CHAPTER 3

User interface

3.1 2D User interface with AWT

This section introduces the elements necessary to create a two-dimensional game with the standard AWT library. This library is included by default in all editions of Java, which allows porting to all platforms. It is also very easy to use, which is particularly interesting when you start.

This section also introduces several common concepts in the design of a video game - so it is relevant to browse it, even if you already know AWT. In this section, we implement graphics components with a naïve approach. In the last part of this chapter, they are formalized to produce an advanced design, capable of managing larger projects.

The source code for the various examples in this section is available in the sample Java project, in the “examples/chap03/awt” folder of the source packages.

3.1.1 Synchronous display with double buffering

3.1.1.1 Display a window

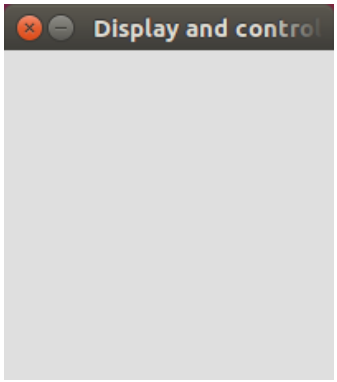
Showing a window in AWT is very simple (file “E01Window.java”):

```
public class E01Window extends Frame {

    public void init() {
        setTitle("Display and controls with AWT");
        setSize(200,200);
        setResizable(false);
        addWindowListener(new WindowAdapter() {
          public void windowClosing(WindowEvent e){
            dispose();
          }
        });
    }

    public static void main(String args[]) {
        E01Window window = new E01Window();
        window.init();
        window.setLocationRelativeTo(null);
        window.setVisible(true);
    }
}
```

The result is a basic window:



The `init()` method prepares the window. We first defines the title with the `setTitle()` method:

```
setTitle("Display and controls with AWT");
```

We define a size with the `setSize()` method:

```
setSize(200,200);
```

We indicate that the window can not be resized with the `setResizable()` method:

```
setResizable(false);
```

Finally, we manage the closing of the window with a window listener which destroys the window with the `dispose()` method when we ask it to be closed (button in the interface or key as Alt + F4):

```
addWindowListener(new WindowAdapter() {
    public void windowClosing(WindowEvent we) {
        dispose();
    }
});
```

The `main()` function instantiates and displays the window. We instantiate the window:

```
E01Window window = new E01Window();
```

We initialize the window parameters:

```
window.init();
```

The window is centred on the screen:

```
window.setLocationRelativeTo(null);
```

Finally, we display the window:

```
window.setVisible(true);
```

3.1.1.2 Create a canvas

To obtain the best possible performance for the display, we opt for a display using a canvas. A canvas represents a rectangular area where you can draw. This approach makes it possible, in particular, to use the double buffering presented in the following section.

We enrich the previous example with this new feature (file “E02Canvas.java”). We add an attribute of type `java.awt.Canvas` with a predefined size:

```
private int canvasWidth = 800;
private int canvasHeight = 600;
private Canvas canvas;
```

Add a `createCanvas()` method to instantiate and set this pattern:

```
public void createCanvas() {
    canvas = new Canvas();
    Dimension dim = new Dimension(canvasWidth,canvasHeight);
    canvas.setPreferredSize(dim);
    canvas.setMinimumSize(dim);
    canvas.setMaximumSize(dim);
    add(canvas);
    pack();
}
```

- The three methods `setXXXSize()` allow you to set a static canvas size.
- The `add()` method adds the canvas to the window.
- Finally, the `pack()` method adjusts the size of the window to contain the canvas. Its final size is that of the canvas plus the one for the window frame.

We add the canvas creation to the `main()` method:

```
public static void main(String args[]) {
    E02Canvas window = new E02Canvas();
    window.init();
    window.createCanvas();
    window.setLocationRelativeTo(null);
    window.setVisible(true);
}
```

The result is a window with a canvas of 800 per 600 pixels:

3.1.1.3 Double buffering and main loop (Game Loop Pattern)

It is advisable to use canvas with a double buffering to display most efficiently. The canvas is not visible on the screen, and its content is displayed when requested. This low-level approach to the display allows you to draw in a non-visible buffer when a second one is displayed. Then, when the first buffer is ready to be displayed, we swap the two buffers: the first is displayed, and the second one is rendered. In doing so, there is no flickering effect: the user does not see the rendering of successive layers, only the final result. Besides, it allows parallelizing the processing: part of the resources is used for rendering, while another part is used for display.

To obtain this display strategy, we add a new `render()` method:

```
public void render() {
    BufferStrategy bs = canvas.getBufferStrategy();
    if (bs == null) {
        canvas.createBufferStrategy(2);
        return;
    }
    Graphics g = null;
    try {
        g = bs.getDrawGraphics();
        g.setColor(Color.black);
        g.fillRect(0,0,canvasWidth,canvasHeight);
        bs.show();
    }
    finally {
        if (g != null)
            g.dispose();
    }
}
```

This method performs the following operations. It checks that the canvas has display buffers with the `getBufferStrategy()` method. If it is not the case, we ask for its creation with 2 buffers with the `createBufferStrategy()` method, and then we leave the method:

```
BufferStrategy bs = canvas.getBufferStrategy();
if (bs == null) {
    canvas.createBufferStrategy(2);
    return;
}
```

To draw in the canvas, we go through the buffer strategy `bs`. This can return a `Graphics` with the `getDrawGraphics()` method. This `Graphics` automatically points to the buffer ready to be drawn. It is necessary to release this `Graphics` at the end of the display whatever the circumstances, so the use of a `try ...`

`finally` block:

```
Graphics g = null;
try {
    g = bs.getDrawGraphics();
   ... Draw in g ...
}
finally {
    if (g != null) {
        g.dispose();
    }
}
```

In the `try ... finally` block, we can draw in `Graphics`, as in the `paintComponent()` method of a `JComponent` Here, we draw a solid rectangle, to give a black background to the canvas:

```
g.setColor(Color.black);
g.fillRect(0,0,canvasWidth,canvasHeight);
```

Finally, we call the `show()` method of the buffer to get the display. With a double buffer, this swaps the buffer to draw with the one to display:

```
bs.show();
```

The `render()` method must be called regularly to update the display. To do this, we implement a new `run()` with a loop that calls it continuously:

```
public void run() {
    while (running) {
        render();
    }
    dispose();
}
```

This method uses a new `running` boolean attribute set to `true`. As long as it has this value, the loop continues infinitely. Once passed to `false`, we leave the loop, then we ask for the destruction of the window with the `dispose()` method. This loop is at the heart of any video game: once embellished with other calls presented in this section, it is a pattern that does not have an “official” name, but can be called the *Game Loop Pattern*.

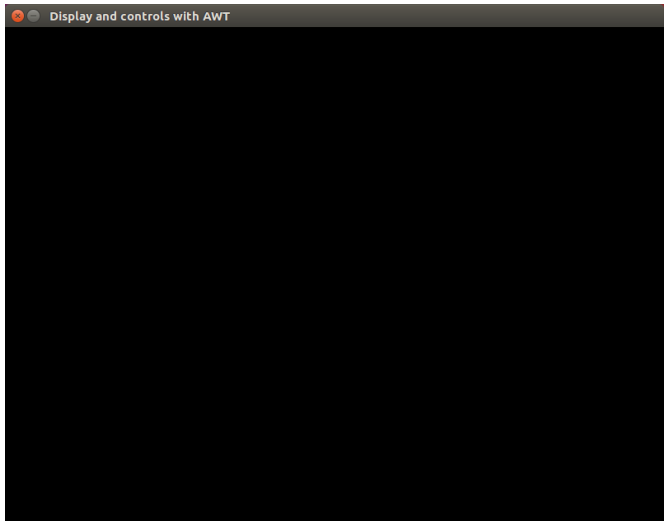
To pass the `running` attribute to `false`, and thus leave the game, we modify the closing behavior of the window in the `init()` method: instead of destroying the window, we pass `running` to `false`:

```
public void init() {
    setTitle("Display and controls with AWT");
    setSize(200,200);
    setResizable(false);
    addWindowListener(new WindowAdapter() {
        public void windowClosing(WindowEvent we) {
            running = false;
        }
    });
}
```

Finally, all that remains is to call the `run()` method in the `main()` method to trigger the loop:

```
public static void main(String args[]) {
    E03DoubleBuffer window = new E03DoubleBuffer();
    window.init();
    window.createCanvas();
    window.setLocationRelativeTo(null);
    window.setVisible(true);
    window.run();
}
```

The complete code for this example is in the file “E03DoubleBuffer.java”. The result is an entirely black canvas:



3.1.1.4 Synchronization

The solution in the previous example performs a constant display, using all available resources. If we observe the use of the processor during its execution, we can see that it uses a full core. On the smallest machines, it can slow down the whole system. This expenditure of resources is useless: even on a small machine, many more images are drawn than displayed. Indeed, screens all have a maximum refresh rate: usually 60 frames per second. This frequency is a function of the sensitivity of the human eye, which hardly perceives the image changes at this frequency.

One solution is not to draw more frames per second than can display the screen. To do this, we modify the `run()` method to impose a pause between two drawings. The pause time is calculated to ensure a fixed number of frames per second. If we run the example "E04DisplaySync.java", the display is the same, but the CPU usage of the program is very low.

Here is the new `run()` method:

```
public void run()
{
    int fps = 60;
    long nanoPerFrame = (long) (1000000000.0 / fps);
    long lastTime = System.nanoTime();
    while (running) {
        long nowTime = System.nanoTime();
        if ((nowTime-lastTime) < nanoPerFrame) {
            continue;
        }
        lastTime = nowTime;
        render();
        long elapsed = System.nanoTime() - lastTime;
        long milliSleep = (nanoPerFrame - elapsed) / 1000000;
        if ( milliSleep > 0) {
            try {
                Thread.sleep (milliSleep);
            } catch (InterruptedException ex) {
                ex.printStackTrace();
            }
        }
    }
    dispose();
}
```