertaining the motion of the bug on the screen.

Initialization
The first modifier in "Bug Motion" is a Miniscript named "Initialization." This is used to initialize the motion of the bug.

Move
The vector motion modifier called "Move" in the Bug Motion behavior is used in conjunction with the vector variable called "motion" to set the motion of the bug.

Change Direction Timer
This messenger sends a "Change Direction Test" author message to the element every 0.80 seconds.

Change Direction Test
To determine when to change the direction of the bug's movement, on receipt of the author message "Change Direction" the Miniscript modifier called "Change Direction Test" uses the rnd (random) math function with the changeOdds integer variable. If the evaluated expression equals 1, the author message "Change Movement" is sent out, causing the Miniscript called "Change Movement" to set new direction variable values. The script of the Change Motion Miniscript is shown in Figure 8.21.

Notice the use of named ranges within the Change Movement script in Figure 8.21. Named ranges allow the instructions of this script to be applied to any mToon with cel ranges named "Up," "Down," "Left," and "Right."

- *Note: Since Miniscript looks for the name of the range rather than specific cel numbers of*

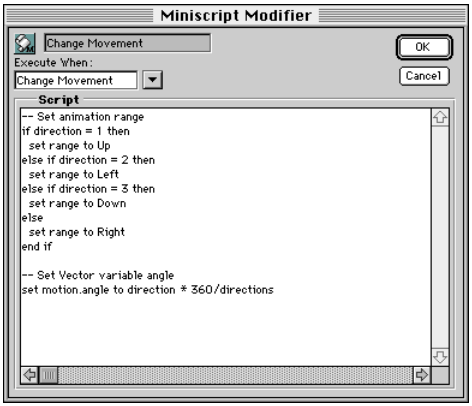


Figure 8.21 The Change Movement Miniscript modifier

the mToon, there may be any number of cels in a new range of the same name.

The last item within the "Bug Motion" behavior is another behavior, called "Bounce Off Container Walls."

Bounce Off Container Walls Behavior
This behavior contains a border detection messenger and a Miniscript modifier used to change the direction of the bug if it hits the walls of its parent container (i.e., in this case, the edges of the scene).

Bounce Off Container Walls Messenger
The border detection messenger in the behavior (also called "Bounce Off Container Walls") sends a "Reverse Direction" author message to the messenger's parent, the behavior called "Bounce Off Container Walls."

Reverse Direction
The next modifier in the behavior Bounce Off Container Walls is a Miniscript called "Reverse Direction." It responds to the author message "Reverse Direction" by doing the calculations necessary to reverse the direction of the butterfly.

The final behavior within the Bug behavior is named "Squishability." This behavior contains the programming necessary to simulate the "squishing" of the bug.

Squishability Behavior
The first two modifiers in this behavior are variables. They are used to set the position of the element when it is squished, and to set the degree to which it is squished.

squishPos
This variable contains the position of the bug on the screen when it was squished.

squishFactor
Next in the behavior is a floating point variable named "squishFactor." Its value affects the amount of the element's squish.

Initialization
The next modifier is a Miniscript called "Initialization." It restores the size of the element to its proper size when the scene starts.

Squish Messenger
The messenger named "Squish" sends the author message "Squish" on receipt of a Mouse Down message.

Squish Miniscript
Next is a Miniscript modifier named "Squish." Its script takes care of changing the size of the element. Figure 8.22 shows the contents of the Squish Miniscript.

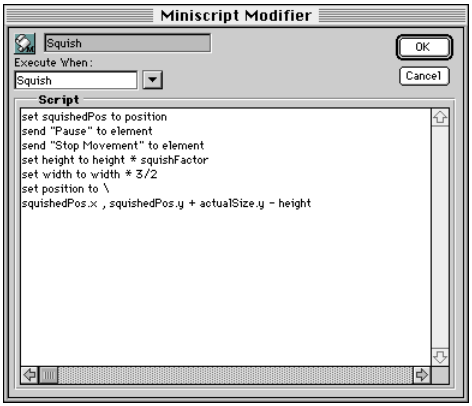


Figure 8.22 The Squish Miniscript modifier

The last modifier in the Squish behavior is a sound modifier named "Squish." It simply plays a sound on receipt of the "Squish" author message sent by the Squish messenger when the user clicks on the bug animation.

- *Note: Using a messenger to send the "Squish" author message allows the author to easily change the message that squishes the targeted object. Programming the Squish behavior in this way also allows different kinds of modifiers to send the message as the result of different activities. For example, a collision messenger placed on the bug element could be configured to send the "Squish" author message. This message would be received by the Squish sound modifier and the Squish Miniscript when the bug collides with another element.*

Blue Butterfly.toon

The programming for the Butterfly is the same as that of the Bug except that it contains an Evasion behavior rather than a Squishability behavior.

Evasion Behavior

The Evasion behavior increases the motion speed of the element when the mouse cursor is over the element. It contains a variable and two Miniscript modifiers.

evasionSpeed

This integer variable stores the magnitude for the accelerated speed of the element's motion.

Evade

The Miniscript called "Evade" sets the magnitude of the motion variable to the value of the evasionSpeed variable on receipt of the Mouse Over message.

Out of Danger

This Miniscript sets the magnitude of the motion variable back to its original speed (using the normalSpeed variable).

CHARACTER INTERACTION

The Character Interaction example features a user-controlled animation along pre-set motion paths in "2.5D" space. Also featured are mTropolis' digital audio playback capabilities.

Description

In this example, the user can control the hopping behavior of a frog in a swamp. The frog can move in one of four directions. Pressing one of four numeric keys (1, 3, 7, and 9) determines which direction the frog jumps. Three butterflies move about the screen, and the frog will eat them if he gets close enough.

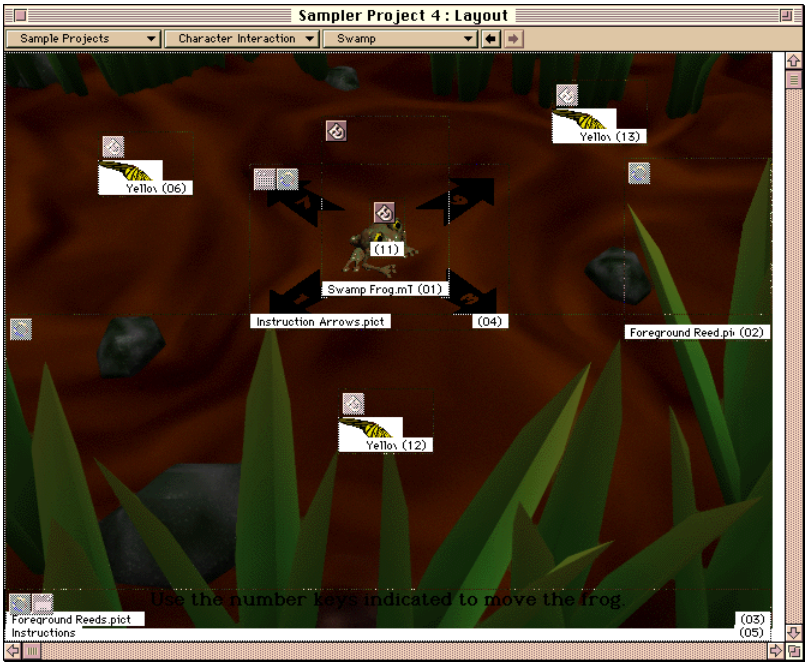


Figure 8.23 The Character Interaction "Swamp" scene in edit mode

Creating the Illusion of 3D Space

The graphic elements of the project are placed in a specific layer order to create the illusion of three-dimensional space.

The Swamp Background.pict linked to the scene is placed at the bottom of the layer order, behind all other graphic elements in the scene. As it is to appear on the background but behind the reeds, the Swamp Frog.toon frog is next in the layer order. To enhance the illusion of depth, the background of the reed graphic elements has been made transparent, allowing both the Swamp Frog and the Swamp Background to appear through the reeds.

To make the butterflies appear in the foreground fluttering above the frog and reeds, they have been given layer order numbers greater than all other graphic elements in the scene. The reed graphics are in two sections, at the bottom and the left of the scene.

Since mTropolis' default transparent color (white) appears on the frog's skin, the frog was rendered on a blue background. This color was then chosen as the transparent color for the mToon.

The Frog Animation

The programming of the frog behavior on the Swamp Frog.toon has been modularized into three other behaviors, called "Keyboard Control," "Initialization," and "Motion." The encapsulation of the frog's functionality into separate parts makes the frog behavior easily changeable. For example, "Keyboard Control" could be easily replaced with a behavior that allows the frog to be controlled with mouse actions rather than keyboard keys.

Each of the three behaviors in "Frog Behavior" will be discussed in turn.

Initialization Behavior

This behavior sets certain attributes when the scene starts. The messenger called "Preload" uses the *Preload Media* command to load the Swamp Frog.toon into memory for maximum performance.

A Start Point point variable stores the frog's screen position at scene start. Two messengers, called "Set Start Position" and "Set End Position" return the frog to this position when the scene starts and ends. This ensures that a frog that has jumped off the screen is restored to its original position when the project is run again.

Keyboard Control Behavior

This behavior contains modifiers that send motion messages to the Swamp Frog.toon element. The "Any Key Cancel Path Keyboard" messenger cancels the frog in the middle of a jump when any key is pressed, allowing the user to change the frog's direction in mid flight.

The other four keyboard messengers send directional author messages ("Up Right," "Up Left," "Down Right," "Down Left") to the Swamp Frog.Toon.n.t. element as the result of a numeric key press (9, 7, 3 and 1 respectively).

Motion Behavior

This behavior contains the modifiers that make the frog jump. Each modifier listens for the author messages that result from a user's key press and responds by showing specific cels in the frog animation. In this way, the frog is moved along the points that make up various motion paths.

The motion is canceled when the modifier receives a "Cancel Path" author message, discussed earlier.

Detection of the Butterflies

Using the Parent/Child Tool, an invisible element called "Sensor" has been attached to (made a child of) the frog element. This element has been configured with a behavior called "Hunting," which allows it to detect the butterfly elements. Note that the element Sensor is smaller than the Swamp Frog.toon element to which it is attached, creating a more specific area of butterfly detection.

Within the behavior called "Hunting" is a collision messenger that sends an "ID Request" author message to the elements that are detected by the Sensor element. The butterfly elements are each programmed to respond to an "ID Request" author message by sending the author message "Fly ID." This message is broadcast, and when it is received by the Sensor element, the first messenger in the Hunting

behavior sends out a "Get Eaten" author message, and the second messenger commands the "Croak.aiff" sound to play. When received by the butterfly element, "Get Eaten" causes the butterfly to disappear from the screen.

To ensure that the butterfly that originates the "Fly ID" author message is sent the message "Get Eaten," Source's Parent is chosen as the destination for the Eat Fly messenger in the frog's Hunting behavior.

Note

• When using the collision messenger to create action/reaction sequences between elements use Source's Parent as the recipient of the message to be sent.

The Contents of the Butterfly Behavior
Each of the butterflies in this project uses an aliased behavior called "Butterfly." The behavior is modularized into two logical parts that are encapsulated into separate behaviors called "Random Motion" and "Life." The contents of each behavior are described below.

Random Motion
The "Random Motion" behavior in the Butterfly behavior contains the programming necessary to move the butterfly around the screen on random motion paths. The first five modifiers in this behavior are variables whose values are accessed at different times by other modifiers in the behavior.

changeOdds
This integer variable defines the odds of the butterfly changing direction.

directions
This integer variable specifies the number of possible directions (out of 360 degrees) in which the butterfly may move.

direction
This integer variable contains the direction (in degrees) in which the bug currently moves.

normalSpeed
This integer variable contains the value that defines the normal speed of the butterfly's movement.

motion
This vector variable contains the angle and magnitude of motion of the butterfly.

Initialization
The Miniscript modifier called "Initialization" initializes the motion of the butterfly when the project begins.

Move
This vector motion modifier is used in conjunction with the motion vector variable to specify the butterfly's motion.

The next modifier in the behavior is a timer messenger called "Change Direction Timer."

Change Direction Timer
This messenger sends a "Change Direction Test" author message to the element every 0.80 seconds.

Change Direction Test
To determine when to change the direction of the butterfly's movement on receipt of Change Direction Test, the Miniscript modifier called "Change Direction Test" uses the rnd

(random) math function with the changeOdds integer variable. If the evaluated expression equals 1, the author message "Change Movement" is sent out, causing the Miniscript called "Change Movement" to set new motion and direction variable values.

The last item within the Random Motion behavior is another behavior called "Bounce Off Container Walls."

Bounce Off Container Walls Behavior

This behavior contains a boundary detection messenger and a Miniscript modifier used to change the direction of the butterfly if it hits the walls of its parent container (i.e., in this case, the edges of the scene).

Bounce Off Container Walls Messenger

The boundary detection messenger in the behavior (also called "Bounce Off Container Walls") sends a "Reverse Direction" author message to the messenger's parent, the behavior called "Bounce Off Container Walls."

Reverse Direction

Next in the behavior Bounce Off Container Walls is a Miniscript called "Reverse Direction." It responds to the author message "Reverse Direction" by doing the calculations necessary to reverse the direction of the butterfly.

Life

The next behavior within the behavior called "Butterfly" is called "Life." It contains two messengers that communicate with the frog animation. The first messenger called "ID" functions simply by sending a "Fly ID" author message whenever it receives an "ID Request" author message.

Get Eaten

The next messenger is named "Get Eaten" and it sends a command to the butterfly to *Stop* on receipt of the "Get Eaten" author message.

Transparent Ink (White BG)

The last modifier in the "Butterfly" behavior is named "Transparent Ink (White BG)." It is used to make the white background of an element transparent. As it is used on other butterflies in the project this graphic modifier has been aliased.

Adding the Swamp Sound

The ambient sound in this project is created by looping the playback of an AIFF sound file. The "Frogs Background.aiff" has been placed on the scene and configured to loop by checking the Loop check box in its element info dialog.

Preload

An aliased messenger named "Preload" has been placed on the Sound element. It simply sends a *Preload Media* command to the element on Scene Started.

- *Note: Sound elements and their modifiers are not visible in the layout view. Switch to the structure view to see the sound element and its messenger in the Swamp Background scene.*

Index