JavaFX
Transformations, Animations, and Visual Effects
Release 8
E50479-01

March 2014
This document describes transformations, timeline animations, and visual effects available in JavaFX.
Preface

About This Document ........................................................................................................................................ v
Audience ............................................................................................................................................................. v
Documentation Accessibility ............................................................................................................................. v
Related Documents ........................................................................................................................................... v
Conventions ....................................................................................................................................................... v

Part I Applying Transformations in JavaFX

1 Transformations Overview
   Introducing Transformations .......................................................................................................................... 1-1

2 Transformation Types and Examples
   Translation .......................................................................................................................................................... 2-1
   Rotation ............................................................................................................................................................ 2-1
   Scaling ............................................................................................................................................................ 2-2
   Shearing .......................................................................................................................................................... 2-3
   Multiple Transformations ................................................................................................................................ 2-4
   Application Files ............................................................................................................................................. 2-4

Part II Creating Transitions and Timeline Animations

3 Animation Basics
   Transitions ..................................................................................................................................................... 3-1
      Fade Transition ............................................................................................................................................. 3-1
      Path Transition .......................................................................................................................................... 3-1
      Parallel Transition .................................................................................................................................... 3-2
      Sequential Transition ................................................................................................................................ 3-3
   Timeline Animation ...................................................................................................................................... 3-4
      Basic Timeline Animation ......................................................................................................................... 3-4
      Timeline Events ....................................................................................................................................... 3-5
   Interpolators ..................................................................................................................................................... 3-7
      Built-in Interpolators ................................................................................................................................. 3-7
      Custom Interpolators ................................................................................................................................. 3-7
4 Tree Animation Example

Project and Elements .............................................................................................................. 4-1
Grass ........................................................................................................................................ 4-2
  Creating Grass .................................................................................................................. 4-2
  Creating Timeline Animation for Grass Movement ......................................................... 4-3
Tree ........................................................................................................................................... 4-4
  Branches .......................................................................................................................... 4-4
  Leaves and Flowers ......................................................................................................... 4-6
  Animating Tree Elements ............................................................................................... 4-6
    Growing a Tree ............................................................................................................ 4-6
    Creating Tree Crown Movement ................................................................................. 4-7
    Animating Season Change ......................................................................................... 4-8
Application Files .................................................................................................................. 4-10

Part III Creating Visual Effects

5 Applying Effects

  Blend Effect .................................................................................................................. 5-1
  Bloom Effect ................................................................................................................ 5-2
  Blur Effects ................................................................................................................... 5-3
    BoxBlur ..................................................................................................................... 5-3
    Motion Blur .............................................................................................................. 5-4
    Gaussian Blur ......................................................................................................... 5-5
  Drop Shadow Effect .................................................................................................... 5-5
  Inner Shadow Effect .................................................................................................... 5-7
  Reflection ..................................................................................................................... 5-8
  Lighting Effect .............................................................................................................. 5-9
  Perspective Effect ........................................................................................................ 5-10
  Creating a Chain of Effects ......................................................................................... 5-11
Application Files .................................................................................................................. 5-13

Part IV Source Code for the Transformations, Animations, and Visual Effects

Tutorial

A Xylophone.java
Preface

This preface describes the document accessibility features and conventions used in this tutorial - *Transformations, Animations, and Visual Effects*.

About This Document

This tutorial describes transformations, timeline animations, and visual effects available in JavaFX.

Audience

This document is intended for JavaFX developers.

Documentation Accessibility


Access to Oracle Support


Related Documents

For more information, see the following documents in the JavaFX documentation set:

- *Getting Started with JavaFX*

Conventions

The following text conventions are used in this document:

<table>
<thead>
<tr>
<th>Convention</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>boldface</strong></td>
<td>Boldface type indicates graphical user interface elements associated with an action, or terms defined in text or the glossary.</td>
</tr>
<tr>
<td>Convention</td>
<td>Meaning</td>
</tr>
<tr>
<td>------------</td>
<td>---------</td>
</tr>
<tr>
<td><em>italic</em></td>
<td>Italic type indicates book titles, emphasis, or placeholder variables for which you supply particular values.</td>
</tr>
<tr>
<td><em>monospace</em></td>
<td>Monospace type indicates commands within a paragraph, URLs, code in examples, text that appears on the screen, or text that you enter.</td>
</tr>
</tbody>
</table>
Part I
Applying Transformations in JavaFX

This tutorial describes transformations supported in JavaFX and contains the following chapters:

- Transformations Overview
- Transformation Types and Examples
This chapter introduces transformations supported in JavaFX.

All transformations are located in the `javafx.scene.transform` package and are subclasses of the `Transform` class.

**Introducing Transformations**

A transformation changes the place of a graphical object in a coordinate system according to certain parameters. The following types of transformations are supported in JavaFX:

- Translation
- Rotation
- Scaling
- Shearing

These transformations can be applied to either a standalone node or to groups of nodes. You can apply one transformation at a time or you can combine transformations and apply several transformations to one node.

The `Transform` class implements the concepts of affine transformations. The `Affine` class extends the `Transform` class and acts as a superclass to all transformations. Affine transformations are based on euclidean algebra, and perform a linear mapping (through the use of matrixes) from initial coordinates to other coordinates while preserving the straightness and parallelism of lines. Affine transformations can be constructed using `ObservableList` rotations, translations, scales, and shears.

**Note:** Usually, do not use the `Affine` class directly, but instead, use the specific Translate, Scale, Rotate, or Shear transformations.

Transformations in JavaFX can be performed along three coordinates, thus enabling users to create three-dimensional (3-D) objects and effects. To manage the display of objects with depth in 3-D graphics, JavaFX implements z-buffering. Z-buffering ensures that the perspective is the same in the virtual world as it is in the real one: a solid object in the foreground blocks the view of one behind it. Z-buffering can be enabled by using the `setDepthTest` class. You can try to disable z-buffering (`setDepthTest(DepthTest.DISABLE)`) in the sample application to see the effect of the z-buffer.

To simplify transformation usage, JavaFX implements transformation constructors with the x-axis and y-axis along with the x, y, and z axes. If you want to create a
Introducing Transformations

two-dimensional (2-D) effect, you can specify only the x and y coordinates. If you want to create a 3-D effect, specify all three coordinates.

To be able to see 3-D objects and transformation effects in JavaFX, users must enable the perspective camera.

Though knowing the underlying concepts can help you use JavaFX more effectively, you can start using transformations by studying the example provided with this document and trying different transformation parameters. For more information about particular classes, methods, or additional features, see the API documentation.

In this document, a Xylophone application is used as a sample to illustrate all the available transformations. You can download its source code by clicking the transformations.zip link.

Figure 1–1  A Xylophone application
Transformation Types and Examples

This document describes specific transformations and provides code examples.

Translation

The translation transformation shifts a node from one place to another along one of the
axes relative to its initial position. The initial position of the xylophone bar is defined
by $x$, $y$, and $z$ coordinates. In Example 2–1, the initial position values are specified by
the $x_{\text{Start}}$, $y_{\text{Pos}}$, and $z_{\text{Pos}}$ variables. Some other variables are added to simplify the
calculations when applying different transformations. Each bar of the xylophone is
based on one of the base bars. The example then translates the base bars with different
shifts along the three axes to correctly locate them in space.

Example 2–1 shows a code snippet from the sample application with the translation transformation.

Example 2–1  Translation

```java
Group rectangleGroup = new Group();
rectangleGroup.setDepthTest(DepthTest.ENABLE);

double $x_{\text{Start}} = 260.0$;
double $x_{\text{Offset}} = 30.0$;
double $y_{\text{Pos}} = 300.0$;
double $z_{\text{Pos}} = 0.0$;
double $bar_{\text{Width}} = 22.0$;
double $bar_{\text{Depth}} = 7.0$;

// Base1
Cube base1Cube = new Cube(1.0, new Color(0.2, 0.12, 0.1, 1.0), 1.0);
base1Cube.setTranslateX($x_{\text{Start}} + 135$);
base1Cube.setTranslateZ($y_{\text{Pos}} + 20.0$);
base1Cube.setTranslateY(11.0);
```

Rotation

The rotation transformation moves the node around a specified pivot point of the
scene. You can use the `rotate` method of the `Transform` class to perform the rotation.

To rotate the camera around the xylophone in the sample application, the rotation
transformation is used, although technically, it is the xylophone itself that is moving
when the mouse rotates the camera.
Scaling

Example 2–2 shows the code for the rotation transformation.

**Example 2–2 Rotation**

```java
Example 2–2 Rotation
class Cam extends Group {
    Translate t = new Translate();
    Translate p = new Translate();
    Translate ip = new Translate();
    Rotate rx = new Rotate();
    { rx.setAxis(Rotate.X_AXIS); }
    Rotate ry = new Rotate();
    { ry.setAxis(Rotate.Y_AXIS); }
    Rotate rz = new Rotate();
    { rz.setAxis(Rotate.Z_AXIS); }
    Scale s = new Scale();
    public Cam() { super(); getTransforms().addAll(t, p, rx, rz, ry, s, ip); }
}

Note that the pivot point and the angle define the destination point the image is moved to. Carefully calculate values when specifying the pivot point. Otherwise, the image might appear where it is not intended to be. For more information, see the API documentation.

Scaling

The scaling transformation causes a node to either appear larger or smaller, depending on the scaling factor. Scaling changes the node so that the dimensions along its axes are multiplied by the scale factor. Similar to the rotation transformations, scaling transformations are applied at a pivot point. This pivot point is considered the point around which scaling occurs.
To scale, use the `Scale` class and the `scale` method of the `Transform` class.

In the Xylophone application, you can scale the xylophone using the mouse while pressing Alt and the right mouse button. The scale transformation is used to see the scaling.

**Example 2–3** shows the code for the scale transformation.

```java
Example 2–3  Scaling
else if (me.isAltDown() && me.isSecondaryButtonDown()) {
    double scale = cam.s.getX();
    double newScale = scale + mouseDeltaX*0.01;
    cam.s.setX(newScale); cam.s.setY(newScale); cam.s.setZ(newScale);
}
```

**Shearing**

A shearing transformation rotates one axis so that the x-axis and y-axis are no longer perpendicular. The coordinates of the node are shifted by the specified multipliers.

To shear, use the `Shear` class or the `shear` method of the `Transform` class.

In the Xylophone application, you can shear the xylophone by dragging the mouse while holding Shift and pressing the left mouse button.

**Figure 2–1  Shearing Transformation**

![Shearing Transformation](image)

**Example 2–4** shows the code snippet for the shear transformation.

```java
Example 2–4  Shearing
else if (me.isShiftDown() && me.isPrimaryButtonDown()) {
    double yShear = shear.getY();
    shear.setY(yShear + mouseDeltaY/1000.0);
    double xShear = shear.getX();
    shear.setX(xShear + mouseDeltaX/1000.0);
}
```
Multiple Transformations

You can construct multiple transformations by specifying an ordered chain of transformations. For example, you can scale an object and then apply a shearing transformation to it, or you can translate an object and then scale it.

Example 2–5 shows multiple transformations applied to an object to create a xylophone bar.

Example 2–5  Multiple Transformations

```
Cube base1Cube = new Cube(1.0, new Color(0.2, 0.12, 0.1, 1.0), 1.0);
base1Cube.setTranslateX(xStart + 135);
base1Cube.setTranslateZ(yPos+20.0);
base1Cube.setTranslateY(11.0);
base1Cube.setScaleX(barWidth*11.5);
base1Cube.setScaleZ(10.0);
base1Cube.setScaleY(barDepth*2.0);
```
Part II
Creating Transitions and Timeline Animations

This tutorial contains information that you can use to create animation in JavaFX and contains the following chapters.

Animation Basics provides basic animation concepts and contains the following parts:

- Transitions
- Timeline Animation
- Interpolators

The Tree Animation Example chapter contains a description of the Tree Animation sample and provides some tips and tricks about animation in JavaFX.
Animation in JavaFX can be divided into timeline animation and transitions. This chapter provides examples of each animation type.

Timeline and Transition are subclasses of the javafx.animation.Animation class. For more information about particular classes, methods, or additional features, see the API documentation.

Transitions

Transitions in JavaFX provide the means to incorporate animations in an internal timeline. Transitions can be composed to create multiple animations that are executed in parallel or sequentially. See the Parallel Transition and Sequential Transition sections for details. The following sections provide some transition animation examples.

Fade Transition

A fade transition changes the opacity of a node over a given time.

Example 3–1 shows a code snippet for a fade transition that is applied to a rectangle. First a rectangle with rounded corners is created, and then a fade transition is applied to it.

Example 3–1  Fade Transition

```java
final Rectangle rect1 = new Rectangle(10, 10, 100, 100);
rect1.setArcHeight(20);
rect1.setArcWidth(20);
rect1.setFill(Color.RED);
...
FadeTransition ft = new FadeTransition(Duration.millis(3000), rect1);
ft.setFromValue(1.0);
ft.setToValue(0.1);
ft.setCycleCount(Timeline.INDEFINITE);
ft.setAutoReverse(true);
ft.play();
```

Path Transition

A path transition moves a node along a path from one end to the other over a given time.
Example 3–2 shows a code snippet for a path transition that is applied to a rectangle. The animation is reversed when the rectangle reaches the end of the path. In code, first a rectangle with rounded corners is created, and then a new path animation is created and applied to the rectangle.

**Example 3–2  Path Transition**

```java
final Rectangle rectPath = new Rectangle(0, 0, 40, 40);
rectPath.setArcHeight(10);
rectPath.setArcWidth(10);
rectPath.setFill(Color.ORANGE);
...
Path path = new Path();
path.getElements().add(new MoveTo(20, 20));
path.getElements().add(new CubicCurveTo(380, 0, 380, 120, 200, 120));
path.getElements().add(new CubicCurveTo(0, 120, 0, 240, 380, 240));
PathTransition pathTransition = new PathTransition();
pathTransition.setDuration(Duration.millis(4000));
pathTransition.setPath(path);
pathTransition.setNode(rectPath);
pathTransition.setOrientation(PathTransition.OrientationType.ORTHOGONAL_TO_TANGENT);
pathTransition.setCycleCount(Timeline.INDEFINITE);
pathTransition.setAutoReverse(true);
pathTransition.play();
```

**Parallel Transition**

A parallel transition executes several transitions simultaneously.

Example 3–3 shows the code snippet for the parallel transition that executes fade, translate, rotate, and scale transitions applied to a rectangle.

**Example 3–3 Parallel Transition**

```java
Rectangle rectParallel = new Rectangle(10, 200, 50, 50);
```
rectParallel.setArcHeight(15);
rectParallel.setArcWidth(15);
rectParallel.setFill(Color.DARKBLUE);
rectParallel.setTranslateX(50);
rectParallel.setTranslateY(75);
...

...  
FadeTransition fadeTransition =  
    new FadeTransition(Duration.millis(3000), rectParallel);  
fadeTransition.setFromValue(1.0f);  
fadeTransition.setToValue(0.3f);  
fadeTransition.setCycleCount(2);  
fadeTransition.setAutoReverse(true);  
TranslateTransition translateTransition =  
    new TranslateTransition(Duration.millis(2000), rectParallel);  
translateTransition.setFromX(50);  
translateTransition.setToX(350);  
translateTransition.setCycleCount(2);  
translateTransition.setAutoReverse(true);  
RotateTransition rotateTransition =  
    new RotateTransition(Duration.millis(3000), rectParallel);  
rotateTransition.setByAngle(180f);  
rotateTransition.setCycleCount(4);  
rotateTransition.setAutoReverse(true);  
ScaleTransition scaleTransition =  
    new ScaleTransition(Duration.millis(2000), rectParallel);  
scaleTransition.setToX(2f);  
scaleTransition.setToY(2f);  
scaleTransition.setCycleCount(2);  
scaleTransition.setAutoReverse(true);  
parallelTransition = new ParallelTransition();  
parallelTransition.getChildren().addAll(  
    fadeTransition,  
    translateTransition,  
    rotateTransition,  
    scaleTransition  
);  
parallelTransition.setCycleCount(Timeline.INDEFINITE);  
parallelTransition.play();

Sequential Transition

A sequential transition executes several transitions one after another.

Example 3–4 shows the code for the sequential transition that executes one after another. Fade, translate, rotate, and scale transitions that are applied to a rectangle.

Example 3–4  Sequential Transition

Rectangle rectSeq = new Rectangle(25,25,50,50);  
rectSeq.setArcHeight(15);  
rectSeq.setArcWidth(15);  
rectSeq.setFill(Color.CRIMSON);  
rectSeq.setTranslateX(50);  
rectSeq.setTranslateY(50);  
...

...  
FadeTransition fadeTransition =  
    new FadeTransition(Duration.millis(1000), rectSeq);
fadeTransition.setFromValue(1.0f);
fadeTransition.setToValue(0.3f);
fadeTransition.setCycleCount(1);
fadeTransition.setAutoReverse(true);

TranslateTransition translateTransition =
    new TranslateTransition(Duration.millis(2000), rectSeq);
translateTransition.setFromX(50);
translateTransition.setToX(375);
translateTransition.setCycleCount(1);
translateTransition.setAutoReverse(true);

RotateTransition rotateTransition =
    new RotateTransition(Duration.millis(2000), rectSeq);
rotateTransition.setByAngle(180f);
rotateTransition.setCycleCount(4);
rotateTransition.setAutoReverse(true);

ScaleTransition scaleTransition =
    new ScaleTransition(Duration.millis(2000), rectSeq);
scaleTransition.setFromX(1);
scaleTransition.setFromY(1);
scaleTransition.setToX(2);
scaleTransition.setToY(2);
scaleTransition.setCycleCount(1);
scaleTransition.setAutoReverse(true);

sequentialTransition = new SequentialTransition();
sequentialTransition.getChildren().addAll(
    fadeTransition,
    translateTransition,
    rotateTransition,
    scaleTransition);
sequentialTransition.setCycleCount(Timeline.INDEFINITE);
sequentialTransition.setAutoReverse(true);
sequentialTransition.play();

For more information about animation and transitions, see the API documentation and the Animation section in the Ensemble project in the SDK.

**Timeline Animation**

An animation is driven by its associated properties, such as size, location, and color etc. Timeline provides the capability to update the property values along the progression of time. JavaFX supports key frame animation. In key frame animation, the animated state transitions of the graphical scene are declared by start and end snapshots (key frames) of the state of the scene at certain times. The system can automatically perform the animation. It can stop, pause, resume, reverse, or repeat movement when requested.

**Basic Timeline Animation**

The code in Example 3–5 animates a rectangle horizontally and moves it from its original position X=100 to X=300 in 500 ms. To animate an object horizontally, alter the x-coordinates and leave the y-coordinates unchanged.
Figure 3–3  Horizontal Movement

Example 3–5 shows the code snippet for the basic timeline animation.

Example 3–5  Timeline Animation

```java
final Rectangle rectBasicTimeline = new Rectangle(100, 50, 100, 50);
rectBasicTimeline.setFill(Color.RED);
...
final Timeline timeline = new Timeline();
timeline.setCycleCount(Timeline.INDEFINITE);
timeline.setAutoReverse(true);
final KeyValue kv = new KeyValue(rectBasicTimeline.xProperty(), 300);
final KeyFrame kf = new KeyFrame(Duration.millis(500), kv);
timeline.getKeyFrames().add(kf);
timeline.play();
```

Timeline Events

JavaFX provides the means to incorporate events that can be triggered during the timeline play. The code in Example 3–6 changes the radius of the circle in the specified range, and KeyFrame triggers the random transition of the circle in the x-coordinate of the scene.

Example 3–6  Timeline Events

```java
import javafx.application.Application;
import javafx.stage.Stage;
import javafx.animation.AnimationTimer;
import javafx.animation.KeyFrame;
import javafx.animation.Timeline;
import javafx.event.ActionEvent;
import javafx.event.EventHandler;
import javafx.scene.Group;
import javafx.scene.Scene;
import javafx.scene.effect.Lighting;
import javafx.scene.layout.StackPane;
import javafx.scene.paint.Color;
import javafx.scene.shape.Circle;
import javafx.scene.text.Text;
import javafx.util.Duration;

public class TimelineEvents extends Application {
    //main timeline
    private Timeline timeline;
    private AnimationTimer timer;

    //variable for storing actual frame
    private Integer i=0;
```
```java
@Override public void start(Stage stage) {
    Group p = new Group();
    Scene scene = new Scene(p);
    stage.setScene(scene);
    stage.setWidth(500);
    stage.setHeight(500);
    p.setTranslateX(80);
    p.setTranslateY(80);
    //create a circle with effect
    final Circle circle = new Circle(20, Color.rgb(156,216,255));
    circle.setEffect(new Lighting());
    //create a text inside a circle
    final Text text = new Text (i.toString());
    text.setStroke(Color.BLACK);
    //create a layout for circle with text inside
    final StackPane stack = new StackPane();
    stack.getChildren().addAll(circle, text);
    stack.setLayoutX(30);
    stack.setLayoutY(30);
    p.getChildren().add(stack);
    stage.show();

    //create a timeline for moving the circle
    timeline = new Timeline();
    timeline.setCycleCount(Timeline.INDEFINITE);
    timeline.setAutoReverse(true);
    //You can add a specific action when each frame is started.
    timer = new AnimationTimer() {
        @Override
        public void handle(long l) {
            text.setText(i.toString());
            i++;
        }
    };
    //create a keyValue with factory: scaling the circle 2times
    KeyValue keyValueX = new KeyValue(stack.scaleXProperty(), 2);
    KeyValue keyValueY = new KeyValue(stack.scaleYProperty(), 2);
    //create a keyFrame, the keyValue is reached at time 2s
    Duration duration = Duration.millis(2000);
    //one can add a specific action when the keyframe is reached
    EventHandler onFinished = new EventHandler<ActionEvent>() {
        public void handle(ActionEvent t) {
            stack.setTranslateX(java.lang.Math.random()*200-100); //reset counter
            i = 0;
        }
    };
    KeyFrame keyFrame = new KeyFrame(duration, onFinished, keyValueX, keyValueY);
    //add the keyframe to the timeline
    timeline.getKeyFrames().add(keyFrame);
    timeline.play();
}
```
Timer.start();
}

public static void main(String[] args) {
    Application.launch(args);
}

Interpolators

Interpolation defines positions of the object between the start and end points of the movement. You can use various built-in implementations of the Interpolator class or you can implement your own Interpolator to achieve custom interpolation behavior.

Built-in Interpolators

JavaFX provides several built-in interpolators that you can use to create different effects in your animation. By default, JavaFX uses linear interpolation to calculate the coordinates.

Example 3–7 shows a code snippet where the EASE_BOTH interpolator instance is added to the KeyValue in the basic timeline animation. This interpolator creates a spring effect when the object reaches its start point and its end point.

Example 3–7 Built-in Interpolator

```java
final Rectangle rectBasicTimeline = new Rectangle(100, 50, 100, 50);
rectBasicTimeline.setFill(Color.BROWN);
...
final Timeline timeline = new Timeline();
timeline.setCycleCount(Timeline.INDEFINITE);
timeline.setAutoReverse(true);
final KeyValue kv = new KeyValue(rectBasicTimeline.xProperty(), 300,
        Interpolator.EASE_BOTH);
final KeyFrame kf = new KeyFrame(Duration.millis(500), kv);
timeline.getKeyFrames().add(kf);
timeline.play();
```

Custom Interpolators

Apart from built-in interpolators, you can implement your own interpolator to achieve custom interpolation behavior. A custom interpolator example consists of two java files. Example 3–8 shows a custom interpolator that is used to calculate the y-coordinate for the animation. Example 3–9 shows the code snippet of the animation where the AnimationBooleanInterpolator is used.

Example 3–8 Custom Interpolator

```java
public class AnimationBooleanInterpolator extends Interpolator {
    @Override
    protected double curve(double t) {
        return Math.abs(0.5-t)*2 ;
    }
}
```

Example 3–9 Animation with Custom Interpolator

```java
public class AnimationBooleanInterpolator extends Interpolator {
    @Override
    protected double curve(double t) {
        return Math.abs(0.5-t)*2 ;
    }
}
```
final KeyValue keyValue1 = new KeyValue(rect.xProperty(), 300);
AnimationBooleanInterpolator yInterp = new AnimationBooleanInterpolator();
final KeyValue keyValue2 = new KeyValue(rect.yProperty(), 0., yInterp);

Application Files

NetBeans Projects
  - animations.zip
This chapter provides details about the Tree Animation example. You will learn how all the elements on the scene were created and animated.

Figure 4–1 shows the scene with a tree.

Figure 4–1  Tree Animation

Project and Elements

The Tree Animation project consists of several files. Each element, like leaves, grass blades, and others are created in separate classes. The TreeGenerator class creates a tree from all the elements. The Animator class contains all animation except grass animation that resides in the GrassWindAnimation class.

The scene in the example contains the following elements:

- Tree with branches, leaves, and flowers
- Grass

Each element is animated in its own fashion. Some animations run in parallel, and others run sequentially. The tree-growing animation is run only once, whereas the season-change animation is set to run infinitely.
The season-change animation includes the following parts:

- Leaves and flowers appear on the tree
- Flower petals fall and disappear
- Leaves and grass change color
- Leaves fall to the ground and disappear

Grass

This section describes how the grass is created and animated.

Creating Grass

In the Tree Animation example, the grass, shown in Figure 4–3 consists of separate grass blades, each of which is created using Path and added to the list. Each blade is then curved and colored. An algorithm is used to randomize the height, curve, and color of the blades, and to distribute the blades on the "ground." You can specify the number of blades and the size of the "ground" covered with grass.

Example 4–1 Creating a Grass Blade

```java
public class Blade extends Path {
    public final Color SPRING_COLOR = Color.color(random() * 0.5, random() * 0.5 + 0.5, 0.).darker();
    public final Color AUTUMN_COLOR = Color.color(random() * 0.4 + 0.3, random() * 0.1 + 0.4, random() * 0.2);
    private final static double width = 3;
    private double x = RandomUtil.getRandom(170);
    private double y = RandomUtil.getRandom(20) + 20;
    private double h = (50 * 1.5 - y / 2) * RandomUtil.getRandom(0.3);
    public SimpleDoubleProperty phase = new SimpleDoubleProperty();

    public Blade() {
        // Constructor code...
    }
}
```
Creating Timeline Animation for Grass Movement

Timeline animation that changes the x-coordinate of the top of the blade is used to create grass movement.

Several algorithms are used to make the movement look natural. For example, the top of each blade is moved in a circle instead of a straight line, and side curve of the blade make the blade look as if it bends under the wind. Random numbers are added to separate each blade movement.

Example 4–2 Grass Animation
class GrassWindAnimation extends Transition {

getElements().add(new MoveTo(0, 0));
final QuadCurveTo curve1;
final QuadCurveTo curve2;
getElements().add(curve1 = new QuadCurveTo(-10, h, h / 4, h));
getElements().add(curve2 = new QuadCurveTo(-10, h, width, 0));

setFill(AUTUMN_COLOR); //autumn color of blade
setStroke(null);

getTransforms().addAll(Transform.translate(x, y));

curve1.yProperty().bind(new DoubleBinding() {
    {
        super.bind(curve1.xProperty());
    }

    @Override
    protected double computeValue() {
        final double xx0 = curve1.xProperty().get();
        return Math.sqrt(h * h - xx0 * xx0);
    }
}); //path of top of blade is circle

//code to bend blade
curve1.controlYProperty().bind(curve1.yProperty().add(-h / 4));
curve2.controlYProperty().bind(curve2.yProperty().add(-h / 4));
curve1.xProperty().bind(new DoubleBinding() {
    final double rand = RandomUtil.getRandom(PI / 4);
    {
        super.bind(phase);
    }

    @Override
    protected double computeValue() {
        return (h / 4) + ((cos(phase.get() + (x + 400.) * PI / 1600 +
rand) + 1) / 2.) * (-3. / 4) * h;
    }

});
final private Duration animationTime = Duration.seconds(3);
final private DoubleProperty phase = new SimpleDoubleProperty(0);
final private Timeline tl = new Timeline(Animation.INDEFINITE);

public GrassWindAnimation(List<Blade> blades) {
    setCycleCount(Animation.INDEFINITE);
    setInterpolator(Interpolator.LINEAR);
    setCycleDuration(animationTime);
    for (Blade blade : blades) {
        blade.phase.bind(phase);
    }
}

@Override
protected void interpolate(double frac) {
    phase.set(frac * 2 * PI);
}

Tree

This section explains how the tree shown in Figure 4–4 is created and animated.

Figure 4–4  Tree

Branches

The tree consists of branches, leaves, and flowers. Leaves and flowers are drawn on
the top branches of the tree. Each branch generation consists of three branches (one top
and two side branches) drawn from a parent branch. You can specify the number of
generations in the code using the NUMBER_OF_BRANCH_GENERATIONS passed in the
constructor of TreeGenerator in the Main class. Example 4–3 shows the code in the
TreeGenerator class that creates the trunk of the tree (or the root branch) and adds
three branches for the following generations.
Example 4–3  Root Branch

private List<Branch> generateBranches(Branch parentBranch, int depth) {
    List<Branch> branches = new ArrayList<>();
    if (parentBranch == null) { // add root branch
        branches.add(new Branch());
    } else {
        if (parentBranch.length < 10) {
            return Collections.emptyList();
        }
        branches.add(new Branch(parentBranch, Type.LEFT, depth));
        branches.add(new Branch(parentBranch, Type.RIGHT, depth));
        branches.add(new Branch(parentBranch, Type.TOP, depth));
    }
    return branches;
}

To make the tree look more natural, each child generation branch is grown at an angle to the parent branch, and each child branch is smaller than its parent. The child angle is calculated using random values. Example 4–4 provides a code for creating child branches.

Example 4–4  Child Branches

public Branch(Branch parentBranch, Type type, int depth) {
    this();
    SimpleDoubleProperty locAngle = new SimpleDoubleProperty(0);
    globalAngle.bind(locAngle.add(parentBranch.globalAngle.get()));
    double transY = 0;
    switch (type) {
    case TOP:
        transY = parentBranch.length;
        length = parentBranch.length * 0.8;
        locAngle.set(getRandom(10));
        break;
    case LEFT:
    case RIGHT:
        locAngle.set(getGaussianRandom(35, 10) * (Type.LEFT == type ? 1 : -1));
        if ((0 > globalAngle.get() || globalAngle.get() > 180) && depth < 4) {
            length = parentBranch.length * getGaussianRandom(0.3, 0.1);
        } else {
            length = parentBranch.length * 0.6;
        }
        break;
    }
    setTranslateY(transY);
    getTransforms().add(new Rotate(locAngle.get(), 0, 0));
    globalH = getTranslateY() * cos(PI / 2 - parentBranch.globalAngle.get() * PI / 180) + parentBranch.globalH;
    setBranchStyle(depth);
    addChildToParent(parentBranch, this);
}
Leaves and Flowers

Leaves are created on top branches. Because the leaves are created at the same time as the branches of the tree, leaves are scaled to 0 by leaf.setScaleX(0) and leaf.setScaleY(0) to hide them before the tree is grown as shown in the Example 4–5. The same trick is used to hide the leaves when they fall. To create a more natural look, leaves have slightly different shades of green. Also, the leaf color changes depending on the location of the leaf; the darker shades are applied to the leaves located below the middle of the tree crown.

Example 4–5  Leaf Shape and Placement

public class Leaf extends Ellipse {
    public final Color AUTUMN_COLOR;
    private final int N = 5;
    private List<Ellipse> petals = new ArrayList<>(2 * N + 1);

    public Leaf(Branch parentBranch) {
        super(0, parentBranch.length / 2., 2, parentBranch.length / 2.);
        setScaleX(0);
        setScaleY(0);
        double rand = random() * 0.5 + 0.3;
        AUTUMN_COLOR = Color.color(random() * 0.1 + 0.8, rand, rand / 2);
        Color color = new Color(random() * 0.5, random() * 0.5 + 0.5, 0, 1);
        if (parentBranch.globalH < 400 && random() < 0.8) { //bottom leaf is darker
            color = color.darker();
        }
        setFill(color);
    }
}

Flowers are created in the Flower class and then added to the top branches of the tree in the TreeGenerator class. You can specify the number of petals in a flower. Petals are ellipses distributed in a circle with some overlapping. Similar to grass and leaves, the flower petals are colored in different shades of pink.

Animating Tree Elements

This section explains techniques employed in the Tree Animation example to animate the tree and season change. Parallel transition is used to start all the animations in the scene as shown in Example 4–6.

Example 4–6  Main Animation

final Transition all = new ParallelTransition(new GrassWindAnimation(grass),
    treeWindAnimation, new SequentialTransition(branchGrowingAnimation,
    seasonsAnimation(tree, grass)));
    all.play();

Growing a Tree

Tree growing animation is run only once, at the beginning of the Tree Animation example. The application starts a sequential transition animation to grow branches one generation after another as shown in Example 4–7. Initially length is set to 0. The root branch size and angle are specified in the TreeGenerator class. Currently each generation is grown during two seconds.
Example 4–7  Sequential Transition to Start Branch Growing Animation

SequentialTransition branchGrowingAnimation = new SequentialTransition();
The code in Example 4–8 creates the Tree growing animation:

Example 4–8  Branch Growing Animation

private Animation animateBranchGrowing(List<Branch> branchGeneration) {
    ParallelTransition sameDepthBranchAnimation = new ParallelTransition();
    for (final Branch branch : branchGeneration) {
        Timeline branchGrowingAnimation = new Timeline(new KeyFrame(duration,
            new KeyValue(branch.base.endYProperty(), branch.length)));
        PauseTransition pauseTransition = new PauseTransition();
        pauseTransition.setOnFinished(t ->
            branch.base.setStrokeWidth(branch.length / 25));
        sameDepthBranchAnimation.getChildren().add(
            new SequentialTransition(
                pauseTransition,
                branchGrowingAnimation));
    }
    return sameDepthBranchAnimation;
}

Because all the branch lines are calculated and created simultaneously, they could appear on the scene as dots. The code introduces a few tricks to hide the lines before they grow. In Example the code duration.one millisecond pauses transition for an unnoticeable time. In the Example 4–9, the branch.base.setStrokeWidth(0) code sets branches width to 0 before the grow animation starts for each generation.

Example 4–9  Tree Growing Animation Optimization

private void setBranchStyle(int depth) {
    base.setStroke(Color.color(0.4, 0.1, 0.1, 1));

    if (depth < 5) {
        base.setStrokeLineJoin(StrokeLineJoin.ROUND);
        base.setStrokeLineCap(StrokeLineCap.ROUND);
    }
    base.setStrokeWidth(0);
}

Creating Tree Crown Movement

In parallel with growing a tree, wind animation starts. Tree branches, leaves, and flowers are moving together.

Tree wind animation is similar to grass movement animation, but it is simpler because only the angle of the branches changes. To make the tree movement look natural, the bend angle is different for different branch generations. The higher the generation of the branch (that is the smaller the branch), the more it bends. Example 4–10 provides code for wind animation.

Example 4–10  Wind Animation

private Animation animateTreeWind(List<Branch> branchGeneration, int depth) {
    ParallelTransition wind = new ParallelTransition();
    for (final Branch branch : branchGeneration) {
        final Rotate rotation = new Rotate(0);
        branch.getTransforms().add(rotation);
    }
}
Timeline windTimeline = new Timeline(new KeyFrame(WIND_CYCLE_DURATION, 
new KeyValue(rotation.angleProperty(), depth * 2))); 
windTimeline.setAutoReverse(true); 
windTimeline.setCycleCount(Animation.INDEFINITE); 
wind.getChildren().add(windTimeline);
}
return wind;

### Animating Season Change

Season-change animation actually starts after the tree has grown, and runs infinitely. The code in **Example 4–11** calls all the season animations:

**Example 4–11  Starting Season Animation**

```java
private Transition seasonsAnimation(final Tree tree, final List<Blade> grass) {
    Transition spring = animateSpring(tree.leafage, grass);
    Transition flowers = animateFlowers(tree.flowers);
    Transition autumn = animateAutumn(tree.leafage, grass);

    SequentialTransition sequentialTransition = new SequentialTransition(spring, flowers, autumn);
    return sequentialTransition;
}
```

Once all the tree branches are grown, leaves start to appear as directed in **Example 4–12**.

**Example 4–12  Parallel Transition to Start Spring Animation and Show Leaves**

```java
private Transition animateSpring(List<Leaf> leafage, List<Blade> grass) {
    ParallelTransition springAnimation = new ParallelTransition();
    for (final Blade blade : grass) {
        springAnimation.getChildren().add(new FillTransition(GRASS_BECOME_GREEN_DURATION, 
        blade, 
        (Color) blade.getFill(), blade.SPRING_COLOR));
    }
    for (Leaf leaf : leafage) {
        ScaleTransition leafageAppear = new ScaleTransition(LEAF_APPEARING_DURATION, leaf);
        leafageAppear.setToX(1);
        leafageAppear.setToY(1);
        springAnimation.getChildren().add(leafageAppear);
    }
    return springAnimation;
}
```
When all leaves are visible, flowers start to appear as shown in Example 4–13. The sequential transition is used to show flowers gradually. The delay in flower appearance is set in the sequential transition code of Example 4–13. Flowers appear only in the tree crown.

**Example 4–13 Showing Flowers**

```java
private Transition animateFlowers(List<Flower> flowers) {
    ParallelTransition flowersAppearAndFallDown = new ParallelTransition();
    for (int i = 0; i < flowers.size(); i++) {
        final Flower flower = flowers.get(i);
        for (Ellipse pental : flower.getPetals()) {
            FadeTransition flowerAppear = new FadeTransition(FLOWER_APPEARING_DURATION, petal);
            flowerAppear.setToValue(1);
            flowerAppear.setDelay(FLOWER_APPEARING_DURATION.divide(3).multiply(i + 1));
            flowersAppearAndFallDown.getChildren().add(new SequentialTransition(new SequentialTransition(
                flowerAppear,
                fakeFallDownAnimation(petal))));
        }
    }
    return flowersAppearAndFallDown;
}
```

Once all the flowers appear on the screen, their petals start to fall. In the code in Example 4–14 the flowers are duplicated and the first set of them is hidden to show it later.

**Example 4–14 Duplicating Petals**

```java
private Ellipse copyEllipse(Ellipse petalOld, Color color) {
    Ellipse ellipse = new Ellipse();
    ellipse.setRadiusX(petalOld.getRadiusX());
    ellipse.setRadiusY(petalOld.getRadiusY());
    if (color == null) {
        ellipse.setFill(petalOld.getFill());
    } else {
        ellipse.setFill(color);
    }
    ellipse.setRotate(petalOld.getRotate());
    ellipse.setOpacity(0);
    return ellipse;
}
```

Copied flower petals start to fall to the ground one by one as shown in Example 4–15. The petals disappear after five seconds on the ground. The fall trajectory of a petal is not a straight line, but rather a calculated sine curve, so that petals seem to be whirling as they fall.

**Example 4–15 Shedding Flowers**

```java
Animation fakeLeafageDown = fakeFallDownEllipseAnimation(leaf, leaf.AUTUMN_COLOR,
```
node -> {
    node.setScaleX(0);
    node.setScaleY(0);
});

The next season change starts when all the flowers disappear from the scene. The leaves and grass become yellow, and the leaves fall and disappear. The same algorithm used in Example 4–15 to make the flower petals fall is used to show falling leaves. The code in Example 4–16 enables autumn animation.

**Example 4–16 Animating Autumn Changes**

```java
private Transition animateAutumn(List<Leaf> leafage, List<Blade> grass) {
    ParallelTransition autumn = new ParallelTransition();
    ParallelTransition yellowLeafage = new ParallelTransition();
    ParallelTransition dissappearLeafage = new ParallelTransition();

    for (final Leaf leaf : leafage) {
        final FillTransition toYellow =
            new FillTransition(LEAF_BECOME_YELLOW_DURATION, leaf, null, leaf.AUTUMN_COLOR);

        Animation fakeLeafageDown = fakeFallDownEllipseAnimation(leaf,
            leaf.AUTUMN_COLOR, node -> {
            node.setScaleX(0);
            node.setScaleY(0);
            });
        dissappearLeafage.getChildren().add(fakeLeafageDown);
    }

    ParallelTransition grassBecomeYellowAnimation = new ParallelTransition();
    for (final Blade blade : grass) {
        final FillTransition toYellow =
            new FillTransition(GRASS_BECOME_YELLOW_DURATION, blade, (Color) blade.getFill(),
                blade.AUTUMN_COLOR);
        toYellow.setDelay(Duration.seconds(1 * random()));
        grassBecomeYellowAnimation.getChildren().add(toYellow);
    }

    autumn.getChildren().addAll(grassBecomeYellowAnimation, new
        SequentialTransition(yellowLeafage, dissappearLeafage));
    return autumn;
}
```

After all leaves disappear from the ground, spring animation starts by coloring grass in green and showing leaves.

**Application Files**

**NetBeans Projects**

- tree_animation.zip
This tutorial contains the following topics:

- Blend Effect
- Bloom Effect
- Blur Effects
- Drop Shadow Effect
- Inner Shadow Effect
- Reflection
- Lighting Effect
- Perspective Effect
- Creating a Chain of Effects
This tutorial describes how to use visual effects to enhance the look of your JavaFX application.

All effects are located in the `javafx.scene.effect` package and are subclasses of the `Effect` class. For more information about particular classes, methods, or additional features, see the API documentation.

### Blend Effect

Blend is an effect that combines two inputs together using one of the predefined blending modes.

In the case of a node blending (`node.setBlendMode()`), the two inputs are:

- The node being rendered (a top input)
- Everything underneath the node (a bottom input)

The determination of the bottom input is based on the following rules:

- All lower Z-order siblings in the same group are included.
- If the group has a defined blend mode, then the process stops, and the bottom input is defined.
- If the group has the default blend mode, then everything underneath the group is included, recursively using this same rule.
- If the process recursively gets back to the root node, then the background paint of the scene is included.

---

**Note:** If the background paint of the scene, which is usually an opaque color, is included in the bottom input, then the `SRC_ATOP` mode renders on a completely opaque bottom source and has no effect. In this case, the `SRC_ATOP` mode is equivalent to `SRC_OVER`.

---

A blending mode defines the manner in which the objects are mixed together. For example, in Figure 5–1, you can see examples of some blending modes applied to a circle that is grouped with a square.
Bloom Effect

Example 5–1 shows a code snippet for the blend effect in the sample application.

Example 5–1  Blend Effect

```java
static Node blendMode() {
    Rectangle r = new Rectangle();
    r.setX(590);
    r.setY(50);
    r.setWidth(50);
    r.setHeight(50);
    r.setFill(Color.BLUE);

    Circle c = new Circle();
    c.setFill(Color.RED);
    c.setCenterX(590);
    c.setCenterY(50);
    c.setRadius(25);
    c.setBlendMode(BlendMode.SRC_ATOP);

    Group g = new Group();
    g.setBlendMode(BlendMode.SRC_OVER);
    g.getChildren().add(r);
    g.getChildren().add(c);
    return g;
}
```

Bloom Effect

The bloom effect makes brighter portions an image appear to glow, based on a configurable threshold. The threshold varies from 0.0 to 1.0. By default, the threshold is set to 0.3.

Figure 5–2 shows the bloom effect at the default threshold and at a threshold of 1.0.
Blur Effects

Figure 5–2  Bloom Effect

Example 5–2 shows a code snippet from the sample application that is using the bloom effect.

Example 5–2  Bloom Example

```java
static Node bloom() {
    Group g = new Group();
    Rectangle r = new Rectangle();
    r.setX(10);
    r.setY(10);
    r.setWidth(160);
    r.setHeight(80);
    r.setFill(Color.DARKBLUE);
    Text t = new Text();
    t.setText("Bloom!*");
    t.setFill(Color.YELLOW);
    t.setFont(Font.font("null", FontWeight.BOLD, 36));
    t.setX(25);
    t.setY(65);
    g.setCache(true);
    g.getChildren().add(r);
    g.getChildren().add(t);
    g.setTranslateX(350);
    g.setEffect(new Bloom());
    Bloom bloom = new Bloom();
    bloom.setThreshold(1.0);
    g.setEffect(bloom);
    return g;
}
```

Blur Effects

Blurring are common effects that can be used to provide more focus to selected objects. With JavaFX you can apply a boxblur, a motion blur, or a gaussian blur.

BoxBlur

The BoxBlur is a blur effect that uses a simple box filter kernel, with separately configurable sizes in both dimensions that control the amount of blur applied to an object, and an iterations parameter that controls the quality of the resulting blur.

Figure 5–3 shows two samples of blurred text.
**Blur Effects**

**Figure 5–3  BoxBlur Effect**

![BoxBlur Effect](image)

Example 5–3 is a code snippet that uses the BoxBlur effect.

**Example 5–3  BoxBlur Example**

```java
static Node boxBlur() {
    Text t = new Text();
    t.setText("Blurry Text!");
    t.setFill(Color.RED);
    t.setFont(Font.font("null", FontWeight.BOLD, 36));
    t.setX(10);
    t.setY(40);
    BoxBlur bb = new BoxBlur();
    bb.setWidth(5);
    bb.setHeight(5);
    bb.setIterations(3);
    t.setEffect(bb);
    t.setTranslateX(300);
    t.setTranslateY(100);
    return t;
}
```

**Motion Blur**

A motion blur effect uses a Gaussian blur, with a configurable radius and angle to create the effect of a moving object.

**Figure 5–4** shows the effect of the motion blur on a text.

**Figure 5–4  Motion Blur Effect**

![Motion Blur](image)

Example 5–4 shows a code snippet that creates a motion blur effect with radius set to 15 and angle set to 45 in the sample application.

**Example 5–4  Motion Blur Example**

```java
static Node motionBlur() {
    Text t = new Text();
    t.setX(20.0f);
    t.setY(80.0f);
    t.setText("Motion Blur");
    t.setFill(Color.RED);
    t.setFont(Font.font("null", FontWeight.BOLD, 60));
    t.setTranslateX(300);
    t.setTranslateY(100);
    return t;
}
```
Gaussian Blur

The Gaussian blur is an effect that uses a Gaussian algorithm with a configurable radius to blur objects. 

Figure 5–5 shows the effect of the Gaussian blur on a text.

Example 5–5 shows a code snippet that blurs the text using Gaussian blur effect.

```
Example 5–5 Gaussian Blur

    static Node gaussianBlur() {
        Text t2 = new Text();
        t2.setX(10.0f);
        t2.setY(140.0f);
        t2.setCache(true);
        t2.setText("Gaussian Blur");
        t2.setFill(Color.RED);
        t2.setFont(Font.font("null", FontWeight.BOLD, 36));
        t2.setEffect(new GaussianBlur());
        return t2;
    }
```

Drop Shadow Effect

A drop shadow is an effect that renders a shadow of the content to which it is applied. You can specify the color, the radius, the offset, and some other parameters of the shadow.

Figure 5–6 shows the shadow effect on different objects.
Example 5–6 shows how to create a drop shadow on text and a circle.

Example 5–6  Text and Circle With Shadows

```java
import javafx.collections.ObservableList;
import javafx.application.Application;
import javafx.scene.*;
import javafx.stage.*;
import javafx.scene.shape.*;
import javafx.scene.effect.*;
import javafx.scene.paint.*;
import javafx.scene.text.*;

public class HelloEffects extends Application {
    Stage stage;
    Scene scene;

    @Override
    public void start(Stage stage) {
        stage.show();

        scene = new Scene(new Group(), 840, 680);
        ObservableList<Node> content = ((Group)scene.getRoot()).getChildren();

        content.add(dropShadow());
        stage.setScene(scene);
    }

    static Node dropShadow() {  
        Group g = new Group();

        DropShadow ds = new DropShadow();
        ds.setOffsetY(3.0);
        ds.setOffsetX(3.0);
        ds.setColor(Color.GRAY);

        Text t = new Text();
        t.setEffect(ds);
        t.setCache(true);
        t.setX(20.0f);
        t.setY(70.0f);
        t.setFill(Color.RED);
        t.setText("JavaFX drop shadow effect");
        t.setFont(Font.font("null", FontWeight.BOLD, 32));

        DropShadow ds1 = new DropShadow();
        ds1.setOffsetY(4.0f);
        ds1.setOffsetX(4.0f);
        ds1.setColor(Color.CORAL);

        Circle c = new Circle();
        c.setEffect(ds1);
        c.setCenterX(50.0f);
        c.setCenterY(325.0f);
        c.setRadius(30.0f);
        c.setFill(Color.RED);
        c.setCache(true);

        g.getChildren().add(t);
    }
}
```

Inner Shadow Effect

An inner shadow is an effect that renders a shadow inside the edges of the given content with the specified color, radius, and offset.

Figure 5–7 shows plain text and the same text with the inner shadow effect applied.

Example 5–7 shows how to create an inner shadow on text.

Example 5–7  Inner Shadow

c:
static Node innerShadow() {
    InnerShadow is = new InnerShadow();
    is.setOffsetX(2.0f);
    is.setOffsetY(2.0f);

    Text t = new Text();
    t.setEffect(is);
    t.setX(20);
    t.setY(100);
    t.setText("Inner Shadow");
    t.setFill(Color.RED);
    t.setFont(Font.font("null", FontWeight.BOLD, 80));

    t.setTranslateX(300);
    t.setTranslateY(300);
}
Reflection

Reflection is an effect that renders a reflected version of the object below the actual object.

---

**Note:** The reflection of a node with a reflection effect will not respond to mouse events or the containment methods on the node.

---

Figure 5–8 shows a reflection applied to text. Use the `setFraction` method to specify the amount of visible reflection.

**Figure 5–8 Reflection Effect**

Reflection in JavaFX...

Example 5–8 shows how to create the reflection effect on text.

**Example 5–8 Text With Reflection**

```java
import javafx.scene.text.*;
import javafx.scene.paint.*;
import javafx.scene.effect.*;
public class HelloEffects extends Application {

    Stage stage;
    Scene scene;

    @Override public void start(Stage stage) {
        stage.show();

        scene = new Scene(new Group(), 840, 680);
        ObservableList<Node> content = ((Group)scene.getRoot()).getChildren();
        content.add(reflection());
        stage.setScene(scene);
    }

    static Node reflection() {
        Text t = new Text();
        t.setX(10.0f);
        t.setY(50.0f);
        t.setCache(true);
        t.setText("Reflection in JavaFX...");
        t.setFill(Color.RED);
        t.setFont(Font.font("null", FontWeight.BOLD, 30));
        Reflection r = new Reflection();
        r.setFraction(0.9);
        t.setEffect(r);
        t.setTranslateY(400);
        return t;
    }
}
```

**Note:** The reflection of a node with a reflection effect will not respond to mouse events or the containment methods on the node.
Lighting Effect

The lighting effect simulates a light source shining on the given content, which can be used to give flat objects a more realistic three-dimensional appearance.

Figure 5–9 shows the lighting effect on text.

Figure 5–9 Lighting Effect

JavaFX Lighting!

Example 5–9 shows how to create a lighting effect on text.

Example 5–9 Text with Applied Lighting Effect

import javafx.application.Application;
import javafx.collections.ObservableList;
import javafx.geometry.VPos;
import javafx.scene.effect.Light.Distant;
import javafx.scene.*;
import javafx.stage.*;
import javafx.scene.shape.*;
import javafx.scene.effect.*;
import javafx.scene.paint.*;
import javafx.scene.text.*;

public class HelloEffects extends Application {
    Stage stage;
    Scene scene;

    @Override public void start(Stage stage) {
        stage.show();

        scene = new Scene(new Group());
        ObservableList<Node> content = ((Group)scene.getRoot()).getChildren();

        content.add(lighting());
        stage.setScene(scene);
    }
}

static Node lighting() {
    Distant light = new Distant();
    light.setAzimuth(-135.0f);
    Lighting l = new Lighting();
    l.setLight(light);
    l.setSurfaceScale(5.0f);
    return l;
}
Text t = new Text();
t.setText("JavaFX\n"+"Lighting!");
t.setFill(Color.RED);
t.setFont(Font.font('null', FontWeight.BOLD, 70));
t.setTranslateX(10.0f);
t.setTranslateY(10.0f);
t.setTextOrigin(VPos.TOP);

t.setEffect(l);

t.setTranslateX(0);
t.setTranslateY(320);

return t;
}

public static void main(String[] args) {
    Application.launch(args);
}

Perspective Effect

The perspective effect creates a three-dimensional effect of otherwise two-dimensional object.

Figure 5–10 shows the perspective effect.

Figure 5–10  Perspective Effect

A perspective transformation can map any square to another square, while preserving the straightness of the lines. Unlike affine transformations, the parallelism of lines in the source is not necessarily preserved in the output.

Note: This effect does not adjust the coordinates of input events or any methods that measure containment on a node. Mouse clicking and the containment methods are undefined if a perspective effect is applied to a node.

Example 5–10 is a code snippet from the sample application that shows how to create a perspective effect.

Example 5–10  Perspective Effect

static Node perspective() {
    Group g = new Group();
    PerspectiveTransform pt = new PerspectiveTransform();
    pt.setUlx(10.0f);
    pt.setUly(10.0f);
    pt.setUrx(210.0f);
    pt.setUry(40.0f);
    g.setEffect(pt);
    return g;
}
pt.setLrx(210.0f);
pt.setLry(60.0f);
pt.setLlx(10.0f);
pt.setLly(90.0f);
g.setEffect(pt);
g.setCache(true);

Rectangle r = new Rectangle();
r.setX(10.0f);
r.setY(10.0f);
r.setWidth(280.0f);
r.setHeight(80.0f);
r.setFill(Color.DARKBLUE);

Text t = new Text();
t.setX(20.0f);
t.setY(65.0f);
t.setText("Perspective");
t.setFill(Color.RED);
t.setFont(Font.font("null", FontWeight.BOLD, 36));
g.getChildren().add(r);
g.getChildren().add(t);
return g;
}

Figure 5–11 shows which coordinates affect the resulting image.

**Figure 5–11  Coordinates for Perspective Effect**

Creating a Chain of Effects

Some of the effects have an input property that you can use to create a chain of effects. The chain of effects can be a tree-like structure, because some effects have two inputs and some do not have any.

In Figure 5–12 the reflection effect is used as an input for the drop shadow effect, which means that first the rectangle is reflected by the reflection effect and then the drop shadow effect is applied to the result.
import javafx.application.Application;
import javafx.collections.ObservableList;
import javafx.scene.*;
import javafx.stage.*;
import javafx.scene.shape.*;
import javafx.scene.effect.*;
import javafx.scene.paint.*;
import javafx.scene.text.*;

public class HelloEffects extends Application {

    Stage stage;
    Scene scene;

    @Override
    public void start(Stage stage) {
        stage.show();

        scene = new Scene(new Group());
        ObservableList<Node> content = ((Group)scene.getRoot()).getChildren();

        content.add(chainEffects());
        stage.setScene(scene);
    }

    static Node chainEffects() {
        Rectangle rect = new Rectangle();
        rect.setFill(Color.RED);
        rect.setWidth(200);
        rect.setHeight(100);
        rect.setX(20.0f);
        rect.setY(20.0f);

        DropShadow ds = new DropShadow();
        ds.setOffsetY(5.0);
        ds.setOffsetX(5.0);
        ds.setColor(Color.GRAY);
        ds.setInput(reflection);
        rect.setEffect(ds);

        return rect;
    }
}
public static void main(String[] args) {
    Application.launch(args);
}

---

**Note:** If you change the last two lines in the `chainEffects()` method to
`reflection.setInput(ds);` and `rect.setEffect(reflection);`, first the drop shadow will be applied to the rectangle, and then the result will be reflected by the reflection effect.

---

For more information about particular classes, methods, or additional features, see the API documentation.

**Application Files**

**NetBeans Projects**

- `visual_effects.zip`
Source Code for the Transformations, Animations, and Visual Effects Tutorial

The following table lists the demo applications in this document with their associated source code files.

<table>
<thead>
<tr>
<th>Tutorial</th>
<th>Source Code</th>
<th>NetBeans Project File</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transformations Overview</td>
<td>Xylophone.java</td>
<td>transformations.zip</td>
</tr>
<tr>
<td>Animation Basics</td>
<td></td>
<td>animations.zip</td>
</tr>
<tr>
<td>Tree Animation Example</td>
<td></td>
<td>tree_animation.zip</td>
</tr>
<tr>
<td>Creating Visual Effects</td>
<td></td>
<td>visual_effects.zip</td>
</tr>
</tbody>
</table>
For a description, see Transformation Types and Examples.

Legal Terms and Copyright Notice

/*
 * Copyright (c) 2010, 2014, Oracle and/or its affiliates.
 * All rights reserved. Use is subject to license terms.
 *
 * This file is available and licensed under the following license:
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 *
 * - Redistributions of source code must retain the above copyright
 *   notice, this list of conditions and the following disclaimer.
 * - Redistributions in binary form must reproduce the above copyright
 *   notice, this list of conditions and the following disclaimer in
 *   the documentation and/or other materials provided with the distribution.
 * - Neither the name of Oracle nor the names of its
 *   contributors may be used to endorse or promote products derived
 *   from this software without specific prior written permission.
 *
 * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
 * 'AS IS' AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
 * LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
 * A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT
 * OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
 * SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
 * LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
 * DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
 * THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
 * (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
 * OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
 */

Code

package xylophone;

import javafx.application.Application;
import javafx.event.EventHandler;
import javafx.geometry.Bounds;
import javafx.scene.DepthTest;
import javafx.scene.Group;
import javafx.scene.PerspectiveCamera;
import javafx.scene.Scene;
import javafx.scene.input.KeyCode;
import javafx.scene.input.KeyEvent;
import javafx.scene.input.MouseEvent;
import javafx.scene.media.AudioClip;
import javafx.scene.paint.Color;
import javafx.scene.paint.CycleMethod;
import javafx.scene.paint.RadialGradient;
import javafx.scene.paint.Stop;
import javafx.scene.shape.Rectangle;
import javafx.scene.transform.Rotate;
import javafx.scene.transform.Scale;
import javafx.scene.transform.Shear;
import javafx.scene.transform.Translate;
import javafx.stage.Stage;

public class Xylophone extends Application {
    double mousePosX;
    double mousePosY;
    double mouseOldX;
    double mouseOldY;
    double mouseDeltaX;
    double mouseDeltaY;

    final Cam camOffset = new Cam();
    final Cam cam = new Cam();

    final Shear shear = new Shear();

    class Cam extends Group {
        Translate t = new Translate();
        Translate p = new Translate();
        Translate ip = new Translate();
        Rotate rx = new Rotate();
        { rx.setAxis(Rotate.X_AXIS); }
        Rotate ry = new Rotate();
        { ry.setAxis(Rotate.Y_AXIS); }
        Rotate rz = new Rotate();
        { rz.setAxis(Rotate.Z_AXIS); }
        Scale s = new Scale();

        public Cam() { super(); getTransforms().addAll(t, p, rx, rz, ry, s, ip); }
    }

    @Override public void start(final Stage stage) {
        stage.setTitle("Xylophone");

        camOffset.getChildren().add(cam);
        resetCam();

        final Scene scene = new Scene(camOffset, 800, 600, true);
        scene.setFill(new RadialGradient(225, 0.85, 300, 300, 500, false,
                                         CycleMethod.NO_CYCLE, new Stop[]
                                         { new Stop(0f, Color.BLUE),
                                          new Stop(1f, Color.LIGHTBLUE) }));

        scene.setCamera(new PerspectiveCamera());

        final AudioClip bar1Note =
        new AudioClip(Xylophone.class.getResource("audio/Note1.wav").toString());
    }
}
final AudioClip bar2Note = new AudioClip(Xylophone.class.getResource("audio/Note2.wav").toString());
final AudioClip bar3Note = new AudioClip(Xylophone.class.getResource("audio/Note3.wav").toString());
final AudioClip bar4Note = new AudioClip(Xylophone.class.getResource("audio/Note4.wav").toString());
final AudioClip bar5Note = new AudioClip(Xylophone.class.getResource("audio/Note5.wav").toString());
final AudioClip bar6Note = new AudioClip(Xylophone.class.getResource("audio/Note6.wav").toString());
final AudioClip bar7Note = new AudioClip(Xylophone.class.getResource("audio/Note7.wav").toString());
final AudioClip bar8Note = new AudioClip(Xylophone.class.getResource("audio/Note8.wav").toString());

Group rectangleGroup = new Group();
rectangleGroup.getTransforms().add(shear);
rectangleGroup.setDepthTest(DepthTest.ENABLE);

double xStart = 260.0;
double xOffset = 30.0;
double yPos = 300.0;
double zPos = 0.0;
double barWidth = 22.0;
double barDepth = 7.0;

// Base1
Cube base1Cube = new Cube(1.0, new Color(0.2, 0.12, 0.1, 1.0), 1.0);
base1Cube.setTranslateX(xStart + 135);
base1Cube.setTranslateZ(yPos+20.0);
base1Cube.setTranslateY(11.0);
base1Cube.setScaleX(barWidth*11.5);
base1Cube.setScaleZ(10.0);
base1Cube.setScaleY(barDepth*2.0);

// Base2
Cube base2Cube = new Cube(1.0, new Color(0.2, 0.12, 0.1, 1.0), 1.0);
base2Cube.setTranslateX(xStart + 135);
base2Cube.setTranslateZ(yPos-20.0);
base2Cube.setTranslateY(11.0);
base2Cube.setScaleX(barWidth*11.5);
base2Cube.setScaleZ(10.0);
base2Cube.setScaleY(barDepth*2.0);

// Bar1
Cube bar1Cube = new Cube(1.0, Color.PURPLE, 1.0);
bar1Cube.setTranslateX(xStart + 1*xOffset);
bar1Cube.setTranslateZ(yPos);
bar1Cube.setScaleX(barWidth);
bar1Cube.setScaleZ(100.0);
bar1Cube.setScaleY(barDepth);

// Bar2
Cube bar2Cube = new Cube(1.0, Color.BLUEVIOLET, 1.0);
bar2Cube.setTranslateX(xStart + 2*xOffset);
bar2Cube.setTranslateY(yPos);
bar2Cube.setScaleX(barWidth);
bar2Cube.setScaleZ(95.0);
bar2Cube.setScaleY(barDepth);

// Bar3
Cube bar3Cube = new Cube(1.0, Color.BLUE, 1.0);
bar3Cube.setTranslateX(xStart + 3*xOffset);
bar3Cube.setTranslateY(yPos);
bar3Cube.setScaleX(barWidth);
bar3Cube.setScaleZ(90.0);
bar3Cube.setScaleY(barDepth);

// Bar4
Cube bar4Cube = new Cube(1.0, Color.GREEN, 1.0);
bar4Cube.setTranslateX(xStart + 4*xOffset);
bar4Cube.setTranslateY(yPos);
bar4Cube.setScaleX(barWidth);
bar4Cube.setScaleZ(85.0);
bar4Cube.setScaleY(barDepth);

// Bar5
Cube bar5Cube = new Cube(1.0, Color.GREENYELLOW, 1.0);
bar5Cube.setTranslateX(xStart + 5*xOffset);
bar5Cube.setTranslateY(yPos);
bar5Cube.setScaleX(barWidth);
bar5Cube.setScaleZ(80.0);
bar5Cube.setScaleY(barDepth);

// Bar6
Cube bar6Cube = new Cube(1.0, Color.YELLOW, 1.0);
bar6Cube.setTranslateX(xStart + 6*xOffset);
bar6Cube.setTranslateY(yPos);
bar6Cube.setScaleX(barWidth);
bar6Cube.setScaleZ(75.0);
bar6Cube.setScaleY(barDepth);

// Bar7
Cube bar7Cube = new Cube(1.0, Color.ORANGE, 1.0);
bar7Cube.setTranslateX(xStart + 7*xOffset);
bar7Cube.setTranslateY(yPos);
bar7Cube.setScaleX(barWidth);
bar7Cube.setScaleZ(70.0);
bar7Cube.setScaleY(barDepth);

// Bar8
Cube bar8Cube = new Cube(1.0, Color.RED, 1.0);
bar8Cube.setTranslateX(xStart + 8*xOffset);
bar8Cube.setTranslateY(yPos);
bar8Cube.setScaleX(barWidth);
bar8Cube.setScaleZ(65.0);
bar8Cube.setScaleY(barDepth);

bar1Cube.setOnMousePressed(new EventHandler<MouseEvent>() {
    @Override
    public void handle(MouseEvent me) {
        bar1Note.play();
    }
});
bar2Cube.setOnMousePressed(new EventHandler<MouseEvent>() {
public void handle(MouseEvent me) { bar2Note.play(); }
}));
bar3Cube.setOnMousePressed(new EventHandler<MouseEvent>() {
    @Override
    public void handle(MouseEvent me) { bar3Note.play(); }
});
bar4Cube.setOnMousePressed(new EventHandler<MouseEvent>() {
    @Override
    public void handle(MouseEvent me) { bar4Note.play(); }
});
bar5Cube.setOnMousePressed(new EventHandler<MouseEvent>() {
    @Override
    public void handle(MouseEvent me) { bar5Note.play(); }
});
bar6Cube.setOnMousePressed(new EventHandler<MouseEvent>() {
    @Override
    public void handle(MouseEvent me) { bar6Note.play(); }
});
bar7Cube.setOnMousePressed(new EventHandler<MouseEvent>() {
    @Override
    public void handle(MouseEvent me) { bar7Note.play(); }
});
bar8Cube.setOnMousePressed(new EventHandler<MouseEvent>() {
    @Override
    public void handle(MouseEvent me) { bar8Note.play(); }
});

rectangleGroup.getChildren().addAll(base1Cube, base2Cube,
   bar1Cube, bar2Cube, bar3Cube,
   bar4Cube, bar5Cube, bar6Cube,
   bar7Cube, bar8Cube);

rectangleGroup.setScaleX(2.5);
rectangleGroup.setScaleY(2.5);
rectangleGroup.setScaleZ(2.5);
cam.getChildren().add(rectangleGroup);

scene.setOnMousePressed(new EventHandler<MouseEvent>() {
    @Override
    public void handle(MouseEvent me) {
        mousePosX = me.getX();
        mousePosY = me.getY();
        mouseOldX = me.getX();
        mouseOldY = me.getY();
        //System.out.println("scene.setOnMousePressed " + me);
    }
});

scene.setOnMouseDragged(new EventHandler<MouseEvent>() {
    @Override
    public void handle(MouseEvent me) {
        mouseOldX = mousePosX;
        mouseOldY = mousePosY;
        mouseOldX = me.getX();
        mouseOldY = me.getY();
        mouseDeltaX = mousePosX - mouseOldX;
        mouseDeltaY = mousePosY - mouseOldY;
        if (me.isAltDown() && me.isShiftDown() &&
            me.isPrimaryButtonDown()) {
double rzAngle = cam.rz.getAngle();
cam.rz.setAngle(rzAngle - mouseDeltaX);
}
else if (me.isAltDown() && me.isPrimaryButtonDown()) {
    double ryAngle = cam.ry.getAngle();
cam.ry.setAngle(ryAngle - mouseDeltaX);
    double rxAngle = cam.rx.getAngle();
cam.rx.setAngle(rxAngle + mouseDeltaY);
}
else if (me.isShiftDown() && me.isPrimaryButtonDown()) {
    double yShear = shear.getY();
shear.setY(yShear + mouseDeltaY/1000.0);
    double xShear = shear.getX();
shear.setX(xShear + mouseDeltaX/1000.0);
}
else if (me.isAltDown() && me.isSecondaryButtonDown()) {
    double scale = cam.s.getX();
    double newScale = scale + mouseDeltaX*0.01;
cam.s.setX(newScale);
cam.s.setY(newScale);
cam.s.setZ(newScale);
}
else if (me.isAltDown() && me.isMiddleButtonDown()) {
    double tx = cam.t.getX();
    double ty = cam.t.getY();
cam.t.setX(tx + mouseDeltaX);
cam.t.setY(ty + mouseDeltaY);
}
});
}
scene.setOnKeyPressed(new EventHandler<KeyEvent>() {
    @Override
    public void handle(KeyEvent ke) {
        if (KeyCode.A.equals(ke.getCode())) {
            resetCam();
shear.setX(0.0);
shear.setY(0.0);
        }
        if (KeyCode.F.equals(ke.getCode())) {
            frameCam(stage, scene);
shear.setX(0.0);
shear.setY(0.0);
        }
        if (KeyCode.SPACE.equals(ke.getCode())) {
            if (stage.isFullScreen()) {
                stage.setFullScreen(false);
                frameCam(stage, scene);
            } else {
                stage.setFullScreen(true);
                frameCam(stage, scene);
            }
        }
    }
});
}
stage.setScene(scene);
stage.show();

//=================================================================
public void frameCam(final Stage stage, final Scene scene) {
    setCamOffset(camOffset, scene);
    setCamPivot(cam);
    setCamTranslate(cam);
    setCamScale(cam, scene);
}

public void setCamOffset(final Cam camOffset, final Scene scene) {
    double width = scene.getWidth();
    double height = scene.getHeight();
    camOffset.t.setX(width/2.0);
    camOffset.t.setY(height/2.0);
}

public void setCamScale(final Cam cam, final Scene scene) {
    final Bounds bounds = cam.getBoundsInLocal();
    final double pivotX = bounds.getMinX() + bounds.getWidth()/2;
    final double pivotY = bounds.getMinY() + bounds.getHeight()/2;
    final double pivotZ = bounds.getMinZ() + bounds.getDepth()/2;
    double width = scene.getWidth();
    double height = scene.getHeight();
    double scaleFactor = 1.0;
    double scaleFactorY = 1.0;
    double scaleFactorX = 1.0;
    if (bounds.getWidth() > 0.0001) {
        scaleFactorX = width / bounds.getWidth(); // / 2.0;
    }
    if (bounds.getHeight() > 0.0001) {
        scaleFactorY = height / bounds.getHeight(); // / 1.5;
    }
    if (scaleFactorX > scaleFactorY) {
        scaleFactor = scaleFactorY;
    } else {
        scaleFactor = scaleFactorX;
    }
    cam.s.setX(scaleFactor);
    cam.s.setY(scaleFactor);
    cam.s.setZ(scaleFactor);
}

public void setCamPivot(final Cam cam) {
    final Bounds bounds = cam.getBoundsInLocal();
    final double pivotX = bounds.getMinX() + bounds.getWidth()/2;
    final double pivotY = bounds.getMinY() + bounds.getHeight()/2;
    final double pivotZ = bounds.getMinZ() + bounds.getDepth()/2;
    cam.p.setX(pivotX);
}

cam.p.setY(pivotY);
cam.p.setZ(pivotZ);
cam.ip.setX(-pivotX);
cam.ip.setY(-pivotY);
cam.ip.setZ(-pivotZ);
}

//========================================================================
// setCamTranslate
//========================================================================
public void setCamTranslate(final Cam cam) {
    final Bounds bounds = cam.getBoundsInLocal();
    final double pivotX = bounds.getMinX() + bounds.getWidth()/2;
    final double pivotY = bounds.getMinY() + bounds.getHeight()/2;
    cam.t.setX(-pivotX);
cam.t.setY(-pivotY);
}

public void resetCam() {
    cam.t.setX(0.0);
cam.t.setY(0.0);
cam.t.setZ(0.0);
cam.rx.setAngle(45.0);
cam.ry.setAngle(-7.0);
cam.rz.setAngle(0.0);
cam.s.setX(1.25);
cam.s.setY(1.25);
cam.s.setZ(1.25);
}

public class Cube extends Group {
    final Rotate rx = new Rotate(0, Rotate.X_AXIS);
    final Rotate ry = new Rotate(0, Rotate.Y_AXIS);
    final Rotate rz = new Rotate(0, Rotate.Z_AXIS);
    public Cube(double size, Color color, double shade) {
        getTransforms().addAll(rz, ry, rx);

            back face
Rectangle backFace = new Rectangle(size, size);
backFace.setFill(color.deriveColor(0.0, 1.0, (1 - 0.5*shade), 1.0));
backFace.setTranslateX(-0.5*size);
backFace.setTranslateY(-0.5*size);
backFace.setTranslateZ(-0.5*size);

// bottom face
Rectangle bottomFace = new Rectangle(size, size);
bottomFace.setFill(color.deriveColor(0.0, 1.0, (1 - 0.4*shade), 1.0));
bottomFace.setTranslateX(-0.5*size);
bottomFace.setTranslateY(0);
bottomFace.setTranslateZ(-0.5*size);
bottomFace.setRotationAxis(Rotate.X_AXIS);
bottomFace.setRotate(90);

// right face
Rectangle rightFace = new Rectangle(size, size);
rightFace.setFill(color.deriveColor(0.0, 1.0, (1 - 0.3*shade), 1.0));
rightFace.setTranslateX(-1*size);
rightFace.setTranslateY(-0.5*size);
rightFace.setTranslateZ(-0.5*size);
rightFace.setRotationAxis(Rotate.Y_AXIS);
rightFace.setRotate(90);

// leftFace
Rectangle leftFace = new Rectangle(size, size);
leftFace.setFill(color.deriveColor(0.0, 1.0, (1 - 0.2*shade), 1.0));
leftFace.setTranslateX(0);
leftFace.setTranslateY(-0.5*size);
leftFace.setTranslateZ(-0.5*size);
leftFace.setRotationAxis(Rotate.Y_AXIS);
leftFace.setRotate(90);

// topFace
Rectangle topFace = new Rectangle(size, size);
topFace.setFill(color.deriveColor(0.0, 1.0, (1 - 0.1*shade), 1.0));
topFace.setTranslateX(-0.5*size);
topFace.setTranslateY(-1*size);
topFace.setTranslateZ(-0.5*size);
topFace.setRotationAxis(Rotate.X_AXIS);
topFace.setRotate(90);

// frontFace
Rectangle frontFace = new Rectangle(size, size);
frontFace.setFill(color);
frontFace.setTranslateX(-0.5*size);
frontFace.setTranslateY(-0.5*size);
frontFace.setTranslateZ(-0.5*size);

getChildren().addAll(backFace, bottomFace, rightFace, leftFace, topFace, frontFace);

public static void main(String[] args) {
    Application.launch(args);
}