Interactive Graphics Using *MathLink*

Chikara Miyaji, University of Tsukuba, JAPAN

Interactive graphics effectively expresses the dynamic structure of a model, and helps improve the intuitive understanding of concepts in the sciences. Also it can be used as a user interface tool.

Presently, *Mathematica* graphics are not interactive but instead are static and merely a side-effect of computation. It is not possible to select a part of a graphic using the mouse, nor can one make interactive dynamic relationships between graphical elements.

The graphics system introduced here is a *MathLink* program for creating interactive graphical objects which the user can manipulate. In this program,

- all graphics — point, line, curve, and text — are represented as objects which can be manipulated by mouse;
- all graphics are manipulated by a kernel evaluation;
- the relationship between objects can be defined as *Mathematica* functions.

This paper outlines the system and discusses its design and implementation.

1. Simple Example of Interactive Graphics

Interactive graphics is implemented using a template-based *MathLink* program (`draw.exe`). This program opens multiple windows, and creates various point, line, curve, and text objects in the window. For example, evaluation of the following expression, **`New[PointObject..]`** creates a point object on the window object **`win1`**. It returns a symbol **`self$10`** (this is explained later), which is assigned to **`point1`**.

```
point1 = New[PointObject, win1]
```

```
self$10
```

The following expression sends a **`move`** message to the object **`point1`** and moves the object to the location **`{100,100}`**.

```
point1[move, {100, 100}];
```

Similarly, a point object accepts **`dispose`**, **`getposition`**, and **`setposition`** messages.

`draw.exe` defines several *MathLink* external calls: **`GetMouse[]`** returns the current mouse location. **`StillDown[]`** returns the current mouse state; if the mouse is pressed, it returns **`True`**, otherwise it returns **`False`**.

When the user clicks on the point object, an expression **`self$10[click,{x,y}]`** is send to the kernel. Hence, **`{x,y}`** is the location of the mouse and **`self$10`** is the symbol of the object. Similarly, **`self$10[drag,{x,y}]`** will be sent to the the kernel when the mouse drags that object.

The **`drag`** method is (pre-)defined as a kernel function.

```
self$10[drag, {x_, y_}] :=
 While[StillDown[link], self$10[move, GetMouse[link]]]
```

This function moves the object **`self$10`** to the mouse position while the mouse is pressed. By this definition, the point object follows the mouse movement during dragging. Using similar strategies, any reaction to mouse events can be defined in the kernel.

Let's create a new point object and a line object on the window **`win1`**.

```
point2 = New[PointObject, win1];
line1 = New[LineObject, win1];
```

Define a kernel function, **`findline[`**p_1**`,`**p_2**`]`**, which returns the coordinates of the perpendicular bisector of the two points p_1 and p_2:

```
findline[p1_, p2_] :=
 Module[{p, x, y, sol},
  p = p1[getposition] - p2[getposition];
  sol = Solve[{{x, y}.p ä 0.0, {x, y}.{x, y} ä len}, {x, y}];
  {x, y} + midpoint[p1, p2] /. sol]

len = LineLength[line1]^2 / 4;

midpoint[p1_, p2_] := (p1[getposition] + p2[getposition]) / 2.0
```

$p = p_1$ `-` p_2 is the vector from p_1 to p_2. **`Solve`** finds *two* vectors, {*x*, *y*}, of (squared) length, *len*, which are perpendicular to *p* by solving the equation {*x*, *y*}.*p* ä 0, subject to the constraint {*x*, *y*}.{*x*, *y*} ä *len*. Adding {*x*, *y*} to the mid-point of points p_1 and p_2 results in the perpendicular bisector coordinates.

```
findline[pt1, pt2]
```

```
{{86.7157, 56.7157}, {143.284, 113.284}}
```

Then, we define the **`move`** method of **`point1`** below:

```
point1[move, {x_, y_}] :=
 (line1[move, findline[point1, point2]]; point1[setposition, {x, y}])
```

This method defines that if **`point1`** receives a **`move`** message, it calculates the new perpendicular bisector position, moves the line to this location, and then sends **`setposition`** to itself. Similarly, we define the method for **`point2`**.

```
point2[move, {x_, y_}] :=
 (line1[move, findline[point1, point2]]; point2[setposition, {x, y}])
```

With these definitions, if the user moves either point, the perpendicular bisector moves accordingly. This is executed by the following sequence: (1) when the mouse is dragged on the point object, a **drag** message is sent to the kernel; (2) from the **drag** definition, a **move** message is sent to the point object while dragging; (3) from the new **move** method definition, the **line** is moved to the perpendicular bisector position as the **point** object is dragged.

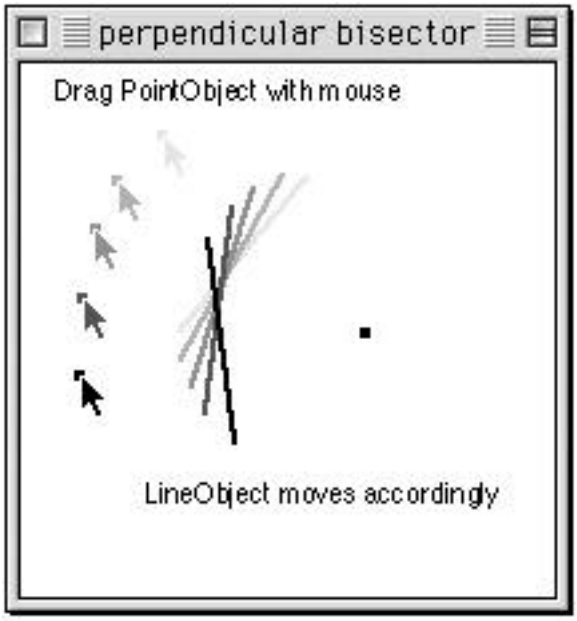

Figure 1. Moving either point causes the perpendicular bisector to move accordingly.

More complicated geometrical relationships are easily implemented by defining appropriate relationships between objects. Real-time implementation of such relationships is impossible with the current notebook front end or in existing interactive drawing applications.

2. Structure of Interactive Graphics

This Interactive Graphics system consists of the kernel, notebook front end, and two *MathLink* programs, `draw.exe` and `serializer.exe`. The system is constructed using an Object Oriented Programming Style (OOPS). The structure of system is divided into four parts:

- OOPS for *Mathematica*;
- wrapping *MathLink* external calls as an object method;
- handling a user's event to the object;
- the mechanism to send expressions from the *MathLink* program to the kernel.

2.1 Object Oriented Programming Style for *Mathematica*

Function closure is used to construct an OOPS in *Mathematica*[3]. The OOPS enables the creation of instance methods, class methods, and multiple inheritance. The expression

```
New[PointObject, w]
```

creates a point object in window object **w**, which is the parent of this object. This function is defined as below:

```
New[PointObject, w_] :=
 Module[{self, mypos = {10, 10}, oindex, super = w},
  self[dispose] := DisposePointObject[oindex, super[port]];
  self[setposition, {x_, y_}] :=
   MovePointObject[mypos = {x, y}, oindex, super[port]];
  self[getposition] := mypos;
  self[selector_, args___] :=
   findmethod[PointObject[self, selector, args], super[selector, args]];
  oindex = NewPointObject[mypos, super[port], self];
  self]

PointObject[self_, move, {x_, y_}] := self[setposition, {x, y}]
```

New[PointObject,w] returns **self** which is defined in the **Module[]**. In *Mathematica*, symbols in a **Module**, like **self**, are made unique by appending a number to the name, here **self$10**. Therefore **New** returns the function **self$10** with message arguments **dispose**, **setposition**, and **getposition**, and these are treated as the instance methods. If there is no pattern match with the instance method, **self[selector_, args___]** will be invoked, and this will look for class methods, then a super method. In the above example a class method **move** is defined for **PointObject**s.

Using this idea, menus, windows, points, lines, curves, and text can be defined as objects.

2.2 Wrapping *MathLink* External Functions as a Method

Although all objects are defined in the kernel, these objects call *MathLink* external functions as the action of the object internally. For example, **New** calls the **NewPointObject** function, the **dispose** method calls the **DisposePointObject** function, and the **setposition** method calls the **MovePointObject** function.

These external functions are created from the *MathLink* template. A template is a file which defines the relation between external C functions and *Mathematica* functions. When the *Mathematica* function is invoked, the arguments are passed to the corresponding external C function in the *MathLink* program, that C function is executed, and then the value is returned. Below is a template which defines the relationship between **NewPointObject** and the C function `newpointobject`.

```
:Begin:
:Function:      newpointobject
:Pattern:       NewPointObject[{x0_,y0_}, windex_Integer, oSymbol_Sym-
bol]
:Arguments:     { x0,y0,windex,oSymbol }
:ArgumentTypes: { Integer,Integer,Integer,Manual }
```

```
:ReturnType:    Integer
:End:
```

`NewPointObject` sends `oSymbol` which is assigned to a module symbol such as **`self$10`**. For this `oSymbol`, the *MathLink* program can send event expressions from this object such as **`self$10[drag,{x,y}]`**.

There are three basic methods for all graphical objects: **`create`**, **`dispose`**, and **`move`**. **`create`** sets up a new object's data; **`dispose`** releases this data; and **`move`** moves the object and draws the object in the new position.

Although making a complete interactive graphics program in C is rather difficult, implementing these three operations is relatively straightforward. In `draw.exe`, all the complicated relationships between objects and user events are defined in the kernel.

2.3 Handling User Events

Our *MathLink* program (`draw.exe`) sends user events to the kernel as expressions. Mouse operations such as click, drag or menu selection are captured by `draw.exe` and sent to the kernel. The form of each expression is a message to the object which the user has selected. For example, if there is a point object (**`self$10`**) and the user drags the mouse on the object then

```
self$10[drag, {x, y}]
```

is sent to the kernel. The reaction to the event depends on the definition of the method. We saw how the **`drag`** method is defined in a previous example. And in that method a *MathLink* external function is called to perform the reaction using `draw.exe` as shown in Figure 2.

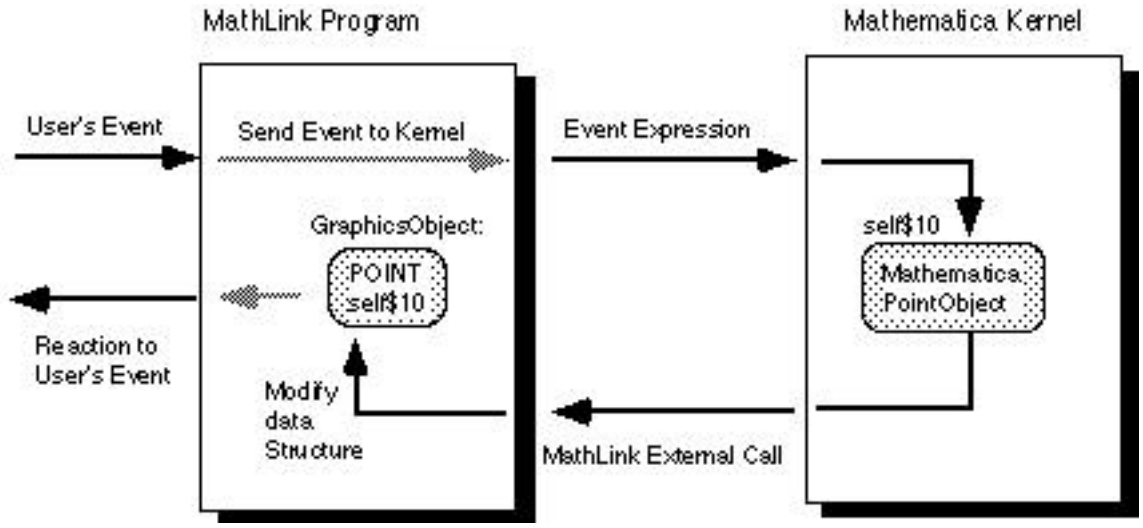

Figure 2. Flow of user events in `draw.exe`.

Because all reactions to events are defined in the kernel, the method definition is completely flexible. Methods can be defined during the execution of the *MathLink* program and, as shown in the previous example, making relationships between objects is very simple.

Basically `draw.exe` has two asynchronous inputs: one receives expressions from the kernel and the other receives user events. Although writing such multiple asynchronous input programs is not easy, our program uses the strategy of extending an existing program. The original source code of `draw.exe` was generated from the *MathLink* template automatically and then the handling of user events was added to the source program. This simplifies the programming considerably.

Currently, `draw.exe` runs on Macintosh and Windows 95/NT4 environments. In general, it is difficult to write system independent graphics programs but using a QuickTime 3.0 layer for the graphics results in 90% of the source code being shared between Macintosh and Windows platforms.

2.4 Sending Expressions to the Kernel

The kernel evaluates expressions from **`$ParentLink`**, which the front end uses. If the expressions from `draw.exe` need to be evaluated, these expressions should be sent to **`$ParentLink`**. For this reason, a relay program (`serializer.exe`) is created.

`serializer.exe` relays all expressions between the front end and the kernel and is therefore transparent to the user. Furthermore `serializer.exe` relays expressions from *MathLink* programs to the kernel. The kernel evaluates these expressions just as it does for expressions from the front end. Using `serializer.exe` *MathLink* programs can send their expressions *asynchronously*.

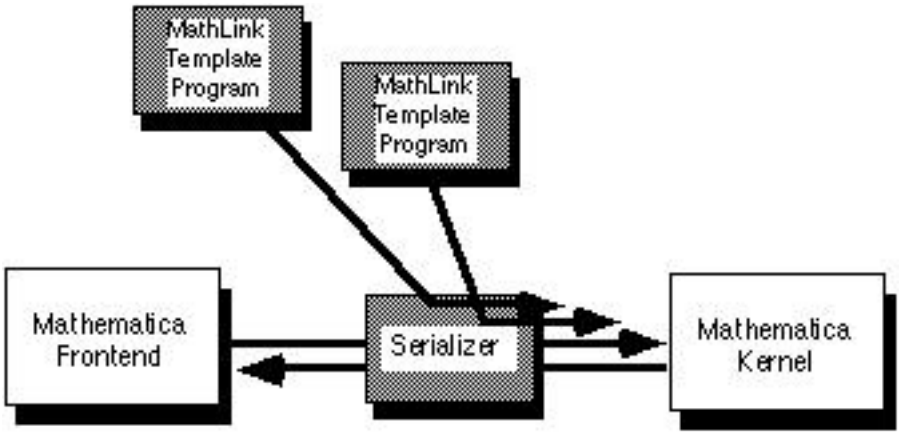

Figure 3. The relationship between `serializer.exe`, *MathLink* programs, the kernel, and the front end.

`serializer.exe` is a *MathLink* program which does the following:

- send expressions from multiple links to the kernel;
- manages the expression between each link and the kernel;
- handles the interrupt and requests from the links.

`serializer.exe` enables the sharing of a kernel with multiple front ends. This makes it possible to use special front ends in conjunction with the notebook front end. For example, an image processing front end[4], an interactive graphics front end, or a combination of both, will work with the current front end and extend its capabilities.

Also, connecting one `serializer.exe` to another `serializer.exe` makes it possible to exchange expressions between *Mathematica* sessions. This connection enables the exchange of notebook cells between two or more front ends which is useful for developing distance education tools[1,5].

Note that `serializer.exe` does increase the time taken for data transfer. Transmission of packets from the kernel to the front end for a typical *Mathematica* 3D graphics (~760kbytes) took 10 seconds using a local kernel and 16 seconds with `serializer.exe` — a 60% increase. However, there is no significant slow down for most practical uses because, in general, packets are usually smaller than 3D output.

`serializer.exe` was built by modifying a *MathLink* template program. Currently, Macintosh, Windows95/NT4, and HP-UX versions exist.

3. Application of draw.exe

One advantage of `draw.exe` is the use of kernel evaluation for the reaction to the user events. Such event reaction can be modified *dynamically*. There are interactive geometry programs such as Cabri, but the ability of these programs to define new geometrical relationships is limited. In contrast, using `draw.exe` one can define *arbitrary* relationships between geometrical objects as *Mathematica* functions. Hence `draw.exe` is not restricted to geometry but has a wide range of applications.

In the following example we show dynamical curve fitting. **`data`** is a list of 50 point objects which appear at random locations in the window **`win2`**. And we create a text object, **`t`**, for displaying the form of the function.

```
data = Table[New[PointObject, win2, PointObjectPosition Æ
     {Random[Integer, {50, 350}], Random[Integer, {50, 350}]}], {50}];

t = New[TextObject, win2, TextObjectFrame -> False,
  TextObjectRectangle -> {{10, 10}, {400, 50}}]
```

`fitpoints` finds the third-order polynomial best-fit curve using **`Fit`**, displays the function in a text object, and returns a list of 100 points which lie on this curve.

```
fitpoints[pts_] :=
 Module[{f},
  f = Fit[Map[#[getposition] &, pts], {1, x, x^2, x^3}, x];
  t[settext, ToString[f]];
  Table[{x, f} /. x Æ i, {i, 1, 400, 4}]]
```

We create a new curve object **`fitcurve`**. The best-fit curve appears in the window.

```
fitcurve = New[CurveObject, win2, fitpoints[data]]
```

After defining the relationship between the data points, **`data`**, and the **`fitcurve`** by the following expression,

```
Map[(#[move, {x_, y_}] :=
     (fitcurve[setdata, fitpoints[data]]; #[setposition, {x, y}])) &, data];
```

then moving *any* point causes the best-fit to be recalculated and displayed. **`Map`** allows us to define this relationship for all point objects at once.

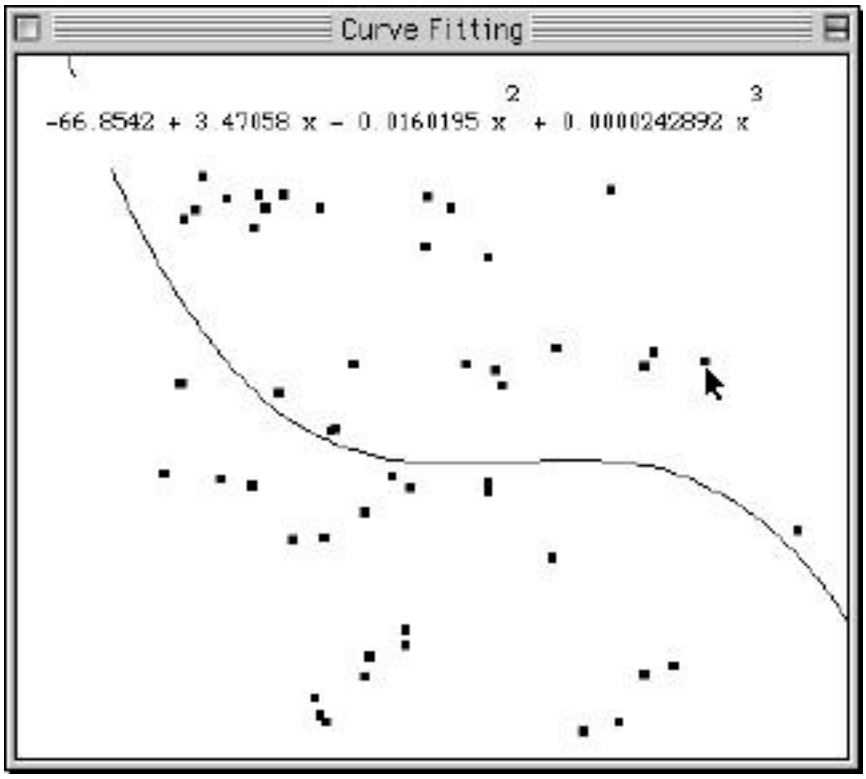

Figure 4. Moving any point object using the mouse causes the fitting curve to be redrawn.

This simple implementation of interactive curve fitting allows the user to examine the influence of the location of a data point on the best-fit curve *dynamically*. Also, changing third-order to fourth-order or to another function is very easy. This example demonstrates the power of the combination of interactive graphics and *Mathematica*.

4. Summary

An interactive graphics system was built using *MathLink*. In this system:

- all graphics are objects in the *MathLink* program (`draw.exe`), and can be manipulated by mouse;
- all objects are defined in the kernel and new methods or relationships between objects are dynamically defined as functions;
- the combination of interaction with user events and the powerful *Mathematica* interpreter makes it easy to create unique and flexible applications.

All executable programs and all source codes in this article are included in the author's book[2].

5. References

[1] Kimura, H. and and Miyaji, C.: Mathematica as a Communication Enhancement Tool in the Classroom, IMS '99, 1999.

[2] Miyaji, C. and Abbott, P.: *MathLink: Network Programming using Mathematica*, Cambridge University Press, 1999 (in press).

[3] Miyaji, C. and Kimura, H.: *Writing Graphical User Interface using Mathematica and MathLink*, Computational Mechanics Pub., Innovation in Mathematics, Proceedings of Second International Mathematica Symposium, 307-314, 1997.

[4] Sato, J., Jankowski, M. and Miyaji, C.: *Interactive Photo Editor FrontEnd using MathLink*, IMS '99, 1999.

[5] Yoshida, K.,Ohashi, S., Kimura, H., and Miyaji, C.: *Teaching High School Mathematics over a Network with Mathematica*, IMS '99, 1999.