# Proscenium Tutorials

This document is to motivate the reader to try tutorials and quickly be able to use Proscenium. Please take a quick look at this document and then start examining tutorial codes. Tutorial codes can be found in the ProsceniumSamples\src folder. These samples can be compiled and run once the Proscenium project is setup. To setup the project, please read the setup instruction.

We will go through tutorials to learn how to create objects and add them to the scene to display them. We will learn how to change material, move objects, and turn on shadows. We also provide a sample (Tutorial05) that does not use our demo/test harness class. For these, we provide five tutorials:

|  |  |
| --- | --- |
| Tutorial01\_SimpleShapes.as | Creates simple shapes and render them. |
| Turorial02\_AnimationAndMaterial.as | Instances mesh, moves objects, and changes colors. |
| Tutorial03\_Shadows.as | Add shadows |
| Tutorial04\_LoadingModels.as | Load models from disk |
| Tutorial05\_SpriteBased.as | A sample application that does not use BasicDemo test harness.  All tutorials, except for Tutorial05\_SpriteBased, use BasicDemo class. BasicDemo creates three lights by default and provides a key event handler. |

## World Coordinate System and Camera

The y-axis is up. By default, the camera looks at -z direction, so x-axis points to the right side of the screen.

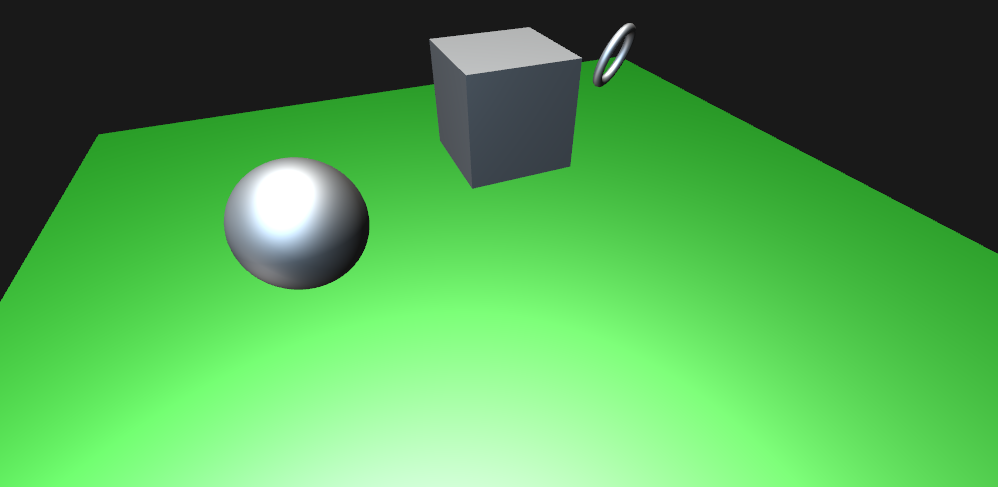
x

y

z

# Tutorial01\_SimpleShapes

First tutorial is to create shapes and add them to the scene. This will make objects appear.



The code is very simple: just to create a plane, a cube, a sphere, and a torus, and then add them to the scene.

Basic shapes can be created using ProceduralGeometry class. See the code below.

***MaterialStandard*** is the material class that implements variety of rendering models. In this tutorial, a green-colored material is created and used as the material for the plane.

***SceneMesh*** is a basic scene graph node that contains mesh. (Other types of scene graph nodes are *SceneLight*, *SceneCamera*, *SceneSkyBox* and others. All node class names start with ‘*Scene’*).

*BasicDemo.scene* is the root node of the scene graph. ***BasicDemo*** automatically creates a few *SceneLight* objects, and provides key event handler codes that enable a camera control.

package

{

import com.adobe.scenegraph.\*;

import flash.display.\*;

import flash.display3D.\*;

import flash.geom.\*;

public class Tutorial01\_SimpleShapes extends BasicDemo

{

public function Tutorial01\_SimpleShapes()

{

super();

}

override protected function initModels():void

{

*// create plane material*

**var** material:MaterialStandard = **new** MaterialStandard();

material.diffuseColor.set( 0, .4, 0 );

material.specularColor.set( .8, .8, .8 );

material.ambientColor.set( .2, .2, .2 );

*// create a plane and add it to the scene*

var plane:SceneMesh = ProceduralGeometry.createPlane( 50, 50, 20, 20,

null, "plane" );

plane.appendTranslation( 0, -2, 0 );

scene.addChild( plane );

*// create a cube and add it to the scene*

var cube:SceneMesh = ProceduralGeometry.createCube( 5 );

cube.appendTranslation( 0, 6, 0 );

scene.addChild( cube );

*// create a torus and add it to the scene*

var torus:SceneMesh = ProceduralGeometry.createDonut( .25, 1.5, 50, 10,

null, "torus" );

torus.appendTranslation( 10, 2, 0 );

var rotAxis:Vector3D = new Vector3D( 1,1,1 );

rotAxis.normalize();

torus.appendRotation( 45, rotAxis );

scene.addChild( torus );

*// create a sphere and add it to the scene*

var sphere:SceneMesh = ProceduralGeometry.createSphere( 3, 50, 50,

null, "sphere" );

sphere.setPosition( -10, 2, 0 );

scene.addChild( sphere );

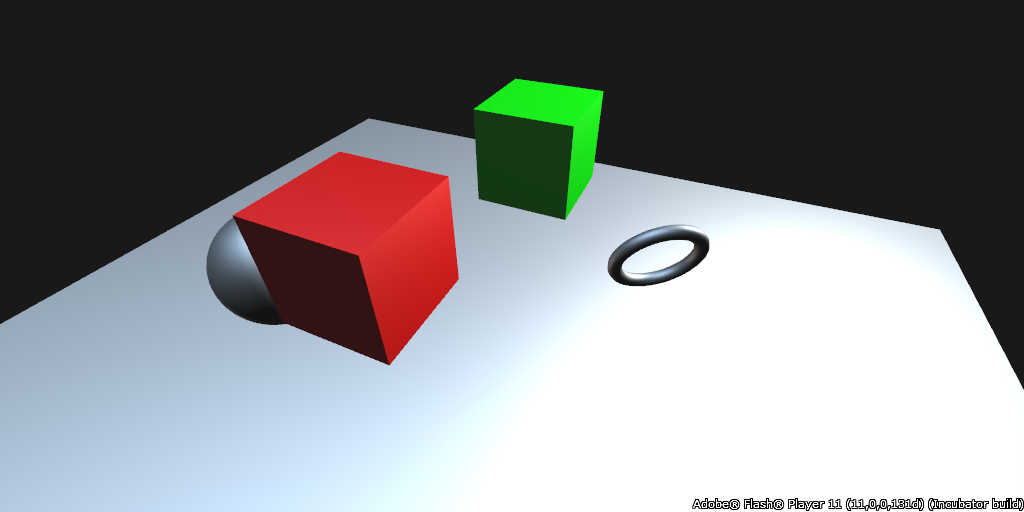
}

}

}

# Tutorial02\_AnimationAndMaterials

Creates an instanced cube, move its location in onAnimate() function, and change its material bases on its location.



onAnimate() method is called with time t and the elapsed time since the previous call: dt. In this method, you can change object position using *SceneMesh****.appendTranslation*** method.

In this sample, a cube is created using *ProceduralGeometry.createCube* method. Another cube is created using *SceneMesh.****instance***() method. This way, two cubes share the same vertex and index buffers.

By default, instanced cube will use the same material named “cubeMtrl”. This can be changed to a different material using *SceneMesh*.***materialBindings***. First create a new material and bind to the instanced cube. You can then later change the material property.

**override** **protected** **function** initModels():**void**

{

…

*// create a new standard material*

**var** material:MaterialStandard = **new** MaterialStandard( **"cubeMtrl"** ); material.diffuseColor.set( 0, 1, 0 );

**var** cube:SceneMesh = ProceduralGeometry.createCube( 5, material, **"cube"** );

cube.appendTranslation( 0, 6, -10 );

scene.addChild( cube );

*// create an instance of the cube (mesh data is shared)*

\_cubeInstanced = cube.instance( **"cube-instanced"** );

\_cubeInstanced.appendTranslation( 0, 6, 0 );

scene.addChild( \_cubeInstanced );

…

}

*// animation is performed in onAnimate*

**override** **protected** **function** onAnimate( t:Number, dt:Number ):**void**

{

\_cubeInstanced.setPosition( Math.cos( t ) \* 3, 6, Math.sin( t ) \* 3 );

*// Since a SceneMesh can have multiple submeshes of different materials,*

*// and since the submeshes can be shared amongst multiple SceneMesh instances,*

*// direct access to the material is not provided.*

*// To change material, one can create a new material and add it to the binding*

*// Note that the material name is used to indicate which material is remapped.*

**if** ( !\_cubeInstanced.materialBindings[ **"cubeMtrl"** ] )

\_cubeInstanced.materialBindings[ **"cubeMtrl"** ] = **new** MaterialStandard( **"cubeMtrl"** );

**if** ( \_cubeInstanced.position.x < 0 )

\_cubeInstanced.materialBindings[ **"cubeMtrl"** ].diffuseColor.set( 1, 0, 0 );

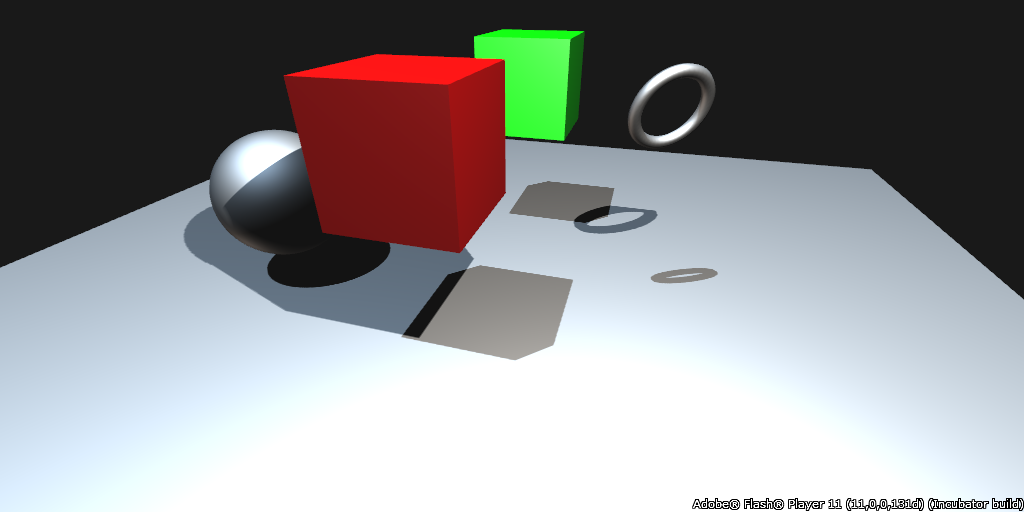
**else**

\_cubeInstanced.materialBindings[ **"cubeMtrl"** ].diffuseColor.set( 0, 1, 0 );

}

# Tutorial03\_Shadows

Shadows can be added by adding casters to the lights.



If *BasicDemo****.shadowMapEnabled*** is set, the two default lights of *BasicDemo* will have shadow enabled.

The next step is to define casters to the lights. For this, Proscenium provides ***SceneLight.addToShadowMap*** method. One easy way is to add *BasicDemo.scene*, which will make every scene graph node caster for the light, e.g., light[1] in this tutorial. Another way is to choose casters carefully, e.g., for light[0] in this tutorial. Therefore, the ground plane will not cast shadow from light[0] in this tutorial.

**override** **protected** **function** initModels():**void**

{

…

**if** ( lights )

{

lights[0].setPosition( 10, 20, 10);

**if** ( lights[0].shadowMapEnabled )

lights[0].addToShadowMap( \_cubeInstanced, cube, torus, sphere ); *// define casters*

**if** ( lights[1].shadowMapEnabled )

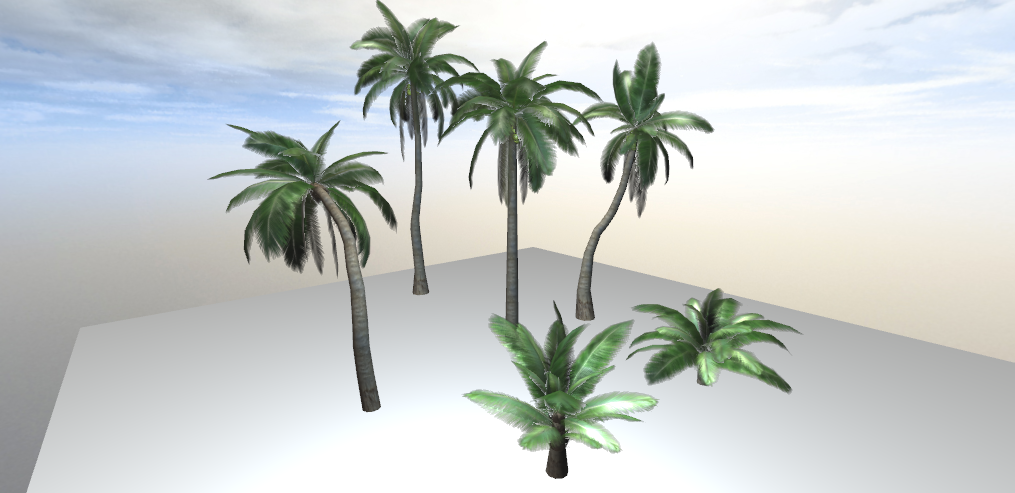
lights[1].addToShadowMap( scene ); *// just set every scene graph object as caster*

}

}

# Tutorial04\_LoadingModels

Proscenium supports models loaded from file. This tutorial loads skybox textures and a model.



***SceneSkyBox*** is the scene graph node object that is specifically provided for skybox. (See TestFog for fogging option)

Texture image file names for the six sky box faces are defined as

protected static const SKYBOX\_FILENAMES:Vector.<String> = new Vector.<String>(6,true);

SKYBOX\_FILENAMES[ 0 ] = "../res/textures/skybox/px.png";

SKYBOX\_FILENAMES[ 1 ] = "../res/textures/skybox/nx.png";

SKYBOX\_FILENAMES[ 2 ] = "../res/textures/skybox/py.png";

SKYBOX\_FILENAMES[ 3 ] = "../res/textures/skybox/ny.png";

SKYBOX\_FILENAMES[ 4 ] = "../res/textures/skybox/pz.png";

SKYBOX\_FILENAMES[ 5 ] = "../res/textures/skybox/nz.png";

In addition, this tutorial sample loads a tree model. Both loading the sky box textures and tree model are done by launching a loader for each. When the loading is done, each loader will issue an event. In this tutorial, we do this two loading tasks in the following way:

1. Sky texture loader is launched first.
2. When the loading is done, *imageLoadComplete* handler will be called.
3. In this handler, create a *SceneSkyBox* object and add it to the scene, and
4. we launch the tree model loader.
5. When the tree model is done, loadComplete handler is called.
6. In this handler, we can add the tree model to the scene.

override protected function initModels():void

{

var plane:SceneMesh = ProceduralGeometry.createPlane(

100,100,20,20,null, "plane" );

plane.transform.appendTranslation( 0, -2, 0 );

scene.addChild( plane );

LoadTracker.loadImages( SKYBOX\_FILENAMES, imageLoadComplete );

}

protected function imageLoadComplete( bitmaps:Dictionary ):void

{

var bitmapDatas:Vector.<BitmapData> = new Vector.<BitmapData>( 6, true );

var bitmap:Bitmap;

for ( var i:uint = 0; i < 6; i++ )

bitmapDatas[ i ] = bitmaps[ SKYBOX\_FILENAMES[ i ] ].bitmapData;

*// sky*

\_sky = new SceneSkyBox( bitmapDatas, false );

scene.addChild( \_sky ); *// skybox must be an immediate child of scene root*

\_sky.name = "Sky"

\_treeLoader = new OBJLoader( "../res/models/PalmTrees/PalmTrees.obj" );

\_treeLoader.addEventListener( Event.COMPLETE, loadComplete);

}

protected function loadComplete( event:Event ):void

{

var tree:SceneNode = new SceneNode( "PalmTrees" );

var manifest:ModelManifest = \_treeLoader.model.addTo( tree );

scene.addChild( tree );

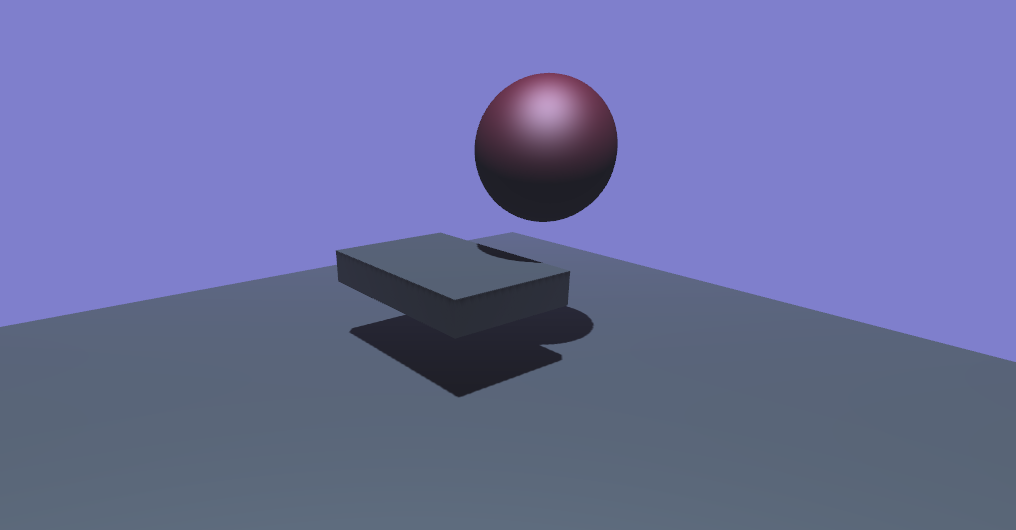
}

# Tutorial05\_SpriteBased

This tutorial shows creating Proscenium applications without using the BasicDemo harness, but by directly extending Sprite.

**public** **class** Tutorial\_SpriteBased **extends** Sprite

By extending Sprite, the developer has more control on adding Sprite to the stage, creating Stage3D context, and configuring and implementing event handlers.



## Creating Proscenium Instance3D

***Instance3D*** is the basic Proscenium class that maintains 3D Molehill context. This class must be created to use Proscenium. In the tutorial, the following code creates *Instance3D*.

instance = **new** Instance3D( stage3D.context3D );

Note that when we were using BasicDemo in previous tutorials, BasicDemo automatically created an Instance3D object: ***BasicDemo.instance***.

## Scene Graph Root

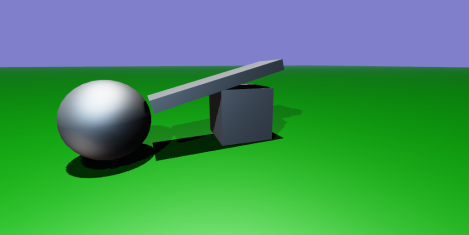
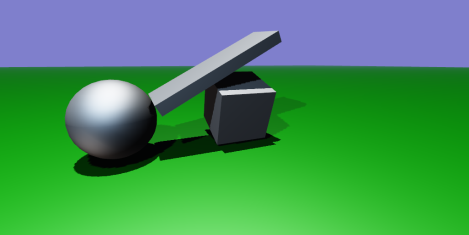
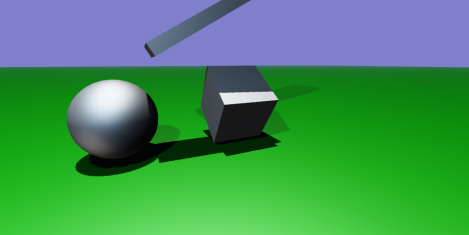
In the previous tutorials, the root was ***scene***, which was BasicDemo.scene. Now, we do not use BasicDemo. The root is simply *Instance3D*.***scene***, which is *instance.scene* in this tutorial (Note that *Instance3D* is a class name and *instance* is a variable.)

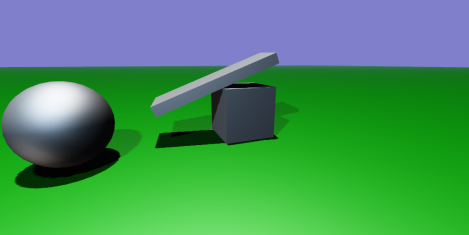
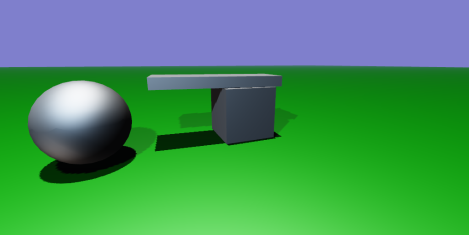
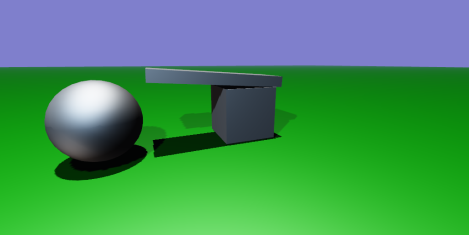
## Other Notes:

Once you have the scene graph root, creating models and adding them to the scene are the same as before. This tutorial creates a light, and enables shadow. This tutorial also has key and mouse event handlers. This tutorial contains a code that turns on fog.

# Tutorial06\_SimplePhysics

In this section, you will take a look at how to make objects fall, collide, and bounce.





1. Open the file named Tutorial05\_SimplePhysice.as located in the sample files folder.
2. Review the functions in the class.

Physics will be enabled by creating a PelletManager object that provides methods used to create Physics-enabled SceneMesh objects. For example, ground plane can be created by call createStaticInfinitePlane, boxes can be created by createBox, and spheres can be created by createSphere.

*// create a plane and add it to the scene*

**var** plane:SceneMesh = mSim.createStaticInfinitePlane( 1000, 1000, 2, 2, material, **"plane"** );

plane.appendTranslation( 0, -2, 0 );

scene.addChild( plane );

*// create cubes and add it to the scene*

**var** cube0:SceneMesh = mSim.createBox( 5, 5, 5 );

cube0.appendRotation( 40, Vector3D.X\_AXIS );

cube0.appendTranslation( 0, 6, 0 );

scene.addChild( cube0 );

*// create a sphere and add it to the scene*

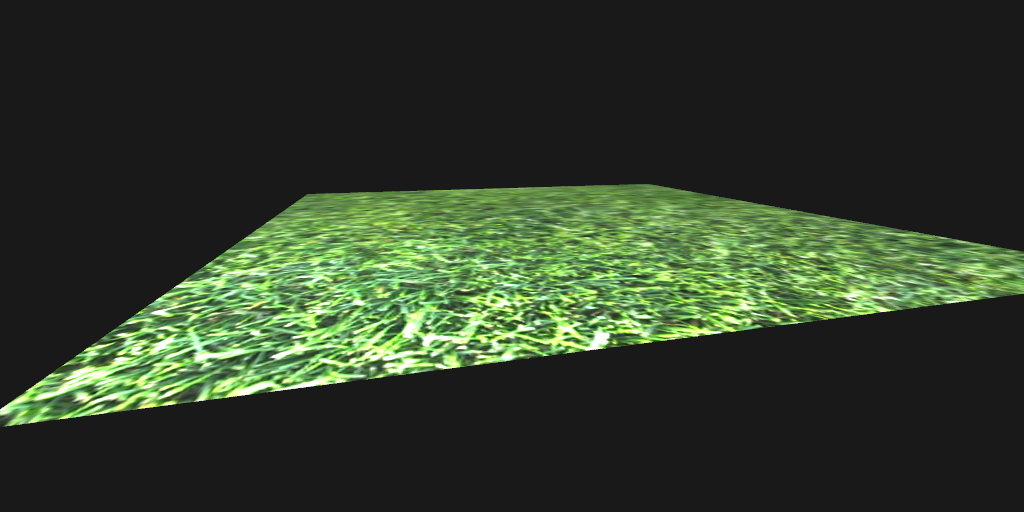
**var** sphere:SceneMesh = mSim.createSphere( 3, 32, 16 );

sphere.setPosition( -10, 2, 0 );

scene.addChild( sphere );

# Tutorial07\_Texture

In this section, you will take a look at codes that assign a texture to a quadrilateral.



1. Open the file named Tutorial07\_Texture.as located in the sample files folder.
2. Review the functions in the class.

Texture map can be embedded by

[**Embed**( source=**"/../res/content/foliage022.jpg"** )]

**protected** **static** **const** BITMAP:Class;

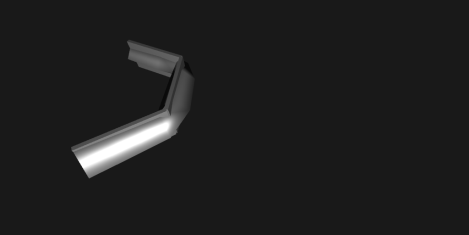
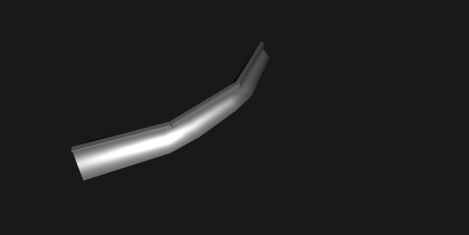
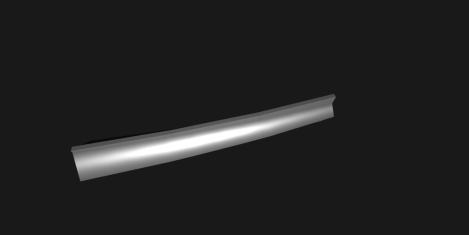
Using this data, a texture map can be created and used for diffuse map as shown in the following code.

**var** textureMap:TextureMap = **new** TextureMap( **new** BITMAP().bitmapData );

material.diffuseMap = textureMap;

# Tutorial 08\_LoadedAnimation

In this section, you will take a look at codes that load COLLADA animation file.



1. Open the file named Tutorial08\_LoadedAnimation.as located in the sample files folder.
2. Review the functions in the class.

To load a COLLADA file, refer to this code.

**public** **var** loader:ColladaLoader;

**override** **protected** **function** initModels():**void** {

loader = **new** ColladaLoader( **"../res/content/AnimatedBones.dae"** );

loader.addEventListener( Event.COMPLETE, onLoad );

}

When the loading completes, add the model to the scene and bind the animation to the scene.

**public** **var** animations:Vector.<AnimationController>, initialized:Boolean;

**public** **function** onLoad( event:Event ):**void** {

**var** manifest:ModelManifest = loader.model.addTo( scene );

animations = loader.model.animations;

**for each** ( **var** anim:AnimationController **in** animations ) {

anim.bind( scene );

}

initialized = **true**;

}

Animation is done by advancing time.

**override** **protected** **function** onAnimate( t:Number, dt:Number ):**void** {

**if** ( !initialized ) **return**;

**for each** ( **var** anim:AnimationController **in** animations )

{

anim.time = ( t % anim.length ) + anim.start;

}

}

# Tutorial09\_ProceduralGeometry

This sample provides codes that create a mesh from scratch.

1. Open the file named Tutorial\_09\_ProceduralGeometry.as located in the sample files folder.
2. Review the functions in the class.