Once you've mastered this basic technique, you can use it to switch between any number of shader programs. This way you can use a variety of different drawing effects in a single scene.

Use What You've Drawn as a Texture Image

One simple but powerful technique is to draw some 3D objects and then use the resulting image as a texture image for another 3D object. Essentially, if you can use the content you've drawn as a texture image, you are able to generate images on-the-fly. This means you do not need to download images from the network, and you can apply special effects (such as motion blur and depth of field) before displaying the image. You can also use this technique for shadowing, which will be explained in the next section. Here, you will construct a sample program, `FramebufferObject`, which maps a rotating cube drawn with WebGL to a rectangle as a texture image. Figure 10.17 shows a screen shot.

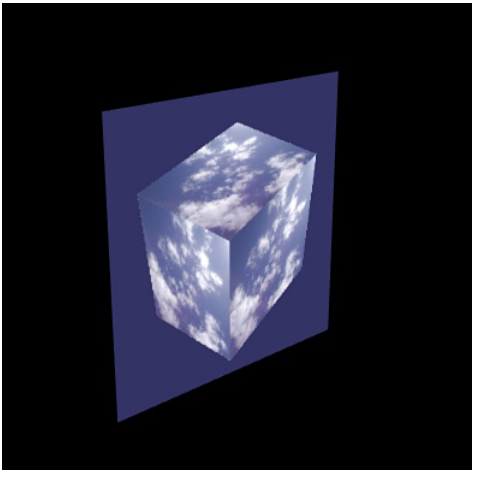

Figure 10.17 FramebufferObject

If you actually run the program, you can see a rotating cube with a texture image of a summer sky pasted to the rectangle as its texture. Significantly, the image of the cube that is pasted on the rectangle is not a movie prepared in advance but a rotating cube drawn by WebGL in real time. This is quite powerful, so let's take a look at what WebGL must do to achieve this.

Framebuffer Object and Renderbuffer Object

By default, the WebGL system draws using a color buffer and, when using the hidden surface removal function, a depth buffer. The final image is kept in the color buffer.

The **framebuffer object** is an alternative mechanism you can use instead of a color buffer or a depth buffer (Figure 10.18). Unlike a color buffer, the content drawn in a framebuffer

object is not directly displayed on the <canvas>. Therefore, you can use it if you want to perform different types of processing before displaying the drawn content. Or you can use it as a texture image. Such a technique is often referred to as **offscreen drawing**.

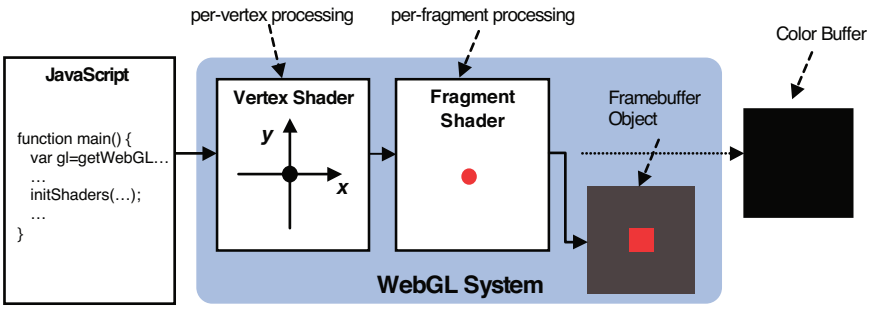

Figure 10.18 Framebuffer object

The framebuffer object has the structure shown in Figure 10.19 and supports substitutes for the color buffer and the depth buffer. As you can see, drawing is not carried out in the framebuffer itself, but in the drawing areas of the objects that the framebuffer points to. These objects are attached to the framebuffer using its **attachment** function. A **color attachment** specifies the destination for drawing to be a replacement for the color buffer. A **depth attachment** and a **stencil attachment** specify the replacements for the depth buffer and stencil buffer.

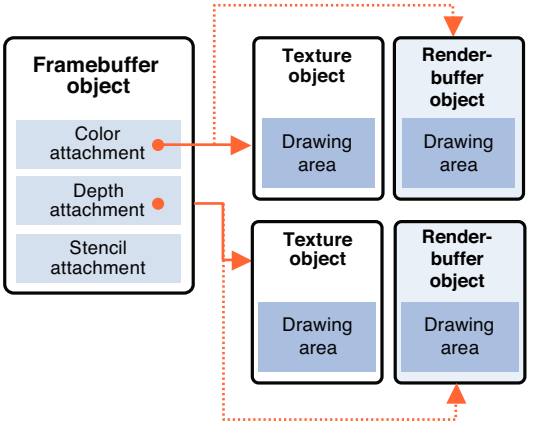

Figure 10.19 Framebuffer object, texture object, renderbuffer object

WebGL supports two types of objects that can be used to draw objects within: the texture object that you saw in Chapter 5, and the **renderbuffer object**. With the texture object, the content drawn into the texture object can be used as a texture image. The render-buffer object is a more general-purpose drawing area, allowing a variety of data types to be written.

How to Implement Using a Drawn Object as a Texture

When you want to use the content drawn into a framebuffer object as a texture object, you actually need to use the content drawn into the color buffer for the texture object. Because you also want to remove the hidden surfaces for drawing, you will set up the framebuffer object as shown in Figure 10.20.

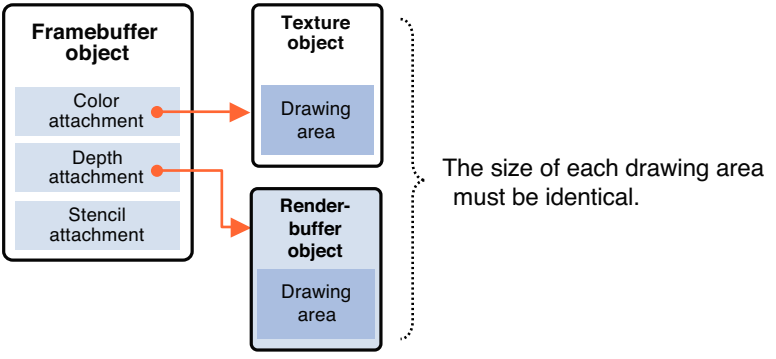

Figure 10.20 Configuration of framebuffer object when using drawn content as a texture

The following eight steps are needed for realizing this configuration. These processes are similar to the process for the buffer object. Step 2 was explained in Chapter 5, so there are essentially seven new processes:

1. Create a framebuffer object (`gl.createFramebuffer()`).
2. Create a texture object and set its size and parameters (`gl.createTexture()`, `gl.bindTexture()`, `gl.texImage2D()`, `gl.Parameteri()`).
3. Create a renderbuffer object (`gl.createRenderbuffer()`).
4. Bind the renderbuffer object to the target and set its size (`gl.bindRenderbuffer()`, `gl.renderbufferStorage()`).
5. Attach the texture object to the color attachment of the framebuffer object (`gl.bindFramebuffer()`, `gl.framebufferTexture2D()`).
6. Attach the renderbuffer object to the depth attachment of the framebuffer object (`gl.framebufferRenderbuffer()`).
7. Check whether the framebuffer object is configured correctly (`gl.checkFramebuffer-Status()`).
8. Draw using the framebuffer object (`gl.bindFramebuffer()`).

Now let's look at the sample program. The numbers in the sample program indicate the code used to implement the steps.

Sample Program (FramebufferObjectj.js)

Steps 1 to 7 of `FramebufferObject.js` are shown in Listing 10.13.

Listing 10.13 FramebufferObject.js (Processes for Steps 1 to 7)

```
 1  // FramebufferObject.js
      ...
24  // Size of offscreen
25  var OFFSCREEN_WIDTH = 256;
26  var OFFSCREEN_HEIGHT = 256;
27
28  function main() {
      ...
55    // Set vertex information
56    var cube = initVertexBuffersForCube(gl);
57    var plane = initVertexBuffersForPlane(gl);
      ...
64    var texture = initTextures(gl);
      ...
70    // Initialize framebuffer object (FBO)
71    var fbo = initFramebufferObject(gl);
      ...
80    var viewProjMatrix = new Matrix4();// For color buffer
81    viewProjMatrix.setPerspective(30, canvas.width/canvas.height, 1.0, 100.0);
82    viewProjMatrix.lookAt(0.0, 0.0, 7.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0);
83
84    var viewProjMatrixFBO = new Matrix4();  // For FBO
85    viewProjMatrixFBO.setPerspective(30.0, OFFSCREEN_WIDTH/OFFSCREEN_HEIGHT,
                                                                  ➥1.0, 100.0);
86    viewProjMatrixFBO.lookAt(0.0, 2.0, 7.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0);
      ...
92      draw(gl, canvas, fbo, plane, cube, currentAngle, texture, viewProjMatrix,
                                                          ➥viewProjMatrixFBO);
      ...
96  }
      ...
263 function initFramebufferObject(gl) {
264   var framebuffer, texture, depthBuffer;
      ...
274   // Create a framebuffer object (FBO)                              <-(1)
275   framebuffer = gl.createFramebuffer();
      ...
281   // Create a texture object and set its size and parameters        <-(2)
282   texture = gl.createTexture(); // Create a texture object
      ...
```

```
287    gl.bindTexture(gl.TEXTURE_2D, texture);
288    gl.texImage2D(gl.TEXTURE_2D, 0, gl.RGBA, OFFSCREEN_WIDTH,
                          ➥OFFSCREEN_HEIGHT, 0, gl.RGBA, gl.UNSIGNED_BYTE, null);
289    gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER, gl.LINEAR);
290    framebuffer.texture = texture;          // Store the texture object
291
292    // Create a renderbuffer object and set its size and parameters
293    depthBuffer = gl.createRenderbuffer();// Create a renderbuffer          <-(3)
       ...
298    gl.bindRenderbuffer(gl.RENDERBUFFER, depthBuffer);                     <-(4)
299    gl.renderbufferStorage(gl.RENDERBUFFER, gl.DEPTH_COMPONENT16,
                                          ➥OFFSCREEN_WIDTH, OFFSCREEN_HEIGHT);
300
301    // Attach the texture and the renderbuffer object to the FBO
302    gl.bindFramebuffer(gl.FRAMEBUFFER, framebuffer);
303    gl.framebufferTexture2D(gl.FRAMEBUFFER, gl.COLOR_ATTACHMENT0,
                                        ➥gl.TEXTURE_2D, texture, 0);     <-(5)
304    gl.framebufferRenderbuffer(gl.FRAMEBUFFER, gl.DEPTH_ATTACHMENT,
                                      ➥gl.RENDERBUFFER, depthBuffer);    <-(6)
305
306    // Check whether FBO is configured correctly                          <-(7)
307    var e = gl.checkFramebufferStatus(gl.FRAMEBUFFER);
308    if (e !== gl.FRAMEBUFFER_COMPLETE) {
309      console.log('Framebuffer object is incomplete: ' + e.toString());
310      return error();
311    }
312
       ...
319    return framebuffer;
320  }
```

The vertex shader and fragment shader are omitted because this sample program uses the same shaders as `TexturedQuad.js` in Chapter 5, which pasted a texture image on a rectangle. The sample program in this section draws two objects: a cube and a rectangle. Just as you did in `ProgramObject.js` in the previous section, you assign multiple buffer objects needed for drawing each object as properties of an `Object` object. Then you store the object to the variables `cube` and `plane`. You will use them for drawing by assigning each buffer in the object to the attribute variable.

The key point of this program is the initialization of the framebuffer object by `initFramebufferObject()` at line 71. The initialized framebuffer object is stored in a variable `fbo` and passed as the third argument of `draw()` at line 92. You'll return to the function `draw()` later. For now let's examine `initFramebufferObject()`, at line 263, step by step. This function performs steps 1 to 7. The view projection matrix for the framebuffer object is prepared separately at line 84 because it is different from the one used for a color buffer.

Create Frame Buffer Object (gl.createFramebuffer())

You must create a framebuffer object before you can use it. The sample program creates it at line 275:

```
275 framebuffer = gl.createFramebuffer();
```

You will use `gl.createFramebuffer()` to create the framebuffer object.

`gl.createFramebuffer()`

Create a framebuffer object.

Parameters	None	
Return value	non-null	The newly created framebuffer object.
	null	Failed to create a framebuffer object.
Errors	None	

You use `gl.deleteFramebuffer()` to delete the created framebuffer object.

`gl.deleteFramebuffer(framebuffer)`

Delete a framebuffer object.

Parameters	framebuffer	Specifies the framebuffer object to be deleted.
Return value	None	
Errors	None	

Once you have created the framebuffer object, you need to attach a texture object to the color attachment and a renderbuffer object to the depth attachment in the framebuffer object. Let's start by creating the texture object for the color attachment.

Create Texture Object and Set Its Size and Parameters

You have already seen how to create a texture object and set up its parameters (`gl.TEXTURE_MIN_FILTER`) in Chapter 5. You should note that its width and height are `OFFSCREEN_WIDTH` and `OFFSCREEN_HEIGHT`, respectively. The size is smaller than that of the `<canvas>` to make the drawing process faster.

```
282   texture = gl.createTexture(); // Create a texture object
       ...
287   gl.bindTexture(gl.TEXTURE_2D, texture);
```

```
288   gl.texImage2D(gl.TEXTURE_2D, 0, gl.RGBA, OFFSCREEN_WIDTH, OFFSCREEN_HEIGHT, 0,
                                                ➥gl.RGBA, gl.UNSIGNED_BYTE, null);
289   gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER, gl.LINEAR);
290   framebuffer.texture = texture; // Store the texture object
```

The `gl.texImage2D()` at line 288 allocates a drawing area in a texture object. You can allocate a drawing area by specifying `null` to the last argument, which is used to specify an `Image` object. You will use this texture object later, so store it in `framebuffer.texture` at line 290.

That completes the preparation for a texture object that is attached to the color attachment. Next, you need to create a renderbuffer object for the depth buffer.

Create Renderbuffer Object (gl.createRenderbuffer())

Like texture buffers, you need to create a renderbuffer object before using it. The sample program does this at line 293.

```
293   depthBuffer = gl.createRenderbuffer();  // Create a renderbuffer
```

You use `gl.createRenderbuffer()` to create the renderbuffer object.

`gl.createRenderbuffer()`

Create a renderbuffer object.

Parameters	None	
Return value	Non-null	The newly created renderbuffer object.
	Null	Failed to create a renderbuffer object.
Errors	None	

You use `gl.deleteRenderbuffer()` to delete the created renderbuffer object.

`gl.deleteRenderbuffer(renderbuffer)`

Delete a renderbuffer object.

Parameters	renderbuffer	Specifies the renderbuffer object to be deleted.
Return value	None	
Errors	None	

The created renderbuffer object is used as a depth buffer here, so you store it in a variable named `depthBuffer`.

Bind Renderbuffer Object to Target and Set Size (gl.bindRenderbuffer(), gl.renderbufferStorage())

When using the created renderbuffer object, you need to bind the renderbuffer object to a target and perform the operation on that target.

```
298    gl.bindRenderbuffer(gl.RENDERBUFFER, depthBuffer);
299    gl.renderbufferStorage(gl.RENDERBUFFER, gl.DEPTH_COMPONENT16,
                                                ➥OFFSCREEN_WIDTH, OFFSCREEN_HEIGHT);
```

The renderbuffer object is bound to a target with `gl.bindRenderbuffer()`.

`gl.bindRenderbuffer(target, renderbuffer)`

Bind the renderbuffer object specified by *renderbuffer* to *target*. If `null` is specified as *renderbuffer*, the *renderbuffer* is unbound from the *target*.

Parameters	target	Must be `gl.RENDERBUFFER`.
	renderbuffer	Specifies the renderbuffer object.
Return value	None	
Errors	INVALID_ENUM	*target* is not `gl.RENDERBUFFER`

When the binding is complete, you can set the format, width, and height of the renderbuffer object by using `gl.renderbufferStorage()`. You must set the same width and height as the texture object that is used as the color attachment.

`gl.renderbufferStorage(target, internalformat, width, height)`

Create and initialize a renderbuffer object's data store.

Parameters	target	Must be `gl.RENDERBUFFER`.
	internalformat	Specifies the format of the renderbuffer.
	`gl.DEPTH_COMPONENT16`	The renderbuffer is used as a depth buffer.
	`gl.STENCIL_INDEX8`	The renderbuffer is used as a stencil buffer.

	`gl.RGBA4`
	`gl.RGB5_A1`
	`gl.RGB565`
		The renderbuffer is used as a color buffer. `gl.RGBA4` (each RGBA component has 4, 4, 4, and 4 bits, respectively), `gl.RGB5_A1` (each RGB component has 5 bits, and A has 1 bit), `gl.RGB565` (each RGB component has 5, 6, and 5 bits, respectively)

	width, height	Specifies the width and height of the renderbuffer in pixels.
Return value	None	
Errors	INVALID_ENUM	Target is not `gl.RENDERBUFFER` or *internalformat* is none of the preceding values.
	INVALID_OPERATION	No renderbuffer is bound to *target*.

The preparations of the texture object and renderbuffer object of the framebuffer object are now complete. At this stage, you can use the object for offscreen drawing.

Set Texture Object to Framebuffer Object (gl.bindFramebuffer(), gl.framebufferTexture2D())

You use a framebuffer object in the same way you use a renderbuffer object: You need to bind it to a target and operate on the target, not the framebuffer object itself.

```
302    gl.bindFramebuffer(gl.FRAMEBUFFER, framebuffer);  // Bind to target
303    gl.framebufferTexture2D(gl.FRAMEBUFFER, gl.COLOR_ATTACHMENT0, gl.TEXTURE_2D,
                                                                    ➥texture, 0);
```

A framebuffer object is bound to a target with `gl.bindFramebuffer()`.

`gl.bindFramebuffer(target, framebuffer)`

Bind a framebuffer object to a target. If *framebuffer* is `null`, the binding is broken.

Parameters	target	Must be `gl.FRAMEBUFFER`.
	framebuffer	Specify the framebuffer object.
Return value	None	
Errors	INVALID_ENUM	*target* is not `gl.FRAMEBUFFER`

Once the framebuffer object is bound to *target*, you can use the *target* to write a texture object to the framebuffer object. In this sample, you will use the texture object instead of a color buffer so you attach the texture object to the color attachment of the framebuffer.

You can assign the texture object to the framebuffer object with `gl.framebufferTexture2D()`.

`gl.framebufferTexture2D(target, attachment, textarget, texture, level)`

Attach a texture object specified by *texture* to the framebuffer object bound by *target*.

Parameters	target	Must be `gl.FRAMEBUFFER`.
	attachment	Specifies the attachment point of the framebuffer.
	`gl.COLOR_ATTACHMENT0`	*texture* is used as a color buffer
	`gl.DEPTH_ATTACHMENT`	*texture* is used as a depth buffer
	textarget	Specifies the first argument of `gl.texImage2D()` (`gl.TEXTURE_2D` or `gl.CUBE_MAP_TEXTURE`).
	texture	Specifies a texture object to attach to the framebuffer attachment point.
	level	Specifies 0 (if you use a MIPMAP in *texture*, you should specify its level).
Return value	None	
Errors	INVALID_ENUM	*target* is not `gl.FRAMEBUFFER`. *attachment* or *textarget* is none of the preceding values.
	INVALID_VALUE	*level* is not valid.
	INVALID_OPERATION	No framebuffer object is bound to *target*.

The 0 in the `gl.COLOR_ATTACHMENT0` used for the *attachment* parameter is because a framebuffer object in OpenGL, the basis of WebGL, can hold multiple color attachments (`gl.COLOR_ATTACHMENT0`, `gl.COLOR_ATTACHMENT1`, `gl.COLOR_ATTACHMENT2`...). However, WebGL can use just one of them.

Once the color attachment has been attached to the framebuffer object, you need to assign a renderbuffer object as a depth attachment. This follows a similar process.

Set Renderbuffer Object to Framebuffer Object (gl.framebufferRenderbuffer())

You will use `gl.framebufferRenderbuffer()` to attach a renderbuffer object to a framebuffer object. You need a depth buffer because this sample program will remove hidden surfaces. So the depth attachment needs to be attached.

```
304    gl.framebufferRenderbuffer(gl.FRAMEBUFFER, gl.DEPTH_ATTACHMENT,
gl.RENDERBUFFER, depthBuffer);
```

gl.framebufferRenderbuffer(target, attachment, renderbuffertarget, renderbuffer)

Attach a renderbuffer object specified by *renderbuffer* to the framebuffer object bound by *target*.

Parameters	target	Must be `gl.FRAMEBUFFER`.
	attachment	Specifies the attachment point of the framebuffer.
	`gl.COLOR_ATTACHMENT0`	*renderbuffer* is used as a color buffer.
	`gl.DEPTH_ATTACHMENT`	*renderbuffer* is used as a depth buffer.
	`gl.STENCIL_ATTACHMENT`	*renderbuffer* is used as a stencil buffer.
	renderbuffertarget	Must be `gl.RENDERBUFFER`.
	renderbuffer	Specifies a renderbuffer object to attach to the framebuffer attachment point
Return value	None	
Errors	INVALID_ENUM	*target* is not a `gl.FRAMEBUFFER`. *attachment* is none of the above values. *renderbuffertarget* is not `gl.RENDERBUFFER`.

Now that you've completed the preparation of the color attachment and depth attachment to the framebuffer object, you are ready to draw. But before that, let's check that the configuration of the framebuffer object is correct.

Check Configuration of Framebuffer Object (gl.checkFramebufferStatus())

Obviously, when you use a framebuffer that is not correctly configured, an error occurs. As you have seen in the past few sections, preparing a texture object and renderbuffer object that are needed to configure the framebuffer object is a complex process that sometimes generates mistakes. You can check whether the created framebuffer object is configured correctly and is available with `gl.checkFramebufferStatus()`.

```
307    var e = gl.checkFramebufferStatus(gl.FRAMEBUFFER);          <- (7)
308    if (gl.FRAMEBUFFER_COMPLETE !== e) {
309      console.log('Frame buffer object is incomplete:' + e.toString());
310      return error();
311    }
```

The following shows the specification of `gl.checkFramebufferStatus()`.

`gl.checkFramebufferStatus(target)`

Check the completeness status of a framebuffer bound to *target*.

Parameters	target	Must be `gl.FRAMEBUFFER`.
Return value	0	*Target* is not `gl.FRAMEBUFFER`.
	Others	
	`gl.FRAMEBUFFER_COMPLETE`	The framebuffer object is configured correctly.
	`gl.FRAMEBUFFER_INCOMPLETE_ATTACHMENT`	One of the framebuffer attachment points is incomplete. (The attachment is not sufficient. The texture object or the renderbuffer object is invalid.)
	`gl.FRAMEBUFFER_INCOMPLETE_DIMENSIONS`	The width or height of the texture object or renderbuffer object of the attachment is different.
	`gl.FRAMEBUFFER_INCOMPLETE_MISSING_ATTACHMENT`	The framebuffer does not have at least one valid attachment.
Errors	INVALID_ENUM	*target* is not `gl.FRAMEBUFFER`.

That completes the preparation of the framebuffer object. Let's now take a look at the `draw()` function.

Draw Using the Framebuffer Object

Listing 10.14 shows `draw()`. It switches the drawing destination to `fbo` (the framebuffer) and draws a cube in the texture object. Then `drawTexturedPlane()` uses the texture object to draw a rectangle to the color buffer.

Listing 10.14 FramebufferObject.js (Process of (8))

```
321  function draw(gl, canvas, fbo, plane, cube, angle, texture, viewProjMatrix,
                                                          ➥viewProjMatrixFBO) {
322    gl.bindFramebuffer(gl.FRAMEBUFFER, fbo);                              <-(8)
323    gl.viewport(0, 0, OFFSCREEN_WIDTH, OFFSCREEN_HEIGHT); // For FBO
324
325    gl.clearColor(0.2, 0.2, 0.4, 1.0); // Color is slightly changed
326    gl.clear(gl.COLOR_BUFFER_BIT | gl.DEPTH_BUFFER_BIT);  // Clear FBO
```

```
327    // Draw the cube
328    drawTexturedCube(gl, gl.program, cube, angle, texture, viewProjMatrixFBO);
329    // Change the drawing destination to color buffer
330    gl.bindFramebuffer(gl.FRAMEBUFFER, null);
331    // Set the size of view port back to that of <canvas>
332    gl.viewport(0, 0, canvas.width, canvas.height);
333    gl.clearColor(0.0, 0.0, 0.0, 1.0);
334    gl.clear(gl.COLOR_BUFFER_BIT | gl.DEPTH_BUFFER_BIT);
335    // Draw the plane
336    drawTexturedPlane(gl, gl.program, plane, angle, fbo.texture, viewProjMatrix);
337  }
```

Line 322 switches the drawing destination to the framebuffer object using `gl.bindFramebuffer()`. As a result, draw operations using `gl.drawArrays()` or `gl.drawElements()` are performed for the framebuffer object. Line 332 uses `gl.viewport()` to specify the draw area in the buffer (an offscreen area).

`gl.viewport(x, y, width, height)`

Set the viewport where `gl.drawArrays()` or `gl.drawElements()` draws. In WebGL, x and y are specified in the `<canvas>` coordinate system.

Parameters	x, y	Specify the lower-left corner of the viewport rectangle (in pixels).
	width, height	Specify the width and height of the viewport (in pixels).
Return value	None	
Errors	None	

Line 326 clears the texture image and the depth buffer bound to the framebuffer object. When a cube is drawn at line 328, it is drawn in the texture image. To make it easier to see the result, the clear color at line 325 is changed to a purplish blue from black. The result of this is that the cube has been drawn into the texture buffer and is now available for use as a texture image. The next step is to draw a rectangle (`plane`) using this texture image. In this case, because you want to draw in the color buffer, you need to set the drawing destination back to the color buffer. This is done at line 330 by specifying `null` for the second argument of `gl.bindFramebuffer()` (that is, cancelling the binding). Then line 336 draws the `plane`. You should note that `fbo.texture` is passed as the texture argument and used to map the drawn content to the rectangle. You will notice that in this sample program, the texture image is mapped onto the back side of the rectangle. This is because WebGL, by default, draws both sides of a polygon. You can eliminate the back face drawing by enabling the **culling function** using `gl.enable(gl.CULL_FACE)`, which increases the drawing speed (ideally making it twice as fast).