

Making Mathematica Manipulate[] programming more efficient using event driven model

Nasser M. Abbasi

May 8, 2012

Contents

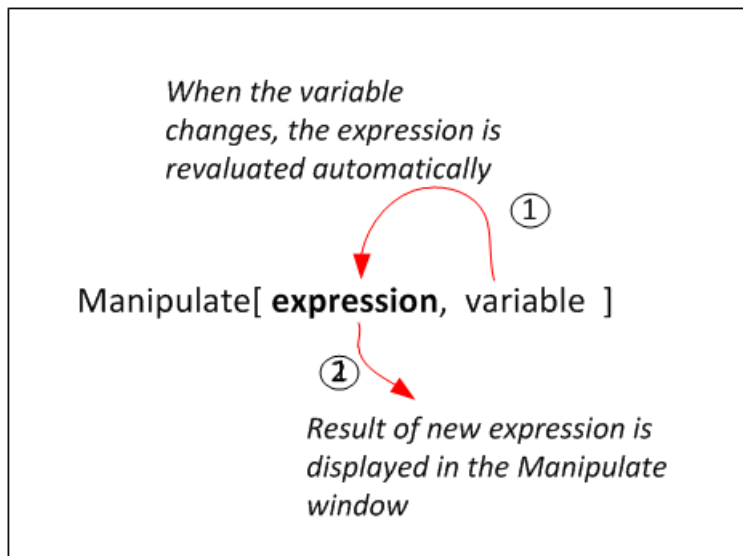
1	Introduction	2
2	Using state machine for Manipulate expression	3
3	Conclusion	9
4	References	9

1 Introduction

This Mathematica notebook with the examples shown below is here `event_programming.nb`

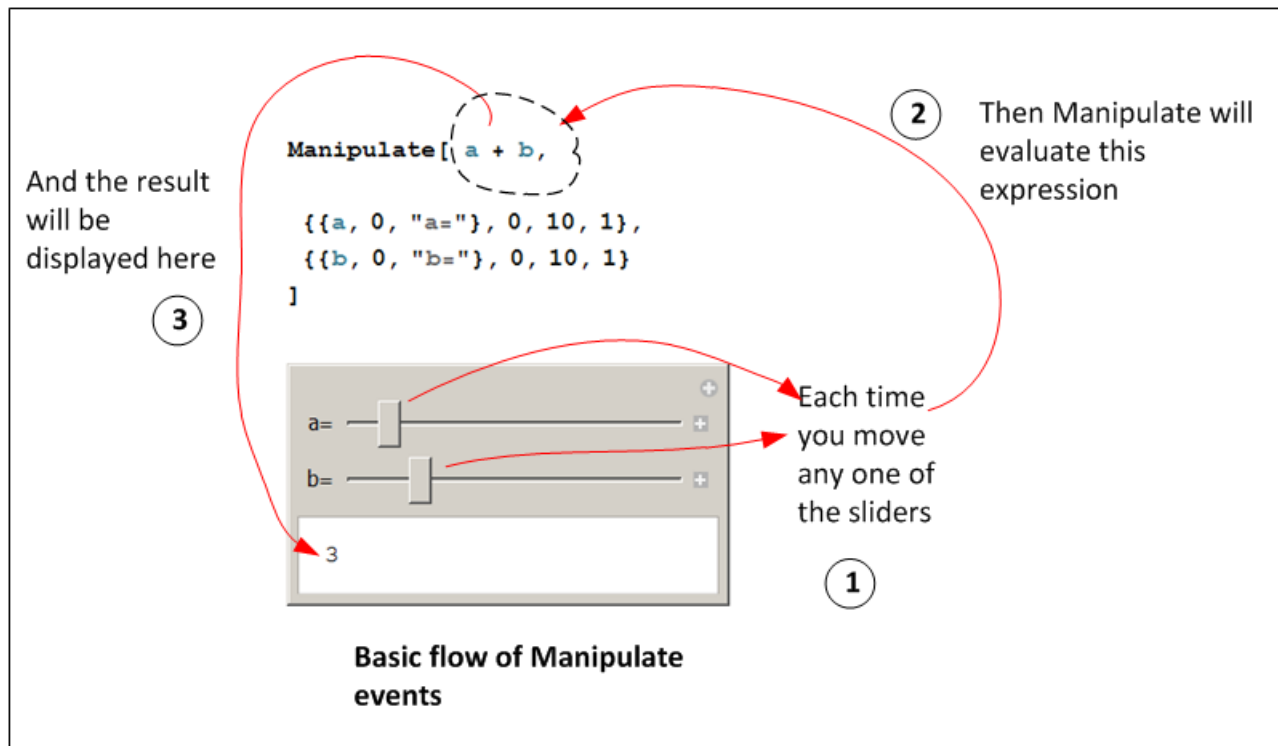
Mathematica's `Manipulate[]` function, introduced in version 6 is a powerful function that makes it easier to develop a user interface (UI) application.

`Manipulate` basic logic flow is simple: It evaluates its expression and display the result. It does this each and every time any one of its control variables changes its value.



The `Manipulate` expression can be as complicated as we make it. It is in general a function of the control variables. Therefore each time any control variable changes value by the use of a slider or other UI components, the effect of this change will immediately reflect in the display of `Manipulate`. `Manipulate` has evaluated the expression using the new value of the variable and updated the display automatically.

The following diagram helps illustrate the above logic flow



But because the evaluation of the expression occurs when any one of the control variable changes, this might not be a very efficient use of resources.

Instead, we would like to know which specific variable changed in order to customise which part of the expression to evaluate. In the following section, we show a model to use to detect which control variable changed and to only perform an action based on that specific change.

2 Using state machine for Manipulate expression

By running the Manipulate expression logic as a state machine which detects what specific variable changed and update an expression based on that event only we can make better use of resources.

The event will be the change of a specific control variable. The state of the machine will be stored in a Manipulate control of type None.

The state of the machine will consist of the event name and any other state information needed. The state will be stored in Manipulate control variable(s) of type None.

This setup is similar to how UI programming is done in traditional setting, where a specific event triggers a callback associated with the event.

To simulate event driven UI, an inner dynamic is added, with a Refresh, inside the Manipulate expression with its own TrackedSymbols.

Each control variable will have its own Dynamic with a TrackedSymbols for its own variable. Inside this Dynamic, the event control variable is set to indicate which control variable has just triggered.

Since an inner Dynamic can have its own TrackedSymbols, then we have effectively moved the task of detecting the change of the control variables from Manipulate down to the inner Dynamics.

The event control variable of type None will be used by the state machine to check which event has

just occurred and then perform an action based on the event.

After the action is completed, the event is reset to special reset value.

The above logic repeats again each time a control variable changes value. Refresh is needed to be used inside the inner `Dynamic[]` and within the `Row[]` construct as will be shown below.

The above method is illustrated with examples, starting with a very simple annotated example, showing how to program a `Manipulate` which displays the name of the variable which was last changed using a slider.

In these examples, a string is used to indicate the event that occurred, but this can be changed, and other type of values can be used. But this is not important for the purpose of illustrating the logic of the overall method.

```

In[16]:= Manipulate [
  Row [{
    Dynamic [Refresh [
      If[Not [event == "reset "],
        {
          result = dispatch [event];
          event = "reset "
        }
      ];
    result ,
    TrackedSymbols -> {event}]]],
  Dynamic [Refresh [event = "a_event "; "", TrackedSymbols -> {a}]],
  Dynamic [Refresh [event = "b_event "; "", TrackedSymbols -> {b}]]],
  {{a, 0, "a"}, 0, 100, 1},
  {{b, 0, "b"}, 0, 30, 1},
  {{event, "reset "}, ControlType -> None },
  TrackedSymbols -> {None },
  Initialization ->
  (
    dispatch [event_string] := Module [{},
      Text @Row [{"event : ", Style [event, Bold ], ". Time is ", Date []}]
    ]
  )
]

```

After dispatch,
the event is
put back to
original value (4)

Notice: we only check
when event is
something other than
the rest event (3)

When event variable
changes, this Dynamic
detects the change, and
dispatches to process the
event (2)

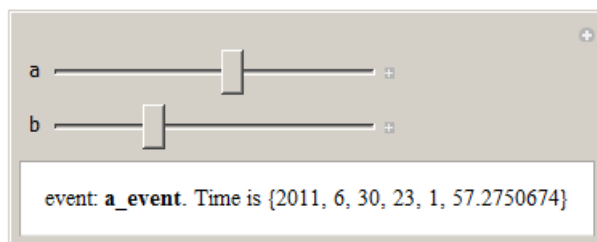
And the result
of the dispatch
is displayed (5)

When you
change 'a' this
Dynamic sets
the event to
'a_event' (1)

This is the dispatch function. It is similar to the callback function in other programming environments. In here, we can switch on the event, and process differently depending on which event it was.

The result returned by the dispatch function will be displayed in the Manipulate window

When moving the 'a' slider in the above example, the following message will be displayed on the Manipulate window



And a similar display results for 'b' when it changes value.

To illustrate the benefit of this approach, we will now use it to show how to numerically solve an ODE. We will have one control variable for the initial condition, where each time the initial condition variable is changed, NDSolve have to be called to obtain a new numerical solution. In addition, we will have a second variable to indicate which color to use to plot the solution with.

Clearly, there is no need to call NDSolve when the color variable is changed as the initial condition did not change.

Using event driven UI programming we are now able to do this. The following digram shows the solution to the above. Notice that the solution is saved in a Manipulate control variable as well as the event name.

```
In[17]:= Manipulate [
```

```
Row [{  
  Dynamic [Refresh [
```

```
    If[Not [event == "reset "],  
      {  
        result = updatePlot [solution , color ],  
        event = "reset "  
      }  
    ];  
    result , TrackedSymbols -> {event } ]],
```

updatePlot is called to make new plot whenever the color or new solution is generated

When initial conditions changes, a new solution is generated and saved in the state variable

```
Dynamic [Refresh [event = "initial_condition "  
  solution = solveODE [initialCondition ];  
  "", TrackedSymbols -> {initialCondition }]],
```

```
Dynamic [Refresh [event = "color "; "", TrackedSymbols -> {color }]]],
```

When the color variable changes, no need to make new solution, only the event is changed

```
{ {initialCondition , 0, "y[0]=", 0, 100, 1},  
Control [ { {color , Blue , "select color "}, {Red -> "Red ", Blue -> "Blue "},  
ControlType -> PopupMenu } ],
```

```
{ {event , "reset "}, ControlType -> None },  
{ {solution , 0}, ControlType -> None },  
{ {result , 0}, ControlType -> None },
```

These variables represent the state

This function generate a solution, and return it back where is saved for later use

```
TrackedSymbols -> {None },
```

```
Initialization ->
```

```
(  
  solveODE [initialCondition_] := Module [{t, y},  
    y /. First @NDSolve [{y'[t] == t Cos [t]^2 , y[0] == initialCondition}, y, {t, 0, 40}]]];
```

This function just plots the solution

```
updatePlot [solution_ , color_] := Module [{t},  
  Plot[Evaluate [solution[t]], {t, 0, 40}, PlotRange -> {{0, 40}, {0, 100}},  
  ImageSize -> 300, PlotStyle -> color]  
]
```

Now the above solution is compared to the solution without using event driven model as is normally done.

This Manipulate expression is evaluated each time any one of the control variables changes

```

In[28]:= Manipulate [
  Module [{y, t, solution },
    solution =
    y /. First @NDSolve [{y'[t] == t Cos [t]^2 , y[0] == initialCondition }, y, {t, 0, 40}];
    Plot[Evaluate [solution [t]], {t, 0, 40}, PlotRange -> {{0, 40}, {0, 100}},
      ImageSize -> 300 , PlotStyle -> color ]
  ],
  {{initialCondition , 0, "y[0]="}, 0, 100, 1},
  Control [{ {color , Blue , "select color "}, {Red -> "Red " , Blue -> "Blue "},
    ControlType -> PopupMenu } ]
]

```

When any one of these changes, the whole expression is evaluated

Out[28]=

We notice that the solution above is much simpler, however it is not efficient on resources. The whole expression is evaluated whenever any one of the variables changes value. In other words, NDSolve is called each time the color variable has changed, which is not required.

3 Conclusion

Mathematica Manipulate can be used to build UI very quickly. Using event driven model can improve the efficiency of using Manipulate by giving the user more control of what action to do based on which variable changes.

Using event driven UI is much more efficient on resources, but it requires more logic to be added to the Manipulate expression.

Using event driven UI programming makes using Manipulate more similar to the event/callback model used by other UI systems where the event is viewed as the change of a specific control variable, and the callback function is the inner Dynamic which detects the change in that specific variable.

4 References

1. Mathematica documentation on Manipulate and Dynamics
2. Useful discussions with John Fultz (Wolfram research) on the subject. Internet newsgroup posting