Node Orientation in JavaFX

Introduction

Node orientation describes the flow of visual data within a control. In the English speaking world, visual data normally flows from left-to-right. In an Arabic or Hebrew world, visual data flows from right-to-left. This is consistent with the reading order of text in both worlds.

Node orientation is different from BIDI, which is the term used to describe the Bidirectional text layout algorithm that governs the visual ordering of characters within a sentence or text fragment. Mirroring (described in detail later on) refers to a particular algorithm that can be used to implement node orientation.

Design Goals

These are the design goals for node orientation in JavaFX:

- Applications must opt into node orientation
- Applications must have full control over orientation within a node
- The node orientation API must be simple, minimal and consistent

There is a temptation to support a default orientation and automatically set orientation based on the current locale or other language specific platform state. This is a mistake. Some Middle Eastern users run with left-to-right and use a right-to-left orientation for text controls only. Further, changing the orientation of an application to right-to-left automatically is unlikely to produce a good result. Images can contain data that is directional. Some nodes should always be on the left or right regardless of orientation, for example, the doors of a car. Finally, some controls, such as text editors, need to be able to switch orientation dynamically in response to a standard key stroke.

Therefore, the default orientation for all nodes should be left-to-right. Applications must explicitly set node orientation based on application specific requirements and state.

Inheritance of Orientation

Inheritance of node orientation allows application developers to specify the orientation of a root node and have it apply to all children, rather than explicitly setting the orientation of each child. This is a useful mechanism that reduces the amount of code developers must write.

Because the orientation of children might be different for each child, the orientation of a child node when explicitly set can override the parent. For example, the top level window might be right-to-left, with the title and close box appearing on the left. A child car node, containing doors that must be on the left and right regardless of the parent, would have orientation explicitly set to left-to-right. However, a button containing an image and text that is associated with a car door would have orientation set to right-to-left so that the text draws on the left and image on the right.

Nodes that don't inherit by default

By default, most nodes will inherit their orientation from their parent. There are some exceptions to this rule. For example, an Image View is left-to-right by default. Since there is no way to determine whether the data in an image is directional and applications sometimes load different directional images based on locale, then inheriting right-to-left orientation is unlikely to give a good result for images.

Nodes that don't inherit by default have an initial left-to-right orientation. However, the orientation of any node is under application control and can be explicitly set rather than inheriting. If an application decides that an image can draw both left-to-right and right-to-left and give a good result, the application will need to explicitly set the orientation.

The following nodes default to left-to-right: Image View, Charts, Indefinite Progress Bar, Indefinite Progress Indicator, Definite Progress Indicator, Check Mark, Canvas, MediaPlayer, WebView.

Node Orientation API

This enum captures the three possible states that node orientation can have:

```
/**
 * A set of values for describing the drawing direction of a node.
 *
 */
public enum NodeOrientation {
    /**
     * Indicates that the node draws from left-to-right.
     */
    LEFT_TO_RIGHT,

    /**
     * Indicates that the node draws from right-to-left.
     */
    RIGHT_TO_LEFT,

    /**
     * Indicates that the node inherits orientation from the parent.
     */
    INHERIT
}
```

In order for application code to behave correctly, the inherited state is important is understand. If an application explicitly sets the root of a hierarchy to left-to-right and then reparents the hierarchy into a parent that is right-to-left, the hierarchy will remain left-to-right.

These methods are in Node and Scene to allow the programmer to get/set and determine the effective orientation:

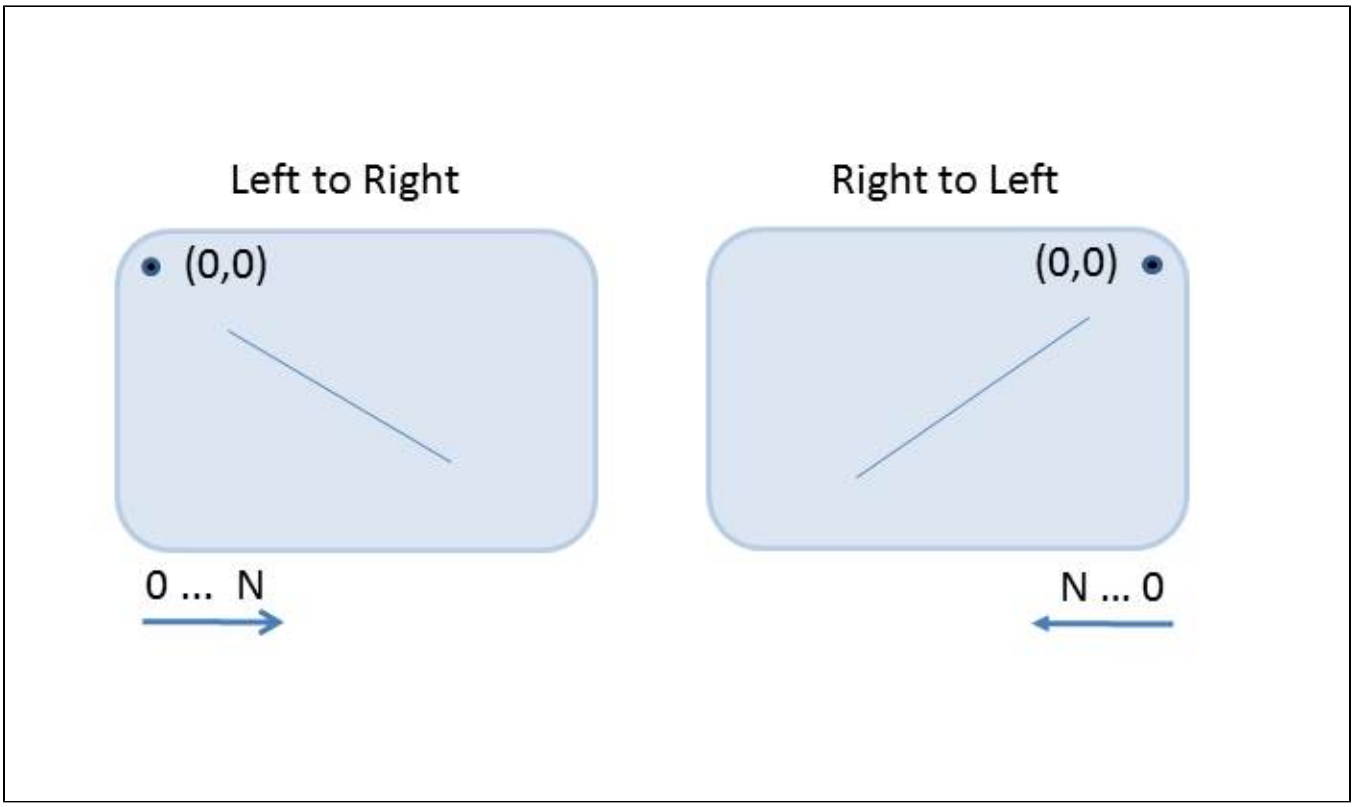
```
public final void setNodeOrientation(NodeOrientation value) {
    //Implementation goes here
}
public final NodeOrientation getNodeOrientation() {
    //Implementation goes here
    return null;
}

/**
 * Property holding NodeOrientation.
 * <p>
 * Node orientation describes the flow of visual data within a node.
 * In the English speaking world, visual data normally flows from
 * left-to-right. In an Arabic or Hebrew world, visual data flows
 * from right-to-left.  This is consistent with the reading order
 * of text in both worlds.  The default value is left-to-right.
 * </p>
 *
 * @return NodeOrientation
 */
public final ObjectProperty<NodeOrientation> nodeOrientationProperty() {
    //Implementation goes here
    return null;
}

/**
 * Returns the NodeOrientation that is used to draw the node.
 * <p>
 * The effective orientation of a node resolves the inheritance of
 * node orientation, returning either left-to-right or right-to-left.
 * </p>
 */
public final NodeOrientation getEffectiveNodeOrientation() {
    //Implementation goes here - search up the parent or return cached value
    return null;
}
/**
 * Determines whether a node should be mirrored when node orientation
 * is right-to-left.
 * <p>
 * When a node is mirrored, the origin is automtically moved to the
 * top right corner causing the node to layout children and draw from
 * right to left using a mirroring transformation.  Some nodes may wish
 * to draw from right to left without using a transformation.  These
 * nodes will will answer {@code false} and implement right-to-left
 * orientation without using the automatic transformation.
 * </p>
 */
public boolean isAutomaticallyMirrored() {
    return true;
}
```

Mirroring

Mirroring is a term used to describe one possible technique that can be used to implement node orientation. The main idea is that the top left corner of a node is moved to the top right.



Application code is written assuming that the orientation is left to right. The constants/concepts LEADING/TRAILING are unnecessary and application code continues to use LEFT/RIGHT and other left to right concepts. Input events are translated automatically such that if an application were to print the x and y coordinates of the mouse during a mouse press event, the numbers will be the same regardless of orientation. The node is unaware that it has been mirrored (but can find out by checking the effective orientation).

Advantages of Mirroring

Mirroring has the advantage that almost all of the existing left-to-right code (including Canvas code that draws in immediate mode) works without modification when node orientation is right-to-left. Without mirroring, code that uses layout nodes will work because the toolkit will be modified to position nodes from right to left, but code that performs absolute positioning or does coordinate math to place children will need to be modified. It's is easy enough (but tedious) to fix the toolkit, but each application developer must fix his or her code.

Disadvantages of Mirroring

The main disadvantage of mirroring is that it requires code that performs coordinate math and code that performs directional actions based on the keyboard to be changed. Failure to change this code is manifested by obvious bugs in the node. This is unfortunate because the main advantage to mirroring is that it avoids code changes and this type of code must be changed.

Coordinate Transformations

Code that performs coordinate transformation between nodes or the desktop where the orientation of the nodes is different can compute the wrong coordinate. This happens most often when a right-to-left node attempts to position top level window (the desktop is always left-to-right).

The example code below positions a stage such that it is always aligned with the left edge of a button regardless how the button ends up being orientated. When the button is on a right-to-left container, mapping (0,0) in the button to scene coordinates gives a point that is in the upper right corner of the button. This is correct as far as the scene is concerned but applying left-to-right coordinate math to compute the point on the screen causes the stage to be aligned with the right side of the button, not the left. In order for the stage to be positioned properly, it is necessary to test for orientation and subtract the width of the button.

```
public class MirrorPosition extends Application {
    public void start(final Stage stage) {
        final Button button = new Button("Button");
        Group group = new Group ();
        final Scene scene = new Scene(group, 200, 200);
        scene.setNodeOrientation(NodeOrientation.RIGHT_TO_LEFT);
        button.setLayoutX(25);
        button.setLayoutY(30);
        button.addEventHandler(MouseEvent.MOUSE_CLICKED,
            new EventHandler<MouseEvent>() {
                public void handle(MouseEvent event) {
                    /* Brute force code to position a stage */
                    Point2D pt = button.localToScene(0, 0);
                    Bounds bounds = button.getLayoutBounds();
                    Stage stage2 = new Stage();
                    stage2.setScene(new Scene(new Group()));
                    NodeOrientation orientation =
                        button.getEffectiveNodeOrientation();
                    double mirrorWidth =
                        orientation == NodeOrientation.RIGHT_TO_LEFT
                            ? bounds.getWidth() : 0;
                    stage2.setX(pt.getX() +
                        scene.getX() + stage.getX() - mirrorWidth);
                    stage2.setY(pt.getY() +
                         scene.getY() + stage.getY() + bounds.getHeight());
                    stage2.setWidth(200);
                    stage2.setHeight(200);
                    stage2.show();
                }
            });
        group.getChildren().addAll(button);
        stage.setScene(scene);
        stage.show();
    }

    public static void main(String[] args) {
        launch(MirrorPosition.class, args);
    }
```

Rather than testing for orientation, a much better approach is to map a full rectangle to scene coordinates. This is cleaner and works in both orientations because the rectangle is automatically flipped when mapped. Here is a code fragment that shows this approach:

```
button.addEventHandler(MouseEvent.MOUSE_CLICKED,
            new EventHandler<MouseEvent>() {
                public void handle(MouseEvent event) {
                    /* Brute force code to position a stage */
                    Bounds bounds = button.getLayoutBounds();
                    bounds = button.localToScene(bounds);
                    Stage stage2 = new Stage();
                    stage2.setScene(new Scene(new Group()));
                    stage2.setX(bounds.getMinX() +
                        scene.getX() + stage.getX());
                    stage2.setY(bounds.getMinY() +
                         scene.getY() + stage.getY() + bounds.getHeight());
                    stage2.setWidth(200);
                    stage2.setHeight(200);
                    stage2.show();
                }
            });
```

Directional Keys

Code that interprets directional keys such as the left and right arrow keys needs to be modified to perform the function always to the left or always to the right, regardless of orientation. A good example of this in concept in action is a spreadsheet. Regardless of orientation, when the user presses the left arrow, the focus should move to the cell to the left. Failure to handle this case causes the application to behave "backwards".

Regardless of a mirrored orientation implementation, directional key code often needs to be changed. For example, a left-to-right oriented tree may wish to fully expand children when a modifier and the left arrow key is pressed. A right-to-left oriented tree may wish to use a right arrow key instead.

Overriding Mirroring

Mirroring is a powerful but invasive. Some nodes may draw correctly when mirrored, but picking and directionality inherent in the data that the node represents, may make a mirrored implementation too complex. For example, a text editor may wish to implement node orientation using right alignment rather than using mirroring. Therefore, mirroring can be overridden on a node by node basis, giving node implementers a choice.

Effects

Effects do not have an orientation. In a mirrored implementation, an effect (for example a shadow) will be mirrored with the node. This is very often the desired behavior. In order to provide an effect that is consistent regardless of orientation (such as a shadow that always comes from a light source in the top left) the application will need to configure parameters that are appropriate for the effect in both orientations.

Conclusion

It is important that the application have full control over node orientation rather than having orientation applied automatically. Inheritance of node orientation is a useful mechanism that reduces the amount of code that application programmers need to write. Mirroring is an interesting implementation technique that can simplify node orientation for toolkit implementers and application developers.