Developer Guide

MetrOpolis™

Version 2.0

mTropolis Developer Guide

Email: mtropolis@quark.com

Web: http://www.quark.com

Introduction

Introduction

The *mTropolis Developer Guide* is your best starting point for learning how to use the mTropolis™ authoring system. The *Developer Guide* provides detailed coverage of mTropolis concepts, tutorials, and examples — everything you'll need to build a foundation of knowledge to apply mTropolis quickly and effectively.

This Introduction provides information on installing and starting mTropolis, followed by a "roadmap" of resources that you can use to learn more about mTropolis in ways that suit your individual learning style. If you're new to mTropolis, please read this section thoroughly before proceeding. The time you spend here learning what resources are available could greatly accelerate and streamline your mTropolis learning experience.

Welcome!

Welcome to mTropolis. mTropolis is a visual, object-oriented authoring system for creating networked, interactive, multimedia applications. Designed for the demanding requirements of consumer CD-ROM developers, mTropolis is now available for mainstream multimedia authoring. mTropolis combines a rich set of features and effects, a time-saving visual interface, and a behavior-oriented programming system to enable rapid development of high-performance multimedia projects. Whether you're creating presentations, interactive product brochures, entertainment or educational CD-ROMs, computer-based training, or some other category of new media application, mTropolis will accelerate your development process and ease collaboration with other development team members. Projects created in mTropolis may be distributed on CD-ROM, DVD, or the Internet, and will run on Microsoft Windows, Mac OS, Netscape Navigator, and Internet Explorer platforms.

Installing mTropolis

To install the mTropolis authoring environment, player, and supporting components on your Macintosh system, mount the mTropolis CD-ROM and follow the instructions in the "Read Me First!" file at the top level of the disc.

To install the mTropolis player, mPire™ plug-in, and other components on your Windows system, mount the mTropolis CD-ROM and follow the instructions in the "README.WRI" file at the top level of the disc.

Please read the "Release Notes 2.0" file installed with mTropolis for late-breaking information about this release.

Starting mTropolis

To start mTropolis, double-click the mTropolis icon. The first time mTropolis is launched, you will be asked for your name, organization, and registration number. Please enter this information to personalize and enable your copy of mTropolis.

At regular intervals, mTropolis will check to see if another copy of the software with the same registration number is running elsewhere on the network. If another copy with the same registration number is detected, an alert will appear and mTropolis will quit. To avoid this inconvenience, please ensure that your copy of mTropolis is installed and run on only one machine at any time, as specified in your *mTropolis End User License Agreement*.

The default installation of mTropolis includes the mTropolis Info interactive title, which will appear every time you start mTropolis. To disable mTropolis Info, simply remove this file from the **Startup** subfolder of your **mPlugins** folder.

Learning mTropolis

Since we all learn in different ways, and since different types of information are best presented in different forms, mTropolis includes instructional and reference information in a variety of formats:

Print Documentation

If you learn best by reading, refer to the following components of mTropolis documentation. These four manuals are provided on-line in PDF format for viewing with the Adobe Acrobat Reader:

- The *mTropolis Developer Guide* (this manual) covers both general object-oriented and mTropolis-specific concepts — including the mTropolis interface, elements, modifiers, behaviors, and messaging. The *Developer Guide* also provides step-by-step instructions for the hands-on tutorials, and brief coverage of some of the source examples mentioned below.
- The *mTropolis Reference Guide* provides detailed coverage of the mTropolis authoring environment — including all menu items, windows, palettes, and other tools — as well as full documentation of all features available for use in your authored projects.
- The *mTropolis Quick Reference* summarizes all of the features — modifiers, system messages and commands, object attributes, Miniscript syntax, and Miniscript functions — available for use in your authored projects. The *Quick Reference* also summarizes all of the mouse/keyboard shortcuts in mTropolis (except for menu shortcuts, which simply appear in the mTropolis menus).
- The *mFactory Object Model (MOM) Reference Guide* is the starting point for C programmers who wish to extend mTropolis by creating new modifiers, tools, or services using the MOM SDK. The MOM SDK is not installed by default; you must choose it via a Custom Install or copy the SDK files from the "Goodies" folder, in the "mTropolis v2.0" folder on the CD-ROM.

Interactive Documentation

If traditional documentation is too dry for you, turn to the following resources to view and manipulate a wide range of interactive, multimedia examples:

- The "Authoring Demonstrations" section of the *Learning mTropolis* project presents screen-captured, narrated videos of the mTropolis authoring environment in action.
- The "Modifier Examples" section of the *Learning mTropolis* project provides one or more interactive examples, including explanatory text, for each of the mTropolis modifiers.
- The "Multimedia Basics" section of the *Learning mTropolis* project contains a range of interactive examples showing common authoring tasks — navigation, cursor control, string manipulation, etc.
- The "Authoring Examples" section of the *Learning mTropolis* project demonstrates several finished, real-world examples — ranging from an interactive product catalog to an electric-motor assembly simulation — of mTropolis in action.
- The *mPack Guide* title, accessible from the "Tools" menu if mPacks tools have been installed, provides interactive documentation and examples for all of the components contained in the standard mTropolis v2.0 mPack libraries.

For more information on mPacks tools, refer to Chapter 9. For more information on the *Learning mTropolis* project, refer to Chapter 10. Note that the *Learning mTropolis* project is not installed by default; you must choose it via a Custom Install or copy it from the "Documentation & Examples" folder, in the "mTropolis v2.0" folder on the CD-ROM.

Hands-On Tutorials

If you learn best by rolling up your sleeves and creating something from the ground up, refer to the following hands-on tutorials:

- The *QuickStart Tutorial* (see Chapter 6) will rapidly acquaint you with the basics of working in mTropolis.

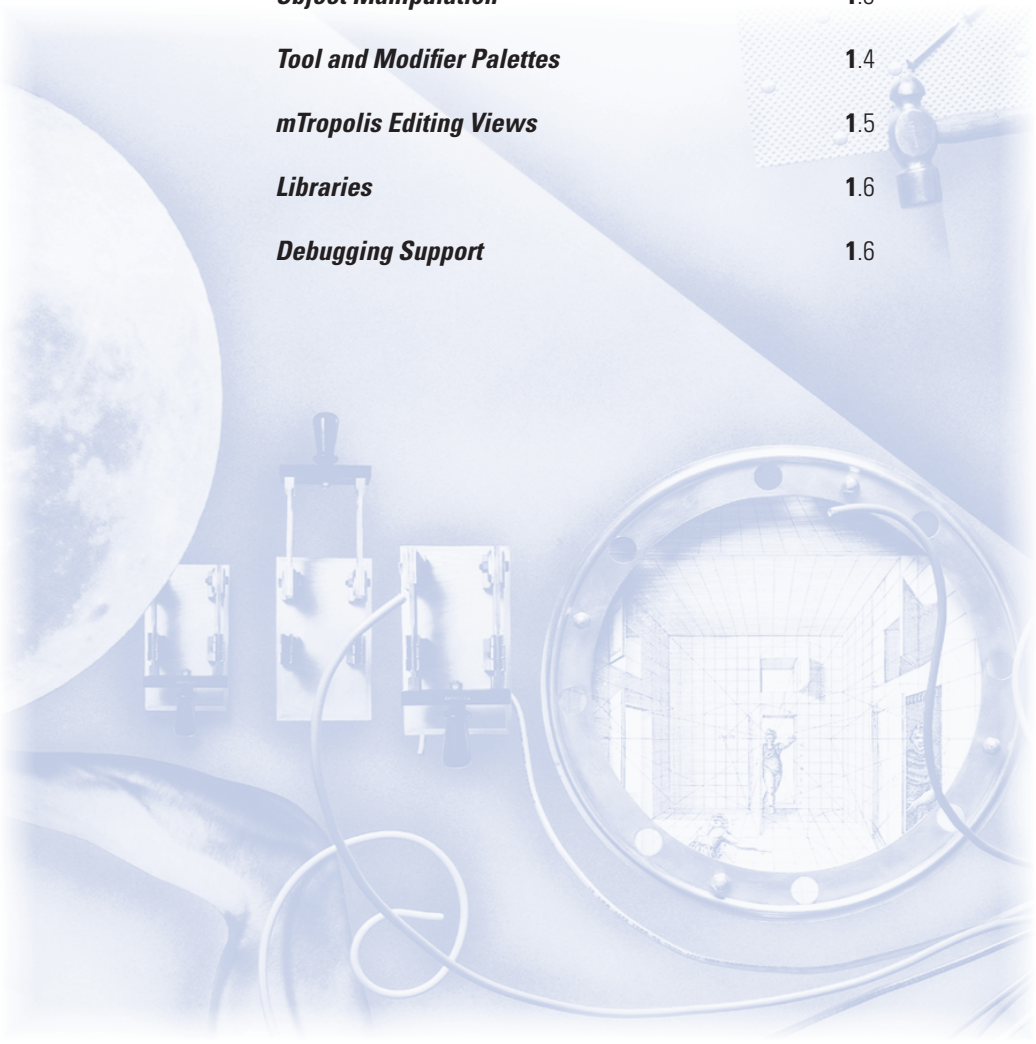
- The *In-Depth Tutorial* (see Chapter 7) provides more extensive coverage of all of the key concepts of mTropolis authoring.
- The *Network Chat Tutorial* (see Chapter 8) assumes familiarity with basic mTropolis authoring concepts and focuses on the new network messaging capabilities of mTropolis v2.0.

Source Examples

mTropolis v2.0 includes a large array of authoring examples provided in source form. If you learn best by dissecting and studying working examples, refer to the following resources:

- The *Learning mTropolis* project (see Chapter 10) is provided in source form, so you may explore and deconstruct any of the authoring behind the Modifier Examples, Multimedia Basics, or Authoring Examples.
- The *mPacks* (see Chapter 9) are intended for easy re-use in your projects, but also serve as excellent source examples of reusable authored components.
- The "Wizard Authoring Example" (see Chapter 11) is a great resource for learning how to author mTropolis wizards—projects that automate the creation of other projects.
- The "mServer Console" and "mChat" projects, provided in the "mPire Net Messaging" subfolder of your "Goodies" folder, are advanced examples of how to build multiuser, networked applications in mTropolis using network messaging.
- The MOM SDK contains a wide range of C-code source examples for various types of custom modifiers.

1 mTropolis Interface



1 mTropolis Interface

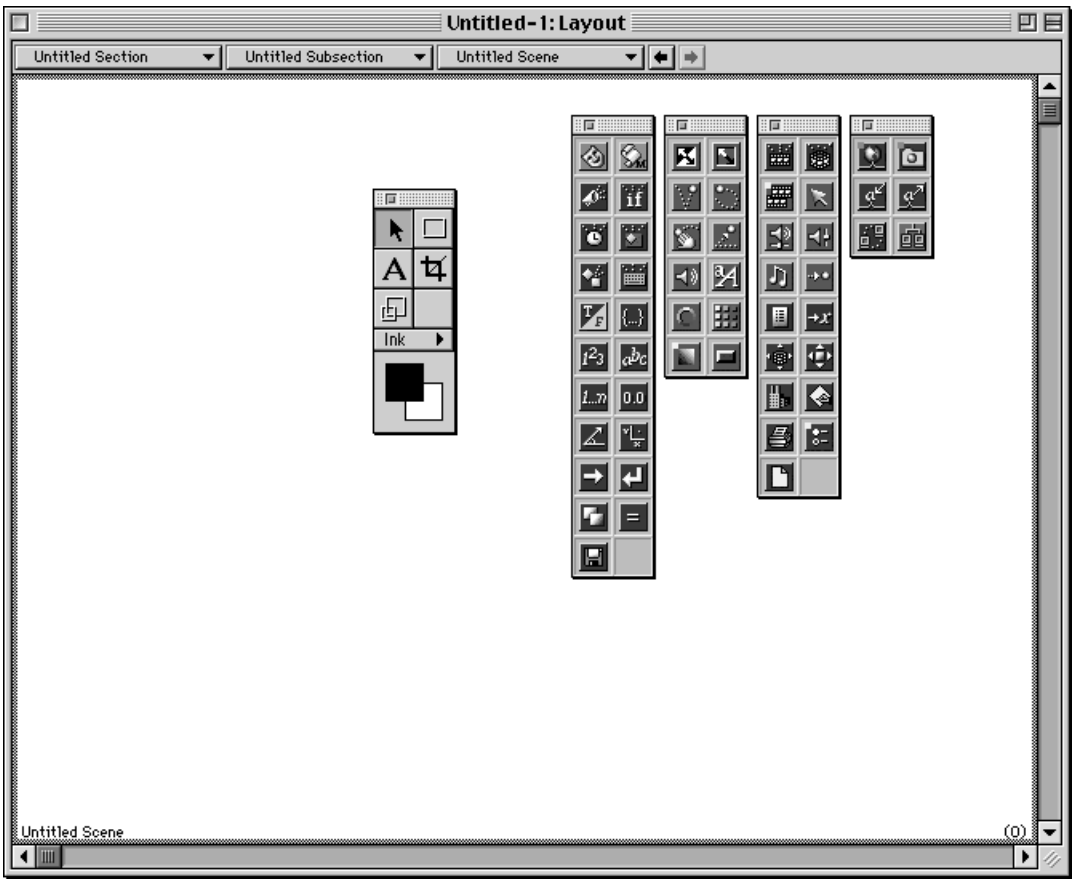
This chapter provides an overview of the mTropolis authoring system's graphical interface.

Layout Window

When you launch mTropolis, a Layout window appears containing scroll bars and pop-up menus for quick navigation around a work in progress. 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 stand-alone title version of your project), can be accomplished with direct manipulation of objects. For example, the internal operations of mTropolis objects can be configured by double-clicking them. Portions of a mTropolis project can be rearranged or even moved to a different project by dragging and dropping. The menus in mTropolis provide access to specialized operations such as



The Layout window with Tool and Modifier palettes

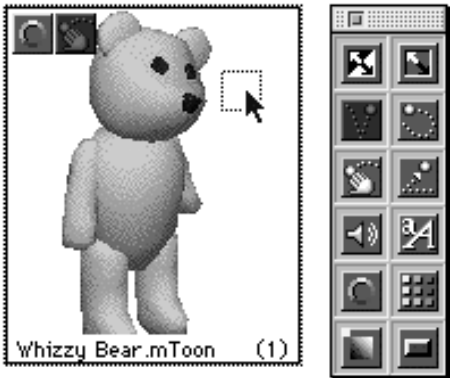
opening files or linking media assets, as well as many of the operations that are also available through direct manipulation.

mTropolis has been designed with careful attention to end user interface consistency. If an object on the screen can be clicked, 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 box that conveys useful information about its internal workings. Generally, this dialog box can be used to reconfigure or customize an object as well.

This consistency means that you can experiment with mTropolis, incrementally applying the knowledge you gain to new tasks. You can concentrate on learning techniques, not interface details, and the resulting 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 modifier objects that you can drag and drop in the process of building a title. When a modifier is dropped on an element, the element inherits the properties of the modifier.



Modifiers can be dragged and dropped onto elements

The **Tool** palette offers the tools most frequently used in building and manipulating the constituent pieces of a mTropolis title. It includes a Selection tool, a Graphic tool, a Text tool, a Crop tool, and a Parent/child tool.

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,* "Palettes."

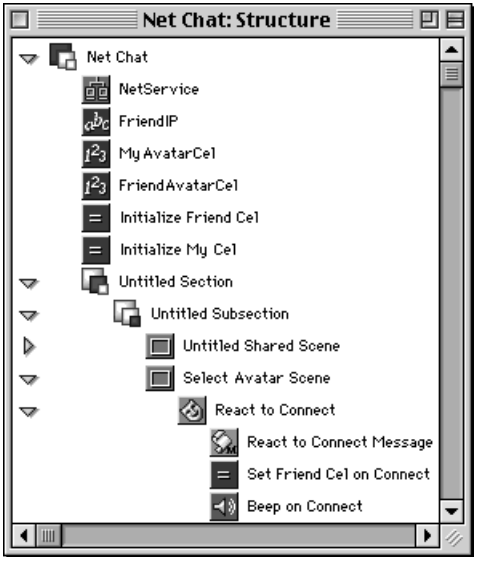
mTropolis Editing Views

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 allows you to directly manipulate the graphical aspects of your title in WYSIWYG fashion — arranging objects, altering their appearance, and doing basic programming by dragging and dropping modifiers. The layout view is described in detail in Chapter 9 of the *mTropolis Reference Guide*, "Layout Window."

Structure View

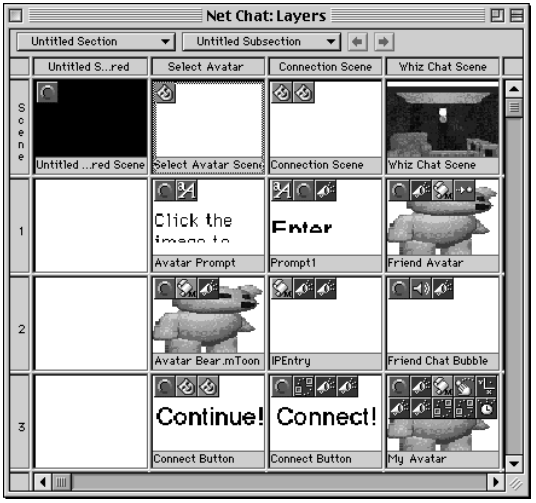
The structure view presents an expandable/collapsible outline that shows the hierarchy of components within your project. This view shows which objects are contained within others and allows you to rearrange the hierarchy by dragging and dropping. The structure view is described in detail in Chapter 8 of the *mTropolis Reference Guide*, "Structure Window."



The structure window

Layers View

The layers view shows all of the 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 of the project. 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 layers window

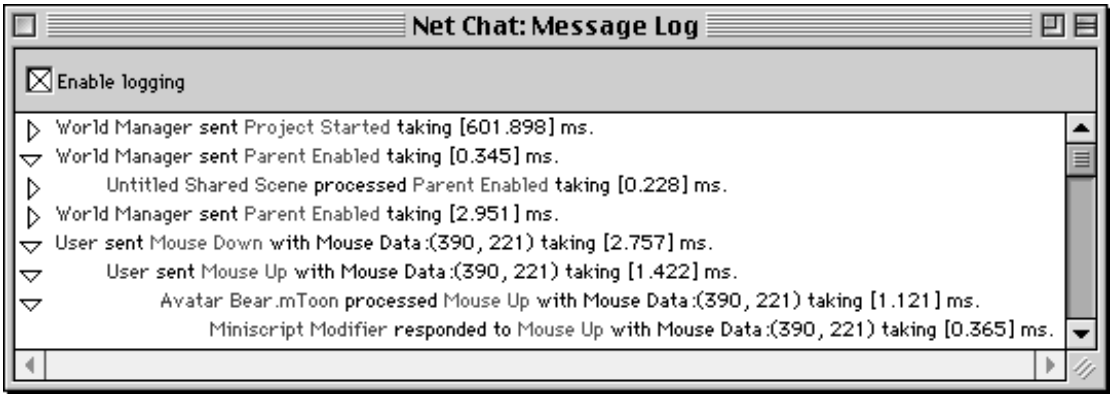
Libraries

Libraries offer the powerful ability to manage large projects, enabling collaboration and promoting reuse of project work. mTropolis libraries can contain any mixture of objects, including media assets, modifiers, or entire structures 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 objects to and from libraries. Libraries are described in detail in the "New, Open, and Save for mTropolis Libraries" section of Chapter 1 of the *mTropolis Reference Guide*. A suite of useful libraries, called "mPacks," that contain useful bits of mTropolis programming have been included with mTropolis.

A mTropolis Library

Debugging Support

mTropolis offers thorough debugging support. The primary mTropolis debugging utility is called the Message Log window. Because mTropolis is an object-oriented system, the flow of a mTropolis project is determined by messages exchanged in conversation between objects. The Message Log provides a complete, contextual history of every message, including those that are end user-generated, that was exchanged between specified objects. The Message Log is described in detail in the "Message Log Window" section of Chapter 7 of the *mTropolis Reference Guide*.



The Message Log window

2. Object-Oriented Design

2. 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, without laboriously scripting every potential action, step by step, or using a complicated programming language.

mTropolis does not impose any particular development metaphor on its users. Artists and programmers can create, manipulate, and evolve any combination of media and logic without being constrained by a specific, 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, end user-generated events. This characteristic of mTropolis titles allows them to more closely model the real world, with many events of different types driving the experience in truly novel directions.

Objects and Messaging: a Definition

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

Software objects

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 are the software counterparts of real-world objects. They model the real-world objects and the 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 attributes. Attributes are intrinsic features of a real-world object, like its height, weight, or color. An interesting example is a clock, which could be described as having the attribute of time.

A software object models a real-world object by duplicating as many of the real-world object's capabilities and attributes as appropriate. For example, a software object modeling a clock might present an image of a clock (with color, height, and weight attributes), as well as have code to keep the time (a clock's time attribute).

An important point about software objects is that they can represent capabilities or attributes 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 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 multi-faced clock object. Now, consider that you could add an object with the abstract capability of movement to create a flying, multi-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 attributes. 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.

Although software objects have the same benefits of simplicity and easy interconnection as do Lego blocks, they 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 versus Procedures

An analogy may help to make the design issue more apparent. Imagine a program designer who wants to model the activities of taxi cabs in a city. Using a procedural programming tool that requires scripting, the designer creates a map of the city and then must define, in advance, all the possible paths that the taxis will be allowed to follow as well as all the conditions under which the taxi 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 professional, 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, then measure their distance from the hotel, 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 greater realism..

There are other significant productivity advantages to object-oriented approaches.

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.

Reusability
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 simulation decides to include trucks in the simulation, the behavior of the taxi can be duplicated, slightly altered to suit that of a truck, and added to the simulation.

Inheritance
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 other approachs.

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 heirs acquire 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.

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. 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 would 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 lower-level development environments such as C++. In mTropolis, the author combines and connects components in conversation, to create sophisticated entities that themselves are building blocks for titles. However, mTropolis does not allow the direct modification of component internals at the author level as do C++ or SmallTalk.

The good news is that mTropolis provides a comprehensive set of programming interfaces (APIs) called the mFactory Object Model (MOM). MOM allows programmers to extend mTropolis with new components, or to modify or over-ride existing components. (See the online documentation for MOM for additional information.)

In addition, special mTropolis projects, called *tools*, can be created using mTropolis. Tools are projects that run in the mTropolis editing environment and examine or manipulate other projects. Tools are an easy way to extend the capabilities of mTropolis. A number of tools, including the Object Watcher and Memory Watcher, are automatically installed with mTropolis. These tools can be accessed from the **Tools** menu. For more information on tools see Chapter 6, "Tools Menu", in the *mTropolis Reference Guide* and Chapter 11, "Wizard Authoring Example", in the manual.

3. mTropolis Basics

3. mTropolis Basics

In this chapter, the 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 Quark 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 racetrack 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. 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, occurs.

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 main 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 (position, size, layer, etc.). Modifiers are programming components that can be added to elements to endow them with new capabilities such as 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 are 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.

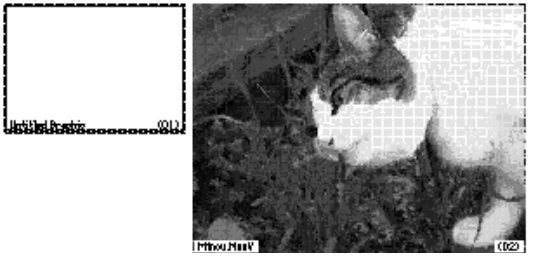
mTropolis projects can be executed and debugged inside the mTropolis development

environment. To do so, the author switches from the default editing mode (in which components are manipulated and configured) to a run-time 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 Run-time Modes" in Chapter 1 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 toward 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.

In the editing environment, elements display thumbnail images of the external media files to which they are linked. For example, the first frame of a digital video is shown within the boundaries of the element. When the author switches from editing mode to run-time mode, the digital video plays on the screen within the element boundaries.



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

The author can configure the operation of the element through its **Element Info** dialog box. This dialog box can be displayed by double-clicking the element.

*The **Element Info** dialog box associated with a video element*

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

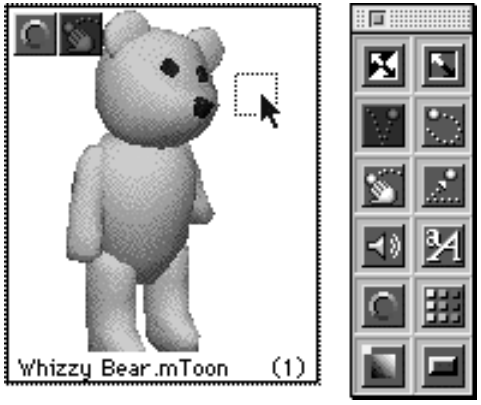
The element's configuration does not directly alter the external media file. Instead, it alters the appearance and behavior of the media at run-time, such as its representation on the screen or its volume level. The media file linked to the element remains unchanged.

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 be entered by end-users at run-time.

Modifiers: Customizing media objects

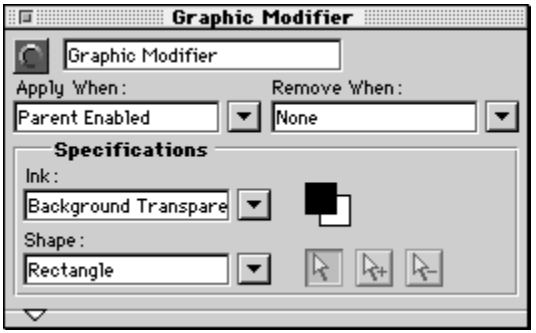
Element components can inherit capabilities from modifiers. In the mTropolis editing environment, this process is simple and direct: An element acquires new capabilities or properties when modifiers are dragged and dropped on them.



Dragging a modifier from a palette and dropping it on a graphic element

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 the element remains unchanged.

Like elements, modifiers can be configured through a dialog box. This dialog box can be displayed by double-clicking the modifier. Each modifier's dialog box is specific to its particular capabilities or properties. The dialog box for configuring a Graphic modifier controls the particular properties that an element would inherit from this modifier — in this case, the appearance of its media. This dialog box also contains the message pop-up menus used to configure the modifier's messaging operation. These functions are discussed in more detail in Chapter 5, "Messaging."



*The **Graphic Modifier** configuration dialog box*

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. A 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, "Modifier Reference" of the *mTropolis Reference Guide*.

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 full-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 a conversation. In a conversation, the participants interact dynamically, changing their responses according to feedback from others. More critically, in object-oriented systems the end 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 end 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 2. 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.

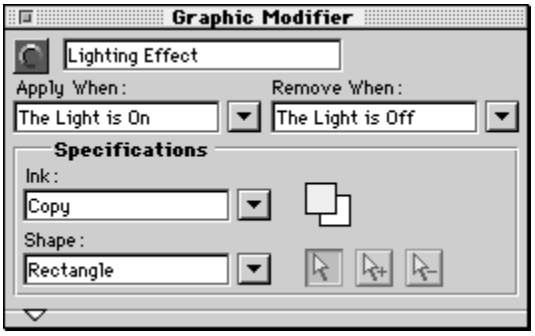
This faithfulness to real-world systems (or really, any system that is internally consistent) means that you can model them more naturally and much more quickly.

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 a component's on-screen representation to display its configuration dialog box. Sending

a message to a mTropolis component is an alternate, dynamic vehicle for invoking these capabilities during run-time. For example, a Graphic modifier component can be configured to lighten its element's media when it receives a message called "The light is on."



*A **Graphic Modifier** dialog box for switching a light on and off*

Messages are signals or requests. These signals are acted on by components configured to respond to them. If a component receives a message for which it is not configured to respond, it merely relays the message to the components that it contains. For example, an element might contain a Graphic modifier. 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.

Messages can be generated by the end user during run-time (end user mouse actions such as mouse clicks generate messages), by the mTropolis environment (mTropolis sends a Scene Started message to components at the start of each scene), and by components themselves (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 (a "The power is off" message) to control the flow of events in a title.

mTropolis provides very powerful but uncomplicated utilities 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 detected 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, he is 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 on 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.

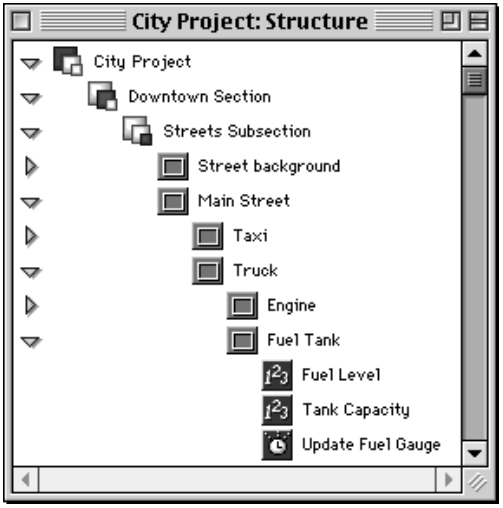
For example, an author could duplicate the balloon component ten times and place the copies in a jet fighter title for comic relief. When a fighter (itself a sophisticated author-defined component) launched, it might generate a "There is wind" message to simulate jet-wash. On receiving the message, the 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 end 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 end user, can be dynamically introduced under title control or end user control. Consider a game like SimCity in which the end 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 run-time, the end user could introduce as many of those components as the game permitted, and they would dynamically interact with each other and with the end user without any additional programming.

The object-oriented nature of mTropolis allows titles to be designed, implemented and analyzed at the same time, saving enormous development time. Although a complex, event-driven system could be modeled in a procedural or command-based authoring tool, the mTropolis messaging system makes interactions 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.



A sample structure view

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, because 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, such as 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*. 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.

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

Now that you understand what a container is, you should also understand that a modifier is not a container. Modifiers are placed in containers, where they do work on behalf of the container, such as 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, 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 media, (such as, pictures, sounds, or animations).

The mTropolis project itself is a structural container that contains section, subsection, scene, 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-drop packages.

Another special feature of behaviors is that they can contain other behaviors within them, and those child behaviors can be parents of other behaviors, to an arbitrary depth. This feature permits the creation of an extremely sophisticated behavior at the top of a container hierarchy of behaviors.

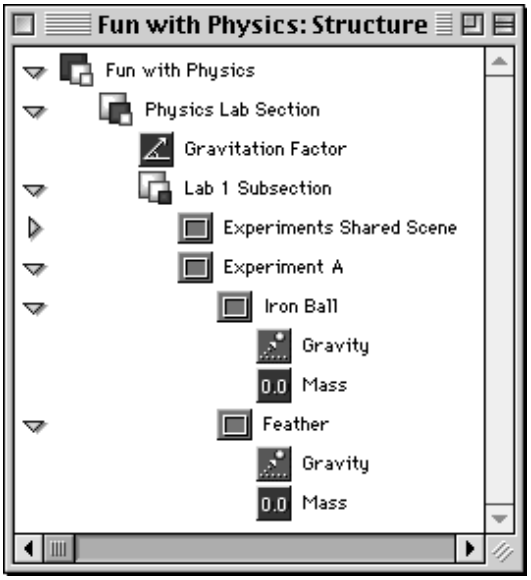
Consider the behavior of a variety of household pests: They avoid light. But cockroaches also run from light only under certain conditions. A light avoidance behavior could be contained within a cockroach behavior, selectively switched on under certain conditions, to quickly, simply, and easily model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



A message sent to the Physics Lab section cascades and relays from component to component

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. In a language like C++, you would have to send a message directly to the gravity component, which means you must remember exactly where the component is. In mTropolis, you have much more flexibility in messaging, thanks to the containment hierarchy.

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 respond.

The advantage of broadcasting is that you can cause global changes without laboriously specifying each and every component that needs to act on a message. In the example above, where gravity modifiers are 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 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 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.

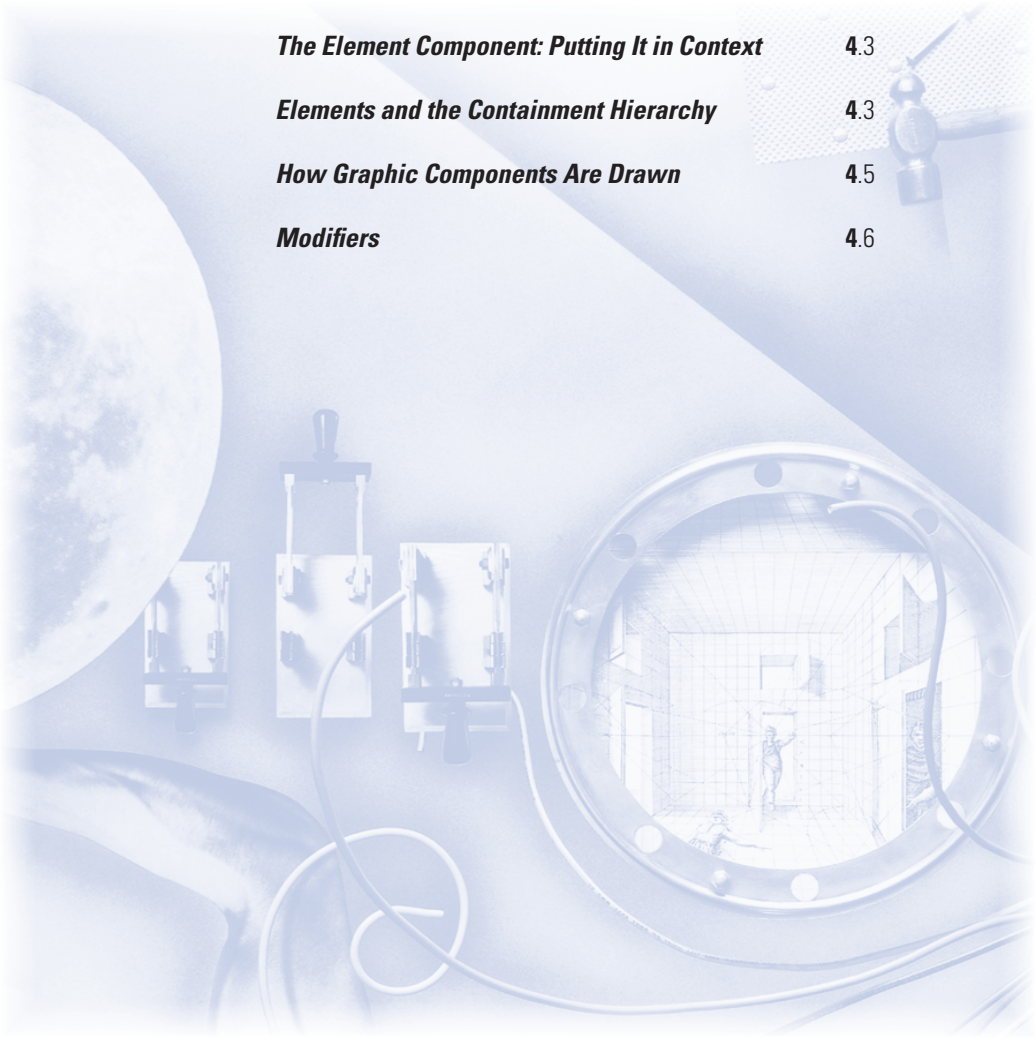
Basic rules of the mTropolis containment hierarchy

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.
- Only scenes and elements can contain elements.

- All mTropolis objects can be the target of a message, but a modifier (or the system/end 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.

4. mTropolis Components



4. mTropolis Components

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

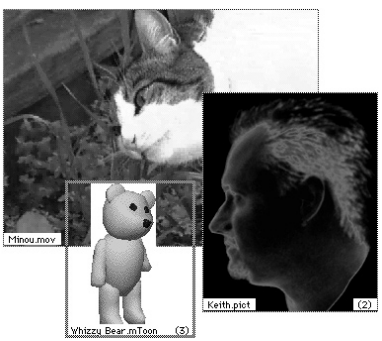
The Element Component: Putting It in Context

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 (for example, the element's position) and controlling the appearance of media they contain (that is, which frame of an animation is currently displayed).

This section gives an overview 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.



Graphic elements linked to a QuickTime movie, mToon, and PICT file

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

Graphic elements
Graphic elements can be linked to images (such as PICTs), digital video (including QuickTime movies and QuickTime VR panoramas), and animations (in the mToon proprietary animation format).

Sound elements
Sound elements can be linked to sound files (such as 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 editable so that end users can modify their contents in run-time.

Elements and the Containment Hierarchy

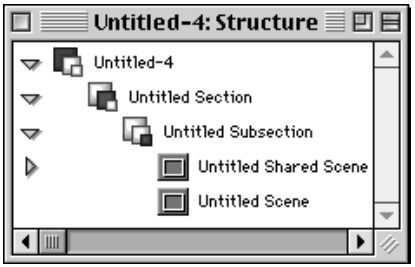
Elements are always contained by other components, either other elements or scene components. When a new project is opened, mTropolis provides a project component (Untitled-1), a section component (Untitled Section), a subsection component (Untitled Subsection), a shared scene component (Untitled Shared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

Project components

A project component is a structural container that holds an entire title within it. The immediate children of a project are sections. A project can only contain sections and modifiers. Projects cannot contain other projects.

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 CD–ROM 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.



Structure view of a new mTropolis project

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 CD–ROM 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 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 sibling 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 background music track, for example, might be placed in the shared scene. Background images or buttons 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.

The shared scene itself is always drawn behind the current scene, but elements on the shared scene can be layered above elements on the current scene.

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 an end user. Scenes are the highest level structural components that are visible to end users — every element that an end user can see or manipulate is contained within a scene.

A scene presents a microcosm (for example, 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. Scenes are, in fact, just a special type of graphical element.

Element components

An element component is the workhorse of mTropolis. Elements can be linked to media such as pictures, sounds, animations, and video. Elements can also contain modifiers (which help to bring the raw media they contain to life) or other child elements.

Consider a scene representing an African plain. The title author would use visual 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. When the author drops that component on the vulture element it 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 could then detect when the vulture elements collided with them.

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, compound vulture element will move along the path as a unit.

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.

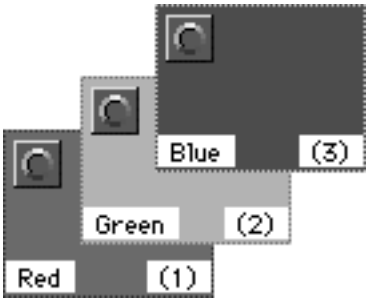
Elements contained by a scene are drawn by default on top of the scene. Scenes are drawn by default on top of the shared scene that services them.

Elements are automatically assigned a layer when they are added to a scene. The layer of new elements increases as they are added to the scene. Each layer contains only a single

element. The layer order can be changed dynamically during run-time. 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*.

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 views the result of layering one element on top of another.

In this "2.5D" approach, elements always appear above elements of a lower layer and below elements of a higher layer when they are redrawn. Because they are clipped, this approach creates the illusion of a 3D perspective.



The effect of graphic layers

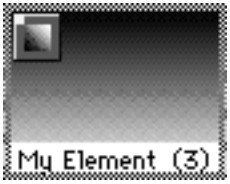
mTropolis provides the option of displaying elements direct to screen, turning an element's draw order off. This feature is useful when you want an element to always draw on top of everything else on screen.

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 seamlessly so they behave just like 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.



The Gradient modifier, placed on a graphic element

While some modifiers have the ability to change the visible characteristics of the elements on 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 (640, 480) is placed on an element, the physical representation of the element does not change, but its content, the value (640, 480), becomes an intrinsic part of the element that contains it.

All modifiers can be configured (their capabilities can be customized) by changing the settings in their dialog boxes. 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 run-time environment have been met. Some messages are generated by mTropolis during run-time 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*.

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 (or 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 "supermodifiers" that provide more complex operations than single modifiers alone.One of the most powerful features of behaviors is that they can be 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 capabilities.

A complete description of the behavior modifier can be found in the "Behavior" section in Chapter 12 of the *mTropolis Reference Guide*.

Aliases

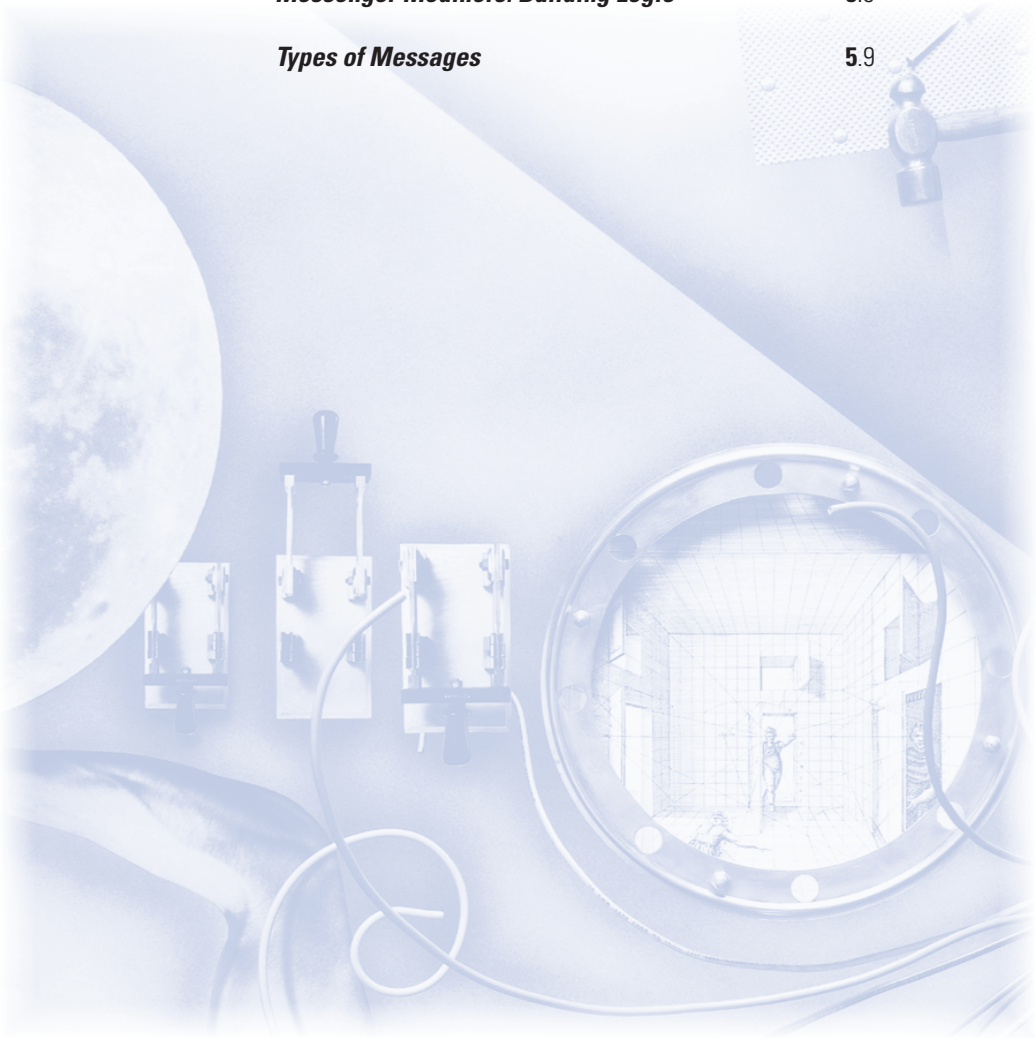
One powerful mTropolis feature is the ability to make a special copy of any modifier (including behavior modifiers), called an alias. Creating an alias makes a master copy of a modifier and places it on mTropolis' **Alias Palette**. Modifiers on the **Alias Palette** can be dragged and dropped onto any element in a project. All copies of an alias placed this way or copied and pasted from another element share the same settings and 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 identical modifiers can be time-consuming. Using aliases of the Graphic modifier in our example would permit the author to make a change to any copy 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 the "Alias Palette" section in Chapter 11 of the *mTropolis Reference Guide*.

5. Messaging



5. Messaging

This chapter focuses on the messaging relationships between components. Chapter 4, “mTropolis Components” outlined the role that components play in the mTropolis authoring environment.

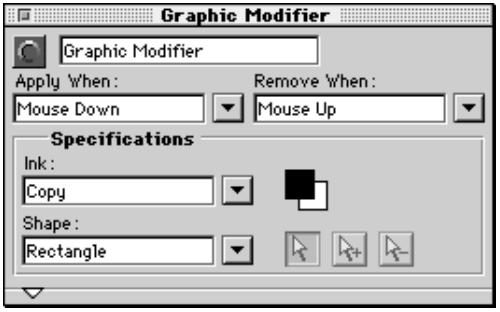
Activating Elements and Modifiers

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 run-time environment itself, such as scene change events or end 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 placed on some arbitrary element, as depicted in the **Graphic Modifier** dialog box below.

In this example, the Graphic modifier listens for a Mouse Down message. When that message is sent to it, it applies its graphic effect. When it receives the Mouse Up message, it returns the element to its default color.



*A **Graphic Modifier** dialog box 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 an end 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.

A typical Messenger dialog box

Messenger Modifiers: Building Logic

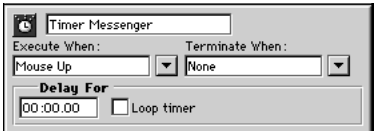
Messenger modifiers (often referred to simply as messengers) are dedicated to sending and receiving messages. Messengers are the components that implement the abstract logic of a mTropolis title. For example, a Timer Messenger listens for a message and delays for a selected period of time. Then it sends another message out.

Messaging

The **Timer Messenger** dialog box shown above and deconstructed below, illustrates the full power of messaging in mTropolis. Through their dialog boxes, 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 Ws" 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 menu, the modifier will activate as it has been configured to do. When the modifier receives the message selected in the **Terminate When** pop-up menu, it will return to an inactive state.



What

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

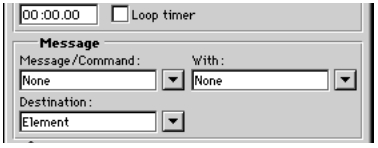
Where

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

With

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

Each of the pop-up menus will be described in turn.



When and what message pop-ups

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

Together, the **When** pop-up menus control the conditions under which the messenger (or any modifier for that matter) will operate.

The first **When** pop-up menu 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 menu selects the message that will return the messenger to an inactive state, just as with any other modifier.

The **What** pop-up menu is distinct from the **When** pop-up menu in that it determines the output of the messenger. The message selected by the **What** pop-up menu 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" and "The Message/Command Pop-Up Menu" sections in Chapter 13 of the *mTropolis Reference Guide* for complete information on these menus.

Where destination pop-up

The **Where** pop-up menu 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

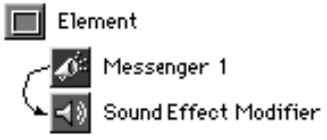
As explained above, the destination for a messenger's output message can be either a specific component, an arbitrary level of the project, or a behavior containment hierarchy. By default, mTropolis automatically handles cascading the message down through the containment hierarchy to the elements, modifiers, 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 Chapter 3. Message destinations are described in detail in the "Destination Option Descriptions" section in Chapter 13 of the *mTropolis Reference Guide*.

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

- The messenger modifier sends a message to the element that contains the messenger. This destination is called the element destination.

The element destination

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

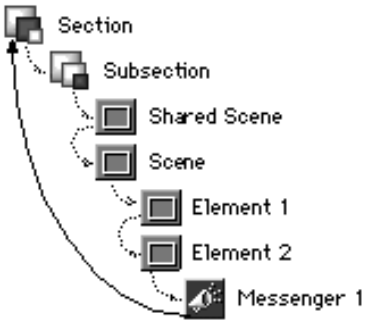


A messenger's sibling destination

- The messenger sends a message to another element (to a sibling of its parent). This pop-up menu is the element's sibling destination.

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 gray lines) to all of the eventual recipients.



A message moving down the containment hierarchy

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 on by modifiers that have specified in their **When** pop-up 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 mTropolis will handle all of the details of cascading the message to the possible recipients.

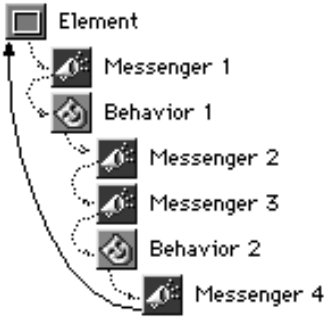
Relative message targeting enables you to target other components by their relation to a modifier, rather than their specific names or positions in the containment hierarchy. For example, you could specify that you want a message sent to a modifier's parent in the containment hierarchy, or its parent's 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 Chapter 14 of the *mTropolis Reference Guide*.

Behavior containment hierarchy and messages

As we mentioned in our discussion of behaviors in Chapter 4, 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, displayed and alterable through the structure view. The following illustrates 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 its siblings, the modifiers contained by Behavior 1. In order for the modifier contained by Behavior 2 to send a message to a modifier on the element (Messenger 1), however, it must target the element.



A message is broadcast to an element containing both a behavior and another element

The next example illustrates how 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 on an element and the modifiers it contains

Behavior window

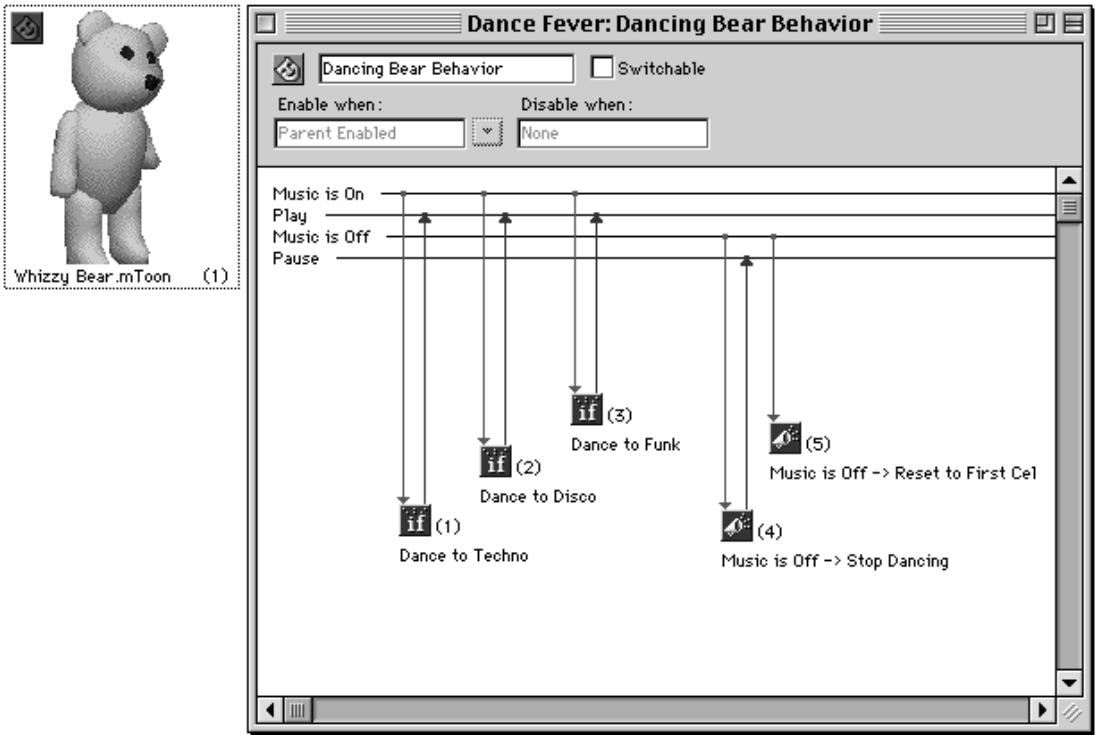
The next page shows a behavior on an element, and the configuration dialog box for the behavior showing the modifiers it contains.

Notice that the behavior dialog box shows the execution order of modifiers (the (1), (2), (3), (4), and (5) indicators next to the icons) and the particular messages that activate them ("Music is On" and "Music is Off"). mTropolis processes messages in the order they are received, and looks for the first modifier that will respond to the message currently being processed.

The "Dancing Bear 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 enable the Whizzy Bear mToon™ to play different animations depending upon what style of music is being played. However, once created, this behavior can be dragged onto other mToon 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** check box in the behavior window enables a behavior to be controlled in this fashion. The **Enable when** pop-up menu specifies the message that will enable the behavior and the **Disable when** pop-up menu 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. It will not allow messages to cascade down to its children.

The ability of behaviors to switch between Enable and Disable is especially powerful when behaviors are nested within other behaviors. As we discussed previously, behaviors can be nested so that they fire activation and deactivation messages downward in a sophisticated cascade. This gating of nested behaviors can be used to model very complex logic in a manageable and reusable manner.



A behavior modifier dialog box

With menu: Sending data

The **With** pop-up menu allows a messenger to pass data along with output by a messenger. For example, information about the current element's screen position, where the end user clicked 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 the **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" section in Chapter 13 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 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 constitutes a demand that the recipient object perform some action, and it cannot be ignored. Command messages are primarily used to control 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 the "Environment Messages," "Author Messages," and "Commands" sections of Chapter 13 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 on it.

Author messages

Author messages are defined by the author by entering the text of a new author message into the **When** pop-up menu 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-up menu, can be controlled by any author message.

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 end 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** pop-up menus as italicized text to distinguish them from other messages.

Commands are different from author/environment messages in the following two respects:

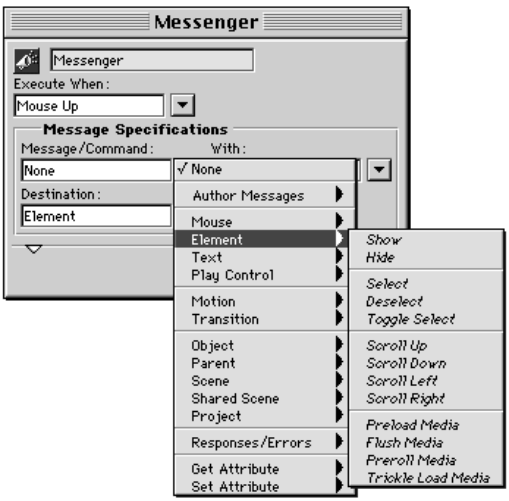
- Commands act directly on elements. Elements do not have to be configured to hear commands, and they respond to them immediately, without interpretation. For example, there are commands that tell a digital video to play or hide, or a still image to show or hide.

  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. *However, commands are never cascaded to children of the destination element.*

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.



*Commands in the **Message/Command** menu's Element section*

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

6. QuickStart Tutorial — A Simple Slideshow

6. QuickStart Tutorial — A Simple Slideshow

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

Getting Started

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 cause the scenes to change based on the end user's mouse actions. The result is a presentation that end users can browse at their own pace.

What you'll need

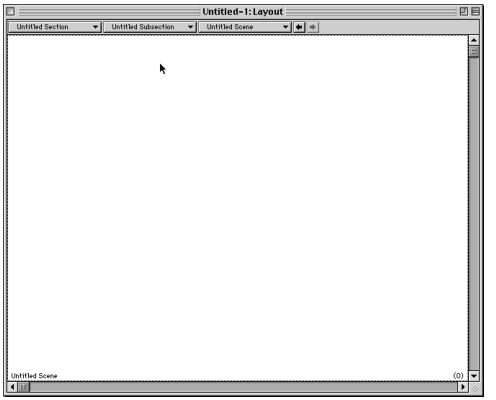
mTropolis must be installed on your machine. Installation instructions can be found in the "Read Me First!" file on the mTropolis CD–ROM.

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, run the installer from the mTropolis CD–ROM, or drag the **Tutorials** folder from the CD to your hard disk.

This project uses 8-bit media. For best performance, make sure your monitor is set to display 256 colors.

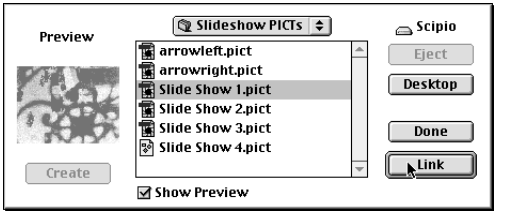
Getting started

1. Launch mTropolis by double-clicking the mTropolis icon. mTropolis appears with a new, untitled Layout window.
2. Click inside the Layout window to choose the Untitled Scene.



Choose the Untitled Scene by clicking inside the Layout window

3. Choose **Link Media ➛ File** from the **File** menu. A dialog box appears.
4. In the file dialog box, choose the file **Slide Show 1.pict** (**Tutorials ➛ Quickstart ➛ Slideshow PICTS**). Click **Link** to link the picture to the scene.



*Click **Link** to link the selected picture to the scene*

5 The Layout window updates to show the new media linked to the scene, and the name of the scene changes to match the name of the media.

The new media is shown in the updated Layout window

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.

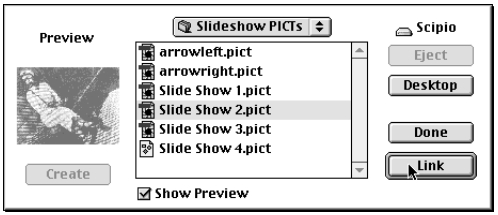
1 Create a new scene by choosing **New Scene** from the Layout window's scene pop-up menu. This is the third pop-up menu button from the left, found at the top of the Layout window. Currently, this button reads Slide Show 1.pict.

*Choose **New Scene** from the Layout window scene pop-up menu*

2 The Layout window changes to show the new, Untitled Scene. The scene is already selected for us so we can link media to it.

3 Choose **Link Media ➛ File** from the **File** menu. A dialog box appears.

4 In the file dialog box, choose 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.



*Choose Slide Show 2.pict from the **Slideshow PICTs** folder, located in the **QuickStart** folder*

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

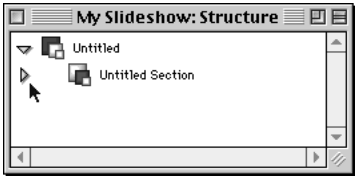
The updated Layout window reflects the new media linked to the scene

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.

1 Choose **Structure Window** from the **View** menu to display the mTropolis Structure window.

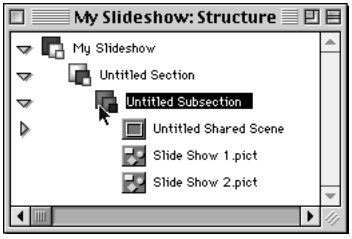
2 The Structure window for the project appears. This window displays a hierarchical view of the project. Components of the project are shown as named icons. Open/close triangles to the left of those icons allow you to reveal or conceal different levels of the project hierarchy.



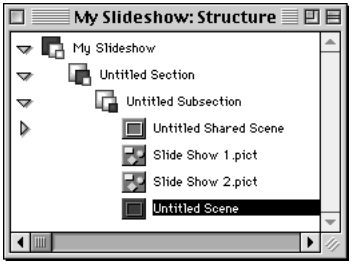
3 Click the triangle next to **Untitled Section** to reveal the Untitled Subsection.

4 Click the triangle next to **Untitled Subsection** to reveal the scenes we are working on.

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

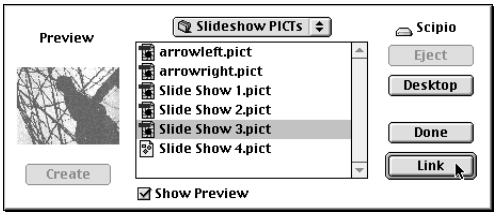


6 Choose **New ➛ Scene** from the **Object** menu to create a new, untitled scene. A new scene appears in the Structure window as shown below.



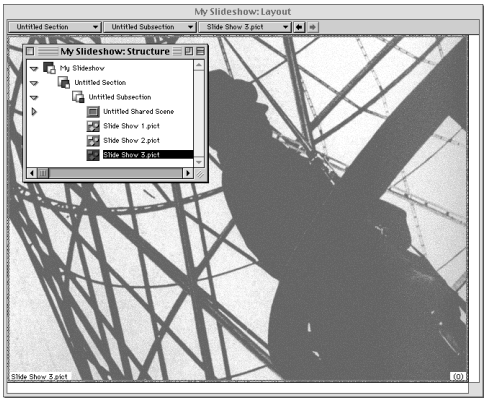
7 The new Untitled Scene is selected and ready to have media linked to it. Choose **Link Media ➛ File** from the **File** menu. A file selection dialog box appears.

8 In the file dialog box, choose 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.



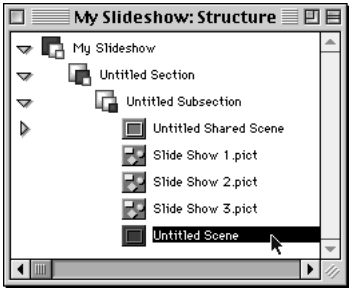
*Choose Slide Show 3.pict from the **Slideshow PICTs** folder and link it to the scene*

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



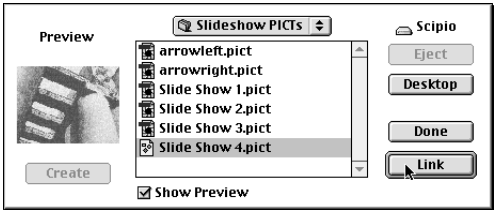
Now we're ready to create the fourth and last scene in our project.

1 Choose **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.



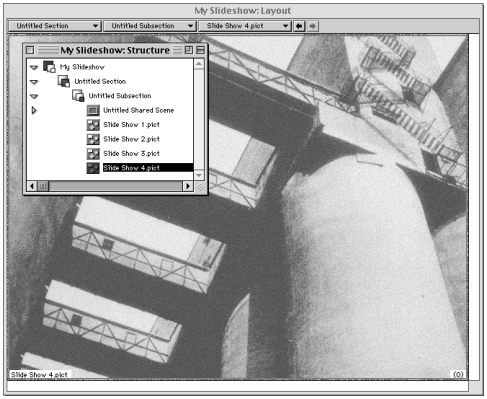
2 Choose **Link Media ➛ File** from the **File** menu. A file selection dialog box appears.

3 In the file dialog box, choose 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.



*Choose Slide Show 4.pict from the **Slideshow PICTs** folder and link it to the scene*

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



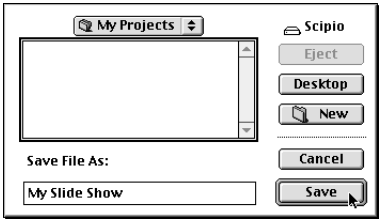
The Structure and Layout windows update to reflect the new scene

Save Your Project

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

1 Select **Save** from the **File** menu.

2 A file selection dialog box appears. Choose a location and name for your project, then click the **Save** button.



*Using the file selection dialog box, choose a location for your project, name it, and click **Save***

Run Your Project

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

1 Press ⌘-T to switch from edit mode to run-time mode.

2 The mTropolis interface disappears and the first scene of the project is displayed.

3 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 end user has no way to access them. We need to add some modifiers to our project to create interactivity.

4 Press ⌘-T again to return to edit mode. The mTropolis editing environment reappears.

Add an Element to the Shared Scene

Now we'll create controls that can be used to navigate through the scenes of our project. Eventually, we'll have a forward and a backward button that end users can click to change the picture 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?

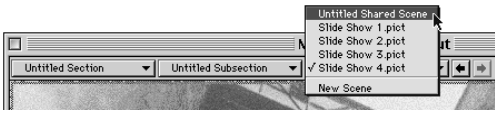
When looking at the Structure window, you may have noticed that our subsection contains a component named Untitled Shared Scene in addition to the four scenes linked to

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 menu to navigate to the shared scene.

1 Choose **Untitled Shared Scene** from the scene pop-up menu.



Choose Untitled Shared Scene in the Layout window's scene pop-up menu

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

The Layout window updates to show the shared scene, which is a black background

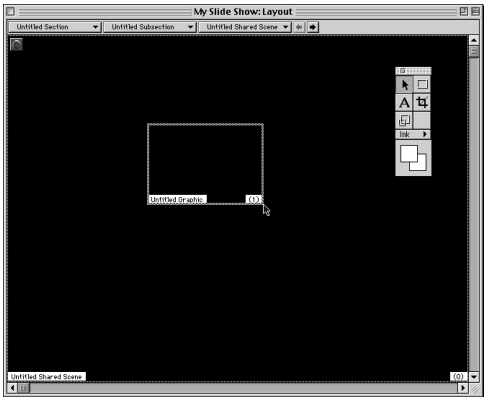
Add a new graphic element

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.

1 If the **Tool** palette is not already visible, select **Tool Palette** from the **View** menu. The **Tool** palette appears.

2 Click the Graphic tool in the **Tool** palette as shown below.

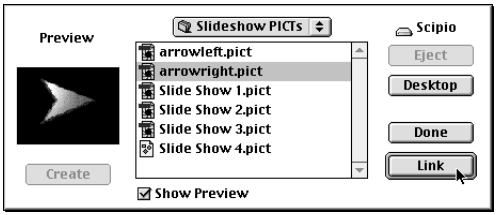
3 Drag the cursor somewhere inside the Layout window and click and drag to create a new graphic element on the shared scene. The size and position of the element are not important.



Create a new graphic element on the shared scene by dragging the cursor anywhere in the Layout window with the Graphic tool selected

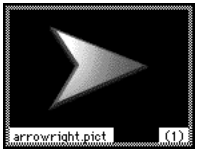
4 Choose **Link Media ➝ File** from the **File** menu. A file selection dialog box appears.

5 In the file dialog box, choose the file **arrowright.pict** from the **Slideshow PICTs** folder, found inside the **QuickStart** folder. Click **Link** to link the picture to the graphic element.



*Choose arrowright.pict from the **Slideshow PICTs** folder and link it to the graphic element*

6 The element updates and resizes to show its new contents. The element's name changes to arrowright.pict as shown below.



!!! 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.

Precise placement of elements

Now we'll use the **Object Info** palette to precisely position the arrow.

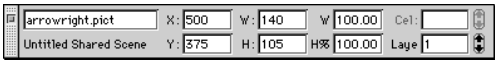
1 If the **Object Info** palette (shown below) is not already visible, choose **Object Info Palette** from the **View** menu. The **Object Info** palette appears. This palette shows sizing and position information for the currently selected object.

2 Click the **arrowright.pict** element to select it.

3 The element's name, position, and size information are displayed in the **Object Info** palette.

4 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.

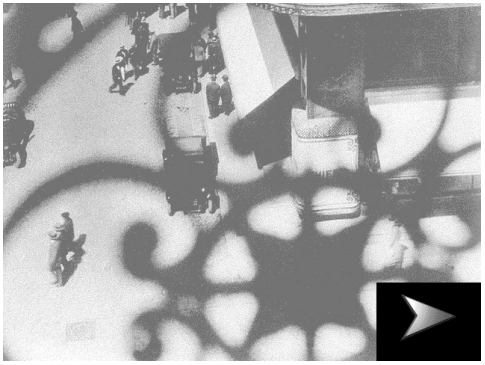
5 In the **Object Info** palette's **Y** field, enter **375.** Press the Return key to confirm the change. The arrowright.pict element moves to its new location. It should now be in the bottom right corner of the scene. The **Object Info** palette should look like the one shown below.



*The **Object Info** palette updated with the new position information for arrowright.pict*

Run the project
Switch to run-time mode to see the effects of our latest addition.

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



The arrow picture is superimposed on the first scene

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

There is no interactivity in our project yet, and our arrow button could probably look better. We'll address these problems in the next few steps.

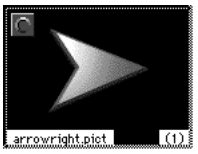
Modify the Appearance of the Arrow Element

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.

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

2 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.

The Graphic modifier icon



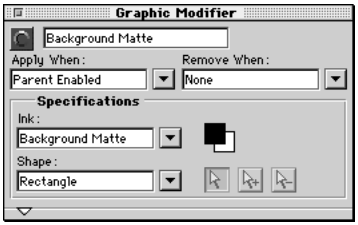
Graphic modifier attached to the arrowright.pict element

3 Double-click the Graphic modifier icon on the element. The **Graphic Modifier's** configuration dialog box appears. This dialog box can be used to customize the Graphic modifier.

4 Choose the name of the modifier (the topmost text field in the modifier which reads "Graphic Modifier") and change it to **Background Matte.**

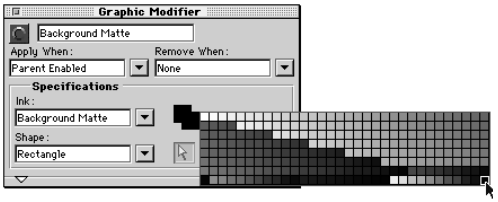
5 In the Specifications section of the dialog box, use the **Ink** pop-up menu to select the **Background Matte** effect. This effect will make the element's background invisible and insensitive to end user mouse clicks.

6 The **Graphic Modifier** dialog box should now look like the one shown below.



7 There is one final change to make. We need to specify which background color is transparent.

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



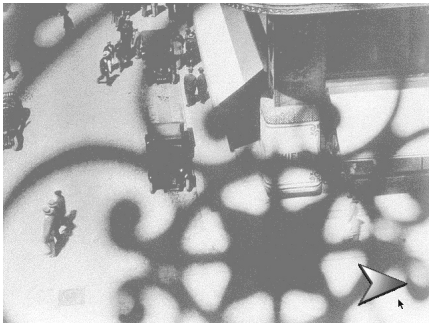
*Choose the black color swatch from the palette in the **Ink** pop-up menu of the **Graphic Modifier** dialog box*

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 black background color will be picked up by the eyedropper.

8 Close the **Graphic Modifier** dialog box by clicking its close box.

Observe the Effect

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



Switch to run-time mode (⌘-T) to observe the effect of the modifier

2 Press ⌘-T to return to edit mode.

Program the Arrow to Trigger a Scene Change

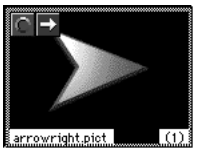
Now that our arrow button looks good, it's time to program some interactivity. We want the arrow to cause a scene change when it is clicked by the end user. mTropolis provides a modifier just for this purpose — the Change Scene modifier.

1 If the **Logic** modifier palette is not already visible, choose **Modifier Palettes ➝ Logic** from the **View** menu. The **Logic** modifier palette appears.

*The **Logic** modifier palette*

2 Drag a Change Scene modifier from the **Logic** modifier palette and drop it on the arrowright.pict element.

The Change Scene modifier icon



The Change Scene modifier icon and the Change Scene modifier applied to the arrowright.pict element

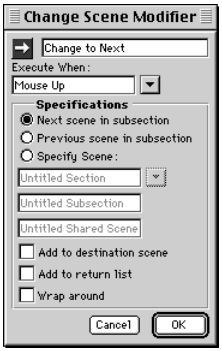
3 Double-click the Change Scene modifier icon on the arrowright.pict element to display its configuration dialog box.

4 Change the modifier's name from Change Scene modifier to `Change to Next`.

5 The default for the **Execute When** pop-up menu is Mouse Up. That is, this modifier will execute when an end user clicks the element. We don't need to change this value.

6 We don't need to make any changes to the Specifications section of the dialog box, either. By default, the modifier is configured to change to the next scene in the subsection.

7 The dialog box should now look like the one shown below.



*The **Change Scene Modifier** dialog box, properly configured for the arrowright.pict element*

8 Click **OK** to dismiss the dialog box.

Run the project

With the addition of the Scene Change modifier, we've enabled some simple interactivity.

1 Press ⌘-T to switch to run-time mode.

2 Notice that the cursor changes when it is over the arrow button, indicating that it can be clicked.

3 Click the arrow button to change to the next scene. Repeated clicks cause the scene to change until the last scene is displayed.

4 Press ⌘-T to return to edit mode.

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

Add a Back Button

Our project lets end 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.

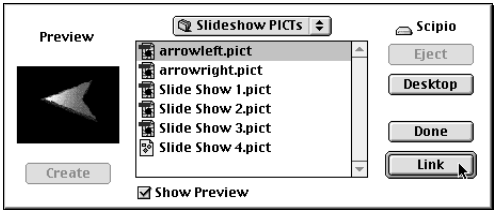
1 Click the **arrowright.pict** element to select it.

2 Choose **Duplicate** from the **Edit** menu. A copy of the element appears.

3 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.

4 Choose **Link Media ➛ File** from the **File** menu. A file selection dialog box appears.

5 In the file dialog box, select the file **arrowleft.pict** from the **Slideshow PICTs** folder, found inside the **QuickStart** folder. Click **Link** to link the picture to the element.



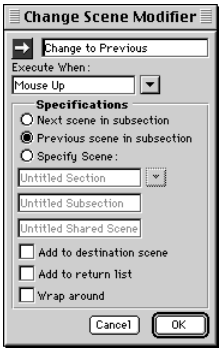
*Choose the arrowleft.pict from the **Slideshow PICTs** folder. Click **Link** to link the picture to the element*

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.

1 Double-click the Change Scene modifier icon on the arrowleft.pict element. The **Change Scene Modifier** dialog box appears.

2 Change the modifier's name from Change to Next to `Change to Previous`.

3 In the **Specifications** section, click the **Previous scene in subsection** button. The dialog box should look like the one shown below.



*The **Change Scene Modifier** dialog box, properly configured for the arrowleft.pict element*

4 Click **OK** to close the dialog box.

Run the project

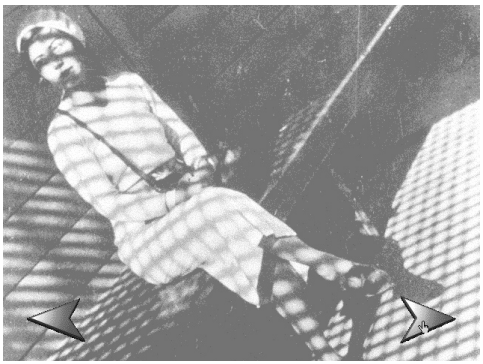
Now end users can navigate both forward and backward through the slideshow. Let's try it out!

1 Press ⌘-T to switch to run-time mode.

2 Click the right arrow button to move forward through the scenes.

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

4 Press ⌘-T to return to edit mode.



Click the arrow buttons to navigate both forward and backward through the slideshow

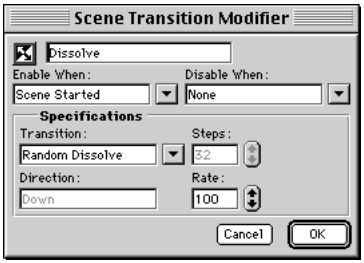
Add Scene Transition Effects

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

1. Use the **scene pop-up** menu or **right scene navigation arrow** found at the top of the Layout window to display the **Slide Show 1.pict** scene.
2. 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.

The Scene Transition modifier icon

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



*The **Scene Transition Modifier** dialog box, properly configured for a dissolve effect*

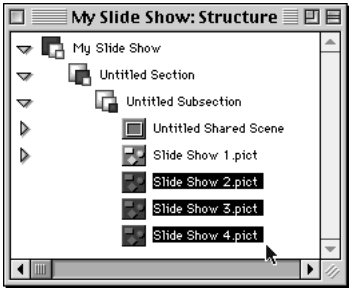
6. Click **OK** to dismiss the dialog box.

Copy the Scene Transition modifier to each scene

Now let's use the structure view to copy the Scene Transition modifier to the other three scenes.

1. While the Scene Transition modifier is still selected, choose **Copy** from the **Edit** menu (⌘-2).
2. Choose **Structure Window** from the **View** menu to display the structure view of our project.

3 In the Structure window, select the three scenes to which we want to copy the Scene Transition modifier. Hold the **Shift** key while clicking the icons for Slide Show 2.pict, Slide Show 3.pict, and Slide Show 4.pict. All three scenes should be highlighted as shown below.



Hold Shift while clicking the second, third, and fourth scenes to select all three

4 Choose **Paste** from the **Edit** menu. A new copy of the Scene Transition modifier will be pasted on each selected scene.

Run the project

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

1 Press ⌘-T to switch to run-time mode.

2 Click the buttons to move through the images in the slideshow. Each scene dissolves into the next.

3 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 difficulty completing this tutorial, you might want to examine a finished version. Choose **Open** from the **File** menu and select the project file **Completed Slideshow** found in the **QuickStart** folder. The 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.

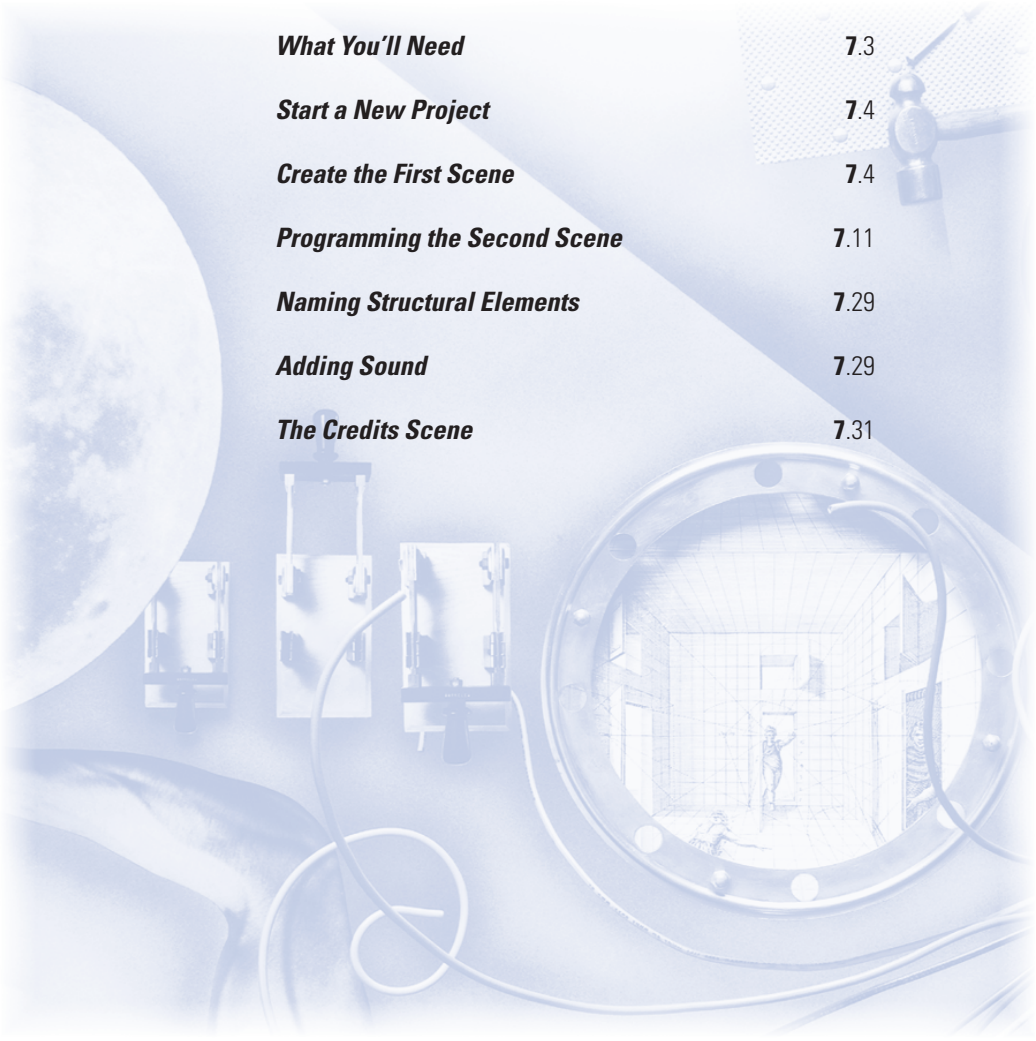
Examining this project may give you ideas about enhancements you can make to your own slideshow.

More Tutorials

The next chapter, "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.

Chapter 8, "Network Tutorial — Avatar Chat," is another challenging tutorial that shows how to use mTropolis to create an application that allows end users to send chat messages to each other over a network.

7 In-Depth Tutorial — mPuzzle



7 In-Depth Tutorial — mPuzzle

This tutorial provides a more detailed introduction to mTropolis. 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. This tutorial shows how to integrate QuickTime movies, animations, and sound in a multiple scene project.

What You'll Need

You'll need the following things before starting this tutorial.

- mTropolis must be installed on your machine. Installation instructions are in the "Read Me First!" file on the mTropolis CD–ROM.
- The tutorial files are installed by default when mTropolis is installed. Tutorial files are in the Tutorials folder of the mTropolis installation. If the tutorial files have not been installed, install by running the installer from the mTropolis CD–ROM, or drag the Tutorials folder from the CD to your hard disk.
- This project uses 8-bit media. For best performance, make sure your monitor is set to display 256 colors.

Tutorial project description

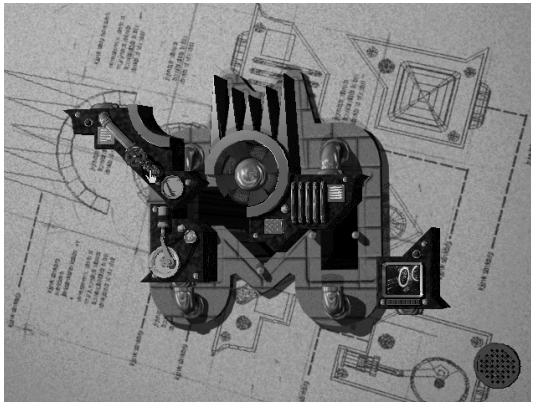
Let's begin by looking at the completed puzzle project.

1. Open the completed tutorial project in mTropolis by dragging the **Completed Tutorial** icon, found in the **In-Depth** folder (contained in the **Tutorials** folder), over the mTropolis icon. If you have multiple versions of mTropolis installed (for example, both the 68K and PPC versions), be sure to choose the correct one for your machine.
2. The project is shown in *edit mode*, where changes can be made to the project. To view the project as an end user would see it, press ⌘-T to switch to *run-time mode*. The project will run from its first scene.

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

The second scene 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" machine springs to life as a series of animations on different parts of the "M" begin to play.

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



Solving the mPuzzle tutorial

Start a New Project

When you are finished exploring the completed puzzle, close the finished tutorial project by choosing **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 choosing **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.

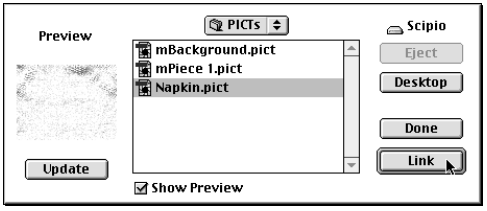
Create the First Scene

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.

1. Click the Untitled Scene (inside the large white region in the Layout window) to select it.
2. Choose **Link Media ➛ File** from the **File** menu. A standard dialog box appears as shown below.



*Choose the Napkin.pict image and click the **Link** button*

3. 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) and click the **Link** button.
4. 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 box. In the next step, we'll link the image's color table to the project so our images will display properly.
5. 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.

1. Ensure that the **Effects Modifier** palette is visible. To do this, choose the **View** menu and look at the **Modifier** palette's cascading menu item. If there is a check mark next to **Effects**, that palette is already visible. If there is not a check mark, choose the **Effects** menu item to display the palette.
2. Drag a Color Table modifier from this palette and drop it on the scene (the background Napkin.pict element). The Color Table modifier attaches itself to the upper left corner of the scene.

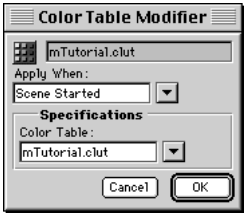
The Color Table modifier icon

3 Double-click the Color Table modifier on the scene to open its dialog box.

4 The highlighted text at the top of the **Modifier** dialog box 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.

5 From the **Color Table** pop-up menu, choose the **Link file** option. A standard dialog box appears. Choose the `mTutorial.clut` file (**Media** folder ➛ **CLUTs** folder).

6 Your **Color Table Modifier** dialog box should now look like the one shown below. Click the **OK** button to dismiss the **Color Table Modifier** dialog box.



Color Table Modifier *dialog box*

7 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.

1 Select the Graphic Element tool (the box-shaped tool) in the Tool palette.

The Graphic Element tool

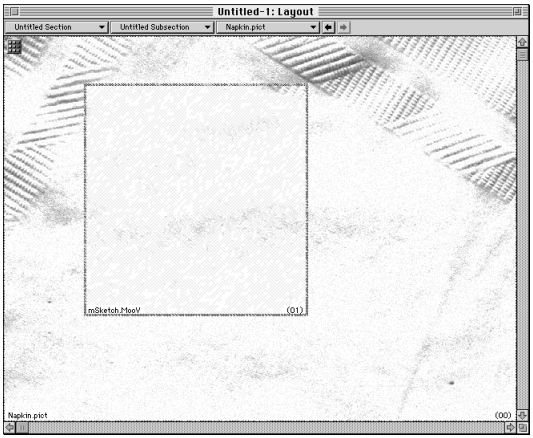
2 Drag anywhere on the scene to create an empty graphic element of any size.

3 Select the new element and choose **Link Media** ➛ **File** from the **File** menu. A standard dialog box appears.

4 Choose the file named mSketch.MooV in the **MOOVs** folder, found inside the **Media** folder.

If the Application preferences (**Edit** ➛ **Preferences** ➛ **mTropolis**) are still set to their defaults, the element will resize automatically to the size of the QuickTime movie. However, if it doesn't resize automatically, choose the element and then **Revert Size** from the **Object** menu.

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

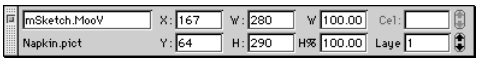


Layout window

Position the movie

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

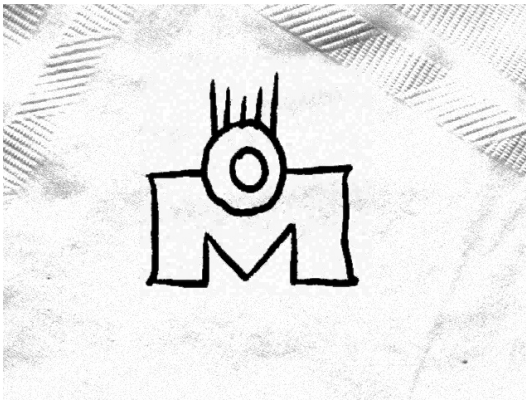
1. If the **Object Info** palette (shown below) is not already visible, choose **Object Info Palette** from the **View** menu. The **Object Info Palette** appears. This palette shows sizing and position information for the currently selected object.
2. Select the mSketch.MooV element, and enter `167` 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.



***Object Info** palette used to position mSketch.Moov*

Test the project

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



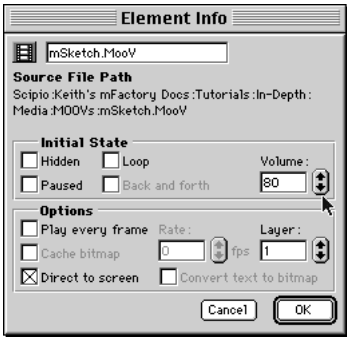
Switch to run-time mode to play the QuickTime movie

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 a number of ways through its **Element Info** dialog box. Next we'll use the movie's **Element Info** dialog box to adjust the movie's volume.

1. Double-click the mSketch.MooV element to open its **Element Info** dialog box, or display the dialog box by selecting the mSketch.MooV element and then choosing **Element Info** from the **Object** menu.

2 In the **Element Info** dialog box, 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 box should look similar to the one shown below.



*Configuring the movie's **Element Info** dialog box*

3 Click **OK** to accept this change and dismiss the **Element Info** dialog box.

4 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.

1 Choose **Save** from the **File** menu (⌘-S).

2 Enter `in-depth project` in the **Save File As** field and store the project as you would any other file.

3 If at any point you want to restore the project to a previously-saved version, choose **Open** from the **File** menu to load the saved file.

4 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 end user clicks before the movie is done, the scene will change.

1 Ensure that the **Logic Modifier** palette is visible. To do this, choose **View ➛ Modifier Palettes.** If there is a check mark next to **Logic**, that palette is already visible. If there is not a check mark, choose the **Logic** menu item to display the palette.

*The **Logic Modifier** palette*

2 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.

The Change Scene modifier icon

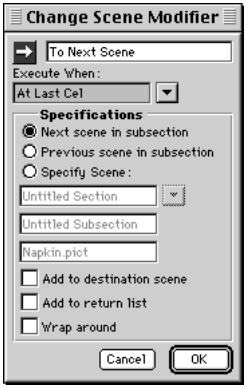
3 Double-click the modifier icon to display its configuration dialog box.

4 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.`

!!! Note: 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.

5 Open the **Execute When** pop-up menu and choose **Play Control ➛ At Last Cel.** Now when the movie ends, a scene change will occur. The **Change Scene Modifier** dialog box should look like the one shown below.



*A **Change Scene Modifier** dialog box configured to execute on the **At Last Cel** message*

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

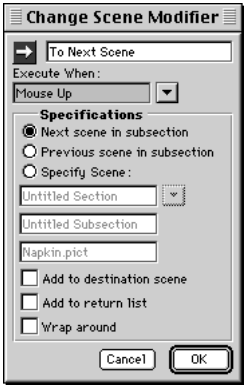
6 Accept the changes to the modifier by clicking the **OK** button. The **Change Scene Modifier** dialog box disappears.

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

1 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 (that is, drop the modifier outside the bounds of the mSketch.Moov element so that it attaches to the Napkin.pict element).

2 Double-click the new Change Scene icon to display its configuration dialog box.

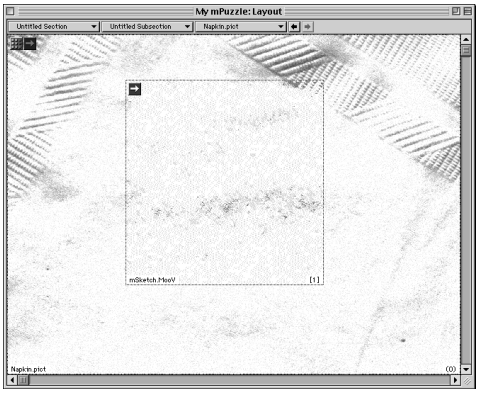
3 From the **Execute When** pop-up menu, choose **Mouse ➛ Mouse Up.** Now this modifier will activate when the end user clicks the scene. Your new **Change Scene** dialog box should look like the one shown below.



*A **Change Scene Modifier** dialog box configured to execute on **Mouse Up***

4 Click **OK** to accept the change. The dialog box 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.



The Layout window showing our first scene

Create a new scene

To create a new scene in the layout view, use the third pop-up menu on the right at the top of the window (where it now reads Napkin.pict). The pop-up menu 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 menu.

1 Choose **New Scene** from the **Scene** pop-up menu as shown in the image below. A new, empty scene (named Untitled Scene) is created. The Layout window changes to display this new scene.

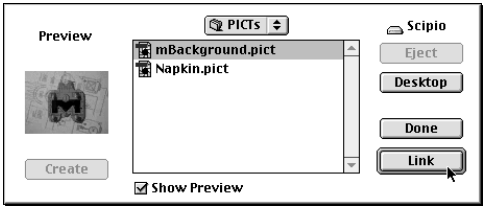
*Choose **New Scene** from the scene pop-up menu*

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

1 Click the Untitled Scene to choose it.

2 Choose the **Link Media → File** option from the **File** menu. A standard dialog box appears.

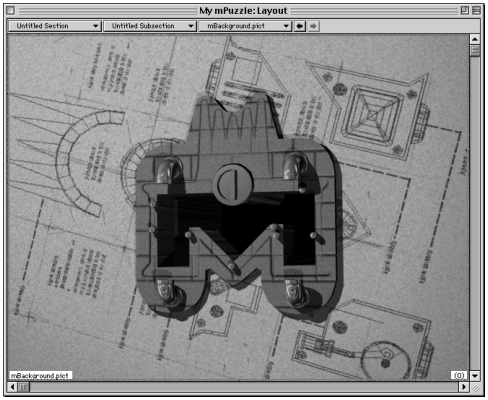
3 Choose the mBackground.pict file from the **PICTs** folder within the **Media** folder.



Link the mBackground.pict image to the new scene

4 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.

5 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.



The new scene linked to the mBackground.pict image

Let's see the effect of adding this scene.

1 Return to the previous scene by using the **Scene** pop-up menu to choose Napkin.pict, or click once on the left scene navigation arrow at the top of the Layout window.

2 Now run the project (press ⌘-T) and view the changes. Notice that when the movie ends, or when you click the first scene, the scene changes to the next scene. Press ⌘-T again to return to edit mode.

3 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.

1 Drag a Scene Transition modifier from the Effects modifier palette and drop it on the Napkin.pict scene.

The Scene Transition modifier

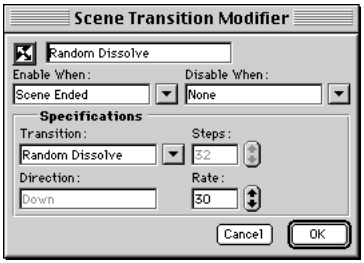
2 Double-click the modifier to display its configuration dialog box.

3 Change its name from Scene Transition modifier to `Random Dissolve`.

4 Choose **Scene** ➛ **Scene Ended** from the **Enable When** pop-up menu.

5 Choose **Random Dissolve** from the **Transition** pop-up menu.

6 Set the value of the Rate option to `30`. The dialog box should look like the one shown below.



Scene Transition Modifier *dialog box*

7 Click the **OK** button to close the **Scene Transition Modifier** dialog box. The dialog box 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.

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 end 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. When all of the puzzle pieces are in place, they become animated.

Adding a puzzle piece to the scene

Navigate to the second scene by choosing mBackground.pict from the **Scene** pop-up menu (the third pop-up menu 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 section, we'll use the **Asset Palette** to manipulate media that has been linked to the project.

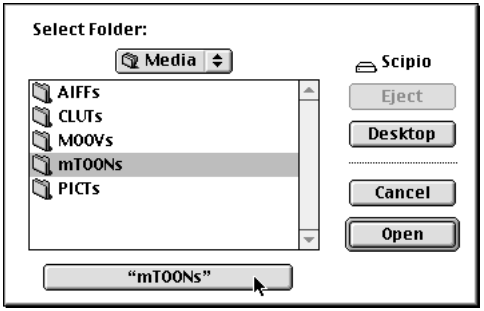
1 Choose **Asset Palette** from the View menu. The **Asset** palette appears as shown below. This palette shows thumbnail images of all of the media assets currently linked to the project.

*The **Asset Palette***

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 — they will only be added to the **Asset** palette.

Link all the media files contained in a directory to the project.

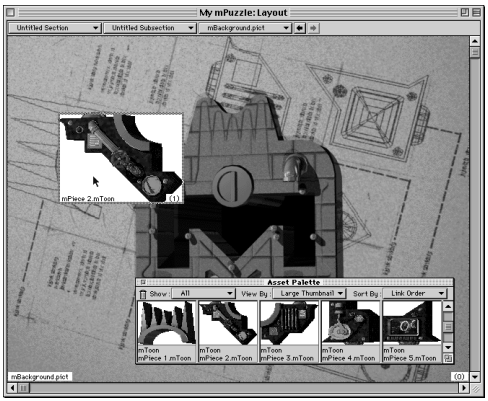
2 Choose **Link Media** ➛ **Folder** from the **File** menu. A standard dialog box appears. Choose the **mTOONs** folder found within the **Media** folder. Click the button at the bottom of the folder selection dialog box when the name **mTOONs** appears in that button as shown below.



Linking an entire folder of media to the project

The **Asset** palette now contains thumbnails for seven new assets. The media assets on the **Asset** palette can be dragged from the palette and dropped on our project. Let's add one of the puzzle pieces to the Layout window.

3 Drag the mToon element named mPiece 2.mToon from the **Asset** palette and drop it on the mBackground.pict scene. The Layout window should look similar to the one shown below.



*Drag the mPiece2.toon from the **Asset** palette and drop it on the scene*

Programming the first puzzle piece

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

Aliasing requires additional explanation. To program the puzzle, we are going to create a single behavior that can be used on *all* of the puzzle pieces. We will alias this behavior so that updates made to one copy are made simultaneously to all of the copies on the other puzzle pieces.

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.

1 Drag a Behavior modifier from the **Logic** modifier palette and drop it onto the puzzle piece mPiece 2.mToon in the Layout window. The Behavior modifier is a special modifier that can contain other modifiers.

The Behavior modifier icon

2 Double-click the Behavior modifier to display its configuration dialog box.

3 Change the name of the Behavior from Behavior to `Puzzle Piece`. Do not close the **Behavior** dialog box.

Making the puzzle piece transparent

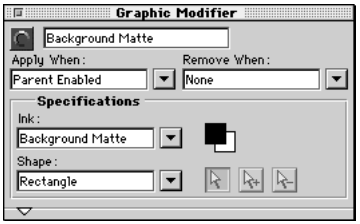
Now let's add the programming that will make the piece transparent to the background.

1 Drag a Graphic modifier from the **Effects Modifier** palette and drop it in the open **Behavior** window.

2 Double-click the Graphic modifier to open its configuration dialog box.

3 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.

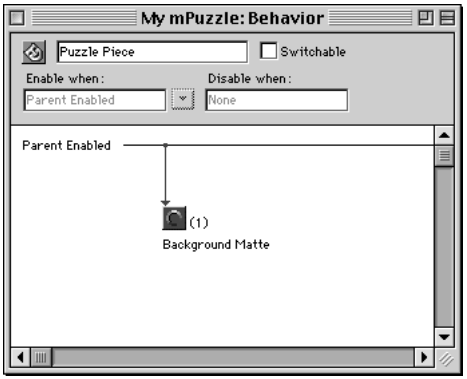
4 Use the **Ink** pop-up menu to choose **Background Matte.** The dialog box should look like the one shown below.



*The **Graphic Modifier** configuration dialog box, configured to apply the Background Matte ink*

5 Close the **Graphic Modifier** dialog box (by clicking its **Close** box) to accept the change.

The **Behavior** window should now look like the one shown below.



*The **Puzzle Piece Behavior** configuration dialog box containing the Background Matte Graphic modifier*

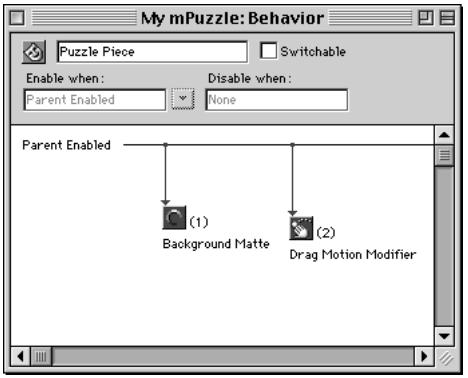
Making the puzzle piece draggable

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

1 Drag a Drag Motion modifier from the **Effects Modifier** palette and drop it on the Puzzle Piece behavior window.

The Drag Motion modifier icon

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. Your Puzzle Piece Behavior window should look like the one shown below.



Puzzle Piece behavior window

2 Close the Puzzle Piece behavior window by clicking its **Close** box.

3 Use ⌘-T to switch to run-time 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 Media** command.

1. Re-open the Puzzle Piece behavior by double-clicking its icon.
2. Drag a Messenger modifier from the **Logic Modifier** palette and drop it in the Puzzle Piece behavior window.

The Messenger modifier icon

3. Double-click the Messenger modifier icon to display the configuration dialog box.
4. Change the name of the messenger from Messenger to `Preload`.
5. Use the messenger's **Execute When** pop-up menu to choose **Scene ➛ Scene Started.**
6. In the messenger's **Message Specifications** section, use the **Message/Command** pop-up menu to choose the **Element ➛ Preload Media** command.
7. No changes need to be made to the **With** and **Destination** menus. The dialog box should look like the one shown below.

Configuring a Messenger modifier to preload the element's media

8. Click **OK** to accept the changes. The **Messenger** dialog box disappears.

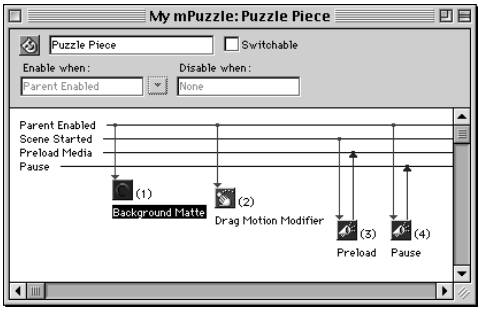
By default, the puzzle piece animations play when the scene starts. However, we want the animations to pause until the puzzle is complete. This can be done by sending a **Pause** command to the puzzle piece.

1. Drag another Messenger modifier from the **Logic Modifier** palette and drop it in the Puzzle Piece behavior window.
2. Double-click the Messenger modifier icon to display its configuration dialog box.
3. Change its name to `Pause`.
4. Use the messenger's **Execute When** pop-up menu to select the **Parent ➛ Parent Enabled** option.
5. Use the messenger's **Message/Command** pop-up menu to select the **Play Control ➛ Pause** option.
6. Leave the **Destination** of the message as **Element** (this is the default destination). The **With** pop-up menu should also remain at the default. The dialog box should look like the one shown below.

Configuring a Messenger modifier to pause the element's media

7 Click **OK** to accept the changes and dismiss the **Messenger** dialog box.

8 Your Puzzle Piece behavior window should look like the one shown below.



Puzzle Piece behavior window

9 Click the behavior's **Close** box to dismiss the Puzzle Piece behavior window.

Note: The animation could also have been paused by setting the element's paused attribute via the **Element Info** dialog box. 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 manually set 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 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 each 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.mToon puzzle piece.

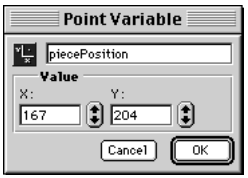
1 Drag a Point Variable modifier from the **Logic Modifier** palette and drop it on the mPiece 2.toon puzzle element.

The Point Variable modifier icon

2 Double-click the Point Variable icon to display its configuration dialog box.

3 Change the variable's name from Point Variable to `piecePosition`. Note there are no spaces in that name!

4 Enter `167` into the modifier's **X** field and `204` into the **Y** field. The dialog box should look like the one shown below.



*Configuring the piecePosition **Point Variable** dialog box for the mPiece2.mToon element*

5 Click **OK** to accept the changes and dismiss the **Point Variable** dialog box.

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 create a simple script that sets the puzzle piece's position to a random position within the boundary of the screen.

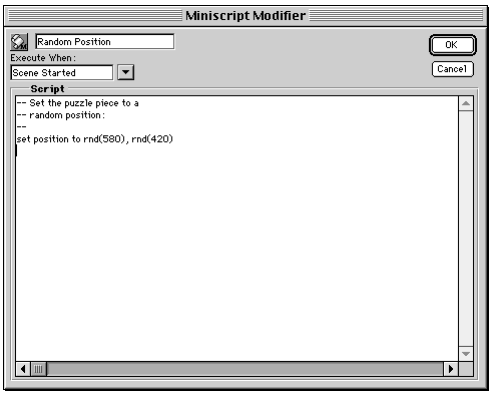
1. Double-click the Puzzle Piece behavior icon to open its window.
2. Drag a Miniscript modifier from the **Logic Modifier** palette and drop it in the Puzzle Piece Behavior window.

The Miniscript modifier icon

3. Double-click the Miniscript modifier icon to display its configuration dialog box.
4. Change the modifier's name from Miniscript Modifier to `Random Position`.
5. Use the **Execute When** pop-up menu to choose **Scene ➛ Scene Started**.
6. 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 **Miniscript Modifier** dialog box should now look like the one shown below.



Writing the Random Position Miniscript modifier

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 make your scripts easier to read and debug.

The last line is a Miniscript statement. It uses the Miniscript function "rnd" to generate a random number between 0 and the number in parentheses for the x and y coordinates of the element's position. When activated, this script will set the puzzle piece's position to the random values generated by the "rnd" function.

7. Click **OK** to accept these changes and dismiss the **Miniscript Modifier** dialog box.

Encapsulating the Modifiers into a Behavior

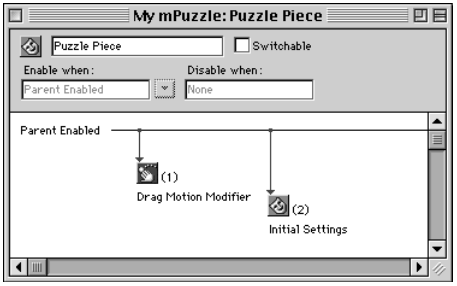
As good housekeeping, we'll encapsulate some of the modifiers we have created into a new behavior. Let's group the modifiers that set the initial characteristics of the puzzle piece together.

1. Drag a new Behavior from the **Logic Modifier** palette and drop it into the open Puzzle Piece Behavior window.

The Behavior modifier icon

2. This time, instead of opening the behavior to rename it, simply click the behavior's name shown below its icon, and enter the new name, `Initial Settings`. Click outside the name when you are through editing the name.

3. Instead of opening the new behavior's dialog box, modifiers can be added to the behavior simply by dragging and dropping their icons on the new behavior's icon. Drag the following modifiers on 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.



The Puzzle Piece Behavior window with Initial Settings behavior added

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 to the value stored in the piecePosition Point Variable whenever the end user drops the piece at a new location. (We attached the variable to the puzzle piece and assigned a value to the variable earlier in this chapter.)

1. Drag an If Messenger from the **Logic Modifier** palette and drop it in the Puzzle Piece Behavior window.

The If Messenger modifier icon

2. Double-click the messenger's icon to display its configuration dialog box.

3. Change the name of the messenger from If Messenger to `Dropped in Valid Position.`

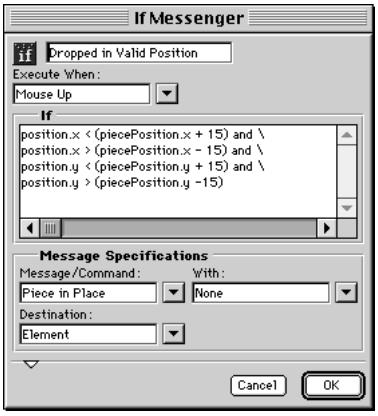
4. By default, this messenger is configured to act on the Mouse Up message. This is what we want, so we don't need to change the **Execute When** pop-up menu.

5. 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.

6 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 box should look like the one shown below.



Configuring an If Messenger to detect when the element is dropped in the correct location

7 Click **OK** to save your changes to this configuration dialog box. An alert appears, asking if you want to create a 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.

1 Drag a new Behavior modifier from the **Logic Modifier** palette and drop it in the Puzzle Piece Behavior window.

2 Click 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.

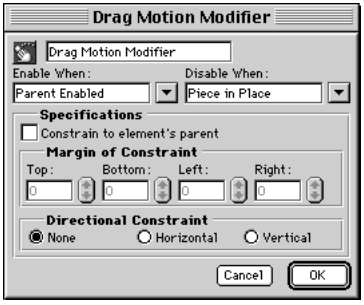
Next we'll configure the previously-created Drag Motion modifier so that the puzzle piece cannot be dragged once it is in place.

1 Drag the Drag Motion modifier icon from its current position in the Puzzle Piece window and drop it into the Piece in Place Behavior.

2 Double-click the Piece in Place Behavior icon to open its window.

3 In the Piece in Place window, double-click the Drag Motion icon to display its configuration dialog box.

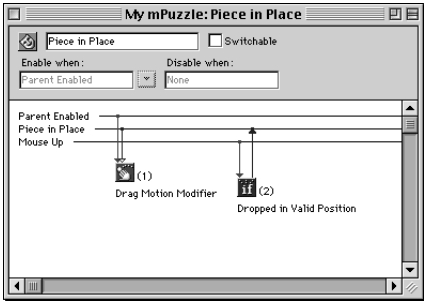
4 We want to disable this modifier when the piece is put in its correct place. Use the **Disable When** pop-up menu to choose **Author Messages ➛ Piece in Place.** The piece will no longer be draggable after this message is received. The dialog box should look like the one shown below.



*Configuring the **Drag Motion Modifier** dialog box to disable on the Piece in Place message*

5 Click **OK** to confirm your changes and dismiss the **Drag Motion Modifier** dialog box.

6 Move the Dropped in Valid Position If Messenger you created previously from the Puzzle Piece Behavior into the Piece in Place Behavior. Simply drag it from the Puzzle Piece Behavior window and drop it in the Piece in Place Behavior window. Your Piece in Place window should now look like the one shown below.



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.

1 Drag a new Miniscript modifier from the **Logic** palette and drop it into the Piece in Place behavior.

2 Double-click the modifier to display its configuration dialog box.

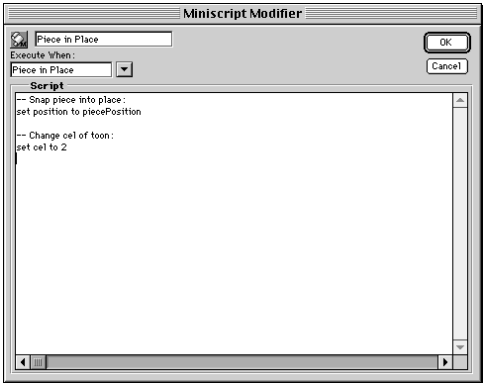
3 Name the new Miniscript modifier `Piece in Place.`

4 Use the **Execute When** pop-up menu to configure the modifier to activate when the Piece in Place author message is received. Choose the **Author Messages ➛ Piece in Place** option from the menu.

5 In the Script text field, enter the following script:

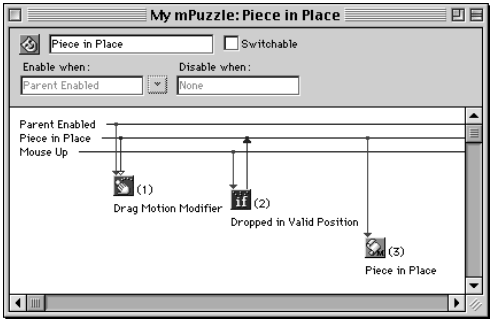
```
-- Snap piece into place:
set position to piecePosition

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

The dialog box should look like the one shown below.



Writing the Piece in Place Miniscript

6 Click **OK** to close the **Miniscript** dialog box. Your Piece in Place behavior should look similar to the one shown below.



Piece in Place behavior

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.

7 Click the close boxes of both the Piece in Place and Puzzle Piece behaviors to dismiss their windows.

You may want to test your programming up to this point. Press ⌘-T to switch to run-time 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. When 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 end user that the puzzle piece is in place.

1 Double-click the Puzzle Piece behavior icon on the mPiece 2.mToon element to open its window.

2 Double-click the Piece in Place behavior icon (in the Puzzle Piece window) to open its window. We'll be adding new modifiers to this behavior.

3 Drag a Sound Effect modifier from the **Effects Modifier** palette and drop it into the Piece In Place behavior window.

The Sound Effect modifier icon

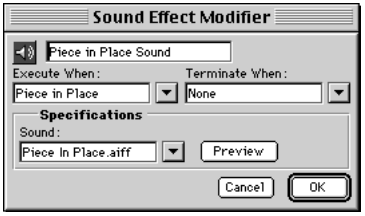
4 Double-click the Sound modifier to display its configuration dialog box.

5 Name the modifier `Piece in Place Sound`.

6 Use the **Execute When** pop-up menu to choose the message that triggers the sound. Choose **Author Messages ➛ Piece in Place.**

7 Use the dialog box's **Sound** pop-up menu to choose a sound to be played. Choose **Link File.** A standard dialog box appears. Choose the Sound file Piece in Place.aiff located in the **AIFFs** folder located in the **Media** folder.

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



*Configuring the Piece in Place sound **Sound Effect Modifier** dialog box*

9 Click **OK** to accept your changes and dismiss the **Sound Effect Modifier** dialog box.

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 ensure 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.

1 Drag a new Messenger modifier from the **Logic Modifier** palette and drop it into the Piece in Place behavior window.

2 Double-click the new messenger to display its configuration dialog box.

3 Rename the messenger `Disable Checks`.

4 Use the dialog box's **Execute When** pop-up menu to choose the **Author Messages ➛ Piece in Place** message.

5 Highlight the text in the **Message/Command** menu and type `Disable Checks`. The dialog box should look like the one shown below.

*Configuring the Disable Checks **Messenger** modifier dialog box*

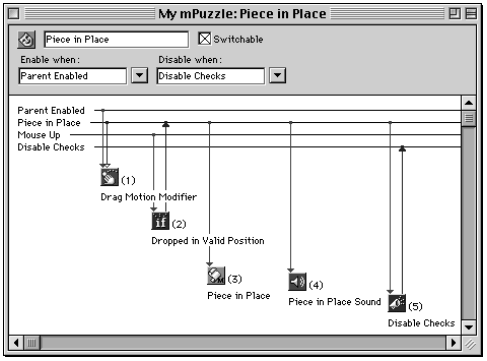
6 Click **OK.** A dialog box appears, asking if you want to create a new author message. Click **OK.**

7 In the Piece in Place behavior window, mark the **Switchable** check box found next to the behavior name.

8 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).

9 Verify that the behavior's **Enable When** message is Parent Enabled, then use the **Disable When** pop-up menu to choose **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 below. We are finished modifying the Piece in Place behavior, so close the window by clicking its **Close** box.



The completed Piece in Place behavior window

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.

1 Start by dragging a new Behavior modifier from the **Logic Modifier** palette and dropping it in the Puzzle Piece behavior window.

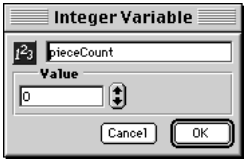
2 Double-click the new behavior's icon to display its configuration dialog box.

3 Change the name of the behavior to `Puzzle Status`.

4 Drag an Integer Variable from the **Logic Modifier** palette and drop it in the Puzzle Status behavior window.

The Integer Variable modifier icon

5 Double-click the Integer Variable's icon to display its configuration dialog box. 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. The dialog box should look like the one shown below.

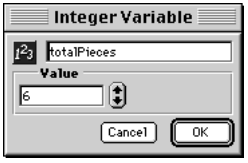


Integer Variable

6 Click **OK** to dismiss the Variable's dialog box.

7 Drag another Integer Variable into the Puzzle Status behavior and name it `totalPieces`.

8 Set the variable's value to `six`. The dialog box should look like the one shown below.



Variable Value

9 Click **OK** to dismiss the Variable's dialog box.

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.

1 Select both Variable icons in the Puzzle Status behavior window (using Shift-click or by simply dragging the pointer across the variables to marquee them).

2 Choose **Make Alias** from the **Object** menu. Aliasing these variables ensures their values will be the same wherever they occur in your project.

3 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.

1 Drag a Set modifier from the **Logic Modifier** palette and drop it in the Puzzle Status behavior window.

The Set modifier icon

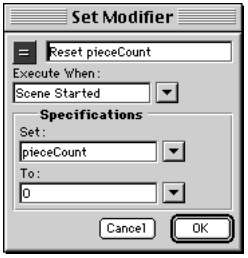
2 Double-click the Set modifier icon to display its configuration dialog box.

3 Change the modifier's name to `Reset pieceCount`.

4 Use the dialog box's **Execute When** pop-up menu to choose **Scene ➛ Scene Started.**

5 Use the modifier's **Set** pop-up menu to choose **Behavior ➛ pieceCount.**

6 Highlight the modifier's **To** field and enter `0` (zero). The dialog box should look like the one shown below.



Set Modifier *dialog box*

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

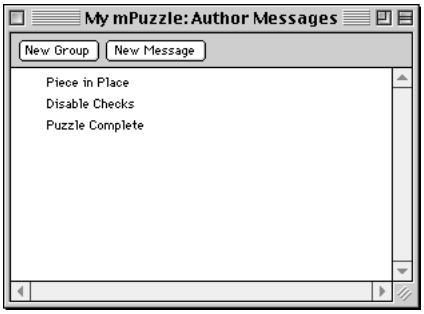
Notifying the environment that the puzzle is complete

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 end 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.

1. Choose **Author Messages Window** from the **View** menu. The Author Messages window appears.
2. Click the **New Message** button in the Author Messages window. An untitled message appears below the two previously-defined messages shown in the window.
3. Click the untitled author message and change its name to `Puzzle Complete, then press return.` The Author Messages window should look like the one shown below.



The Author Messages window

4. Close the close box window by clicking its **Close** button.

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

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

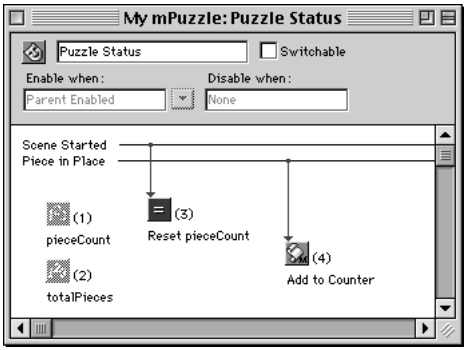
```
-- Increase the pieceCount:
set pieceCount to pieceCount + 1

-- 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 box should look like the one shown below.

Writing the Add to Counter miniscript

8 Click **OK** to dismiss the **Miniscript Modifier** dialog box.

9 Your Puzzle Status behavior window should look similar to the one shown below. Close the Puzzle Status behavior window by clicking its **Close** box.



The completed Puzzle Status behavior window

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.

1 Add a new Behavior modifier to the Puzzle Piece behavior.

The Behavior modifier icon

2 Double-click the Behavior icon to display its configuration dialog box.

3 Change the name of the behavior to `Puzzle Complete.`

4 Check the **Switchable** check box at the top of the Behavior window. The **Enable When** and **Disable When** pop-up menus become available.

5 Use the **Enable When** pop-up menu to choose **Author Messages ➛ Puzzle Complete.** Use the **Disable When** pop-up menu to choose **Parent ➛ Parent Enabled.**

6 Add a Miniscript modifier to the Puzzle Complete behavior window.

The Miniscript modifier icon

7 Double-click the Miniscript modifier to display its configuration dialog box.

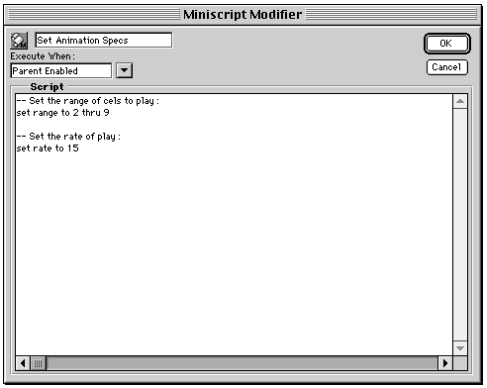
8 Change the Miniscript modifier's name to `Set Animation Specs.`

9 Use the **Execute When** pop-up menu to choose the **Parent ➛ Parent Enabled** message.

10 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 box should look like the one shown below.



Writing the Set Animation Specs miniscript

11 Click the **OK** button to dismiss the **Miniscript Modifier** dialog box.

12 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.

The Messenger modifier icon

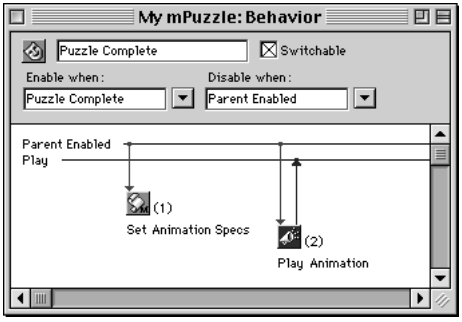
13 Double-click the Messenger icon to display its configuration dialog box. Change the Messenger's name to `Play Animation` and configure it to execute on the **Parent ➛ Parent Enabled** message using the **Execute When** pop-up menu.

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

Configuring the Play Animation Messenger modifier dialog box

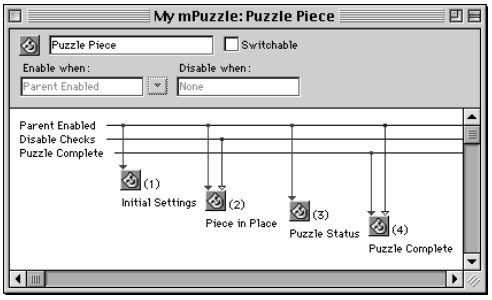
15 Click **OK** to close the **Messenger** dialog box and confirm the changes.

16 Your Puzzle Complete behavior should now look like the one shown below. Close the Puzzle Complete behavior window by clicking its Close box.



The completed Puzzle Complete behavior window

17 Your Puzzle Piece behavior should look like the one shown below. Close the Puzzle Piece behavior window by clicking its Close box.



The completed Puzzle Piece Behavior

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

Adding and programming the other puzzle pieces

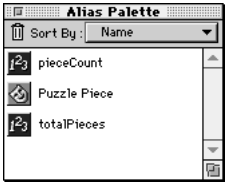
We have just created a reusable software component that we are going to apply to all of the puzzle pieces. Let's add them now.

1 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 appear on the **Alias** palette.

2 Drag the rest of the puzzle pieces (mPiece 1.mToon, mPiece 3.mToon, mPiece 4.mToon, mPiece 5.mToon, and mPiece 6.mToon) from the **Asset** palette into the Layout window.

3 Select the Puzzle Piece behavior icon found on the mPiece 2.mToon element.

4 Choose **Make Alias** from the **Object** menu. Notice that the Puzzle Piece behavior is added to the **Alias** palette as shown below.

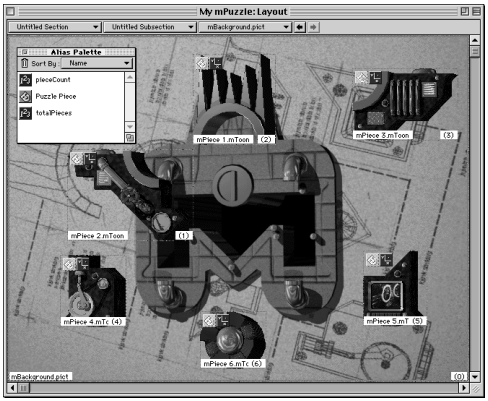


*The Puzzle Piece alias is added to the **Alias** palette*

5 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 on 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.

6 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. Hold down the Option key and drag a copy of piecePosition from mPiece 2.mToon to each piece. Using Option-drag makes a copy of the variable instead of moving the original.

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



Layout window with all puzzle pieces. Each piece contains the Puzzle Piece aliased behavior and a copy of the piecePosition variable

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

1. For each new puzzle piece, double-click the piecePosition variable to display its configuration dialog box. Change the X and Y values to the appropriate value shown in the following table. The value of mPiece 2.mToon has already been set, but is shown below for completeness.

Element Name	X	Y
mPiece 1.mToon	244	85
mPiece 2.mToon	167	204
mPiece 3.mToon	294	199
mPiece 4.mToon	167	241
mPiece 5.mToon	356	237
mPiece 6.mToon	257	166

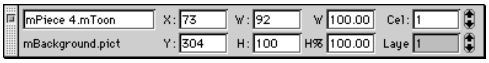
Changing the layer order of pieces

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 with higher layer order numbers will draw on top of elements with lower layer order numbers.

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

1. If it is not already displayed, open the **Object Info** palette by selecting **Object Info Palette** from the **View** menu.
2. Select each puzzle piece, and enter its correct layer order number in the Layer field of the **Object Info** palette as shown below. The table below shows the correct layer order number. When the pieces have been assigned the layer orders shown in the following table, the completed "M" machine will look its best.

Assett Name	Layer
mPiece 4.mToon	1
mPiece 1.mToon	2
mPiece 2.mToon	3
mPiece 6.mToon	4
mPiece 3.mToon	5
mPiece 5.mToon	6



*Setting the layer order number of mPiece 4.mToon using the **Object Info** palette*

3. 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 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.

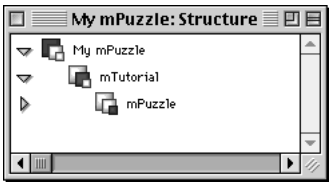
Renaming Sections and Subsections

We're going to use the mTropolis structure view to rename the Section and Subsection used in this project.

1. Select **Structure Window** from the **View** menu. The Structure window appears.
2. In the Structure window, click the text label that reads Untitled Section. When the field becomes editable, change the name to `mTutorial`.
3. 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 more 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.
4. Click the text label that reads Untitled Subsection. When the field becomes editable, change the name to `mPuzzle`.

!!! Note: 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.



The Structure window with renamed section and subsection components

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.

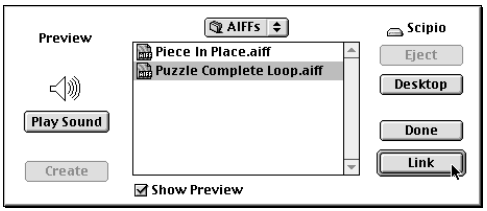
Adding a sound element to the puzzle

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.

1. In the Structure window, click the open/close triangle next to the mPuzzle subsection to reveal our project's scenes.
2. Now click 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.

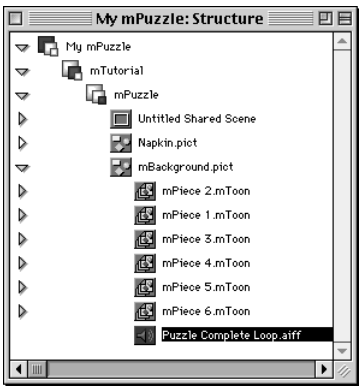
3 To add a new sound object, choose the mBackground.pict element and choose **New ➛ Sound** from the **Object** menu. A sound element icon appears below the mToon icons.

4 Make sure that the Sound icon is selected and choose **File ➛ Link Media ➛ File.** A standard dialog box appears. Choose the file Puzzle Complete Loop.aiff found in the **AIFFs** folder within the **Media** folder and click the **Link** button as shown below.



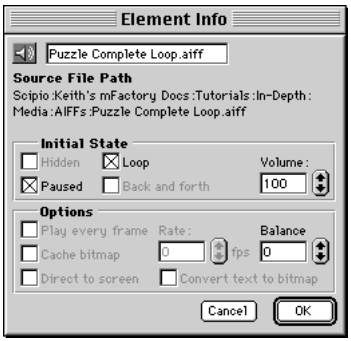
Linking the Puzzle Complete Loop.aiff sound to the new sound element

5 The name of the sound element icon changes to reflect the name of the media. Your Structure window should now look like the one shown below.



The Structure window, showing the new sound element linked to the Puzzle Complete Loop.aiff sound

6 Double-click the sound icon to display its **Element Info** dialog box. In the dialog box's **Initial State** section, select the Paused and Loop options. The **Element Info** dialog box should look like the one shown below.



Configuring the sound element to be paused and looped

7 Click **OK** to confirm the changes and close the dialog box.

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

1 Drag a Messenger modifier on the Puzzle Complete Loop.aiff sound icon. The icon will seem to disappear in 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.

The Messenger modifier icon

2 Double-click the Messenger icon to display its configuration dialog box.

3 Change the Messenger's name to `Play when Puzzle Complete.`

4 Use the **Execute When** pop-up menu to choose **Author Messages ➛ Puzzle Complete.**

5 Use the **Message/Command** pop-up menu to choose the command **Play Control ➛ Play.** Now when activated, this modifier will cause the sound element to begin playing. The dialog box should look like the one shown below.

Configuring the Play when Puzzle Complete ***Messenger*** *modifier dialog box*

6 Click **OK** to confirm the changes and dismiss the **Messenger** dialog box.

7 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

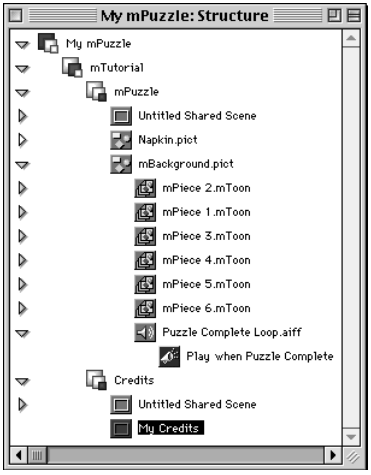
After all this work, it's time for some hard-earned recognition. Making your own credits screen is a good start. We are going to create a simple credit roll in a new subsection of the project.

1 Create a new subsection by choosing the **New Subsection** option from the **Subsection** pop-up menu 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.

2 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.`

3 Click the subsection's open/close triangle to reveal its scenes.

4 Highlight the name of the Untitled Scene and change it to `My Credits.` Your structure view should look similar to the one shown below.



The Structure window, showing the new Credits subsection and My Credits scene

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

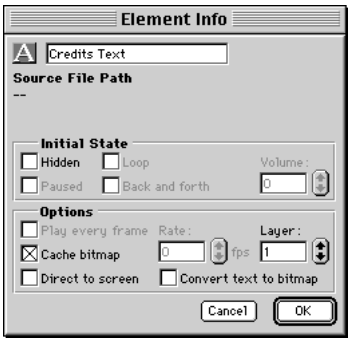
1. Click the scene in the Layout window.
2. Select the Text tool from the **Tool** palette. The cursor changes to an I-beam with a small square next to it.

*Selecting the Text tool from the **Tool** palette*

3. Create a Text element by dragging one to 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.
4. 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.
5. 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
   Burlinggame, CA 94010

6. When you are finished entering your text, select all of the text by dragging over it with the I-beam cursor. Now you can choose various text options from the **Format** menu. Pick a **Font**, **Size**, **Style**, and **Alignment** that appeals to you.
7. Return to the **Tool** palette and choose the Selection tool (the arrow). The cursor changes back to an arrow.
8. Double-click the new Text element to display its **Element Info** dialog box. Change the element's name to `Credits Text`. The dialog box should look like the one shown below.



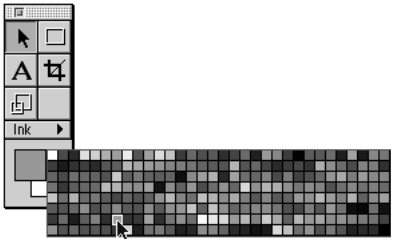
*Changing the name of the Text element using the **Element Info** dialog box*

9. Click **OK** to confirm your change and dismiss the **Element Info** dialog box.

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

1. Make sure that the Text element is selected.

2 Choose 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 choose a color as shown below.

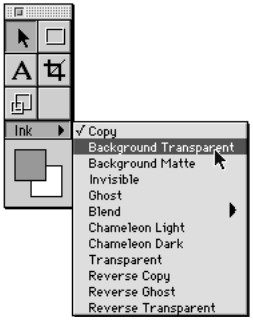


Choosing a color

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.

1 Make sure that the Text element is selected and choose **Ink** in the **Tool** palette. A pop-up menu appears. Choose **Background Transparent** as the ink option as shown below.



***Ink** pop-up menu in the **Tool** palette*

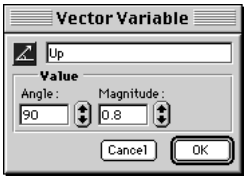
Now let's modify the text element so that it scrolls up and off the screen. To do this we will use a Vector Motion modifier. This modifier works in conjunction with the Vector Variable.

1 Drag a Vector Variable from the **Logic Modifier** palette and drop it on the Text element.

The Vector Variable modifier icon

2 Double-click the Vector Variable icon to display its configuration dialog box.

3 Change the variable's name to `Up`. Type `90` in the Angle field and `0.8` in the Magnitude field. The dialog box should look like the one shown below.



***Vector Variable** dialog box*

4 Click **OK** to confirm the changes and close the dialog box.

5 Drag a Vector Motion modifier from the **Effects Modifier** palette and drop it on the Text element.

The Vector Motion modifier icon

6 Double-click the Vector Motion modifier icon to display its configuration dialog box.

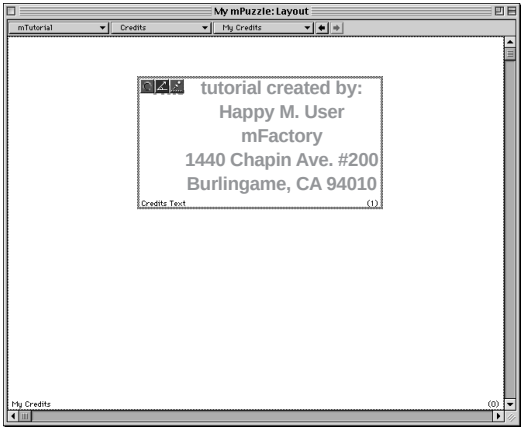
7 Change the modifier's name to `Move Up`.

8 Use the **Execute When** pop-up menu to choose the **Scene ➛ Scene Started** message.

9 Use the **Vector** pop-up menu to choose the name of the Vector Variable associated with this Vector Motion. Select the **Credits Text ➛ Up** option.

10 Click **OK** to dismiss the **Vector Motion** dialog box.

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



The My Credits scene as shown in the Layout window

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.

1 Drag a Boundary Detection Messenger from the **Logic Modifier** palette and drop it onto the Text element.

The Boundary Detection Messenger modifier icon

2 Double-click the Boundary Detection modifier icon to display its configuration dialog box.

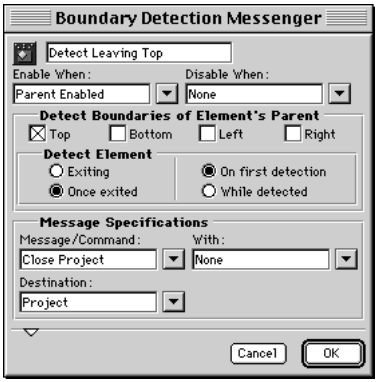
3 Change the name of the modifier to `Detect Leaving Top.`

4 In the **Detect Boundaries of Element's Parent** section, check only the **Top** check box. The **Bottom**, **Left**, and **Right** options should be unchecked.

5 In the **Detect Element** section, click the **Once Exited** and **On first detection** buttons.

6 In the **Message Specifications** section, use the **Message/Command** pop-up menu to choose the **Project ➛ Close Project** command.

7 Use the **Destination** pop-up menu to choose **Project** as the destination for the command. The dialog box should look like the one shown below.



Boundary Detection Messenger *dialog box*

8 Click **OK** to confirm the changes and dismiss the **Boundary Detection Messenger** dialog box.

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 end user the ability to abort the play of the credits.

1 Drag a Messenger modifier from the **Logic Modifier** palette and drop it on the My Credits scene.

The Messenger modifier icon

2 Double-click the Messenger icon to display its configuration dialog box.

3 Change the name of the `Messenger to Close Title.`

4 Leave the **Execute When** message set to its default (**Mouse Up**). Use the **Message/Command** pop-up menu to choose **Project ➛ Close Project.** Use the **Destination** pop-up menu to choose **Project.** Now, if an end user clicks on the title scene before the credits have finished rolling, the project just ends. The dialog box should look like the one shown below.

Messenger *dialog box*

5 Click **OK** to dismiss the dialog box.

Using the shared scene

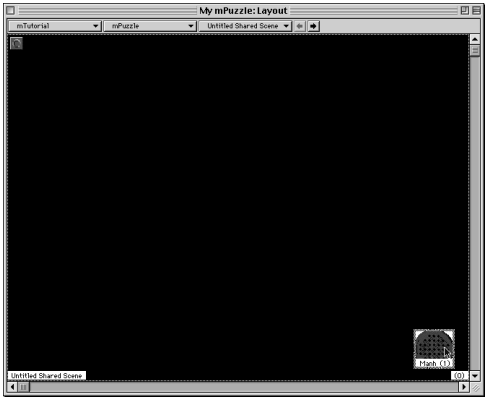
You might have noticed that even though we have created a fully-functional credits scene, there's no way for it to be activated from earlier scenes in the project!

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.

1 Use the controls at the top of the Layout window to navigate to the shared scene of the mPuzzle subsection. To do this, choose mPuzzle from the **Subsection** pop-up menu

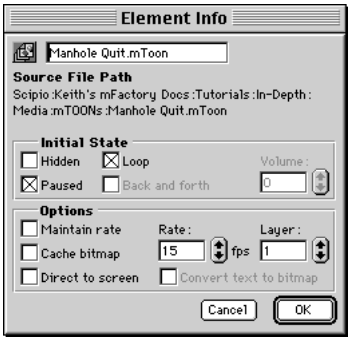
(the second pop-up menu from the left that currently reads Credits). Then choose Untitled Shared Scene from the **Scene** pop-up menu (the third pop-up menu from the left).

2. If the **Asset** palette is not visible, display it by choosing **Asset Palette** from the **View** menu.
3. Drag the Manhole Quit.mToon asset from the **Asset** palette and drop it on the scene. A new Graphic Element containing the manhole mToon appears.
4. 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.



Positioning the Manhole Quit.mToon element in the lower right of the shared scene

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



***Element Info** dialog box*

7. Click **OK** to confirm the change and close the dialog box.

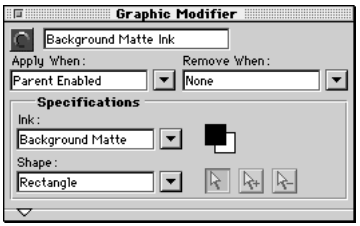
Let's make the manhole's background transparent.

1. Drag a Graphic modifier from the **Effects Modifier** palette and drop it on the Manhole Quit.mToon element.

The Graphic modifier icon

2. Double-click the Graphic modifier's icon to display its configuration dialog box.

3 Change the modifier's name to `Background Matte Ink`. Use the **Ink Effect** pop-up menu to choose **Background Matte.** The dialog box should look like the one shown below.



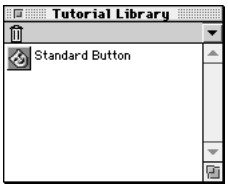
Graphic Modifier *dialog box*

4 Click the dialog box's **Close** box to accept the changes and dismiss the dialog box.

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 mTropolis components (for example, sections, subsections, scenes, elements, behaviors, and modifiers) in a file that is separate from the project.

1 Choose **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.



The ***Tutorial Library*** *palette*

2 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. 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.

1 Drag a Messenger from the **Logic Modifier** palette and drop it on the Manhole Quit.mToon element.

The Messenger modifier icon

2 Double-click the Messenger icon to display its configuration dialog box.

3 Change the name of the Messenger to `Un-Highlight`.

4 Use the **Execute When** pop-up menu to choose **Author Messages ➛ Un-Highlight.** This author message was automatically created when you added the **Standard** button behavior to your project.

5 Use the **Message/Command** pop-up menu to choose the **Set Attribute ➛ Set cel** command.

6 Choose the contents of the **With** pop-up menu 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.

7 Leave the **Destination** pop-up menu set on **Element**, its default.

8 The dialog box should look like the one shown below.

*Configuring the Un-Highlight **Messenger** modifier dialog box*

9 Click **OK** to close the dialog box.

10 Now place another Messenger modifier on the manhole element.

11 Double-click the new Messenger icon to display its configuration dialog box.

12 Change the name of the Messenger to `Highlight`.

13 Use the **Execute When** pop-up menu to choose **Author Messages ➛ Highlight.** This Author Message was automatically created when you added the **Standard** button behavior to your project.

14 Use the **Message/Command** pop-up menu to choose the **Set Attribute ➛ Set cel** command.

15 Select the contents of the **With** pop-up menu 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.

16 Leave the **Destination** pop-up menu set to **Element**, its default.

17 The dialog box should look like the one shown below.

*Configuring the Highlight **Messenger** modifier dialog box*

18 Click **OK** to close the dialog box.

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

1 Drag a Change Scene modifier from the **Logic Modifier** palette and drop it on the Manhole Quit.mToon element.

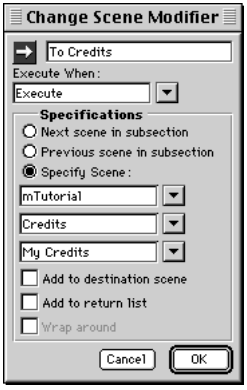
The Change Scene modifier icon

2 Double-click the Change Scene modifier to display its configuration dialog box.

3 Change the name of the modifier to `To Credits`.

4 Use the **Execute When** pop-up menu to choose the **Author Messages ➛ Execute** message.

5 In the **Specifications** section of the dialog box, click the **Specify Scene** button. Three pop-up menus become active. These menus can be used to choose the section, subsection, and scene that you will change. Choose the **mTutorial** section, **Credits** subsection, and **My Credits** scene. The dialog box should look like the one shown below.



*Configuring the **Change Scene Modifier** dialog box to go to the **My Credits** scene*

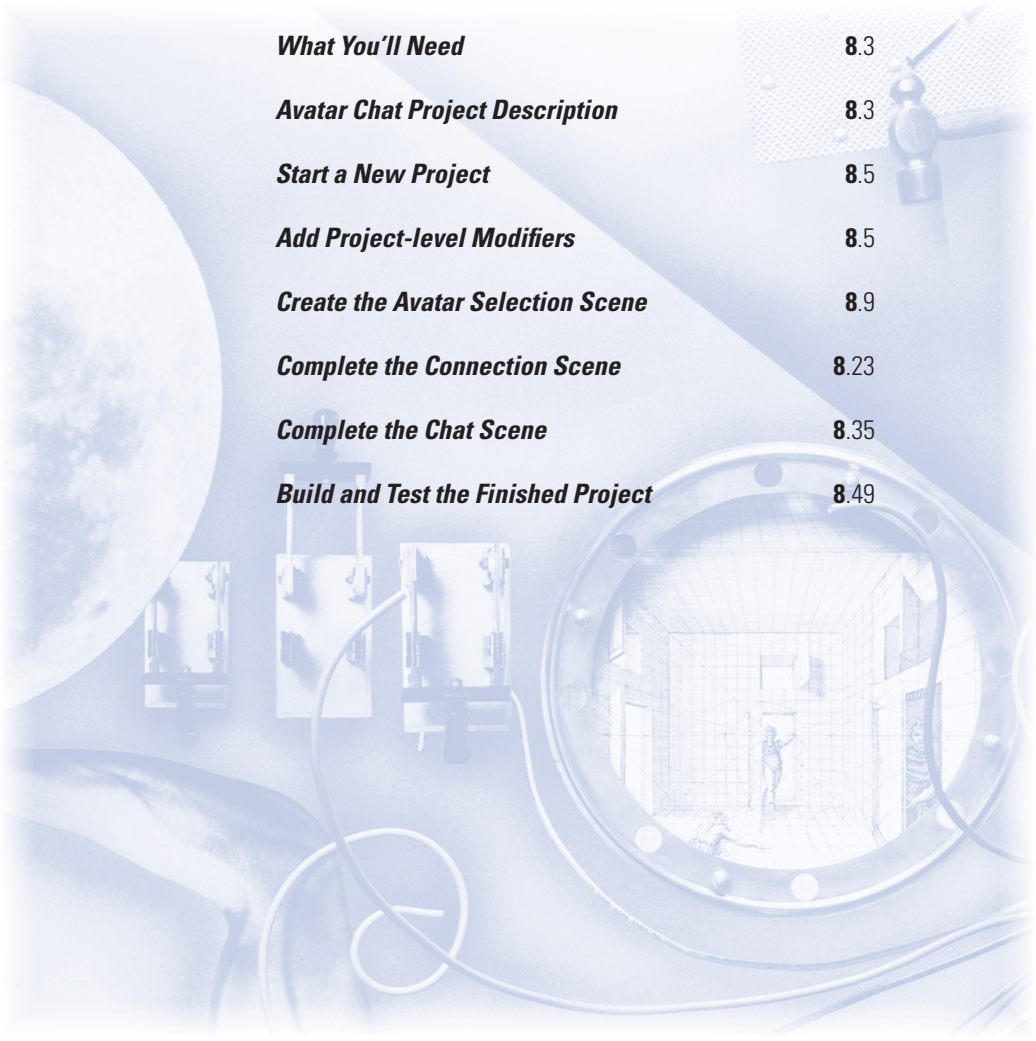
6 Click **OK** to accept the changes and close the **Change Scene Modifier** dialog box.

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. When the credits are through rolling (or if you click the scene), the project ends and mTropolis returns to edit mode.

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 tutorial project!

8 Network Tutorial — Avatar Chat



8. Network Tutorial — Avatar Chat

This tutorial shows the creation of a network-enabled mTropolis title. The project created in this tutorial allows two end users on different computers to type messages to each other using “avatars” (on-screen characters that represent the user). This is the most complicated tutorial included with mTropolis. Before attempting this tutorial, you should have completed the QuickStart Tutorial — A Simple Slideshow (found in Chapter 6) and the In-Depth Tutorial— mPuzzle (found in Chapter 7), or have previous experience with mTropolis programming.

What You'll Need

You'll need the following things before starting this tutorial.

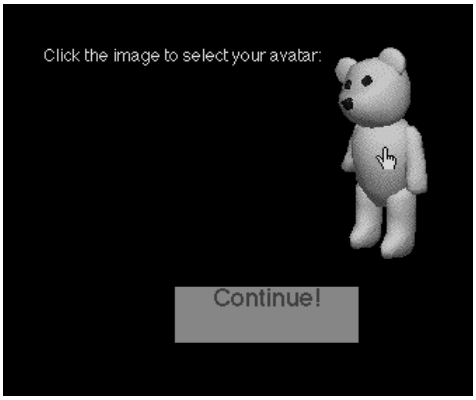
- mTropolis must be installed on your machine. Installation instructions are in the "Read Me First!" file on the mTropolis CD–ROM.
- The tutorial files are installed by default when mTropolis is installed. **Tutorial** files are in the **Tutorials** folder of the mTropolis installation. If the tutorial files have not been installed, run the installer from the mTropolis CD–ROM, or drag the Tutorials folder from the CD to your hard disk.
- This project uses 8-bit media. For best performance, make sure your monitor is set to display 256 colors.
- To use and test the avatar chat project, it's helpful to have access to a second computer on a local area network (using TCP/IP) or via an Internet connection. However, a second, networked computer is *not* required to follow the tutorial and create the network chat title.

Avatar Chat Project Description

Let's begin by looking at the completed avatar chat project. To use this project, you'll need two computers connected by a TCP/IP network. It's also helpful if both machines are physically near each other, or if you have a friend to help you by playing the title on the second machine!

1. We'll assume that you've installed mTropolis on one of the computers. The second computer must also have the mTropolis player installed. Use the mTropolis CD–ROM to install the player on the second machine. Because the avatar chat title is a cross-platform title, it doesn't matter if the second computer is a Mac OS or Windows 95/NT machine. Players can also be copied directly from the mTropolis CD–ROM's **mTropolis Players** folder. Make sure to copy both the player application and its associated **mPlugIns** folder as described in the **Deploying mTropolis Players** file (also found in the **mTropolis Players** folder).
2. Copy the **NetChat.mfx** file to the second computer. This file is the built title version of the avatar chat tutorial. For Mac OS machines, this file can be found in the **Network Chat** folder (contained in the **Tutorials** folder) on the mTropolis CD–ROM. For Windows machines, this file can be found in the **Doc** folder on the mTropolis CD–ROM.
3. Start the avatar chat title on each computer by dragging the **NetChat.mfx** icon onto the **mTropolis Player** or **mtplay32.exe** icon (the icon that represents the mTropolis player application). Alternatively, if you installed the **NetChat.mfx** file in the same folder as the mTropolis player application, you can simply double-click the player icon and the **NetChat.mfx** title will automatically play.

4 The NetChat title should start playing. The first scene (see figure below) allows you to select the avatar you will use to chat with your friend. Click the image of the bear to choose one of 8 different bears (there are left- and right-facing bears in 4 colors). When your desired avatar is visible, click the green **Continue!** button.



Selecting an avatar in the first scene

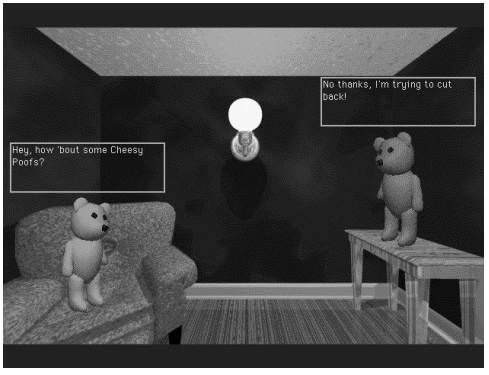
5 A new scene appears (see figure below). This scene contains an editable text element (the white area below the "Enter your friend's IP address:" prompt). Click inside the text element and type the IP address of the other machine. This address must be a valid (and correct!) IP address (for example, 206.86.76.62). After entering the address, click the **Continue!** button.

Note that only one player needs to enter the other player's IP address. mTropolis is able to obtain the other IP address automatically.

Entering the IP address of your friend's machine

6 If the IP address is entered correctly, both machines play a "popping" sound and a new scene appears (see figure below). If the other player has not yet chosen an avatar, an informational prompt appears on both machines and the new scene appears when the slower player selects an avatar and clicks the **Continue!** button.

7 When the chat scene is displayed, you'll see both of the chosen avatars (hopefully, you've picked different avatars to avoid confusion). You can move your own avatar by dragging it around the screen. As you move your avatar, it also moves around on your friend's screen. To chat, click inside the chat bubble that is above your avatar's head. Type the message you want to send, then click outside the chat bubble. Messages typed by your friend appear in the chat bubble above his/her avatar.



Discussing pressing social issues in NetChat

8 When you get tired of chatting with your friend and jumping around on the furniture, quit the NetChat title by pressing ⌘-Q (on Mac OS) or Alt-F4 (on Windows).

Start a New Project

Now we'll recreate the avatar chat project. Launch the mTropolis editor. Start a new project by choosing **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.

Add Project-level Modifiers

This project needs a number of modifiers at the project level. To enable network messaging functionality, the project needs a Net Messaging Service modifier. In addition, we need a number of variables that can be accessed from all scenes in the project. By putting these variables at the project level (the topmost structural level), they become global — they can be accessed from anywhere in the project.

Add the Net Messaging Service

Let's start by adding the Net Messaging Service to the project component.

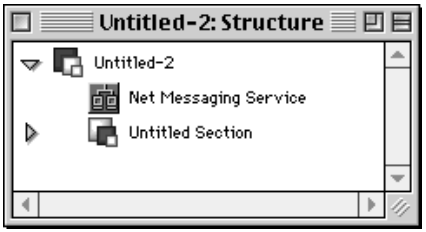
1 Choose **Structure Window** from the **View** menu (⌘-2). Only the structure view can be used to add modifiers to the project component itself. The structure view shows the project component and the one section component in the new project.

2 If it is not already visible, display the **Network Modifier** palette by choosing **Modifier Palettes ➛ Network** from the **View** menu.

3 Drag the Net Messaging Service modifier from the **Network** palette and drop it on the project component in the Structure window.

The Net Messaging Service modifier icon

The Structure window view should look like the one shown below.



4 Double-click the Net Messaging Service icon to open its configuration dialog box.

5 Change the service's name from Net Messaging Service to `NetService` (note that there are no spaces in this new name). Later, we'll refer to this modifier in a Miniscript statement. Making its name a single word now will make that script simpler.

6 By default, the Net Messaging Service is always active. It enables on Project Started and never disables. In our project, this is the desired behavior, so there's no need to change the defaults. For network messaging to work, the Net Messaging Service must be present in the project and enabled. Disabling the Net Messaging Service disables the network messaging capabilities of a project.

Your **Net Messaging Service** dialog box should look like the one shown below. Click **OK** to dismiss the dialog box.

Add Global Modifiers to the Project

We'll need three global modifiers for this project: a String Variable to store the network address of the remote computer (the computer being used by the friend we want to chat with), and two Integer Variables to store the mToon cel numbers associated with the avatars selected by each end user.

1 If it is not already visible, display the **Logic Modifier** palette by choosing **Modifier Palettes ➛ Logic** from the **View** menu.

2 Drag a String Variable from this palette and drop it on the project component in the Structure window.

The String Variable icon

3 Click the name of the String Variable in the Structure Window. The name becomes editable. Change its name from the default String Variable modifier to `FriendIP`. This variable will be used to hold the IP address of the remote computer.

4 Drag an Integer Variable from the **Logic** palette and drop it on the project component in the Structure window.

The Integer Variable icon

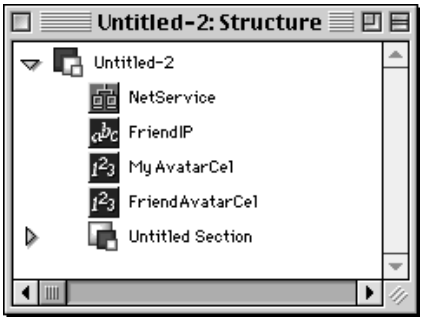
5 Click the name of the Integer Variable in the Structure Window. The name becomes editable. Change its name from the default Integer Variable to `MyAvatarCel`. This variable will be used to hold the number of the mToon cel selected as an avatar.

6 Drag another Integer Variable from the **Logic** palette and drop it on the project component in the Structure window.

The Integer Variable icon

7 Click the name of the Integer Variable in the structure window. The name becomes editable. Change its name from the default Integer Variable to `FriendAvatarCel`. This variable will be used to hold the number of the mToon cel selected as an avatar by the end user of the remote network project.

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



Add Set Modifiers to the Project

Now we'll add two Set modifiers to the project. These modifiers will set the default values of the MyAvatarCel and FriendAvatarCel variables when the project starts.

1 Drag a Set modifier from the **Logic** palette and drop it on the project component in the Structure window.

The Set modifier icon

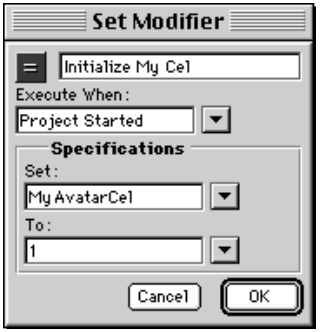
2 Double-click the modifier to display the **Set Modifier** dialog box.

3 Change the modifier's name to `Initialize My Cel`.

4 Use the **Execute When** pop-up menu to select **Project ➛ Project Started.**

5 In the **Specifications** section, use the **Set** pop-up menu to choose the variable **MyAvatarCel** from the cascading list of variables.

6 Highlight the text of the **To** pop-up menu (it currently reads **None**) and type `1` (the number "one"). Your **Set Modifier** dialog box should look like the one shown below.



7 Click **OK** to dismiss the **Set Modifier** dialog box.

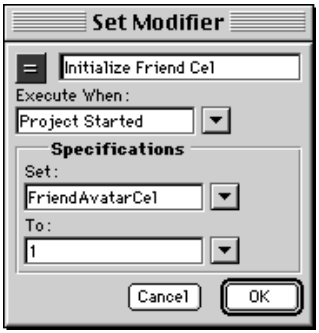
8 In the Structure window, click the Initialize My Cel icon to ensure that this modifier is selected.

9 Choose **Duplicate** from the **Edit** menu (⌘-D). A copy of the modifier appears below the original in the Structure Window.

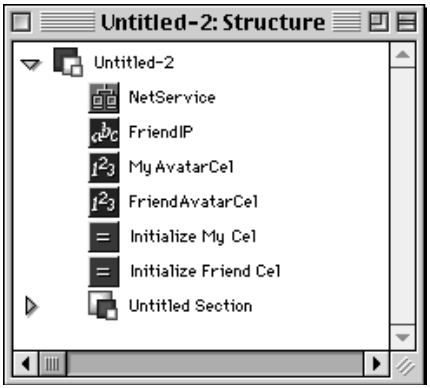
10 Double-click the new modifier to open its configuration dialog box.

11 Change the modifier's name from Initialize My Cel to `Initialize Friend Cel`.

12 Use the **Set** pop-up menu to choose the variable **FriendAvatarCel** from the cascading list of variables. Your **Set Modifier** dialog box should look like the one shown below.



13 Click **OK** to dismiss the **Set Modifier** dialog box. Your Structure window should look like the one shown below.



Save the Project

Now would be a good time to save your work.

1 Choose **Save** from the **File** menu (⌘-S).

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

3 If, at any point, you want to restore the project to a previously-saved version, choose **Open** from the **File** menu to load the saved file.

4 Notice that the name of the project component, and the titles of the Layout and Structure windows, change to reflect the name of the project file.

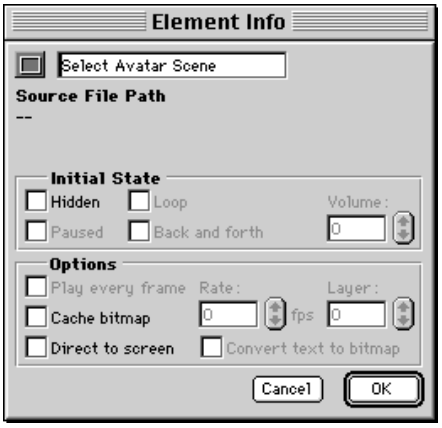
Create the Avatar Selection Scene

There will be three scenes in our finished project. In the first scene, the end user can select the avatar that they will use to represent themselves in the chat space. Each avatar is actually a single cel in an mToon.

Rename the Scene

Click the Layout window to bring it to the front. The Layout window shows the first, and only, scene in the project.

1. Double-click the scene inside the Layout window to display its **Element Info** dialog box.
2. Change the name of the scene to `Select Avatar Scene`. Your scene's **Element Info** dialog box should look like the one shown below.



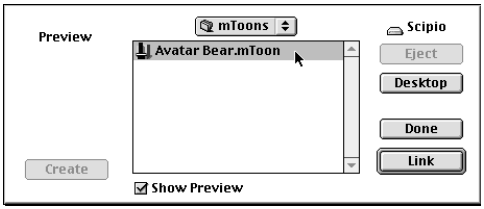
3. Click **OK** to dismiss the **Element Info** dialog box.

Add the Avatar mToon Element

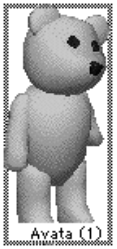
Now we'll add the mToon element that lets the end user select an avatar.

1. Select the Graphic tool in the **Tool** palette as shown below.

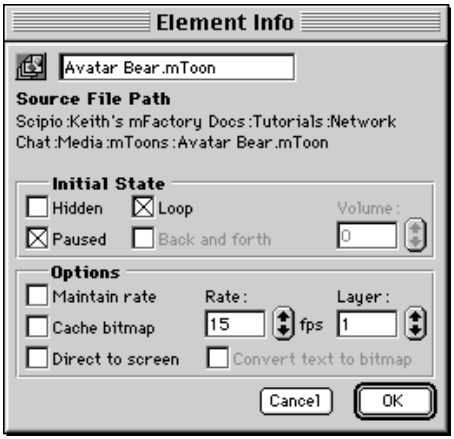
2. Click and drag the cursor somewhere near the center of the scene to create a small graphic element.
3. Choose **Link Media** ➛ **File** (⌘-L) from the **File** menu. A file selection dialog box appears.
4. In the file dialog box, select the file **Avatar Bear.mToon** from the **mToons** folder, found inside the **Media** subfolder of the **Network Chat** folder. Click **Link** to link the mToon to the graphic element.



5 The element updates to show the linked media as shown below.



6 Double-click the element to display its **Element Info** dialog box. Click the **Paused** check box. Your dialog box should look like the one shown below.



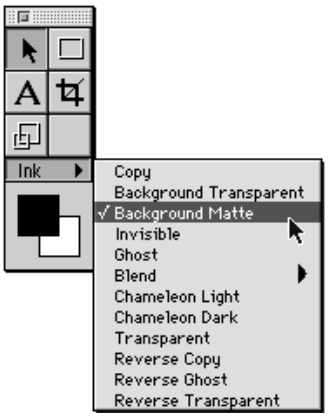
7 Click **OK** to dismiss the **Element Info** dialog box.

Program the Avatar Bear

Now let's add some basic programming to the Avatar Bear.mToon element to make it a clickable selector.

1 Click the Avatar Bear.mToon element to ensure that it is selected.

2 Choose **Background Matte** from the **Tool** palette's **Ink** pop-up menu. A Graphic modifier icon appears on the Avatar Bear element.



3 Drag a Miniscript modifier from the **Logic** palette and drop it on the Avatar Bear.mtoon element.

The Miniscript modifier icon

4 Double-click the Miniscript modifier icon to display its configuration dialog box. Change the modifier's name to `Change Cel`.

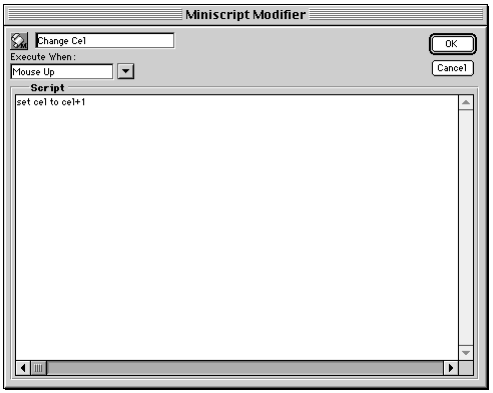
5 We want this Miniscript to be executed on Mouse Up, so we don't need to change the **Execute When** pop-up menu.

6 Enter the following statement in the **Script** field:

```
set cel to cel + 1
```

This script causes the cel displayed by the mToon to change each time the end user clicks on the mToon.

7 Your Miniscript modifier should look like the one shown below. Click **OK** to dismiss the **Miniscript Modifier** dialog box.



8 Drag a Messenger modifier from the **Logic** palette and drop it on the Avatar Bear element.

The Messenger modifier icon

9 Double-click the Messenger modifier icon to display its configuration dialog box. Change the modifier's name to `Change Cel`.

10 Use the **Message/Command** pop-up menu to select **Get Attribute ➛ Get cel.**

11 Use the **With** pop-up menu to select **MyAvatarCel** from the cascading list of variables off of the project name item. Your **Messenger Modifier** dialog box should look like the one shown below.

12 This messenger retrieves the number of the currently-displayed mToon cel each time the end user clicks and stores it in the MyAvatarCel global variable. Click **OK** to dismiss the **Messenger Modifier** dialog box.

Add a Text Prompt Next to the Avatar

Now let's add a text prompt that tells the end user what to do in this scene.

1 Select the Text tool from the **Tool** palette. The cursor changes to an I-beam cursor. Drag on the scene to create a wide text element to the left of the Avatar Bear.mtoon element.

*Selecting the Text tool in the **Tool** palette*

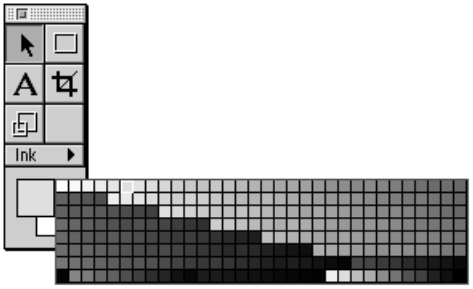
2 When you release the mouse button, an insertion point appears in the new text element. Type the following text into the element:

```
Click the image to select your avatar:
```

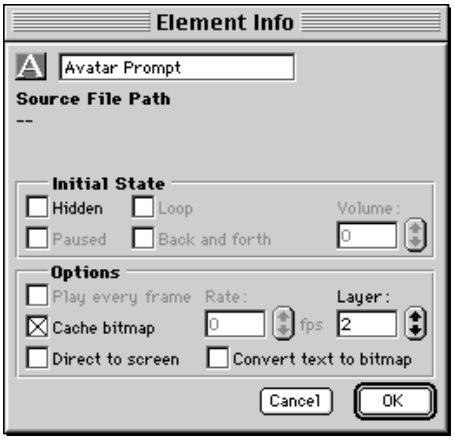
3 Click outside the text element to end text entry. The cursor becomes a pointer again. If your text element is not big enough to display the full text of the prompt, simply resize it by dragging its borders with the Selection tool.

4 Click the text element to ensure that it is selected. Use the **Ink** pop-up menu on the **Tool** palette to choose **Background Transparent.** A Graphic modifier appears on the text element.

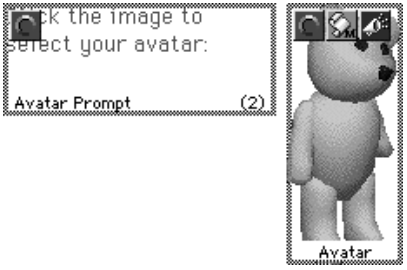
5 Click and drag on the **Foreground Color Swatch** in the **Tool** Palette to display a palette of foreground colors as shown below. Choose a light foreground color that appeals to you (we used a bright yellow in the original version of the NetChat title). Note that in the Layout window, the scene's background appears white but during run-time mode the shared scene (which is black by default) shows through. Pick a color that will show up well against black.



6 The text changes color. Double-click the text element to display its **Element Info** dialog box. Change the element's name to `Avatar Prompt`. Your **Element Info** dialog box should look like the one shown below. Click **OK** to dismiss the dialog box.



7 Your text prompt and Avatar Bear.mToon element should look similar to the ones shown below. If you need to reposition the elements, simply drag them to move them around the scene.



8 Save your project again by pressing ⌘-S. Remember to save the project periodically as you go through the tutorial!

Test the Project

Press ⌘-T to run the project. You'll see the text prompt and the avatar mToon. Click the mToon to change its cel. Note that since the

mToon is configured to loop, after the eighth avatar is displayed the first one is shown again. To return to edit mode, press ⌘-T again.

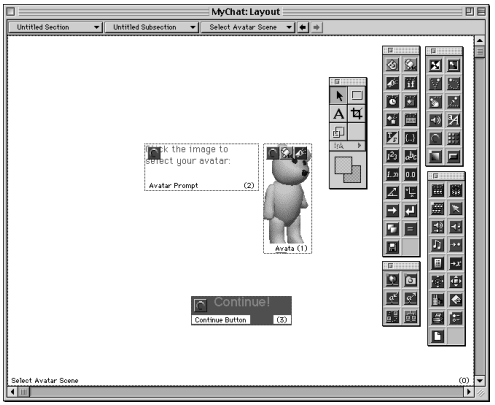
Add the Continue Button

Now let's add the button that allows the end user to continue to the next scene when he/she is through selecting an avatar.

1. Select the Text tool from the **Tool** palette.
2. Drag out a text element, toward the bottom of the scene, to be used as our continue button (make it about twice as wide as the Avatar Bear.mToon).
3. When you release the mouse button, an insertion point appears in the new text element. Type the following text into the element:

   `Continue!`

4. While the insertion point is still inside the text element, drag over the Continue! text to select it.
5. Use the **Font**, **Size**, and **Alignment** options in the **Format** menu to choose a type style that appeals to you. In the original NetChat title, we used Helvetica, 18 points, Center aligned.
6. Click outside the text element to end text entry. The cursor becomes a pointer again. If you want to resize the text element, simply drag its borders with the Selection tool.
7. Double-click the text element to display its **Element Info** dialog.
8. Change the element's name to `Continue Button`. Also click the **Convert Text to Bitmap** check box. We chose this option so that end users of this title on other machines will not have to have the font used by this button installed. When built into a title, this element will be converted into a bitmapped picture that can be displayed by any machine.
9. Use the **Foreground Color Swatch** and **Background Color Swatch** on the **Tool** palette to choose a foreground and background color for this button. The foreground color is the color of the text. The background color is the color of the area inside the element's boundaries. In the original NetChat title, we chose bright red on green for this button. A Graphic modifier appears on the continue button element.
10. Your Layout window should look something like the one shown below.



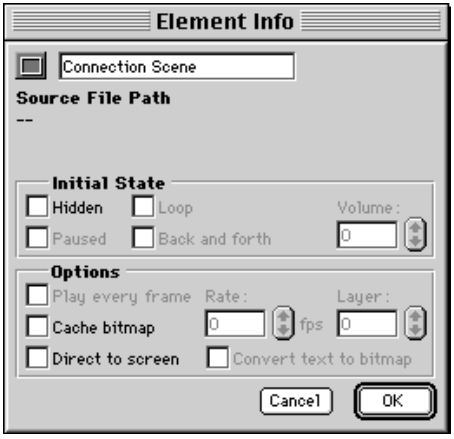
Create New Scenes

Eventually, we'll program the Continue Button element so that it changes to the next scene when clicked. First we need to create the next scene. While we're at it, we'll also create the third scene, which is where end users will be able to chat.

1 Choose **New Scene** from the Layout window's **Scene** pop-up menu. This is the pop-up menu that currently reads **Select Avatar Scene.**

2 A new scene is created and the Layout window changes to show that new, untitled scene.

3 Double-click the new scene to display its **Element Info** dialog box. Change its name to `Connection Scene`. Your dialog box should look like the one shown below. Click **OK** to dismiss the dialog box.



4 The name of the scene changes, as reflected in the Layout window's **Scene** pop-up menu.

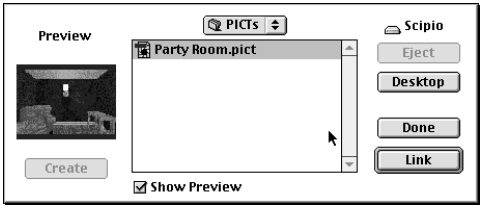
5 Now we'll create the third and final scene. Again, choose **New Scene** from the Layout window's **Scene** pop-up menu.

6 A new scene is created and the Layout window changes to show that new, untitled scene.

7 Click inside the Layout window to select the new, untitled scene.

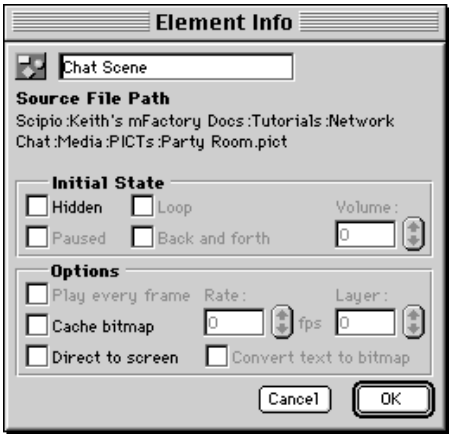
8 Choose **Link Media ➛ File** from the **File** menu. A file selection dialog box appears.

9 In the file dialog box, choose the file **Party Room.pict** from the **PICTs** folder, found inside the **Media** subfolder of the **Network Chat** folder. Click **Link** to link the PICT to the graphic element.



10 The scene updates to show the new media linked to the scene. The name of the scene also changes.

11 Double-click the scene to display its **Element Info** dialog box. Change its name from Party Room.pict to `Chat Scene`. The **Element Info** dialog box should look like the one shown below. Click **OK** to dismiss the **Element Info** dialog box.



Program the Continue Button

Now that we've created all of our scenes, let's return to the Select Avatar Scene and finish programming the **Continue** button.

1 Use the **Scene** pop-up menu or the previous scene arrow at the top of the layout view to navigate back to the Select Avatar Scene.

2 In the Select Avatar Scene, drag a behavior modifier from the **Logic** palette and drop it on the Continue Button element.

The Behavior modifier icon

3 Double-click the Behavior icon to display the behavior window.

4 Change the behavior's name to `Default Button Behavior`.

5 Click the behavior's **Switchable** check box. The **Enable When** and **Disable When** pop-up menus become active. The default enable message, **Parent Enabled**, is fine and does not need to be changed. However, we do need to change the disable message.

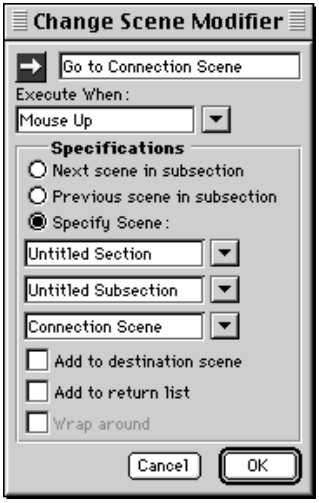
6 Use the **Disable When** pop-up menu to choose **Author Messages** ➝ **New Author Message.** The **New Author Message** dialog box appears. Type `Connect` in the dialog box's Name field. Your **New Author Message** dialog box should look like the one shown below. Click **OK** to dismiss the **New Author Message** dialog box.

7 The **Disable When** pop-up menu changes to read Connect.

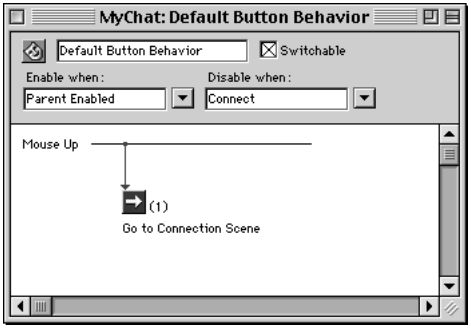
8 Drag a Change Scene modifier from the **Logic** palette and drop it into the Default Button behavior window.

The Change Scene modifier icon

9 Double-click the Change Scene modifier icon to display its configuration dialog box. Change the modifier's name to `Go to Connection Scene`. In the Specifications section, click the **Specify Scene** button and choose **Connection Scene** from the **Scene** pop-up menu (the last pop-up menu in this dialog box). Your dialog box should look like the one shown below. Click **OK** to dismiss the **Change Scene Modifier** dialog box.



10 Your Default Button Behavior window should now look like the one shown below. That's all the programming this behavior requires, so close the Default Button Behavior window by clicking its **Close** box.



More Button Programming

The Default Button Behavior programs the **Continue** Button to act as a change scene button. When the end user clicks the button, the Change Scene modifier inside the behavior is triggered and the scene changes to the Connection Scene.

However, this is not always what we want the button to do. Remember that two end users will be running two different copies of this program at the same time. They will each be presented with a scene that allows them to choose an avatar. After choosing an avatar, they see the Connection Scene where one of them will enter the other end user's IP address.

If one of the end users is faster than the other and chooses an avatar, goes to the Connection Scene, enters an IP address, and clicks the Connect! button while the other end user is still pondering his or her avatar choices, there's no point in having the slower end user see the Connection Scene.

As you'll see when we program the Connection Scene, that scene's Connect button sends an author message called "Connect" to the remote project. It sends the mToon cel selected by the end user of that project as "with" data. If the receiving project is still on the Select Avatar scene, it will respond with a message called "Please Wait" that instructs the connecting project to wait while the slower end user chooses their avatar. When the slower end user has chosen his/her avatar (the end user has clicked the Continue button), the receiving project sends out a "Send Avatar" message that tells the connecting project that it is all right to proceed to the Chat Scene.

The rest of the programming we add to the Select Avatar Scene handles this case.

1. Drag another Behavior modifier from the **Logic** palette and drop it on the Continue Button element.

The Behavior modifier icon

2. Double-click the Behavior icon to display the behavior window.
3. Change the Behavior's name to `Complete Connection`.
4. Click the behavior's **Switchable** check box. The **Enable When** and **Disable When** pop-up menus become active.
5. Use the behavior's **Enable When** pop-up menu to choose **Author Messages** ➛ **Connect.** The Disable When default of **None** is fine and doesn't need to be changed.
6. If it is not already visible, display the **Network Modifier** palette by choosing **Modifier Palettes** ➛ **Network** from the **View** menu.
7. Drag a Net Messenger modifier from the **Network** palette and drop it inside the Complete Connection window. The Net Messenger is very similar to the standard mTropolis messenger. It can be used to send any mTropolis message to another mTropolis project over a TCP/IP network.

The Net Messenger icon

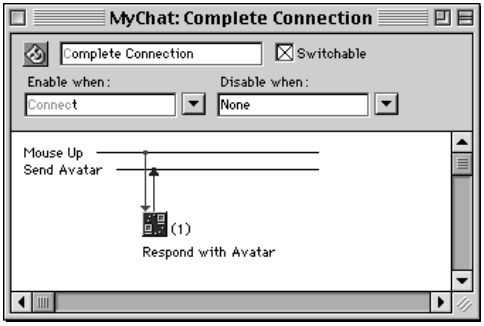
8. Double-click the Net Messenger icon to display the **Net Messenger** dialog box. Change the modifier's name to `Respond with Avatar`.
9. The **Execute When** default of Mouse Up is fine, don't make any changes to that pop-up menu.
10. Use the **Message/Command** pop-up menu (which currently reads None) to choose **Author Messages** ➛ **New Author Message.** The **New Author Message** dialog box appears. In this dialog box's **Name** field, enter `Send Avatar`. Your **New Author Message** dialog box should look like the one shown below. Click **OK** to dismiss the **New Author Message** dialog box.

11 Use the **With** pop-up menu of the **Net Messenger** dialog box to choose **MyAvatarCel** from the cascading list of variables shown off of the menu item that contains the project's name.

12 Use the **Destination** pop-up menu to choose **Active Scene** as the destination for the Send Avatar message.

13 Use the **Host** pop-up menu to select **FriendIP** from the cascading list of variables shown off of the menu item that contains the project's name. Your **Net Messenger** dialog box should look like the one shown below. Click **OK** to dismiss this dialog box.

14 Your Complete Connection behavior window should now look like the one shown below. We're through programming this behavior, so close it by clicking its **Close** box.



Add the "Already Connected" Prompt

If the Select Avatar Scene receives the "Connect" message from a remote project, it would be nice to tell the end user that their friend is already connected and waiting for him/her to choose an avatar.

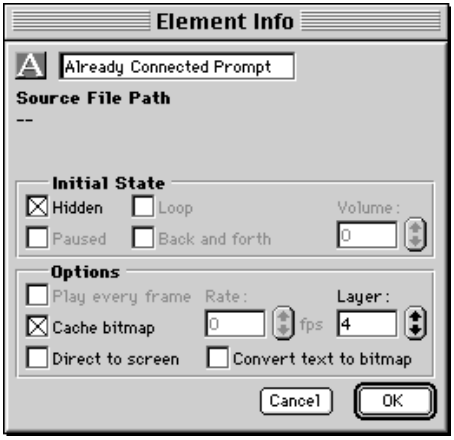
1 Select the Text tool from the **Tool** palette. The cursor changes to an I-beam cursor. Drag on the scene to create a wide text element above the Avatar Prompt and Avatar Bear elements.

2 When you release the mouse button, an insertion point appears in the new text element. Type the following text in the element:

```
Your friend is already connected.
Please select an avatar and click
the Continue button.
```

3 Click outside the text element to end text entry. The cursor becomes a pointer again. If your text element is not big enough to display the full text of the prompt, simply resize it by dragging its borders with the Selection tool.

4 Double-click the new text element to display its **Element Info** dialog box. Change the element's name to `Already Connected Prompt`. We also want this element to be hidden initially and show itself only when told to, so click the **Hidden** check box. Your **Element Info** dialog box should look like the one shown below. Click **OK** to dismiss the **Element Info** dialog box.



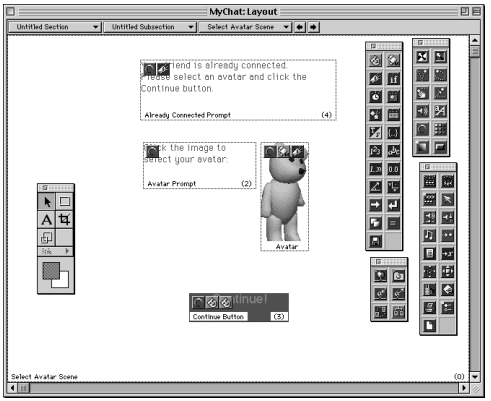
5 We want this prompt to have the same appearance as the Avatar Prompt. We can reuse the Graphic modifier programming that we used earlier. Hold down the Option key while dragging the Graphic modifier from the Avatar Prompt and dropping it on the **Already Connected Prompt.** A copy of the Graphic modifier is added to the Already Connected Prompt.

6 Let's add the programming that will make this element display itself when it receives the Connect message. Drag a Messenger modifier from the **Logic** palette and drop it on the Already Connected Prompt.

The Messenger modifier icon

7 Double-click the Messenger modifier icon to display its configuration dialog box. Change its name to `Show on Connect`. Use the **Execute When** pop-up menu to choose **Author Messages ➛ Connect.** Use the **Message/Command** pop-up menu to choose **Element ➛ Show.** Your **Messenger** dialog box should look like the one shown below. Click **OK** to dismiss the dialog box.

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



Add the "React to Connect" Programming to the Scene

There is one final piece of programming we need to add to the Select Avatar Scene. This programming will process a Connect message if it is received.

1. Drag a Behavior modifier from the **Logic** palette and drop it on the Select Avatar Scene. Be careful to drop it on the scene itself, not on any of the other elements that we previously added to the scene. The modifier attaches itself to the upper left corner of the scene.

The Behavior modifier icon

2. Double-click the Behavior icon to display the Behavior window. Change the behavior's name to `React to Connect`.
3. Drag a Miniscript modifier from the **Logic** palette and drop it in the React to Connect window.

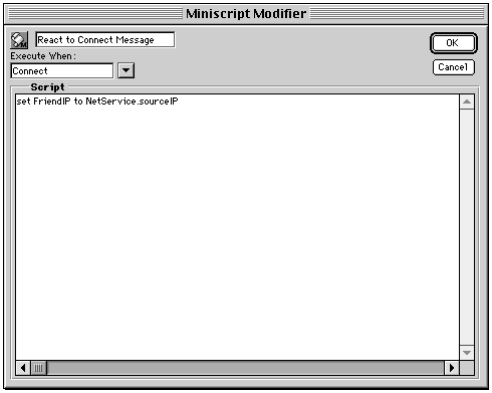
The Miniscript modifier icon

4. Double-click the Miniscript modifier icon to display its configuration dialog box. Change its name to `React to Connect Message`. Use the **Execute When** pop-up menu to choose **Author Messages ➛ Connect.** Enter the following statement in the Script field:

```
set FriendIP to NetService.
sourceIP
```

This modifier listens for the Connect message. Because this message arrives from a machine at some remote location on a network, the Net Messaging Service modifier records the network address from which the message originates. This address is stored in the Net Messaging Service modifier's "sourceIP" attribute. The Miniscript statement above reads this attribute and stores its value in the FriendIP global variable, which we created earlier.

Your **Miniscript Modifier** dialog box should look like the one shown below. Click **OK** to dismiss the dialog box.

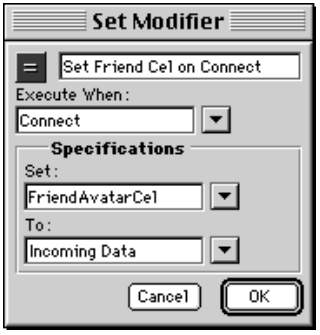


5 Drag a Set modifier from the **Logic Modifier** palette and drop it in the React to Connect window.

The Set modifier icon

6 Double-click the Set modifier icon to display its configuration dialog box. Change its name to `Set Friend Cel on Connect`. Use the **Execute When** pop-up menu to choose **Author Messages ➛ Connect.** Use the **Set** pop-up menu to choose **FriendAvatarCel** from the cascading list of variables found off of the project component's name. Use the **To** pop-up menu to choose **Incoming Data**.

Your **Set Modifier** dialog box should look like the one shown below. Click **OK** to dismiss the dialog box.



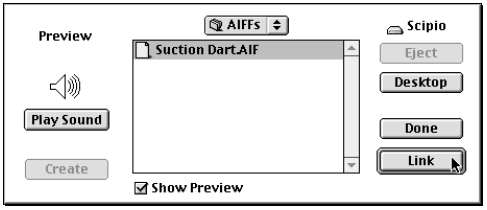
As mentioned before, when the Connect message is sent, it will be sent with the end user's avatar cel selection as "With" data. This modifier reads that data (when received, it is called "Incoming Data") and stores it in the FriendAvatarCel global variable that we created earlier.

7 If it is not already visible, display the **Effects Modifier** palette by choosing **Modifier Palettes ➛ Effects** from the **View** menu. Drag a Sound Effect modifier from the **Effects** palette and drop it in the React to Connect window.

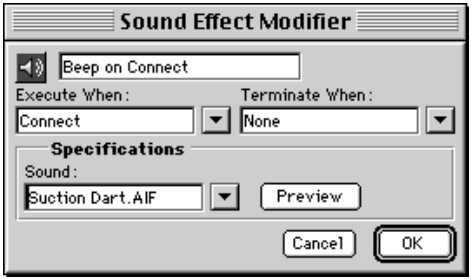
The Sound Effect modifier icon

8 Double-click the Sound Effect modifier icon to display its configuration dialog box. Change its name to `Beep on Connect`. Use the **Execute When** pop-up menu to choose **Author Messages ➛ Connect.**

9 Use the **Sound** pop-up menu to choose **Link File.** A standard file selection dialog box appears. Choose the file Suction Dart.AIF, found in the AIFFs folder, found inside the Media subfolder of the **Network Chat** folder. Click **Link** to link the sound to the modifier.



Your **Sound Effect Modifier** dialog box should look like the one shown below. Click **OK** to dismiss the dialog box.

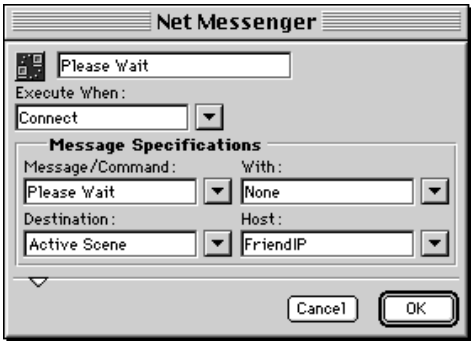


10 Drag a Net Messenger from the **Network** palette and drop it in the React to Connect window.

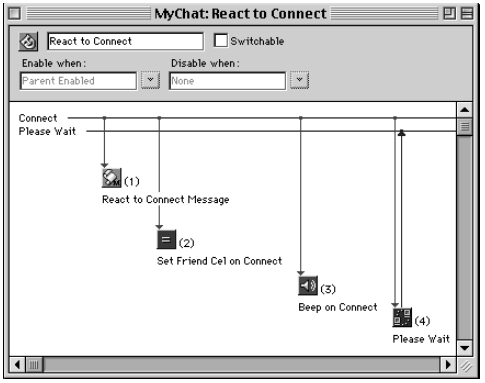
The Net Messenger icon

11 Double-click the Net Messenger icon to display its configuration dialog box. Change its name to `Please Wait`. Use the **Execute When** pop-up menu to choose **Author Messages ➛ Connect.** Use the **Message/Command** pop-up menu to choose **Author Messages ➛ New Author Message.** The **New Author Message** dialog box appears. In that dialog box's name field, enter `Please Wait`, as shown below. Click **OK** to dismiss the **New Author Message** dialog box.

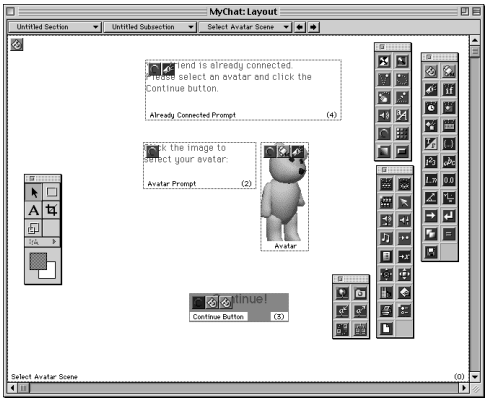
Use the **Destination** pop-up menu to choose **Active Scene.** Use the **Host** pop-up menu to choose **FriendIP** from the cascading list of variables. Your **Net Messenger** dialog box should look like the one shown below. Click **OK** to dismiss the dialog box.



12 Your React to Connect window should now look like the one shown below. We are finished programming this behavior. Dismiss the window by clicking its **Close** button.



The first scene of our project is now complete. Your scene should look similar to the one shown below. If you haven't saved your work in a while, now would be a good time!



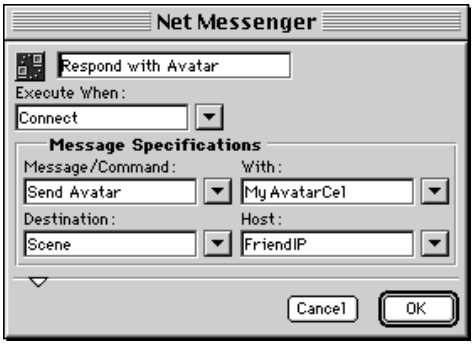
Complete the Connection Scene

Earlier in this tutorial, we created the second scene in the tutorial and named it Connection Scene. This scene is the one in which the end user can enter the network IP address for his/her friend's computer. This scene also contains the logic that completes the network connection between the two communicating projects.

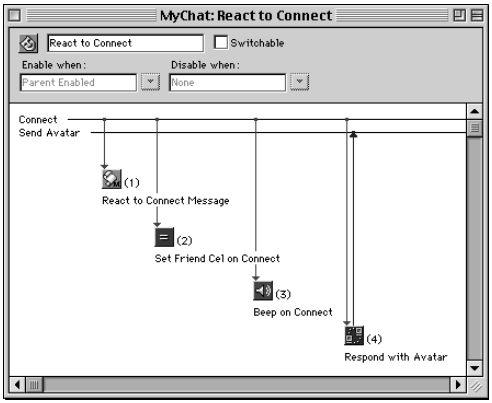
1. While the Layout window is still displaying the Select Avatar Scene, select the React to Connect behavior icon, in the upper left corner of the scene.
2. Copy this behavior to the clipboard by pressing ⌘-C or choosing **Copy** from the **Edit** menu.
3. Click the Next Scene arrow at the top of the Layout window to display the Connection Scene as shown below.

4. Press ⌘-V to paste the behavior on the Connection Scene. A copy of the behavior appears in the upper left corner of the scene.
5. We can reuse most of the programming in this behavior, but need to make one change. Double-click the behavior icon to display the new React to Connect window.
6. Double-click the **Please Wait** Net Messenger icon to display its configuration dialog box. Change its name from Please Wait to `Respond with Avatar`. Use the **Message/Command** pop-up menu to choose **Author Messages ➛ Send Avatar.** Use the **With** pop-up menu to choose **MyAvatarCel** from the cascading list of variables. Use the **Destination** pop-up menu to choose **Scene.** Use the **Host** pop-up menu to choose **FriendIP.**

   Your **Net Messenger** dialog box should look like the one shown below. Click **OK** to dismiss the dialog box.



7 The edited React to Connect behavior window should look like the one shown below. This behavior processes an incoming Connect message and sends the Send Avatar message in response. Close the React to Connect window by clicking its **Close** box.



Create the React to Send Avatar Behavior

Now we need to create the behavior that processes an incoming Send Avatar message and changes the scene to the Chat Scene. Again, we'll reuse some of the programming that we did earlier.

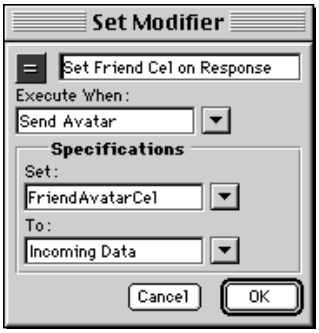
1 Click the React to Connect behavior icon to select it.

2 Duplicate the behavior by pressing ⌘-D. A copy of the behavior appears to the right of the original.

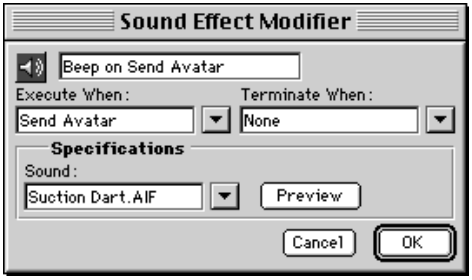
3 Double-click the copy to display its behavior window. Change its name to `React to Send Avatar`.

4 Delete the React to Connect Message Miniscript by selecting it and pressing **Delete**.

5 Double-click the Set Friend Cel on Connect Set modifier. Change its name to `Set Friend Cel on Response`. Use the **Execute When** pop-up menu to choose **Author Messages ➛ Send Avatar.** Your **Set Modifier** dialog box should look like the one shown below. Click **OK** to dismiss the dialog box.



6 Double click the Beep on Connect Sound Effect modifier. Change its name to `Beep on Send Avatar`. Use the **Execute When** pop-up menu to choose **Author Messages ➛ Send Avatar.** Your **Sound Effect Modifier** dialog box should look like the one shown below. Click **OK** to dismiss the dialog box.

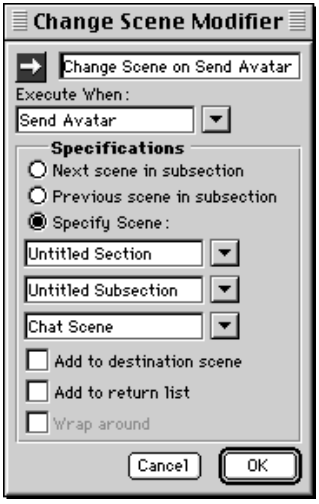


7 Double-click the Respond with Avatar Net Messenger. Change its name to `Send Go to Chat Scene`. Use the **Execute When** pop-up menu to choose **Send Avatar.** Use the **Message/Command** pop-up menu to choose **Author Messages ➛ New Author Message** and create the author message `Go to Chat Scene`. Use the **With** pop-up menu to choose **None.** Your **Net Messenger** dialog box should look like the one shown below. Click **OK** to dismiss the dialog box.

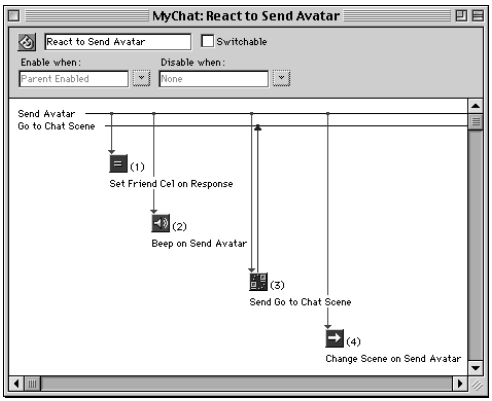
8 Drag a Change Scene modifier from the **Logic** palette and drop it in the React to Send Avatar window.

The Change Scene modifier icon

9 Double click the Change Scene modifier. Change its name to `Change Scene on Send Avatar`. Use the **Execute When** pop-up menu to choose **Author Messages ➛ Send Avatar.** In the **Specifications** section of the dialog box, click the **Specify Scene** button and use the **Scene** pop-up menu to choose **Chat Scene.** Your **Change Scene Modifier** dialog box should look like the one shown below. Click **OK** to dismiss the dialog box.



10 The React to Send Avatar behavior is now complete. Your React to Send Avatar behavior window should look like the one shown below. Dismiss this window by clicking its **Close** box.



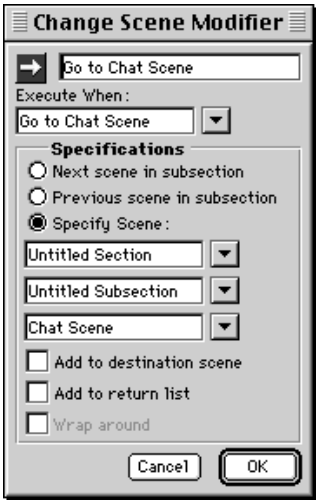
Add the Go to Chat Scene Modifier

The behavior we just created sends a message called Go to Chat Scene. We need to add the Change Scene modifier that listens for this message and changes to the Chat Scene in response.

1 Drag a Change Scene modifier from the **Logic** palette and drop it on the Connection Scene.

The Change Scene modifier icon

2 Double click the Change Scene modifier. Change its name to `Go to Chat Scene`. Use the **Execute When** pop-up menu to choose **Author Messages ➛ Go to Chat Scene.** In the **Specifications** section of the dialog box, click the **Specify Scene** button and use the **Scene** pop-up menu to choose **Chat Scene.** Your **Change Scene Modifier** dialog box should look like the one shown below. Click **OK** to dismiss the dialog box.



Add the IP Address Entry Field

Now we'll add the components that let the end user enter his/her friend's IP address.

1 Select the Text tool from the **Tool** palette. The cursor changes to an I-beam cursor. Drag on the scene to create a wide text element near the top of the scene.

*Selecting the Text tool in the **Tool** palette*

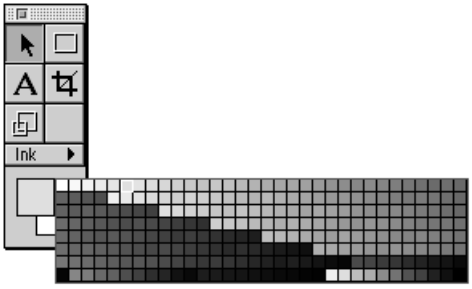
2 When you release the mouse button, an insertion point appears in the new text element. Type the following text into the element:

```
Enter your friend's IP address:
```

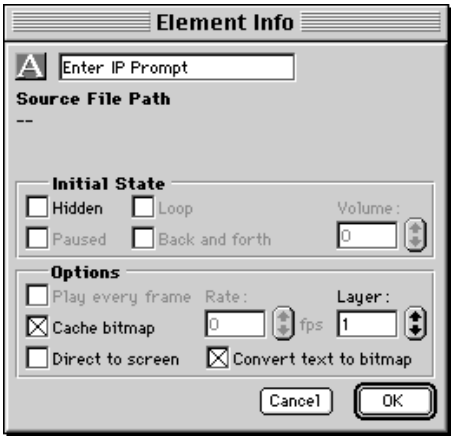
3 Click outside the text element to end text entry. The cursor becomes a pointer again. If your text element is not big enough to display the full text of the prompt, simply resize it by dragging its borders with the Selection tool.

4 Click the text element to ensure that it is selected. Use the **Ink** pop-up menu on the **Tool** palette to choose **Background Transparent.** A Graphic modifier appears on the text element.

5 Click and drag the **Foreground Color Swatch** in the **Tool** palette to display a palette of foreground colors as shown below. Choose a light foreground color that appeals to you (we used a bright yellow in the original version of the NetChat title because it shows up well against the black background).



6 The text changes color. Double-click the text element to display its **Element Info** dialog box. Change the element's name to `Enter IP Prompt`. Also click the **Convert Text to Bitmap** check box. We'll be changing the type style in this text element and want to be sure it looks the same on all platforms. Your **Element Info** dialog box should look like the one shown below. Click **OK** to dismiss the dialog box.

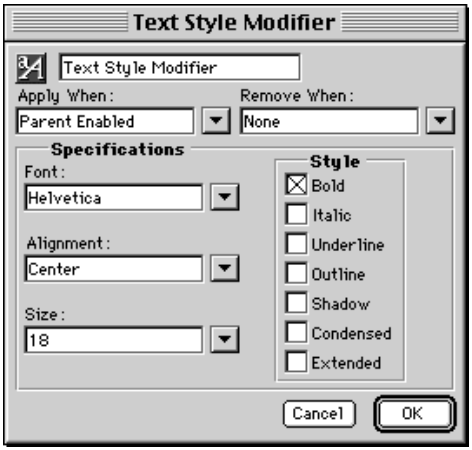


7 Drag a Text Style modifier from the **Effects** palette and drop it on the Enter IP Prompt text element.

The Text Style modifier icon

8 Previously, we used the **Format** menu to format text. In this case, we'll use the Text Style modifier. Differences between these two ways of formatting text are described in Chapter 3 of the *mTropolis Reference Guide*, "Format Menu."

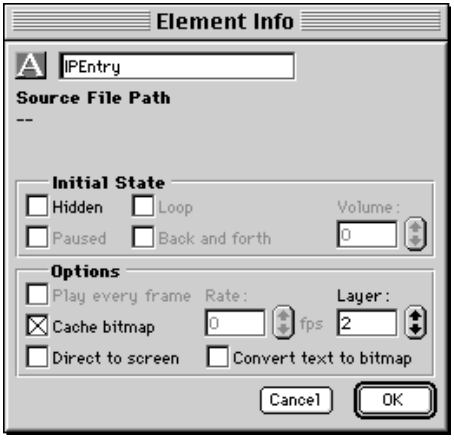
Double-click the Text Style modifier icon. Choose a font, style, and size that appeals to you. In the original NetChat title, we used Helvetica, center aligned, 18 points. Your dialog box should look similar to the one shown below. Click **OK** to dismiss the dialog box and apply the text style formatting.



9 Select the Text tool from the **Tool** palette once again. The cursor changes to an I-beam cursor. Drag on the scene to create a wide text element just below the Enter IP Prompt text element. This is where the end user will enter his/her friend's IP address.

10 When you release the mouse button, an insertion point appears in the new text element. We want this text element to be empty by default, so just click outside the text element to return to the Selection tool.

11 Double-click the new text element. Change its name to `IPEntry`. Its **Element Info** dialog box should look like the one shown below. Click **OK** to dismiss the **Element Info** dialog box.



12 Drag a Miniscript modifier from the **Logic** palette and drop it in the IPEntry text element.

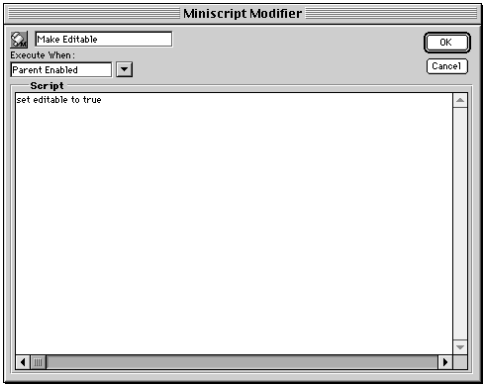
The Miniscript modifier icon

13 Double-click the Miniscript modifier icon to display its configuration dialog box. Change its name to `Make Editable`. Use the **Execute When** pop-up menu to choose **Parent ➛ Parent Enabled.** Enter the following statement in the **Script** field:

```
set editable to true
```

When executed during run-time mode, this statement makes the text element editable by the end user. Clicking inside the element puts an insertion point in the element. Clicking outside the element ends text entry (and generates an Edit Done message).

Your **Miniscript Modifier** dialog box should look like the one shown below. Click **OK** to dismiss the dialog box.



14 Drag another Miniscript modifier from the **Logic** palette and drop it into the IPEntry text element.

The Miniscript modifier icon

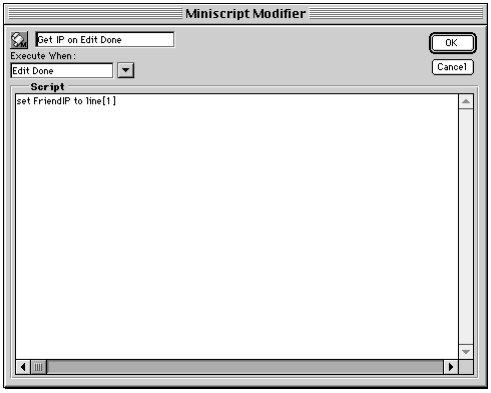
15 Double-click the Miniscript modifier icon to display its configuration dialog box. Change its name to `Get IP on Edit Done`. Use the **Execute When** pop-up menu to choose **Text ➛ Edit Done.** Enter the following statement in the Script field:

```
set FriendIP to line[1]
```

Note that the character inside the square brackets is the number one (1) not a lowercase letter "L". This modifier listens for the end user to finish entering text in the element. It then reads the first line

of text from the element (using the "line" attribute) and stores it in the global variable FriendIP.

Your **Miniscript Modifier** dialog box should look like the one shown below. Click **OK** to dismiss the dialog box.



Add the Connect Button

Now let's add a text element that can be used as the Connect button. When the end user clicks this element, mTropolis will attempt to connect to the remote computer specified by the end user in the IPEntry text element.

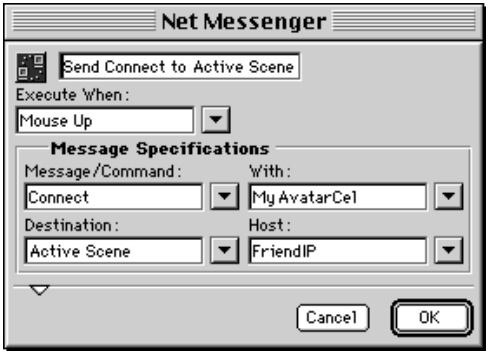
Since this button will be similar to the Continue button we created in the Select Avatar Scene, let's copy the **Continue Button** as a starting point.

1. Use the Previous Scene arrow at the top of the Layout window to navigate back to the Select Avatar Scene. That scene is displayed in the Layout window.
2. Click the Continue button element to select it, then press ⌘-C to copy it to the clipboard.
3. Use the Next Scene arrow at the top of the Layout window to return to the Connection Scene.
4. Press ⌘-V to paste the Continue Button in the Connection Scene.
5. Delete the two Behavior modifiers found on the copy of the Continue Button.
6. Double-click the Continue Button to display its **Element Info** dialog box. Change its name to `Connect Button`.
7. Select the Text tool from the **Tool** palette. Click inside the Connect Button element and highlight the text, which currently reads "Continue!". Change the text to `Connect!`. Click outside the element to end text editing.
8. Drag a Net Messenger from the **Network** palette and drop it on the Connect Button element.

The Net Messenger icon

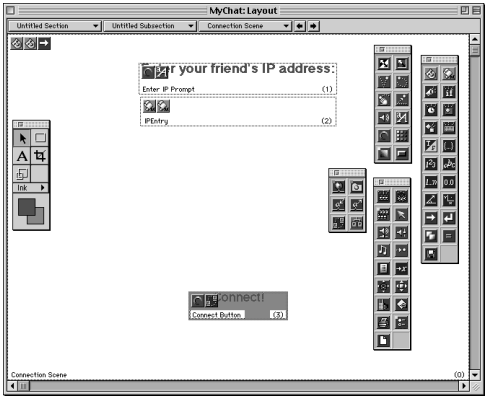
9. Double-click the Net Messenger icon. Change the modifier's name to `Send Connect to Active Scene`. Don't change the default **Execute When** message of **Mouse Up.** Use the **Message/Command** pop-up menu to choose **Author Messages ➛ Connect.** Use the **With** pop-up menu to choose the **My AvatarCel** variable. Use the **Destination** pop-up menu to choose **Active Scene.** Use the **Host** pop-up menu to choose FriendIP.

Your **Net Messenger** dialog box should look like the one shown below. Click **OK** to dismiss the dialog box.



This modifier initiates the connection process whenever the Connect Button is clicked.

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



Add Basic Error Checking

There are many things that can go wrong when trying to establish a network connection. For example, a network connection may not be available, the address of the remote computer may have been entered incorrectly, etc. To detect some of these problems, we'll add some basic error checking and an informational prompt to the Connection Scene.

1 Select the Text tool from the **Tool** palette. The cursor changes to an I-beam cursor. Drag on the scene to create a wide text element above the **Connect Button.**

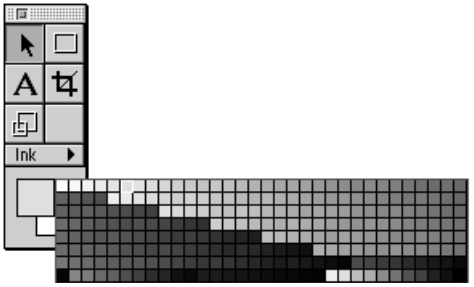
2 When you release the mouse button, an insertion point appears in the new text element. Type the following text into the element:

```
Connection timed out: Your friend
is not yet running Avatar Chat or
there is another network problem.
Check the IP address, or wait and
try your connection again...
```

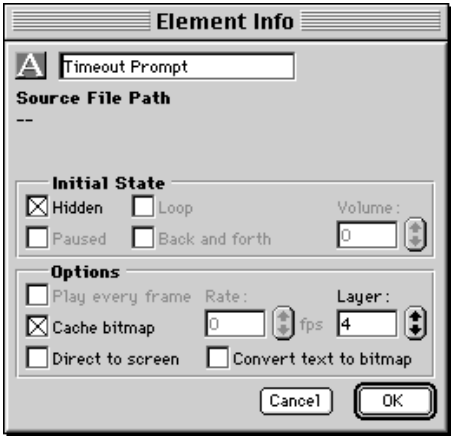
3 Click outside the text element to end text entry. The cursor becomes a pointer again. If your text element is not big enough to display the full text of the prompt, simply resize it by dragging its borders with the Selection tool.

4 Click the text element to ensure that it is selected. Use the **Ink** pop-up menu on the **Tool** palette to choose **Background Transparent.** A Graphic modifier appears on the text element.

5 Click and drag the **Foreground Color Swatch** in the **Tool** palette to display a palette of foreground colors as shown below. Choose a light foreground color that appeals to you (we used a bright yellow in the original version of the NetChat title because it shows up well against the black background).



6 The text changes color. Double-click the text element to display its **Element Info** dialog box. Change the element's name to `Timeout Prompt`. Also click the **Hidden** check box. We don't want this element to be visible by default. Your **Element Info** dialog box should look like the one shown below.



7 Drag a Messenger modifier from the **Logic** palette and drop it on the Connect Button element. We'll configure this messenger to show the Timeout Prompt when it detects a network timeout error.

The Messenger modifier icon

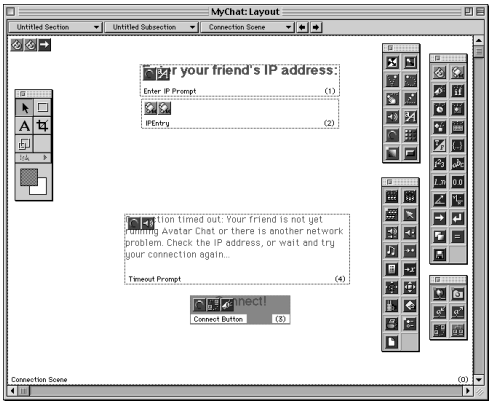
8 Double-click the Messenger modifier icon. Change its name to `Timeout Messenger`. Use the **Execute When** pop-up menu to choose **Modifier Messages ➛ Net Messaging ➛ Connection Timed Out.** Use the **Message/Command** pop-up menu to choose **Element ➛ Show**. Use the **Destination** pop-up menu to choose **Element's Siblings ➛ Timeout Prompt.** Your **Messenger** dialog box should look like the one shown below. Click **OK** to dismiss the dialog box.

9 Drag a Sound Effect modifier from the **Effects** palette and drop it on the Timeout Prompt element. We'll configure this Sound Effect to play a system beep when the Timeout Prompt is shown.

The Sound Effect modifier icon

10 Double-click the Sound Effect modifier icon. Change its name to `Beep when Shown`. Use the **Execute When** pop-up menu to choose **Element ➛ Shown**. The default system beep sound is fine, so you don't need to change the **Sound** pop-up menu. Your **Sound Effect Modifier** dialog box should look like the one shown below. Click **OK** to dismiss the dialog box.

11 Your Connection Scene should now look like the one shown below.



Add the Please Wait Prompt

Recall that there is the possibility that the remote end user may still be choosing his/her avatar when a connection is attempted. In that case, the remote project responds to the "Send Avatar" message with a message called "Please Wait." If this message is received by the Connection Scene, we need to instruct the end user to wait for his/her friend to select an avatar. We also want to disable the IPEntry text field and the Connect Button. These elements can be disabled by hiding them when the "Please Wait" message is received.

1 Select the Text tool from the **Tool** palette. The cursor changes to an I-beam cursor. Drag on the Connection Scene to create a wide text element between the Timeout Prompt and the IPEntry text elements.

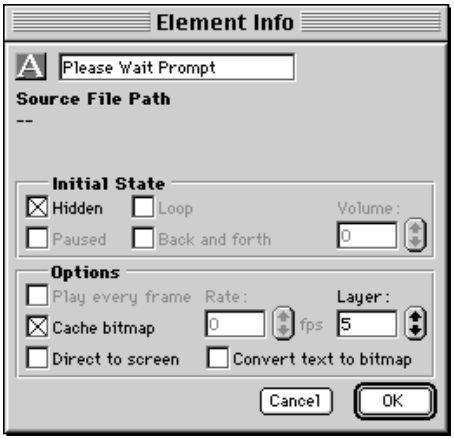
2 When you release the mouse button, an insertion point appears in the new text element. Type the following text into the element:

```
Your friend is still selecting an
avatar, please wait...
```

3 Click outside the text element to end text entry. The cursor becomes a pointer again. If your text element is not big enough to display the full text of the prompt, simply resize it by dragging its borders with the Selection tool.

4 Copy the Graphic modifier from the previously-created Timeout Prompt by holding the Option key while dragging the Graphic modifier from the Timeout Prompt and dropping it on the new text element. The element inherits the same color properties as the Timeout Prompt.

5 Double-click the new text element to display its **Element Info** dialog box. Change its name to `Please Wait Prompt`. Also click its **Hidden** check box to make the element invisible until explicitly told to show during run-time. Your **Element Info** dialog box should look like the one shown below. Click **OK** to dismiss the dialog box.



6 Drag a Messenger modifier from the **Logic** palette and drop it on the Please Wait Prompt text element. We'll configure this messenger to show the Please Wait Prompt when it receives the Please Wait message.

The Messenger modifier icon

7 Double-click the Messenger modifier icon. Change its name to `Show on Wait`. Use the **Execute When** pop-up menu to choose **Author Messages ➛ Please Wait.** Use the **Message/Command** pop-up menu to choose **Element ➛ Show.** Your **Messenger** dialog box should look like the one shown below. Click **OK** to dismiss the dialog box.

8 We need to add a similar Messenger to all the other elements on the Connection Scene. This Messenger, however should be configured to hide the elements when the Please Wait message is received.

Drag a Messenger modifier from the **Logic** palette and drop it on the **Enter IP Prompt** text element. We'll configure this messenger to hide the element when it receives the Please Wait message.

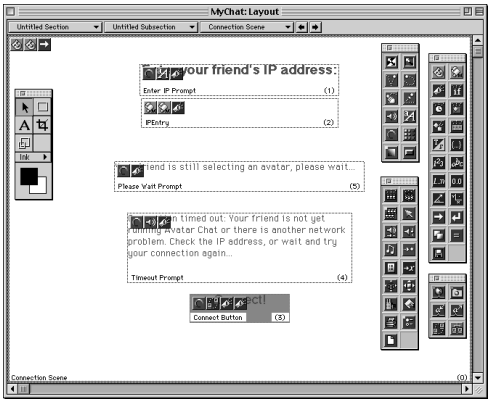
The Messenger modifier icon

9 Double-click the Messenger modifier icon. Change its name to `Hide on Wait`. Use the **Execute When** pop-up menu to choose **Author Messages ➛ Please Wait.** Use the **Message/Command** pop-up menu to choose **Element ➛ Hide.** Your **Messenger** dialog box should look like the one shown below. Click **OK** to dismiss the dialog box.

10 We need to copy this new Messenger modifier to all the other elements in the scene except for the Please Wait Prompt. Hold down the Option key and drag the Hide on Wait Messenger from the Enter IP Prompt element and drop it on the IPEntry text element.

Repeat this step to copy the Hide on Wait messenger to the Timeout Prompt and Connect Button elements.

11 Your Connection Scene is now complete. It should look like the one shown below.



Complete the Chat Scene

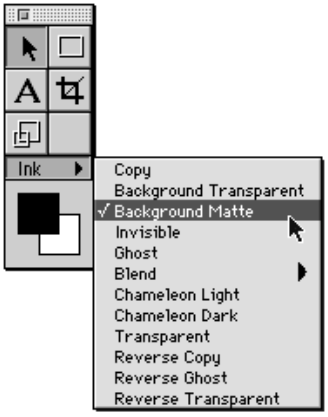
Now we'll complete the final scene in the project. The Chat Scene is the one in which the two end users' avatars appear. End users can move their avatars around the scene and enter chat text. We'll start by creating the avatars and their chat bubbles (the text entry fields used to enter chat text).

1 Use the Next Scene arrow at the top of the Layout window to navigate to the Chat Scene. The Chat Scene is displayed in the Layout window. You should see the Party Room picture that we previously linked to this scene.

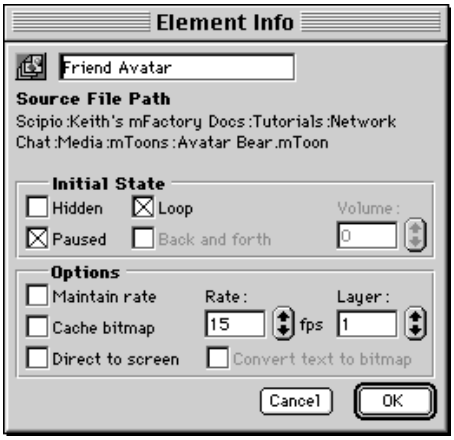
2 Display the **Asset Palette** by choosing **Asset Palette** from the **View** menu. This palette shows thumbnails of all the media files that we have linked to the project.

3 Drag the thumbnail of the **Avatar Bear.mToon** asset thumbnail from the **Asset Palette** and drop it somewhere on the right side of the scene. A new Avatar Bear.mToon element appears. It is also selected.

4 Use the **Ink** pop-up menu on the **Tool** palette to choose **Background Matte** as shown below. A Graphic modifier appears on the Avatar Bear.mToon element and its background becomes transparent.

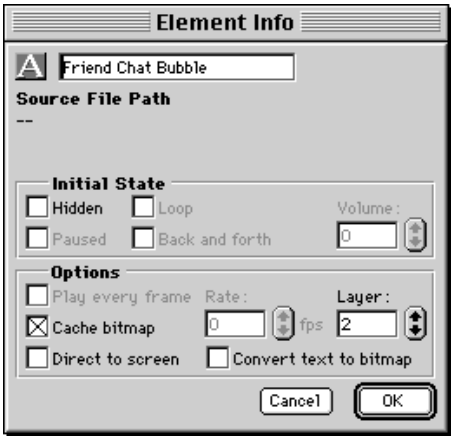


5 Double-click the Avatar Bear.mToon element to display its **Element Info** dialog box. Change its name to `Friend Avatar`. Also click its **Paused** check box. Even though this is an mToon, we don't want it to animate, we just want to display one of its cels (the cel chosen by the end user of the remote project). Your **Element Info** dialog box should look like the one shown below. Click **OK** to dismiss the dialog box.



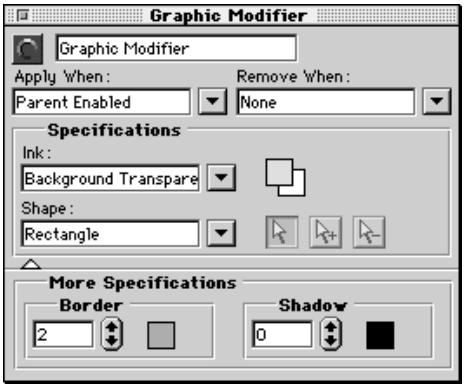
6 Select the Text tool from the **Tool** palette. The cursor changes to an I-beam cursor. Drag on the Chat Scene to create a wide text element centered just above the Friend Avatar element. We want this text element to be empty by default, so simply click outside the text element to end text entry.

7 Double-click the new text element to display its **Element Info** dialog box. Change its name to `Friend Chat Bubble.` Your **Element Info** dialog box should look like the one shown below. Click **OK** to dismiss the dialog box.

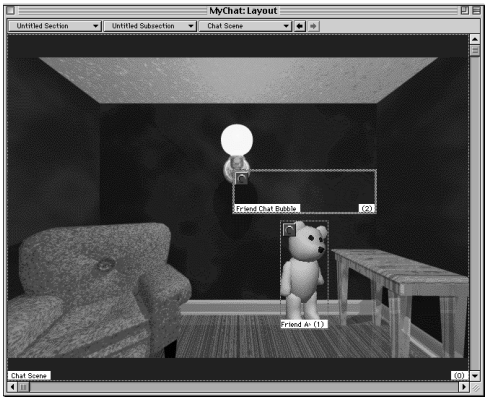


8 Drag a Graphic modifier from the **Effects** palette and drop it on the Friend Chat Bubble text element.

9 Double-click the Graphic modifier. Use the **Ink** pop-up menu to choose **Background Transparent.** Use the Foreground Color swatch to choose a light color to be used for the text. Click the open/close triangle at the bottom of the dialog box to reveal the **More Specifications** section. Enter 2 in the **Border** field and choose a light color from the color swatch next to the border field. This will give the text element a visible border. Your **Graphic Modifier** dialog box should look similar to the one shown below. Click the dialog box's **Close** box to dismiss the dialog box.



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



10 Click the Friend Chat Bubble element to ensure that it is selected. Select the Parent/Child tool from the **Tool** palette as shown below.

Selecting the Parent/Child tool

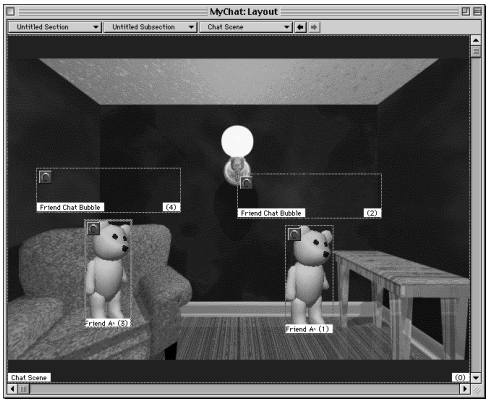
11 Click inside the Friend Chat Bubble and drag the cursor down over the Friend Avatar element. A line appears connecting the Friend Chat Bubble to the cursor. Release the mouse button when the cursor is over the Friend Avatar element.

The Friend Chat Bubble element has been made a child of the Friend Avatar element.

Now when the Friend Avatar element is moved, the Friend Chat Bubble moves with it.

12 Click the Friend Avatar element to select it. Press ⌘-D to duplicate this element. A copy of the bear mToon appears. Note that the chat bubble is also duplicated, because it is a child of the element we duplicated.

13 Drag this copy of the element (the copy is currently selected and appears in front of the original) to the left side of the screen. Your Layout window should now look similar to the one shown below.



14 Double-click the copy of the Friend Avatar element (that is, the bear on the left side of the screen) to display its **Element Info** dialog box. Change its name to `My Avatar`. Click **OK** to dismiss the **Element Info** dialog box.

15 Double-click the copy of the Friend Chat Bubble text element (the chat bubble on the left side of the screen) to display its **Element Info** dialog box. Change its name to `My Chat Bubble`. Click **OK** to dismiss the **Element Info** dialog box.

16 Now would be good time to save your work!

Program the Chat Bubbles

Now that we have created the two avatars for the chat scene, we can program them to send and receive text messages. The My Chat Bubble text element represents the chat bubble that the end user can edit to enter chat text they want to *send* to his/her friend. The Friend Chat Bubble element is used to display text *received from* his/her friend.

Program My Chat Bubble

We'll start with the My Chat Bubble text element (the one on the left side of the screen).

1 Drag a Miniscript modifier from the **Logic** palette and drop it in the My Chat Bubble text element.

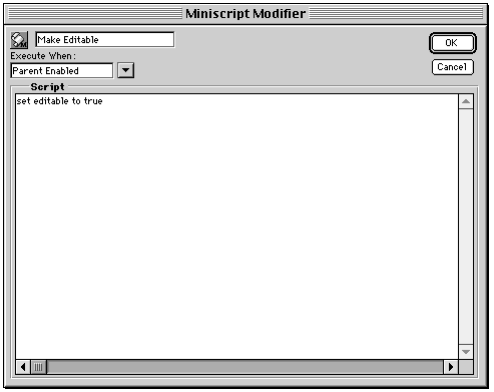
The Miniscript modifier icon

2 Double-click the Miniscript modifier icon to display its configuration dialog box. Change its name to `Make Editable`. Use the **Execute When** pop-up menu to choose **Parent ➛ Parent Enabled.** Enter the following statement in the **Script** field:

```
set editable to true
```

When executed during run-time mode, this statement makes the text element editable by the end user. Clicking inside the element puts an insertion point in the element. Clicking outside the element ends text entry (and generates an Edit Done message).

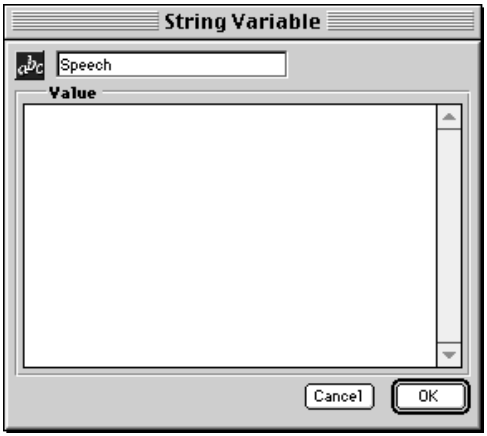
Your **Miniscript Modifier** dialog box should look like the one shown below. Click **OK** to dismiss the dialog box.



3 Drag a String Variable from the **Logic** palette and drop it on My Chat Bubble.

The String Variable icon

4 Double-click the String Variable icon. Change the Variable's name to `Speech`. Leave the value field empty. Your **String Variable** dialog box should look like the one shown below. Click **OK** to dismiss the dialog box.



5 Drag a Messenger modifier from the **Logic** palette and drop it on My Chat Bubble.

The Messenger icon

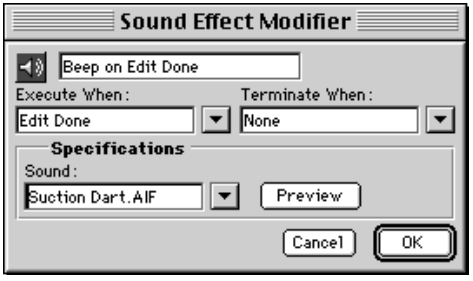
6 Double-click the Messenger modifier icon. Change its name to `Get Speech`. Use the **Execute When** pop-up menu to choose **Text ➛ Edit Done.** Use the **Message/ Command** pop-up menu to choose **Get Attribute ➛ Get Text.** Use the **With** pop-up menu to choose **My Chat Bubble ➛ Speech.** Your **Messenger** dialog box should look like the one shown below. Click **OK** to dismiss the dialog box.

This modifier listens for when the end user clicks outside the My Chat Bubble element, which causes the Edit Done message to be generated. In response, it takes the current contents of the text element and stores them in the Speech String Variable.

7 Drag a Sound Effect modifier from the **Effects** palette and drop it on My Chat Bubble.

The Sound Effect modifier icon

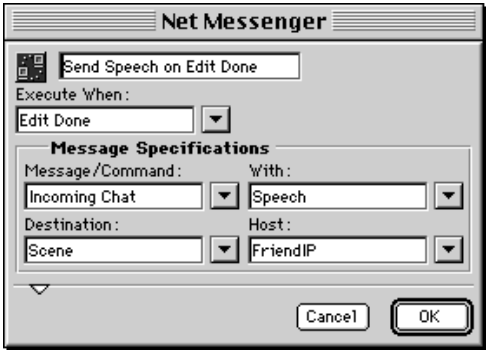
8 Double-click the Sound Effect modifier. Change its name to `Beep on Edit Done`. Use the **Execute When** pop-up menu to choose **Text ➛ Edit Done.** Use the **Sound** pop-up menu to choose **Suction Dart.AIF.** Your **Sound Effect Modifier** dialog box should look like the one shown below. Click **OK** to dismiss the dialog box.



9 Drag a Net Messenger from the **Network** palette and drop it on My Chat Bubble.

The Net Messenger icon

10 Double-click the Net Messenger icon. Change the Messenger's name to `Send Speech on Edit Done`. Use the **Execute When** pop-up to select **Text ➛ Edit Done.** Use the **Message/Command** pop-up menu to choose **Author Messages ➛ New Author Message.** The **New Author Message** dialog box appears. Use it to create a message called `Incoming Chat` and click **OK** to dismiss the dialog box. Use the **With** pop-up menu to choose **My Chat Bubble ➛ Speech.** Use the **Destination** pop-up menu to choose **Scene.** Use the **Host** pop-up menu to choose **FriendIP.** Your **Net Messenger** dialog box should look like the one shown below.



This messenger listens for Edit Done, then sends the Incoming Chat message to the remote project with the contents of the Speech Variable as accompanying data.

11 The My Chat Bubble programming is now complete.

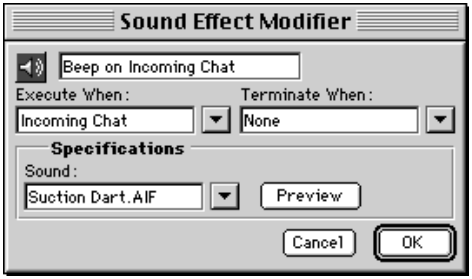
Program Friend Chat Bubble

Now let's turn our attention to the other chat bubble (named Friend Chat Bubble) found on the right side of the scene.

1 Drag a Sound Effect modifier from the **Effects** palette and drop it on the Friend Chat Bubble text element.

The Sound Effect modifier icon

2 Double-click the Sound Effect modifier. Change its name to `Beep on Incoming Chat`. Use the **Execute When** pop-up menu to choose **Author Messages ➛ Incoming Chat.** Use the **Sound** pop-up menu to choose **Suction Dart.AIF.** Your **Sound Effect Modifier** dialog box should look like the one shown below. Click **OK** to dismiss the dialog box.



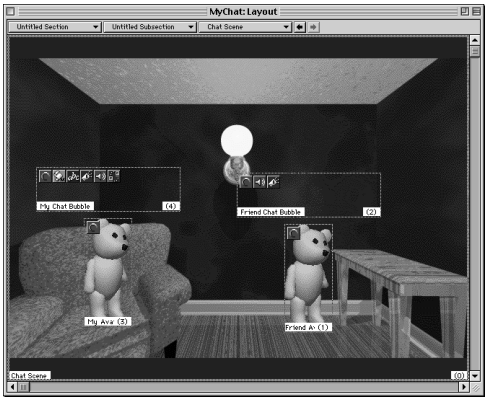
3 Drag a Messenger modifier from the **Logic** palette and drop it on Friend Chat Bubble.

The Messenger icon

4 Double-click the Messenger modifier icon. Change its name to `Set Text to Incoming Chat`. Use the **Execute When** pop-up menu to choose **Author Messages ➛ Incoming Chat.** Use the **Message/Command** pop-up menu to choose **Set Attribute ➛ Set text.** Use the **With** pop-up menu to choose **Incoming Data.** Your **Messenger** dialog box should look like the one shown below. Click **OK** to dismiss the dialog box.

This modifier listens for the Incoming Chat message. When that message is received, it sets the text contained in the element to the text that arrives with the Incoming Chat message (the contents of the Speech Variable sent by the remote project).

5 The programming for the Friend Chat Bubble element is now complete. Your Chat Scene should look like the one shown below.



Program My Avatar

Now let's program the My Avatar element (the bear on the left side of the screen).

1 Drag a Messenger modifier from the **Logic** palette and drop it on My Avatar.

The Messenger icon

2 Double-click the Messenger modifier icon. Change its name to `Set Avatar Cel`. Use the **Execute When** pop-up menu to choose **Scene ➛ Scene Started.** Use the **Message/Command** pop-up menu to select **Set Attribute ➛ Set cel.** Use the **With** pop-up menu to choose **MyAvatarCel.** Your **Messenger** dialog box should look like the one shown below. Click **OK** to dismiss the dialog box.

This messenger sets the cel of the Avatar Bear mToon to the same cel selected by the end user in the Select Avatar scene.

3 Drag a Miniscript modifier from the **Logic** palette and drop it in the My Avatar element.

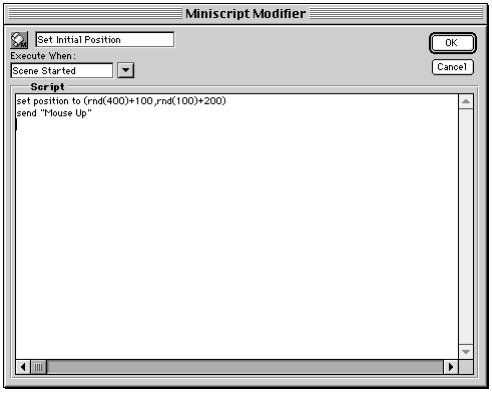
The Miniscript modifier icon

4 Double-click the Miniscript modifier icon to display its configuration dialog box. Change its name to `Set Initial Position`. Use the **Execute When** pop-up menu to choose **Scene ➛ Scene Started.** Enter the following statements in the Script field:

```
set position to \
(rnd(400)+100,rnd(100)+200)
send "Mouse Up"
```

When executed during run-time mode, this statement sets the position of the element to a random location within the scene. The modifier then sends the Mouse Up message to the element. This message will trigger other modifiers (that we'll add in the following steps) on the element and cause the element to move smoothly to the random location.

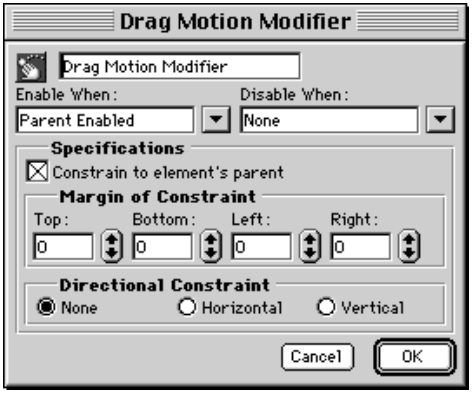
Your **Miniscript Modifier** dialog box should look like the one shown below. Click **OK** to dismiss the dialog box.



5 Drag a Drag Motion modifier from the **Effects** palette and drop it on My Avatar.

The Drag Motion modifier icon

6 Double-click the Drag Motion modifier. Click the **Constrain to element's parent** check box. Your **Drag Motion Modifier** dialog box should look like the one shown below. Click **OK** to dismiss the dialog box.

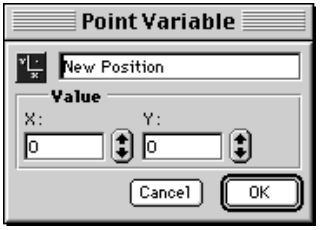


This modifier allows the avatar to be dragged around (but not outside of) the scene by the end user.

7 Drag a Point Variable from the **Logic** palette and drop it on My Avatar.

The Point Variable icon

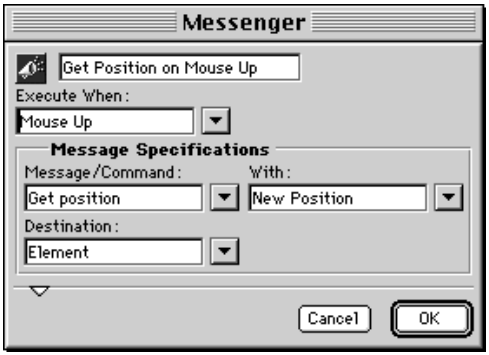
8 Double-click the Point Variable. Change its name to `New Position`. Your **Point Variable** dialog box should look like the one shown below. Click *OK* to dismiss the dialog box.



9 Drag a Messenger modifier from the **Logic** palette and drop it on My Avatar.

The Messenger icon

10 Double-click the Messenger modifier icon. Change its name to `Get Position on Mouse Up`. Use the **Message/Command** pop-up menu to choose **Get Attribute ➛ Get position.** Use the **With** pop-up menu to choose **My Avatar ➛ New Position.** Your **Messenger** dialog box should look like the one shown below. Click **OK** to dismiss the dialog box.



This Messenger listens for the Mouse Up message. Mouse Up will be generated whenever the end user drags his/her avatar and then releases the mouse. In response, this Messenger reads the current position of the My Avatar element and stores it in the variable New Position. Later, we'll create a Net Messenger that sends this information to the remote project.

11 Click the Get Position on Mouse Up messenger (the one we just created) to select it. Press ⌘-D to duplicate this Messenger. A copy of the Messenger appears to the right of the original.

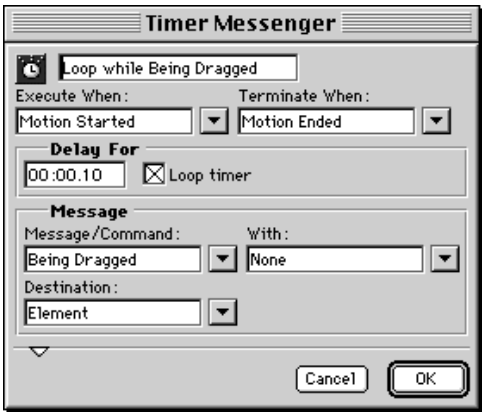
12 Double-click the new Messenger. Change its name to `Get Position when Dragged`. Use the **Execute When** pop-up menu to choose **Author Messages ➛ New Author Message.** In the **New Author Message** dialog box that appears, enter `Being Dragged`. Click **OK** to dismiss the **New Author Message** dialog box. Your **Messenger Modifier** dialog box should look like the one shown below. Click **OK** to dismiss the dialog box.

This messenger reads the current position of the My Avatar element and stores it in the variable New Position whenever the Being Dragged message is received. Later, we'll create a Timer Messenger that sends the Being Dragged message repeatedly whenever the end user is dragging the avatar. We'll also create a Net Messenger that sends this updated position information to the remote project.

13 Drag a Timer Messenger from the **Logic** palette and drop it on My Avatar.

The Timer Messenger icon

14 Double-click the Timer Messenger. Change its name to `Loop while Being Dragged`. Use the **Execute When** pop-up menu to choose **Motion ➛ Motion Started.** Use the **Terminate When** pop-up menu to choose **Motion ➛ Motion Ended.** Enter `00:00.10` in the Delay For field. Click the **Loop Timer** check box. Use the **Message/Command** pop-up menu to choose **Author Messages ➛ Being Dragged.** Your **Timer Messenger** dialog box should look like the one shown below. Click **OK** to dismiss the dialog box.

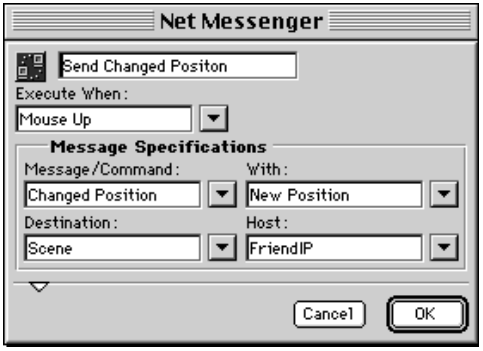


This messenger detects when the avatar is being dragged by the end user (when it starts and stops moving). While that motion is happening, it sends the Being Dragged message ten times per second. The Being Dragged message causes the element to update its New Position variable and send that information to the remote project. This information will be used to simulate smooth drag motion of the Friend Avatar element.

15 Drag a Net Messenger from the **Network** palette and drop it on My Avatar.

The Net Messenger icon

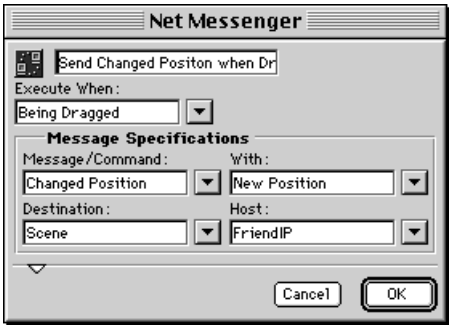
16 Double-click the Net Messenger icon. Change the Messenger's name to `Send Changed Position`. Use the **Message/Command** pop-up menu to choose **Author Messages ➛ New Author Message.** The **New Author Message** dialog box appears. Use it to create a message called `Changed Position` and click **OK** to dismiss the dialog box. Use the **With** pop-up menu to choose **My Avatar ➛ New Position.** Use the **Destination** pop-up menu to choose **Scene.** Use the **Host** pop-up menu to choose **FriendIP.** Your **Net Messenger** dialog box should look like the one shown below.



This Messenger sends the avatar's new position to the remote project whenever the end user releases the mouse (when the end user has dragged the avatar to a new position).

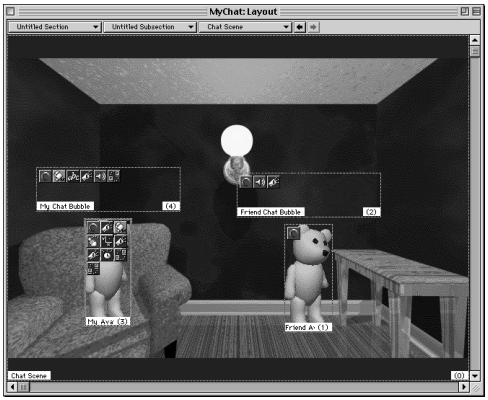
17 Click the **Send Changed Position** Net Messenger (the one we just created) to select it. Press ⌘-D to duplicate this messenger. A copy of the Net Messenger appears to the right of the original.

18 Double-click the copy of the Net Messenger. Change its name to `Send Changed Position when Being Dragged`. Use the **Execute When** pop-up menu to choose **Author Messages ➛ Being Dragged.** Your **Net Messenger** dialog box should look like the one shown below. Click **OK** to dismiss the dialog box.



This Messenger sends the avatar's new position to the remote project whenever the element receives the Being Dragged message. Recall that this message is generated 10 times per second by the Timer Messenger whenever the avatar is being dragged by the end user.

19 The My Avatar element is now fully programmed. Your Chat Scene should look like the one shown below.



Program Friend Avatar

The final steps in this tutorial involve programming the behavior of the Friend Avatar element (the bear on the right side of the scene). This element is not draggable by the end user. Instead, it represents the avatar selected by the end user of the remote project. It moves whenever the remote end user moves his/her avatar.

1 Drag a Messenger modifier from the **Logic** palette and drop it on the Friend Avatar element.

The Messenger icon

2 Double-click the Messenger modifier icon. Change its name to `Set Avatar Cel`. Use the **Execute When** pop-up menu to choose **Scene ➛ Scene Started.** Use the **Message/ Command** pop-up menu to choose **Set Attribute ➛ Set cel.** Use the **With** pop-up menu to choose **Friend AvatarCel** Your **Messenger** dialog box should look like the one shown below. Click **OK** to dismiss the dialog box.

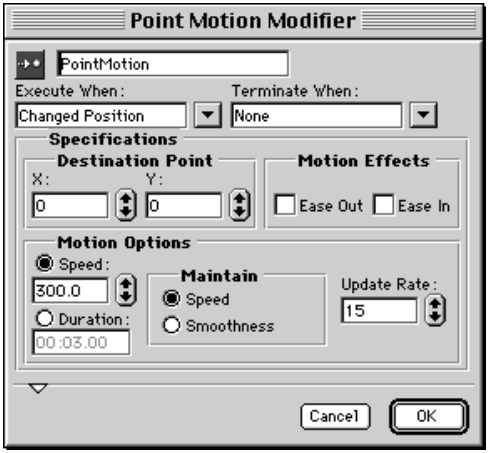
This Messenger sets the cel of the Avatar Bear mToon to the cel selected by the end user of the remote chat project.

3 If the **Extras** palette is not already visible, display it by choosing **Modifier Palettes ➛ Extras** from the **View** menu.

4 Drag a Point Motion modifier from the **Extras** palette and drop it on Friend Avatar.

The Point Motion modifier icon

5 Double-click the Point Motion modifier icon. Change its name to `PointMotion` (note that there are no spaces in that name). Use the **Execute When** pop-up menu to choose **Author Messages ➛ Changed Position.** In the **Motion Options** section of the dialog box, enter `300` in the Speed field. Enter `15` in the Update Rate field. Your **Point Motion Modifier** dialog box should look like the one shown below. Click **OK** to dismiss the dialog box.



The Point Motion modifier moves an element in a straight line to a new position. Note that we left the X and Y destination values at zero. We'll set the destination values using a Miniscript modifier.

6 Drag a Miniscript modifier from the **Logic** palette and drop it into the Friend Avatar element.

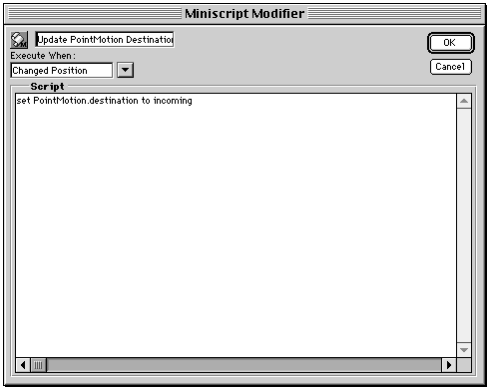
The Miniscript modifier icon

7 Double-click the Miniscript modifier icon to display its configuration dialog box. Change its name to `Update Point-Motion Destination`. Use the **Execute When** pop-up menu to choose **Author Messages ➛ Changed Position.** Enter the following statement in the Script field:

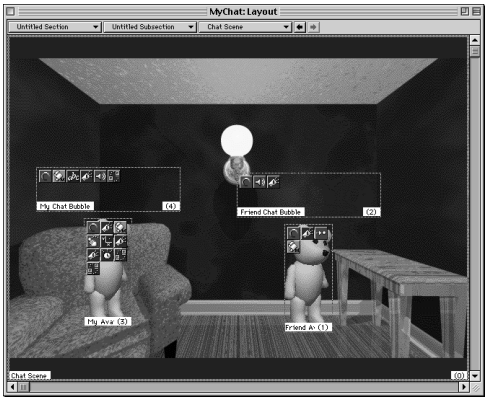
```
set PointMotion.destination to
incoming
```

When the Changed Position message is received, this statement changes the destination value of the Point Motion modifier (named PointMotion) to the value of the incoming data. Recall that the Changed Position message is sent with the New Position point value. As a result, the Friend Avatar element will move smoothly to its new position whenever the Changed Position message is received.

Your **Miniscript Modifier** dialog box should look like the one shown below. Click **OK** to dismiss the dialog box.



8 The Friend Avatar element programming is complete. Your Chat Scene should now look like the one shown below.



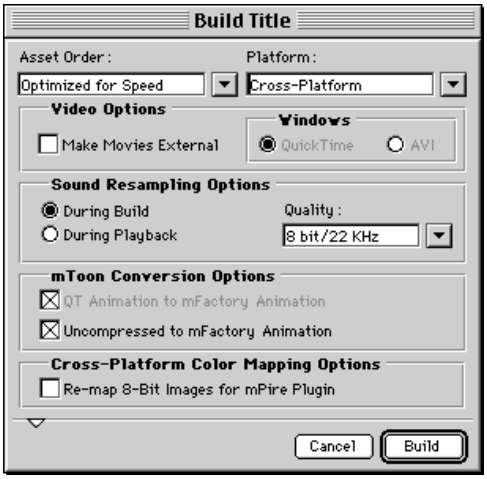
Your avatar chat project is now complete! Save your work by pressing ⌘-S.

Build and Test the Finished Project

Your finished avatar chat project can be tested in the same way that we test examined the **NetChat.mfx** title (described at the beginning of this chapter). First, however, we need to turn our completed project into a mTropolis title file.

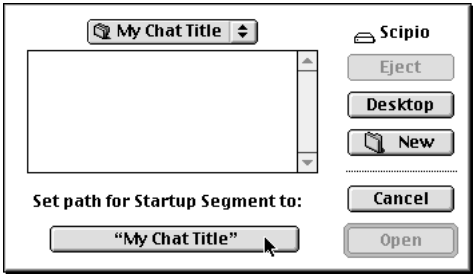
1 If you haven't already saved your finished project, save it now by pressing ⌘-S.

2 Choose **Build Title** from the **File** menu. The **Build Title** dialog box appears as shown below.



3 The default options are fine for building this project. The title file created will be a cross-platform file, so it can run on both Mac OS and Windows machines (with the proper version of the mTropolis player).

4 Click the **Build** button to start the building process. A standard folder selection dialog box appears. Choose the folder in which you want your built title file to be saved.



5 A number of alerts appear. These alerts show the progress of the Build Title process.

6 When the Build Title process is finished, your built title file can be found in the folder you specified previously. Its name will be the name of the project file, appended with ".mfx". For example, if you saved your project as **MyChat**, the built title file will be named **MyChat.mfx.**

Your built title file can be tested just like we tested the the NetChat.mfx title.

1 We'll assume that one of the computers is the one on which you've installed mTropolis. The second computer must also have the mTropolis player installed. Use the mTropolis CD–ROM to install the player on the second machine. Because the avatar chat title is a cross-platform title, it doesn't matter if the second computer is a Mac OS or Windows 95/NT machine. Players can also be copied directly from the mTropolis CD–ROM's **mTropolis Players** folder. Make sure to copy both the player application and its associated **mPlugIns** folder as described in the **Deploying mTropolis Players** file (also found in the **mTropolis Players** folder).

2 Copy your built title file to the second computer.

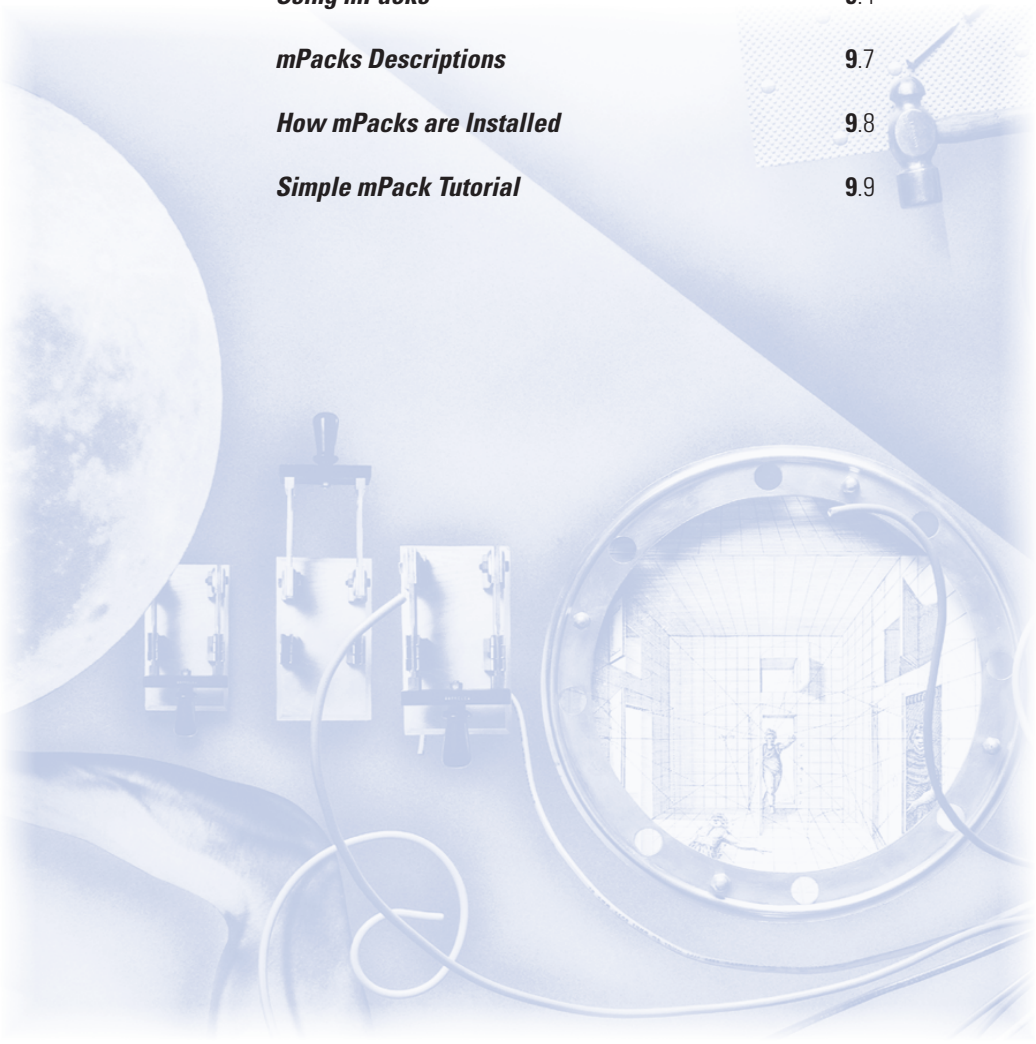
3 For both machines, start the avatar chat title by dragging the the icon of your built title file on the **mTropolis Player** or **mtplay32.exe** icon (the icon that represents the mTropolis player application). Alternatively, if you installed the file in the same folder as the mTropolis player application, you can simply double-click the player icon and the title will automatically play.

4 Your avatar chat title should start playing. It should behave similarly to the NetChat.mfx title supplied by Quark (described at the start of this tutorial).

5 Have fun playing with the avatar chat title!

6 If your title does not work properly, you might want to open the **Completed Net Chat** project file (found in the **Network Chat** folder) and compare it to your own avatar chat project file.

9. mPacks



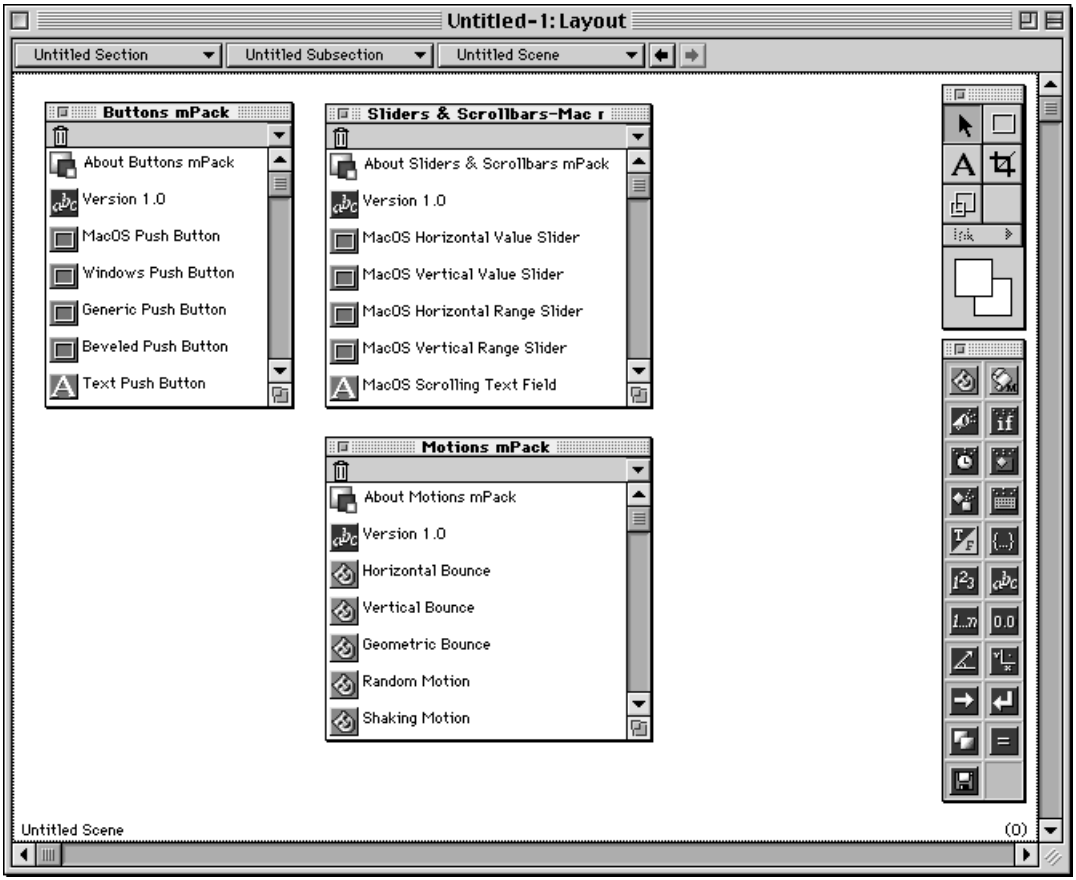
9. mPacks

This chapter describes the “mPacks” included with mTropolis. mPacks are sets of libraries that are installed with mTropolis. These libraries contain useful pre-fabricated elements and behaviors that can be easily reused in your own mTropolis projects.

mPack Basics

mPacks are libraries of pre-fabricated mTropolis components. These libraries are installed automatically when you install mTropolis.

Each mPack contains elements and behaviors related to a specific authoring task. For example, the Buttons mPack contains many types of pre-programmed buttons that can be customized for use in any your projects.

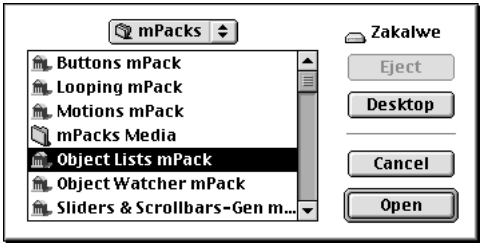


The Buttons mPack, Motions mPack, and Sliders & Scrollbars mPack

Opening mPacks

mPacks are standard mTropolis library files — they can be opened in the same way you would open any other mTropolis project or library.

To open a specific mPack, choose **File ➛ Open** (⌘-O). A standard file selection dialog box appears. Use the dialog box to navigate to the folder in which you installed mTropolis. This is the folder that contains the mTropolis editor and player icons. Inside this folder is a folder named **mPlugins.** The mPlugins folder contains a folder named **mPacks.** The mPacks folder contains the individual mPack libraries. Select the mPack you want to open and click the **Open** button.



*Opening an mPack from the **File ➛ Open** dialog box*

The selected mPack opens as a new library palette.

Closing mPacks

Like any other mTropolis library, mPacks can be closed by clicking the close icon found in the upper left corner of the mPack palette.

Using mPacks

Any object in an mPack can be used by simply dragging its icon from the mPack palette and dropping it in a mTropolis editing window. Note that element objects can be dropped on scenes, while behavior and modifier objects can be dropped on scenes, elements, or other behaviors. The documentation for each mPack component describes how that component is intended to be used.

Note: Many mPack components use the same aliases. If multiple copies of an mPack component (or two mPacks that use the same aliases) are dropped in a project, mTropolis displays the following warning:

mTropolis displays this warning whenever a copy of an existing alias is added to a project. The alert will appear even if the contents of the new alias are the same as the existing one. Clicking either **Replace** or the default, **Use Existing**, will usually have the same effect (because both the "new" and "existing" aliases are exactly the same). However, if you have customized any of the aliases used inside an mPack that is already in your project, click **Use Existing** to ensure that your work is not overwritten.

Each mPack has a special section component that contains documentation for each of the objects that the mPack contains. The use of this documentation component is described below. The **Tools** menu also contains a menu option that can be used to view mPack documentation.

Viewing mPack Documentation

There are two ways to view documentation for mPacks. The simplest is to choose **mPack Guide** from the **Tools** menu. The **mPack Guide** window appears. Like the other **Tool** menu options, this tool is actually a mTropolis

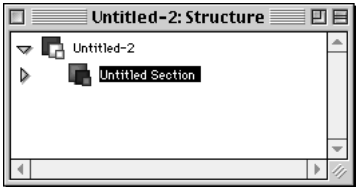
title that runs in a window. The **mPack Guide** contains interactive documentation for each mPack.

Another, more flexible way to view mPack documentation is to use the mPack documentation section included with each mPack. This documentation is useful because it is still in "project" form and can be examined in edit mode. You can examine (and reuse) all the programming Quark used in creating the mPack documentation and examples.

The section component that contains documentation can be found at the top of each mPack palette. For example, the Buttons mPack has a component named "About Buttons mPack." These section components contain mTropolis scenes that describe each component and show an example of that component in action.

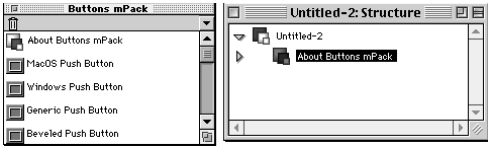
To use these sections:

1. Create a new mTropolis project by choosing **File** ➝ **New** ➝ **Project** (⌘-N). A new Layout window for that project appears.
2. Display that project's Structure window by choosing **Structure Window** from the **View** menu (⌘-2). The project's Structure window appears.
3. Click the Untitled Section component found in the Structure window and delete it by pressing the Delete key, or by choosing **Clear** from the **Edit** menu.



Choosing the existing Untitled Section component so it can be deleted

4. Drag the "About" component from the mPack palette and drop it on the project component in the Structure window.



The About Buttons mPack component has been dropped on the project component of an untitled project

5. View the mPack documentation by running the new project. Choose **File** ➝ **Run** ➝ **From Start** (⌘-T). mTropolis enters runtime mode and the mPack documentation main menu appears. More information about using the mPack documentation can be found below.

mPacks Documentation Main Menu

All the mPacks documentation sections have a similar interface. When the section is first displayed, a main menu appears. This menu lists all of the components in the mPack.

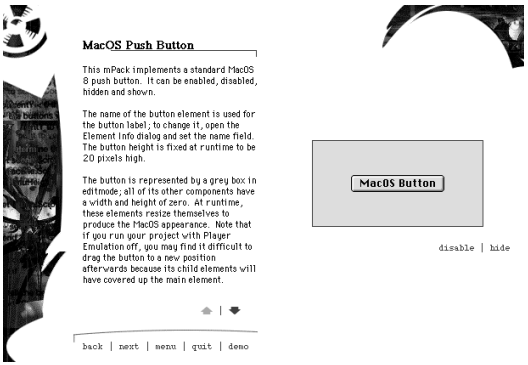
- Click a component name to display a scene that contains text documentation for that component and a working example.
- Click the "**quit**" button to quit the documentation and return to the mTropolis edit mode. You can also quit the documentation project by pressing ⌘-T.
- Click the "**next**" button to see documentation for the first component in the list.

*The main menu for the "About Buttons mPack" section. Click a title in the list (**Mac OS Push Button**) to see documentation and an example for that component*

mPacks Documentation Example Scenes

After clicking a component name in the main menu, a new scene appears. This scene contains a scrolling region on the left side of the screen that describes the component. On the right side of the screen is a working example of the component in action.

- Click the scroll arrows beneath the text region to view more text.
- Play with the working example on the right side of the screen by clicking or dragging the various interface elements. Some of the examples for the Motions mPack also use the keyboard.



Documentation scene for an individual mPack component. A scrolling text element on the left side describes the component. On the right side, a working example shows how the component looks and works in a project

- Click the "**quit**" button to quit the documentation and return to the mTropolis edit mode. You can also quit the documentation project by pressing ⌘-T.
- Click the "**back**" button (or press the left arrow key) to see documentation for the previous mPack component.
- Click the "**next**" button (or press the right arrow key) to see documentation for the next mPack component.
- Click the "**menu**" button (or press **m**) to return to the main menu.

The text descriptions of each component include a general introduction that describes the mPack component, followed by more detailed sections as described below:

- **User Parameters:** This section describes any variables at the top level of the mPack component whose values can be changed to customize the behavior of the component. To change the values of these variables, double-click the variable's icon to display its configuration dialog box. Note that the icons

of some mPack components may appear rather small in the layout view. Use the Structure window to get a clearer view of the elements and modifiers that make up an mPack component.

- **Messages Received:** This section describes any messages that can be *sent* to the component (using a Messenger modifier or Miniscript send statement) and the effects of those messages. For example, many mPack components support messages called "Enable" and "Disable." Sending the Disable message to the component causes it to become inactive (disabled button components become unavailable and do not respond to end user mouse clicks). Sending the Enable message to the component makes it responsive again.

- **Messages Sent:** This section describes any messages that can be generated by the component and the destinations for those messages. These messages can be used to trigger various actions in your project.

- **Internal Messages:** This section lists any messages used inside the component. Sending these messages to a component may produce unpredictable results.

- **Example:** This section describes the example shown on the right side of the screen.

mPacks Descriptions

Brief descriptions of the mPacks included with mTropolis are listed below.

Buttons mPack

The Buttons mPack contains elements that simulate clickable buttons in a wide variety of styles. Components include a Mac OS Push Button, Windows Push Button, Text Button, mToon Button, Radio Buttons, and Check boxes. It also contains behaviors that can be added to buttons to enhance their functionality.

This mPack is useful for creating mTropolis projects that look like standard Mac OS or Windows programs.

Looping mPack

This mPack contains behaviors that can be used to create "looping" (repeating) actions in mTropolis. Components include a looping behavior, timed looping behavior, and a looping behavior that makes multiple clones of other mTropolis objects.

Motions mPack

This mPack contains sophisticated behaviors that give elements different kinds of complex motions. Components include behaviors that enable various types of bouncing, orbital motion, controlled ground motion (for creating elements that "drive" like cars), and controlled space motion (for creating elements that "fly through space" like a spaceship). It also contains a behavior that detects arrow key presses for use with the ground and space motions, a behavior that makes an element's motion "wrap around" inside of its parent element, and sample "cone" mToons that can be used with the motion behaviors.

Object Lists mPack

This mPack contains behaviors that relate to lists of mTropolis objects. These behaviors help automate the rather tedious process of creating lists of mTropolis objects. The auto hide/show behavior can be dropped on an element to cause all of that element's children

to automatically hide or show themselves whenever the element is shown or hidden. The "List Child Elements" behavior creates an object list that contains all of an element's children, no matter how deeply they are nested.

Object Watcher mPack

This mPack is designed for use with the Object Watcher tool (described in Chapter 6 of the *mTropolis Reference Guide*. The ObjectWatcher behavior can be used to make the Object Watcher tool display information about an object during run-time.

Sliders & Scrollbars mPacks

These mPacks contain elements that simulate scrollbars and sliders in a wide variety of styles. Examples include horizontal and vertical Mac OS scrollbars, horizontal and vertical Windows scrollbars, scrolling text fields, and horizontal and vertical sliders. Each different look (Mac OS, Windows, and a generic grayscale appearance) is stored in a separate mPack library.

These mPacks are useful for creating mTropolis projects that look like standard Mac OS or Windows programs.

How mPacks are Installed

When mTropolis is installed, the mPacks are copied to their own folder, named **mPacks**, found inside the **mPlugins** subfolder of the folder in which you installed mTropolis.

Enabling Automatic Opening of mPacks

If you use certain mPacks regularly, you may want them to open automatically when you launch mTropolis or begin working on a specific project.

To open mPacks automatically when mTropolis is launched:

- Choose the desired mPacks in the finder.
- Choose **Make Alias** from the Finder's **File** menu (⌘-M).
- Move the new alias(es) into the **Startup** subfolder of your **mPlugins** folder.

To open mPacks automatically with a specific project:

- Open the project with which you want to automatically open mPacks.
- Choose **Edit** ➛ **Preferences** ➛ **Project**.
- Choose **Libraries** in the **Show** popup menu.
- Click **Add** to choose each mPack that should open automatically when the project opens.
- Click **OK**.
- Save the Project.

Installing New mPacks

To install a new mPack, simply copy the new mPack library file (and any media folders that go with that mPack) into the **mPacks** folder, found inside the **mPlugins** subfolder of the folder in which you installed mTropolis. To make the new mPack open automatically, put an alias of that mPack inside the **Startup** folder, as described in "Automatic Opening of mPacks," above.

Remember that mPacks are simply mTropolis library files, so it's easy to customize them or make your own. For example, you might copy all of the Mac OS-style buttons, sliders, and scrollbars into a single mPack.

Simple mPack Tutorial

This brief tutorial shows how to use the Buttons mPack to create clickable buttons that hide and show another mTropolis element. The instructions below assume that you have some familiarity with the basic use of mTropolis (you should have completed the QuickStart Tutorial — A Simple Slideshow described in Chapter 6).

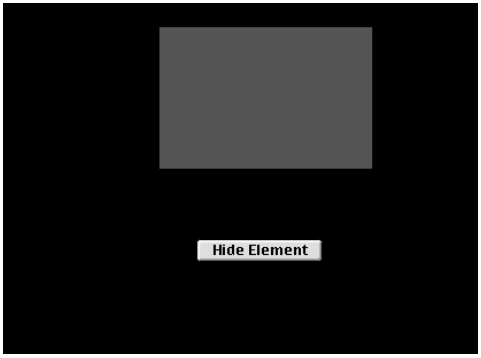
Start a New Project and Create a Graphic Element

1. Create a new mTropolis project by choosing **File** ➛ **New** ➛ **Project** (⌘-N). A new Layout window for that project appears.
2. Use the Graphic tool to create a new element of any size somewhere on the scene. Double-click the element to display its **Element Info** dialog box. Change the name of the element to `My Element`. Click **OK** to dismiss the **Element Info** dialog box.
3. Select the element and use the foreground color swatch on the **Tool** palette to assign a color other than black to the element. We want to make sure we can see the element during run-time mode! Alternatively, you could link a PICT, mToon, or QuickTime media file to the element.

Add a Button from the Buttons mPack to the Scene

1. Open the Buttons mPack.
2. Drag the icon labeled "Mac OS Push Button" from the Buttons mPack and drop it somewhere on the scene, preferably somewhere below the graphic element we created in the previous steps.
3. You'll see a gray box that represents our new Mac OS-style button. As described in the Button mPack's documentation, this button gets its label from the name of the button element itself.

   Double click the button element (not the modifiers found on that element) to display its **Element Info** dialog box. Change the button's name from the default `Mac OS Push Button` to `Hide Element`. Click **OK** to dismiss the **Element Info** dialog box.
4. Press ⌘-T to run the project. Note that the button has all the behavior you would expect in a standard Mac OS button. It changes appearance when clicked and registers a "click" only if the mouse is released while the cursor is actually inside the boundaries of the button. The button causes a system beep when clicked, but doesn't do much else. Press ⌘-T again to return to edit mode.



*A graphic element and the **Hide Element** Mac OS-style push button*

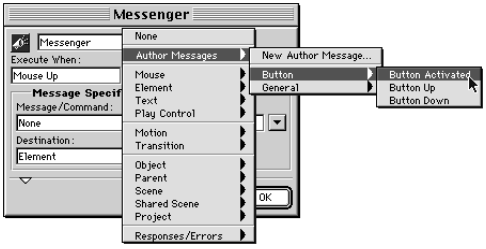
Add Functionality to the Button

Now let's customize our Mac OS-style button to actually make it hide the graphic element when clicked.

1. As noted in the documentation for this button, the push button receives a "Button Activated" message when it is clicked. We'll use a Messenger modifier to listen for this message and send the **Hide** command to the graphic element in response.

Drag a Messenger modifier from the **Logic** palette and drop it on the button. Be careful not to drop it inside of the behavior icon that is already present on the button.

2 Double click the Messenger modifier icon to display its configuration dialog box. Use the **Execute When** pop-up menu to choose **Author Messages ➛ Button ➛ Button Activated.** Note that the **Button** menu found in the Author Messages selection was added to our project automatically when we added the Mac OS-style push button to the project.



Selecting the Button Activated message

3 Use the **Message/Command** pop-up menu to choose **Element ➛ Hide.**

4 Use the **Destination** pop-up menu to choose **Element's Siblings ➛ My Element.** Your Messenger modifier dialog box should look like the one shown below.

Messenger configured to send Hide to My Element in response to the Button Activated Message

5 Click **OK** to dismiss the Messenger modifier's configuration dialog box.

6 Press ⌘-T to run the project. Now when the **Hide Element** button is clicked, the graphic element should become hidden, disappearing from the scene. Press ⌘-T to return to edit mode.

Add a Show Button

It would be nice to be able to show our element again. We'll duplicate our existing Hide Element Button and change it slightly to turn it into a **Show Element** button.

1 In edit mode, click the gray **Hide Element** button to select it. Make sure you click in the gray area of the button element itself and not on the modifiers found on the button.

2 Duplicate the button by pressing ⌘-D. A copy of the button appears. Move this new button somewhere below the **Hide Element** button.

3 Double click the new button element (not the modifiers found on that element) to display its **Element Info** dialog box. Change the button's name from `Hide Element` to `Show Element`. Click **OK** to dismiss the **Element Info** dialog box.

4 Double click the Messenger modifier icon found on the **Show Element** button. The Messenger modifier configuration dialog box appears. This Messenger is currently configured to send the **Hide** command. Use the **Message/Command** pop-up menu to choose **Element ➛ Show** instead. The dialog box should now look like the following example.

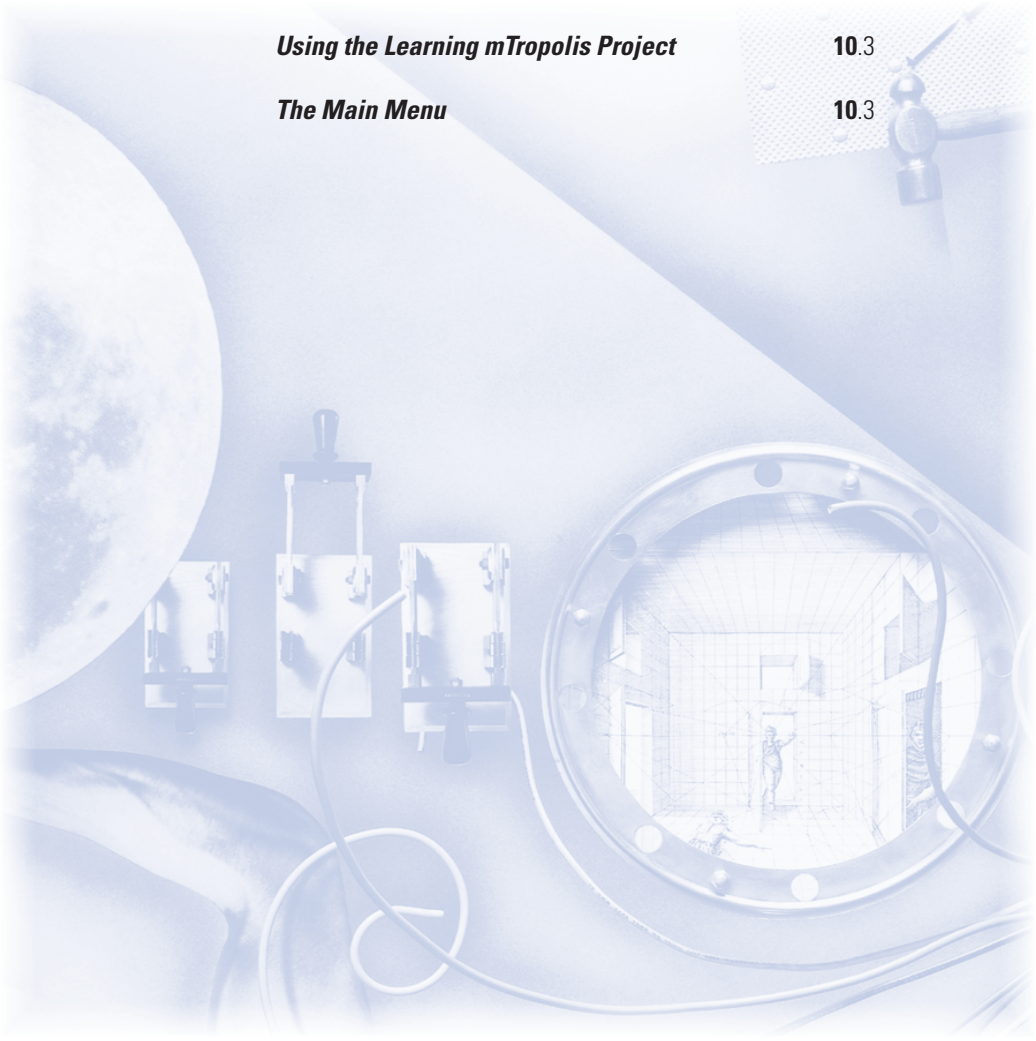
Messenger configured to send Show to My Element in response to the Button Activated Message

5 Click **OK** to dismiss the **Messenger** modifier dialog box.

6 Press ⌘-T to run the project. When the **Hide Element** button is clicked, the graphic element should become hidden. When the **Show Element** button is clicked, the graphic element should appear on the scene once again. Alternately click the `Hide Element` and `Show Element` buttons until the novelty wears off. Press ⌘-T to return to edit mode.

7 That concludes the simple mPack tutorial! Take a break and think of other fun things to do with mPacks.

10. Learning mTropolis Project



10. Learning mTropolis Project

This chapter describes the "Learning mTropolis" project which includes many examples of using mTropolis to create both basic and advanced interactive multimedia. 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/Examples subfolder of your mTropolis installation.

Running and Examining the Project

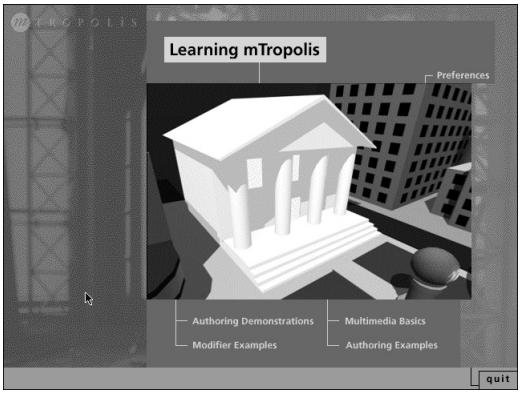
To examine this project, launch mTropolis then choose **File ➛ Open.** A standard file selection dialog box appears. Use this dialog box to choose the **Learning mTropolis** project found in the **Documentation/Examples** subfolder of the folder in which mTropolis is installed. Click **Open.** The Learning mTropolis project's Layout window appears.

To run the project from its beginning, press ⌘-T to enter run-time mode. The project begins playing from its first scene. Click anywhere to display the main menu.

The Learning mTropolis startup screen

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.

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 run-time mode.



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

The Main Menu

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

Authoring Demonstrations

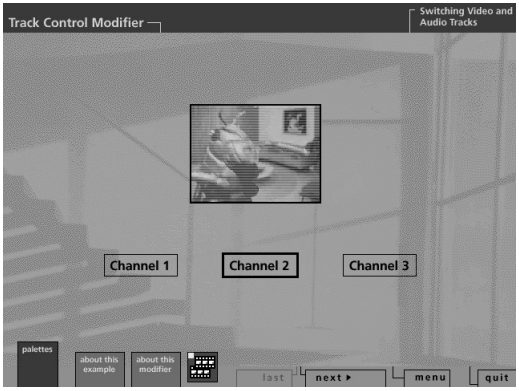
The "Authoring Demonstrations" section of the project presents two QuickTime movies that show examples of authoring in mTropolis. A slider at the bottom of the screen can be used to jump forward or backward through the selected movie. There are two demonstrations:

- **mPuzzle Demonstration:** This demonstration shows the creation of an animated puzzle, very similar to the mPuzzle tutorial found in Chapter 7.
- **Goldfish Demonstration:** This demonstration shows the creation of a simple project in which a goldfish swims back and forth across the screen while a baby fish circles around it. It shows how to create interactive behaviors, mToons and simple scripts in mTropolis.

Modifier Examples

The "Modifier Examples" section shows each of the mTropolis modifier palettes. Click any of the modifiers to see at least one example of using that modifier. Clicking a modifier takes you to a different scene that illustrates the use of that modifier. Each example scene contains an **About this Modifier** button, **About this Example** button, and a picture of the modifier's icon. Click each button for more information about the example.

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



What's playing on RoachTV? Find out in the Track Control modifier example

Multimedia Basics

The scenes in this section illustrate how common multimedia features can be programmed in mTropolis. Click any of the example names (such as "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 in your own projects.

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

Examples in this section include:

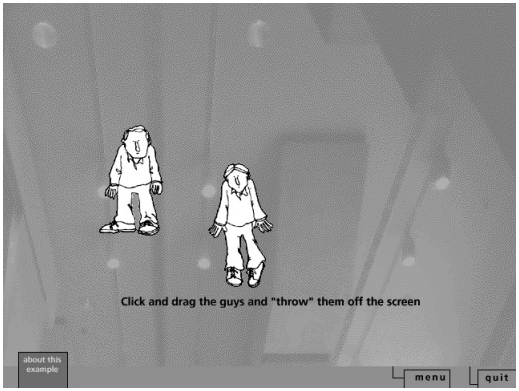
- **Button Gallery**: Shows different types of button behaviors.

- **Calculated Fields:** Shows fields that display time and date, plus a simple slider.
- **Changing Cursors:** Demonstrates changing the cursor as it passes over different parts of the screen.
- **Communicating:** Illustrates using mTropolis messages to simulate real world conditions. Two lightbulbs and three switches in an electrical circuit are modeled.
- **Controlling Audio:** Demonstrates playing, mixing, and changing the volumes of sounds.
- **Controlling mToons:** Demonstrates changing the ranges and rates of mToons.
- **Linear Navigation:** Shows navigation between scenes.
- **Revealing Objects:** Demonstrates hiding and showing objects.
- **Scene Navigation:** Uses scene changes to simulate navigating a 3D space.
- **Scene Transitions:** Illustrates the different transitions that can be applied during scene changes.
- **String Functions:** Demonstrates the use of many mTropolis string (text) manipulation functions.

Authoring Examples

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

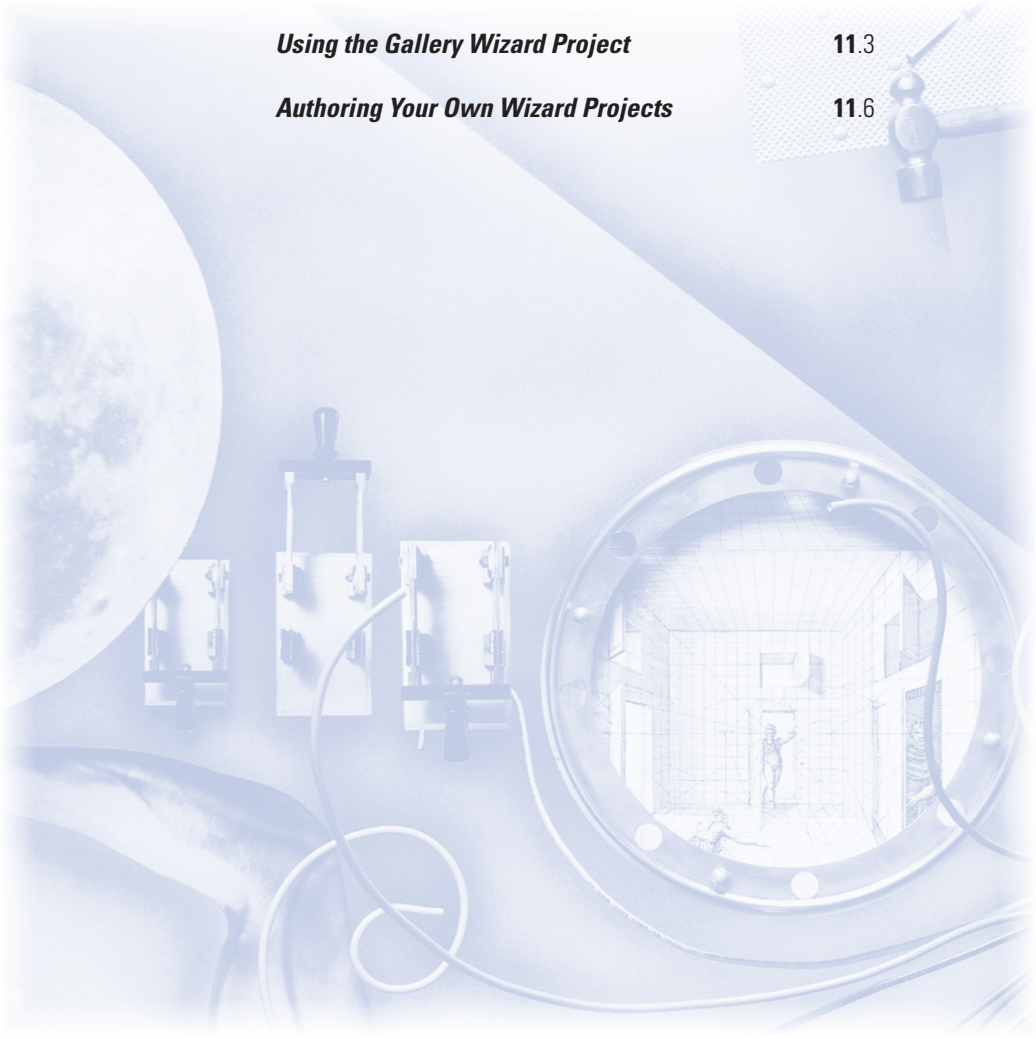
- **Autonomous Behaviors:** This example shows how mTropolis programming is independent of the media used in an object. Clicking the "brain" icon in the upper right corner of the scene causes the behaviors of the bug and butterfly to be switched.
- **Character Interaction:** This example uses a collision detection modifier to detect when the end user-controlled frog contacts a butterfly. The frog "eats" the butterflies when they are hit. Keyboard Messengers are used to detect the end 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 (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 "balance" attribute is used to pan the sounds as the end user pans around each node.
- **QTVR Objects:** Click this button to display a project that uses QuickTime VR object movies to show the items available in an interactive catalog.
- **CBT Module:** Click this button to display a "computer-based training" project. Scenes show the end user how to assemble an electrical motor then test the end user's proficiency.
- **mPuzzle:** This is a finished version of the puzzle tutorial described in Chapter 7, "In-Depth Tutorial — mPuzzle."
- **Throw Guy:** Pick up mToons and "throw" them off the screen, with surprising results.



Look out below, it's Throw Guy

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

11 Wizard Authoring Example



11. Wizard Authoring Example

This chapter describes the "Gallery Wizard" project. A "wizard" is a special type of mTropolis project that can be used to automate the creation of new mTropolis projects. Wizard projects use some of the features described in Chapter 6 of the *mTropolis Reference Guide,* "Tools Menu." The Gallery Wizard is a fairly sophisticated example of the types of projects that can be created.

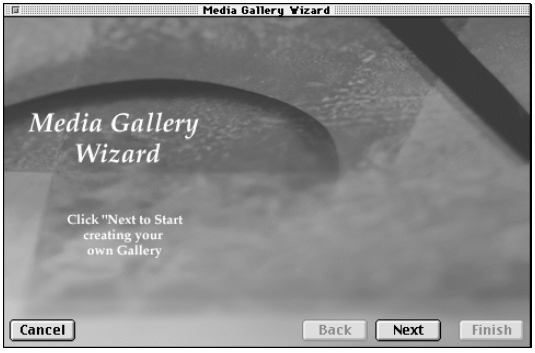
Using the Gallery Wizard Project

This project can be found in the **Wizard Authoring Example** folder found inside the **Documentation/Examples** folder of your mTropolis installation.

Running and Examining the Project

To run the wizard project, choose **Gallery Wizard** from the **Tool** Menu. The figure below will appear. To examine this project, launch mTropolis then choose **File ➝ Open**. A standard file selection dialog box appears. Use this dialog box to choose the **Gallery Wizard** project. The **Gallery Wizard** project's Layout window appears.

Press ⌘-T to run the Gallery Wizard. This project uses the Window Prefs modifier to run inside of a window. The project begins playing (see figure below), but the mTropolis interface continues to be visible.



The Gallery Wizard main menu

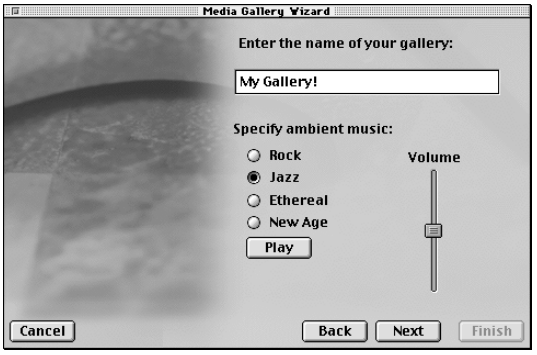
A wizard is a "helper" application that guides an end user through a series of steps to create a customized project or portion of a project. The Gallery Wizard creates multimedia "photo album" projects that can contain images, movies, and text annotations.

Follow the instructions below to use the Gallery Wizard to create your own gallery project.

Starting a New Gallery

Click the **Next** button on the Gallery Wizard's first screen to start creating a new gallery.

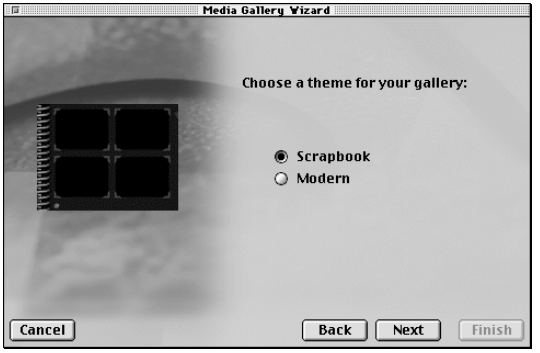
After clicking the **Next** button, a new scene appears (see figure below). This scene has controls that allow you to enter a name for your gallery, select background music to be played while browsing the gallery, and set the volume level for the background music.



The Gallery Wizard name selection scene

When you are through entering a name and selecting a music type, click the **Next** button. The Gallery Wizard's theme selection scene (see figure below) appears.

The theme selection scene contains buttons that allow you to select one of two different gallery styles ("Scrapbook" or "Modern"). Both gallery types are functionally identical. They differ only in their appearance. The Scrapbook theme creates a gallery that looks like a traditional photo album. The Modern theme creates a more contemporary- looking gallery.



The Gallery Wizard theme selection scene

Once you have selected a theme for your scrapbook, click the **Next** button. The Gallery Wizard creates a new mTropolis project (that appears behind the Wizard) in which gallery elements will be placed. The Gallery Wizard displays its page layout scene (see figure next page). This scene is used to create a new page in the gallery.

Creating Gallery Pages

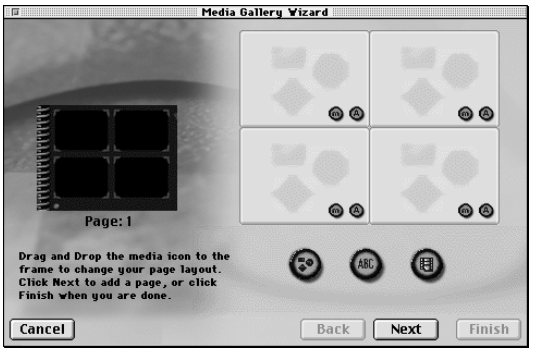
The Gallery Wizard's page layout scene (see figure on next page) is used to select the images, movies, and text that will be displayed on each page.

Each gallery page can display up to four elements. These four elements are represented by the four quadrants shown in the upper right corner of the page layout scene.

By default, these elements are graphic elements, but they can also display text or QuickTime movies. To change the type of an element, drag one of the three circular icons (that represent graphic, text, or QuickTime movie) that appear below the page and drop it on one of the quadrants. The background of that quadrant changes to reflect the new media type.

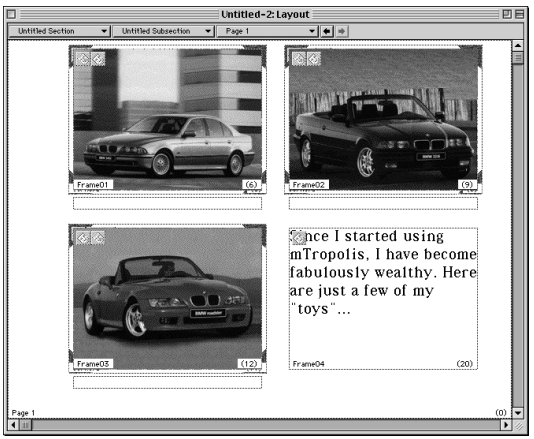
When you have chosen the type of media for each quadrant, media can be linked to that quadrant by clicking that quadrant's **m** button. If the quadrant is a graphic element or QuickTime element, a file selection dialog box appears. Use the file selection dialog box to select a picture or movie from the **Sample Media** folder, found inside the **Wizard Authoring Example** folder. If the quadrant is a text element, a text entry field appears. Use it to enter a witty description.

In addition, each graphic element displayed on the page can have a caption. Click the **A** button in a quadrant to display a text entry field. Use it to enter the caption for that element.



Creating a new gallery page

When you are finished designing your page, click the **Next** button. The Gallery Wizard creates the new gallery page and adds it to the gallery project that it created earlier. The figure below shows one of these pages. The Gallery Wizard displays a new page layout scene that can be used to create more pages in the gallery project. If you want to create more pages, use the page layout controls to add more pictures, movies, and text.



A scene from a project created by the Gallery Wizard

When you are finished adding pages to your gallery project, click the Gallery Wizard's **Finish** button. The Gallery Wizard closes, leaving mTropolis in edit mode with your new gallery project ready to run.

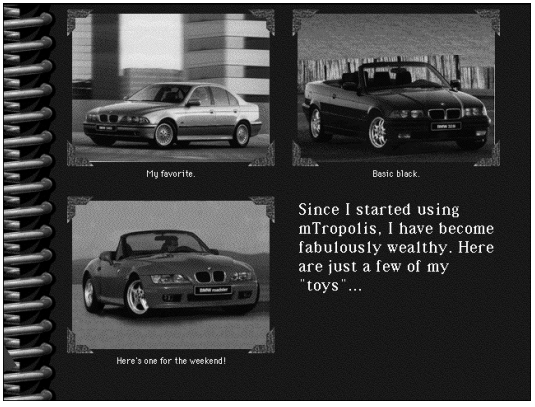
Running the Completed Gallery Project

After you have clicked the Gallery Wizard's **Finish** button, your new gallery project is ready to run. Click the gallery project's Layout window to make it the current project, then press ⌘-T to run the project from the beginning. mTropolis enters run-time mode and displays your gallery project.

The completed gallery project has an opening scene (see figure below) that displays the title of your gallery. Click the **Next** button to display the individual pages in your gallery.

The opening scene of a completed gallery project

The figure below shows a sample gallery page. If there are more pages in your gallery, there will be a **Next** button to click to display the next gallery page. Each page also has a **Back** button to return to the previous gallery page.



Browsing through a completed gallery project

When you are finished examining your gallery project, press ⌘-T again to return to edit mode. You can save your gallery project by selecting **File ➛ Save** or by pressing ⌘-S, just as you would with any other mTropolis project.

Examining the Gallery Wizard Programming

If you want to examine the programming that makes the Gallery Wizard work, simply open the **Gallery Wizard** project (found in the **Wizard Authoring Example** folder found inside the **Documentation** subfolder of the folder in which mTropolis is installed) once again and use the different mTropolis editing views to examine the project instead of running it.

Authoring Your Own Wizard Projects

The Gallery Wizard Authoring example is provided as a source example to use as a model for the creation of your own wizards. The process of creating a wizard project is similar to the creation of mTropolis tools. See Chapter 6 of the *mTropolis Reference Guide,* "Tools Menu" for a list of features that are useful in the creation of tools and wizards.

In brief, the following mTropolis features are useful when creating wizard projects:

- The Window Prefs modifier, which allows mTropolis projects to run inside of a window.
- The "Asset" attribute, which allows you to dynamically link media to a project in edit mode.
- mPacks (described in Chapter 9) which contain pre-built mTropolis components such as buttons and sliders.

The following sections describe some of the basic techniques used to create wizard projects.

Wizard Project Creation

Wizard projects are created entirely within mTropolis, just like any other project. Before starting a wizard project, do some basic design work. A quick storyboard of your project will help you determine the window size for your wizard. You generally do not want to change the dimensions of this window during the authoring process. Make sure you factor in the amount of information you need to display in each scene of the project and the screen size of the target platform.

Create a new project, then choose **Edit ➛ Preferences ➛ Project** and set the **Draw Area Size** to your wizard window size. Note that title bars are drawn outside of this area. It's convenient to set the draw area size at this early stage in the project, because each new scene you create will be this size.

The Window Prefs modifier enables projects to be run within a window. Using the Structure window, drag this modifier from the **Extras** and drop it on the project component. Open the Window Prefs modifier's configuration dialog box, set the window size to the same dimensions you previously entered as the draw area size. Also select an appropriate window style (palette is usually the best option) so that your wizard window will "float" above your other windows.

Project Structure

A wizard project is split into two parts — the *wizard interface* and the *wizard template*. The interface presents the appropriate information to the end user, and that gathered information is used to configure the template portion. The configured template (or parts of that template) can then be copied (via cloning) into a new mTropolis project (created by the wizard using the "newProject" attribute).

The interface portion of a wizard project is authored like any other mTropolis project. In the case of the Gallery Wizard example, all navigation controls are located on the shared scene of a single section. Author messages have been set up to send appropriate messages to the Active Scene destination. A minimum of two behaviors are found in each scene of the Gallery Wizard interface. One behavior sets the state of each navigation button, while the other receives the navigation control messages. Since the end user may alter their option settings on a particular scene many times, no action (such as cloning, linking media, or renaming elements) is taken until the navigation message is received.

The template section of a wizard project is manipulated via attributes. For example, if the end user is given the ability to adjust the volume of an object, the object's volume attribute can be accessed directly. See Chapter 15 of the *mTropolis Reference Guide,* "Attributes," for documentation of each attribute.

In the Gallery Wizard project, a good example of template manipulation and cloning can be found in the "Foundation Creation" behavior on the scene named "Create Gallery Foundation" in the wizard section.

Cloning

After you have manipulated objects in your template section you can clone them into a new project. For example, the "Foundation Creation" behavior mentioned above, uses a Miniscript modifier with the following script to create a new project and set its basic appearance:

```
-- Create a new, empty project:

set system.newProject to true

-- Get the index number of the new
-- project:

set newProjectIndex to \
system.projectCount
```

```
-- Store an object reference to the
-- project:

set oDestination to \
system.project[newProjectIndex]
```

The variable `oDestination` is an Object Reference Variable that stores a reference to the new project. This Object Reference is then used in all subsequent authoring as the cloning destination.

Using the new structure attributes (which let you "walk" the structure hierarchy without having to know the names of objects in the project), the wizard project can access any objects in the new project. See Chapter 15 of the *mTropolis Reference Guide,* "Attributes," for a table of structure attributes and complete descriptions.

For a good example of object manipulation and cloning, examine the "Foundation Creation" behavior in the scene named "Create Gallery Foundation" in the wizard section of Gallery Wizard example.

Asset Linking

Another mTropolis feature that is useful in the creation of wizard projects is the Asset attribute. This attribute can be used to link an element to an external media file, just as if the end user had selected **File ➛ Link Media**. Using the Asset attribute in this way only works on projects running in the mTropolis editor (it does not work in the mTropolis player). For example, the Miniscript statement:

```
set myelement.asset to \
"My HD:My Media Folder:Image.pict"
```

would link the element `myelement` to the PICT image `Image.pict` found in the folder `My Media Folder` on the drive `My HD`.

Setting the Asset attribute to "" (an empty string) causes a file selection dialog box to be displayed. This feature lets the end user specify any media. This use of the asset attribute is demonstrated in the "link" scene of the wizard interface section of the Gallery Wizard example.

Note: The element type (PICT, QuickTime, mToon, or sound) must match the type of the file when an asset is linked in this manner. For example, to set the Asset attribute to an mToon, the element must have already been linked to an mToon asset.

Index

U

V

W


mFactory™


103691