developer GUIDE•

version 1.1

mTROPOLiS

Contents

Chapter 1. Introduction and Documentation Roadmap
Documentation Roadmap 1.8
  Exploring mTropolis 1.9
Conventions Used in this Manual 1.9

Chapter 2. mTropolis Interface
Object Manipulation 2.11
Tool and Modifier Palettes 2.12
mTropolis Editing Views 2.13
Libraries 2.14
Debugging Support 2.14

Chapter 3. Object-Oriented Design
Objects and Messaging: a Definition 3.17
  Design Example: Objects vs. Procedures 3.18

Chapter 4. mTropolis Basics
Overview: mTropolis Objects at Work 4.23
Elements and Modifiers: Building “Media Objects” 4.23
  Elements: Creating Media Objects 4.24
  Modifiers: Customizing Media Objects 4.25
Messaging and User Interaction 4.27
  The Conversational Model 4.27
  Messaging Basics 4.27
  Benefits of the Messaging System 4.28
Structure in mTropolis: a Hierarchy 4.30
  The mTropolis Containment Hierarchy 4.30
  Messaging and the Containment Hierarchy 4.32

Chapter 5. mTropolis Components
The Element Component: Putting it in Context 5.35

Elements and External Media 5.35
Elements and the Containment Hierarchy 5.36
Project Components 5.36
Section Components 5.36
Subsection Components 5.36
Shared Scene Components 5.36
Scene Components 5.37
Element Components 5.37
How Graphic Components are Drawn 5.38
Modifiers 5.39
Behavior Modifiers 5.39
Aliases 5.40

Chapter 6. Messaging

Activating Elements and Modifiers 6.41
Messenger Modifiers: Building Logic 6.41
"When" and "What" Message Pop-Ups 6.43
"Where" Destination Pop-Up 6.43
Relative Message Targeting 6.45
Behavior Containment Hierarchy and Messages 6.45
"With" Menu: Sending Data 6.47
Types of Messages 6.47
Author Messages and Environment Messages 6.48
Commands: Control Signals 6.48

Chapter 7. QuickStart Tutorial—A Simple Slideshow

How it Works 7.51
What You'll Need 7.51
Starting Out 7.51
Create the Next Scene 7.52
Create the Last Two Scenes 7.53
Save Your Project 7.55
Run Your Project 7.55
Add an Element to the Shared Scene 7.56
What's a Shared Scene? 7.56
Navigate to the Untitled Shared Scene 7.56
Add a New Graphic Element 7.56
Run the Project 7.58

Modify the Appearance of the Arrow Element 7.58
    Observe the Effect 7.59
Program the Arrow to Trigger a Scene Change 7.59
    Run the Project 7.60
Add a Back Button 7.61
    Run the Project 7.61
Add Scene Transition Effects 7.62
    Copy the Scene Transition Modifier to Each Scene 7.62
    Run the Project 7.63
Troubleshooting 7.63
The Advanced Slideshow 7.63
More Tutorials 7.63

Chapter 8. In-Depth Tutorial—mPuzzle
What You'll Need 8.65
    Tutorial Project Description 8.65
Start a New Project 8.66
Create the First Scene 8.66
    Adding the Background Image 8.66
    Using a Custom Color Palette 8.67
    Add the Logo Movie 8.68
    Position the Movie 8.68
    Test the Project 8.69
    Changing an Element's Properties 8.69
    Saving the Project 8.69
    Using a Modifier to Change the Scene 8.70
    Create a New Scene 8.71
    Adding a Scene Transition 8.72
Programming the Second Scene 8.73
    Adding a Puzzle Piece to the Scene 8.73
    Programming the First Puzzle Piece 8.74
    Keeping Track of the Puzzle Pieces in Place 8.85
    Animating the M Machine 8.89
    Adding and Programming the Other Puzzle Pieces 8.91
Naming Structural Elements 8.92
Adding Sound 8.93
The Credits Scene 8.95
    Using Libraries 8.101

Chapter 9. Learning mTropolis Project
Using the Learning mTropolis Project 9.105
The Main Menu 9.105

Index

Chapter 1. Introduction and Documentation Roadmap

Welcome to mTropolis. mTropolis is a truly object-oriented multimedia application development environment that produces stand-alone, platform-optimized titles. mTropolis helps you build better titles, faster, propelled by better collaboration between you and your coworkers, and 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.

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. The mTropolis development environment itself runs on Macintosh or Power Macintosh (native) machines.

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 writing a line of script or code. Programmers can work through complex problems in a framework designed to manage complexity without imposing development metaphors. Complex systems, such as sophisticated collision detection systems, can be created without writing 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 *mTropolis 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. This guide also provides a conceptual overview of the mTropolis framework. This guide focuses 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 that they have a fundamental understanding of the development process.

Detailed descriptions of mTropolis menu options, palettes, and programming features can be found in the *mTropolis Reference Guide*.

DOCUMENTATION ROADMAP

Because mTropolis is an interactive tool, instructional material is provided in both traditional print and an interactive format. The mTropolis documentation set consists of the following components.

mTropolis Quick Reference
This eight page guide contains lists of keyboard shortcuts, Miniscript commands, element attributes, and pictures of mTropolis palettes.

mTropolis Developer Guide
This guide introduces basic mTropolis concepts and object-oriented programming. It also contains both simple and in-depth tutorials, and descriptions of mTropolis authoring examples.

mTropolis Reference Guide
This guide contains detailed descriptions of everything you want to know about mTropolis functionality.

Learning mTropolis Project
Use this mTropolis project to learn about multimedia basics, view simple projects, and see authoring walkthroughs. The "Learning mTropolis" project can be found in the "Documentation" folder on the mTropolis CD-ROM.

QuickStart Tutorial
Complete this simple tutorial to orient yourself to the mTropolis environment. Refer to Chapter 7, "QuickStart Tutorial—A Simple Slideshow" in the *mTropolis Developer Guide*.

In-Depth Tutorial
Complete this advanced tutorial to fully familiarize yourself with the mTropolis environment. Refer to Chapter 8, "In-Depth Tutorial—mPuzzle" in the *mTropolis Developer Guide*.

MOM Reference Guide

Would you like to customize the mTropolis environment? Learn about the mFactory Object Model using the *MOM Reference Guide*. Refer to the “About MOM” document in the “mFactory Object Model” folder on the mTropolis CD-ROM for more information.

Exploring mTropolis

mTropolis’ instructional material provides various levels of guidance. The following sections suggest paths through the instructional material, depending on the level of guidance you would like.

Lots of Guidance

- Read Chapter 2, “mTropolis Interface”, Chapter 3, “Object-Oriented Design”, Chapter 4, “mTropolis Basics”, Chapter 5, “mTropolis Components” and Chapter 6, “Messaging” in the *mTropolis Developer Guide*. Time: about 1 hour.
- Complete the tutorial in Chapter 7, “QuickStart Tutorial—A Simple Slideshow” in the *mTropolis Developer Guide*. Time: about 30 minutes.
- View the “Authoring Walkthroughs” in the “Learning mTropolis” project, found in the Documentation folder of the mTropolis CD-ROM. Time: 30 minutes.

Less Guidance

- Complete the tutorial in Chapter 8, “In-Depth Tutorial—mPuzzle” in the *mTropolis Developer Guide*. Time: 4-6 hours.

  It is recommended to complete the tutorial in segments, rather than in a single sitting.

- Look at “Modifier Examples” in the “Learning mTropolis” project, found in the Documentation folder of the mTropolis CD-ROM. Time: 1 to 2 hours.

  Consult Chapter 12, “Modifier Reference” in the mTropolis Reference Guide for information on specific modifiers.

- Look at “Multimedia Basics” in the “Learning mTropolis” project. Time: about 1 hour.
- Look at “Authoring Examples” in the “Learning mTropolis” project, and read Chapter 9, “Learning mTropolis Project” in the *mTropolis Developer Guide*. Time: about 1 hour.

As You Need It

- Refer to the *mTropolis Reference Guide* as you need information about different aspects of mTropolis, including menus, views, palettes, modifiers, messages and commands, Miniscript, attributes, and MovieTrax.
- Read the “About MOM” document in the “mFactory Object Model” folder on the mTropolis CD-ROM for information about accessing MOM documentation and examples.

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 "⌘".

Hot Tip

The name mTropolis is pronounced like the word "metropolis". The name mFactory is pronounced "em factory".

Chapter 2. mTropolis Interface

This chapter provides an overview of the mTropolis interface. When you start mTropolis, a window appears (the *Layout* window) with pulldown 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. You can work on multiple projects simultaneously. Each project has its own layout window like the one described above.

OBJECT MANIPULATION

Every action in mTropolis, except very specialized, global operations (such as creating a standalone title version of your project) can

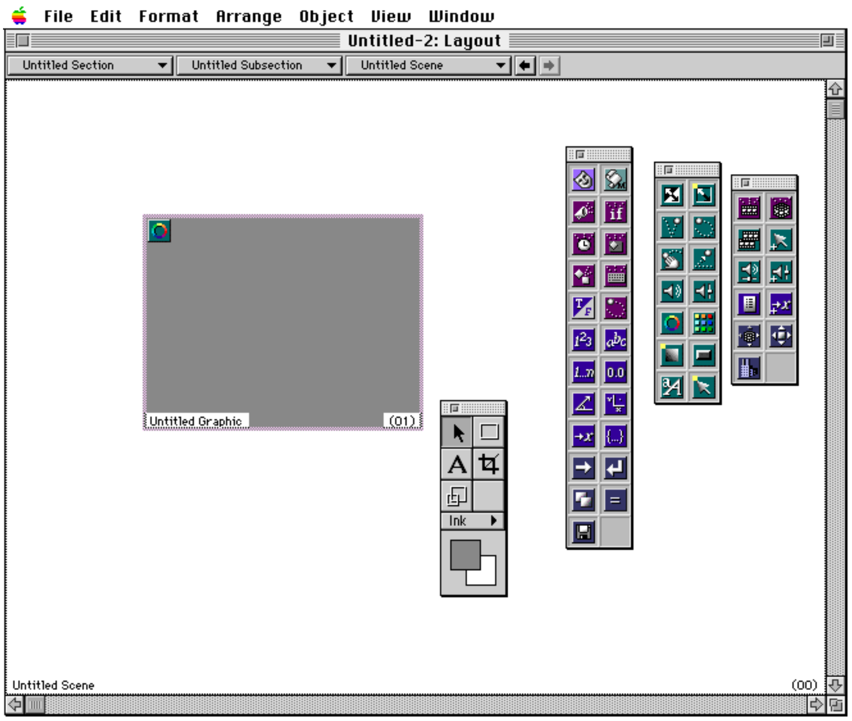


Figure 2.1 The layout window with tool and modifier palettes

be accomplished with direct manipulation. For example, 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 pulldown 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 also available through direct manipulation.

mTropolis has been designed with careful attention to user interface consistency. If an object on the 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 the screen can be double-clicked and it will, at the very least, display 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

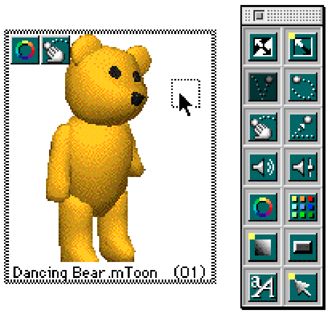


Figure 2.2 Modifiers can be dragged and dropped onto elements

objects that you can drag and drop in the process of building a title (Figure 2.2). The tool palette offers the tools most frequently used in building and manipulating the constituent pieces of a mTropolis title.

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. More information on the various mTropolis palettes can be found in Chapter 11 of the *mTropolis Reference Guide*, “Palette Reference”.

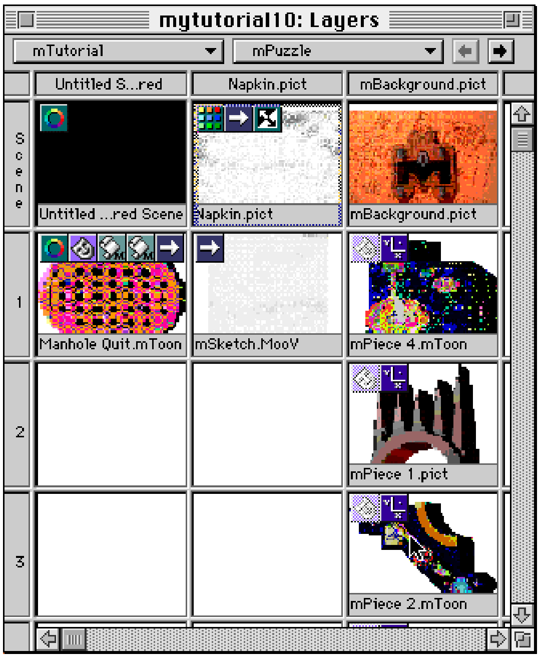


Figure 2.3 The Layers view

mTROPOLIS EDITING VIEWS

mTropolis offers three primary views of your work, each of which allows you to edit and manage your project in different ways. The 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 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 shows the hierarchy of components within your project. This view shows which objects are contained within others and allows you to rearrange this

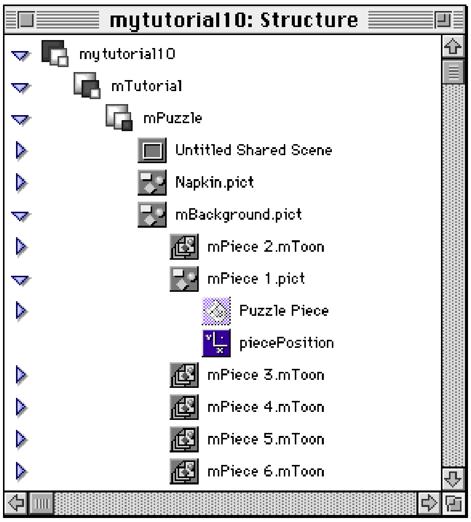


Figure 2.4 The Structure view

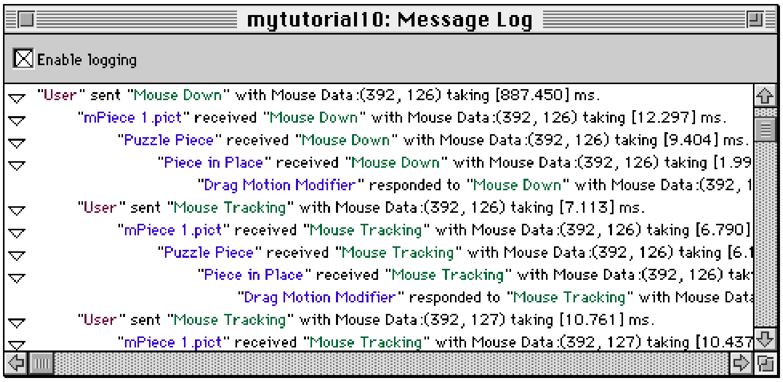
hierarchy at will by dragging and dropping. The Structure view is described in detail in Chapter 8 of the *mTropolis Reference Guide,* “Structure Window”.

LIBRARIES

mTropolis offers a very powerful facility for managing large projects, enabling collaboration 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 “New, Open, and Save for mTropolis Libraries” on page 2.23 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 7.73 of the *mTropolis Reference Guide*.



```
"User" sent "Mouse Down" with Mouse Data:(392, 126) taking [887.450] ms.
    "mPiece 1.pict" received "Mouse Down" with Mouse Data:(392, 126) taking [12.297] ms.
        "Puzzle Piece" received "Mouse Down" with Mouse Data:(392, 126) taking [9.404] ms.
            "Piece in Place" received "Mouse Down" with Mouse Data:(392, 126) taking [1.99
                "Drag Motion Modifier" responded to "Mouse Down" with Mouse Data:(392, 1
    "User" sent "Mouse Tracking" with Mouse Data:(392, 126) taking [7.113] ms.
        "mPiece 1.pict" received "Mouse Tracking" with Mouse Data:(392, 126) taking [6.790]
            "Puzzle Piece" received "Mouse Tracking" with Mouse Data:(392, 126) taking [6.1
                "Piece in Place" received "Mouse Tracking" with Mouse Data:(392, 126) tak
                    "Drag Motion Modifier" responded to "Mouse Tracking" with Mouse Data
    "User" sent "Mouse Tracking" with Mouse Data:(392, 127) taking [10.761] ms.
        "mPiece 1.pict" received "Mouse Tracking" with Mouse Data:(392, 127) taking [10.437
```

Figure 2.5 The Message Log window

Debugging Support

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 3.20.

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 almost no one mistakes them for each other. Software objects act literally as the software counterparts of real-world objects. They model the real-world objects and interactions between those 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. For example, 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 virtual world modeled by 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. For example, 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 stored 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, 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.

Extending the mTropolis Environment

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 sophisticated 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 the online documentation for MOM for additional information.

Objects and Messaging: a Definition

Chapter 4. mTropolis Basics

In this chapter, mTropolis' implementation of object-oriented 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 required 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 tools and modifiers for creating software objects and a messaging system for connecting those objects together. This system allows the author to focus on visually laying out the project and defining the messages that will bind the project's components together without having to "reinvent the wheel" every time a simple operation, such as 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. Components can be combined 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 a new component in mTropolis is as simple 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 a mTropolis project in the editing environment and a stand-alone title made from that project 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*. Elements are essentially containers for media and have built-in code for maintaining their basic state (e.g., their position, size, layer, etc.). Modifiers are programming components that can be added to elements to endow them with new capabilities (e.g., collision detection). Both elements and modifiers can be configured via dialog boxes.

Combining modifiers with elements, configuring their operation, and creating conversations between these modified elements is at the heart of the authoring process in mTropolis. Once created, media elements can be freely shared and reused just like any other mTropolis component.

mTropolis projects 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 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 "Run—Switching Between Edit and runtime Modes" on page 2.36 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. External media types supported by mTropolis include simple bitmaps, animations, and time-based media such as digital video and audio files.

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 QuickTime digital video file.

In the editing environment, elements display thumbnail images of the external media files to which they are linked. For example, in Figure 4.1 the first frame of a digital video is

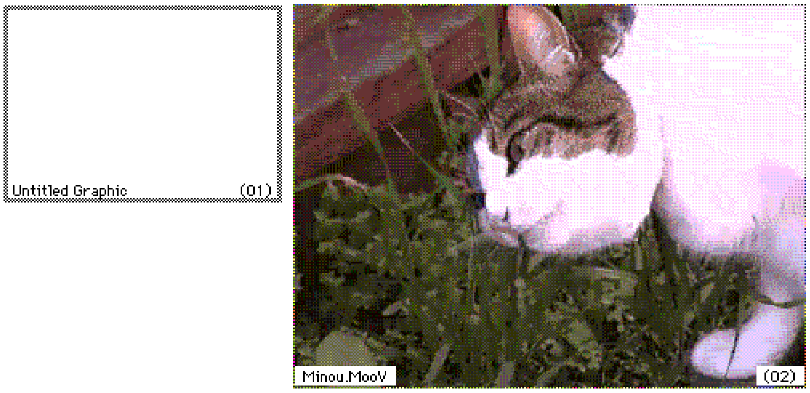


Figure 4.1 A new graphic element, and a graphic element linked to a QuickTime movie

shown within the boundaries of the element. When the author switches from editing mode to runtime 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 can be displayed by double-clicking on the element.

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

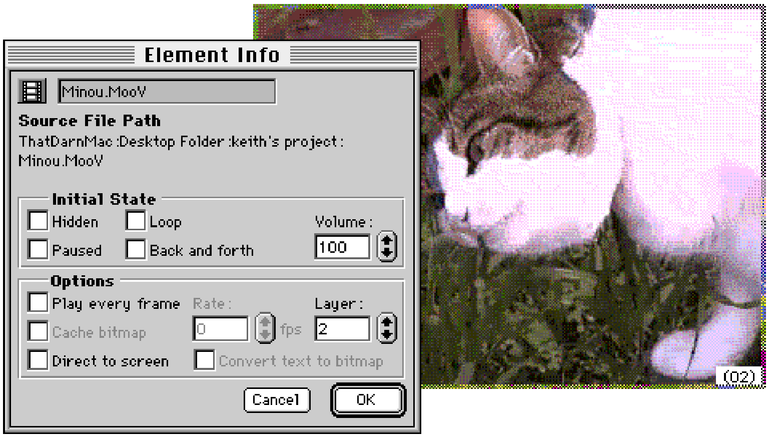
The element's configuration does not directly alter the external media file. Instead, it alters the appearance and behavior of the media at runtime, such as its representation on the screen, or its volume level. The media file linked to the element remains unchanged.

Three types of elements can be created in the mTropolis editor:

- Graphic elements, which can be linked to still bitmaps, animations, or digital video.
- Sound elements, which can be linked to digital audio files.
- Text elements, which can contain text entered in the editing environment or entered by users at runtime.

Modifiers: Customizing Media Objects

Element components can inherit capabilities from modifiers. In the mTropolis editing 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).



Minou.MooV

Source File Path
ThatDarnMac:Desktop Folder:keith's project:
Minou.MooV

Initial State
☐ Hidden ☐ Loop Volume: 100
☐ Paused ☐ Back and forth

Options
☐ Play every frame Rate: 0 fps Layer: 2
☐ Cache bitmap
☐ Direct to screen ☐ Convert text to bitmap

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

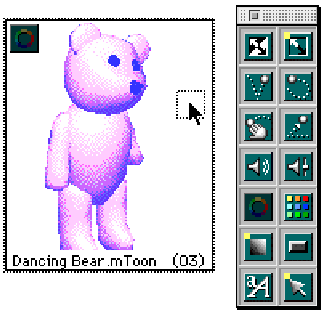
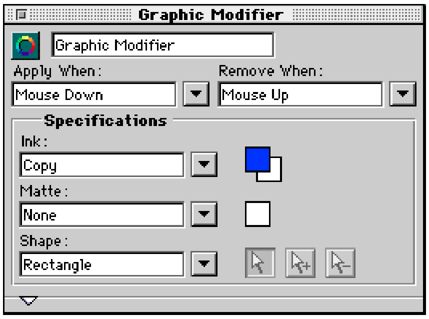


Figure 4.3 Dragging a modifier from a palette and dropping it on a graphic element.



Graphic Modifier

Apply When: Mouse Down

Remove When: Mouse Up

Specifications

Ink: Copy

Matte: None

Shape: Rectangle

Figure 4.4 The Graphic Modifier configuration dialog.

When an element inherits new capabilities or properties from modifiers its media is not permanently altered. For example, an element containing an image of a yellow bear can contain a modifier with a special ink effect that causes the bear to appear blue. The appearance of the *element* is changed, but the external *media* linked to that remains unchanged.

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".

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—anywhere within a mTropolis project or even in entirely new projects.

MESSAGING AND USER INTERACTION

Messaging is the glue that binds an object-oriented system together. As a fully fledged object-oriented development environment, mTropolis is no different. As you craft your title using mTropolis components, you program them to interact using 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 internally consistent but also 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 unpredictability because the content and course of the conversation is not predetermined. Thus, the conversational model truly reflects the real world: every conversation, whether it’s about particle physics or 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.

Messaging Basics

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 display its configuration dialog. Sending a message to a mTropolis component is an alternate, dynamic vehicle for invoking these capabilities during runtime. For example, the graphic modifier component shown in Figure 4.5 is configured to lighten its element's media 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. If a component receives a message that it is not configured to respond to, it merely relays the message to the components that it contains. For example, an element might contain the graphic modifier shown in Figure 4.5. When the element receives a "The light is off" message, it passes 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.

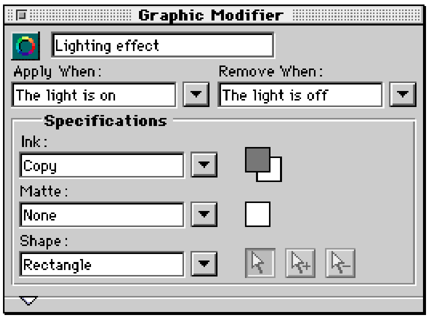

Graphic Modifier
Lighting effect
Apply When: The light is on
Remove When: The light is off
Specifications
Ink: Copy
Matte: None
Shape: Rectangle

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

Messages can be generated by the user during runtime (e.g., user mouse actions such as mouse clicks generate messages), the mTropolis environment (e.g., mTropolis sends a Scene Started message to components at the start of each scene), and by components themselves (e.g., the Collision Messenger sends a message when it detects a collision between elements). Regardless of the source of a message, any 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 provides very powerful but uncomplicated facilities for controlling the scope and flow of message conversations. For example, a message can be sent to every 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 with customized names. These *author messages* can be detect or sent just like any built-in message.

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 an “I’m Popped!” message. To effectively reuse the balloon component in any new title, someone unfamiliar with the balloon component would only have to understand these two 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 and 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. For example, a container ship can be the parent of the shipping 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

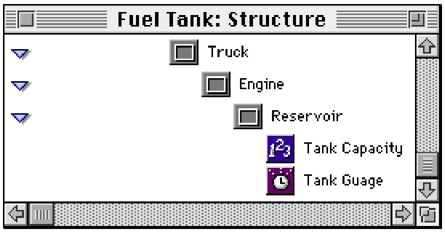


Figure 4.6 A sample Structure View

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.

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 cooperating modifiers into a single location. 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 behavior is that they can contain other behaviors within them, and those child behaviors can be parents to of other behaviors, 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 household 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.

Messaging and the Containment Hierarchy

The containment hierarchy fulfills an important function in addition 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 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 variable modifiers those constants at the project level (i.e., the top-most level of the containment hierarchy). Thus, the project would contain two modifiers as well as sections representing different experiments or rooms in a virtual physics lab.

The physics tutorials in this title could 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." An author message called "Turn off Gravity" can be sent to an entire section of the project. mTropolis automatically "cascades" and "relays" the message from the section level on down through the entire containment hierarchy. Any component, contained anywhere in the hierarchy, capable of responding to the "Turn off Gravity" message would do its stuff.Figure 4.7 shows the path of a message sent to a section of a project.

The advantage of broadcasting is that you can cause global changes without laboriously specifying each and every component that

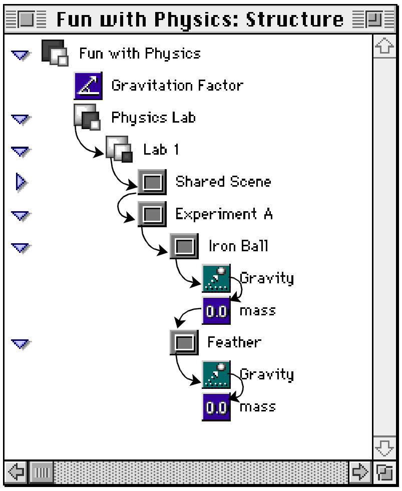


Figure 4.7 Message passing between components. A message sent to the "Physics Lab" section cascades and relays from component to component.

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 single scene of the project. The message would be sent only to the scene, then cascade to the elements and modifiers in that scene.

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 containers 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 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.

Structure in mTropolis: a Hierarchy

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 (e.g., which frame of an animation is currently displayed).

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 format sound files).

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

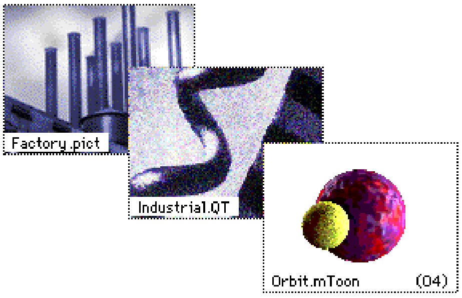


Figure 5.1 Graphic elements linked to a PICT, QuickTime movie, and mToon.

editable so that users can modify their contents in runtime.

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.

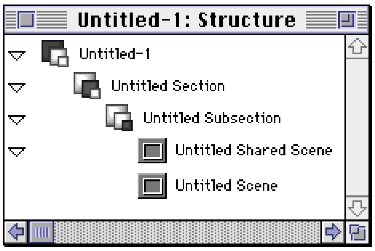


Figure 5.2 Structure view of a new mTropolis project.

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. Elements in a shared scene are visible or audible in any scene in the same 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 might 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 action to an audience, a scene in a mTropolis title contains all the components to do the same for a user. Scenes are the highest-level structural components that are visible to users—everything element that a user can see or manipulate is contained within a scene.

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

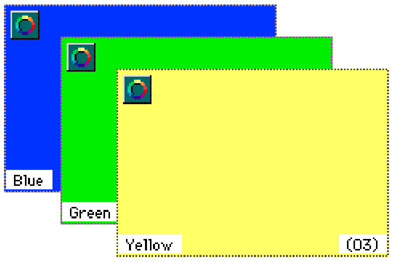


Figure 5.3 The effect of graphic layers

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 creates 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.

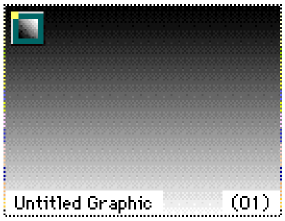


Figure 5.4 The gradient modifier, placed on a graphic element

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".

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.

One of the most powerful features of behaviors lies is 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 complete description of the behavior modifier can be found in “Behavior” on page 12.129 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 11.110 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 hierarchy. Modifiers can also receive messages from the mTropolis runtime environment itself, such as scene change events or user mouse events.

Elements themselves do not send messages, but they can receive certain special messages (called *commands*) 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 applied its graphic effect. When it receives the Mouse Up message, it returns the element to its default color.

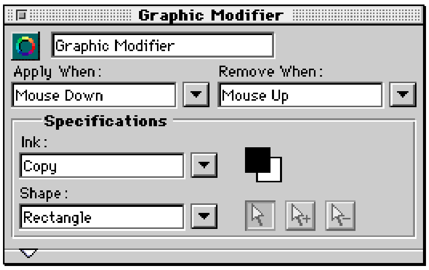


Figure 6.1 A Graphic Modifier dialog configured to activate on Mouse Down and deactivate on Mouse Up messages.

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 and receiving messages. Messengers are the components that implement 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.

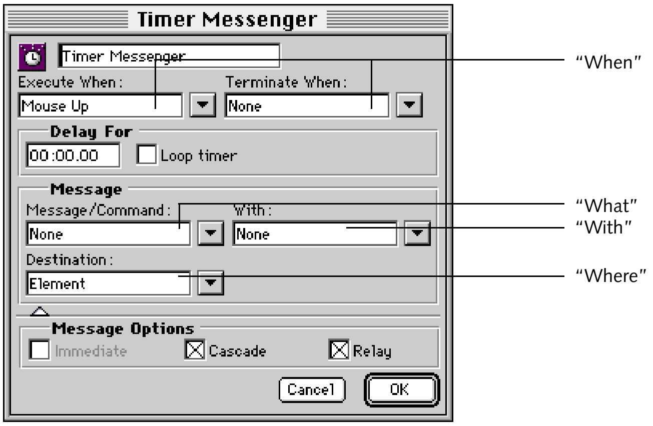
The timer messenger dialog shown below (Figure 6.2) illustrates the full power of messaging in mTropolis. Through their dialogs, messengers can be configured to send specific messages, to specific destinations with any data that the author wants to send and receive. Messaging is a powerful, but simple 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. 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.

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


Timer Messenger

Name: Timer Messenger
Execute When: Mouse Up
Terminate When: None
Delay For: 00:00.00
Loop timer: [ ]
Message
Message/Command: None
With: None
Destination: Element
Message Options
Immediate: [ ]
Cascade: [✓]
Relay: [✓]

Figure 6.2 A typical messenger dialog.

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 built-in, 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 13.225 of the *mTropolis Reference Guide* and "The Message/Command Pop–Up Menu" on page 13.226 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, element, or 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.

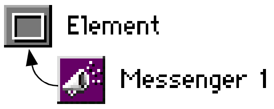
Container Hierarchy and Messages

As explained above, the destination for a messenger's output message can be either a specific component, an arbitrary level of the project, or a behavior containment hierarchy. By default, mTropolis automatically handles cascading the message down through the containment hierarchy to the elements, modifiers, or behaviors that might be listening for the message that was sent (though these defaults can be changed). Remember, we explained the power of the containment hierarchy for controlling the flow of messages in "Messaging and the Containment Hierarchy" on page 4.32. Message destinations are described in detail in "Destination Option Descriptions" on page 13.250 of the *mTropolis Reference Guide*.

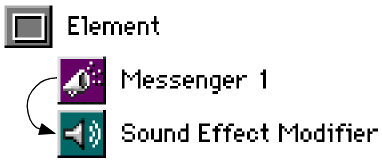
Some examples of possible message destinations in the container hierarchy are described below:

- The messenger modifier sends a message to the element that contains the messenger.

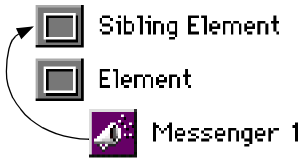
This destination is called the "element" destination:



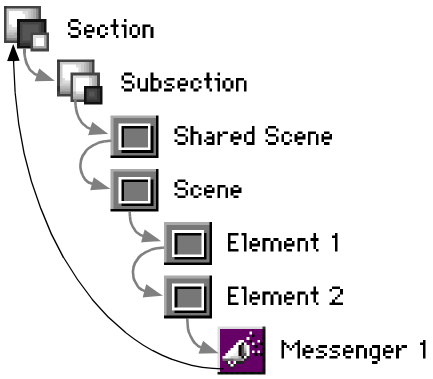
- The messenger sends a message to another modifier contained by the same element (i.e., a sibling of the messenger). This destination is called a "messenger's sibling" destination:



- The messenger sends a message to another element (i.e., to a sibling of its parent). This destination is called an "element's sibling" destination:



The illustration below depicts a messenger sending a message (the black line) to an arbitrary level in the containment hierarchy and how that message cascades down (the grey lines) to all of the eventual recipients.



In the preceding illustration, the messenger contained by "Element 2" sends a message to the Section component. The section is the container (and parent) to the subsection, which is in turn the container and parent to the shared scene and scene which is the container and parent to elements 1 and 2. When the messenger targets its message at the section, that message cascades down to every component in the section's portion of the project 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.

Relative Message Targeting

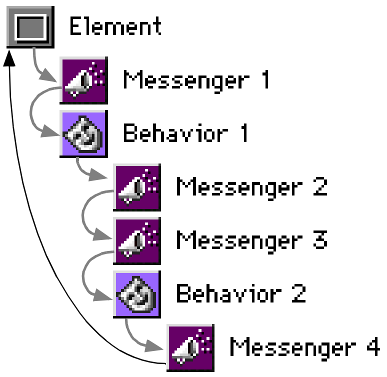
A very powerful feature of messaging in conjunction with the containment hierarchy is relative message targeting. You have seen that you can specify some abstract level of the containment hierarchy as a target, and that mTropolis will handle all of the details of cascading the message to the possible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

The "building blocks" for describing relative message destinations are shown in Table 14.2 on page 14.265 of the *mTropolis Reference Guide*.

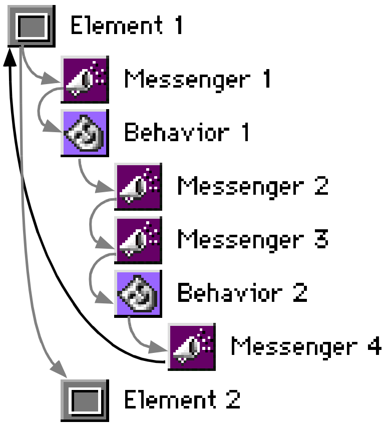
Behavior Containment Hierarchy and Messages

As we mentioned in our discussion of behaviors in "Behavior Modifiers" on page 5.39, behaviors can contain modifiers, and can also 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. The messenger contained in Behavior 2 can directly 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 (Messenger 1), however, 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 Window

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

Notice that the behavior dialog box shows the execution order of modifiers (the (1), (2), and (3) indicators next to the icons) 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 component. The behavior in this example contains the modifiers that would be activated

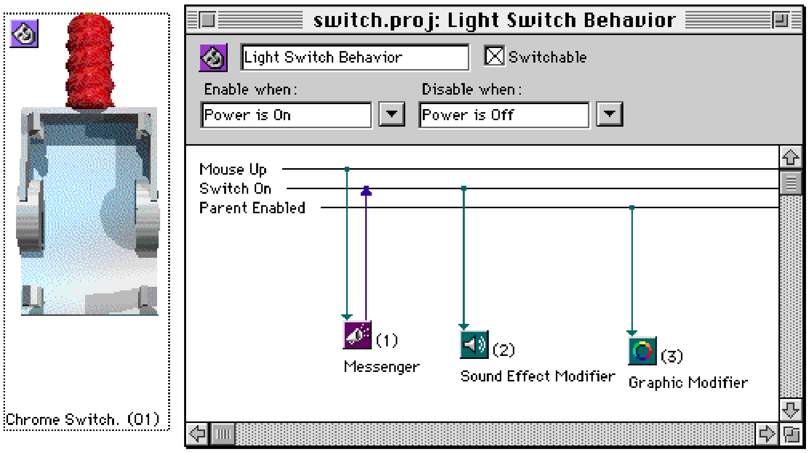


Figure 6.3 A behavior modifier dialog.

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. In this way, behaviors promote easy reusability.

Behaviors, like modifiers, can be activated and deactivated through the receipt of messages. When checked, the "Switchable" checkbox in the behavior window (Figure 6.3) enables a behavior to be controlled in this fashion. The "Enable when" pop-up specifies the message that will enable the behavior and the "Disable when" pop-up specifies the message that will disable the behavior. When a behavior is deactivated, all of the modifiers within that behavior are effectively "deaf" to any messages that arrive.

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.

"With" Menu: Sending Data

The "With" message pop-up allows a messenger to pass data along with the message output 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), a constant value (entered in "With" pop-up menu in appropriate syntax), the value that it received from the message that activated it (the "incoming data"), or the contents of any variable modifier accessible to the messenger.

For example, you might want to send the value stored in a variable called "Light Intensity" with a message called "Light is On".

See "The "With" Pop-Up Menu" on page 13.248 of the *mTropolis Reference Guide* for complete information on this menu.

TYPES OF MESSAGES

Messages in mTropolis can be divided into three types. The first two types, *author messages* (messages created by the author of a project) and the *environment messages* (built-in messages sent by mTropolis), act as signals or conveyors of information. For example, the Mouse Up message signals the recipient that someone clicked the mouse.

The third type of message is called a *command*. Sending a command message constitutes a demand that the recipient perform some action, and it cannot be ignored. Command messages (usually simply called "commands") are primarily used to control the behavior of elements. For example, when an element receives the *Play* command, its media immediately starts to play.

This section describes the various types of messages available for use in mTropolis. More information on these messages can be found in "Environment Messages" on page 13.226 of the *mTropolis Reference Guide*, "Author Messages" on page 13.228 of the *mTropolis Reference Guide*, and "Commands" on page 13.228 of the *mTropolis Reference Guide*.

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.

Author Messages

Author messages are defined by the author by entering the text of a new author message into the "When" pop-up of a modifier. mTropolis asks if you wish to create a new author message.

Author messages are never sent by the mTropolis environment, 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 in 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.

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.

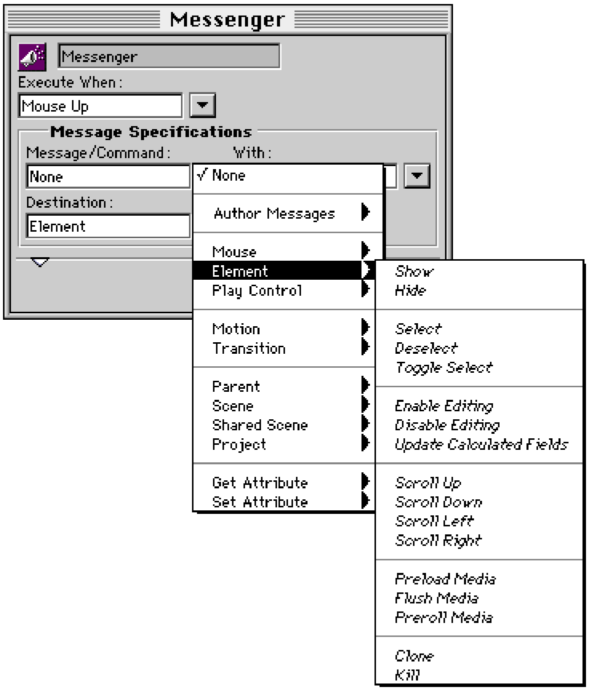
  Since commands act directly on elements, they do not affect modifiers. The author cannot, for example, command a modifier to *Play* or *Pause*. However, modifiers can receive and interpret commands, or pass them on to elements if directed to do so.

- 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 is a message named “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.

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


Messenger

Messenger

Execute When: Mouse Up

Message Specifications

Message/Command: None

With:

Destination: Element

Figure 6.4 Commands in the Message/Command menu’s “Element” section.

Types of Messages

Chapter 7. QuickStart Tutorial—A Simple Slideshow

In this tutorial, we'll create a simple slideshow—four images that the user can flip through like a book.

HOW IT WORKS

This project takes advantage of mTropolis' structural hierarchy. The scenes in a subsection of a mTropolis project can be thought of as a stack of cards, or the pages in a book. Here, we'll create four scenes that each contain one image. Using the scene change modifier, we can cause the scenes to change based on user mouse actions. The end result is a presentation that users can browse at their own pace.

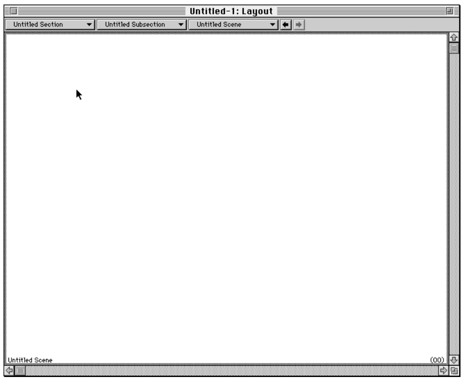
WHAT YOU'LL NEED

- 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. Media files used in this tutorial can be found in the **Tutorials** folder of the mTropolis installation. If the tutorial files have not been installed, they can be installed by running the installer from the mTropolis CD-ROM, or by dragging the **Tutorials** folder from the CD to your hard disk.
- For best performance, make sure your monitor is set to display 256 colors.

STARTING OUT

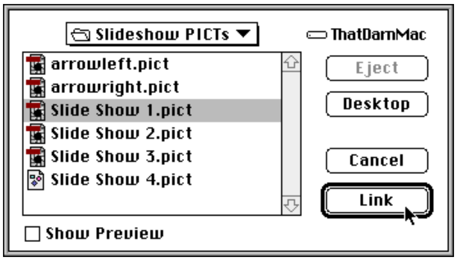
Start mTropolis by double-clicking the mTropolis icon. mTropolis appears with a new, untitled Layout window.

- Click inside the Layout window to select the Untitled Scene.

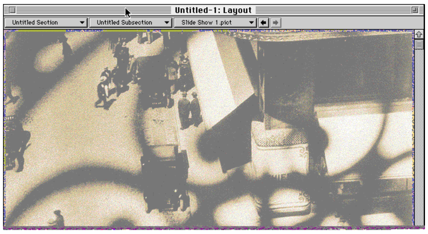


- Select **Link Media-File** from the File menu. A standard file selection dialog appears.
- In the file dialog, select the file **Slide Show 1.pict** from the **Slideshow PICTs** folder, found inside the **QuickStart** folder (located

in the **Tutorials** folder). Click **Link** to link the picture to the scene.



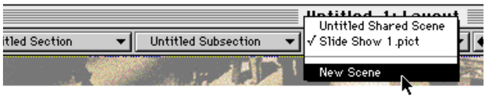
- The Layout window updates to show the new media linked to the scene.



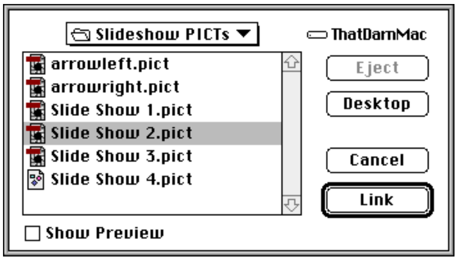
CREATE THE NEXT SCENE

We've imported the media for our first scene. Now we need to create the second scene and link a picture to it.

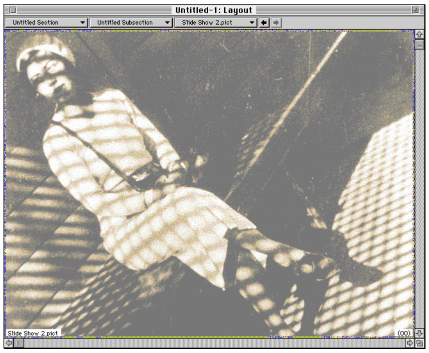
- Create a new scene by selecting **New Scene** from the Layout window's scene pop-up menu. This is the third pop-up menu button from the left, found at the top of the Layout window. Currently, this button reads **Slide Show 1.pict**.



- The Layout window changes to show the new, Untitled Scene. The scene is already selected for us so we can link media to it.
- Select **Link Media-File** from the File menu. A standard file selection dialog appears.
- In the file dialog, select the file **Slide Show 2.pict** from the **Slideshow PICTs** folder, found inside the **QuickStart** folder. Click **Link** to link the picture to the scene.



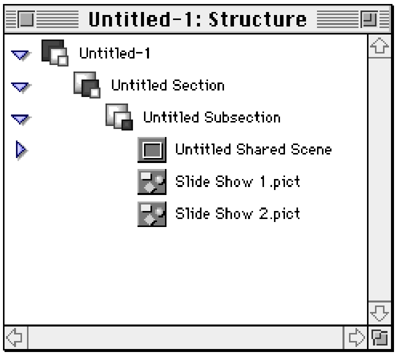
- The Layout window updates to show the new media linked to the scene.



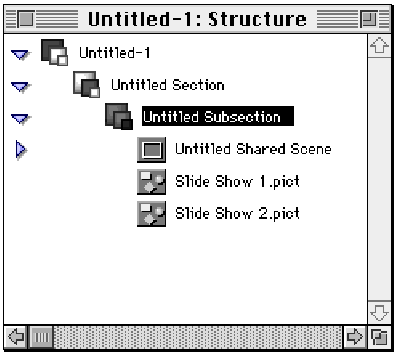
CREATE THE LAST TWO SCENES

We created the second scene of our project using the Layout window's scene pop-up menu. We'll use a different method to create the last two scenes.

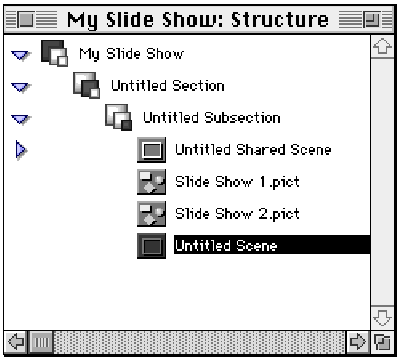
- Select **Structure Window** from the View menu to display mTropolis' Structure window.
- The Structure window for the project appears. This window displays a hierarchical view of the project. Components of the project are shown as named icons. Open/close triangles to the left of those icons allow you to reveal or conceal different levels of the project hierarchy.
- Click the triangle next to **Untitled Section** to reveal the Untitled Subsection.
- Click the triangle next to **Untitled Subsection** to reveal the scenes we are working on.
- The Structure window should now look like the one shown below.



- Click on the icon for the **Untitled Subsection** to select that subsection. The Structure window should look like the one shown below.



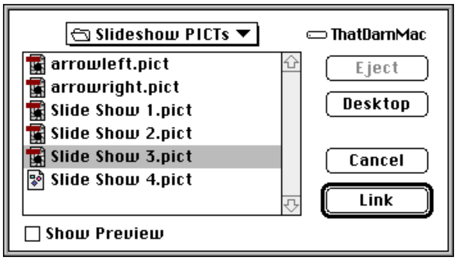
- Select **New-Scene** from the Object menu to create a new, untitled scene. A new scene appears in the Structure window as shown below.



My Slide Show: Structure

- My Slide Show
  - Untitled Section
    - Untitled Subsection
      - Untitled Shared Scene
      - Slide Show 1.pict
      - Slide Show 2.pict
      - Untitled Scene

- The new Untitled Scene is selected and ready to have media linked to it. Select **Link Media-File** from the File menu. A file selection dialog appears.
- In the file dialog, select the file **Slide Show 3.pict** from the **Slideshow PICTs** folder, found inside the **QuickStart** folder. Click **Link** to link the picture to the scene.

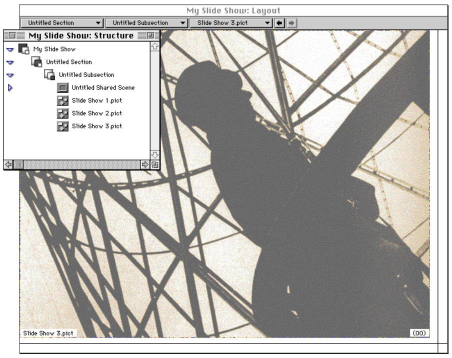


Slideshow PICTs

ThatDarnMac

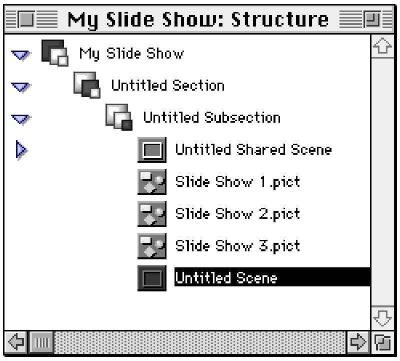
- arrowleft.pict
- arrowright.pict
- Slide Show 1.pict
- Slide Show 2.pict
- Slide Show 3.pict
- Slide Show 4.pict

- Notice that both the Structure and Layout windows update to reflect the new scene as shown below.



Now we're ready to create the fourth and last scene in our project.

- Select **New-Scene** from the Object menu once again. Another new scene appears in the structure window. Your Structure window should look like the one shown below.

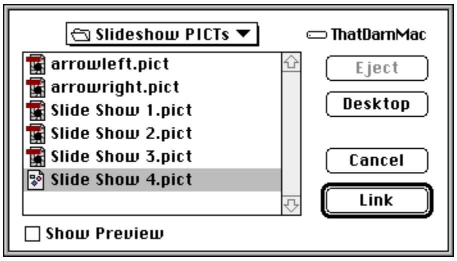


My Slide Show: Structure

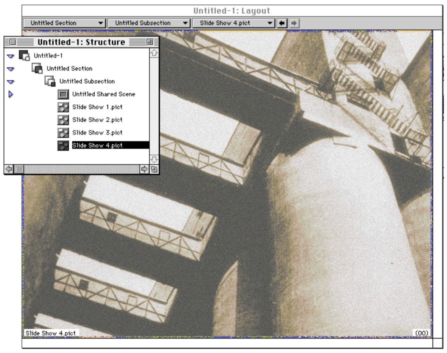
- My Slide Show
  - Untitled Section
    - Untitled Subsection
      - Untitled Shared Scene
      - Slide Show 1.pict
      - Slide Show 2.pict
      - Slide Show 3.pict
      - Untitled Scene

- Select **Link Media-File** from the File menu. A file selection dialog appears.

- In the file dialog, select the file **Slide Show 4.pict** from the **Slideshow PICTs** folder, found inside the **QuickStart** folder. Click **Link** to link the picture to the scene.



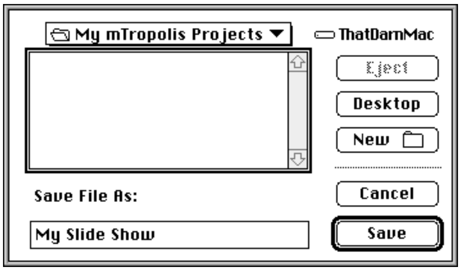
- Again, both the Structure and Layout windows update to reflect the new scene as shown below.



SAVE YOUR PROJECT

Before we continue any further, you should save your work up to this point.

- Select **Save** from the File menu.
- A file selection dialog appears. Choose a location and name for your project, then click the **Save** button.



RUN YOUR PROJECT

So far, we've worked in mTropolis' *edit mode*. To see the project as users would see it after it has been saved as a built title, we can switch to *runtime mode*.

- Press **⌘-T** to switch from edit mode to runtime mode.
- The mTropolis interface disappears and the first scene of the project is displayed.
- Move the cursor around the screen. Click anywhere you like. You'll see that nothing happens. But there's nothing wrong with your project. All of the scenes are in there, but the user has no way to access them. We need to add some *modifiers* to our project to create interactivity.
- Press **⌘-T** again to return to edit mode. The mTropolis editing environment reappears.

ADD AN ELEMENT TO THE SHARED SCENE

Now we'll create an interface that can be used to navigate through the scenes of our project. Eventually, we'll have a "forward" and a "backward" button that users can click to change the picture shown on the screen.

We want these buttons to be visible in every scene of the project. By placing them on the *shared scene,* they will always be visible.

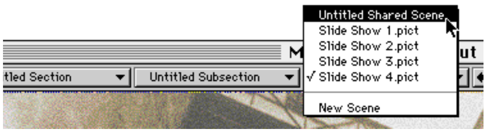
What's a Shared Scene?

When looking at the Structure window, you may have noticed that our subsection contains a component named "Untitled Shared Scene" in addition to the four scenes linked to the slide show images. This component is a special type of scene. Any media placed on the shared scene will be visible in every other scene in the subsection.

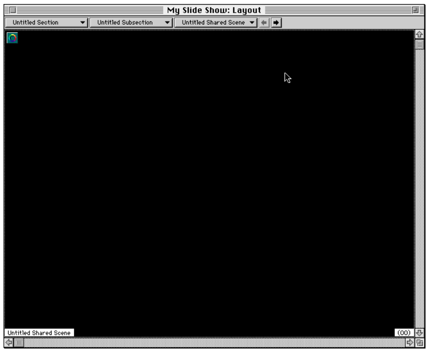
Navigate to the Untitled Shared Scene

Use the Layout window's scene pop-up to navigate to the shared scene.

- Select **Untitled Shared Scene** from the scene pop-up.



- The Layout window updates to show the shared scene, which is currently just a black background as shown below.



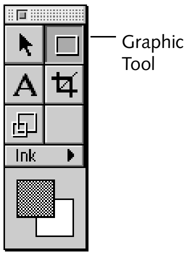
Add a New Graphic Element

Now we'll add a new graphic element to the shared scene. This graphic will eventually be programmed to act as our "forward" button for controlling the slideshow.

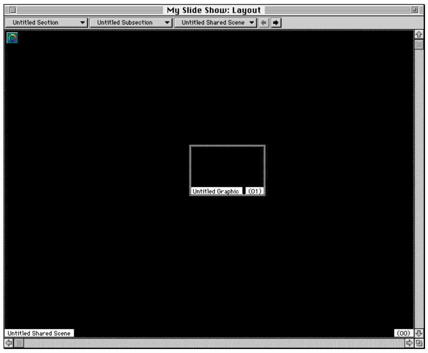
- If the Tool palette (shown below) is not already visible, select **Tool Palette** from the View menu. The Tool palette appears.

- Click on the graphic tool in the tool palette.

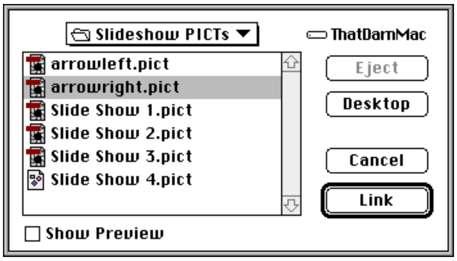
- Drag the cursor somewhere inside the Layout window to create a new graphic element on the shared scene. The



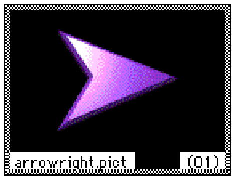
size and position of the element are not important.



- Select **Link Media-File** from the File menu. A file selection dialog appears.
- In the file dialog, select the file **arrowright.pict** from the **Slideshow PICTs** folder, found inside the **QuickStart** folder. Click **Link** to link the picture to the graphic element



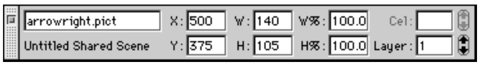
- The element updates and resizes to show its new contents. The element's name changes to "arrowright.pict".



Notice that the element can be dragged to any location in the Layout window to position it within the scene. Elements can also be positioned 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.
- Click on the **arrowright.pict** element to select it.
- The element's name, position, and size information are displayed in the object info palette.
- In the Object Info palette's **X** field, enter **500**. Press the return key to confirm the change. The arrowright.pict element moves to its new location.
- In the Object Info palette's **Y** field, enter **375**. Press the return key to confirm the change. The arrowright.pict element moves to its new location. It should now be in the bottom right corner of the scene. The

Object Info palette should look like the one shown below.

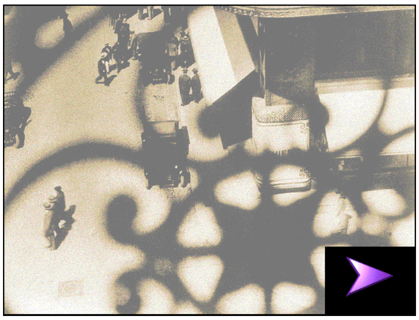


arrowright.pict X: 500 W: 140 W%: 100.0 Cel:
Untitled Shared Scene Y: 375 H: 105 H%: 100.0 Layer: 1

Run the Project

Switch to runtime mode to see the effects of our latest addition.

- Press **⌘-T** to switch to runtime mode. The first scene appears with the arrow picture superimposed on it as shown below.



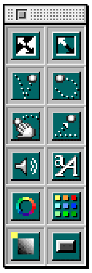
- Press **⌘-T** again to return to edit mode.

There isn't yet any interactivity in our project, and our arrow button could probably be made to look better. We'll address these problems in the next few steps.

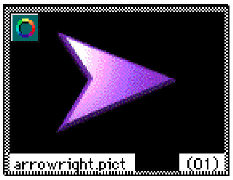
MODIFY THE APPEARANCE OF THE ARROW ELEMENT

The arrowright.pict element would look better if its black background were transparent. The graphic modifier can be used to alter the appearance of the element.

- If the Effects modifier palette (shown below) is not already visible, select **Modifier Palettes-Effects** from the View menu. The Effects modifier palette appears.

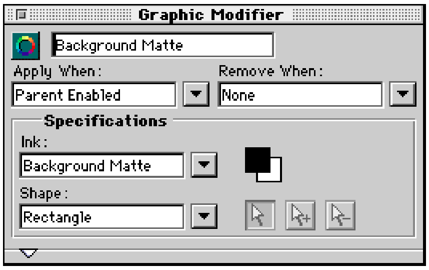


- Drag a graphic modifier from the Effects modifier palette and drop it on the arrowright.pict element. The graphic modifier icon attaches itself to the upper left corner of the element as shown below.



- Double-click the graphic modifier icon on the element. The graphic modifier's configuration dialog appears. This dialog can be used to customize the graphic modifier.
- Select the name of the modifier (the topmost text field in the modifier which reads "Graphic Modifier") and change it to **Background Matte**.
- In the Specifications section of the dialog, use the Ink pop-up to select the **Background Matte** effect. This effect will make the element's background invisible and insensitive to user mouse clicks.

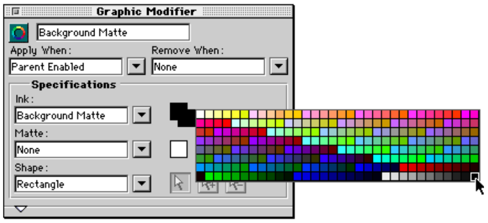
- The Graphic Modifier dialog should now look like the one shown below.



Background Matte
Parent Enabled
None
Background Matte
Rectangle

- There is one final change that needs to be made. We need to select the background color to be made transparent.

  There are two small color swatches next to the Ink pop-up menu. Click and hold on the rightmost color swatch until a palette appears (as shown below). Drag the pointer to the black color square (the rightmost color in the lowest row of the palette) and release the mouse button.



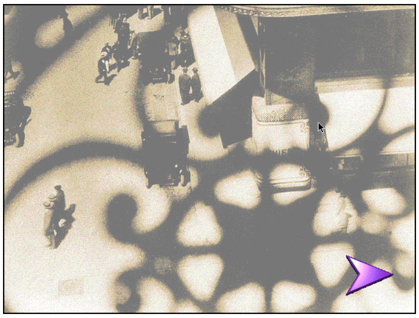
Background Matte
Parent Enabled
None
Background Matte
None
Rectangle

  Alternatively, you can continue dragging the pointer past the palette. When off of the palette, the pointer changes to an “eyedropper”. Drag the eyedropper to the background of the arrowright.pict element and release the mouse button. The background color will be “picked up” by the eyedropper.

- Close the Graphic Modifier dialog by clicking its close box.

Observe the Effect

- Observe the effect of the modifier by switching to runtime mode (press **⌘-T**). The background of the arrow should now be transparent so the scene appears as shown below.

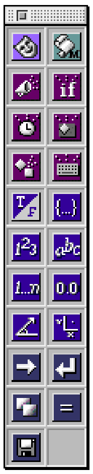


- Press **⌘-T** to return to edit mode.

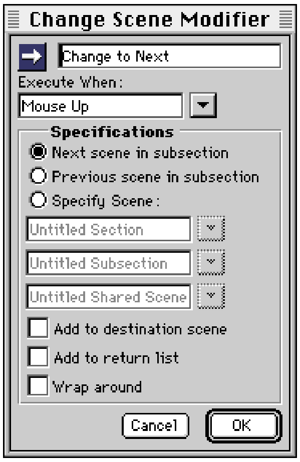
PROGRAM THE ARROW TO TRIGGER A SCENE CHANGE

Now that our arrow button looks good, its time to program some interactivity. We want the arrow to cause a scene change when it is clicked by the user. mTropolis provides a modifier just for this purpose—the change scene modifier.

- If the Logic modifier palette is not already visible, select **Modifier Palettes-Logic** from the View menu. The Logic modifier palette appears.



- Drag a change scene modifier from the Logic modifier palette and drop it on the arrowright.pict element.
- Double-click the change scene modifier icon to display its configuration dialog.
- Change the modifier's name from Change Scene Modifier to **Change to Next**.
- The default for the Execute When pop-up is Mouse Up. That is, this modifier will execute when a user clicks on the element. We don't need to change this value.
- We don't need to make any changes to the Specifications section of the dialog, either. By default, the modifier is configured to change to the next scene in the subsection.
- The dialog should now look like the one shown below.


Change Scene Modifier

Change to Next

Execute When: Mouse Up

Specifications
- [✓ Next scene in subsection]
- Previous scene in subsection
- Specify Scene:

Untitled Section

Untitled Subsection

Untitled Shared Scene

- [ ] Add to destination scene
- [ ] Add to return list
- [ ] Wrap around

- Click **OK** to dismiss the dialog.

Run the Project

With the addition of the scene change modifier, we've enabled some simple interactivity.

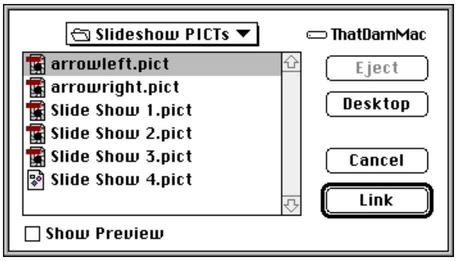
- Press **⌘-T** to switch to runtime mode.
- Notice that the cursor changes when it is over the arrow button, indicating that it can be clicked.
- Click the arrow button to change to the next scene. Repeated clicks cause the scene to change until the last scene is displayed.
- Press **⌘-T** to return to edit mode.

Now is a good time to save your project by selecting **Save** or **Save As** from the File menu.

ADD A BACK BUTTON

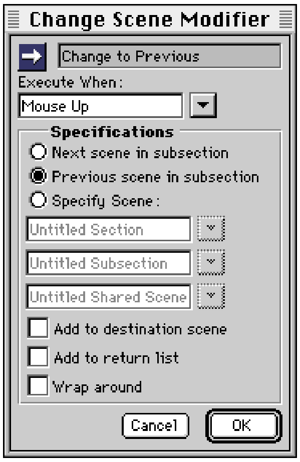
Our project lets users flip forward through the slideshow, but there's no way for them to go back to previously viewed images. Let's add a "back" button.

- Click the **arrowright.pict** element to select it.
- Select **Duplicate** from the Edit menu. A copy of the element appears.
- Use the Object Info palette to reposition this element. Enter **0** in the **X** field of the Object Info palette. Enter **375** in the **Y** field. The element should move to the lower left corner of the scene.
- Select **Link Media-File** from the File menu. A file selection dialog appears.
- In the file dialog, select the file **arrowleft.pict** from the **Slideshow PICTs** folder, found inside the **QuickStart** folder. Click **Link** to link the picture to the element.


Slideshow PICTs
ThatDarnMac
arrowleft.pict
arrowright.pict
Slide Show 1.pict
Slide Show 2.pict
Slide Show 3.pict
Slide Show 4.pict

Now we need to adjust the programming of the left arrow's change scene modifier so that it steps to the previous scene instead of the next scene.

- Double-click the change scene modifier icon on the arrowleft.pict element. The Change Scene Modifier dialog appears.
- Change the modifier's name from Change to Next to **Change to Previous**.
- In the **Specifications** section, click the "**Previous scene in subsection**" radio button. The dialog should look like the one shown below.


Change Scene Modifier
Change to Previous
Execute When: Mouse Up
Specifications
○ Next scene in subsection
● Previous scene in subsection
○ Specify Scene:
Untitled Section
Untitled Subsection
Untitled Shared Scene
☐ Add to destination scene
☐ Add to return list
☐ Wrap around

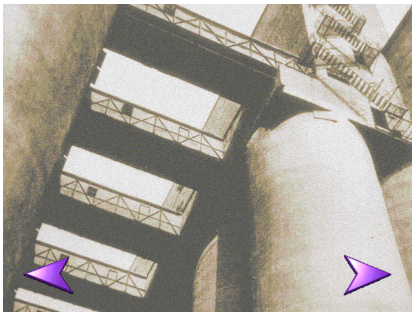
- Click **OK** to dismiss the dialog.

Run the Project

Now users can navigate both forward and backward through the slideshow.

- Press **⌘-T** to switch to runtime mode.
- Click the right arrow button to move forward through the scenes.

- Click the left arrow button to move backward through the scenes.

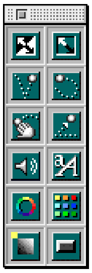


- Press **⌘-T** to return to edit mode.

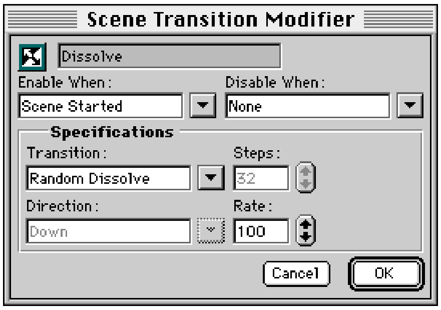
ADD SCENE TRANSITION EFFECTS

As a final enhancement to our slideshow, we'll add some scene transition effects. mTropolis' scene transition modifier adds can be used to add special effects such as fades and wipes to scene changes.

- Use the **scene pop-up** or **right scene navigation arrow**, found at the top of the Layout window, to display the **Slide Show 1.pict** scene.
- Drag a scene transition modifier from the Effects palette and drop it on the scene. The modifier attaches itself to the top left corner of the scene.



- Double-click the scene transition modifier icon to display its configuration dialog.
- Change the modifier's name from Scene Transition Modifier to **Dissolve**.
- Use the **Transition** pop-up to select the **Random Dissolve** effect. The dialog should look like the one shown below.



Dissolve

Enable When: Scene Started

Disable When: None

Specifications

Transition: Random Dissolve

Steps: 32

Direction: Down

Rate: 100

- Click **OK** to dismiss the dialog.

Copy the Scene Transition Modifier to Each Scene

- While the scene transition modifier is still selected, choose **Copy** from the Edit menu.
- Click the **right scene navigation arrow**, found at the top of the Layout window, to display the next scene (Slide Show 2.pict).
- Click on the scene to select it.
- Choose **Paste** from the Edit menu. A new copy of the scene transition modifier appears on the scene.
- Click the right scene navigation arrow again to display the third scene (Slide Show 3.pict).
- Click on the scene to select it.
- Once again, choose **Paste** from the Edit menu. A scene transition modifier appears on the Slide Show 3.pict scene.

- Repeat these steps once more to add the modifier to the fourth scene (Slide Show 4.pict).

Run the Project

Now each scene fades in with a nice dissolve effect instead of the abrupt change we saw before.

- Press **⌘-T** to switch to runtime mode.
- Click the buttons to move through the images in the slideshow. Each scene "dissolves" into the next.
- Press **⌘-T** to return to edit mode.

That's all there is to creating a simple slideshow presentation with mTropolis! Don't forget to save your work.

TROUBLESHOOTING

If you have difficulties completing this tutorial, you might want to examine a finished version. Select **Open** from the File menu and select the project file **Completed Slideshow** found in the **QuickStart** folder. A "Completed Slideshow" Layout and Structure window will appear.

THE ADVANCED SLIDESHOW

It takes only a little more effort to add sound, motion, and more elaborate effects to the slideshow. The **Advanced Slideshow** project, found in the **QuickStart** folder contains an expanded version of the simple slideshow tutorial. Open this project in mTropolis and press **⌘-T** to run it.

Notice that the arrow buttons move and make a sound when they are clicked. Examining the buttons in edit mode (they are located on the Untitled Shared Scene) shows that they contain sophisticated behaviors that control their actions.

Examining this project may give you ideas about enhancements that you can make to your own slideshow.

MORE TUTORIALS

Chapter 8, "In-Depth Tutorial—mPuzzle" contains another, more challenging mTropolis tutorial. The mPuzzle tutorial may take several hours to complete, but it demonstrates many more mTropolis programming concepts. It contains examples of creating behaviors, using "aliased" modifiers, writing Miniscript code, and using animation files.

More Tutorials

Chapter 8. In-Depth Tutorial—mPuzzle

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 **Tutorials** folder of the mTropolis installation. If the tutorial files have not been installed, they can be installed by running the installer from the mTropolis CD-ROM, or by dragging the **Tutorials** 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 **In-Depth** folder (contained in the **Tutorials** 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.
- 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 8.1) 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 places, the "M"

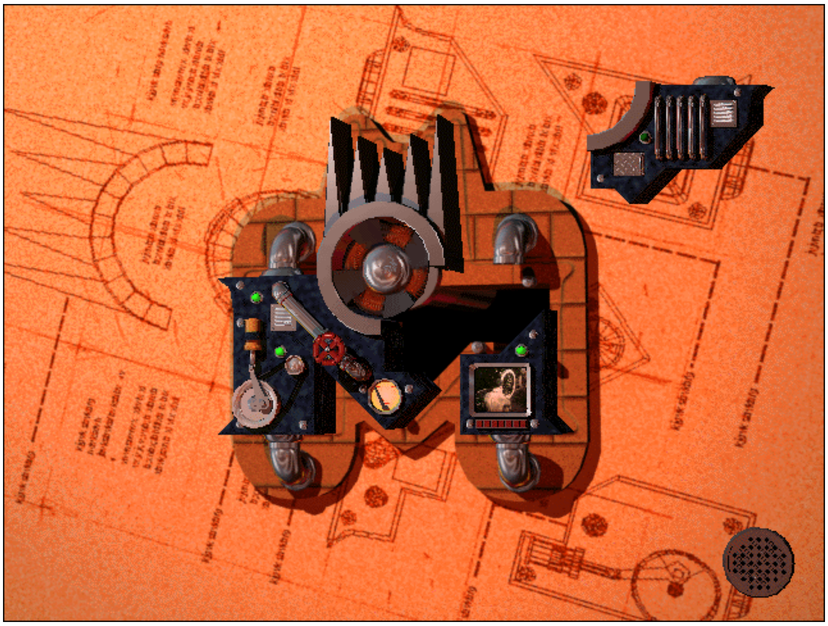


Figure 8.1 Solving the mTutorial puzzle

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.

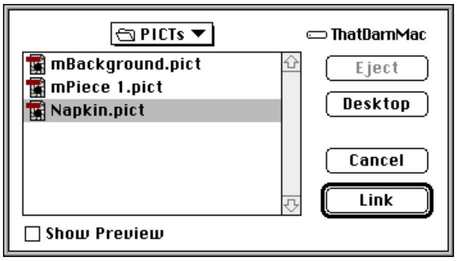
CREATE THE FIRST SCENE

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

Adding the Background Image

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

- Click on the Untitled Scene (i.e., inside the large white region in the Layout window) to select it.
- Choose **Link Media-File** from the File menu. A standard file selection dialog appears as shown below.

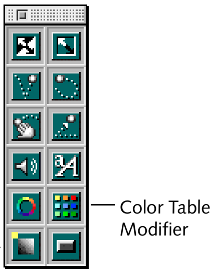


- Choose the image named **Napkin.pict** from the **PICTs** folder, found inside the **Media** folder (located in the In-Depth subfolder of the Tutorials folder).
- An alert appears, warning that this 8-bit image uses a custom color table and may not appear as expected. Click "OK" to dismiss the dialog. In the next step, we'll link the image's color table to the project so our images will display properly.
- The Napkin.pict image fills the scene.

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 the Effects Modifier Palette 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 **Effects**, that palette is already visible. If there is not a checkmark, select the **Effects** 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). The color table modifier attaches itself to the upper left corner of the scene.



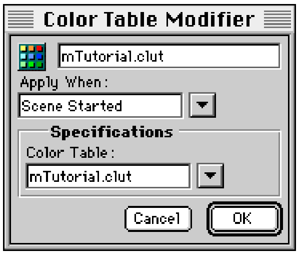
Hot Tip
Hold down the control key while moving the cursor over the modifier palette to view each modifier's name.

- 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 **Media**

folder. Your Color Table Modifier dialog should now look like the one shown below. Click the "OK" button to dismiss the Color Table Modifier dialog.


Color Table Modifier

mTutorial.clut

Apply When: Scene Started

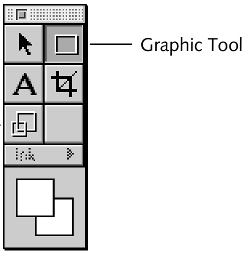
Specifications

Color Table: mTutorial.clut

- To view the effect of a color table that has been applied to a scene while in edit mode, choose **mTutorial.clut** from the **Preview Color Table** cascading menu option, found in the View menu. The screen updates to reflect the new color palette.

Add the Logo Movie

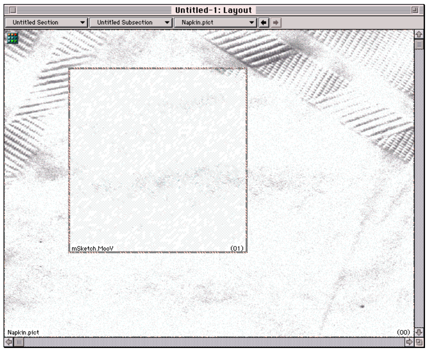
Now let's put the logo QuickTime movie on top of this background image. First we'll create an element to contain the QuickTime movie.



- 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 **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.

The Layout window should look similar to the one shown below.

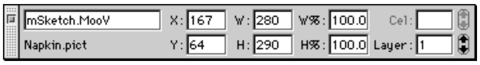


Position the Movie

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.

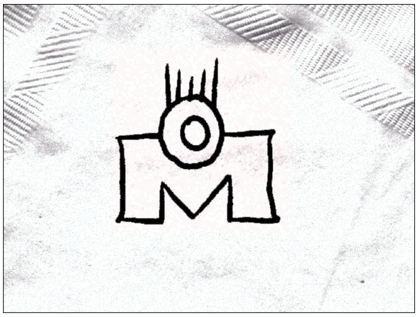
- Select the mSketch.MooV element, and enter 167 in the "X" field and 64 in the "Y" field. Use the tab key to jump between fields and the enter key to confirm the final data entry.



mSketch.MooV X: 167 W: 280 W%: 100.0 Cel:
Napkin.pict Y: 64 H: 290 H%: 100.0 Layer: 1

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.

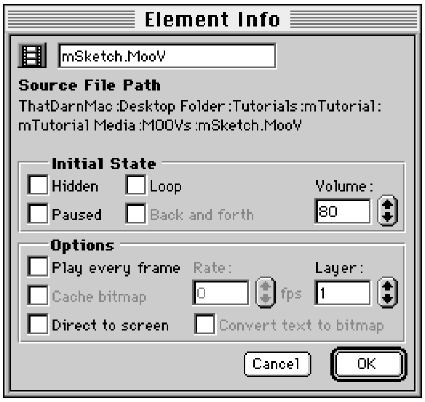


Changing an Element's Properties

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. The Element Info dialog should look similar to the one shown below.



Element Info

mSketch.MooV

Source File Path
ThatDarnMac:Desktop Folder:Tutorials:mTutorial:mTutorial Media:MOOVs:mSketch.MooV

Initial State
- [ ] Hidden
- [ ] Loop
- [ ] Paused
- [ ] Back and forth

Volume: 80

Options
- [ ] Play every frame
- [ ] Cache bitmap
- [ ] Direct to screen
- [ ] Convert text to bitmap

Rate: 0 fps
Layer: 1

- 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).

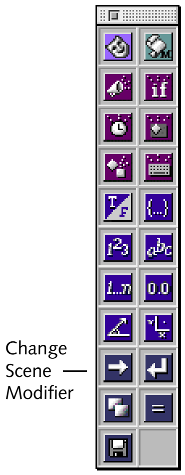
- Name and store the project as you would any other application file.

- 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.
- Notice that the title of your Layout window changes to reflect the new name of your project.

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.

- Ensure that the Logic Modifier Palette 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 **Logic**, that palette is already visible. If there is not a checkmark, select the **Logic** menu item to display the palette.
- Drag a **change scene** modifier from the Logic modifier palette and drop it on the movie element. The modifier icon attaches itself to the upper left corner of the mSketch.Moov 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 8.2. Now when the movie ends, a scene change will occur.

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.

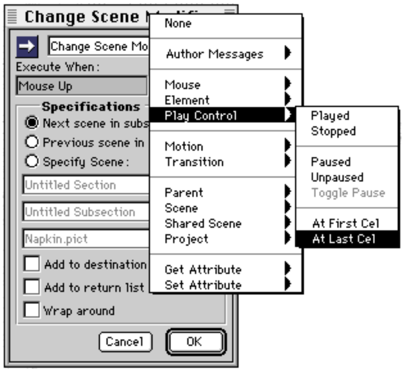
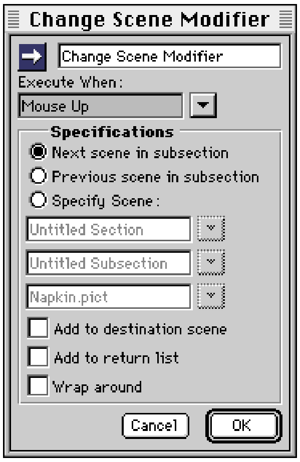


Figure 8.2 Configuring the "Execute When" field of the Change Scene Modifier

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. A new change scene modifier icon appears next to the previously-created one. 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 Napkin.pict 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. Your new Change Scene dialog should look like the one shown below.


Change Scene Modifier

Change Scene Modifier

Execute When: Mouse Up

Specifications

- (●) Next scene in subsection
- ( ) Previous scene in subsection
- ( ) Specify Scene:

Untitled Section

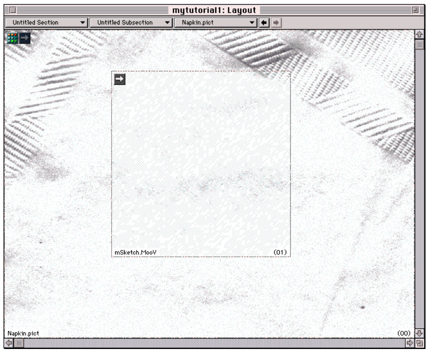
Untitled Subsection

Napkin.pict

- [ ] Add to destination scene
- [ ] Add to return list
- [ ] Wrap around

- Click OK to accept the change. The dialog disappears.

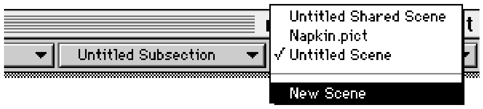
The Layout window should now look similar to the one shown below. We are now ready to create the next scene in the project.



Create a New Scene

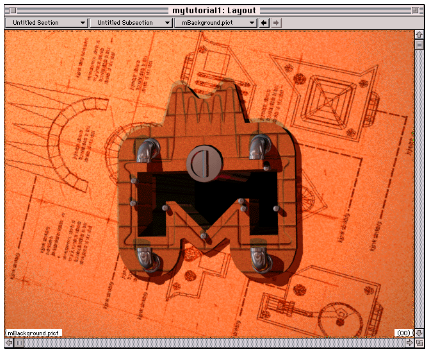
To create a new scene in the Layout view, use the third pop-up on the right at the top of the window (where it now reads “Napkin.pict”). The pop-up lists all scenes in the current subsection and a “New Scene” option that can be used to create new scenes. New Scene always appears as the last item in this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

- Select **New Scene** from the scene pop-up as shown in the image below. 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 Untitled 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 **Media** folder.
- The custom color table alert appears again. Click “Don’t Warn Again” to dismiss the alert. It will not appear again while we’re working on this project. Since we’re already viewing the correct color palette for this image, the image will appear correctly.
- The mBackground.pict file fills the scene as shown in the image below. Note also that the name of our new scene changes to “mBackground.pict”.



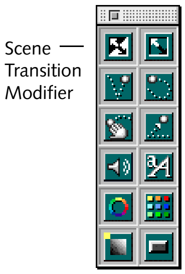
Let’s see the effect of adding this scene.

- Return to the previous scene by using the scene pop-up menu 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.
- The Layout window reappears, showing the Napkin.pict scene.

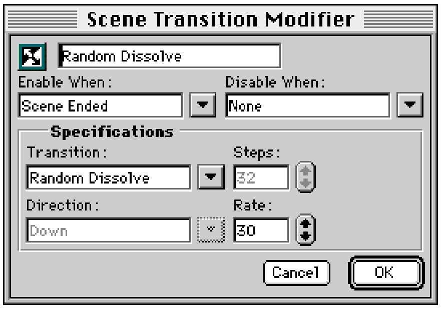
Adding a Scene Transition

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 Effects 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-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**. The dialog should look like the one shown below.



- 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.

PROGRAMMING THE SECOND SCENE

Now let's add the components that make up the second scene.

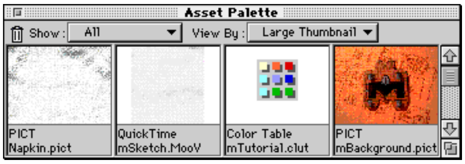
- Navigate to the second scene by selecting **mBackground.pict** from the scene pop-up menu (the third pop-up from the right at the top of the Layout window) or click the right scene navigation arrow (also found at the top of the Layout window).

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.

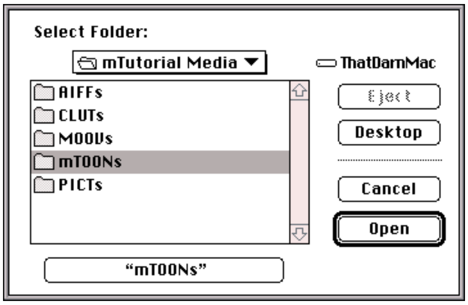
- Select **Asset Palette** from the View menu. The Asset Palette appears.



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 folder selection dialog appears. Select the **mTOONs** folder found within the **Media** folder. Click on the button at the bottom of the folder selection dialog when the name "**mTOONs**" appears in that button as shown below.



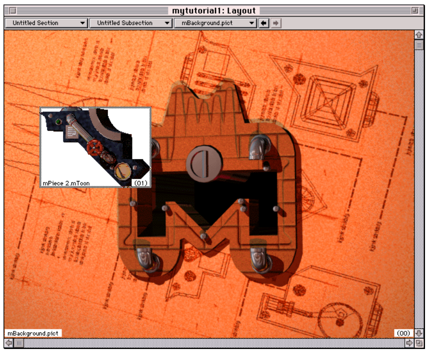
Also link one more file into the project. First, ensure that no elements in the scene are selected. If no elements are selected, the next media file we link will appear only in the Asset Palette, instead of appearing in the scene.

- Click on the scene to select only the scene.
- Shift-click on the scene to deselect it. Now we are sure that no elements are selected.

- 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 **Media** folder. 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.

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 Layout window.

- Drag the mToon element named **mPiece 2.toon** from the Asset Palette and drop it onto the mBackground.pict scene. The Layout window should look similar to the one shown below.



Programming the First Puzzle Piece

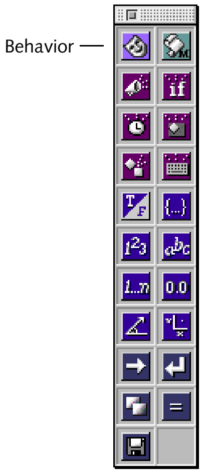
Since all of our puzzle pieces need to behave in a similar fashion, we will program one element first and then create an *alias* of that programming that we can copy onto all the other puzzle pieces.

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.

- Drag a **behavior** modifier from the Logic 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.

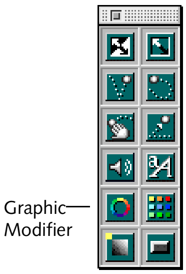


- Double-click the behavior modifier to display its configuration dialog.
- Change the name of the behavior from Behavior to **Puzzle Piece**. Do not close the behavior dialog.

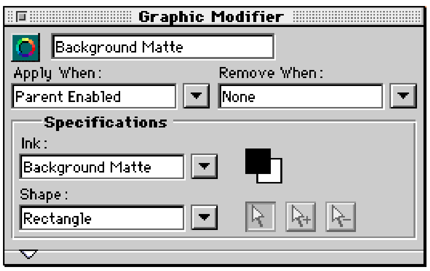
Making the Puzzle Piece Transparent

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

- Drag a graphic modifier from the Effects 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 **Background Matte**. Its purpose is to apply a background matte effect to the element with which it is associated.
- Use the Ink pop-up menu to select the **Background Matte** option. The dialog should look like the one shown below.



Graphic Modifier

Background Matte

Apply When: Parent Enabled

Remove When: None

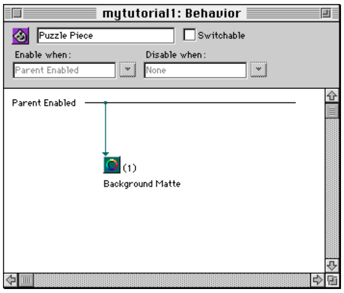
Specifications

Ink: Background Matte

Shape: Rectangle

- Close the Graphic Modifier dialog (by clicking on its close box) to accept the change.

The behavior dialog should now look like the one shown below.



mytutorial1: Behavior

Puzzle Piece

Switchable

Enable when: Parent Enabled

Disable when: None

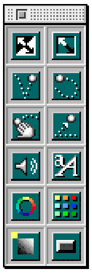
Parent Enabled

(1) Background Matte

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 Effects 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.

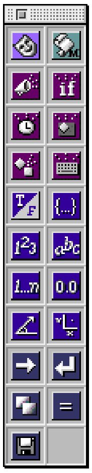
Close the Puzzle Piece behavior window by clicking its close box.

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.

Changing Other Aspects of the Puzzle Piece Behavior

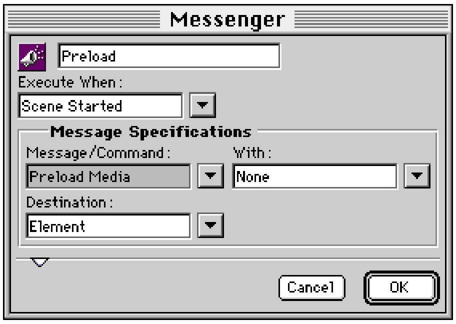
Since these elements are reasonably small animation files, and we want good playback performance, we will preload them into RAM. This can be accomplished by sending the puzzle piece a *Preload* command.

- Re-open the Puzzle Piece behavior by double-clicking its icon.
- Drag a messenger modifier from the Logic 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*** command.

- No change needs to be made to the With and Destination menus. The dialog should look like the one shown below.

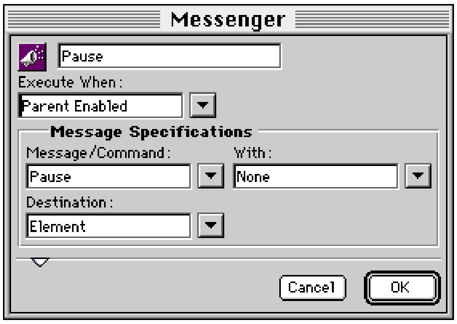


Preload
Execute When: Scene Started
Message Specifications
Message/Command: Preload Media
With: None
Destination: Element

- Click OK to accept the changes. The Messenger dialog disappears.

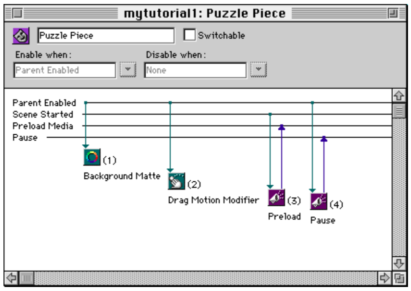
By default, the puzzle piece animations play when the scene starts. However, we want the animations to be paused until the puzzle is complete. The animation can be paused by sending a *Pause* command to the puzzle piece.

- Drag another messenger modifier from the Logic 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. The dialog should look like the one shown below.



Pause
Execute When: Parent Enabled
Message Specifications
Message/Command: Pause
With: None
Destination: Element

- Click OK to accept the changes and dismiss the Messenger dialog.
- Your Puzzle Piece behavior window should look like the one shown below.



Puzzle Piece
Switchable
Enable when: Parent Enabled
Disable when: None

- Click the behavior's close box to dismiss the Puzzle Piece behavior window.

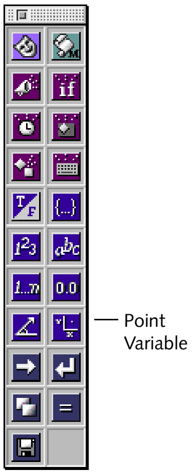
Note that the animation could also have been paused by setting the element's "paused" attribute via the Element Info dialog. However, by using a messenger to pause the animation and placing that messenger in a behavior that will be used on all the other puzzle pieces, we have eliminated the need to individually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

Adding the Puzzle Snap-In Function

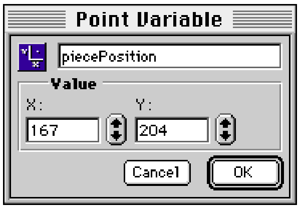
Let’s now program the “snap-in” functionality of the puzzle piece. First, we need a way to store the screen coordinates of the correct position of the puzzle piece. We can use a point variable to store this value. Since every puzzle piece will use the same (aliased) behavior, but will all have different 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 Logic 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**. Note that there are no spaces in that name!
- Enter **167** into the modifier’s X field and **204** into the Y field. The dialog should look like the one shown below.


Point Variable

piecePosition

Value

X: 167 Y: 204

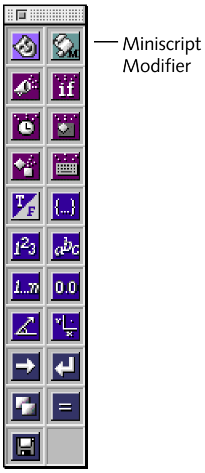
- Click OK to accept the changes and dismiss the Point Variable dialog.

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.

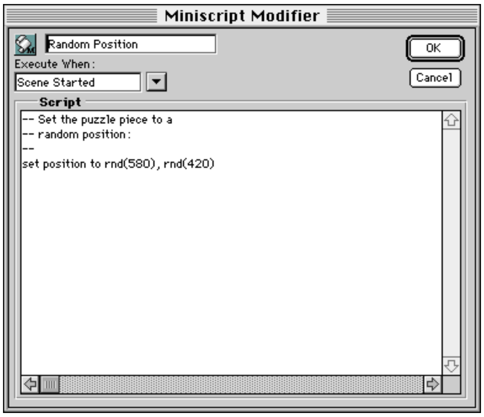
- Double-click the Puzzle Piece behavior icon to open its window.
- Drag a **Miniscript** modifier from the Logic modifier palette and drop it into the Puzzle Piece behavior window.



- 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 below.



Random Position

Execute When: Scene Started

Script

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

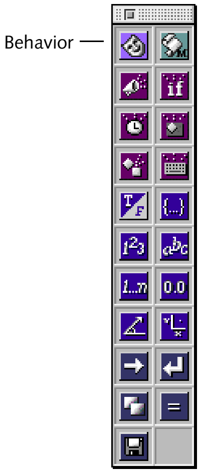
The first three lines of our script are comments—the two dashes that start each line tell mTropolis to ignore any text that follows on that line. Comments are not required for the script to function properly, but help to make your scripts easier to read and debug.

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

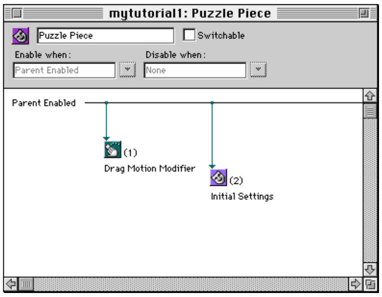
- Click OK to accept these changes and dismiss the Miniscript Modifier dialog.

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.

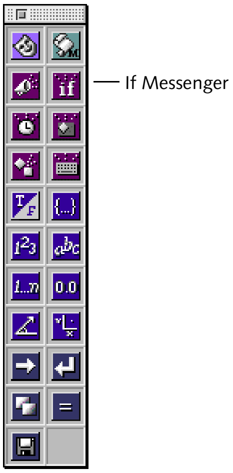


- Drag a new **behavior** from the Logic modifier palette and drop it into the open Puzzle Piece behavior window.
- This time, instead of opening the behavior to rename it, simply click on the behavior's name shown below its icon, and enter the new name, **Initial Settings**. Click outside the name when 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 **Background Matte**, 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. The Puzzle Piece behavior window should now look like the one shown below.



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 Logic 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

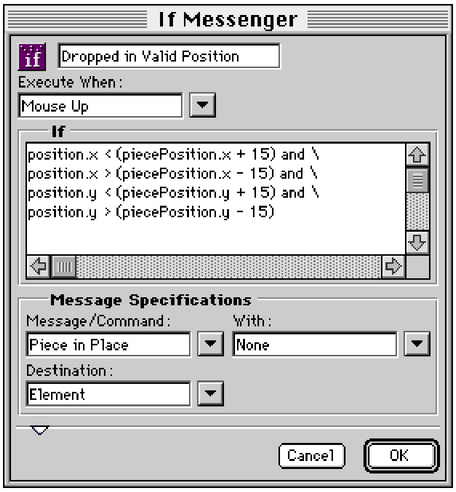
what we want, so we don't need to change the Execute When pop-up.

- In the "If" text field, replace the existing text ("true") with 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. The dialog should look like the one shown below.


If Messenger

Dropped in Valid Position

Execute When: Mouse Up

If

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

Message Specifications

Message/Command: Piece in Place

With: None

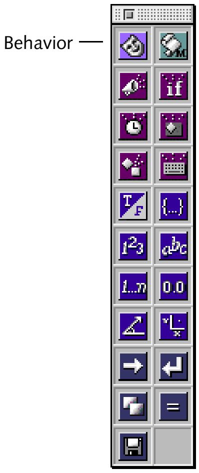
Destination: Element

- Click "OK" to save your changes to this configuration dialog. An alert appears, asking if you want to create the new author message. Click "OK" on this alert.

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 Logic modifier palette and drop it into the Puzzle Piece behavior window.



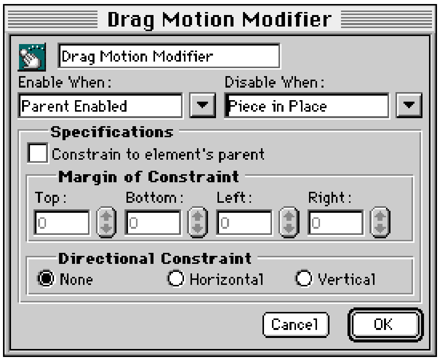
- Click on the name of the behavior that appears below the icon and enter the new name, **Piece in Place**. Click outside of the name when you are done.

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 in the Puzzle Piece window 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. The dialog should look like the one shown below.


Drag Motion Modifier

Drag Motion Modifier

Enable When: Parent Enabled

Disable When: Piece in Place

Specifications

☐ Constrain to element's parent

Margin of Constraint

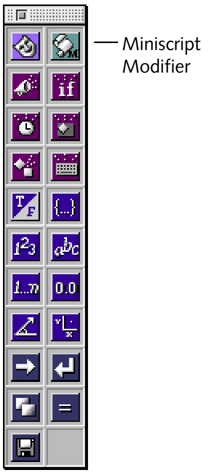
Top: 0 Bottom: 0 Left: 0 Right: 0

Directional Constraint

[✓ None] Horizontal Vertical

- Click OK to confirm your changes and dismiss the Drag Motion Modifier dialog.
- Move the "Dropped in Valid Position" If messenger you created previously from the Puzzle Piece behavior 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.

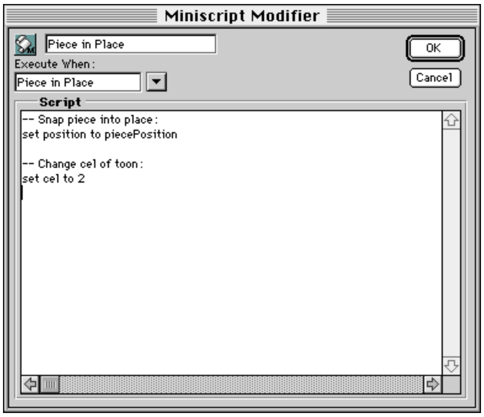


- Drag a new **Miniscript** modifier from the Logic palette and drop it into the Piece in Place behavior.
- Double-click the modifier to display its configuration dialog.
- Name the new Miniscript modifier **Piece in Place**.
- Use the Execute When pop-up menu to configure the modifier to activate when 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
```

The dialog should look like the one shown below.



Piece in Place

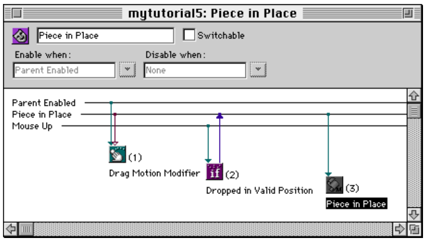
Execute When: Piece in Place

```
-- 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. Your Piece in Place behavior should look similar to the one shown below.

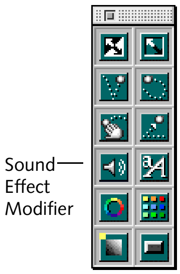


You may want to test your programming up to this point. Press ⌘-T to switch to runtime mode. When the puzzle appears, drag the puzzle piece and drop it near its correct position (this piece belongs on the left-side slanted line of the “M”). You should notice it snap into place when you release it. Once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

Adding the Snap Sound

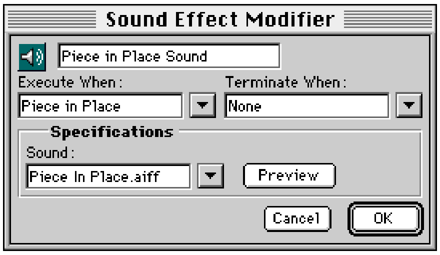
A sound effect would be nice feedback to notify the user that the puzzle piece is in place.

- Drag a sound effect modifier from the Effects 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 located in the **Media** folder.

- Click the Preview button to preview the sound. The dialog should look like the one shown below.



Sound Effect Modifier

Piece in Place Sound

Execute When: Piece in Place

Terminate When: None

Specifications

Sound: Piece In Place.aiff

- Click OK to accept your changes and dismiss the Sound Effect Modifier dialog.

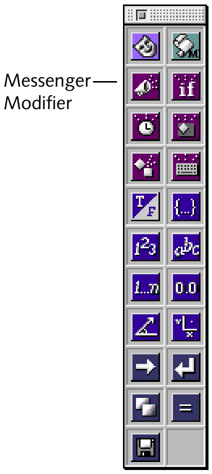
Disabling the Piece in Place Behavior

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.

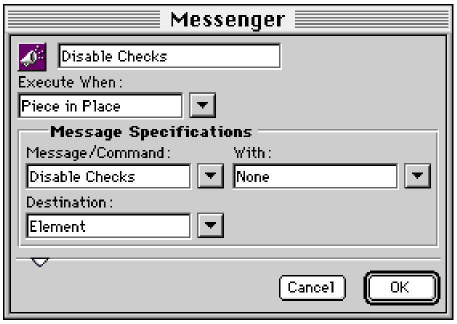
In our project, it makes sense to deactivate the Piece in Place behavior once a piece is actually in place. By switching this behavior off, we can insure that the project doesn't keep checking for puzzle pieces that are already in their proper places.

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

- Drag a new messenger modifier from the Logic 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**. The dialog should look like the one shown below.



Messenger

Disable Checks

Execute When: Piece in Place

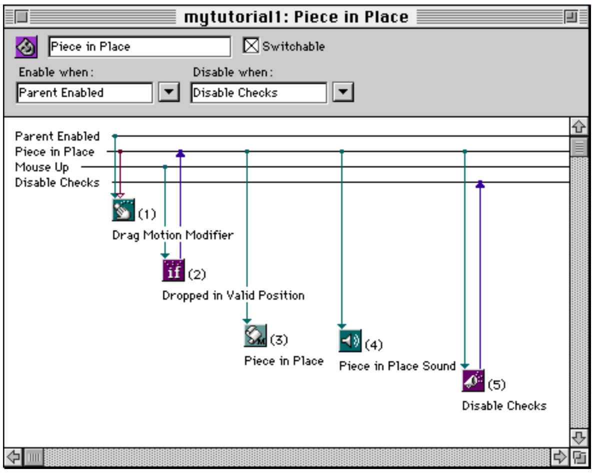
Message Specifications

Message/Command: Disable Checks

With: None

Destination: Element

- Click OK. A dialog appears, asking if you want to create the new author message. Click OK.


mytutorial1: Piece in Place

Piece in Place ☒ Switchable

Enable when: Parent Enabled

Disable when: Disable Checks

Figure 8.3 The completed Piece in Place behavior

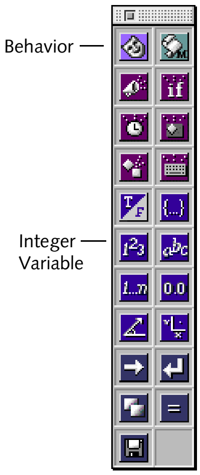
- In the Piece in Place behavior window, select the **Switchable** checkbox found next to the behavior name.
- Two previously inactive pop-up menus become accessible. Now the behavior has Enable When and Disable When pop-up menus that can be used to specify the messages that activate and deactivate this 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 8.3. We are finished modifying the Piece in Place behavior, so close the window by clicking its close button.

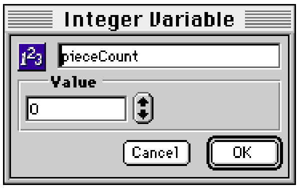
Keeping Track of the Puzzle Pieces in Place

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.

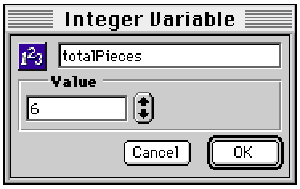
- Start by dragging a new behavior modifier from the Logic modifier palette and dropping it into the Puzzle Piece behavior window.



- 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 Logic 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.


Integer Variable
pieceCount
Value
0

- 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.


Integer Variable
totalPieces
Value
6

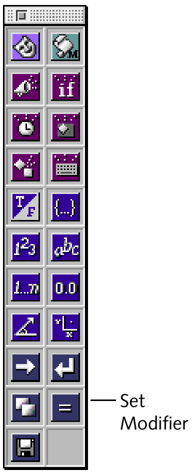
- Click OK to dismiss the variable's dialog.

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.

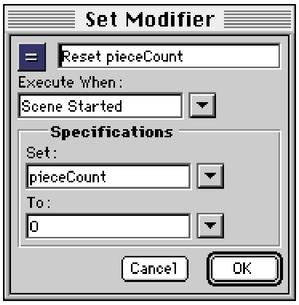
Whenever the puzzle scene is first displayed, we want to make sure that the pieceCount variable is initialized to zero.

- Drag a **set modifier** from the Logic modifier palette and drop it into the Puzzle Status behavior window.



- Double-click the set modifier icon to display its configuration dialog.
- Change the modifier's name to **Reset pieceCount**.
- Use the dialog's Execute When pop-up to select the **Scene-Scene Started** message.
- Use the modifier's Set pop-up to select **Behavior-pieceCount**.
- Highlight the modifier's To field and enter **0** (zero).

The dialog should look like the one shown below.


Set Modifier
Reset pieceCount
Execute When:
Scene Started
Specifications
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.

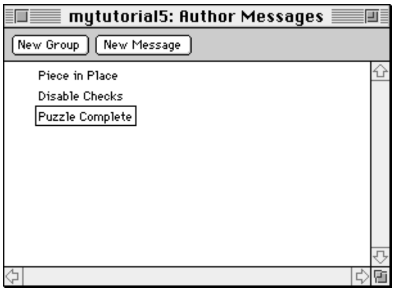
Notifying the Environment that the Puzzle Is Complete

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.
- Click the untitled author message and change its name to **Puzzle Complete**. The

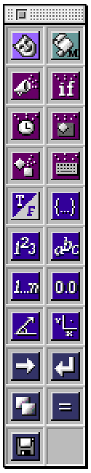
Author Messages window should look like the one shown below.



Piece in Place
Disable Checks
Puzzle Complete

- Close the Author Messages window by clicking its close button.

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

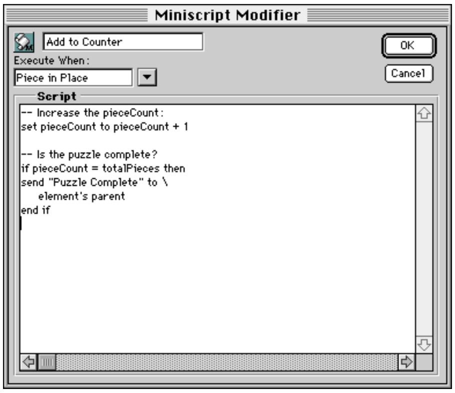


- Drag a new Miniscript modifier to the Puzzle Status behavior window.
- Double-click the modifier's icon to display its configuration dialog. Change its name to **Add To Counter** and configure the Execute When field to execute on **Author Messages-Piece in Place**.
- Enter the following script in the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



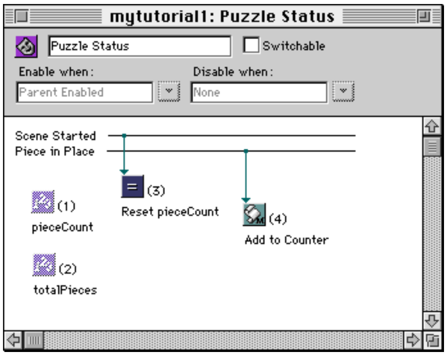
Add to Counter
Execute When:
Piece in Place
Script

```
-- 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
```

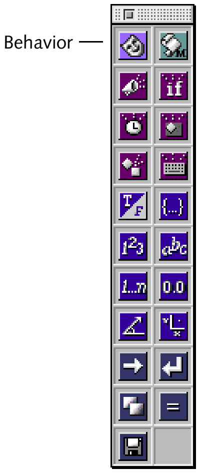
- Click OK to dismiss the Miniscript Modifier dialog.
- Your Puzzle Status behavior window should look similar to the one shown below. Close

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.

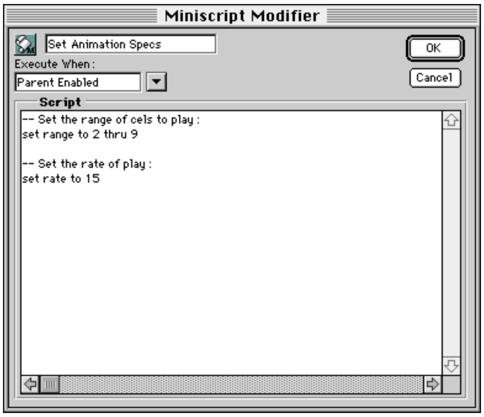


- Add a new **behavior** modifier to the Puzzle Piece behavior.
- Double-click the behavior icon to display its configuration window.
- Change the name of the behavior to **Puzzle Complete**.
- Select the **Switchable** box at the top of the behavior window. The Enable When and Disable When pop-ups become 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. The dialog should look like the one shown below.



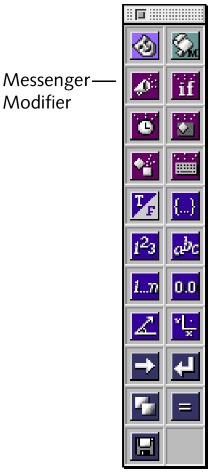
Set Animation Specs
Execute When:
Parent Enabled
Script

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

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

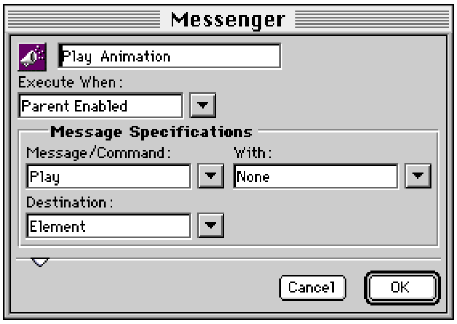
- Click the OK button to dismiss the Miniscript Modifier dialog.

- Drag a **messenger** modifier from the Logic 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.

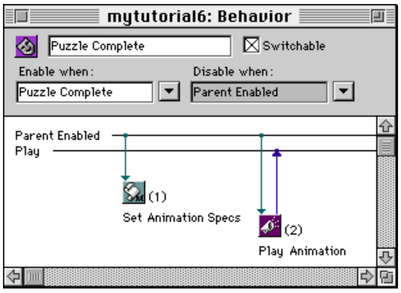
- Use the Message/Command pop-up menu to select the **Play Control-*Play*** command. When sent to the element, this command will make its animation begin playing. The dialog should look like the one shown below.



Play Animation
Execute When:
Parent Enabled
Message Specifications
Message/Command: Play
With: None
Destination: Element

- Click OK to close the messenger dialog and confirm the changes.

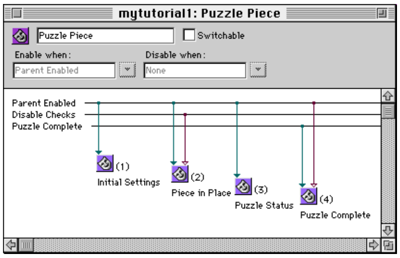
- Your Puzzle Complete behavior should now look like the one shown below. Close the Puzzle Complete behavior window by clicking its close button.



Puzzle Complete
☒ Switchable
Enable when: Puzzle Complete
Disable when: Parent Enabled

- Your Puzzle Piece behavior should look like the one shown below. Close the Puzzle

Piece behavior window by clicking its close button.

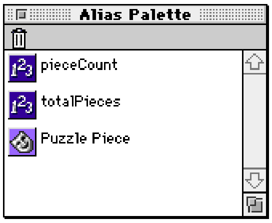


If you haven't saved your project in a while, now is a good time to select **Save** or **Save As** from the File menu.

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 as shown below.



- 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 8.1. 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

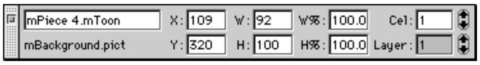
Table 8.1: piecePosition values for each puzzle piece

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 8.2, in the "Layer" field of the Object Info Palette. When the pieces have been assigned the layer orders shown in Table 8.2, the completed "M" machine will look its best.



mPiece 4.mToon X: 109 W: 92 W%: 100.0 Cel: 1
mBackground.pict Y: 320 H: 100 H%: 100.0 Layer: 1

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 8.2: 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.

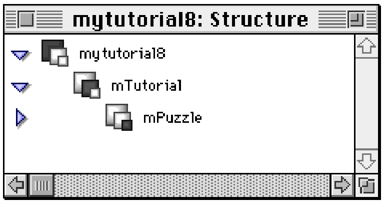
NAMING STRUCTURAL ELEMENTS

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**.
- Note that the top-level “project” component has the same name as the file your project is saved in.

Your Structure window should look similar to 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 elements 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 **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 8.4.
- Double-click the sound icon to display its Element Info dialog. In the dialog’s Initial State section, select the **Paused** and **Loop**

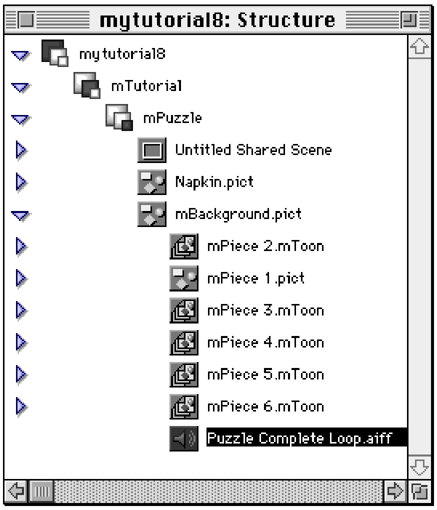
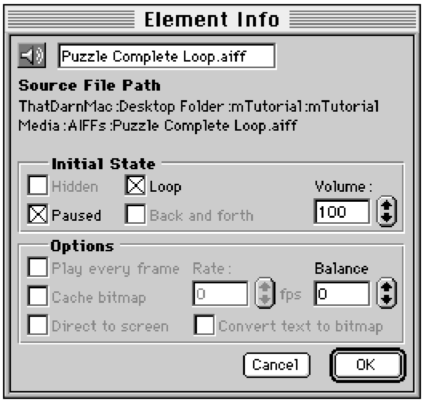


Figure 8.4 Adding a sound element to the structure window

options. The Element Info dialog should look like the one shown below.


Element Info

Puzzle Complete Loop.aiff

Source File Path
ThatDarnMac:Desktop Folder:mTutorial:mTutorial Media:AIFFs:Puzzle Complete Loop.aiff

Initial State
Hidden: ☐ Loop: ☒ Volume: 100
Paused: ☒ Back and forth: ☐

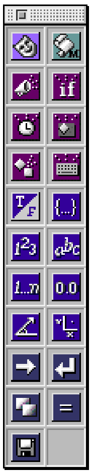
Options
Play every frame: ☐ Rate: 0 fps Balance: 0
Cache bitmap: ☐
Direct to screen: ☐ Convert text to bitmap: ☐

- Click OK to confirm the changes and close the dialog.

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

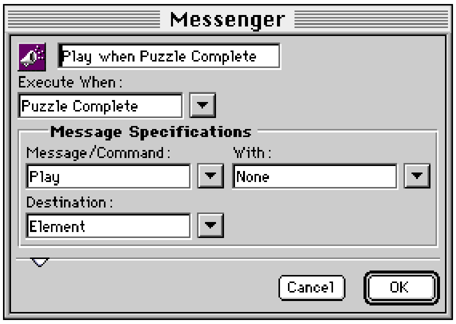
- Drag a **messenger** onto the sound icon. The icon will seem to "disappear" into the sound icon. Click the sound icon. Click the sound icon's open/close triangle to reveal the next level of the structure hierarchy—the sound element's modifiers. Now the Messenger icon is visible.

Messenger— Modifier



- 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.

The dialog should look like the one shown below.



Play when Puzzle Complete
Execute When: Puzzle Complete
Message Specifications
Message/Command: Play
With: None
Destination: Element

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

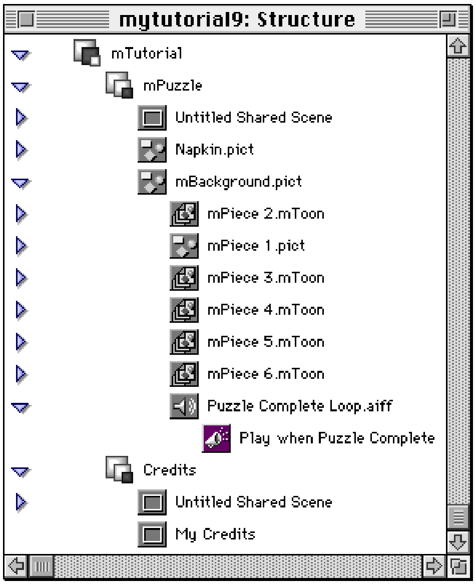
Press ⌘-T to run the project and hear the difference when the puzzle is completed. Press ⌘-T to return to edit mode.

This is another good time to save your project. Select **Save** or **Save As** from the File menu to save your work.

THE CREDITS SCENE

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

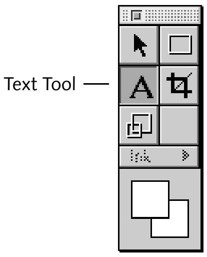
- Create a new subsection by choosing the **New Subsection** option from the Subsection pop-up on the *Layout window.* This menu is the second menu from the left at the top of the Layout window (its label currently reads "mPuzzle"). A new "Untitled Subsection" is created. The Layout window updates to 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**. Your structure view should look similar to the one shown below.



mTutorial
mPuzzle
Untitled Shared Scene
Napkin.pict
mBackground.pict
mPiece 2.mToon
mPiece 1.pict
mPiece 3.mToon
mPiece 4.mToon
mPiece 5.mToon
mPiece 6.mToon
Puzzle Complete Loop.aiff
Play when Puzzle Complete
Credits
Untitled Shared Scene
My Credits

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

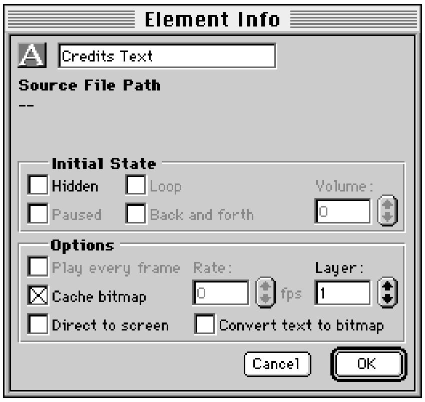
- Click on the scene in the Layout window.



- Select the text tool from the tool palette. The cursor changes to an I-beam with a small square next to it.
- Create a text element 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**. The dialog box should look like the one shown below.



Element Info

Credits Text

Source File Path
--

Initial State
[ ] Hidden [ ] Loop Volume: 0
[ ] Paused [ ] Back and forth

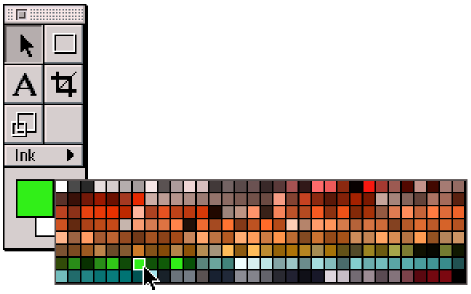
Options
[ ] Play every frame Rate: 0 fps Layer: 1
[✓] Cache bitmap
[ ] Direct to screen [ ] Convert text to bitmap

- 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

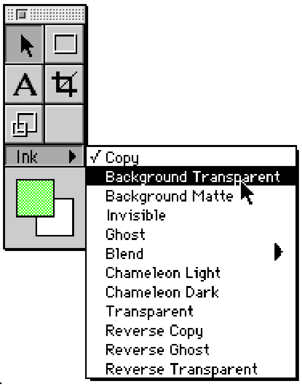
color in the palette and release the mouse button to select a color as shown below.



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

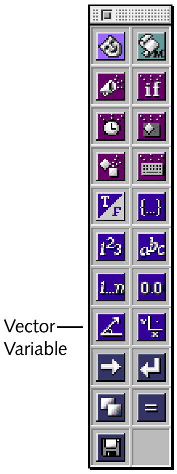
We should also make the background of the text element transparent.

- 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.

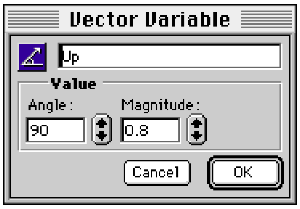


Now let's modify the text element so that it scrolls up and off the screen. To do this we will use a vector motion modifier. This modifier works in conjunction with the vector variable.

- Drag a **vector variable** from the Logic 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. The dialog should look like the one shown below.


Vector Variable

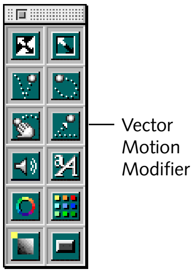
Up

Value

Angle : 90

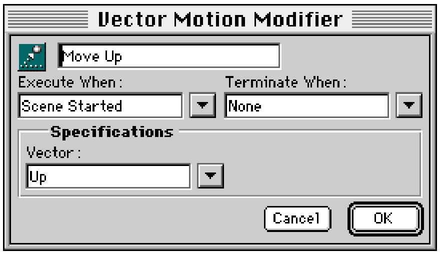
Magnitude : 0.8

- Click OK to confirm the changes and close the dialog.
- Drag a vector motion modifier from the Effects 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.

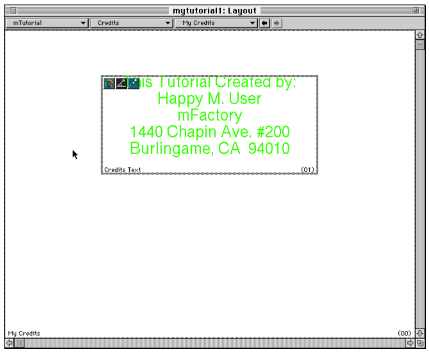
- Use the Vector pop-up to select the name of the vector variable associated with this vector motion. Select the **Credits Text-Up** option. The dialog should look like the one shown below.



Move Up
Execute When: Scene Started
Terminate When: None
Specifications
Vector: Up

- Click "OK" to dismiss the Vector Motion dialog.

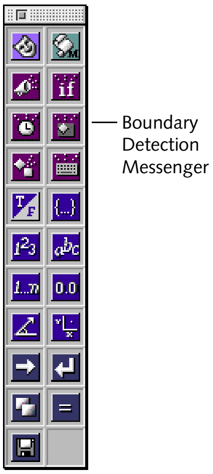
Your Layout window should now look similar to the one shown below.



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

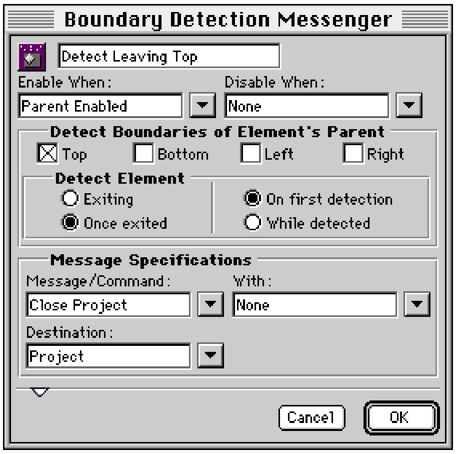
Now press ⌘-Y to run the project from this scene (⌘-T runs the project from the very first scene). The text element should rise from its starting position to the top of the screen. Press ⌘-Y again to return to edit mode.

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 Logic 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.

The dialog should look like the one shown below.


Boundary Detection Messenger

Detect Leaving Top

Enable When: Parent Enabled

Disable When: None

Detect Boundaries of Element's Parent: [✓ Top] Bottom Left Right

Detect Element: Exiting [✓ Once exited] — [✓ On first detection] While detected

Message Specifications

Message/Command: Close Project

With: None

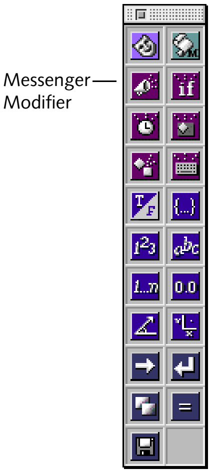
Destination: Project

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

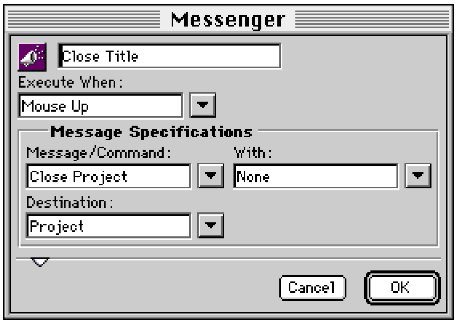
In the Layout window, you might want to position the text element slightly below the bottom of the frame of the scene. When the text scrolls up it will look like a credit roll like you might see at the end of a movie.

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 Logic 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. The dialog should look like the one shown below.


Messenger

Close Title

Execute When: Mouse Up

Message Specifications

Message/Command: Close Project

With: None

Destination: Project

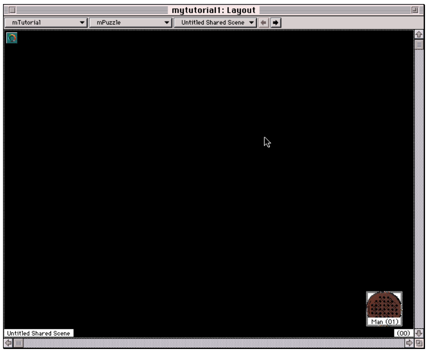
- Click OK to dismiss the dialog.

Using the Shared Scene

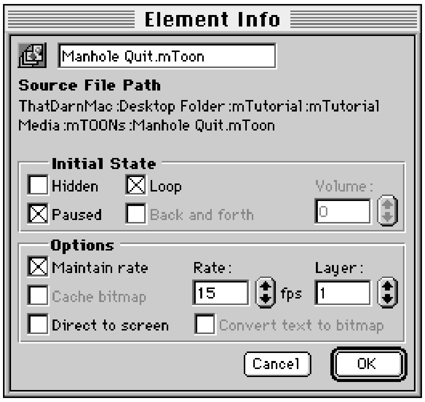
You might have noticed that even though we have created a fully-functional credits scene, there's no way for it to be activated from earlier scenes in the project!

We'll create a button on the shared scene of the mPuzzle subsection 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"). Then select **Untitled Shared Scene** from the scene pop-up menu (the third pop-up from the left).
- 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 **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. Your shared scene should look something like the one shown below.



- Double-click the Manhole Quit.mtoon element to open its Element Info dialog.
- In the Initial State section of the dialog, ensure that the **Paused** checkbox is checked. The dialog should look like the one shown below.



Element Info

Manhole Quit.mToon

Source File Path
ThatDarnMac:Desktop Folder:mTutorial:mTutorial Media:mTOONs:Manhole Quit.mToon

Initial State
☐ Hidden ☒ Loop Volume: 0
☒ Paused ☐ Back and forth

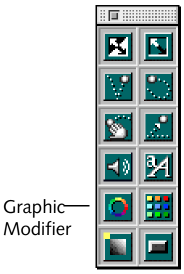
Options
☒ Maintain rate Rate: 15 fps Layer: 1
☐ Cache bitmap
☐ Direct to screen ☐ Convert text to bitmap

Cancel OK

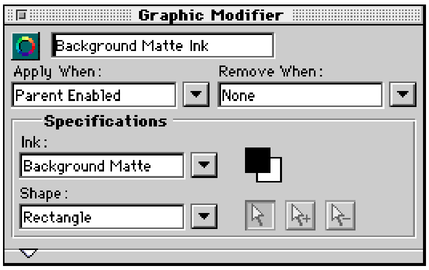
- Click OK to confirm the change and close the dialog.

Let's make the manhole's background transparent.

- Drag a **graphic modifier** from the Effects 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 **Background Matte Ink**. Use the Ink Effect pop-up to select **Background Matte**. The dialog should look like the one shown below.


Graphic Modifier

Background Matte Ink

Apply When: Parent Enabled

Remove When: None

Specifications

Ink: Background Matte

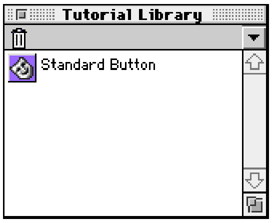
Shape: Rectangle

- 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 file named **Tutorial Library**, found in the **In-Depth** folder. The Tutorial Library appears as its own palette as shown below.

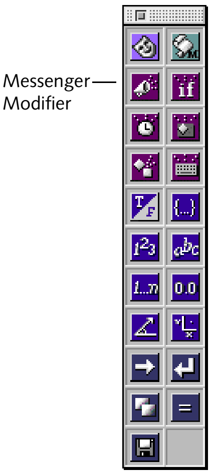


- Drag the behavior named Standard Button from the Tutorial Library palette and drop it on the Manhole Quit.mToon element.

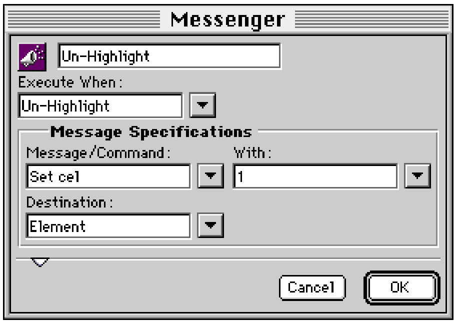
This behavior emulates a button that changes its appearance when it is clicked on. The behavior sends out three author messages: "Highlight," "Un-Highlight," and "Execute." To use this behavior, add it to a graphic that you want to act as a button, configure its two states (highlight/un-highlight) and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

- Drag a **messenger** from the Logic modifier palette and drop it on the Manhole Quit.mToon element.



- Double-click the messenger icon to display its configuration dialog.
- Change the name of the messenger 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.
- Use the Message/Command pop-up to select the **Set Attribute-Set cel** command.
- Select the contents of the With pop-up and enter **1** (one). When received by the mToon, the *Set cel* command and the "with" value 1 will cause the mToon to display its first cel.
- Leave the Destination pop-up set to **Element**, it's default.
- The dialog should look like the one shown below.



Messenger

Un-Highlight

Execute When: Un-Highlight

Message Specifications

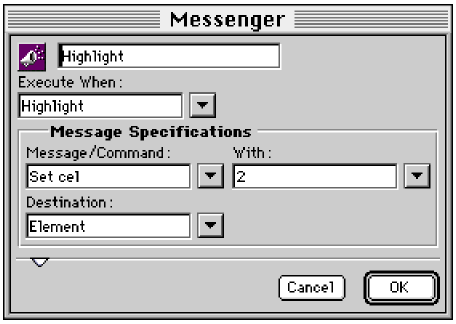
Message/Command: Set cel

With: 1

Destination: Element

- Click OK to dismiss the dialog.
- Now place another messenger modifier on the manhole element.
- Double-click the new messenger icon to display its configuration dialog.
- Change the name of the messenger 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.
- Use the Message/Command pop-up to select the **Set Attribute-Set cel** command.
- Select the contents of the With pop-up and enter **2**. When received by the mToon, the *Set cel* command and the "with" value 2 will cause the mToon to display its second cel.
- Leave the Destination pop-up set to **Element**, it's default.

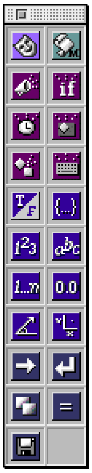
- The dialog should look like the one shown below.



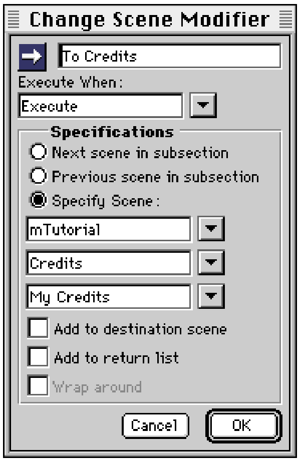
- Click OK to dismiss the dialog.

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 Logic 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.


Change Scene Modifier

- 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. The dialog should look like the one shown below.



- Click OK to accept the changes and dismiss the Change Scene 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 mPuzzle project!

The Credits Scene

Chapter 9. Learning mTropolis Project

This chapter describes the "Learning mTropolis" project which includes many examples of using mTropolis to create both basic and advanced multimedia titles. Examining the programming used to create the scenes of this project is a good way to continue your exploration of mTropolis.

USING THE LEARNING mTROPOLIS PROJECT

This project can be found in the **Documentation** subfolder of your mTropolis installation.

To examine this project, start mTropolis then select **Open** from the File menu. A standard file selection dialog appears. Use this dialog to select the **Learning mTropolis** project found in the **Documentation** subfolder of the folder in which mTropolis is installed. Click "**Open**". The Learning mTropolis project's Layout window appears.

To run the project from its beginning, press **⌘-T** to enter runtime mode. The project begins playing from its first scene (Figure 9.1). Click anywhere to display the main menu (Figure 9.2).

To examine the programming for any of the scenes in the Learning mTropolis project, press **⌘-.** (**Command-period**). mTropolis switches to edit mode with the current scene displayed in the Layout window.

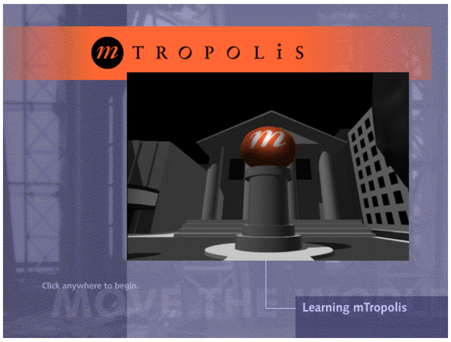


Figure 9.1 The Learning mTropolis Startup Screen

After you have examined a scene in edit mode, you can continue viewing the project from that scene. To run the project from the scene currently displayed in the Layout window, press **⌘-Y** to enter runtime mode.

THE MAIN MENU

The Learning mTropolis Main Menu (Figure 9.2) shows four options at the bottom of the screen. Click on an option to go to that section of the project. These sections are described below.

Authoring Demonstration

The "Authoring Demonstration" section of the project presents a QuickTime movie that shows an example of authoring in mTropolis. A slider at the bottom of the screen can be used to jump forward or backward through the movie.

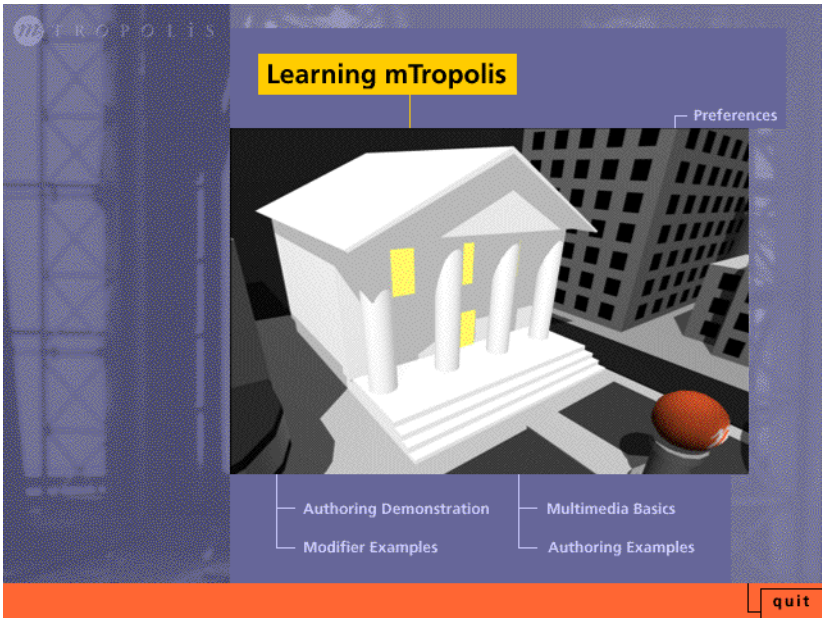


Figure 9.2 The Learning mTropolis Main Menu. Click on one of the four choices at the bottom of the screen.

Modifier Examples

The "Modifer Examples" section shows each of the mTropolis modifer palettes. Click on any of the modifiers to see at least one example of using that modifier.

While viewing any of the modifier examples, you can press **⌘-.** (**Command-period**) to enter runtime mode and see the programming for that example. Press **⌘-Y** to return to runtime mode where you left off.

Multimedia Basics

The scenes in this section illustrate how common multimedia features can be programmed in mTropolis. Click on any of the example names (e.g., "Button Gallery") to see the basic example scenes.

Many of the objects in these scenes can be used as "clip programming" in your own mTropolis projects. For example, buttons from the "Button Gallery" scenes can be cut and pasted into your own projects.

While viewing any of the multimedia basics examples, you can press **⌘-.** (**Command-period**) to enter runtime mode and see the programming for that example. Press **⌘-Y** to return to runtime mode where you left off.

Authoring Examples
The scenes in the Authoring Examples section contain more elaborate examples of mTropolis programming as described below.

- Autonomous Behaviors: This example shows how mTropolis programming is independent of the media used in an object. Clicking the "brain" icon in the upper right corner of the scene causes the behaviors of the bug and butterfly to be switched.
- Character Interaction: This example uses a collision detection modifier to detect when the user-controlled frog contacts a butterfly. The frog "eats" the butterflies when they are hit. Keyboard messengers are used to detect the user keypresses that control the frog.
- Multi-node QTVR: This example shows a multi-node QuickTime VR panorama movie that can be navigated with either the standard QuickTime VR mouse motions (i.e., by dragging the mouse over the movie to pan and pressing control/option to zoom in/out), or by clicking on the "node map" shown below the movie. mTropolis sound elements have been used to add more life the QuickTime VR movie. The sound panning modifier is used to pan the sounds as the user pans around each node.
- mPuzzle: This is a finished version of the puzzle tutorial described in Chapter 8, "In-Depth Tutorial—mPuzzle".

While viewing any of the authoring examples, you can press **⌘-.** (**Command-period**) to enter runtime mode and see the programming for that example. Press **⌘-Y** to return to runtime mode where you left off.

The Main Menu

Index

A
advanced tutorial 8.65
alias palette 5.40
aliases 5.40
asset palette 2.12
At First Cel message 6.48
author messages 6.48

B
basics of mTropolis 4.23
behaviors 4.31, 5.39
  switchability 5.40

C
capabilities 3.18
children 4.30
Close Project command 6.49
cocktail party 4.27
collaboration between artists and programmers 3.17
collision detection 4.24
commands 6.48
components 5.35
  reusing 4.27
containment hierarchy 4.30
  rules of 4.33
conventions
  documentation 1.9
conversation 4.27

D
data
  sending with messages 6.47
debugging facilities 2.14
design
  changes 3.19
Destination pop-up menu 6.42
documentation
  conventions 1.9
  roadmap 1.7
dragging and dropping 4.23

E
edit mode 4.25
Element Info dialog box 4.25
elements 4.23
  activating 6.41
  and media 5.35
  configuring 4.25
  graphic 5.35
  in context 5.35
  in detail 5.37
  layer order of 5.38
  sound 5.35
  text 5.35
encapsulation 3.20
environment messages 6.48
examples 9.105

G
gradient modifier 5.39
graphic elements 5.35

H
hierarchy 4.30

I
inheritance 3.20, 4.30

interface overview 2.11

L

layer order 5.38
layers view 2.13
layout view 2.13
  positioning elements 5.35
Learning mTropolis project 9.105
libraries 2.14

M

media
  and elements 5.35
media objects 4.23
  customizing 4.25
Message Log window 2.14
Message/Command pop-up menu 6.42
messages 3.18
  author 6.48
  broadcasting 4.32
  environment 6.48
  flow of 4.28
  sending 4.28
  sending data with 6.47
  targeting 4.33
  types of 6.47
messaging 4.27, 6.41
  and the containment hierarchy 4.32
  as conversation 4.27
  basics 4.27
  benefits of 4.28
  defined 3.17
messengers
  timer 6.42
  using to build logic 6.41
mFactory Object Model 3.21
modifier palettes 2.12
modifiers 4.23, 4.25
  activating 6.41
  configuring 4.26
  messenger 6.41
  using 5.39
MOM 3.21
Motion Started message 6.48
mouse click 5.39
Mouse Down message 6.48
mTropolis
  authoring 4.24
  basics 4.23
  interface 2.11
  structure 4.30

O

Object Manipulation 2.11
object-oriented design 3.17
objects
  capabilities of 3.18
  defined 3.17
  encapsulation 3.20
  inheritance 3.20
  manipulating 2.11
  media 4.23
  properties of 3.18
  publishing 3.20
  versus procedures 3.18

P

palettes 2.12
parents 4.30
Play command 6.47, 6.49
procedural model 3.19
productivity benefits or object-oriented systems 3.20
programmer, happy 3.21
programming
  visual 4.23
projects 5.36
  defined 2.11
properties 3.18, 5.39
prototyping 4.29

puzzle tutorial 8.65

Q

quickstart tutorial 7.51

R

rapid prototyping 4.29
real-world systems 4.27
reusability 3.19
roadmap
  documentation 1.7
runtime mode 4.25

S

Scene Ended message 6.48
scenes 5.37
sections 5.36
shared scenes 5.36
siblings 4.30
software objects 3.17, 4.23
sound elements 5.35
stand-alone title 4.23
Stop command 6.49
structure
  in mTropolis 4.30
structure view 2.13
subsections 5.36
supermodifiers 4.31
switchable behaviors 5.40

T

targets
  of messages 4.33
text elements 5.35
timer messenger 6.42
title
  stand-alone 4.23
tool palettes 2.12
tutorial
  advanced 8.65
  quickstart 7.51
types of messages 6.47

U

user interaction 4.27, 4.29

V

variables 4.26
vector motion 4.26
views 2.13
visual programming 4.23

W

When pop-up menu 6.42
With pop-up menu 6.42