

Lecture 4: Interactive Graphics

Written by Ben Yan, from materials by Jerry Cain

Event Listener Functions 🎧

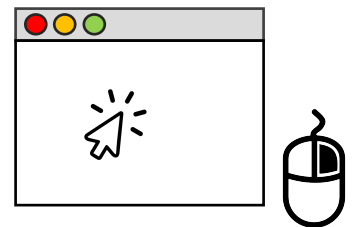
Graphical applications usually make it possible for a user to interface with and even direct a program's actions by using an input device such as a mouse. Programs that support this are considered *interactive*.

As terminology, user actions (e.g., *mouse clicks*, *keystrokes*) are called **events**, and programs that respond to these events are called **event-driven**. For this lecture, we'll write functions that aren't called immediately, but rather sit pretty and **listen** in the background for a particular event type. **Only when that event occurs, the listener or callback function is called.** The general JavaScript template for this is:

```
let eventAction = function(e) {
    code that executes each time this event occurs
}
gw.addEventListener(event, eventAction);
```

For example, if **event** is **"click"**, we'd respond to mouse clicks. More broadly, the **event** types are:

"click"	The user clicks the mouse in the window.
"dblclick"	The user double-clicks the mouse.
"mousedown"	The user presses the mouse button.
"mouseup"	The user releases the mouse button.
"mousemove"	The user glides the mouse with the button up.
"drag"	The user drags the mouse with the button down.



Certain user actions can spark more than one mouse event, e.g., clicking the mouse generates a **"mousedown"** event, a **"mouseup"** event, and a **"click"** event, in that order. Furthermore, each callback function accepts a parameter **e**, which is a data structure with information on the mouse event.

Also note that **function name(parameters)** and **let name = function(parameters)** are equivalent syntax; the latter mainly emphasizes that all functions are assigned to a variable with a name.

Function Closure and Cookie Clicker 🍪

The narrative strength of declaring a function this way—using the assignment operator—occurs when **you declare one function as a local variable inside another function**. That way, the *inner* function can access the *outer* function's local variables, an essential feature for interactive programs as we'll see ☺.

Below is a short program that increments a counter variable, **numCookies**, each time a click occurs.

```
function CookieClicker() { // my own example! not in lecture slides :D
    let numCookies = 0;
    let clickAction = function() {
        numCookies++; // nested function can access this outer variable!
    }
    gw.addEventListener("click", clickAction);
}
```

Draw Dots 🎨

Consider the following interactive program, which draws a dot on the graphics window whenever the user clicks the mouse. Each dot is a filled `GOval` circle centered at the coordinates of the mouse click.

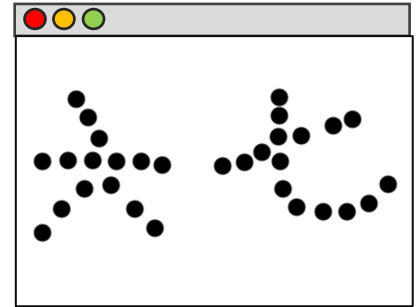
```
/* Constants (in pixels) */
const GWINDOW_WIDTH = 500;
const GWINDOW_HEIGHT = 300;
const DOT_SIZE = 6; // diameter in pixels

function DrawDots() {
    let gw = GWindow(GWINDOW_WIDTH, GWINDOW_HEIGHT);

    let clickAction = function(e) {
        let dotX = e.getX() - DOT_SIZE / 2;
        let dotY = e.getY() - DOT_SIZE / 2;

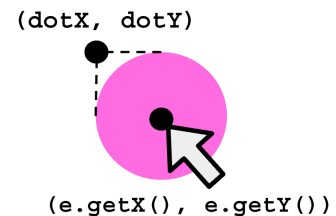
        let dot = GOval(dotX, dotY, DOT_SIZE, DOT_SIZE);
        dot.setFilled(true);
        gw.add(dot);
    };

    gw.addEventListener("click", clickAction);
}
```



The key to understanding this program is the `clickAction` callback function, which is only called into action when the mouse is clicked on-screen. To that end, consider the first 3 lines of its implementation.

```
let clickAction = function(e) {
    let dotX = e.getX() - DOT_SIZE / 2;
    let dotY = e.getY() - DOT_SIZE / 2;
```



The input parameter is mouse event object `e`, which supports two methods you'll want to hammer into memory (**LIKE PLS REMEMBER THEM** 😊). Those are `getX()` and `getY()`, which return the `x` and `y` coordinates of where the click occurred in the `GWindow`'s coordinate system. From there, `e.getX()` and `e.getY()` are offset by the dot's radius, as a `GOval` is positioned by its top-left coordinate.

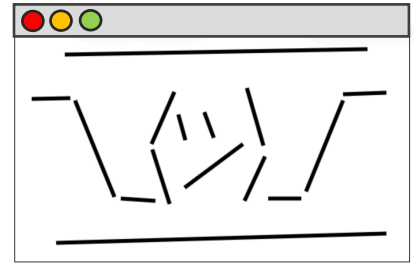
The rest of its implementation creates the filled circle and adds it to the screen, centered at the click.

```
let dot = GOval(dotX, dotY, DOT_SIZE, DOT_SIZE);
dot.setFilled(true);
gw.add(dot);
```

Crucially, note that `clickAction` has access to the `gw` variable in the outer function `DrawDots`, as `gw` is included in `clickAction`'s *closure* by nesting. A function's **closure** is defined as the combination of its own implementation and all variables available to it, and is a big, big reason this program works 😊.

Draw Lines

Consider our second interactive graphics program, **DrawLines**, which enables to draw lines with the mouse—by pressing down to create a starting point, and dragging to extend the endpoint as far as we wish.



```
/* Constants */
const GWINDOW_WIDTH = 500;
const GWINDOW_HEIGHT = 300;

function DrawLines() {
  let gw = GWindow(GWINDOW_WIDTH, GWINDOW_HEIGHT);
  let line = null;

  let mousedownAction = function(e) {
    line = GLine(e.getX(), e.getY(), e.getX(), e.getY());
    gw.add(line);
  };
  let dragAction = function(e) {
    line.setEndPoint(e.getX(), e.getY());
  };

  gw.addEventListener("mousedown", mousedownAction);
  gw.addEventListener("drag", dragAction);
}
```

First, note the *closure* of the two listener functions, which can both access the window **gw**, and a shared **line** object. These inner functions then work collaboratively to initialize and extend the line on-screen.

```
let gw = GWindow(GWINDOW_WIDTH, GWINDOW_HEIGHT);
let line = null;
let mousedownAction = function(e) { ... }; // called upon "mousedown"
let dragAction = function(e) { ... };      // called upon "drag"
```

The implementation of **mousedownAction** sets the shared variable **line** to a new zero-length line. That is, its start and end points are the same, and both are the coordinates of the mousedown event **e**.

```
let mousedownAction = function(e) {
  line = GLine(e.getX(), e.getY(),
               e.getX(), e.getY());
  gw.add(line);
};
```

The implementation of **dragAction** then takes the **line** object initialized by the “mousedown” handler above, and alters its endpoint to where the mouse is currently located. Note the starting point remains the same as the mousedown, and that **GLine** objects support methods **setStartPoint(x, y)** and **setEndPoint(x, y)**.

```
let dragAction = function(e) {
  line.setEndPoint(e.getX(), e.getY());
};
```

