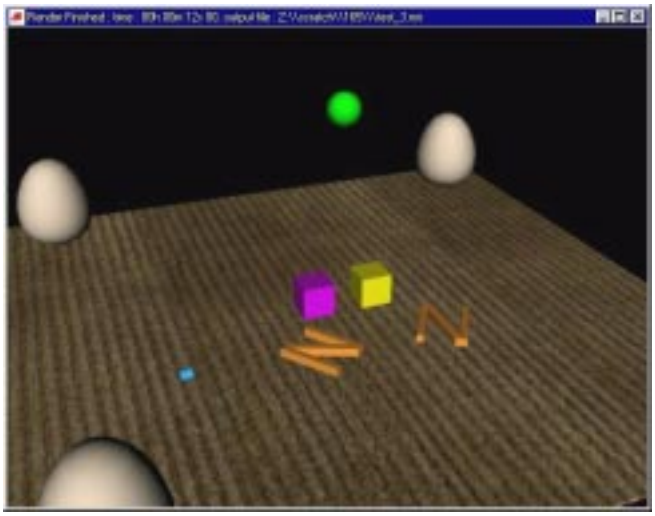


ANIMATING WITH MIRAI'S PROGRAMMING OPERATIONS

This tutorial will teach you how to incorporate Mirai's programming operations into your scripts.

ESTIMATED TIME REQUIRED

90 Minutes

WINGED EDGE TECHNOLOGIES

LEARNING GOALS

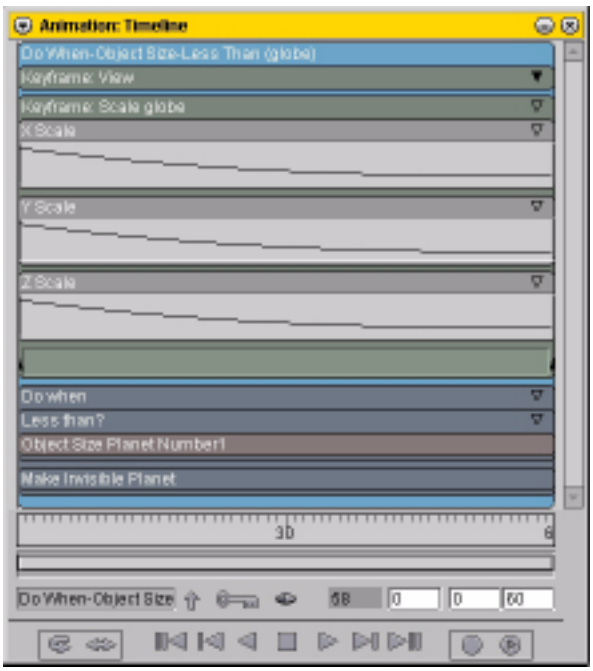
In this tutorial you will learn the following:

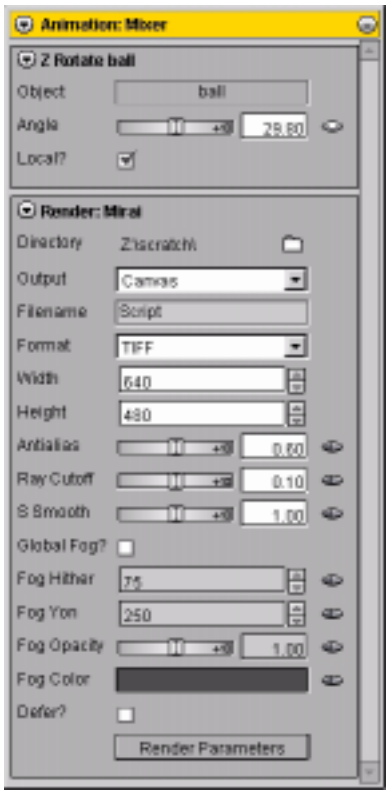
- A thorough understanding of how scripts are evaluated (animated)
- Practical uses for the various *Programming* class of operations

Before you can successfully begin to use programming operations, you need to have a clear understanding of what a script is and how scripts are evaluated in the timeline.

SCRIPTS AND THEIR STRUCTURE

When you animate, you change elements over time. In Mirai, you to this by creating scripts. Scripts can be modified in either the Animation: Timeline or the Animation: Mixer.

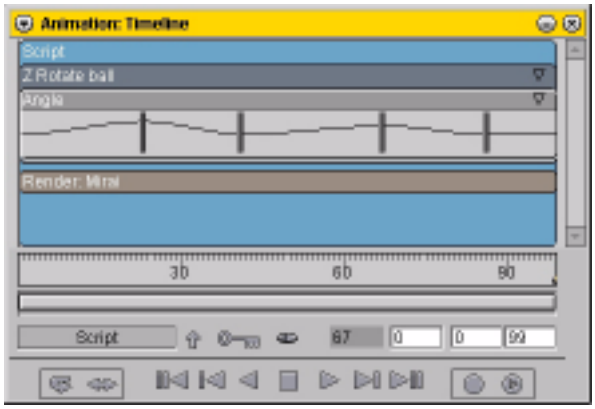




A script is organized into a hierarchical grouping of *channels*. There are two basic types of channels:

- those that perform some operation
- those that pass a value

Consider a simple script like this:



The highest level channel is the script itself. It contains another channel (*Z Rotate*) which in turn contains another channel (*Angle*).

When you animate a script, here's what happens:

1. Mirai increments the frame number. Normally, every frame is animated, but you can control the increment with a preference (*File>Preferences>Animation>Animation>Animation Increment*).
2. Once the current frame has been determined, Mirai starts evaluating the channels from the top down. If it encounters a channel which contains subchannels, those subchannels are evaluated first. So evaluation order is top-to-bottom and inside-out (that is, channels nested most deeply within a given channel are evaluated first). You can have operations nested as many levels as you want.
3. When all the channels have been evaluated, the frame increments again and the process is repeated.

There are all different kinds of operations, but all scripts follow the same basic idea. For example, as the simple script described above animates from left to right, here's what happens:

1. The frame is incremented (in this case, let's assume that the *Animation Increment* is 1) to frame 1.
2. Starting from the top down, the first channel encountered is the *Z Rotate*. However, Mirai detects that there is a subchannel (*Angle*). *Angle* is evaluated, and the value it contains at frame 1 is passed to the parent channel.
3. Now that the subchannel has been evaluated, the *Z Rotate* channel can be executed. It takes the value passed to it by the *Angle* subchannel, and rotates the specified object by that amount around the Z axis.
4. Mirai continues down and encounters the next channel, which is a *Render* operation. Since this channel has no subchannels, it is executed according to the channel properties you've set up ([M] to check the properties of any channel). You can also modify settings for the channel in the mixer.

VARIABLES

If you want to use the values passed by a channel in more than one place in a script, you must first *name* the channel. You do this by assigning the channel a variable. You can only assign variables to channels which pass values. For example, you cannot assign a variable to channel which contains a *Key-frame: Rotate* operation, but you can assign one to the curve channel beneath it. Note that you can assign variables to operation channels (such as *Add* or *Multiply*) whose purpose is to return a value.

To assign a variable to a channel:

1 [M] on the channel. In the *Set Variable* field, enter the name of the variable you want to use.

Be descriptive in your variable names. Also keep in mind that Mirai lets you assign the same variable name to more than one channel, so you need to be a little careful.

We'll describe later how to reference a variable from another operation.

REPLACING CHANNELS

When you add an operation to a script, Mirai makes its best guess about what kind of subchannels you'll want for that operation. However, since we now know that values can be passed either directly (e.g. from a curve channel) or from another operation (e.g., an Add operation), you need to know how to replace channels of one type with channels of another.

To replace a channel:

1 [R] on the channel and choose *Replace*.

Depending on the channel type, you'll see one or more of the following options:

- *Replace with Curve*

  Replaces the selected channel with a simple curve channel.

- *Replace with Constant*

  Replaces the selected channel with a constant value. The channel disappears and the constant becomes a property of the parent channel. You can [M] on the parent channel to check the constant's value.

- *Replace with Variable*

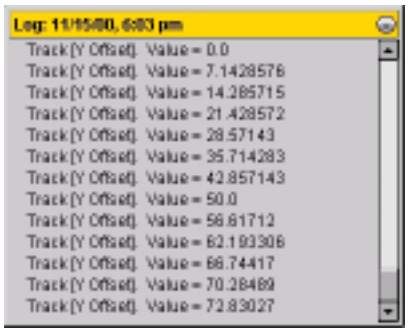
  Prompts you to select a variable that has been defined elsewhere in the script. After choosing a variable from the pop-up menu, replaces the selected channel with a *Get Variable Value* channel and assigns the selected variable to the channel.

- *Replace with Operation*

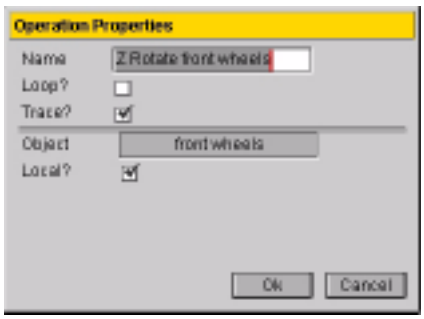
  Replaces the selected channel with an operation. When you choose this option, a list of possible operations is displayed in a pop-up menu. Choose the type of operation that should replace the channel.

USING TRACE AND THE PRINT VALUE OPERATION

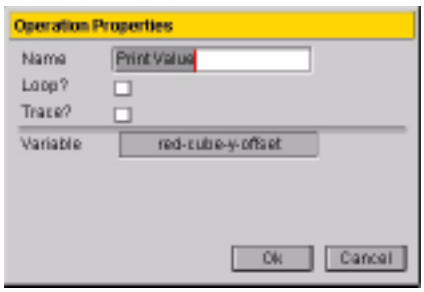
Both the *Trace* option and *Print Value* operation help you follow a script more closely as it is executing. The output for both of these options is sent to the Mirai Log Window. To open the Log Window, choose *Help>System>Log Window...*



The *Trace* option can be enabled for almost any channel (even those which do not return a value). If enabled for channel that passes a value, the value is printed; for other channels, the display varies. For example, it may display the transformation matrix for the element or other relevant information.



The *Print Value* operation is a little more specialized, in that it is used to print the value of a variable at each frame. When you add this operation, you're prompted to select the variable whose value you want to print at each frame. You can also [L] on the *Variable* field to select a variable for the operation.



PROGRAMMING OPERATIONS

The programming operations can be found in two classes of operations:

- Programming
- Geometry Programming
- Transform Combinations (selected)

The following operations are described in this tip:

Table 1 *Programming Operations*

OPERATION	DESCRIPTION
Add	Adds the values returned by two subchannels and passes the result to the parent channel.
Call Script	Executes one script from inside another, with various options for offsetting or compressing time.
Copy Transform	Copies the transformation matrix from one object to another.
Distance from Location	Measures the distance between an object and a specific set of coordinates and passes the result to the parent channel.
Distance from Object	Measures the distance between two objects and passes the result to the parent channel.
Distance from Plane	Measures the distance between an object and a specific plane and passes the result to the parent channel.
Divide	Divides the value in the first subchannel by the value in the second subchannel and passes the result to the parent channel.
Do when	Evaluates whether a condition is met; if so, some further operation is performed.
Do if then else	Evaluates whether a condition is met; if so one action is performed, if not a second action is performed.
Expression	Evaluates a LISP expression and passes the resulting value to the parent channel
Get Variable Value	Retrieves the current value of the channel referenced by the named variable and passes it to the parent channel.
In Range?	Tests whether a value passed from any subchannels is within a specified range. If so, the operation passes a “1” or “true” value to the parent channel, otherwise, it passes a “0” or “false.”
Initialize Transform	Initializes the transformation matrix of the specified object to its home state. This is the equivalent of using the Home command on the object in a 3D viewer.
Less Than?	Tests whether a value passed from any subchannels is less than a specified constant. If so, the operation passes a “1” or “true” value to the parent channel, otherwise, it passes a “0” or “false.”
Mix	Mixes the values of two subchannels together, based on a proportional value specified in a third subchannel.
Multiply	Multiplies the value in the first subchannel by the value in the second subchannel and passes the result to the parent channel.
Object Size	Retrieves the object size and passes the result to the parent channel.
Pause	Temporarily halts execution of the script when it encounters this channel. Execution can be completely halted or resumed using a pop-up dialog.
Perform Frame Action	Performs some LISP operation on the specified object.
Print Value	Prints the value being passed by the channel. The output is shown in the Log Window.
Renormalize	The Renormalize operation takes one set of values which are between a specified minimum and maximum range and converts them so that they fall between a new minimum and maximum. The result is passed to the parent channel.
Subtract	Subtracts the value in the second subchannel from the value in the first subchannel and passes the result to the parent channel.
Timewarp	Timewarp sets a curve that affects the apparent flow of time in a script.

SAMPLE SCRIPTS

The scripts included in the sample scene illustrate how to use (almost) all of the programming channels. (The main exception is the *Perform Frame Action* operation, which is available, but typically only used by those who have purchased the *Mirai Development Kit*.)

Note: In all the scripts below, we used *Keyframe: View* and *Set Camera Format* operations to put the camera where we wanted. We won't describe those operations for each script in our descriptions.

COPY TRANSFORM

Objects are animated in Mirai by manipulating their transformation matrix (often called "transform" for short). Keyframing translation, scaling, and rotation of an object are managed by modifying the transform, not the body of the object itself. The *Copy Transform* operation lets you apply the transform of one object to another so that they move similarly.

1. Select *Render Domain>Mirai*.
2. Select *Work Lights*.
3. Load the script "Copy Transform (Plane)."
4. Animate the script. The plane is animated using the *Fly along Trajectory* operation; this operation works by updating the transformation matrix so that the object moves to a position along the trajectory determined by the value in the *Fraction* channel.

5. Turn on the *Copy Transformation plane 2* channel.
6. Animate the script. This time both the plane and the saucer move along the wire. Notice, however, that there is no operation explicitly moving the saucer along the wire; instead, we copy the transform of the plane to the saucer at each frame and update the saucer's position accordingly.

7 Turn on the *Make Invisible plane* channel and animate again. The plane is still animated, but after the transform has been copied to the saucer, this channel hides it.

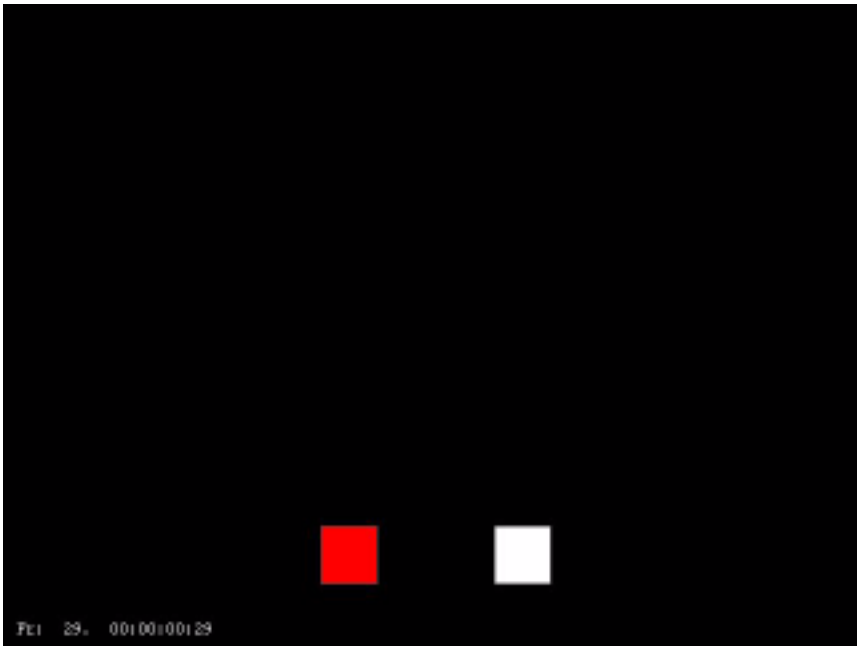
If you want to animate a number of objects in the same way, but don't want to animate each one explicitly, you can use the Copy Transform operation very effectively. You might, for example, use a script similar to this to render out a number of sequences for different airplanes. Its structure might look something like this:

```
Animate object 1
Make Invisible object 1

Initialize Transform object 2
Copy Transformation (object 1 to object 2)
Render object 2
Make Invisible Object 2

Initialize Transform object 3
Copy Transformation (object 1 to object 3)
Render object 3
Make Invisible Object 3
```

And so forth. With a structure like this, you could do several renders in one pass from a single script.

GET VARIABLE

In this script, the motion of one cube is driven by the curve associated with the motion of another cube, using the Get Variable operation.



1. Select *Render Domain>OpenGL.*
2. Select *Work Lights*.
3. Load the script “Get Variable (Plane)”. The script has four channels:
   - *Keyframe: View*
   - *Set Camera Format*
   - *Y Move red cube*
   - *Y Move white cube*

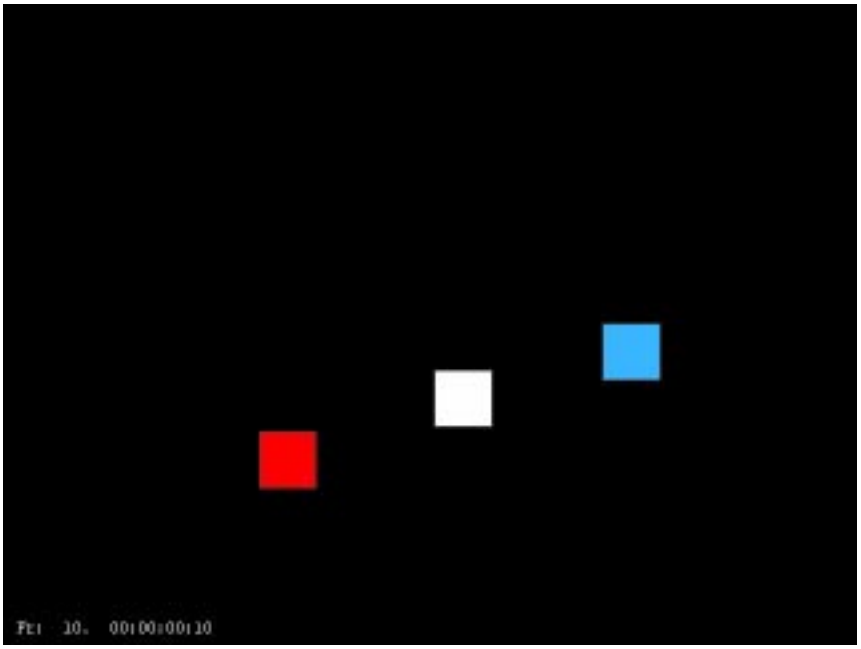
Note that the curve channel for the second *Y Move* operation has been replaced with a *Get Variable Value* operation. Here’s how you might use the *Get Variable* operation:

4. Keyframe the Y translation of the red cube.
5. [M] on the curve channel for the *Y Move* red cube operation and enter the variable name “red-cube-y-offset” in the *Set Variable* field.
6. [R] on the curve channel for the *Y Move* white cube operation and choose *Replace>Replace with Variable*. A pop-up menu appears under the pointer, showing all variables available in the current script. We chose the variable we assigned to the red cube’s translation.

The curve is automatically replaced with a *Get Variable Value* operation, and the variable that it uses appears in the channel’s label.

ADD

In this script, the motion of one cube is driven by adding the values found in two other curves. This script uses both the *Add* and *Get Variable* programming operations.



1. Select *Render Domain>OpenGL*.
2. Select *Work Lights*.
3. Load the script “Math-Add (Cubes)”.

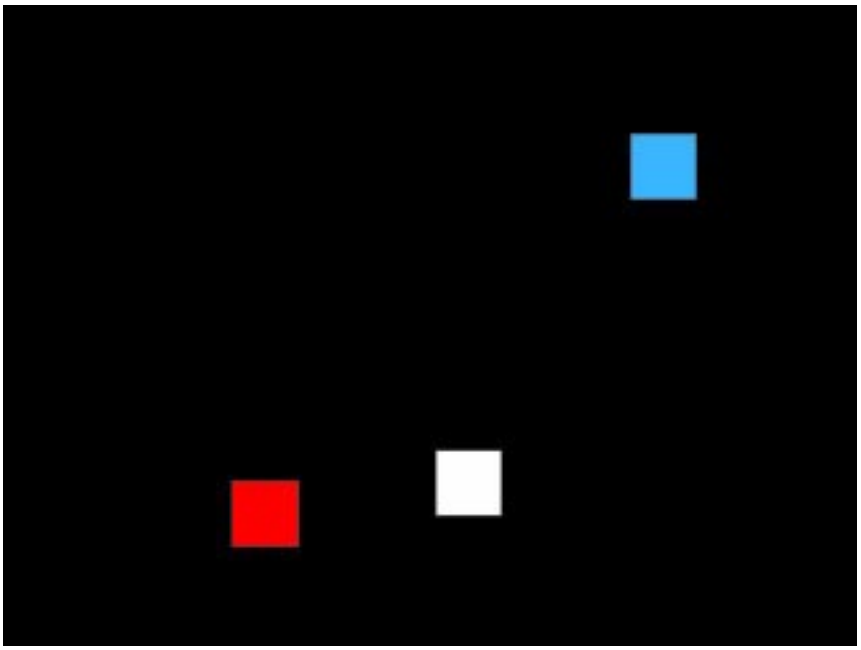
If you expand the *Y Move blue cube* channel, you can see that the curve channel has been replaced with an *Add* operation, and the curve channels that you might expect to see under an *Add* operation have been replaced with *Get Variable* operations.

4. Here’s how you might use the *Add* operation:
5. Keyframe the Y translation of the red cube.
6. Keyframe the Y translation of the red cube.
7. [M] on the curve channel for the *Y Move red cube* operation and enter the variable name “red-cube-up” in the *Set Variable* field.
8. [M] on the curve channel for the *Y Move* white cube operation and enter the variable name “white-cube-up” in the *Set Variable* field.
9. [R] on the curve channel for the *Y Move blue cube* operation and choose *Replace>Replace with Operation>Add*.
10. Open the *Add* channel. By default, the *Add* operation has two curve channels whose values are added together. The result of this addition is passed up by the operation. Since we want to add the values in the two other channels together, we need to do one more step.
11. [R] on the *Number1* curve channel and choose *Replace>Replace with Variable* and choose the first variable you want to add (in this case, “red-cube-up”).
12. [R] on the *Number2* curve channel and choose *Replace>Replace with Variable* and choose the second variable you want to add (in this case, “white-cube-up”).

Now, when you animate, the value from the two other curves are added together and passed up to the *Y Move blue cube* operation. At any given frame, Y translation values for the red and white cube are added together and used to translate the blue cube.

MULTIPLY

This script is similar to the previous script, except that we've moved the *Multiply* operation to the top level and retrieve its result using a *Get Variable* operation. Also, the two values are multiplied rather than added.



1. Select *Render Domain>OpenGL*.
2. Select *Work Lights*.
3. Load the script "Math-Multiply (Cubes)".

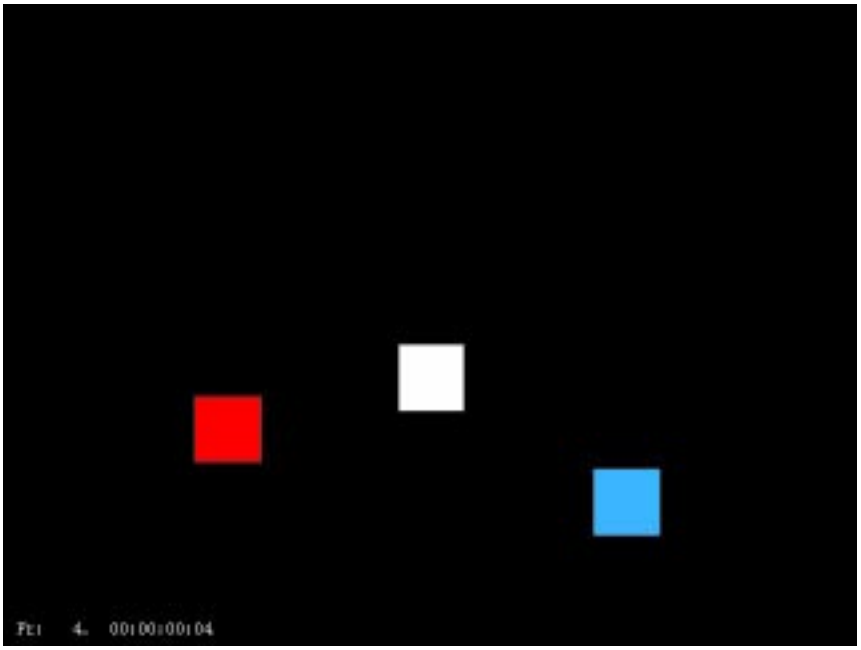
In this case, we added the *Multiply* operation at the top level, just as we'd add any other operation. The two curve channels were replaced with *Get Variable* channels which referenced the variables assigned to the curve channels for the red and white cube. However, because we moved the operation to the top level, we also needed to assign a variable to the channel itself. If you [M] on the *Multiply* operation, you'll see that it has been assigned the variable "red+white."

4. Finally, we did a [R] on the curve channel for the *Y Move blue cube* operation and did a *Replace>Replace with Variable* and chose "red+white" from the pop-up list.

The script structures in both this and the previous section are both completely legal. It's up to you to determine whether or not it makes sense to put the *Multiply* (or similar channel) at the top level or nested below the operation you wish to pass the result to.

SUBTRACT

This script is identical to the Math-Add script described above, except that the value in the second channel is subtracted from the value in the first channel at each frame.



1. Load the script “Math-Subtract (Cubes)”.
2. Animate the script.

Notice that the blue cube moves in a negative direction this time.

ALL

In this script, we’ve simply included all the various flavors of the math operations (except for the Divide operation) and assigned unique variables to each.

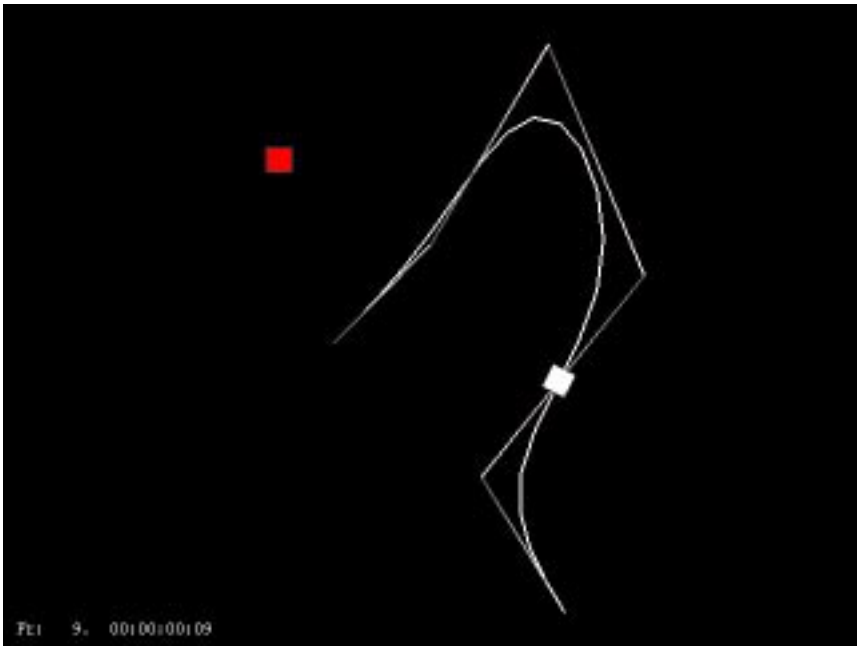
1. Select *Render Domain>OpenGL*.
2. Select *Work Lights*.
3. Load the script “Math-All (Cubes)”.
4. Animate the script.

You can keep all the math programming channels active while the script executes; while each performs the its assigned operation, the value is not being passed to anything except the script itself.

With a setup like this, you can simply [M] on the *Get Variable Value* under the *Y Move blue cube* operation and choose which variable you want to use. In this case, you can chose from the variable assigned to any of the math operations to drive the blue cube (red+white, red-white, or red*white).

RENORMALIZE

The *Renormalize* operation takes one set of values which are between a specified minimum and maximum range and converts them so that they fall between a new minimum and maximum. This is useful when you want to drive an operation that needs a different set of values; for example, rotations take values that are based around a range of 0-360, while displacements are animated using a typical range of 0-1. The renormalize function lets you convert the values in one range to another.



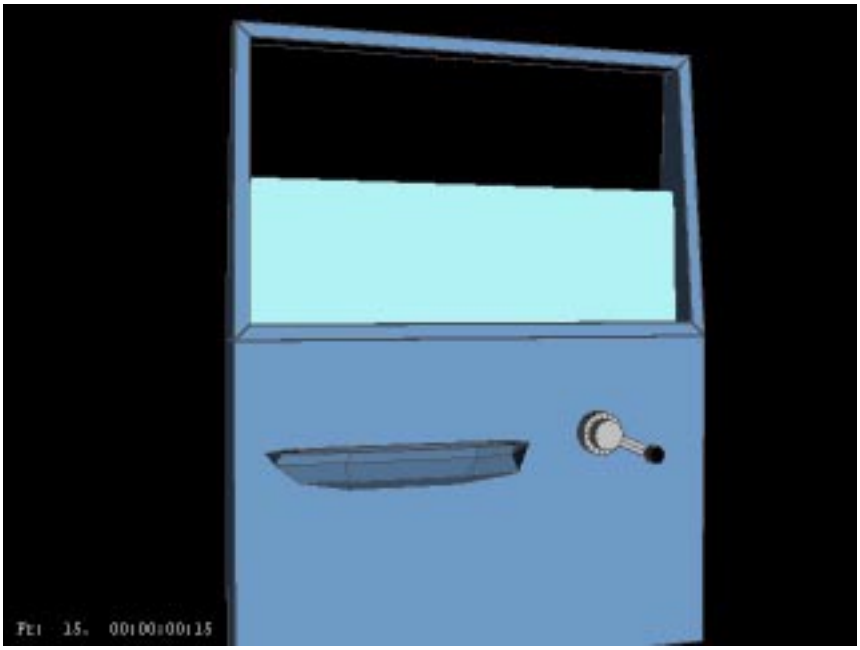
1. Select *Render Domain>OpenGL*.
2. Select *Work Lights*.
3. Load the script “Renormalize (Trajectory).”
4. Animate the script.
5. Open the *Y Move>Y Offset* channel. [M] on the cue in the middle of the script and note its value (250). The shape of the curve tells you that the maximum value for the cube’s Y translation will be 250. We’ve assigned a variable (red-cube-up) to the channel too.
6. The *Print Value* channel is just a check to print the value of (red-cube-up) to the *Log Window*. You can use a *Print Value* channel like this to check the maximum value reached during execution.
7. The next channel contains a *Fly Along Trajectory* operation which animates the white cube. We wanted the white cube to move a distance along the wire that corresponded to the distance traveled by the red cube. However, the *Fly Along Trajectory* operation takes values between 0 and 1, so we needed to renormalize the 0-250 values of the red cube’s translation to values between 0 and 1.
8. We replaced the curve channel for the *Fly Along Trajectory* operation with a *Renormalize* operation, then replaced the *Input* channel with a *Get Variable Value* operation that referenced the “red-cube-up” variable.
9. [M] on the *Renormalize* channel. We know the values passed up by the *Get Variable Value* are between 0 and 250, so we specify those values for the input minimum and maximum. The output values should be between 0 and 1.

With this setup, when the red cube is translated, the white cube translates correspondingly along the trajectory. When the red cube reaches its maximum height, the value (250) is converted to a value of one (1.0), which is passed to the *Fly Along Trajectory* operation.

Now, you can change both the input and output ranges to achieve different effects. Suppose, for example, that you want the white cube to animate only halfway along the wire when the red cube reaches its maximum translation. [M] on the *Renormalize* channel and change the *Output Max* to .5 and animate the script again. The white cube moves only halfway along the trajectory when the red cube is fully translated.

RENORMALIZE: PART 2

You can use the same principle describe in the previous section to drive all kinds of operations.



1. Select *Render Domain>OpenGL*.
2. Select *Work Lights*.
3. Load the script “Renormalize (Car Door)”
4. Animate the script.

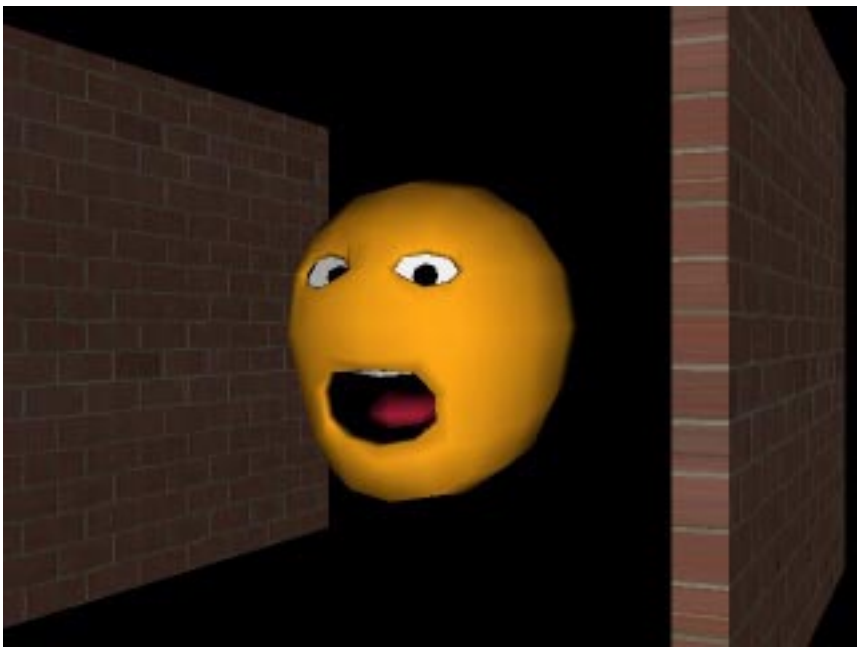
In this example, we want to drive the motion of the window based on the rotation of the window crank. So first, we animated the window crank, and determined the maximum value it would reach (1000 degrees).

Next, we built the structure of the script shown and interactively adjusted the *Output Max* value until the window moved the way we wanted it to (either in the Animation:Mixer or using [M] on the *Renormalize* channel).

You can experiment by changing the maximum value for the window crank (speeding it up or slowing it down), and changing the input maximum accordingly in the *Renormalize* channel to match that new value.

DISTANCE FROM OBJECT

In this script, you'll use the *Distance from Object* operation to make a character react; the closer an object comes to the character, the more excited he'll become.



1. Select *Render Domain>OpenGL*.
2. Select *Work Lights*.
3. Load the script "Distance from Object (Mr. Happy)".

Let's analyze how this script is put together:

4. The *X Move wall* channel animates the motion of the "wall" object. Assign a variable to the curve channel and name it "wall-move". You can keyframe the motion of the wall so that it comes close (but doesn't quite hit) Mr. Happy.
5. The *X Move Copy of wall* operation animates the copy of the wall along the same axis, but in a mirrored fashion around the X axis. To get that result, you need to multiply the X translation of the original wall object by -1. How can you do that? When you first add the *Y Move* operation, there's a curve channel under the operation. Replace the Y Move's curve channel with a *Multiply* operation. Then [R] on the first *Number* curve and choose *Replace>Replace with Constant* and set a value of -1. Finally, [R] on the second *Number* curve and choose *Replace>Replace with Variable* and choose the variable you assigned to the curve channel for the *X Move* operation (in this case, "wall-move").
6. Next, you need to find out how far the wall is from Mr. Happy. (Keep in mind that this operation measures the distance from object center to object center.) [M] on the *Distance from Object* channel. There are four fields in this channel:
   - *Object* is the first object for the test.
   - *Other Object* is the second object for the test.
   - *Normalize*, if selected, normalizes the actual returned values to new values between 0 and 1.
   - *Global* specifies whether to use local or global coordinates to test with.

7 In this case, make sure both check boxes are selected. [M] on the *Distance from Object* wall and turn on the *Trace* option. Turn the *Normalize* option off and animate again. Note that the values returned represent the actual distance (in world units) between the two objects. Now turn *Normalize* on and animate again. This time, values returned are between 0 (when the objects are closest together) and 1 (when the objects are furthest apart). In this case, the 0-1 range is much more useful to us, because we're going to animate displacements which also typically take values of 0-1. So we'll leave this option on for now. We also assigned a variable to this channel (and called it "wall-to-mr-happy").

8 The only problem now is that the values are "flip-flopped." In this case, we wanted more and more of a displacement to be applied as the distance shrank. We did that by subtracting the result of the *Distance from Object* test from 1. We did that by using the *Subtract* operation, replacing the first curve channel with a constant (set to "1") and replacing the second curve channel with the normalized variable previously assigned to the *Distance from Object wall* operation. We also assigned a variable to this channel (and called it "inverted-distance").

Now we're ready to animate the displacements. There are three operations left.

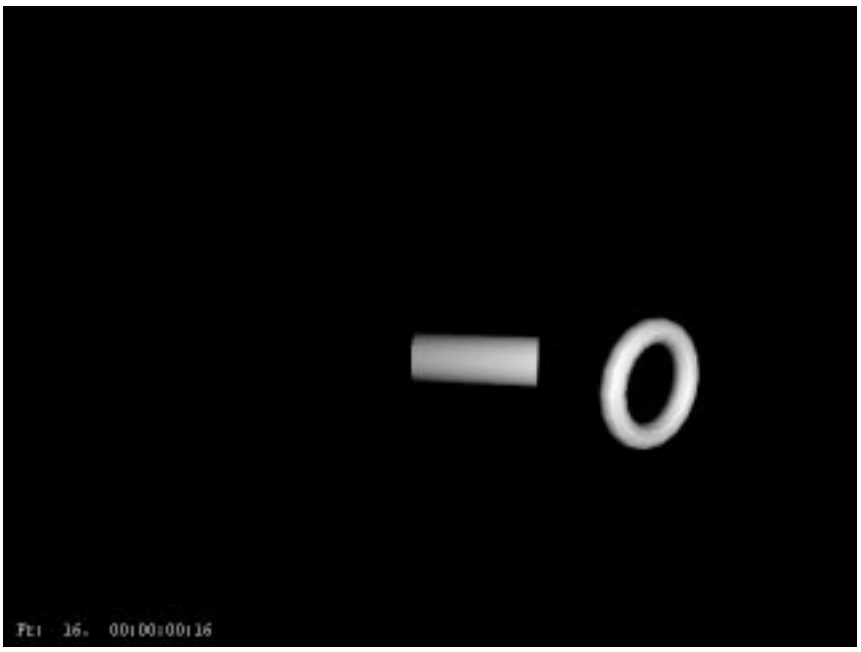
9 *Set Shape: home* initializes Mr. Happy to the specified state at each frame.

10 The curve channels for *Displace Shape: blink* and *Displace Shape: open* have been replaced with *Get Variable* operations, both referencing the "inverted-distance" variable.

Now when you animate, the two displacements are passed values between 0 and 1, depending on how close the wall comes to Mr. Happy.

MIX

The Mix operation lets you blend two curves. You can use this in a wide variety of situations, so we'll just explain how it works here.



1 Select *Render Domain>OpenGL*.

2 Select *Work Lights*.

3 Open the script “Mix (Torus)”.

4 Animate the script.

5 Open the *Mix Angle* channel. There are three subchannels, each of which contains a curve. The value passed is arrived at based on this formula:

```
(Number1 * (1-Proportion)) + (Number 2 * Proportion)
```

Simply put, this means that part of the value from the curve in the first channel is added to part of the curve value in the second channel. As the proportion value is increased, more of the second curve is passed to the parent channel.

You can use Mix when you want to modify a curve channel without destroying the original (for example, if it were reference through a variable by another operation).

PAUSE

The Pause operation lets you temporarily halt the execution of a script, perform some action in Mirai, then continue or halt execution of the script by responding to a dialog. A common use of the Pause operation is when doing some paint operation that requires human intervention on a sequence of images.

1 Load the script “Pause (Rotoscope)”. There are five channels in the script:

- Load from File
- Save Canvas to Image
- Pause
- Load Image
- Save Canvas to File

2 When you rewind or animate the script, you’re asked if you want to create a canvas. If you’re going to do some rotoscoping, you should create a canvas of the same dimensions as your images.

3 The *Load from File* loads a sequence of images to the canvas.

4 The *Save Canvas to Image* copies the image to an editable image. In this case, we created a dummy image (sample-image) to be the target image. At each frame, the next image in the sequence is copied to the sample-image.

5 When the *Pause* channel is encountered, a dialog appears asking whether or not to continue execution. At this point, you can edit the new “sample-image” using the paint tools (for example, to paint out some rigging or harness).

6 When you’re done editing the image, continue execution of the script. The next channel copies “sample-image” back to the canvas and the *Save Canvas to File* operation writes out the contents of the canvas to file (typically a different name and/or directory).

TIMEWARP

Timewarp sets a curve that affects the apparent flow of time in a script. For example, if you have a script that manipulates a number of objects, and wanted to slow down the beginning of an animation without adjusting all the curve channels for each operation, you could apply a Slow In curve on the whole script using the Timewarp operation. In order to do that, you'd add a channel to hold the Timewarp operation at the top of the script. The Timewarp operation affects all channels below it in the same parent channel.

The Timewarp operation works by using the value passed to the operation to determine which frame to execute in the script. If a value of 0 is passed, the first frame is executed, a value of .5 animates the frame halfway through the script, and a value of 1 animates the frame at the end of the script.

When you animate a script, as each frame executes, Mirai passes a value to each channel from top to bottom, telling it the current frame. When it encounters a Timewarp channel, however, that value is temporarily replaced by the value in the curve. The value of the curve at that frame tells Mirai what frame to execute for any affected channels (as described above). You can include multiple Timewarp channels to affect timing for individual channels or sections of a script.

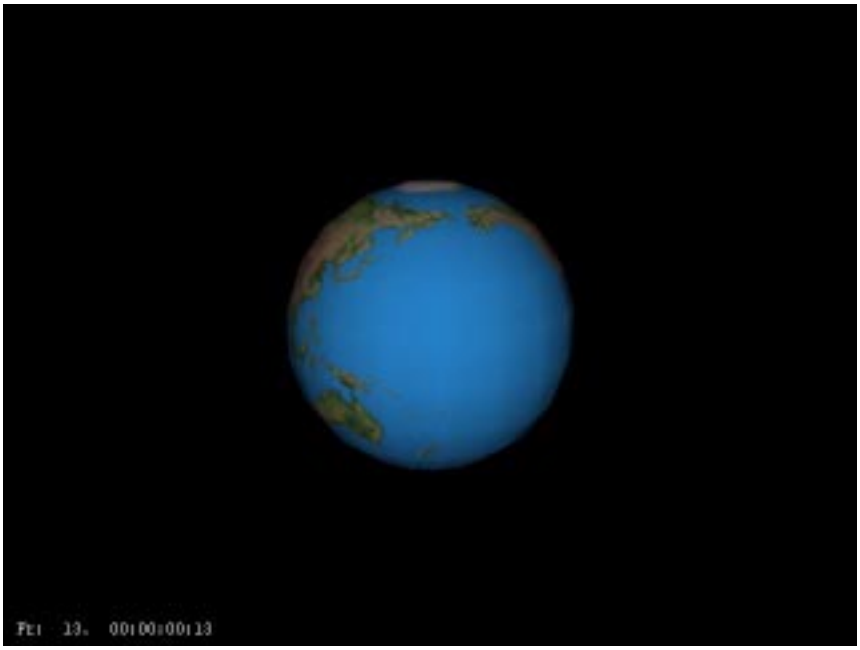
1. Select *Render Domain>OpenGL*.
2. Select *Work Lights*.
3. Load the script "Timewarp (Cube & Octahedron)".
4. Animate the script. Both the red and white cube move similarly along the Y axis.
5. Activate the *Timewarp* channel and animate the script again.

This time, the motion of the white cube has a slow-in feel to it. However, this was done not by directly changing the shape of the curve, but by changing which frame is executed for channels below the *Timewarp* operation. You can turn on the *Trace* option for the *Timewarp* channel to get a better idea of what's going on. For example, if you [L] at frame 15, note that the output value is only .18. This means that when you animate frame 15, the *Y Move* channel is actually animating at frame 5 (.18 x 30, rounded down). As the script continues to animate, the slow in values finally catch up so that the two cubes end up animating at the same true frame.

You can add multiple Timewarp channels to adjust the flow locally as needed.

DO WHEN OBJECT SIZE LESS THAN?

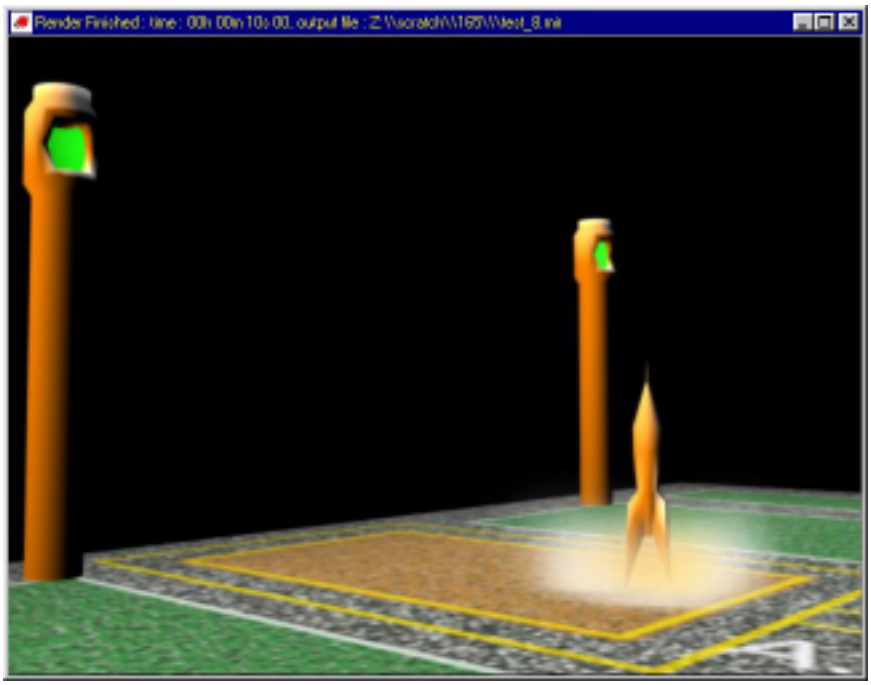
The Do When operation is the simpler of two test operations (the other is Do If Then Else, described later). This operation evaluates whether or not a certain condition is true; if it is, one or more operations are executed. You can use any of the "test" operations to perform the test, as described above.



1. Select *Render Domain>OpenGL*.
2. Select *Work Lights*.
3. Load the script "Do When-Object Size-Less Than (Planet)"
4. Animate the script. The script performs a simple test operation. The *Do When* operation lets you perform a test, and, if that test is true, then perform some additional action.
5. When you create a Do When operation, two containers are created inside the top level channel. You need to replace the first container with some sort of test operation. In this case, we chose a *Less Than?* operation that compares a passed value to a constant. In this case, it checks the value passed by the *Object Size* operation. You set the test constant with a [M] on the *Less Than* channel; note that it's currently set to 5.0, so the operation returns false if the passed value is greater than 5.0 and true when the value is less than 5.0.
6. As the globe gets smaller and smaller, its size eventually drops below 5.0; when it does, and *only* when it does, the rest of the *Do When* operation is executed (in this case, the planet becomes invisible). An operation like this could save lots of render time, for example, by making complex objects invisible when they became so small as to become nearly invisible.
7. Note that the operation to be carried out when the test condition is met should *not* be a dynamic operation. Mirai does not currently support "triggers" so you can't, for example, *start* an operation when test conditions are met. Operations that typically follow a test might include *Make Visible*, *Make Invisible*, *Keyframe: View*, *Activate Simulation*, and other operations which toggle states of objects, lights, or the camera. (The *Call Script* operation can be used to simulate "triggers"; see the section below.)

DO WHEN
IN RANGE?
DISTANCE FROM PLANE

Here's another example (a little more complex) of using the Do When operation to control flow of execution through a script.



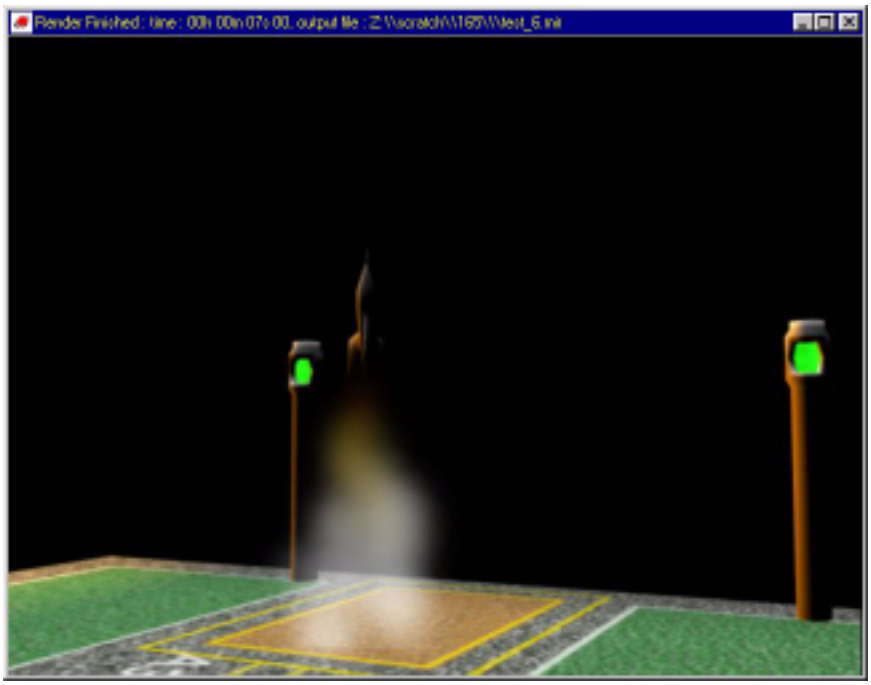
1. Select *Render Domain>Mirai*.
2. Select *Scene Lights*.
3. Load the script "Do When-In Range?-Distance from Plane (Rocket)"
4. Animate the script.
5. The rocket descends toward the launch pad. When it reaches a certain distance from the plane the following occur:
   - A camera move
   - A particle system is activated

Note that turning on simulations (such as particle groups) can be tricky. If the test for the Do When condition tests false at any point in the remainder of the script, the simulation stops during any such frames and is only continued when the condition once again becomes true.

In this case, we used the *Distance from Plane* operation as our test. We could just as easily have used *Distance from Object* or *Distance from Location* operations to test the condition. It just so happens that our scene has a flat plane which provides an easy test.

DO IF THEN ELSE
IN RANGE?
DISTANCE FROM PLANE

By now, you should be able to figure out how the script in this section works pretty easily. We introduce a new kind of test, the *Do If Then Else* operation:



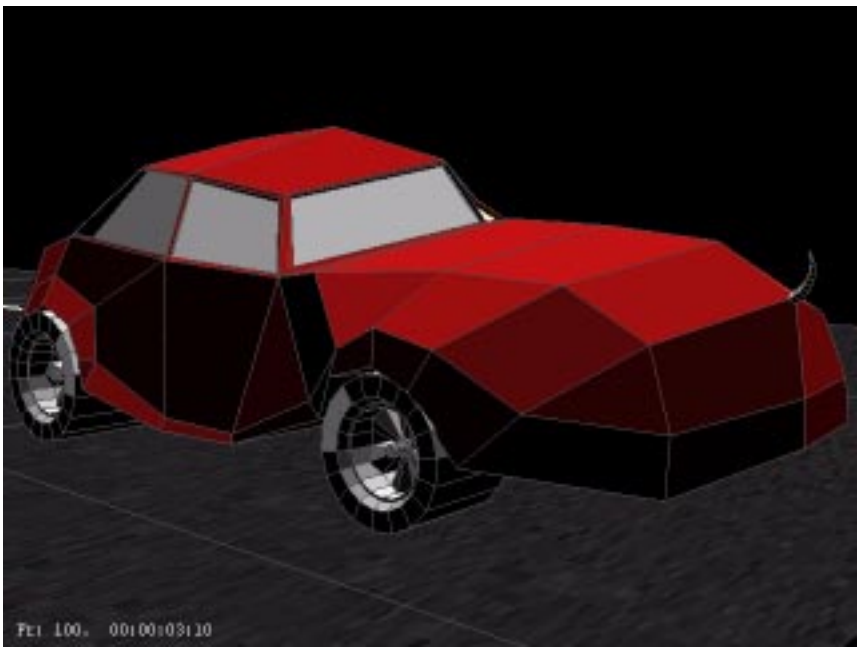
1. Select *Render Domain>Mirai*.
2. Select *Scene Lights*.
3. Load the script “Do if then Else-In Range-Distance from Plane”.
4. Animate the script

In this case, the rocket is taking off, and we want to trigger the lights to change from red to green (to signal a successful launch) when the rocket reaches a certain distance from the ground.

Just as with the *Do When* operation, a test is performed. However, one of two actions is *always* performed; the first if the test is true, the second if the test is false. In this case, the red lights are displayed when the *Distance from Plane* value is less than 40, and the green lights are made visible when the distance is greater than 40.

EXPRESSION

The Expression operation lets you enter a LISP expression which is evaluated at each frame. The operation returns a value. While typically reserved for users of Mirai's Developer Kit, we can show you a little bit of how this operation might be used.



1. Select *Render Domain>Mirai*.
2. Select *Scene Lights*.
3. Load the scene "Expression (Car)".
4. Animate the script.

The "magic" in this script happens through the LISP form that's been specified in the two *Expression* channels.

```
(/ (*-360(/(*DYNAMICS-INTERNALS::*CURRENT-FRAME-NUMBER*(GEOMETRY:3D-LENGTH(GEOM-
ETRY:FIND-BODY(GEOMETRY:OBJECT-NAMED "car-path"))))100))42.4)
```

It looks quite complex, but it's not too bad. The *Fly along Trajectory* moves the car along the trajectory from start to finish. At each frame, the expression retrieves the length of the wire "car-path" and divides it by 100 (the number of frames in the script). It then multiplies that value by -360 and multiplies it by 42.4 (the circumference of the wheel). This expression returns a value (in degrees) that says how much a cylinder of the specified circumference should rotate given its distance traveled along the wire.

Note that the tires are two different sizes. If you [M] on the *Z Rotate rear wheels>Expression* channel, then [M] on the *Expression* field, you'll notice that the only difference is the circumference of the wheel.

When you animate, both wheels rotate appropriately, the smaller front wheel (circumference=42.4) slightly faster than the larger (circumference=46.8) back wheel.

CALL SCRIPT

The *Call Script* operation calls the specified script from within the current script. If you are coordinating a large number of complex sub-animations in your script, you may find it useful to animate them in individual scripts, then create a main script which calls those other scripts as appropriate.

For example, if you created a main script to do the bulk of your animation, then created a second script to animate the walk cycle for a character, you could simply call the walk cycle script whenever you wanted to use it.

Also, because the script is modular, debugging problem areas of the script is easier.

In the same example, if you found an error in the walk cycle script, you could simply make a change to it, then reanimate the main script, and the motion would be corrected every place the walk cycle script was called.

The *Call Script* function has been described in greater detail in a separate tip which can be found at the following location:

http://www.wingededge.com/support/support_mirai_tips.shtml

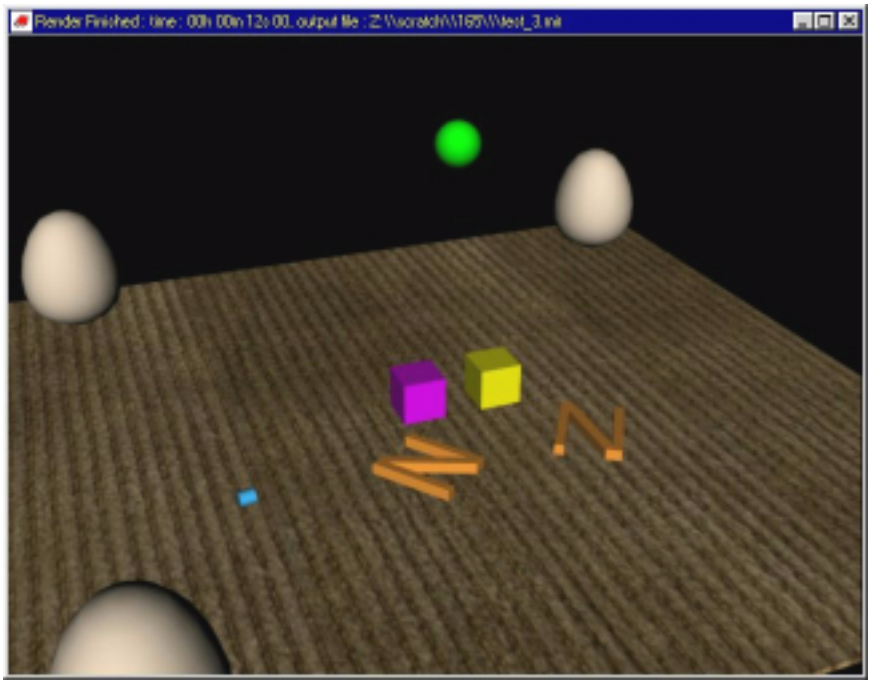
PERFORM FRAME ACTION

This operation lets you evaluate a LISP form at each frame; it is reserved for users of Mirai's Developer Kit. If you're a registered developer and want more information on this operation, contact us at the address below:

support@wingededge.com

GRAND FINALE

Finally, here's a sample of a complete "machine" which is animated based on the translation of a single cube and a variable assigned to a *Timewarp* channel. All other animation is driven based on the distance traveled by a single cube.



1. Select *Render Domain>OpenGL*.
2. Choose *Work Lights*.
3. Animate the script.

By now, you should be able to figure out the workings of the entire script!