WOLFRAM TECHNOLOGY CONFERENCE

Creating Accessible Dynamic Content

Kyle Keane

Wolfram Research

Our everyday workflows are increasingly becoming solely reliant on automation and computation. To ensure equal opportunity for all people, innovations in the accessibility of these technologies are becoming increasingly necessary. In this talk we will discover that within Mathematica's core language there exists a robust and complete set of tools for realizing groundbreaking accessibility. When this set of tools is combined with the real-time dynamic computation and painless deployment available with CDF player, the necessary revolution in technological accessibility becomes a reality.

Introduction

In this workshop we will walk through the creation of accessible dynamic content.

We will mainly spend our time looking at modular components and scable solutions for creating custom UI with minimal reliance on esoteric functionality.

This will not be an extensive course on accessibility, it is rather a demonstration of how to develop building blocks that can be then reused without the need for extensive programming.

There are five major types of disabilities which are primarily considered when creating accessible computer content.

- No Vision (NV)
- Low Vision (LV)
- Deaf (D)
- Motoric (M)
- Cognitive (C)

Focus Indication

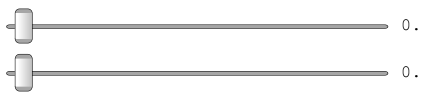
Since there are no established practices for programming accessibility into a *Mathematica* code, we need to think about how to approach the problem ***pragmatically***.

WARNING: Adding Focus Indication to a Slider when Clicked

There is a simple way to do this, but there will be complications down the road.

```
In[1]:= Grid[{
           {Slider[Dynamic[x], {0, 1}, Background → Dynamic[ControlActive[Black, White]]], Dynamic[x]},
           {Slider[Dynamic[y], {0, 1}, Background → Dynamic[ControlActive[Black, White]]], Dynamic[y]}
          }]
```

Out[1]=

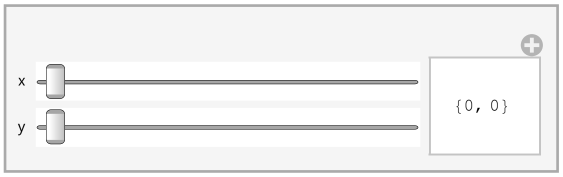


```
0.
0.
```

Unfortunately, this doesn't always behave as expected

```
In[2]:= Manipulate[{x, y},
         {x, 0, Slider[Dynamic[x], {0, 1}, Background → Dynamic[ControlActive[Black, White]]] &},
         {y, 0, Slider[Dynamic[y], {0, 1}, Background → Dynamic[ControlActive[Black, White]]] &}
        ]
```

Out[2]=



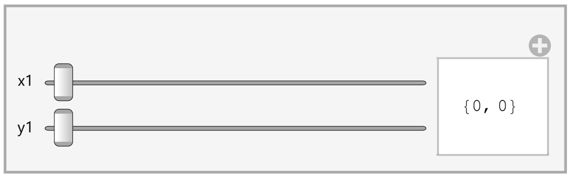
PRIMER: Using Customized Sliders in a Manipulate

Manipulate comes with controls built in, but the built in focus indication may not be enough for a low-vision user and is only displayed when a control is active.

Standard Controls

```
In[3]:= Manipulate[{x1, y1},
         {x1, 0, 1, Slider},
         {y1, 0, 1, Slider}
        ]
```

Out[3]=

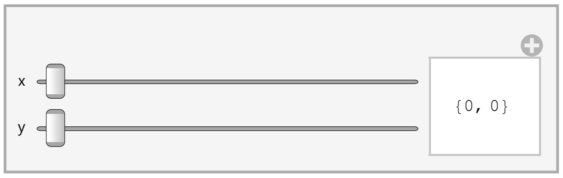


Custom Controls

We can also use our own controls to do the same thing, this will allow us to modify the appearance and function of a control object.

```
In[4]:= Manipulate[{x, y},
         {x, 0, Slider[Dynamic[x], {0, 1}] &},
         {y, 0, Slider[Dynamic[y], {0, 1}] &}
        ]
```

Out[4]=



Defining Custom Control Objects

Adding Focus Indication to a Slider when Clicked {LV, C, M}

In order to enable new functionality we will be utilizing EventHandler[expr,{“event:>action”}]
We will also use Module[] so that the variables will be treated locally.

```
In[5]:= clickedSlider[x_, {xmin_, xmax_}] :=
         Module[
          {insideQ},
          EventHandler[
           Slider[x, {xmin, xmax}, Background → Dynamic[If[insideQ, Black, White]]]
            ,
           {"MouseDown" ⧴ (insideQ = True), "MouseUp" ⧴ (insideQ = False)},
           PassEventsDown → True
          ]
         ]
```

Now, we can simply use our new slider

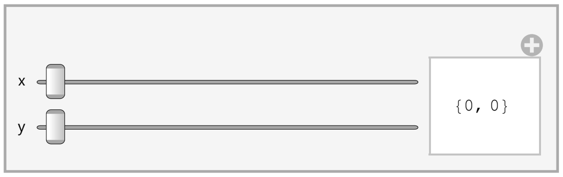
```
In[6]:= {clickedSlider[Dynamic[x], {0, 1}], Dynamic[x]}
```

Out[6]=

, 0.

and we can include it in any location we might want a slider, e.g. Manipulate[]

```
In[7]:= Manipulate[{x, y},
         {x, 0, clickedSlider[Dynamic[x], {0, 1}] &},
         {y, 0, clickedSlider[Dynamic[y], {0, 1}] &}
        ]
```

Out[7]=


{0, 0}

Adding Focus Indication to a Slider when Moused Over {LV, C, M}

We can also add focus indication that shows which control the mouse will affect.

Here is our new slider

```
In[8]:= mouseoverSlider[x_, {xmin_, xmax_}] :=
         Module[
          {insideQ},
          EventHandler[
           Slider[x, {xmin, xmax}, Background → Dynamic[If[insideQ, Black, White]]]
            ,
           {"MouseEntered" ⧴ (insideQ = True), "MouseExited" ⧴ (insideQ = False)}
          ]]
```

We can use our new slider in the same way as before

```
In[9]:= {mouseoverSlider[Dynamic[x], {0, 1}], Dynamic[x]}
```

Out[9]=

, 0.

Adding Sonic Indication when Moused Over {NV, LV, C, M}

These are our spoken results for easy modification

```
In[10]:= in[x_String] := StringJoin["in ", x]
         out[x_String] := StringJoin["out ", x]
```

Here is our new slider, we simply add tasks to our EventHandler[]

```
In[12]:= mousespeakSlider[x_, {xmin_, xmax_}, name_String] :=
          Module[
           {insideQ},
           EventHandler[
            Slider[x, {xmin, xmax}, Background → Dynamic[If[insideQ, Black, White]]]
            ,
            {"MouseEntered" :→ (insideQ = True; Speak[in[name]]), "MouseExited" :→ (insideQ = False;
                Speak[out[name]])}
           ]]
```

We can use it just like before

```
In[13]:= {mousespeakSlider[Dynamic[x], {0, 1}, "x"], Dynamic[x]}
```

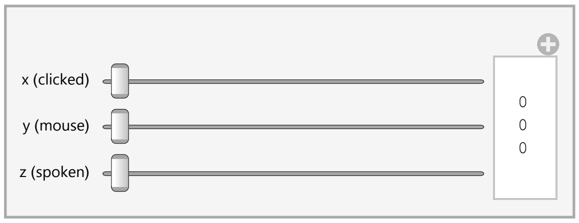
Out[13]=

, 0.}

SUMMARY: All Three Inside a Manipulate

```
In[14]:= Manipulate[Column[{x, y, z}],

          {{x, 0, "x (clicked)"}, clickedSlider[Dynamic[x], {0, 1}] &},
          {{y, 0, "y (mouse)"}, mouseoverSlider[Dynamic[y], {0, 1}] &},
          {{z, 0, "z (spoken)"}, mousespeakSlider[Dynamic[z], {0, 1}, "z"] &}

         ]
```

Out[14]=



Mouse Driven UI

Here are two methods to acquire and use the mouse position inside an object, the second is better for working with Sound.
Optimally, the strengths of each method can be used in tandem for appropriate tasks.

Method 1 (better for heavy computations)

Acquiring the Mouse Position

This is the point where most people can easily program in *Mathematica*

```
In[15]:= mousepos[] :=
          Annotation[
           Graphics[Rectangle[], ImageSize → 100],
           Dynamic[MousePosition["GraphicsScaled"]],
           "Mouse"]
```

Again, this becomes just another object

```
In[16]:= mousepos[]
         Dynamic[MouseAnnotation[]]
```

Out[16]=

```
Out[17]= Null
```

Acquiring the Entry Time

Let’s pre-define our events

```
In[18]:= indicator =
           {"MouseEntered" :→ {Speak["entered"], insideQ = True},
            "MouseExited" :→ {Speak["leaving"], insideQ = False}};
```

Here we define our object

```
In[19]:= entered[] :=
          EventHandler[
           Graphics[{Dynamic[FaceForm[If[insideQ, Green, Red, Red]]], Rectangle[]}, ImageSize → 100],
           indicator]
```

Now, we can use it like any other object

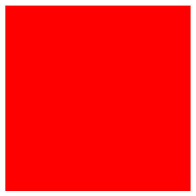
```
In[20]:= entered[]
```

Out[20]=

Using this Information: An example

This plays a Midi sound that has volume proportional to the distance of the mouse cursor from the lower lefthand corner

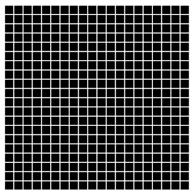
In[21]:=
```
indicator3 =
  {"MouseEntered" :→ {Speak["entered"], insideQ3 = True},
   "MouseExited" :→ {Speak["leaving"], insideQ3 = False}};

Column[{
  Dynamic[
   If[insideQ3,
    {mouse = MousePosition["GraphicsScaled"],
     dist = EuclideanDistance[mouse, {0, 0}]}, "Outside", initialized]
  ],
  EventHandler[
   Dynamic[Graphics[
     Flatten[
      {If[insideQ3, FaceForm[Green], FaceForm[Red], FaceForm[Red]],
       Rectangle[],
       If[insideQ3,
        {Arrowheads[Large], Thick, White, Arrow[{{0, 0}, mouse}]}]
      }
     ], ImageSize → 100]]
   ,
   indicator3],
  Dynamic[
   If[insideQ3,
    EmitSound[Sound@SoundNote[2, 10, "Flute", SoundVolume → dist]],
    EmitSound[Sound@SoundNote[SoundVolume → 0]]
   ], UpdateInterval → 1
  ]
 }]
```

Out[22]=
```
Outside
```

```
Null
```

This shows how we can break a graphic into sections and pass location information to an arbitrary function

```
In[38]:= indicator2 =
           {"MouseEntered" :→ {Speak["entered"], insideQ2 = True},
            "MouseExited" :→ {Speak["leaving"], insideQ2 = False}};

         grid[{xmin_, ymin_}, {xmax_, ymax_}, xdiv_, ydiv_, size_] :=
          EventHandler[Graphics[
            Flatten[
             Table[{EdgeForm[White],
               Annotation[
                Rectangle[{i 1/xdiv, j 1/ydiv}, {(i + 1) 1/xdiv, (j + 1) 1/ydiv}],
                {i, j}
                (*{{i,j},pitch=IntegerPart[-5+10Sqrt[i^2+j^2]/Sqrt[((xdiv-1)xmax)^2+((ydiv-1)ymax)^2]],EmitSound[Sound@SoundNote[pitch,1,"Flute"]]}*),
                "Mouse"
               ]
              }, {i, xmin, (xdiv - 1) xmax}, {j, ymin, (ydiv - 1) ymax}
             ]
            ], ImageSize → size],
           indicator2]

         empty[{f__}] := f

         Column[{grid[{0, 0}, {1, 1}, 20, 20, 100],

           ExpressionCell[Grid[{
              {"Mouse Annotation", Dynamic[MouseAnnotation[]]},
              {"Passed Values", Dynamic[
                If[insideQ2,
                 Which[
                  Evaluate[
                   empty[
                    Flatten[
                     Table[{MouseAnnotation[] == {i, j}, f[i, j]}, {i, 0, 19}, {j, 0, 19}], 2]
                   ]]]]]}}]]}]
```

Out[41]=



```
Mouse Annotation  Null
  Passed Values   Null
```

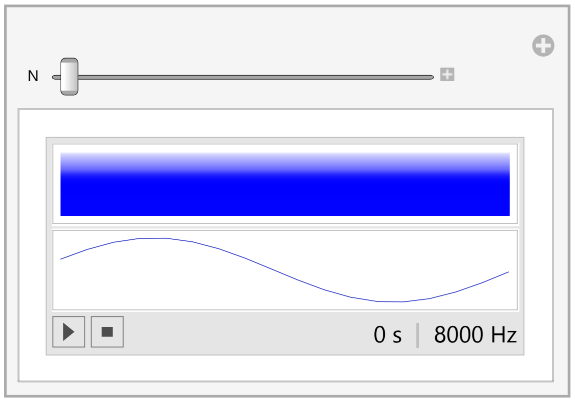
Method 2 (better for including sound)

Creating a sound object

We will work with beeps, which require us to use a certain number of wavelengths played at a certain interval

```
In[27]:= λ[f_, δt_] :=
          Play[Sin[f 2 π t], {t, 0, δt}]

         Manipulate[λ[440, N/440], {N, 1, 100}, ControlPlacement → Top]
```

Out[28]=



Creating a soniified object

To create sound that will only play while the mouse is inside a specific object, we will use RunScheduledTask[]

```
In[29]:= beepingSquare[BeepsPerSecond_, RelativeLengthPerBeep_] :=
          Graphics[{
            EventHandler[
             Rectangle[],
             {"MouseEntered" :>
               {
                test = True,
                n = 0,
                Quiet[RemoveScheduledTask[task]],
                task =
                 RunScheduledTask[
                  {
                   mouse = MousePosition["GraphicsScaled"],
                   dist = EuclideanDistance[mouse, {0, 0}],
                   If[test, Quiet@EmitSound@Sound@λ[300 dist + 250, RelativeLengthPerBeep/BeepsPerSecond]]
                  }
                  , {1/BeepsPerSecond, 50}]
               },
              "MouseExited" :> {test = False, Quiet[RemoveScheduledTask[task]]}
             }]
           }, ImageSize → 100]
```

Using our object

```
In[30]:= beepingSquare[5, .8]
```

Out[30]=

Keyboard Navigation

In order to create custom hotkeys we will utilize an InputField[] and an EventHandler[].

These hotkeys can be used to trigger arbitrary code and can thus be used to control UI elements as well as provide real-time feedback about live calculations.

Using an InputField as a Homebase

Let’s create a homebase that is activated by clicking inside the InputField[]

```
In[31]:= homebase[activated_String, hotkeys__] :=
          Row[{
            EventHandler[
             EventHandler[
              InputField["click here to activate", FieldSize → {15, 2}],
              {"MouseClicked" :→ Speak[activated]},
              PassEventsDown → True],
             hotkeys,
             PassEventsDown → False]
           }]
```

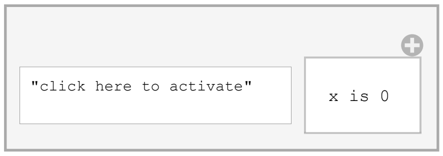
Now we can list some HotKey Commands

```
In[32]:= hotkeys =
           {{"KeyDown", "x"} :→ {If[x < 10, x++], Speak[ToString[StringForm["x is ``", x]]]},
            {"KeyDown", "y"} :→ {y++}};
```

Placing our homebase inside a Manipulate

```
In[33]:= Manipulate[
          StringForm["x is ``", x],
          {{x, 0, "x"}, None},
          ExpressionCell[homebase["success", hotkeys]],
          SaveDefinitions → True
         ]
```

Out[33]=

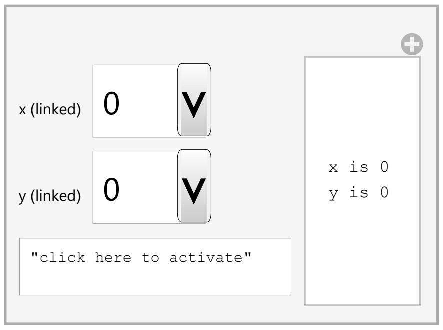


```
"click here to activate"      x is 0
```

In this way controls are also linked up as they should be

```
In[34]:= Manipulate[Column[{StringForm["x is ``", x], StringForm["y is ``", y]}],

          {{x, 0, "x (linked)"}, PopupMenu[Dynamic[x], Range[0, 10]] &},
          {{y, 0, "y (linked)"}, PopupMenu[Dynamic[y], Range[0, 99]] &},

          ExpressionCell[homebase["success", hotkeys]]

          , SaveDefinitions → True
         ]
```

Out[34]=



```
x is 0
y is 0
```