Using drawing operations

Although it is advisable to use the built-in components if possible, there are occasions where you may need or wish to create a completely new custom component. This may be to perform some specific drawing tasks or a unique user interface item. JUCE also handles this elegantly.

First, create a new Introjucer project and name it `Chapter02_05`. To perform drawing tasks in a component, you should override the `Component::paint()` function. Change the contents of the `MainComponent.h` file to:

```
#ifndef __MAINCOMPONENT_H__
#define __MAINCOMPONENT_H__

#include "../JuceLibraryCode/JuceHeader.h"

class MainContentComponent :  public Component
{
public:
  MainContentComponent();
  void paint (Graphics& g);
};
#endif
```

Change the contents of the `MainComponent.cpp` file to:

```
#include "MainComponent.h"

MainContentComponent::MainContentComponent()
{
  setSize (200, 200);
}

void MainContentComponent::paint (Graphics& g)
{
  g.fillAll (Colours::cornflowerblue);
}
```

Build and run the application to see the resulting empty window filled with a blue color.

The `paint()` function is called when the component needs to redraw itself. This might be due to the component having been resized (which of course you can try out using the corner resizer), or specific calls to invalidate the display (for example, the component displays visual representation of a value and this is no longer the currently stored value). The `paint()` function is passed a reference to a `Graphics` object. It is this `Graphics` object that you instruct to perform your drawing tasks. The `Graphics::fillAll()` function used in the code above should be self-explanatory: it fills the entire component with the specified color. The `Graphics` object can draw rectangles, ellipses, rounded rectangles, lines (in various styles), curves, text (with numerous shortcuts for fitting or truncating text within particular areas) and images.

The next example illustrates drawing a collection of random rectangles using random colors. Change the `paint()` function in the `MainComponent.cpp` file to:

```
void MainContentComponent::paint (Graphics& g)
{
  Random& r (Random::getSystemRandom());
  g.fillAll (Colours::cornflowerblue);

  for (int i = 0; i < 20; ++i) {
    g.setColour (Colour (r.nextFloat(),
                         r.nextFloat(),
                         r.nextFloat(),
                         r.nextFloat()));

    const int width = r.nextInt (getWidth() / 4);
    const int height = r.nextInt (getHeight() / 4);
    const int left = r.nextInt (getWidth() - width);
    const int top = r.nextInt (getHeight() - height);

    g.fillRect (left, top, width, height);
  }
}
```

This makes use of multiple calls to the JUCE random number generator class `Random`. This is a convenient class that allows the generation of pseudo-random integers and floating-point numbers. You can make your own instance of a `Random` object (which is recommend if your application uses random numbers in multiple threads), but here we simply take a copy of a reference to a global "system" `Random` object (using the `Random::getSystemRandom()` function) and use it multiple times. Here, we fill the component with a blue background and generate 20 rectangles. The color is generated from randomly generated floating point ARGB values. The call to the `Graphics::setColour()` function sets the current drawing color that will be employed by subsequent drawing commands. A randomly generated rectangle is also created by first choosing width and height (each being a maximum value of one-quarter of the parent component's width and height respectively). Then the position of the rectangle is randomly selected; again this is done using the parent component's width and height but this time subtracting the width and height of our random rectangle to ensure its right and bottom edges are not off-screen. As mentioned previously, the `paint()` function is called each time the component needs to be redrawn. This means we will get a completely new set of random rectangles as the component is resized.

Changing the drawing command to `fillEllipse()` rather than `fillRect()` draws a collection of ellipses instead. Lines can be drawn in various ways. Change the `paint()` function as follows:

```
void MainContentComponent::paint (Graphics& g)
{
  Random& r (Random::getSystemRandom());
  g.fillAll (Colours::cornflowerblue);

  const float lineThickness = r.nextFloat() * 5.f + 1.f;
  for (int i = 0; i < 20; ++i) {
    g.setColour (Colour (r.nextFloat(),
                         r.nextFloat(),
                         r.nextFloat(),
                         r.nextFloat()));

    const float startX = r.nextFloat() * getWidth();
    const float startY = r.nextFloat() * getHeight();
    const float endX = r.nextFloat() * getWidth();
    const float endY = r.nextFloat() * getHeight();

    g.drawLine (startX, startY,
                endX, endY,
                lineThickness);
  }
}
```

Here, we choose a random line thickness (between one and six pixels wide) before the `for()` loop and use it for each line. The start and end positions of the lines are also randomly generated. To draw a continuous line there are a number of options, you could:

- **store the last end point of the line and use this as the start point of the next line; or**
- **use a JUCE `Path` object to build a series of line drawing commands and draw the path in one pass.**

The first solution would be something like this:

```
void MainContentComponent::paint (Graphics& g)
{
  Random& r (Random::getSystemRandom());
  g.fillAll (Colours::cornflowerblue);

  const float lineThickness = r.nextFloat() * 5.f + 1.f;

  float x1 = r.nextFloat() * getWidth();
  float y1 = r.nextFloat() * getHeight();
  for (int i = 0; i < 20; ++i) {
    g.setColour (Colour (r.nextFloat(),
                         r.nextFloat(),
                         r.nextFloat(),
                         r.nextFloat()));

    const float x2 = r.nextFloat() * getWidth();
    const float y2 = r.nextFloat() * getHeight();
    g.drawLine (x1, y1, x2, y2, lineThickness);
    x1 = x2;
    y1 = y2;
  }
}
```

The second option is slightly different; in particular, each of the lines that make up

the path must be same color:

```
void MainContentComponent::paint (Graphics& g)
{
  Random& r (Random::getSystemRandom());
  g.fillAll (Colours::cornflowerblue);

  Path path;
  path.startNewSubPath (r.nextFloat() * getWidth(),
                        r.nextFloat() * getHeight());
  for (int i = 0; i < 20; ++i) {
    path.lineTo (r.nextFloat() * getWidth(),
                 r.nextFloat() * getHeight());
  }

  g.setColour (Colour (r.nextFloat(),
                       r.nextFloat(),
                       r.nextFloat(),
                       r.nextFloat()));

  const float lineThickness = r.nextFloat() * 5.f + 1.f;
  g.strokePath (path, PathStrokeType (lineThickness));
}
```

Here the path is created before the `for()` loop and each iteration of the loop adds a line segment to the path. These two approaches to line drawing clearly suit different applications. The path drawing technique is heavily customizable, in particular:

- **The joints at the corners of the line segments can be customized with the `PathStrokeType` class (for example, to make the corners slightly rounded).**
- **The lines need not be straight: they can be Bezier curves.**
- **The path may include other fundamental shapes such as rectangles, ellipses, stars, arrows and so on.**

In addition to these line drawing commands, there are accelerated functions specifically for drawing horizontal and vertical lines (that is, non-diagonal). These are the `Graphics::drawVerticalLine()` and `Graphics::drawHorizontalLine()` functions.

Intercepting mouse activity

To help your component respond to mouse interaction, the `Component` class has six important callback functions that you can override:

- **`mouseEnter()`: Called when the mouse pointer enters the bounds of this component and the mouse buttons are *up*.**
- **`mouseMove()`: Called when the mouse pointer moves within the bounds of this component and the mouse buttons are *up*. A `mouseEnter()` callback will always have been received first.**
- **`mouseDown()`: Called when one or more mouse buttons are pressed while the mouse pointer is over this component. A `mouseEnter()` callback will always have been received first and it is highly likely one or more `mouseMove()` callbacks will have been received too.**
- **`mouseDrag()`: Called when the mouse pointer is moved following a `mouseDown()` callback on this component. The position of the mouse pointer may be outside the bounds of the component.**
- **`mouseUp()`: Called when the mouse button is released following a `mouseDown()`**

callback on this component (the mouse pointer will not necessarily be over this component at this time).

- `mouseExit()`: Called when the mouse pointer leaves the bounds of this component when the mouse buttons are *up* and after a `mouseUp()` callback if the user has clicked on this component (even if the mouse pointer exited the bounds of this component some time ago).

In each of these cases, the callbacks are passed a reference to a `MouseEvent` object that can provide information about the current state of the mouse (where it was at the time of the event, when the event occurred, which modifier keys on the keyboard were down, which mouse buttons were down, and so on). In fact, although these classes and function names refer to the "mouse" this system can handle multi-touch events and the `MouseEvent` object can be ask which "finger" was involved in such cases (for example, on the iOS platform).

To experiment with these callbacks, create a new Introjucer project and name it `Chapter02_06`. Use the following code for this project.

The `MainComponent.h` file declares the class with its various member functions and data:

```
#ifndef __MAINCOMPONENT_H__
#define __MAINCOMPONENT_H__

#include "../JuceLibraryCode/JuceHeader.h"

class MainContentComponent : public Component
{
public:
  MainContentComponent();
  void paint (Graphics& g);

  void mouseEnter (const MouseEvent& event);
  void mouseMove (const MouseEvent& event);
  void mouseDown (const MouseEvent& event);
  void mouseDrag (const MouseEvent& event);
  void mouseUp (const MouseEvent& event);
  void mouseExit (const MouseEvent& event);

  void handleMouse (const MouseEvent& event);

private:
  String text;
  int x, y;
};
#endif
```

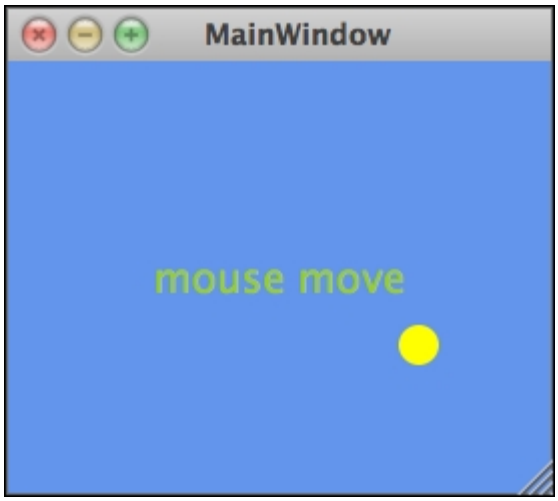
The `MainComponent.cpp` file should contain the following code. First, add the constructor and the `paint()` function. The `paint()` function draws a yellow circle at the mouse position and some text showing the current phase of the mouse interaction:

```
#include "MainComponent.h"

MainContentComponent::MainContentComponent()
: x (0), y (0)
{
  setSize (200, 200);
}

void MainContentComponent::paint (Graphics& g)
{
  g.fillAll (Colours::cornflowerblue);
  g.setColour (Colours::yellowgreen);
  g.setFont (Font (24));
  g.drawText (text, 0, 0, getWidth(), getHeight(),
              Justification::centred, false);
  g.setColour (Colours::yellow);
  const float radius = 10.f;
  g.fillEllipse (x - radius, y - radius,
                 radius * 2.f, radius * 2.f);
}
```

Then add the mouse event callbacks and our `handleMouse()` function described as follows. We store the coordinates of the mouse callbacks with reference to our component and store a `String` object based on the type of callback (mouse down, up, move, and so on). Because the storage of the coordinates is the same in each case, we use the `handleMouse()` function, which stores the coordinates from the `MouseEvent` object in our class member variables `x` and `y`, and pass this `MouseEvent` object from the callbacks. To ensure that the component redraws itself, we must call the `Component::repaint()` function.

```
void MainContentComponent::mouseEnter (const MouseEvent& event)
{
  text = "mouse enter";
  handleMouse (event);
}

void MainContentComponent::mouseMove (const MouseEvent& event)
{
  text = "mouse move";
  handleMouse (event);
}

void MainContentComponent::mouseDown (const MouseEvent& event)
{
  text = "mouse down";
  handleMouse (event);
}

void MainContentComponent::mouseDrag (const MouseEvent& event)
{
  text = "mouse drag";
  handleMouse (event);
}

void MainContentComponent::mouseUp (const MouseEvent& event)
{
  text = "mouse up";
  handleMouse (event);
}

void MainContentComponent::mouseExit (const MouseEvent& event)
{
  text = "mouse exit";
  handleMouse (event);
}

void MainContentComponent::handleMouse (const MouseEvent& event)
{
  x = event.x;
  y = event.y;
  repaint();
}
```

As shown in the following screenshot, the result is a yellow circle that sits under our mouse pointer and a text message in the center of the window that gives feedback as to the type of mouse event most recently received:


MainWindow
mouse move