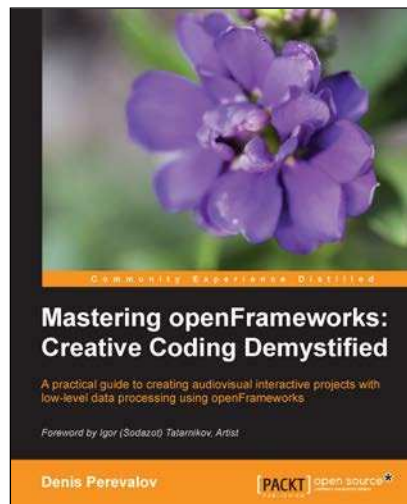


# Mastering openFrameworks: Creative Coding Demystified

Denis Perevalov



## Chapter No. 7 "Drawing in 3D"

## In this package, you will find:

A Biography of the author of the book

A preview chapter from the book, Chapter NO.7 "Drawing in 3D"

A synopsis of the book's content

Information on where to buy this book

## About the Author

**Denis Perevalov** is a computer vision research scientist. He works at the Institute of Mathematics and Mechanics of the Ural Branch of the Russian Academy of Sciences (Ekaterinburg, Russia). He is the co-author of two Russian patents on robotics computer vision systems and an US patent on voxel graphics. Since 2010 he has taught openFrameworks in the Ural Federal University. From 2011 he has been developing software for art and commercial interactive installations at [kuflex.com](http://kuflex.com) using openFrameworks. He is the co-founder of interactive technologies laboratory [expo32.ru](http://expo32.ru) (opened in 2012).

**For More Information:**

[www.packtpub.com/mastering-openframeworks-creative-coding-demystified/book](http://www.packtpub.com/mastering-openframeworks-creative-coding-demystified/book)

# Mastering openFrameworks: Creative Coding Demystified

openFrameworks is a simple and powerful C++ toolkit designed to develop real-time projects with focus on generating and processing graphics and sound. Nowadays, this is a popular platform for experiments in generative and sound art and creating interactive installations and audiovisual performances.

*Mastering openFrameworks: Creative Coding Demystified* covers programming openFrameworks 0.8.0 for Windows, Mac OS X, and Linux. It provides a complete introduction to openFrameworks, including installation, core capabilities, and addons. Advanced topics like shaders, computer vision, and depth cameras are also covered.

You will learn everything you need to know to create your own projects, ranging from simple generative art experiments to big interactive systems consisting of a number of computers, depth cameras, and projectors.

This book focuses on low-level data processing, which allows you to create really unique and cutting-edge works.

## What This Book Covers

*Chapter 1, openFrameworks Basics*, covers installing openFrameworks, the structure of openFrameworks projects, and creating the pendulum-simulation project.

*Chapter 2, Drawing in 2D*, explains the basics of two-dimensional graphics, including drawing geometric primitives, working with colors and drawing in the offscreen buffer. It also contains a generative art example of using numerical instability for drawing.

*Chapter 3, Building a Simple Particle System*, teaches the basics of particle system modeling and drawing. By the end of this chapter, you will build a fully featured project that can be used as a sketch for further experiments with particles.

*Chapter 4, Images and Textures*, covers the principles of working with images, including loading images from file; rendering it on the screen with different sizes, color, and transparency; creating new images; and modifying existing images. It also touches the basics of image warping and video mapping.

*Chapter 5, Working with Videos*, covers basic and advanced topics on playing, layering, and processing videos, including playing video files, processing live video grabbed from a camera, and working with image sequences. This chapter contains an implementation of the slit-scan effect and a simple video synthesizer, which uses a screen-to-camera feedback loop to create vivid effects on prerecorded videos.

**For More Information:**

[www.packtpub.com/mastering-openframeworks-creative-coding-demystified/book](http://www.packtpub.com/mastering-openframeworks-creative-coding-demystified/book)

*Chapter 6, Working with Sounds*, explains how to play sound samples, synthesize new sounds, and get sounds from the microphone. It includes the project wherein we generate music using bouncing-ball simulation, the PWM synthesizer, and the image-to-sound transcoding. Finally, it teaches us how to use spectrum analysis for creating an audio-reactive visual project.

*Chapter 7, Drawing in 3D*, covers representing, modifying, and drawing 3D objects. It includes examples of drawing a sphere-shaped cloud of triangles, an oscillating surface, and a twisting 3D knot.

*Chapter 8, Using Shaders*, explains how to use fragment, vertex, and geometry shaders for creating 2D video effects and 3D object deformations.

*Chapter 9, Computer Vision with OpenCV*, teaches the basics of computer vision using the OpenCV library. It explains how to perform filtering and correct perspective distortions in images and how to look for motion areas and detect bright objects in the videos. It includes an advanced example of using optical flow for video morphing.

*Chapter 10, Using Depth Cameras*, covers using depth cameras in openFrameworks projects using the ofxOpenNI addon. It includes an example of the projector-camera interactive system, which lets us draw abstract images on the wall. The example can be used as a sketch for creating interactive walls, tables, and floors.

*Chapter 11, Networking*, covers how to use OSC and TCP protocols in your openFrameworks projects for creating distributed projects that run on several computers. It includes an image-streaming example.

*Appendix A, Working with Addons*, teaches the basic principles of addons, explains how to link addons to your projects, and discusses some of the most useful addons.

*Appendix B, Perlin Noise*, explains the principles of using Perlin noise, which is employed in many of the examples in the book.

**For More Information:**

[www.packtpub.com/mastering-openframeworks-creative-coding-demystified/book](http://www.packtpub.com/mastering-openframeworks-creative-coding-demystified/book)

# 7

## Drawing in 3D

3D graphics often looks more impressive than 2D graphics because 3D has unique expressive capabilities, such as depth, perspective, and shading. Also, the third dimension allows objects to interweave and twist in the space in ways that are hard to achieve using 2D graphics. In this chapter we will cover the basics of rendering and animating 3D surfaces and primitive clouds with openFrameworks. We'll cover the following topics:

- Simple 3D drawing
- Using of Mesh
- Enabling lighting and setting normals
- Texturing
- Working with vertices

### 3D basics

Working with 3D means working with objects modeled in the three-dimensional scene, where the dimensions are horizontal (x), vertical (y), and depth (z). The resulting 3D scene is projected either onto a 2D image to show it on the screen, two 2D images for stereoscreen, or even printed as a 3D object using a 3D printer.

**For More Information:**

[www.packtpub.com/mastering-openframeworks-creative-coding-demystified/book](http://www.packtpub.com/mastering-openframeworks-creative-coding-demystified/book)

## Representation of 3D objects

Each 3D object is represented using a number of elementary primitives such as points, line segments, triangles, or other polygons. Methods of the object's representation are as follows:

- An object is a number of surfaces assembled from polygonal primitives such as triangles and quadrangles (often called **quads**). This method is used in 3D-modeling software for representing "surface" objects, such as a human body, a car, a building, and also clothes and a rippled water surface.
- An object is a number of **curves** assembled from line segments. Such a representation is used for modeling hair and fur.
- An object is a huge number of small points called **particles**. This is representation of objects without distinct shape: smoke, clouds, fire, and a waterfall (see *Chapter 3, Building a Simple Particle System*).

These methods refer to realistic representation of real-world objects. We are interested in experimental 3D, so we can play with representations freely. For example:

- Triangles can be used to draw some clouds made from triangles but not smooth surfaces
- Thousands of long curves can interweave inside a volume with specified bounds, creating an evolving "hairy" 3D object
- Particles can represent a rigid 3D object that suddenly changes its shape in a complex way

In *openFrameworks*, you can represent and draw 3D objects by yourself; see the *Simple 3D drawing* section. But normally it is preferable to use a powerful `ofMesh` class, which lets you represent and draw surfaces, curves, particles, and distinct primitives at the fastest speed; see the *Using ofMesh* section. Also you can manipulate the static and animated 3D models stored in files such as 3DS; see the *Additional topics* section.

## 3D scene rendering

In this chapter we will consider rendering a 3D scene on a 2D screen (and will not consider stereoscreens and 3D printers).

Recall that, when we draw a flat 2D scene, we just imprint objects such as images and curves onto the screen at the specified coordinates. And the order of the object's drawing defines its visibility; the last object is visible as a whole and can occlude the objects drawn before it.

The rendering of a 3D scene differs from the case of a 2D scene because the object's visibility here is defined by its z coordinate (depth). By default, in openFrameworks, points with a zero value for the z coordinate forms an xy plane, which is used for 2D drawing. Increasing and decreasing the value of the z coordinate leads to moving the objects closer or farther correspondingly.

openFrameworks graphics is based on **Open Graphics Library (OpenGL)**, which renders objects using z-buffering technology. This technology just stores z values for each screen pixel in a special buffer, called **z-buffer** (or **depth buffer**). During rendering, if the z value of the object's pixel is greater than the z value in the buffer, the pixel is rendered and the z-buffer is updated to this value. Otherwise, the object's pixel is not rendered.

By default, the z-buffering is disabled. To enable it, call the following function:

```
ofEnableDepthTest();
```

When enabled, the z-buffer clears automatically at each frame, together with the background drawing (if you do not call `ofSetBackgroundAuto( false )`). To disable z-buffering, use the `ofDisableDepthTest()` function.



There is another 3D rendering technology, called **ray tracing**. Instead of directly projecting the pixels of primitive onto the screen, it simulates light ray propagation from the light sources to the camera. Such a method is a natural way to construct shadows and other natural-world lighting effects. It is used for the highest quality 3D graphics and is available in 3D animation software. But its real-time implementations are currently very resource intensive, and we do not consider them here.

The volumetric nature of the 3D objects introduces new attributes into the 3D scene. These are lights, the object's materials interacting with lights, the 3D scene perspective, and virtual cameras. See the *Enabling lighting and setting normals* and *Additional topics* sections for more information.

Note, the modern approach in 3D that includes advanced lighting and shading, object's shape manipulation, and the rendered scene postprocessing requires using shaders; see *Chapter 8, Using Shaders*, for further details.



openFrameworks is a thin wrapper over OpenGL, so it provides low-level functionality, which is great for working with custom-generated 3D graphics. However, if you need to work with 3D worlds consisting of many life-like models and characters, it is probably better to use some other 3D engine, such as Unity 3D. We use Unity 3D for complex 3D world rendering and add interactivity by controlling it from openFrameworks' project, which processes sensors such as depth cameras. openFrameworks and Unity 3D are connected via OSC network protocol; see *Chapter 11, Networking*.

Now we will consider a simple 3D drawing example with openFrameworks.

## Simple 3D drawing

For simple 3D drawing in openFrameworks, follow these steps:

1. Add the `ofEnableDepthTest()` function call in the beginning of the `testApp::draw()` function to enable z-buffering. If you omit it, all the graphics objects will be rendered without respect to their z coordinate in correspondence with the graphical primitives' rendering order.
2. Draw primitives as follows:
  - The `ofLine(x1, y1, z1, x2, y2, z2)` function draws a line segment between points  $(x1, y1, z1)$  and  $(x2, y2, z2)$ . There is an overloaded version of the function, `ofLine(p1, p2)`, where `p1` and `p2` have type `ofPoint`. Use the `ofSetColor()` and `ofSetLineWidth()` functions to adjust its rendering properties of color and line width.



In *Chapter 2, Drawing in 2D*, we used the `ofPoint` class to represent 2D points using its fields `x` and `y`. Actually, `ofPoint` has a third field `z`, which, by default, is equal to zero. So `ofPoint` can represent points in 3D. Just declare `ofPoint p` and work with values `p.x`, `p.y`, and `p.z`.

- The `ofTriangle(p1, p2, p3)` function draws a triangle with vertices in points `p1`, `p2`, and `p3`. Use the `ofSetColor()`, `ofFill()`, `ofSetLineWidth()`, and `ofNoFill()` functions to adjust its rendering properties.



- The `ofRect( x, y, z, w, h )` function draws a rectangle with the top-left corner at  $(x, y, z)$  and the width  $w$  and height  $h$ , oriented parallel to the screen plane. If you need to get a rotated rectangle, you need to rotate the coordinate system using the `ofRotate()` function.

To draw arbitrary polygons – for example, quadrangles – use the following method:

```
ofBeginShape();           //Begin shape
ofVertex( x1, y1, z1 );   //The first vertex
ofVertex( x2, y2, z2 );   //The second vertex
//...
ofVertex( xn, yn, zn );   //The last vertex
ofEndShape();            //End shape
```

If `ofFill()` was called before drawing, the shape will be drawn filled and closed. If `ofNoFill()` was called before drawing, just an unclosed polygon will be drawn.

3. Translate, scale, and rotate the rendered objects by manipulating the coordinate system:
  - The `ofTranslate( x, y, z )` function translates the coordinate system by vector  $(x, y, z)$
  - The `ofScale( x, y, z )` function scales the coordinate system by factors  $(x, y, z)$
  - The `ofRotate( angle, x, y, z )` function rotates the coordinate system along vector  $(x, y, z)$  by `angle` degrees

As in a 2D case, use `ofPushMatrix()` and `ofPopMatrix()` to store and retrieve the current coordinate system in a matrix stack.

Now we will illustrate these steps in an example.

## The triangles cloud example

Let's draw 1500 random triangles, located at an equal distance from the center of the coordinates. This will look like a triangle cloud in the shape of a sphere. To make the visualization more interesting, colorize the triangles with random colors from black to red and add constant rotation to the cloud.



This is example 07-3D/01-TrianglesCloud.

**For More Information:**

[www.packtpub.com/mastering-openframeworks-creative-coding-demystified/book](http://www.packtpub.com/mastering-openframeworks-creative-coding-demystified/book)

The example is based on the `emptyExample` project in `openFrameworks`. In the `testApp.h` file, inside the `testApp` class declaration, add arrays `vertices` and `colors` to hold the vertices and the colors of the triangles and variables `nTri` and `nVert` corresponding to the number of triangles and their vertices:

```
vector<ofPoint> vertices;
vector<ofColor> colors;
int nTri;          //The number of triangles
int nVert;        //The number of the vertices equals nTri * 3
```

The `setup()` function fills the arrays for the triangles' vertices and colors. The vertices of the first triangle are stored in `vertices[0]`, `vertices[1]`, and `vertices[2]`. The vertices of the second triangle are stored in `vertices[3]`, `vertices[4]`, `vertices[5]`, and so on. In general, the vertices of the triangle with index `i` (where `i` is in range from 0 to `N-1`) are stored in the vertices with the indices `i * 3`, `i * 3 + 1`, and `i * 3 + 2`.

```
void testApp::setup() {
    nTri = 1500;          //The number of the triangles
    nVert= nTri * 3;     //The number of the vertices

    float Rad = 250;     //The sphere's radius
    float rad = 25;      //Maximal triangle's "radius"
                        //(formally, it's the maximal coordinates'
                        //deviation from the triangle's center)

    //Fill the vertices array
    vertices.resize( nVert );          //Set the array size
    for (int i=0; i<nTri; i++) {        //Scan all the triangles
        //Generate the center of the triangle
        //as a random point on the sphere

        //Take the random point from
        //cube [-1,1]x[-1,1]x[-1,1]
        ofPoint center( ofRandom( -1, 1 ),
                        ofRandom( -1, 1 ),
                        ofRandom( -1, 1 ) );
        center.normalize(); //Normalize vector's length to 1
        center *= Rad;      //Now the center vector has
                            //length Rad

        //Generate the triangle's vertices
        //as the center plus random point from
        //[-rad, rad]x[-rad, rad]x[-rad, rad]
        for (int j=0; j<3; j++) {
            vertices[ i*3 + j ] =
```

```

        center + ofPoint( ofRandom( -rad, rad ),
                          ofRandom( -rad, rad ),
                          ofRandom( -rad, rad ) );
    }
}

//Fill the array of triangles' colors
colors.resize( nTri );
for (int i=0; i<nTri; i++) {
    //Take a random color from black to red
    colors[i] = ofColor( ofRandom( 0, 255 ), 0, 0 );
}
}

```

The `update()` function is empty here, and the `draw()` function enables z-buffering, which rotates the coordinate system based on time, and draws the triangles with the specified colors.

```

void testApp::draw(){
    ofEnableDepthTest();    //Enable z-buffering

    //Set a gradient background from white to gray
    //for adding an illusion of visual depth to the scene
    ofBackgroundGradient( ofColor( 255 ), ofColor( 128 ) );

    ofPushMatrix();    //Store the coordinate system

    //Move the coordinate center to screen's center
    ofTranslate( ofGetWidth()/2, ofGetHeight()/2, 0 );

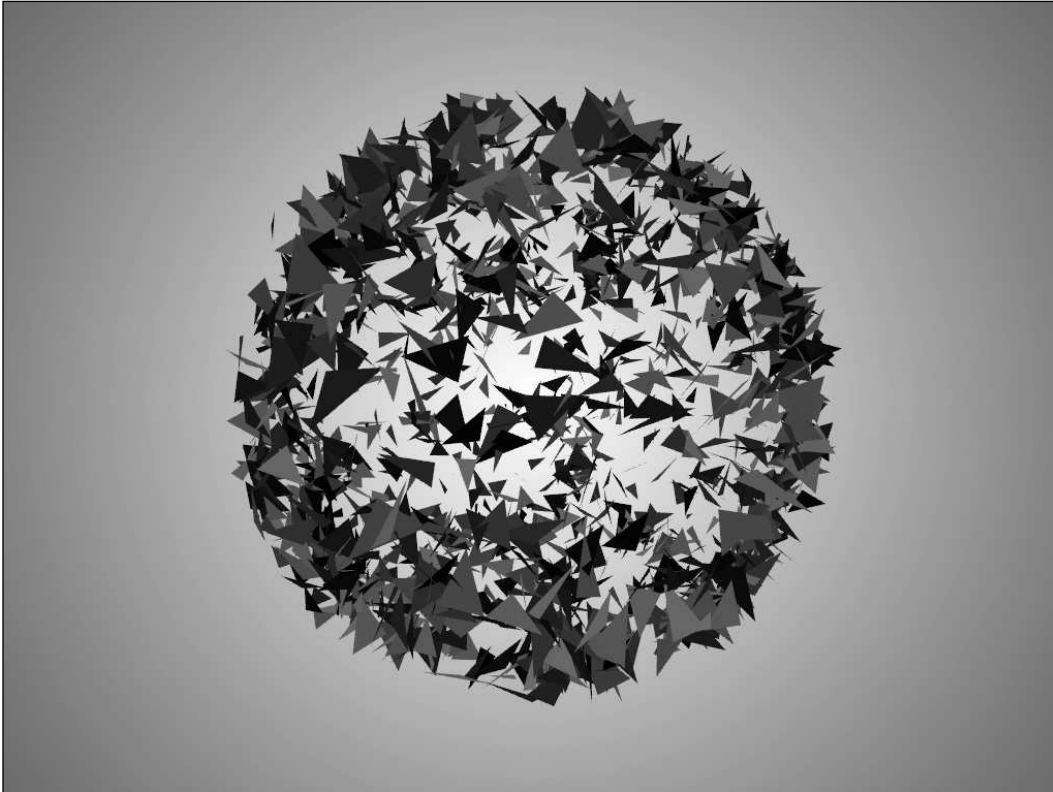
    //Calculate the rotation angle
    float time = ofGetElapsedTimef();    //Get time in seconds
    float angle = time * 10; //Compute angle. We rotate at speed
                                //10 degrees per second
    ofRotate( angle, 0, 1, 0 );    //Rotate the coordinate system
                                //along y-axis

    //Draw the triangles
    for (int i=0; i<nTri; i++) {
        ofSetColor( colors[i] );    //Set color
        ofTriangle( vertices[ i*3 ],
                   vertices[ i*3 + 1 ],
                   vertices[ i*3 + 2 ] ); //Draw triangle
    }

    ofPopMatrix();    //Restore the coordinate system
}

```

Run the code and you will see a sphere-like rotating cloud of triangles as shown in the following screenshot:



To draw the background, we use the `ofBackgroundGradient( color1, color2, type )` function. It creates the gradient filling of type `type` for the entire application's screen, with colors interpolated from `color1` to `color2`. The possible values of `type` are as follows:

- `OF_GRADIENT_CIRCULAR` – This type gives a circular color gradient with the center being the center of screen. This is the default value.
- `OF_GRADIENT_LINEAR` – This type gives you a top-to-bottom gradient.
- `OF_GRADIENT_BAR` – This type gives you a center-to-top and a center-to-bottom gradient.

Note that each triangle moves and rotates on the screen but its color always remains unchanged. The reason for this is that we don't use light and normals, which control how a graphics primitive is lit and shaded.

The simplest way to add lighting and normals is using the `ofMesh` class, which we will consider now.

## Using ofMesh

The `ofMesh` class is a powerful class that is used for representing, modifying, and rendering 3D objects. By default, it draws triangle meshes, but it can also be used for drawing curves and points.

The `ofMesh` class performs rendering of many thousands and even millions of triangles by one OpenGL call, at the highest possible speed. Even though using `ofMesh` will at first seem slightly more complicated than using `ofTriangle()`, it will give you more flexibility in creating and modifying 3D objects in return. So it is highly recommended that you use `ofMesh` for 3D in all cases, except the very beginning or for learning 3D. You can use `ofMesh` not only for 3D but for 2D graphics as well.



openFrameworks has one more class, named `ofVBOMesh`, that is used for working with meshes. The class name means "mesh based on **Vertex Buffer Object (VBO)**". This class is similar to `ofMesh`, but it renders significantly faster when the vertices of the mesh are not changing. See details of its usage and performance in comparison with `ofMesh` in openFrameworks example `examples/gl/vboExample`.

To draw a surface consisting of a number of triangles, follow these steps:



This is example `07-3D/02-PyramidMesh`. It is based on the `emptyExample` project in openFrameworks.

1. Declare an object `mesh` of type `ofMesh` in the `testApp` class declaration:  

```
ofMesh mesh;
```
2. Add the vertices of the surface triangles to the mesh using the `mesh.addVertex( p )` function. Note that if a vertex belongs to several triangles, you should specify these vertices just once. This feature is very useful for changing the surface; you change the position of just one vertex, and all the triangles will be drawn correctly.

Vertices are added to the end of a special array of vertices in the mesh and are later referenced by indices in this array. So the first vertex has the index 0, the second vertex has the index 1, and so on. For example, to draw a pyramid, we specify its four vertices as follows:

```
//Pyramid's base vertices with indices 0, 1, 2
mesh.addVertex( ofPoint( -200, -100, -50 ) );
mesh.addVertex( ofPoint( 200, -100, -50 ) );
mesh.addVertex( ofPoint( 0, 200, 0 ) );

//Pyramid's top vertex with index 3
mesh.addVertex( ofPoint( 0, 0, 50 ) );
```

3. Add the triangles by specifying the indices of the vertices for each triangle using the `mesh.addTriangle( index1, index2, index3 )` function. Be careful to order this in the clockwise direction for correct lighting. In our pyramid example, we specify just three of its four triangles, so that you can see the interior of the object.

```
//Vertices with indices 3, 2, 0
mesh.addTriangle( 3, 2, 0 );
```

```
//Vertices with indices 3, 1, 2
mesh.addTriangle( 3, 1, 2 );
```

```
//Vertices with indices 3, 0, 1
mesh.addTriangle( 3, 0, 1 );
```

4. Draw a mesh in the `testApp::draw()` function using the `mesh.draw()` function. You may need coordinate system transformations for moving and rotating the object. For example, a rotating pyramid can be drawn with the following code in `testApp::draw()`:

```
ofEnableDepthTest(); //Enable z-buffering

//Set a background
ofBackgroundGradient( ofColor( 255 ), ofColor( 128 ) );

ofPushMatrix(); //Store the coordinate system

//Move coordinate center to screen's center
ofTranslate( ofGetWidth()/2, ofGetHeight()/2, 0 );

//Rotate the coordinate system
float time = ofGetElapsedTimef(); //Get time in seconds
float angle = time * 30; //Rotate angle
```

```

ofRotate( angle, 0, 1, 1 );

ofSetColor( 0, 128, 0 ); //Set a dark green color
mesh.draw();           //Draw the mesh


ofPopMatrix();          //Restore the coordinate system

```

When you run this code, you will see the pyramid is uniformly colored a dark green color. It looks like some animated 2D polygon and it is hard to make out that this is really a 3D pyramid surface. To see the mesh as a 3D object, you need to enable lighting for the scene and add normals information to the mesh. Let's do it.

## Enabling lighting and setting normals

Lighting is needed for different parts of the surface to have different shading, depending on their orientation to the viewer. Such shading makes the surfaces look much more interesting than if just rendered with a uniform color because it emphasizes the 3D curvature of the surfaces. `openFrameworks` has an `ofLight` class for controlling light sources.

 This is example 07-3D/03-PyramidLighting. This example is a good starting point for drawing smooth surfaces using the `setNormals()` function. It is a continuation of example 07-3D/02-PyramidMesh.

To use one light source with default parameters, add the following line in the `testApp` class declaration:

```
ofLight light;
```

Add the following line in the `testApp::setup()` function to enable it:

```
light.enable(); //Enabling light source
```

For the light to interact with the mesh properly, you need to set up normal vectors for all the vertices using the `mesh.addNormal( normal )` function. Each normal vector should have unit length and direction perpendicular to the surface in the vertex. Information about the normals gives `openFrameworks` information about the correct lighting of the surface. Across the chapter, we will use the `setNormals()` function for normals computing, which we will discuss.

## Computing normals using the setNormals() function

To compute normals for a mesh consisting of triangles, you can use the following function:

```
//Universal function which sets normals for the triangle mesh
void setNormals( ofMesh &mesh ){

    //The number of the vertices
    int nV = mesh.getNumVertices();

    //The number of the triangles
    int nT = mesh.getNumIndices() / 3;

    vector<ofPoint> norm( nV ); //Array for the normals

    //Scan all the triangles. For each triangle add its
    //normal to norm's vectors of triangle's vertices
    for (int t=0; t<nT; t++) {
        //Get indices of the triangle t
        int i1 = mesh.getIndex( 3 * t );
        int i2 = mesh.getIndex( 3 * t + 1 );
        int i3 = mesh.getIndex( 3 * t + 2 );

        //Get vertices of the triangle
        const ofPoint &v1 = mesh.getVertex( i1 );
        const ofPoint &v2 = mesh.getVertex( i2 );
        const ofPoint &v3 = mesh.getVertex( i3 );

        //Compute the triangle's normal
        ofPoint dir = ( (v2 - v1).crossed( v3 - v1 ) ).normalized();

        //Accumulate it to norm array for i1, i2, i3
        norm[ i1 ] += dir;
        norm[ i2 ] += dir;
        norm[ i3 ] += dir;
    }

    //Normalize the normal's length
    for (int i=0; i<nV; i++) {
        norm[i].normalize();
    }


    //Set the normals to mesh
    mesh.clearNormals();
    mesh.addNormals( norm );
}
```



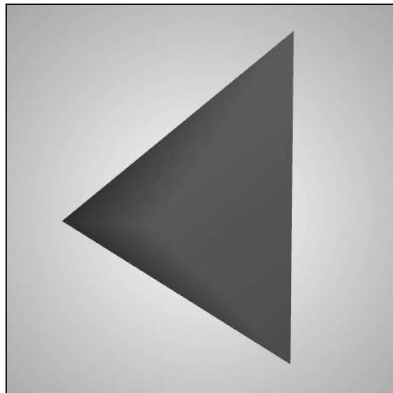
To use it in your project, insert this function at the end of the `testApp.cpp` file, and add its declaration in the `testApp.h` file (outside the `testApp` class):

```
//Universal function which sets normals for the triangle mesh
void setNormals( ofMesh &mesh );
```

Now you can call `setNormals( mesh )` and the normals will be computed. You need to call the `setNormals( mesh )` function after each modification of vertices of mesh for the normals to be up-to-date.


 Scaling using `ofScale()` while drawing affects not only the object's vertices but the normals vectors too, and it can make shading improper. So when using normals, just avoid scaling or recalculating the normals so that they have unit length even after the usage of `ofScale()`.

With lighting and normals, the pyramid looks a little more like a 3D object, which changes its shade depending on its orientation:



Note that the lightness of all the surface triangles mainly depends on the orientation of the central ("top") vertex of the pyramid. The reason is that shading of each triangle is computed by interpolating the normals of its vertices, and in our case, the normal of the central vertex is perpendicular to the pyramid's base. Such an approach works well for drawing smooth surfaces; see the *The oscillating plane example* section. Although in our case of pyramid, it can look a little bit unnatural.

To obtain the most natural visualization of the pyramid with sharp edges, we need to draw triangles independently without formally creating any common vertices.

**For More Information:**

[www.packtpub.com/mastering-openframeworks-creative-coding-demystified/book](http://www.packtpub.com/mastering-openframeworks-creative-coding-demystified/book)

## Drawing sharp edges

The simplest way to achieve sharp edges is to add the vertices for all the triangles in mesh and not use the `addTriangle()` function at all and then call the `mesh.setupIndicesAuto()` function, which sets indices automatically such that vertices (0, 1, 2) are used for drawing the first triangle, vertices (4, 5, 6) for the second triangle, and so on.

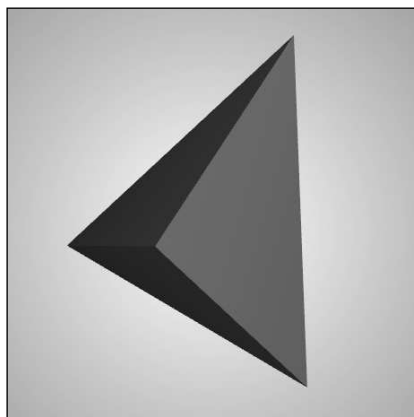


This is example 07-3D/04-PyramidSharpEdges. This example is a good starting point for drawing sharp 3D objects. It is based on example 07-3D/03-PyramidLighting.

In the example with the pyramid, replace all the lines with `addVertex()` and `addTriangle()` with the following lines:

```
//Pyramid's base vertices
ofPoint v0 = ofPoint( -200, -100, 0 );
ofPoint v1 = ofPoint( 200, -100, 0 );
ofPoint v2 = ofPoint( 0, 200, 0 );
//Pyramid's top vertex
ofPoint v3 = ofPoint( 0, 0, 100 );
//Add triangles by its vertices
mesh.addVertex( v3 ); mesh.addVertex( v2 ); mesh.addVertex( v0 );
mesh.addVertex( v3 ); mesh.addVertex( v1 ); mesh.addVertex( v2 );
mesh.addVertex( v3 ); mesh.addVertex( v0 ); mesh.addVertex( v1 );
mesh.setupIndicesAuto(); //Set up indices
```

As a result, you will see a pyramid with sharp edges as shown in the following screenshot:



We have considered a basic workflow with meshes. Now we will consider other useful capabilities of the `ofMesh` class.

## Drawing line segments and points

Instead of `mesh.draw()`, you can use the following functions:

- The `mesh.drawWireframe()` function draws only surface edges without the interiors of the triangles. Such a mode of drawing is called **wireframe drawing**; it is very useful for debugging, and of course, can be used as an effect.
- The `mesh.drawVertices()` function draws only vertices of the mesh. It is useful for debugging and also as an effect.

Also, to represent not only triangular surfaces but also objects consisting of line segments or points, use the `mesh.setMode( mode )` function, where `mode` has type `ofPrimitiveMode` enumeration. To see all the possible values for `mode`, check its definition. We will mention only three values:

- `OF_PRIMITIVE_TRIANGLES` is a default value, which draws a mesh as triangles. We had considered how to use this `mode` in the pyramid examples mentioned earlier.
- `OF_PRIMITIVE_LINES` draws a mesh as a number of line segments.
- `OF_PRIMITIVE_POINTS` draws a mesh as a number of points.

Let's consider the last two modes in detail.

## Drawing line segments

Calling `mesh.setMode( OF_PRIMITIVE_LINES )` switches `mesh` to a mode in which it draws line segments. After calling this function, add all vertices of segments using `mesh.addVertex( p )`, and for each segment, it adds the indices of the vertices using the following code:

```
mesh.addIndex( i1 ); //Index of segment's first vertex
mesh.addIndex( i2 ); //Index of segment's second vertex
```

For example, to draw a tripod, create the mesh using the following code:

```
mesh.setMode( OF_PRIMITIVE_LINES );
mesh.addVertex( ofPoint( 0, 0, 0 ) ); //Vertex 0
mesh.addVertex( ofPoint( -100, -100, 0 ) ); //Vertex 1
mesh.addVertex( ofPoint( 100, -100, 0 ) ); //Vertex 2
```

```
mesh.addVertex( ofPoint( 0, 100, 0 ) );          //Vertex 3

mesh.addIndex( 0 ); mesh.addIndex( 1 ); //Segment 0
mesh.addIndex( 0 ); mesh.addIndex( 2 ); //Segment 1
mesh.addIndex( 0 ); mesh.addIndex( 3 ); //Segment 2
```

Note that for correct lighting you need to specify normals, which normally cannot be defined for lines. So the best idea is to disable lighting using the `ofDisableLighting()` function before drawing and then enabling it again using the `ofEnableLighting()` function:

```
ofDisableLighting();          //Disable lighting
mesh.draw();                  //Draw lines
ofEnableLighting();           //Enable lighting
```

## Drawing points

Calling `mesh.setMode( OF_PRIMITIVE_POINTS )` switches mesh to a mode in which it draws its vertices as points.

Additionally, call `glPointSize( size )` to specify point size in pixels, and call `glEnable( GL_POINT_SMOOTH )` to draw circular points (instead of square points as on some graphics cards). For example, add the following lines after specifying tripod vertices in the previous example:

```
mesh.setMode( OF_PRIMITIVE_POINTS );
glPointSize( 10 );
glEnable( GL_POINT_SMOOTH );
```

Once you run the code, you will see four circles, corresponding to the tripod's vertices.

## Coloring the vertices

It is possible to specify the colors of the vertices. In this case, you must provide a color for all the vertices using the `mesh.addColor( color )` function; for example, `mesh.addColor( ofColor( 255, 0, 0 ) )`. Note that in this case, the `ofSetColor()` function will not affect the drawing of the mesh. Remember: you should call this function as many times as you call the `mesh.addVertex()` function.

## Texturing

You can wrap any image or texture on the surface using the `mesh.addTexCoord( texPoint )` function. Here `texPoint` is of the `ofPoint` type. It is a 2D point that should lie in range  $[0, w] \times [0, h]$ , where  $w \times h$  is the size of the image that you want to use as a texture. Remember that you should call this function as many times as you call the `mesh.addVertex()` function so that all the vertices will have texture coordinates.

During rendering each primitive of the mesh (whether triangle, line, or point depends on the mesh's mode), the texture coordinates of each rendered pixel will be calculated by OpenGL as interpolation of texture coordinates of the primitive's vertices. Resulting texture coordinates for the pixel are used for the pixel's color computing. In other words, the final pixel color is computed using three values: the color given by the texture, the color of the last `ofSetColor()` calling, and the shading information obtained from the light and normals data. To change the algorithm of computing pixel color and the use of fragment shaders, see *Chapter 8, Using Shaders*.

For example, let's wrap the `sunflower.png` image onto the pyramid.



This is example 07-3D/05-PyramidTextured. It is a continuation of example 07-3D/04-PyramidSharpEdges.

Copy the image into the `bin/data` folder of the project, and declare the `ofImage` image in the `testApp` class declaration. Then add the following lines in `testApp::setup()`:

```
//Set up a texture coordinates for all the vertices
mesh.addTexCoord( ofPoint( 100, 100 ) ); //v3
mesh.addTexCoord( ofPoint( 10, 300 ) ); //v2
mesh.addTexCoord( ofPoint( 10, 10 ) ); //v0

mesh.addTexCoord( ofPoint( 100, 100 ) ); //v3
mesh.addTexCoord( ofPoint( 300, 10 ) ); //v1
mesh.addTexCoord( ofPoint( 10, 300 ) ); //v2

mesh.addTexCoord( ofPoint( 100, 100 ) ); //v3
mesh.addTexCoord( ofPoint( 10, 10 ) ); //v0
mesh.addTexCoord( ofPoint( 300, 10 ) ); //v1
//Load an image
image.loadImage( "sunflower.png" );
```

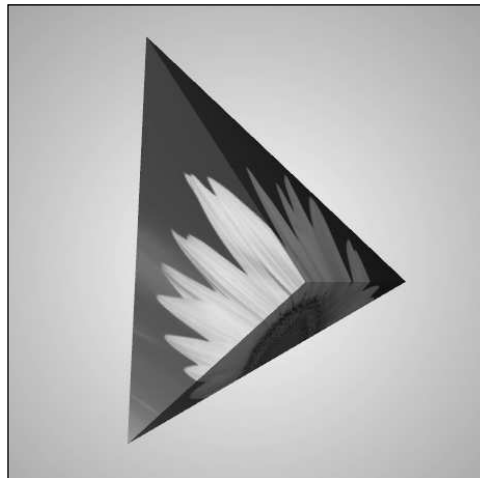
Finally, in `testApp::draw()`, find the following lines:

```
ofSetColor( 0, 128, 0 ); //Set a dark green color
mesh.draw();
```

Replace the preceding lines with the following:

```
ofSetColor( 255, 255, 255 ); //Set white color
image.bind(); //Use image's texture for drawing
mesh.draw(); //Draw mesh
image.unbind(); //End using image's texture
```

After running the preceding code, you will see the pyramid with a wrapped texture as shown in the following screenshot:



## Working with vertices

There are a number of functions for accessing the vertices and their properties:

- The `getNumVertices()` function returns the number of vertices.
- The `getVertex( i )` function returns the position of the vertex with index `i`.
- The `setVertex( i, p )` function sets the position of vertex `i` to `p`. Note that this function can change the vertex but it cannot add a new vertex. So if `i` is greater or equal to `mesh.getNumVertices()`, you need to add a vertex (or vertices) using the `mesh.addVertex( p )` function as described in the *Using of Mesh* section.

- The `removeVertex( i )` function deletes the vertex with index `i`. Be very careful when using this function; after deleting a vertex, you should probably also delete the corresponding normal, color, and texture coordinate, and change the indices of the triangles to keep its coherence.
- The `clearVertices()` function deletes all the vertices. See corresponding cautions for `removeVertex()`.
- The `clear()` function clears the mesh, including its vertices, normals, and all other arrays.

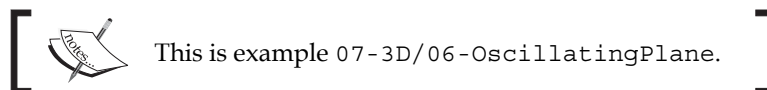
After changing vertices, you will most probably need to update the normals using the `setNormals( mesh )` function, as described in the *Computing normals using the `setNormals()` function* section.

There are similar functions for controlling normals, colors, texture coordinates, and indices; for example, functions `getNumNormals()`, `getNumColors()`, `getNumTexCoords()`, and `getNumIndices()` return number of normals, colors, texture coordinates, and indices respectively.

Let's see a simple example of modifying the positions of the vertices.

## The oscillating plane example

This example demonstrates how to create a flat plane from triangles and then oscillate its vertices to obtain a dynamic surface. Also, the color of vertices will depend on the oscillation amplitude.



The example is based on the `emptyExample` project in `openFrameworks`. Begin with adding the declaration and definition of the `setNormals()` function, as described in the *Computing normals using the `setNormals()` function* section. Then in the `testApp.h` file, in the `testApp` class declaration, add definitions of `mesh` and `light`:

```
ofMesh mesh;           //Mesh
ofLight light;        //Light
```

In the beginning of the `testApp.cpp` file, add constants with vertex grid size:

```
int W = 100;           //Grid size
int H = 100;
```

The `setup()` function adds vertices and triangles to the mesh and enables lighting:

```
void testApp::setup() {
    //Set up vertices and colors
    for (int y=0; y<H; y++) {
        for (int x=0; x<W; x++) {
            mesh.addVertex(
                ofPoint( (x - W/2) * 6, (y - H/2) * 6, 0 ) );
            mesh.addColor( ofColor( 0, 0, 0 ) );
        }
    }
    //Set up triangles' indices
    for (int y=0; y<H-1; y++) {
        for (int x=0; x<W-1; x++) {
            int i1 = x + W * y;
            int i2 = x+1 + W * y;
            int i3 = x + W * (y+1);
            int i4 = x+1 + W * (y+1);
            mesh.addTriangle( i1, i2, i3 );
            mesh.addTriangle( i2, i4, i3 );
        }
    }
    setNormals( mesh ); //Set normals
    light.enable(); //Enable lighting
}
```

The `update()` function changes the z coordinate of each vertex using Perlin noise (refer to *Appendix B, Perlin Noise*) and also sets its color between the range blue to white:

```
void testApp::update() {
    float time = ofGetElapsedTimef(); //Get time
    //Change vertices
    for (int y=0; y<H; y++) {
        for (int x=0; x<W; x++) {
            int i = x + W * y; //Vertex index
            ofPoint p = mesh.getVertex( i );
            //Get Perlin noise value
            float value =
                ofNoise( x * 0.05, y * 0.05, time * 0.5 );
            //Change z-coordinate of vertex
        }
    }
}
```



```

        p.z = value * 100;
        mesh.setVertex( i, p );
        //Change color of vertex
        mesh.setColor( i,
                       ofColor( value*255, value * 255, 255 ) );
    }
}
setNormals( mesh ); //Update the normals
}

```

The `draw()` function draws the surface and slowly rotates it:

```

void testApp::draw(){
    ofEnableDepthTest(); //Enable z-buffering

    //Set a gradient background from white to gray
    ofBackgroundGradient( ofColor( 255 ), ofColor( 128 ) );

    ofPushMatrix(); //Store the coordinate system

    //Move the coordinate center to screen's center
    ofTranslate( ofGetWidth()/2, ofGetHeight()/2, 0 );

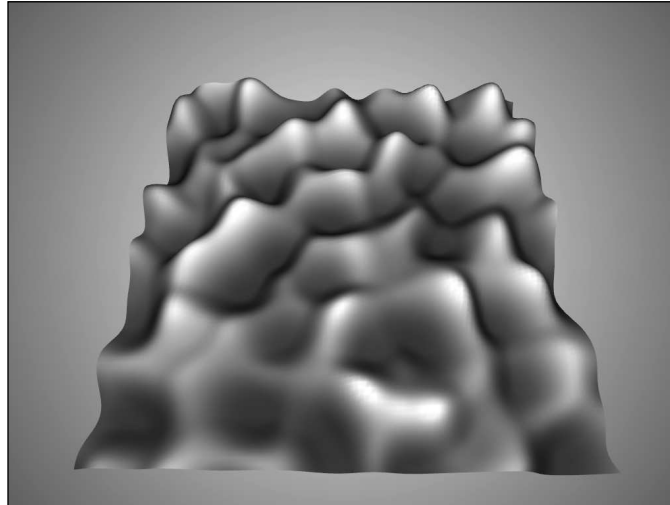
    //Calculate the rotation angle
    float time = ofGetElapsedTimef(); //Get time in seconds
    float angle = time * 20; //Compute angle. We rotate at speed
    //20 degrees per second
    ofRotate( 30, 1, 0, 0 ); //Rotate coordinate system
    ofRotate( angle, 0, 0, 1 );

    //Draw mesh
    //Here ofSetColor() does not affects the result of drawing,
    //because the mesh has its own vertices' colors
    mesh.draw();

    ofPopMatrix(); //Restore the coordinate system
}

```

Run the example and you will see a pulsating surface that slowly rotates on the screen:



Now replace in the `testApp::draw()` function in the line `mesh.draw()`; by the following line:


```
mesh.drawWireframe();
```

Now, run the project and you will see the wireframe structure of the surface.

Until now you knew how to create simple animated smooth surfaces and disconnected clouds of primitives. Let's consider an advanced example of constructing a smooth surface that grows and twists in space.

## The twisting knot example

In this example we will create a tube-like surface, that is formed from a number of deformed circles. At each `update()` call, we will generate one circle and connect it with the previous circle by adding triangles to the surface. At each step the circle will slowly move, rotate, and deform in space. As result, we will see a growing and twisting 3D knot.

[  This is example 07-3D/07-TwistingKnot. ]

The example is based on the `emptyExample` project in `openFrameworks`. Begin with adding declaration and definition of the `setNormals()` function, as is described in the *Computing normals using the `setNormals()` function* section. Then in the `testApp.h` file, in the `testApp` class declaration, add definitions of the `mesh`, `light`, and `addRandomCircle()` function:

```
ofMesh mesh;           //Mesh
ofLight light;        //Light
void addRandomCircle( ofMesh &mesh ); //Main function which
                        //moves circle and adds triangles to the object
```

In the beginning of the `testApp.cpp` file, add the constants and the variables for the circle that will be used for knot generation:

```
//The circle parameters
float Rad = 25;           //Radius of circle
float circleStep = 3;    //Step size for circle motion
int circleN = 40;        //Number of points on the circle

//Current circle state
ofPoint pos;             //Circle center
ofPoint axeX, axyY, axyZ; //Circle's coordinate system
```

The `setup()` function sets the initial values of the circle's position and also enables lighting with `light`, using its default settings:

```
void testApp::setup(){
    pos = ofPoint( 0, 0, 0 ); //Start from center of coordinate
    axeX = ofPoint( 1, 0, 0 ); //Set initial coordinate system
    axyY = ofPoint( 0, 1, 0 );
    axyZ = ofPoint( 0, 0, 1 );
    light.enable();          //Enable lighting
    ofSetFrameRate( 60 );   //Set the rate of screen redrawing
}
```

The `update()` function just calls the `addRandomCircle()` function, which adds one more circle to the knot:

```
void testApp::update(){
    addRandomCircle( mesh );
}
```

The `draw()` function draws the mesh on the screen. Note that we use the `mesh.getCentroid()` function, which returns the center of mass of mesh's vertex array. In other words, we apply it for the shift coordinate system of `Translate( -mesh.getCentroid() )`, which helps us to draw our object positioned in the center :

```
void ofApp::draw(){
    ofEnableDepthTest();    //Enable z-buffering

    //Set a gradient background from white to gray
    ofBackgroundGradient( ofColor( 255 ), ofColor( 128 ) );

    ofPushMatrix(); //Store the coordinate system
    //Move the coordinate center to screen's center
    ofTranslate( ofGetWidth()/2, ofGetHeight()/2, 0 );

    //Calculate the rotation angle
    float time = ofGetElapsedTimef(); //Get time in seconds
    float angle = time * 20;          //Compute the angle.
                                        //We rotate at speed 20 degrees per second
    ofRotate( angle, 0, 1, 0 );       //Rotate the coordinate system
                                        //along y-axis

    //Shift the coordinate center so the mesh
    //will be drawn in the screen center
    ofTranslate( -mesh.getCentroid() );

    //Draw the mesh
    //Here ofSetColor() does not affects the result of drawing,
    //because the mesh has its own vertices' colors
    mesh.draw();

    ofPopMatrix(); //Restore the coordinate system
}
```

The most important function in the example is `addRandomCircle()`. It pseudorandomly moves the circle, adds new vertices from the circle to the object's vertex array, and adds corresponding triangles to the object. It also sets colors for the new vertices.

```
void ofApp::addRandomCircle( ofMesh &mesh ){
    float time = ofGetElapsedTimef();    //Time

    //Parameters - twisting and rotating angles and color
    float twistAngle = 5.0 * ofSignedNoise( time * 0.3 + 332.4 );
    float rotateAngle = 1.5;
```

```
ofFloatColor color( ofNoise( time * 0.05 ),
                    ofNoise( time * 0.1 ),
                    ofNoise( time * 0.15 ));
color.setSaturation( 1.0 ); //Make the color maximally
                           //colorful

//Rotate the coordinate system of the circle
axeX.rotate( twistAngle, axyZ );
axyY.rotate( twistAngle, axyZ );

axeX.rotate( rotateAngle, axyY );
axyZ.rotate( rotateAngle, axyY );

//Move the circle on a step
ofPoint move = axyZ * circleStep;
pos += move;

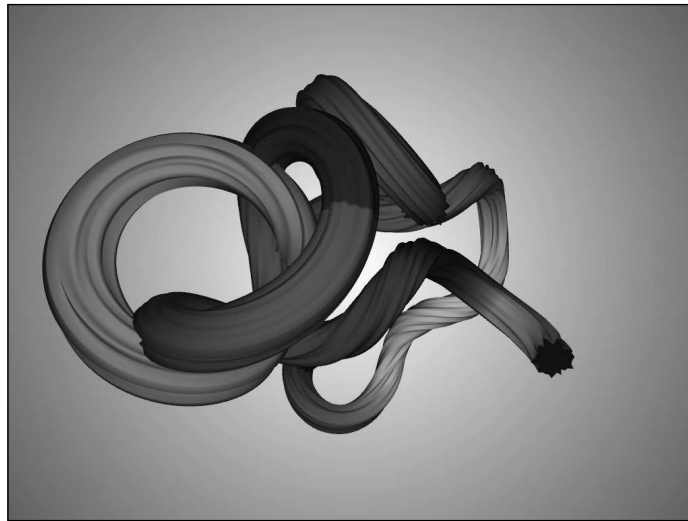
//Add vertices
for (int i=0; i<circleN; i++) {
    float angle = float(i) / circleN * TWO_PI;
    float x = Rad * cos( angle );
    float y = Rad * sin( angle );
    //We would like to distort this point
    //to make the knot's surface embossed
    float distort = ofNoise( x * 0.2, y * 0.2,
                             time * 0.2 + 30 );
    distort = ofMap( distort, 0.2, 0.8, 0.8, 1.2 );
    x *= distort;
    y *= distort;

    ofPoint p = axeX * x + axyY * y + pos;
    mesh.addVertex( p );
    mesh.addColor( color );
}

//Add the triangles
int base = mesh.getNumVertices() - 2 * circleN;
if ( base >= 0 ) { //Check if it is not the first step
    //and we really need to add the triangles
    for (int i=0; i<circleN; i++) {
        int a = base + i;
        int b = base + (i + 1) % circleN;
        int c = circleN + a;
        int d = circleN + b;
    }
}
```

```
        mesh.addTriangle( a, b, d ); //Clock-wise
        mesh.addTriangle( a, d, c );
    }
    //Update the normals
    setNormals( mesh );
}
}
```

Run the example and you will see a growing and twisting knot, as shown in the following screenshot:



Note that we control the rate of `testApp::update()` callings (and hence the `addRandomCircle()` rate) using the `ofSetFrameRate( 60 )` call in `testApp::setup()`. If you change the rate, say to `ofSetFrameRate( 30 )`, you will obtain a differently shaped knot. To make the resultant shape independent of frame rate, you should make the `circleStep` parameter dependent on the time between current and previous frames.



At each `update()` call, the application constantly adds new vertices and triangles to the object. Then it recalculates all the normals, though many of the triangles did not change. So application performance will degrade with time because the `setNormals()` function will take more and more computing power. To solve this problem, you can optimize the `setNormals()` function so it does not recalculate the unchanged normals and does not check the old triangles at all.

## Additional topics

In this chapter we mainly considered representing and drawing 3D objects using `openFrameworks`. For further learning, we suggest studying the following topics:

- Working with the `ofLight` class to control lights, that is, the type of light (spot light and point light), its position, light direction, and color parameters. See `openFrameworks` examples `examples/3d/normalsExample` and `examples/3d/advanced3dExample`.
- Working with the `ofCamera` and `ofEasyCam` classes to control the camera, that is, the position of the observer of the 3D scene. The camera lets you move easily through the virtual 3D world and also change perspective parameters. See `openFrameworks` examples `examples/3d/cameraRibbonExample` and `examples/3d/easyCamExample`.
- Using 3D model files with the `.3ds` and `.dae` extensions. You can load and draw such files as static or animated objects. Note that you can use 3D file models as a source of vertex data for further manipulation and processing. See `openFrameworks` examples `examples/3d/modelNoiseExample`, `examples/addons/3DModelLoaderExample`, and `examples/addons/assimpExample`.
- Rendering volumetric data using the **marching cubes** algorithm. This technique allows rendering isolines of an arbitrary function defined in some volume. It opens the possibility of drawing complex surfaces with constantly changing shape and number of connected components, such as metaballs. To use this algorithm, download and install the `ofxMarchingCubes` addon from `ofxaddons.com` and see its example. For more details on installing addons see *Appendix A, Working with Addons*.

## Summary

In this chapter we learned how to represent, modify, and draw 3D objects using the `ofMesh` class and also how to perform simple 3D drawing with the `ofTriangle()` function. We looked at examples of drawing a sphere-shaped cloud of triangles, a oscillating surface, and a twisting 3D knot.

In the next chapter, we will cover how to use shaders to process images and 3D object geometry.

**For More Information:**

[www.packtpub.com/mastering-openframeworks-creative-coding-demystified/book](http://www.packtpub.com/mastering-openframeworks-creative-coding-demystified/book)

## Where to buy this book

You can buy Mastering openFrameworks: Creative Coding Demystified from the Packt Publishing website: <http://www.packtpub.com/mastering-openframeworks-creative-coding-demystified/book>.

Free shipping to the US, UK, Europe and selected Asian countries. For more information, please read our [shipping policy](#).

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.



[www.PacktPub.com](http://www.PacktPub.com)

**For More Information:**

[www.packtpub.com/mastering-openframeworks-creative-coding-demystified/book](http://www.packtpub.com/mastering-openframeworks-creative-coding-demystified/book)