CS 106 Winter 2016
Craig S. Kaplan

Module 03
Graphical User Interfaces

Topics

- **The model-view-controller paradigm**
- **Direct manipulation**
- **User interface toolkits**
- **Building interfaces with ControlP5**

Readings

- **None**

Introduction

A program's user interface comprises all the mechanisms by which we tell the program what we want to it to do while it's running, and all the mechanisms the program uses to give us feedback on what it's doing.

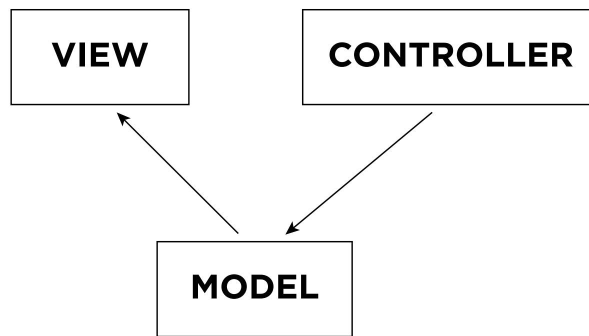
You've learned quite a bit about how to coax rich visual output from Processing, which is half of the equation. But we've seen very little about how to talk back to our sketches. This is an important step in our ability to create fancier software—there's only so much intuitive control you can get out of the keyboard together with `mouseX` and `mouseY`.

This is an enormous, deep topic. We teach two undergraduate CS courses on user interfaces, and Human-Computer Interaction (HCI) is a very active area of CS research. I can't hope to cover this area in any significant detail. I won't say much about effective user interface design. My main goal is to give you a look at user interfaces from a programmer's point of view, so that you know what's possible, so that you can add more control to your sketches, and so that

you can gain better understanding of the interfaces in your world.

The Model-View-Controller paradigm

There are many ways to think about the structure of an interactive program. The most established paradigm is called “model-view-controller”, or MVC. A program that uses an MVC architecture can be decomposed into three main components:



- The **model** is the underlying collection of information that the program is *about*. In a text editor, the model is a string of characters. In an image editor, it’s the image, and so on.
- The **view** is the means by which the program shows us the current state of the model.
- The **controller** is the set of controls through which the user operates the program (i.e., manipulates the model).

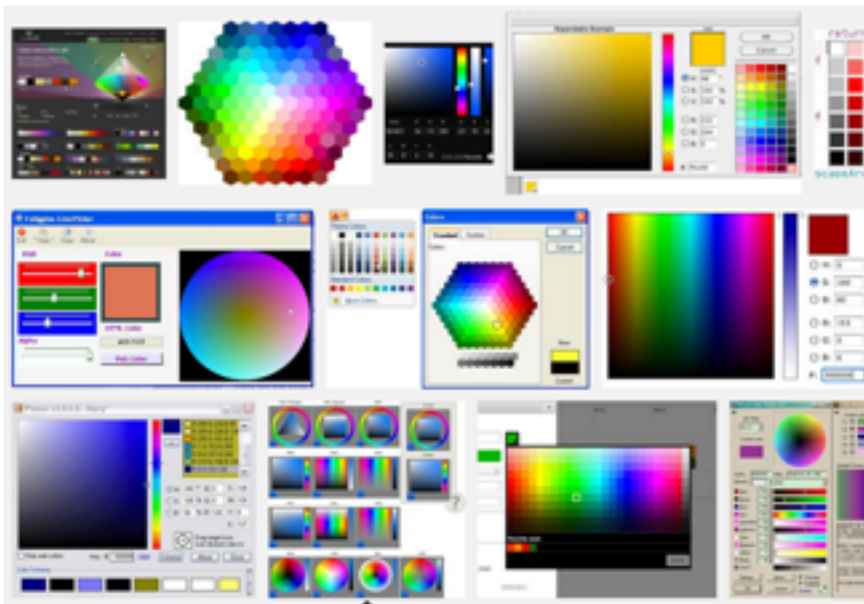
We could think of the controller as being physical devices like the keyboard and mouse, but it’s more useful to think of it as the *code* that processes information from those devices.

The arrows in the diagram are significant: the controller makes changes to the model, and the model notifies the view that it should update itself. (We could augment the diagram with a **user**, who perceives the view and operates the controller, thereby creating a closed loop; but we don’t need to write code to control the user!)

Most frameworks for writing apps are built around an explicit breakdown of functionality into views and controllers, and leave the model up to the

programmer. Mostly, though, it’s a *paradigm*: a way of thinking about programming.

MVC is useful because it *decouples* the components of your program—it forces you to think about partitioning your code, which gives you more flexibility. If the model knows too much about the view, then it’s hard to add in new views, or have multiple views operating at the same time. Similarly, if the model knows about only one controller, it’s hard to adapt to new input devices or on-screen widgets. In an ideal world, the model doesn’t need to know anything at all about the view or controller.



With colour selectors, the model is an RGB colour, but there are many possible controllers (and views).

In a Processing sketch, the model is the information being stored (mostly likely, the global variables). The view is the sketch window. The controller is the functionality you create to respond to events.

Example sketch: SimplestMVC

```
color the_colour;
```
MODEL

```
void setup()
{
  size( 200, 200 );
}
```

VIEW

```
void draw()
{
  background( the_colour );
}
```

CONTROLLER

```
void mouseMoved()
{
  int b = int( map( mouseX, 0, width, 0, 255 ) );
  the_colour = color( b );
}
```

Direct manipulation

In a *direct manipulation* interface, the view and controller are tightly linked. The view is a concrete visual representation of the model, decorated with user interface "handles". These handles can be manipulated to edit the model continuously and incrementally. We might interpret direct manipulation as a thinner, more immediate physical interaction metaphor: when you want to change your model, you reach out and touch it.

Direct manipulation relies on *hit testing*: checking whether a given mouse position lies within the responsive area of a given handle. You saw a hint of hit testing in CS 105 (checking if a point lies within a circle). If we choose handles with simple shapes, their hit tests will hopefully be simple too.

Example sketch: DirectManipDot

When there are multiple independent handles in the interface, the code will take on a consistent structure:

- We maintain a list (say, an array) of all the handles in the program.
- We also maintain a global variable to remember which handle is currently being manipulated, if any. In an object-oriented interface we might point right to the active handle. Or we might use an index into the array above.
- In the `draw()` function, we walk over the list, asking all the handles to draw themselves.
- In a `mousePressed()` function, we walk over the list, performing a hit test on each handle until we find one that overlaps the mouse location. We must be careful about order when there are overlaps: handles will be drawn back-to-front, but we should check them front-to-back!
- In a `mouseDragged()` function, we update the position of the currently active handle, if there is one.
- In `mouseReleased()`, we make sure to reset the state of the program so that no handle is currently being manipulated.

Example sketch: DirectManipMulti

Example sketch: DirectManipBezier

Hit testing can be tricky for some handles. If we want to move thin lines around, we might define a "halo" around the line that responds to mouse events. If objects have complicated shapes, their hit tests might be complicated too. One way around that is to put a complicated shape inside a simpler one like a box, and respond to events anywhere in that box.

A fancier approach is to paint a secondary image that never gets shown on the screen (using the `PGraphics` class in Processing), in which every handle is drawn with a solid colour that tells us its identity. Then, when we receive a mouse press, we check the colour of the pixel at the mouse location in the off-screen image, and use that to deduce which handle was touched.

Example sketch: Jigsaw

User interface toolkits

There's a characteristic set of interactions that we typically want to carry out when communicating with a program:

- Perform an action (e.g., click an OK button)
- Select from among a set of choices
- Adjust a continuous value within a range
- Enter some text

Clicking an on-screen button is a good way to ask a program to perform a discrete action. You could code that yourself, but the low-level behaviour of a button is actually pretty subtle—it takes a huge amount of programming to get it right. Pick a button in your favourite application, and explore how it responds to unusual interactions!

It's only natural then that programmers like to make use of *toolkits*, standardized sets of reusable, modular, programmable widgets. They greatly reduce programmer effort while simultaneously guaranteeing greater consistency across apps. Users quickly learn the visual language of sets of widgets.

Typical toolkits are AWT and Swing in Java, Qt and Gtk in the open-source world, Apple's UIKit framework, and HTML forms.

ControlP5

User interface toolkits are often very large and complex, and require a moderate amount of knowledge and programming (or the use of auxiliary interface authoring tools). The ControlP5 library is

designed to be a slimmed-down toolkit for use within Processing sketches.

As this point, things get more boring: the library's out there, we just need to read its documentation, look at example code, and learn how to speak its language.

As with the libraries we saw in Module 02, the first step is to install ControlP5. Then we can declare our intention to use it in a sketch:

```
import controlP5.*;
```

Next we need to create a global variable that represents the user interface subsystem, and initialize it in the `setup()` function.

```
ControlP5 ui;

void setup()
{
  size( 500, 500 );
  ui = new ControlP5( this );
}
```

Here's an example where we have no alternative but to use the keyword `this`. The ControlP5 library needs to know about the sketch that it'll be running in.

Now it's easy to add new user interface widgets to a sketch. The `ControlP5` object (`ui` in this case) responds to a bunch of messages with names like `addButton()`, `addKnob()`, and `addSlider()`. These functions will create the appropriate controllers and make them visible in the sketch.

If you want, you can store the result of a function like `addButton()` to a local or global variable, so that you can refer to it later.

```
Button quit = ui.addButton( "Quit" );
```

ControlP5 does some clever behind-the-scenes trickery to ensure that its controllers get drawn on top of your sketch. You don't have to do anything yourself. But you *do* need a `draw()` function, even an empty one, otherwise ControlP5 never told to draw the interface.

Once you assign the controller to a variable (`quit` in this case), you can send it a sequence of messages to ask it to configure itself.

```
quit.setPosition(100,100);
quit.setSize(200,19);
```

ControlP5 also offers a neat trick for setting a bunch of properties at once. The property setting functions all re-return the passed-in controller, to which you can then send more messages.

```
Button quit = ui.addButton( "Quit" )
  .setValue(0)
  .setPosition(100,100)
  .setSize(200,19);
```

Don't be alarmed by the odd formatting here. It's the same mechanism of sending a message to an object, rewritten to emphasize that we're setting a bunch of properties in sequence.

Admittedly, that trick is perhaps *too* neat: it uses non-obvious language features to work. You can ignore the fact that you don't yet know why this works and just use it anyway. Or if it makes you uncomfortable you can use a sequence of independent message calls, as in the example before this one.

The last step in setting up a ControlP5 interface is to add behaviour to the widgets. The toolkit looks for a new hook function called `controlEvent()`, similar to `keyPressed()` or `mouseDragged()`. You write a `controlEvent()` function, which receives notification when the user interacts with any widget. The function is passed information about which widget was the target of the interaction:

```
public void controlEvent(ControlEvent ev) {
  println(ev.getController().getName());
}
```

This event handler only prints the name of the controller that received the event. But of course, you can use the controller's identity or name to decide what to do with the event.

ControlP5 offers another clever hack here: you can also define specific event handler functions based on the name of the widget you created. I won't talk about that in class, but you can experiment with it if you want.

The most reliable way to find out who was the target of the event is to use the `isFrom()` message, which returns a `boolean` that tells you whether the passed-in

controller was indeed the place where the event happened.

```
if( ev.isFrom( quit ) ) {
  println( "Quitting time!" );
  exit();
}
```

Example sketch: SimplestToolkit

Example sketch: ThreeButtons

Example sketch: FancyButtons

Example sketch: BusyBox

Beyond widgets

A toolkit is a great way to get a lot of functionality into the hands of the user quickly and easily. But it's not the necessarily the best approach. Many UI designers believe in the motto “less is more”: no matter how good your interface is, there could always be less of it.

One way to curb the runaway proliferation of menus and toolbars is better user modelling: if you understand what actions the user will perform and in what order, you can streamline the interface to support those actions.

Too many of the web-based interfaces at our university are designed around creating one widget for each piece of data in a database, instead of modelling the typical workflows that users will want.

As computations becomes cheaper, sensing technology gets better, and devices without keyboards gain in popularity, we're also starting to see many tools for more naturalistic interfaces:

- **Gestural interfaces** allow the user to express commands with looser, more fluid motions. The system must work harder to interpret which gestures have been given and what they mean.
- **Multitouch interfaces** have more or less become the norm for mobile devices in the past ten years. These devices can identify and track multiple touch points simultaneously, further enriching the gestural vocabulary that's possible. Processing has a useful library for multitouch interaction, written in part by UW researchers.
- **Voice interfaces** get gradually more popular as speech recognition gets better. (But I'm not sure voice control is useful in the context of Processing sketches.)
- **3D interfaces** such as the WiiMote, Kinect, and Leap Motion, permit the whole body to be used as an interaction device. This is great for games; the full potential of these devices in desktop interaction has probably not been fully realized yet.
- **Eye tracking** is slowly becoming more affordable, and promises to be very useful for interaction.
- **Other sensors**, particularly on mobile devices, can detect interesting new forms of interaction such as tilting and shaking (not to mention geographic location). See also the Myo Armband, by UW startup Thalmic Labs.