

JavaFX

Getting Started with JavaFX 3D Graphics

Release 8.0 Developer Preview

E20484-01

September 2013

Beta Draft

Getting Started with JavaFX 3D Graphics, Release 8.0 Developer Preview

E20484-01

Copyright © 2013 Oracle and/or its affiliates. All rights reserved.

Primary Author: Cindy Castillo

Contributing Author: John Yoon

Contributor:

This software and related documentation are provided under a license agreement containing restrictions on use and disclosure and are protected by intellectual property laws. Except as expressly permitted in your license agreement or allowed by law, you may not use, copy, reproduce, translate, broadcast, modify, license, transmit, distribute, exhibit, perform, publish, or display any part, in any form, or by any means. Reverse engineering, disassembly, or decompilation of this software, unless required by law for interoperability, is prohibited.

The information contained herein is subject to change without notice and is not warranted to be error-free. If you find any errors, please report them to us in writing.

If this software or related documentation is delivered to the U.S. Government or anyone licensing it on behalf of the U.S. Government, the following notice is applicable:

U.S. GOVERNMENT RIGHTS Programs, software, databases, and related documentation and technical data delivered to U.S. Government customers are “commercial computer software” or “commercial technical data” pursuant to the applicable Federal Acquisition Regulation and agency-specific supplemental regulations. As such, the use, duplication, disclosure, modification, and adaptation shall be subject to the restrictions and license terms set forth in the applicable Government contract, and, to the extent applicable by the terms of the Government contract, the additional rights set forth in FAR 52.227-19, Commercial Computer Software License (December 2007). Oracle USA, Inc., 500 Oracle Parkway, Redwood City, CA 94065.

This software is developed for general use in a variety of information management applications. It is not developed or intended for use in any inherently dangerous applications, including applications which may create a risk of personal injury. If you use this software in dangerous applications, then you shall be responsible to take all appropriate fail-safe, backup, redundancy, and other measures to ensure the safe use of this software. Oracle Corporation and its affiliates disclaim any liability for any damages caused by use of this software in dangerous applications.

Oracle is a registered trademark of Oracle Corporation and/or its affiliates. Other names may be trademarks of their respective owners.

This software and documentation may provide access to or information on content, products, and services from third parties. Oracle Corporation and its affiliates are not responsible for and expressly disclaim all warranties of any kind with respect to third-party content, products, and services. Oracle Corporation and its affiliates will not be responsible for any loss, costs, or damages incurred due to your access to or use of third-party content, products, or services.

This documentation is in prerelease status and is intended for demonstration and preliminary use only. It may not be specific to the hardware on which you are using the software. Oracle Corporation and its affiliates are not responsible for and expressly disclaim all warranties of any kind with respect to this documentation and will not be responsible for any loss, costs, or damages incurred due to the use of this documentation.

Contents

1 Overview	
Sample of 3D Graphics Use Cases	1-1
3D Feature in JavaFX 2.x Releases.....	1-2
2 3D Shapes	
Pre-defined Shapes	2-1
User-defined Shapes.....	2-2
3 Camera	
Perspective Camera.....	3-1
Field of View	3-2
Clipping Planes	3-2
Y-down versus Y-up.....	3-3
4 Subscene	
Creating a SubScene.....	4-1
5 Lights	
6 Materials	
7 Picking	
Creating a PickResult Object	7-1
Methods for the PickResult Object.....	7-1
8 Building a 3D Sample Application	
Create the Project.....	8-1
Create the Scene.....	8-2
Set Up the Camera.....	8-3
Build the Axes	8-4
Build the Molecule.....	8-6
Add Camera Viewing Controls.....	8-7

Part I

About This Tutorial

This tutorial contains information about the JavaFX 3D graphics functionality available in JavaFX 8.

It is assumed that you have an intermediate level of Java and JavaFX knowledge.

Download JDK 8 Developer Preview release from

<http://jdk8.java.net/download.html>.

This document contains the following chapters that discusses the available 3D features and also steps you through building a sample application using some of those features:

- [Overview](#)
- [3D Shapes](#)
- [Camera](#)
- [Subscene](#)
- [Lights](#)
- [Materials](#)
- [Picking](#)
- [Building a 3D Sample Application](#)

This chapter provides an overview of the JavaFX 3D graphics features currently available through the Java APIs for JavaFX.

The JavaFX 3D graphics APIs provide a general purpose three-dimensional graphics library for the JavaFX platform. You can use 3D geometry, cameras, and lights to create, display, and manipulate objects in 3D space.

Sample of 3D Graphics Use Cases

Figure 1-1 shows a snapshot of the JavaFX 3D application example that was demonstrated at the JavaOne 2012 keynote session. It was built as a proof of concept on an early prototype of JavaFX SDK with added 3D Mesh, Camera and Lighting support. You can view it at <http://www.youtube.com/embed/AS26gZrYNy8?rel=0> web site.

Figure 1-1 JavaFX 3D Application Sample



A sampling of other JavaFX 3D graphics use cases are as follows:

- Inventory and Process Visualization
- Scientific and Engineering Visualization
- 3D Charting
- Mechanical CAD and CAE
- Medical Imaging

- Product Marketing
- Architectural Design and Walkthroughs
- Advanced User Experience
- Mission Planning
- Training
- Entertainment

3D Feature in JavaFX 2.x Releases

In the JavaFX 2.x releases, it is possible to create two-dimensional objects and transform them in 3D space. You can subclass the `Group` class to create your own custom group and set the transform sub-matrices to anything you want. You are able to simulate the behavior of transform groups of other 3D content concentration packages, such as Maya and 3D Studio Max, because you can customize which sub-matrices are part of that transform group. See [Applying Transformations in JavaFX](#) to learn more about this transformation feature.

[Example 1-1](#) shows a sample code that creates a `Group` subclass, `Xform`, that has a translation, a pivot, three rotations, a scale, and the inverse pivot.

Example 1-1 3D Transforms Code Sample

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

The `Xform` subclass is created from `Group` because groups are originally designed for two-dimensional (2D) UI layout. The pivot of a node is recalculated under certain conditions for 2D UI layout, but if you subclass group and create `Xform`, as shown in [Example 1-1](#) and use those new transforms, it bypasses the 2D UI layout.

Although, 2D UI pivot recalculation is very desirable for UI controls in a 2D layout, it is not something you would want in a 3D layout. The pivot point is recomputed as the center of the Node's layout bounds and so any change to the layout bounds will cause the pivot point to change, which ends up automatically moving your object. So, for a `Group` node, any change to its children, including position, geometry, effect, orientation, or scale, will cause the group's layout bounds to change. This will automatically move the object in unintended ways, when it comes to 3D layout, but in desirable ways when it comes to 2D. So, in a 3D layout, you definitely want to bypass the automatic pivot recomputation.

Some of the useful 3D Transform methods on `Node` are listed in [Example 1-2](#).

Example 1–2 Useful 3D Transform Methods on Node

```
Transform getLocalToParentTransform()
Transform getLocalToSceneTransform()
public Point3D sceneToLocal(Point3D scenePoint)
public Point3D sceneToLocal(double sceneX, double sceneY, double sceneZ)
public Point3D localToScene(Point3D localPoint)
public Point3D localToScene(double x, double y, double z)
public Point3D parentToLocal(Point3D parentPoint)
public Point3D parentToLocal(double parentX, double parentY, double parentZ)
public Point3D localToParent(Point3D localPoint)
public Point3D localToParent(double x, double y, double z)
```

3D Shapes

This chapter gives information about the Shapes3D API that is available with the JavaFX 3D Graphics library.

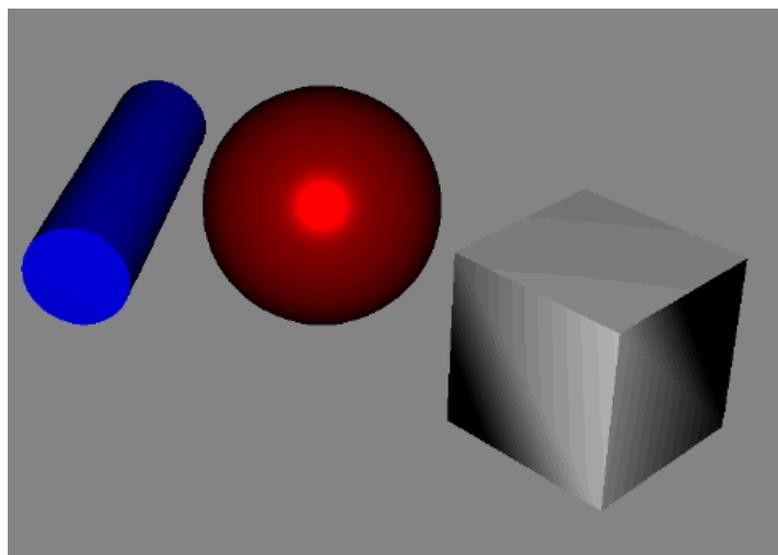
There are two types of 3D shapes:

- [Pre-defined Shapes](#)
- [User-defined Shapes](#)

Pre-defined Shapes

The pre-defined 3D shapes are provided to make it easier for you to quickly create 3D objects out-of-the-box. They include boxes, cylinders, and spheres, whose sample usages are shown in [Figure 2-1](#).

Figure 2-1 *Pre-defined Shapes*



[Example 2-1](#) shows the Shape3D class hierarchy. It includes the MeshView class, which defines a surface with the specified 3D mesh data. Also included are the Box, Cylinder, and Sphere subclasses.

Example 2-1 *Shape3D Class Hierarchy*

```
java.lang.Object
```

```
javafx.scene.Node
    javafx.scene.shape.Shape3D
        javafx.scene.shape.MeshView
        javafx.scene.shape.Box
        javafx.scene.shape.Cylinder
        javafx.scene.shape.Sphere
```

Use the following information to create the pre-defined shapes:

- To create a Box, specify the dimensions of width, height, and depth.

```
Box myBox = new Box(width, height, depth);
```

- To create a cylinder, specify the radius and height.

```
Cylinder myCylinder = new Cylinder(radius, height);
Cylinder myCylinder2 = new Cylinder(radius, height, divisions);
```

- To create a sphere, specify the radius.

```
Sphere mySphere = new Sphere(radius);
Sphere mySphere2 = new Sphere(radius, divisions);
```

User-defined Shapes

[Example 2-2](#) shows the JavaFX Mesh class hierarchy, which contains the `TriangleMesh` subclass. Triangle mesh is the most typical kind of mesh used in 3D layouts.

Example 2-2 Mesh Class Hierarchy

```
java.lang.Object
    javafx.scene.shape.Mesh (abstract)
        javafx.scene.shape.TriangleMesh
```

The `TriangleMesh` contains separate arrays of points, texture coordinates, and faces that describe a triangulated geometric mesh. Use the following steps to create a `TriangleMesh` instance:

1. Create a new instance of a `triangleMesh`.

```
mesh = new TriangleMesh();
```

2. Define the set of points, which are the vertices of the mesh.

```
float points[] = { ... };
mesh.getPoints().addAll(points);
```

3. Describe the texture coordinates for each vertex.

```
float texCoords[] = { ... };
mesh.getTexCoords().addAll(texCoords);
```

4. Using the vertices, build the Faces, which are triangles that describe the topology.

```
int faces[] = { ... };
mesh.getFaces().addAll(faces);
```

5. Define the smoothing group to which each face belongs.

```
int smoothingGroups[] = { ... };
mesh.getFaceSmoothingGroups().addAll(smoothingGroups);
```

Smoothing group adjusts the normal on the vertices for the face to either be smooth or faceted. If every single face has a differing smoothing group, then the mesh will be very faceted. If every single face has the same smoothing group, then the mesh will look very smooth.

This chapter describes the Camera API that is included with the JavaFX 3D Graphics feature.

The camera is now a node and can be added to the JavaFX scene graph. It allows you to move the camera around in a 3D UI layout. This is different from the 2D layout where you did not need to move the camera around.

In JavaFX, the camera coordinate system is as follows:

- X-axis pointing to the right
- Y-axis pointing down
- Z-axis pointing away from the viewer or into the screen.

To create a camera and add it to the scene, use the following lines of code:

```
Camera camera = new PerspectiveCamera(true);
scene.setCamera(camera);
```

Use the following code to add a camera to the scene graph.

```
Group cameraGroup = new Group();
cameraGroup.getChildren().add(camera);
root.getChildren().add(cameraGroup);
```

To rotate the camera and move the cameraGroup, use

```
camera.rotate(45);
cameraGroup.setTranslateZ(-75);
```

Perspective Camera

JavaFX provides a perspective camera for rendering a 3D scene. This camera defines a viewing volume for a perspective projection. The viewing volume can be changed by changing the value of the `fieldOfView`.

[Example 3-1](#) shows the two constructors for creating a Perspective Camera:

Example 3-1 Constructors for PerspectiveCamera

```
PerspectiveCamera()

PerspectiveCamera(boolean fixedEyeAtCameraZero)
```

The latter constructor is a new constructor in JavaFX 8 and allows you to control the camera position with the specified `fixedEyeAtCameraZero` flag so that it renders what the camera would see in a 3D environment.

The following constructor should be used for 3D graphics programming:

```
PerspectiveCamera(true);
```

When the option `fixedEyeAtCameraZero` is set to `true`, a `PerspectiveCamera` is constructed with its eye position fixed at $(0, 0, 0)$, in its coordinate space, regardless of the change in the dimension of the projection area or window resize.

When the `fixedEyeAtCameraZero` is set to the default value of `false`, the coordinate system defined by the camera has its origin in the upper left corner of the panel. This mode is used for 2D UI controls rendered with a perspective camera, but is not useful for most 3D graphics applications. The camera is moved when the window is resized, for example, to maintain the origin in the upper left corner of the panel. That is exactly what you want for a 2D UI layout, but not in a 3D layout. So, it is important to remember to set the `fixedEyePosition` to `true` when you are doing 3D graphics.

Field of View

The camera's field of view can be set as follows:

```
camera.setFieldOfView(double value);
```

The larger the field of view, the more perspective distortion and size differences increase.

- *Fisheye* lenses have a field of view of up to 180 degrees and beyond.
- *Normal* lenses have a field of view between 40 and 62 degrees.
- *Telephoto* lenses have a field of view of 1 (or less) degrees to 30 degrees.

Clipping Planes

You can set the near clipping plane of the Camera in the local coordinate system as follows:

```
camera.setNearClip(double value);
```

To set the far clipping plane of the Camera in the local coordinate system, use the following:

```
camera.setFarClip(double value);
```

Setting the near or far clipping planes determines the viewing volume. If the near clipping plane is too great, it will basically start clipping the front of the scene. If it is too small, then it will start clipping the back of the scene.

Tip: Don't set the near clipping value to a smaller value than is needed or the far clipping value to a larger value than is needed because strange visual artifacts may start appearing.

The clipping planes need to be set so that enough of the scene can be seen. But the viewing range should not be set so large that a numerical error is encountered. If the near clipping plane is too large a value, the scene starts getting clipped. But if the near clipping plane too small, a different kind of visual artifact will appear due to a value

being too close to zero. If the far clipping plane is too large a value, a numerical error is also encountered, especially if the near clipping plane is too small a value.

Y-down versus Y-up

Most 2D graphics coordinate systems (including UI) have Y increasing as you go down the screen. This is true of PhotoShop, JavaFX, and Illustrator. Basically most 2D packages work this way. Many 3D graphics coordinate systems typically have Y increasing as you move up the screen. Some 3D graphics coordinate systems have Z increasing as you move up, but most have Y increasing as you move up the screen.

Y down versus Y up are both correct in their own context. In JavaFX, the camera's coordinate system is Y-down, which means X axis points to the right, Y axis is pointing down, Z axis is pointing away from the viewer or into the screen.

If you want to think of a 3D scene as Y-up, you could create an Xform node, called `root3D`, under `root`, as shown in [Example 3-2](#). You set its `rx.setAngle` property to 180 degrees, basically turning it upside down. Then, add your 3D elements to your `root3D` node and put your camera under `root3D`.

Example 3-2 Create Xform node, root3D

```
root3D = new Xform();
root3D.rx.setAngle(180.0);
root.getChildren().add(root3D);
root3D.getChildren().add(...); // add all your 3D nodes here
```

You can also create an Xform node called `cameraXform` and put it under the `root`, as shown in [Example 3-3](#). You turn it upside down, and put your camera under the `cameraXform`.

Example 3-3 Create a cameraXform Node

```
Camera camera = new PerspectiveCamera(true);
Xform cameraXform = new Xform();
root.getChildren().add(cameraXform);
cameraXform.getChildren().add(camera);
cameraXform.rz.setAngle(180.0);
```

An even better way, with subtle difference on the camera node, is to add a 180 degree rotation to the camera. The rotation used is not the one provided for you because you want to avoid the auto pivoting. In [Example 3-4](#), the camera is turned 180 degrees and it is then added to the camera as a child of `cameraXform`. The subtle difference is that the `cameraXform` retains very pristine values and in its default position, everything is zeroed out, including the translations and rotations.

Example 3-4 Create cameraXform and Rotate

```
Camera camera = new PerspectiveCamera(true);
Xform cameraXform = new Xform();
root.getChildren().add(cameraXform);
cameraXform.getChildren().add(camera);
Rotate rz = new Rotate(180.0, Rotate.Z_AXIS);
camera.getTransforms().add(rz);
```


This chapter gives information about the use of Subscenes in JavaFX.

The SubScene node is a container for content in a scene graph. It is a special Node for scene separation. It can be used to render part of the scene with a different camera. You can use a SubScene node if you want to have Y-up for 3D objects and Y-down for 2D UI objects in your layout.

Some of the possible SubScene use cases are:

- overlay for UI controls (needs a static camera)
- Underlay for background (static or updated less frequently)
- “Heads-up” display
- Y-up for your 3D objects and Y-down for your 2D UI.

Creating a SubScene

[Example 4-1](#) shows the two constructors for creating a new instance of a SubScene node in your application.

Example 4-1 SubScene Constructors

```
// Creates a SubScene for a specific root Node with a specific size.  
SubScene(Parent root, double width, double height, boolean depthBuffer,  
         boolean antiAliasing)
```

```
// Constructs a SubScene consisting of a root, with a dimension of width and  
// height, specifies whether a depth buffer is created for this scene and  
// specifies whether scene anti-aliasing is requested.  
SubScene(Parent root, double width, double height)
```

Once you have created a SubScene, you can modify it by using the available methods to specify or obtain the height, root node, width, background fill of the SubScene, the type of camera used to render the SubScene, or whether the SubScene is anti-aliased. See the JavaFX 8 API docs for more information.

This chapter describes the Light API included in the JavaFX 3D graphics library.

Light is now also defined as a node in the scene graph. A default light is provided if the set of active lights contained in the scene is empty. Each light contains a set of affected nodes. If a set of nodes is empty, all nodes on the scene (or subscene) are affected. If a Parent node is in that set of nodes, then all of its children are also affected.

Light interacts with the geometry of a Shape3D object and its material to provide rendering result. Currently, there are two types of light sources:

- `AmbientLight` - a light source that seems to come from all directions.
- `PointLight` - an attenuated light source that has a fixed point in space and radiates light equally in all directions away from itself.

[Example 5-1](#) shows the Light Class Hierarchy.

Example 5-1 Light Class Hierarchy

```
javafx.scene.Node
  javafx.scene.LightBase (abstract)
    javafx.scene.AmbientLight
    javafx.scene.PointLight
```

To create point light and add it to the Scene, do the following:

```
PointLight light = new PointLight();
light.setColor(Color.RED);
```

Use the following to add light to the scene graph:

```
Group lightGroup = new Group();
lightGroup.getChildren().add(light);
root.getChildren().add(lightGroup);
```

Rotate the light 45 degrees with the following line of code:

```
light.rotate(45);
```

To move the lightGroup and have light moves with it, use something similar to the following code.

```
lightGroup.setTranslateZ(-75);
```

The `setTranslateZ()` method sets the value of the property `translateZ`, which is set to -75 in the example code above. This value will be added to any translation defined by the transforms `ObservableList` and `layoutZ`.

This chapter describes the Material class of the JavaFX 3D Graphics library.

Materials contains a set of rendering properties. [Example 6-1](#) shows the Material class hierarchy and that the PhongMaterial is sub-classed from the Material class.

Example 6-1 Material Class Hierarchy

```
java.lang.Object
  javafx.scene.paint.Material (abstract)
    javafx.scene.paint.PhongMaterial
```

PhongMaterial class provides definitions of properties represent a form of Phong shaded material:

- Diffuse color
- Diffuse map
- Specular map
- Specular color
- Specular power
- Bump map or normal map
- Self-illumination map

Materials are shareable among multiple Shape3D nodes.

[Example 6-2](#) shows how to create a PhongMaterial, set its diffuseMap properties, and use the material for a shape.

Example 6-2 Working with Material

```
//Create Material
Material mat = new PhongMaterial();
Image diffuseMap = new Image("diffuseMap.png");
Image normalMap = new Image("normalMap.png");

// Set material properties
mat.setDiffuseMap(diffuseMap);
mat.setBumpMap(normalMap);
mat.setSpecularColor(Color.WHITE);

// Use the material for a shape
shape3d.setMaterial(mat);
```


This chapter describes the `PickResult` API that is included with the JavaFX 3D Graphics feature.

The `PickResult` API was already available for 2D primitives with the Perspective Camera. However, there were existing limitations when it was used with depth buffer so the `PickResult` class has been added to the `javafx.scene.input` package. It is a container object that contains the result of a pick event.

The `PickResult` argument has been added to all the constructors of the `MouseEvent`, `MouseEvent`, `MouseEvent`, `MouseEvent`, `MouseEvent` and `TouchPoint` classes so that information about the user's pick is returned. The newly added `getPickResult()` method in these classes returns a new `PickResult` object that contains information about the pick. Another method added is `getZ()` which returns the depth position of the event relative to the origin of the `MouseEvent`'s source.

Creating a `PickResult` Object

The JavaFX API provides three constructors for creating a new instance of a `PickResult` object in your application, as shown in [Example 7-1](#).

Example 7-1 *PickResult Constructors*

```
// Creates a pick result for a 2D case where no additional information
// is needed. Converts the given scene coordinates to the target's local
// coordinate space and stores the value as the intersected point. Sets
// intersected node to the given target, distance to 1.0, face to
// FACE_UNDEFINED and texCoord to null
PickResult(EventTarget target, double sceneX, double sceneY)

// Creates a new instance of PickResult for a non-3d-shape target. Sets face
// to FACE_UNDEFINED and texCoord to null.
PickResult(Node node, Point3D point, double distance)

// Creates a new instance of PickResult
PickResult(Node node, Point3D point, double distance, int face,
           Point2D texCoord)
```

Methods for the `PickResult` Object

Once you have created a `PickResult` object in your code, you can use the following methods to work with the information passed from the classes that handle the events.

Example 7-2 PickResult Methods

```
// Returns the intersected node. Returns null if there was no intersection
// with any node and the scene was picked.
public final Node getIntersectedNode()

// Returns the intersected point in local coordinate of the picked Node. If
// no node was picked, it returns the intersected point with the projection
// plane.
public final Point3D getIntersectedPoint()

// Returns the intersected distance between camera position and the
// intersected point
public final double getIntersectedDistance()

// Returns the intersected face of the picked Node, FACE_UNDEFINED if the
// node doesn't have user-specified faces or was picked on bounds.
public final int getIntersectedFace()

// Return the intersected texture coordinates of the picked 3d shape. If the
// picked target is not Shape3D or has pickOnBounds==true, it returns null.
// Returns new Point2D presenting the intersected TexCoord
public final Point2D getIntersectedTexCoord()
```

Building a 3D Sample Application

This chapter provides the steps to build a simple application, `MoleculeSampleApp`, that uses some of the JavaFX 3D graphics features that were discussed earlier in this document.

The steps in this tutorial chapter use the NetBeans 4 IDE to help you develop the `MoleculeSampleApp` application.

The following files are included for this tutorial:

- `MoleculeSampleApp.zip` - the completed NetBeans project for the `MoleculeSampleApp` application.
- `Xform.java` - method that declares the `Xform` class.
- `buildMolecule()` - method that creates the 3D water molecule object.
- `handleMouse()` and `handleKeyboard()` - methods that allow you to use the mouse and keyboard to manipulate the camera's view in the scene.

This tutorial contains the following sections.

- [Create the Project](#)
- [Create the Scene](#)
- [Set Up the Camera](#)
- [Build the Axes](#)
- [Build the Molecule](#)
- [Add Camera Viewing Controls](#)

Create the Project

Use the NetBeans IDE to create the JavaFX project, `MoleculeSampleApp`.

1. From the **File** menu, choose **New Project**.
2. In the New Project wizard, choose the **JavaFX** application category and **JavaFX Application** project. Click **Next**.
3. Type **MoleculeSampleApp** for the Project Name. Enter the path for the **Project Location** text field or click **Browse** to navigate to the folder you want to use for this project.
4. Click **Finish**.

When you create a JavaFX project, NetBeans IDE provides a Hello World source code template as a starting point. You will replace that template source code in the next sections.

Create the Scene

Create the scene that will hold your molecule UI layout.

1. First, download the `Xform.java` file and save it to the `moleculesampleapp` source folder of the `moleculesampleapp` project.

This file contains the source code for the `Xform` sub-class that is derived from the `Group` class. Using the `Xform` node prevents the automatic recalculation of the position of a group node's pivot when the children of the group node is changed in a 3D UI layout. The `Xform` node allows you to add your own types of transforms and rotation. The file contains a translation component, three rotation components, and a scale component. Having the three rotation components is helpful when changing rotation values frequently, such as in changing the angle of the camera in the scene.

2. If not already opened in the IDE editor, open the `MoleculeSampleApp.java` file that was created with the project creation. Replace the import statements at the top of the file with the import statements shown in [Example 8-1](#).

Example 8-1 Replacement Import Statements

```
import javafx.application.Application;
import static javafx.application.Application.launch;
import javafx.scene.Group;
import javafx.scene.Scene;
import javafx.scene.paint.Color;
import javafx.stage.Stage;
import moleculesampleapp.Xform;
```

3. Replace the rest of the body of the code in `MoleculeSampleApp.java` with the lines of code shown in [Example 8-2](#). The code creates a new scene graph with an `Xform` as its node.

Example 8-2 Replacement Body of Code

```
/**
 * MoleculeSampleApp
 */
public class MoleculeSampleApp extends Application {

    final Group root = new Group();
    final Xform world = new Xform();

    private void buildScene() {
        System.out.println("buildScene");
        root.getChildren().add(world);
    }

    @Override
    public void start(Stage primaryStage) {
        System.out.println("start");
        buildScene();

        Scene scene = new Scene(root, 1024, 768, true);
        scene.setFill(Color.GREY);
    }
}
```

```

        primaryStage.setTitle("Molecule Sample Application");
        primaryStage.setScene(scene);
        primaryStage.show();
    }

    /**
     * The main() method is ignored in correctly deployed JavaFX application.
     * main() serves only as fallback in case the application can not be
     * launched through deployment artifacts, e.g., in IDEs with limited FX
     * support. NetBeans ignores main().
     *
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        System.setProperty("prism.dirtyopts", "false");
        launch(args);
    }
}

```

4. Press **Ctrl+S** to save the file.

Set Up the Camera

Set up the camera in a hierarchy of Group class with Xform instances. Perform the translation and rotation on the camera to change its default location.

1. At the top of the MoleculeSampleApp.java file, add the following import statement for the Camera class.

Example 8-3 Import the Perspective

```
import javafx.scene.PerspectiveCamera;
```

2. Add the following lines of code, shown in bold below, so that it appears after the declaration statement for the world object, as shown in [Example 8-4](#).

These lines of code create an instance of a perspectiveCamera and three instances of the public class Xform, which extends the Group class. The Xform class is defined in the Xform.java file you added to your NetBeans project in the previous section of this document.

Example 8-4 Add Variables for the Camera

```

final Group root = new Group();
final Xform world = new Xform();
final PerspectiveCamera camera = new PerspectiveCamera(true);
final Xform cameraXform = new Xform();
final Xform cameraXform2 = new Xform();
final Xform cameraXform3 = new Xform();
final double cameraDistance = 450;

```

3. Copy the lines of code for the buildCamera() method, shown in [Example 8-5](#). Add them right after the lines for the buildScene() method.

The buildCamera() method sets the camera to have the view upside down instead of the default JavaFX 2D Y-down. So the scene is viewed as a Y-up (Y-axis pointing up) scene.

Example 8–5 Add the buildCamera() Method

```
private void buildCamera() {
    root.getChildren().add(cameraXform);
    cameraXform.getChildren().add(cameraXform2);
    cameraXform2.getChildren().add(cameraXform3);
    cameraXform3.getChildren().add(camera);
    cameraXform3.setRotateZ(180.0);

    camera.setNearClip(0.1);
    camera.setFarClip(10000.0);
    camera.setTranslateZ(-cameraDistance);
    cameraXform.ry.setAngle(320.0);
    cameraXform.rx.setAngle(40);
}
```

4. In the `start()` method, add the call to the `buildCamera()` so that it appears as shown in bold in [Example 8–6](#)

Example 8–6 Add Method Call to buildCamera()

```
buildScene();
buildCamera();
```

5. Set the camera in the scene by copying the line of code shown in bold in [Example 8–7](#) and adding it to the end of the `start()` method.

Example 8–7 Set the Camera in the Scene

```
primaryStage.show();
scene.setCamera(camera);
```

6. Save the file with Ctrl+S.

Build the Axes

Add the 3D axes that you will use to build this molecule. The `Box` class is used to create the axes and the `PhongMaterial` is used to set the specular and diffused colors. By default in JavaFX, the Y-axis is down.

Per the usual convention, the X-axis is shown in the color red, Y-axis is shown in green, and Z-axis in blue.

1. Copy the following declaration shown in bold in [Example 8–8](#) and add it to just after the line where `root` is declared.

Example 8–8 Create the axisGroup

```
final Group root = new Group();
final Group axisGroup = new Group();
```

2. Add the `buildAxes()` method shown in [Example 8–9](#) to after the `buildCamera()` method.

Example 8–9 Add buildAxes() Method

```
private void buildAxes() {
    System.out.println("buildAxes()");
    final PhongMaterial redMaterial = new PhongMaterial();
    redMaterial.setDiffuseColor(Color.DARKRED);
    redMaterial.setSpecularColor(Color.RED);
}
```

```

final PhongMaterial greenMaterial = new PhongMaterial();
greenMaterial.setDiffuseColor(Color.DARKGREEN);
greenMaterial.setSpecularColor(Color.GREEN);

final PhongMaterial blueMaterial = new PhongMaterial();
blueMaterial.setDiffuseColor(Color.DARKBLUE);
blueMaterial.setSpecularColor(Color.BLUE);

final Box xAxis = new Box(240.0, 1, 1);
final Box yAxis = new Box(1, 240.0, 1);
final Box zAxis = new Box(1, 1, 240.0);

xAxis.setMaterial(redMaterial);
yAxis.setMaterial(greenMaterial);
zAxis.setMaterial(blueMaterial);

axisGroup.getChildren().addAll(xAxis, yAxis, zAxis);
world.getChildren().addAll(axisGroup);
}

```

3. Fix the imports by adding the following import statements shown in [Example 8–10](#).

Example 8–10 Add Two Import Statements

```

import javafx.scene.paint.PhongMaterial;
import javafx.scene.shape.Box;

```

4. Add the call to `buildAxes()` method, as shown in bold in [Example 8–11](#).

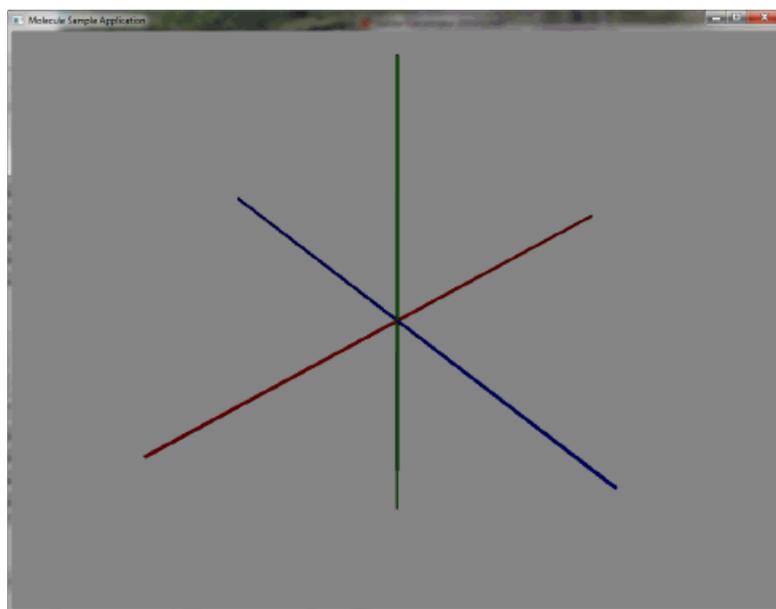
Example 8–11 Add Call to `buildAxes()` Method

```

buildCamera();
buildAxes();

```

5. Compile and run the project by right-clicking the **MoleculeSampleApp** node in the project window and choose **Run**. A window appears with the 3D axes, as shown in [Figure 8–1](#).

Figure 8–1 The 3D Axes

Build the Molecule

In this section, you build the molecule UI. This is where you use the 3D features such as PhongMaterial, Sphere, and Cylinder. The Xform class is used

1. To declare the moleculeGroup Xform, copy the line of code shown in bold in [Example 8–12](#). Paste it after the cameraDistance variable.

Example 8–12 Declare the moleculeGroup Xform

```
final double cameraDistance = 450;
final Xform moleculeGroup = new Xform();
```

2. Add the following import statements for the classes used in the buildMolecule() method:

Example 8–13 Add Import Statements for build Molecule()

```
import javafx.scene.shape.Cylinder;
import javafx.scene.shape.Sphere;
import javafx.scene.transform.Rotate;
```

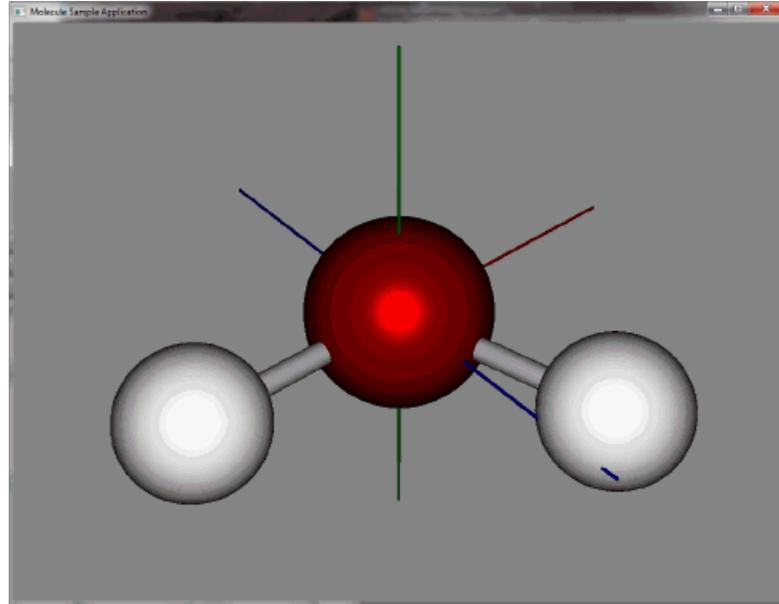
3. Click the buildMolecule() link on the right-hand column of this tutorial to open the buildMolecule.html file. Copy the body of code for buildMolecule() and paste it after the buildAxes() method in the MoleculeSampleApp.java file.
4. In the start() method, add the call to the buildMolecule() method so that it appears as shown in bold in [Example 8–14](#).

Example 8–14 Add the Call to the buildMolecule() Method

```
buildCamera();
buildAxes();
buildMolecule();
```

- Now run the project and you should see something similar to [Figure 8-2](#).

Figure 8-2 *The Water Molecule 3DModel*



Add Camera Viewing Controls

The `handleMouse()` and `handleKeyboard()` methods allow you to see the different camera views. The source has been provided for you to demonstrate the use of the mouse and the keyboard to

- Add the import statement for the `javafx.animation.Timeline` class to the top of the file.
- Add the declaration for variables that are used in the `handleMouse()` and `handleKeyboard()` source code you are about to add. Copy the code shown in [Example 8-15](#) and paste after the line for the `moleculeGroup` declaration.

Example 8-15 Add Variables Used

```
private Timeline timeline;
boolean timelinePlaying = false;
double ONE_FRAME = 1.0/24.0;
double DELTA_MULTIPLIER = 200.0;
double CONTROL_MULTIPLIER = 0.1;
double SHIFT_MULTIPLIER = 0.1;
double ALT_MULTIPLIER = 0.5;

double mousePosX;
double mousePosY;
double mouseOldX;
double mouseOldY;
double mouseDeltaX;
double mouseDeltaY;
```

- Click the `handleCameraViews` link on the right-hand side bar and open the file.

4. Copy the import statements used in the `handleMouse()` and `handleKeyboard()` methods, as shown in [Example 8–16](#). Paste them at the top of the `MoleculeSampleApp.java` file.

Example 8–16 Add the Import Statements

```
import javafx.event.EventHandler;
import static javafx.scene.input.KeyCode.*;
import javafx.scene.input.KeyEvent;
import javafx.scene.input.MouseEvent;
import javafx.util.Duration;
import javafx.scene.Node;
```

5. Copy the lines of code for the `handleMouse()` and the `handleKeyboard()` methods in the `handleCameraViews.html` file. Add them after the `buildMolecule()` method in the `MoleculeSampleApp.java` file.
6. In the `start()` method, add the calls to the `handleKeyboard()` and `handleMouse()` methods that you just added. Copy the lines of code shown in bold in [Example 8–17](#) and paste them after the `scene.setFill(Color.GREY)` line.

Example 8–17 Add Method Calls

```
Scene scene = new Scene(root, 1024, 768, true);
scene.setFill(Color.GREY);
handleKeyboard(scene, world);
handleMouse(scene, world);
```

7. Save the file.
8. Compile and run the project. Use the following mouse or keyboard strokes to get the different views.
 - Hold the left mouse button and drag the mouse right or left and up or down to rotate the camera view of the molecule around the axes.
 - Hold the right mouse button and drag the mouse to the left to move camera view away from the water molecule model. Drag the mouse to the right to move the camera view closer to the molecule model.
 - `Ctrl+s` is used for showing and hiding the molecule.
 - `Ctrl+x` is used for showing and hiding the axes.