COSC 414/519I: Computer Graphics

2023

Shan Du

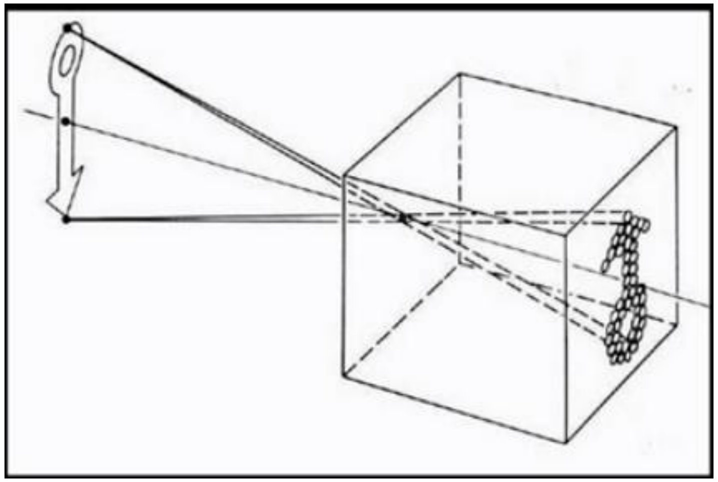
Image Formation

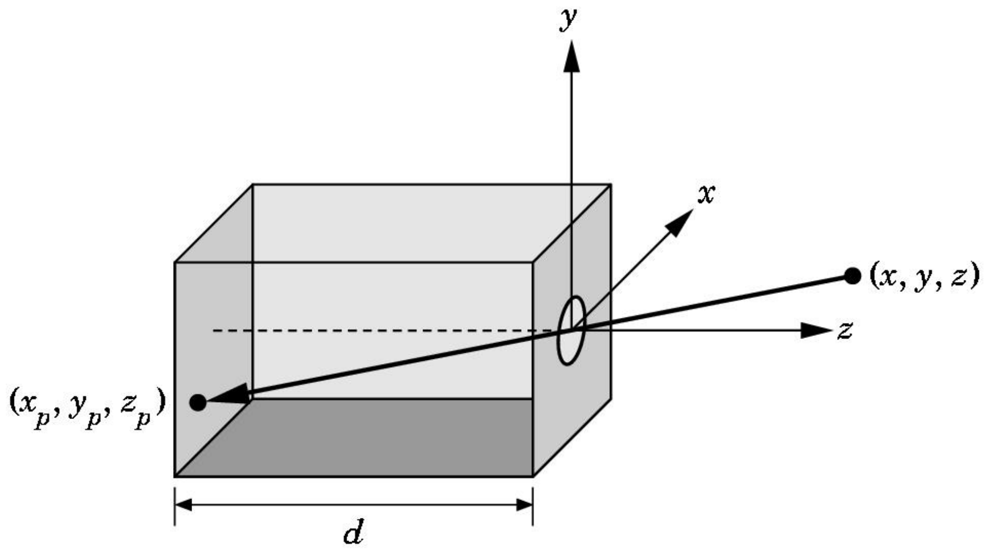
- In computer graphics, we form images which are generally two dimensional using a process analogous to how images are formed by physical imaging systems
  - Cameras
  - Microscopes
  - Telescopes
  - Human visual system

Imaging Systems

- The pinhole camera is a simple example of an imaging system that will enable us to understand the functioning of cameras and of other optical images.
- A pinhole camera is a box with a small hole in the center of one side; the film is placed inside the box on the side opposite the pinhole.

Pinhole Camera

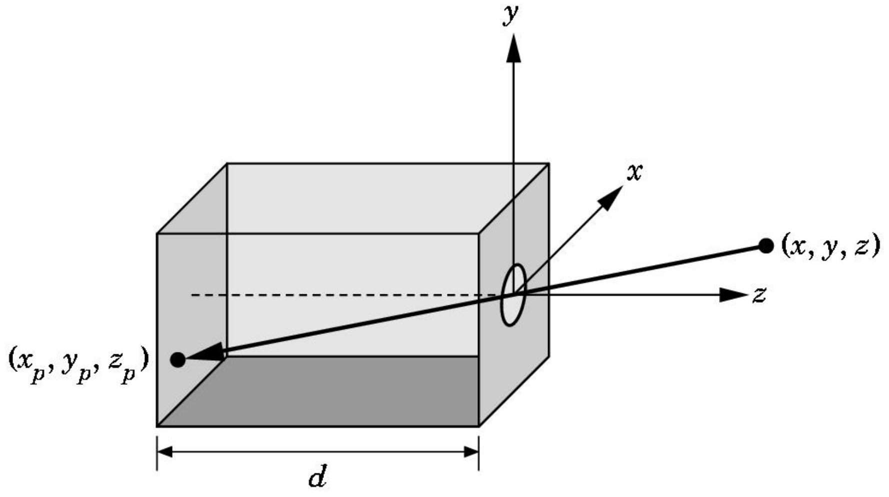




- Suppose that we orient our camera along the z-axis, with the pinhole at the origin of our coordinate system. We assume that the hole is so small that only a single ray of light, emanating from a point, can enter it.

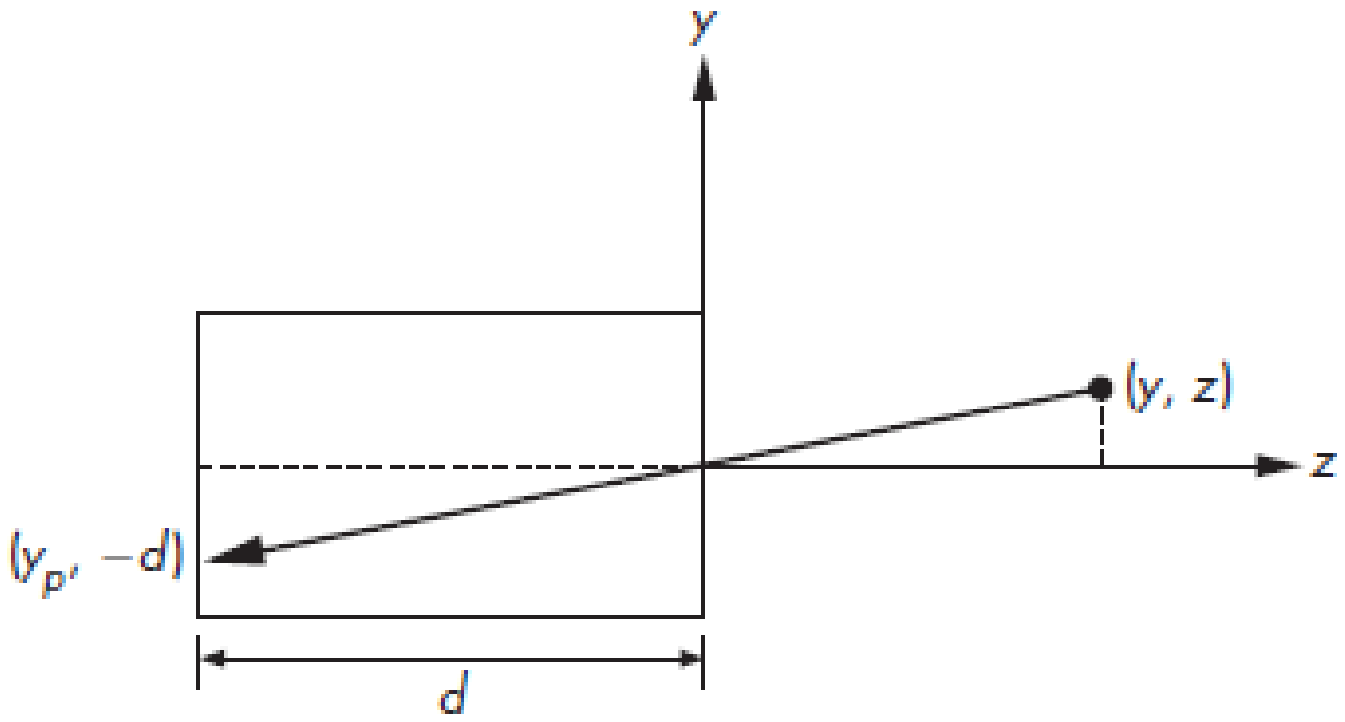
Pinhole Camera

- The film plane is located a distance *d* from the pinhole.



- Use trigonometry to find projection of point at (x, y, z)

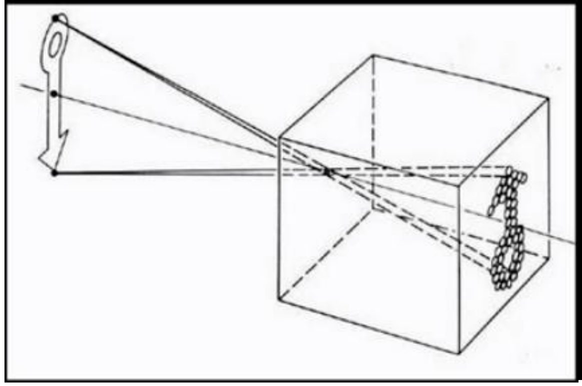
$$x_p = -\frac{x}{z/d} \quad y_p = -\frac{y}{z/d} \qquad z_p = d$$



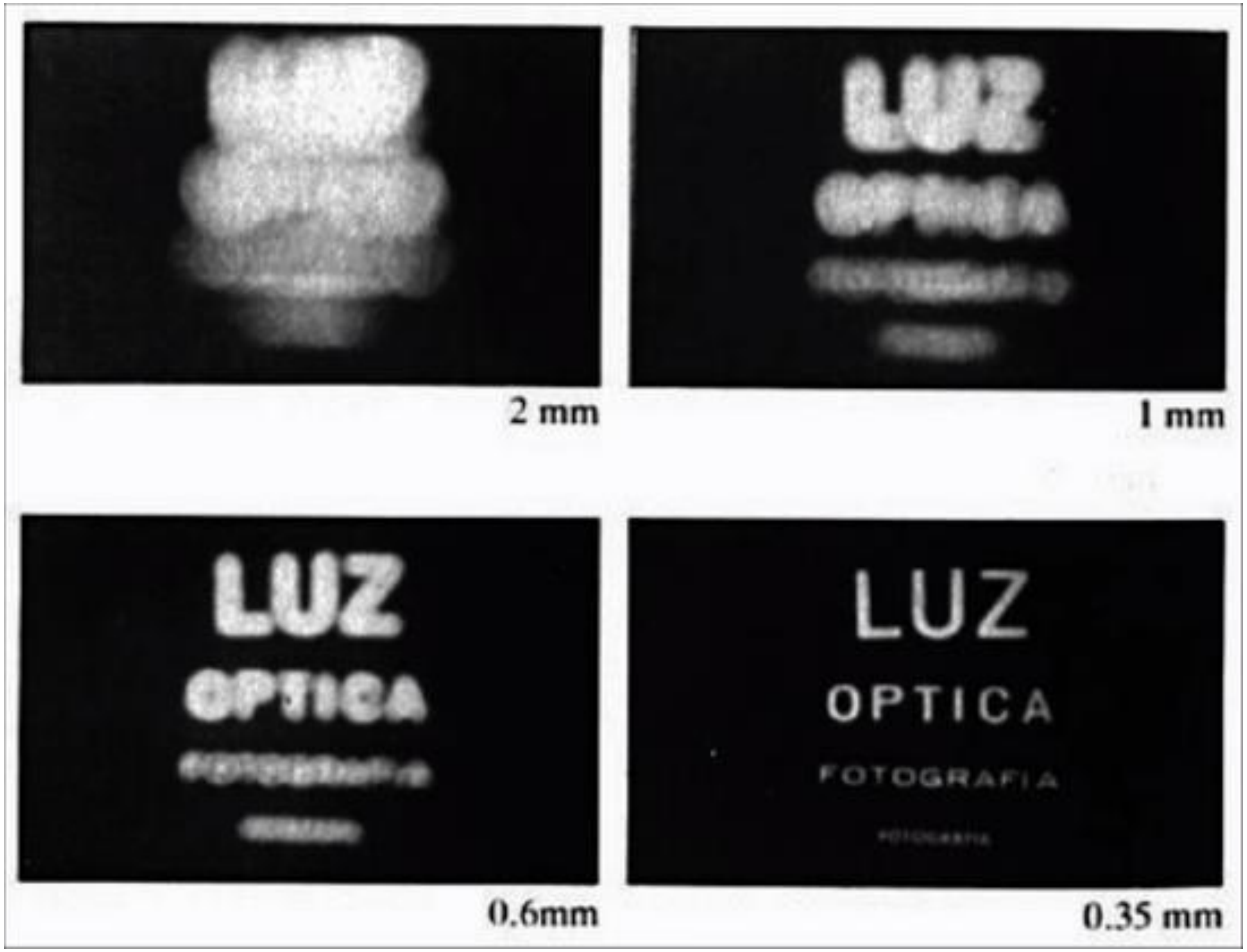
Side view of pinhole camera

The point (x_p, y_p, d) is called the projection of the point (x, y, z).

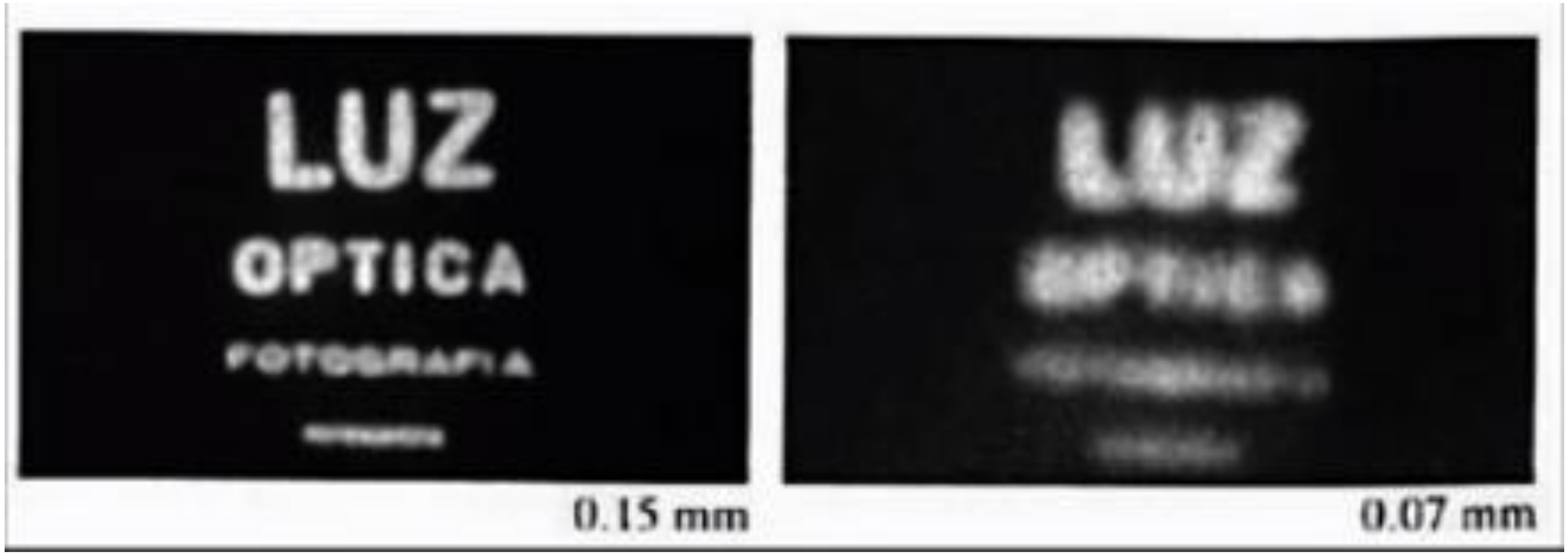
Pinhole Camera



Pinhole (Aperture)



Pinhole (Aperture)

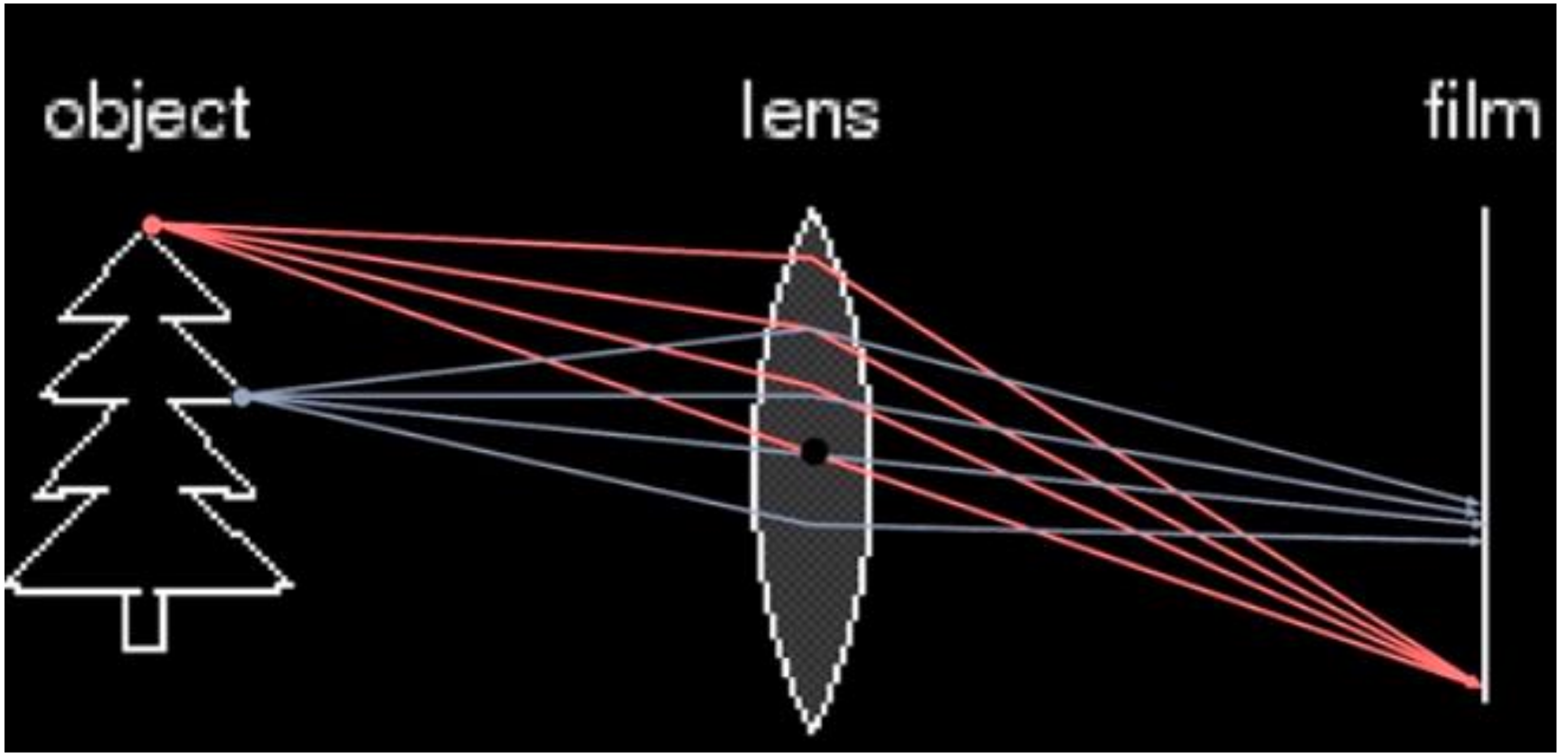


Diffraction

Replacing A Pinhole With A Lens

- The lens gathers more light that can pass through the pinhole. The larger the aperture of the lens, the more light the lens can collect.
- By picking a lens with the proper focal length - a selection equivalent to choosing *d* for the pinhole camera – we can achieve any desired field of view.
- However, lenses do not have an infinite depth of field: Not all distances from the lens are in focus.

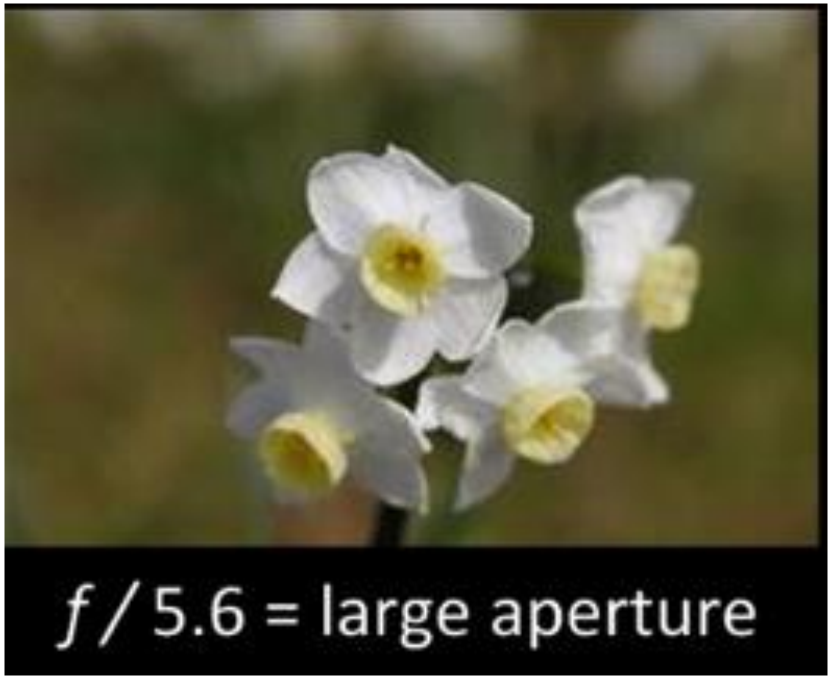
Lens and Focus

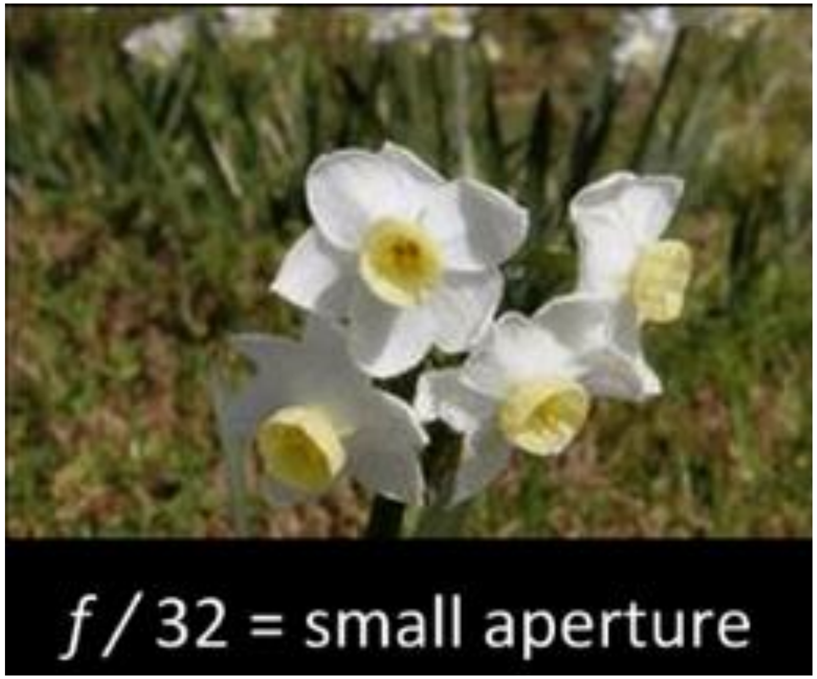


Circle of Confusion

Depth of Field

- The ideal pinhole camera has an infinite depth of field: Every point within its field of view is in focus.

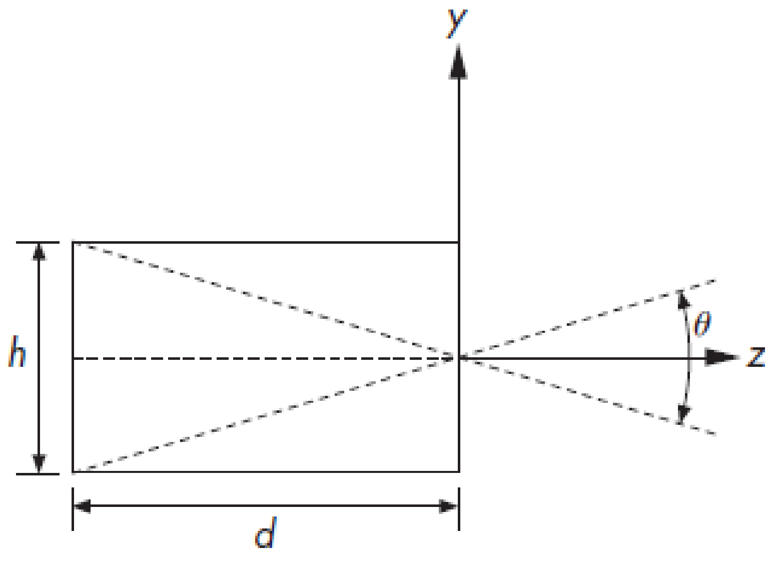




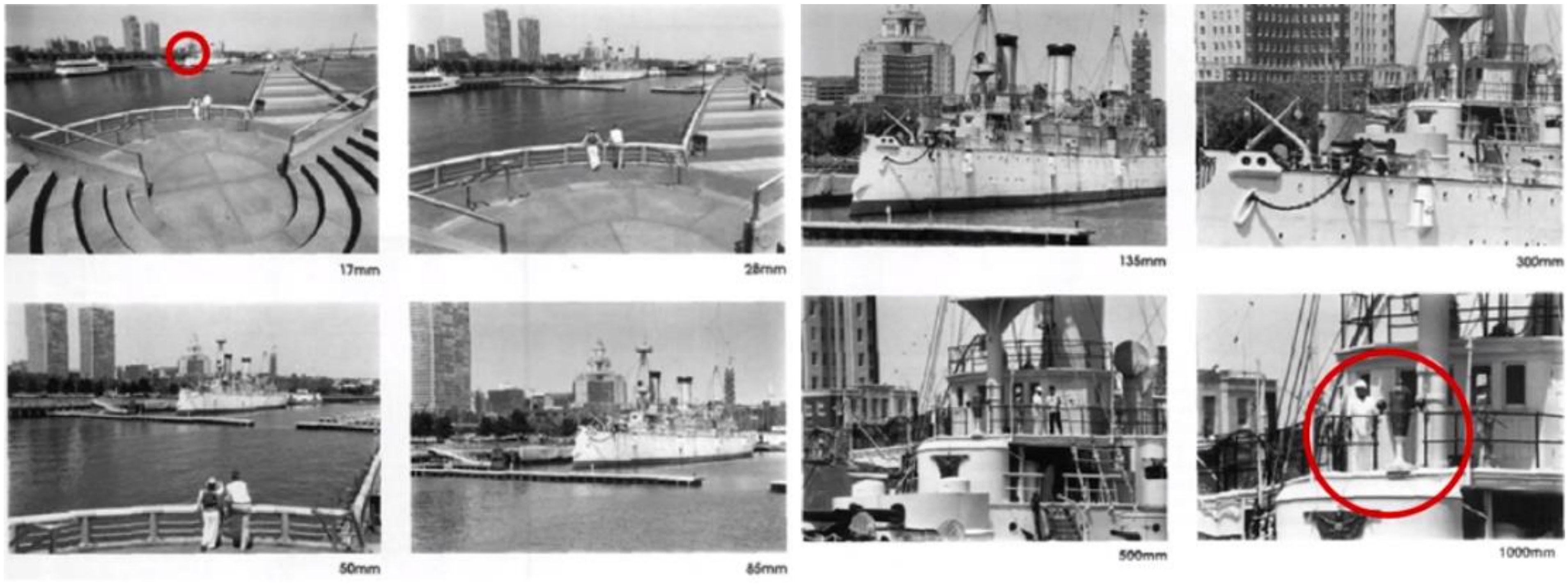
Field (Angle) of View

- Field of view describes the viewable area that can be imaged by a lens system. We can use angle to denote it where h is the sensor size.

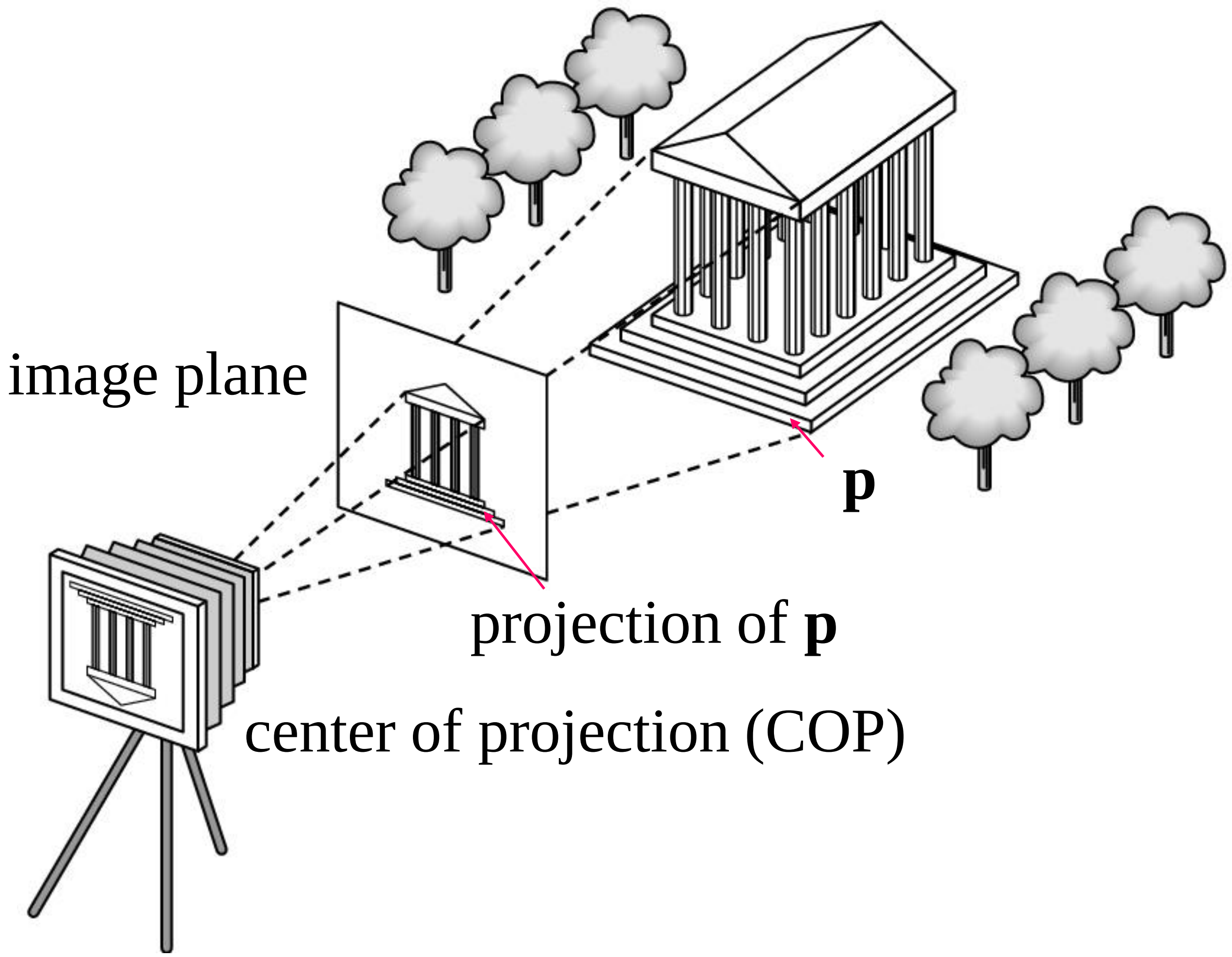
$$\theta = 2tan^{-1}\frac{h}{2d}$$



Field of View

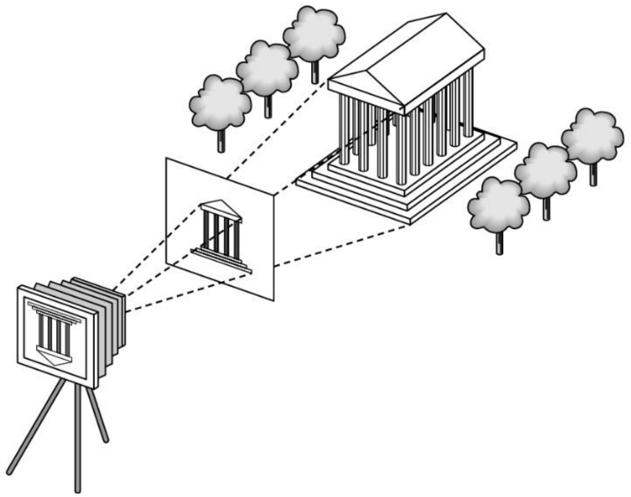


Synthetic Camera Model



Synthetic Camera Model

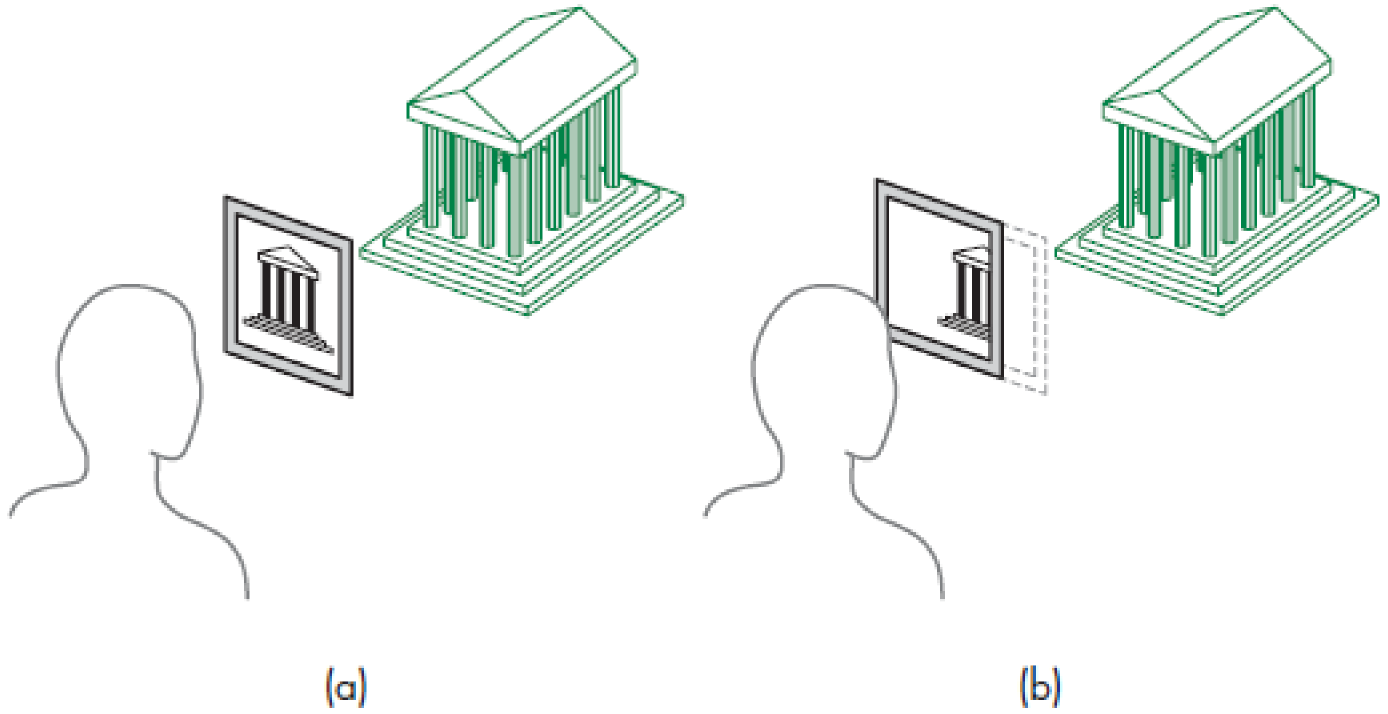
- We can find the image of a point on the object on the virtual image plane by drawing a line, called a **projector**, from the point to the center of the lens, or the **center of projection (COP)**.
- The virtual image plane that we have moved in front of the lens is called the **projection plane**.
- The image of the point is located where the projector passes through the projection plane.



Clipping Window

- We must also consider the limited size of the image. Not all objects can be imaged onto the pinhole camera’s film plane.
- Place a clipping rectangle, or clipping window, in the projection plane.
- This rectangle acts as a window through which a viewer, located at the COP, sees the world.
- Given the location of the COP, the location and orientation of the projection plane, and the size of the clipping rectangle, we can determine which objects will appear in the image.

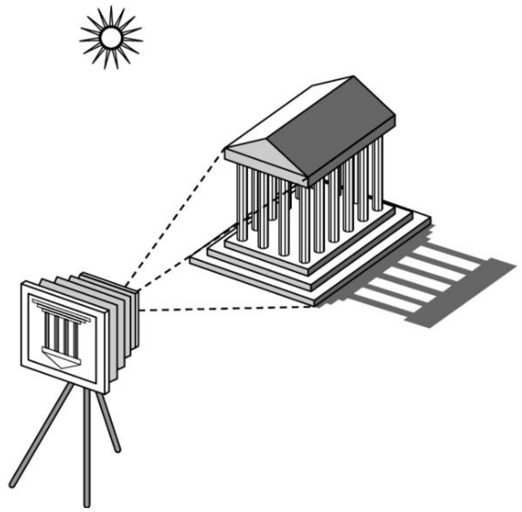
Clipping Window



(a) Window in initial position. (b) Window shifted.

Elements of Image Formation

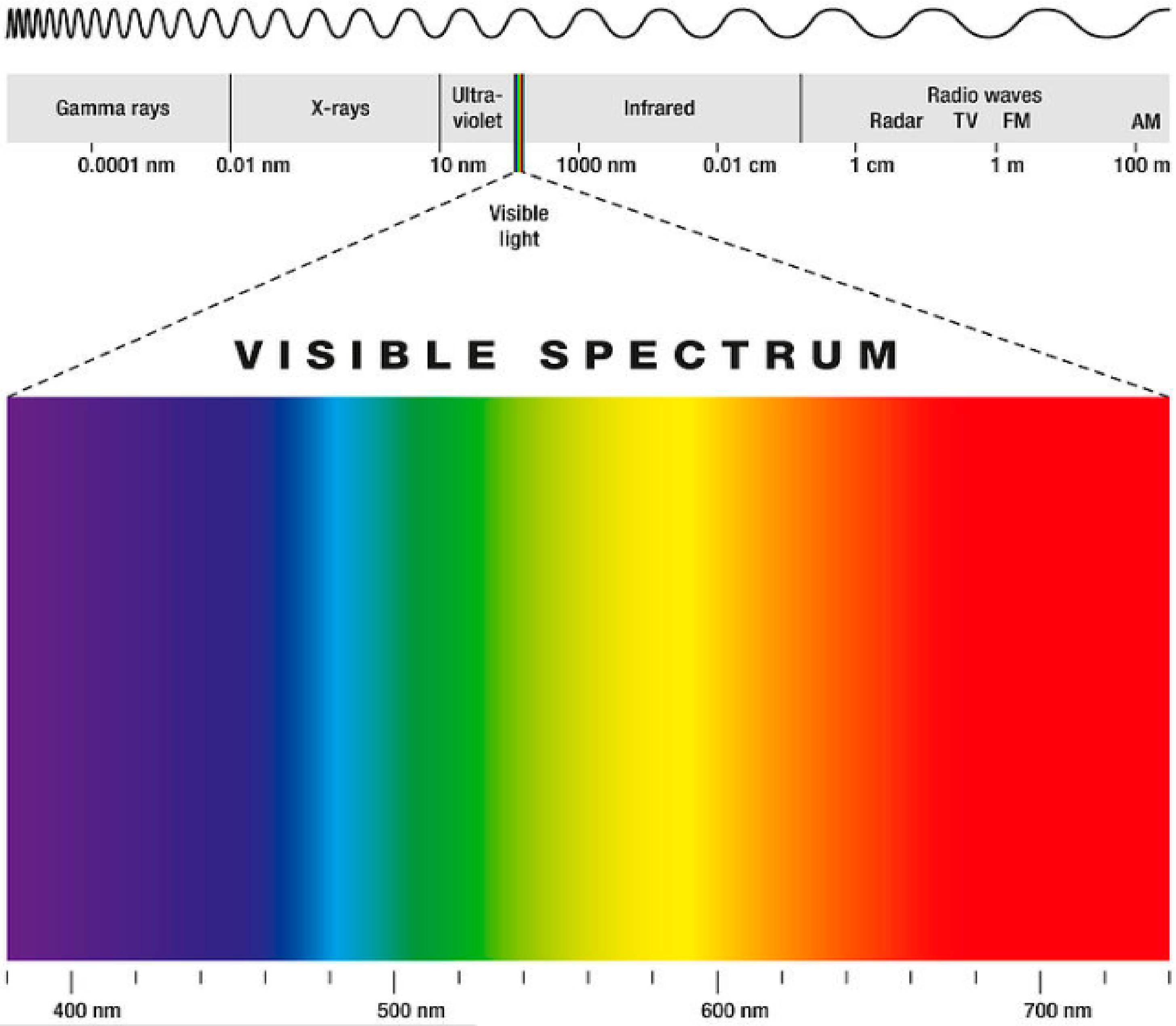
- Objects
- Viewer
- Light source(s)



- Attributes that govern how light interacts with the materials in the scene
- Note the independence of the objects, the viewer, and the light source(s)

Light

- *Light* is the part of the electromagnetic spectrum that causes a reaction in our visual systems
- Generally these are wavelengths in the range of about 350-780 nm (nanometers)
- Long wavelengths appear as reds and short wavelengths as blues



Gamma rays
X-rays
Ultra-violet
Infrared
Radio waves
Radar
TV
FM
AM
0.0001 nm
0.01 nm
10 nm
1000 nm
0.01 cm
1 cm
1 m
100 m
Visible light
VISIBLE SPECTRUM
400 nm
500 nm
600 nm
700 nm

Light

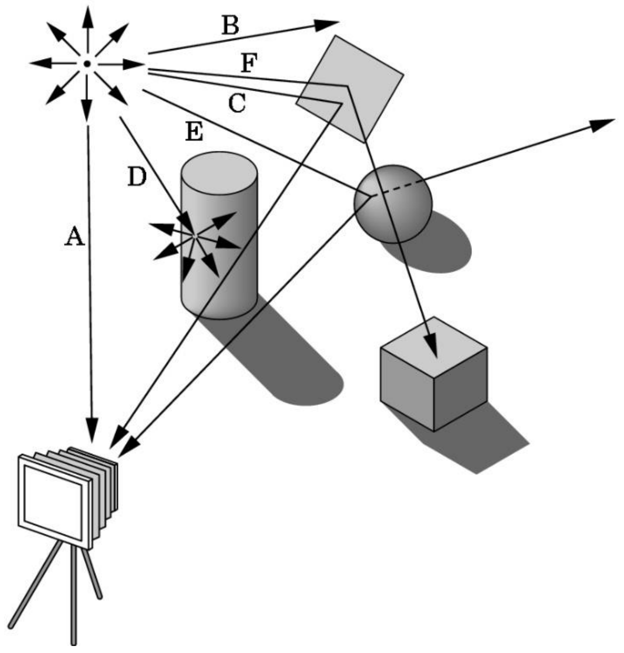
- Modeled geometrically, light travels in straight lines, from the sources to those objects with which it interacts.
- An ideal point source emits energy from a single location at one or more frequencies equally in all directions.
- More complex sources, such as a lightbulb, can be characterized as emitting light over an area and by emitting more light in one direction than another.

Light

- A ray is a semi-infinite line that emanates from a point and travels to infinity in a particular direction.
- Because light travels in straight lines, we can think in terms of rays of light emanating in all directions from one point source.
- A portion of these infinite rays contributes to the image on the film plane of our camera.

Ray Tracing and Geometric Optics

- One way to form an image is to follow rays of light from a point source finding which rays enter the lens of the camera. However, each ray of light may have multiple interactions with objects before being absorbed or going to infinity.

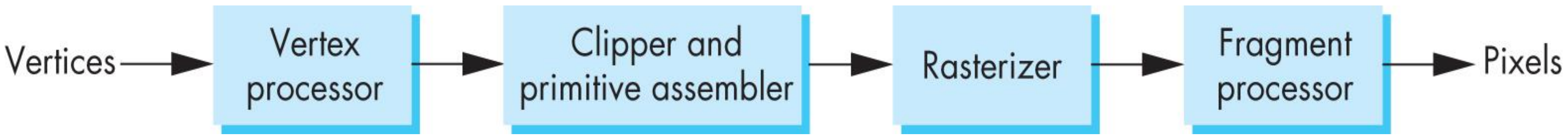


3D Geometric Models

- Describe 3D objects using mathematical primitives such as spheres, cubes, cones, and polygons.
- The most ubiquitous type of model is composed of 3D triangles with shared vertices, which is often called a triangle mesh.

Graphics Pipeline

- Four major steps in the imaging process:
  1. Vertex processing
  2. Clipping and primitive assembly
  3. Rasterization
  4. Fragment processing



Graphics Pipeline

- Vertex processing: Each vertex is processed independently. The major function is coordinate transformation. We represent each change of coordinate systems by a matrix and successive changes by multiplying the individual matrices into a single matrix.
- Clipping and primitive assembly: Assemble set of vertices into primitives and then define the field of view on a primitive-by-primitive basis rather than on a vertex-by-vertex basis.

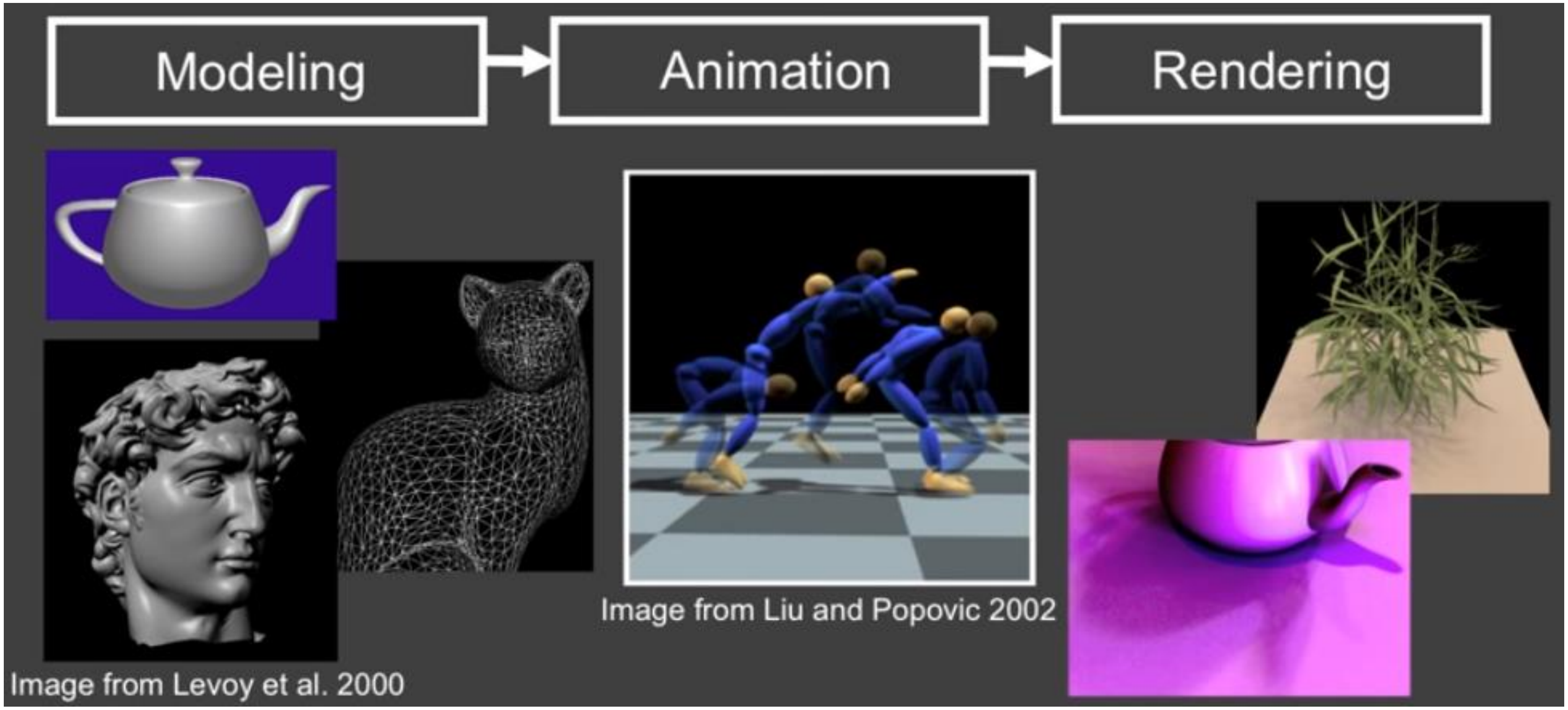
Graphics Pipeline

- Rasterization: Convert primitives’ vertices to pixels in framebuffer. The output of the rasterizer is a set of fragments for each primitives. A fragment can be thought of as a potential pixel that carries its information, including its color and location, that is used to update the corresponding pixel in the framebuffer. Fragments can also carry along depth information that allows later stages to determine whether a particular fragment lies behind other previously rasterized fragments for a given pixel.

Graphics Pipeline

- Fragment processing: Some fragments may not be visible because the surfaces that they define are behind other surfaces. The color of a fragment may be altered by texture mapping or bump mapping. The color of the pixel that corresponds to a fragment can also be read from the framebuffer and blended with the fragment's color to create translucent effects.

3D Graphics Pipeline



Rendering

- Rasterization and raytracing are fundamentally two different ways in which you can create images.
  - Rasterization essentially goes through all the geometric primitives and determines where in the screen they should go.
  - Raytracing does the opposite thing which goes to each point or each pixel in the screen and determines which geometric primitive that corresponds to.
  - They each have their advantages and disadvantages, namely raytracing can produce higher quality images, but has historically been slower.

Structure of WebGL Applications

- Web pages using WebGL are created by using three languages: HTML5 (as a Hypertext Markup Language), JavaScript, and OpenGL shading language (GLSL) - a special programming language similar to C for programming sophisticated visual effects.
- Because GLSL is generally written within JavaScript, only HTML and JavaScript files are actually necessary for WebGL applications.
- So, although WebGL does add complexity to the JavaScript, it retains the same structure as standard dynamic web pages, only using HTML and JavaScript files.

Your First Step with WebGL

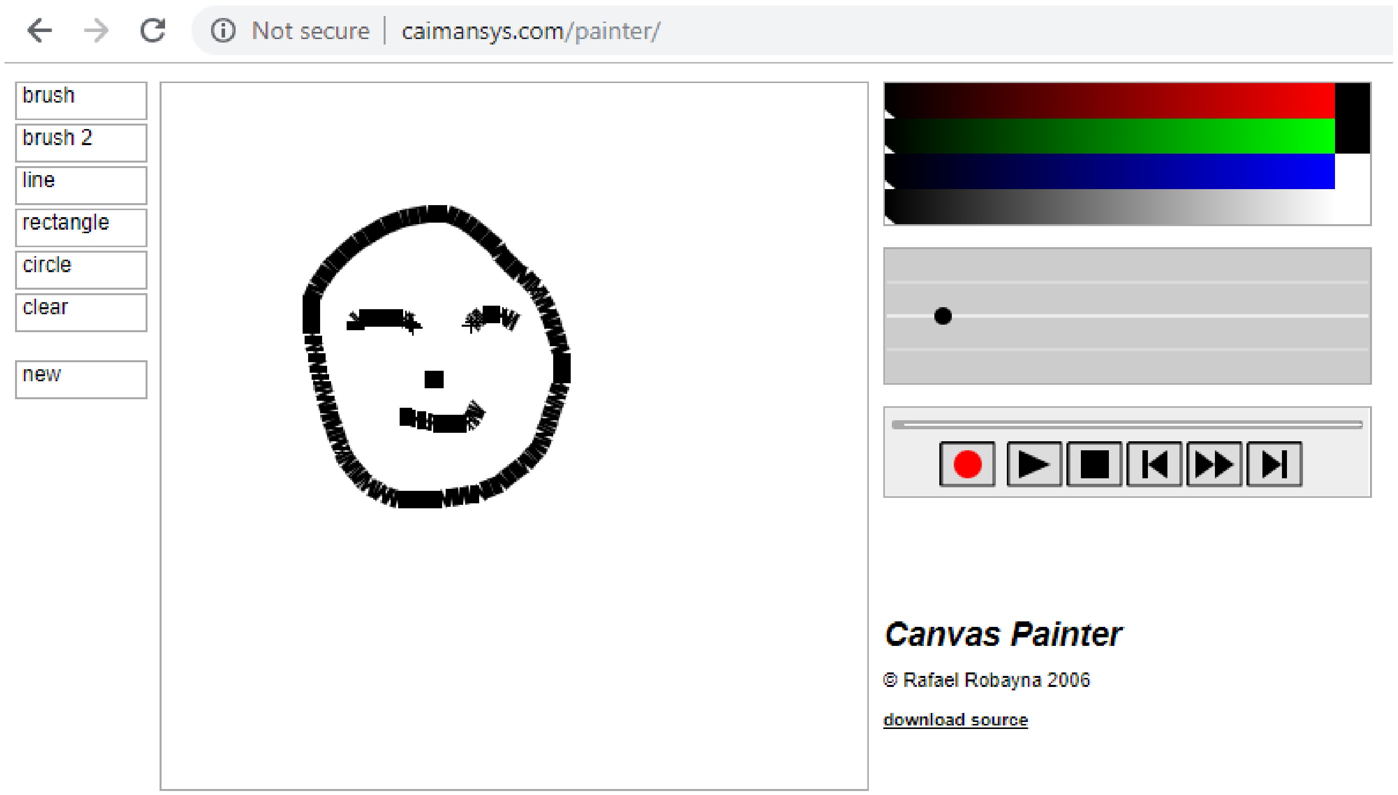
- WebGL applications use both HTML and JavaScript to create and draw 3D graphics on the screen.
- To do this, WebGL utilizes the <canvas> element, introduced in HTML5, which defines a drawing area on a web page.
- Without WebGL, the <canvas> element only allows you to draw two-dimensional graphics using JavaScript. With WebGL, you can use the same element for drawing three-dimensional graphics.

What is a Canvas?

- In a similar manner to the way artists use paint canvases, the <canvas> tag defines a drawing area on a web page.
- Then, rather than using brush and paints, you can use JavaScript to draw anything you want in the area. You can draw points, lines, rectangles, circles, and so on by using JavaScript methods provided for <canvas> .

What is a Canvas?

- A drawing tool using the <canvas> element (http://caimansys.com/painter/)

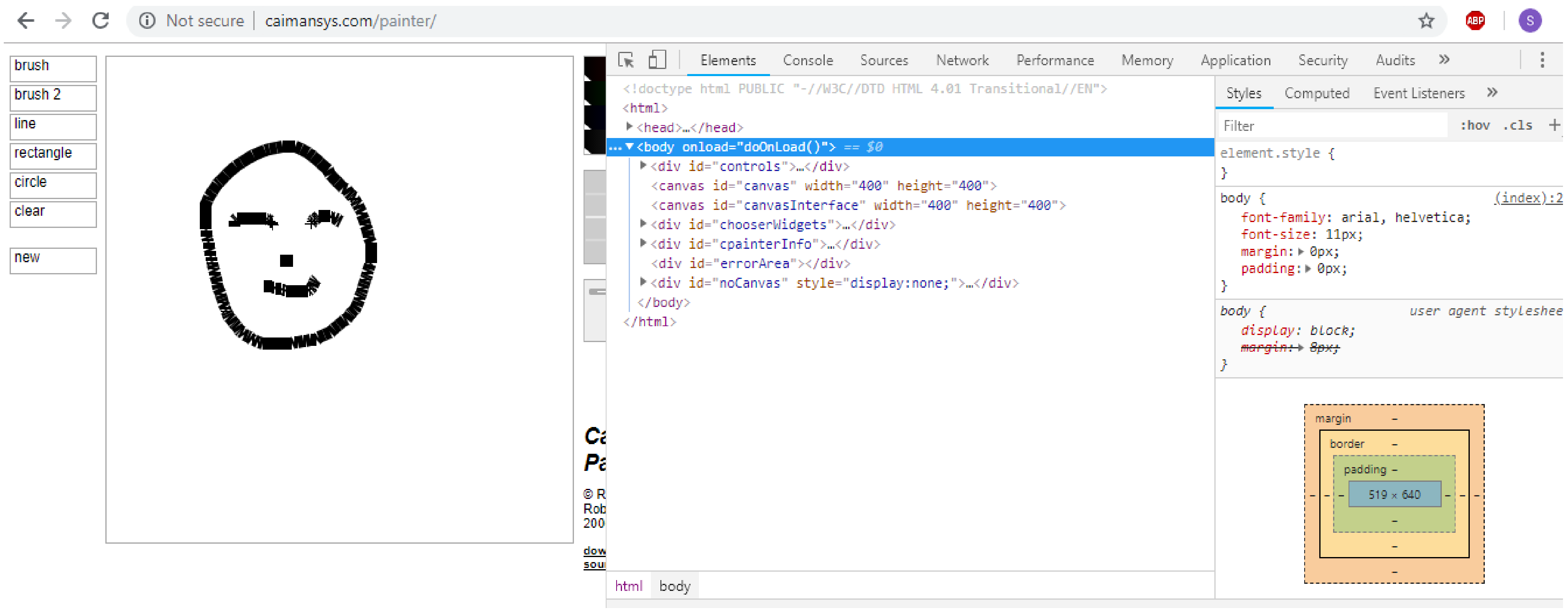


Canvas Painter

© Rafael Robayna 2006

What is a Canvas?

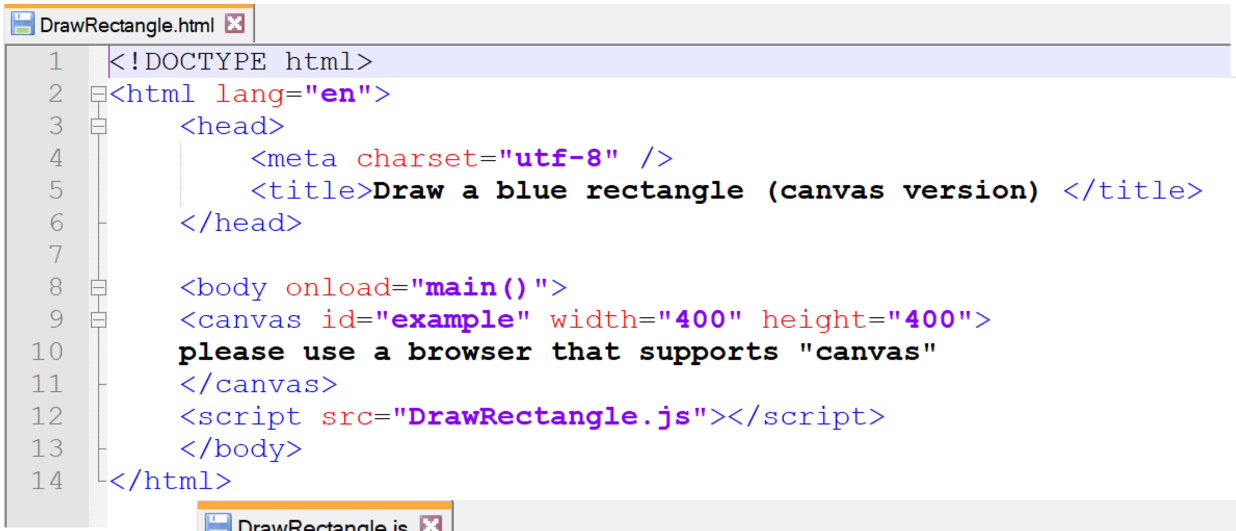
- Ctrl+Shift+J



```
<!doctype html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
 ▶<head>…</head>
…▼<body onload="doOnLoad()"> == $0
   ▶<div id="controls">…</div>
    <canvas id="canvas" width="400" height="400">
    <canvas id="canvasInterface" width="400" height="400">
   ▶<div id="chooserWidgets">…</div>
   ▶<div id="cpainterInfo">…</div>
    <div id="errorArea"></div>
   ▶<div id="noCanvas" style="display:none;">…</div>
  </body>
</html>
```

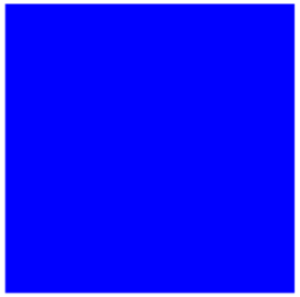
```
element.style {
}
body {                                   (index):2
   font-family: arial, helvetica;
   font-size: 11px;
   margin: ▶ 0px;
   padding: ▶ 0px;
}
body {                       user agent styleshee
   display: block;
   ~~margin: ▶ 8px;~~
}
```

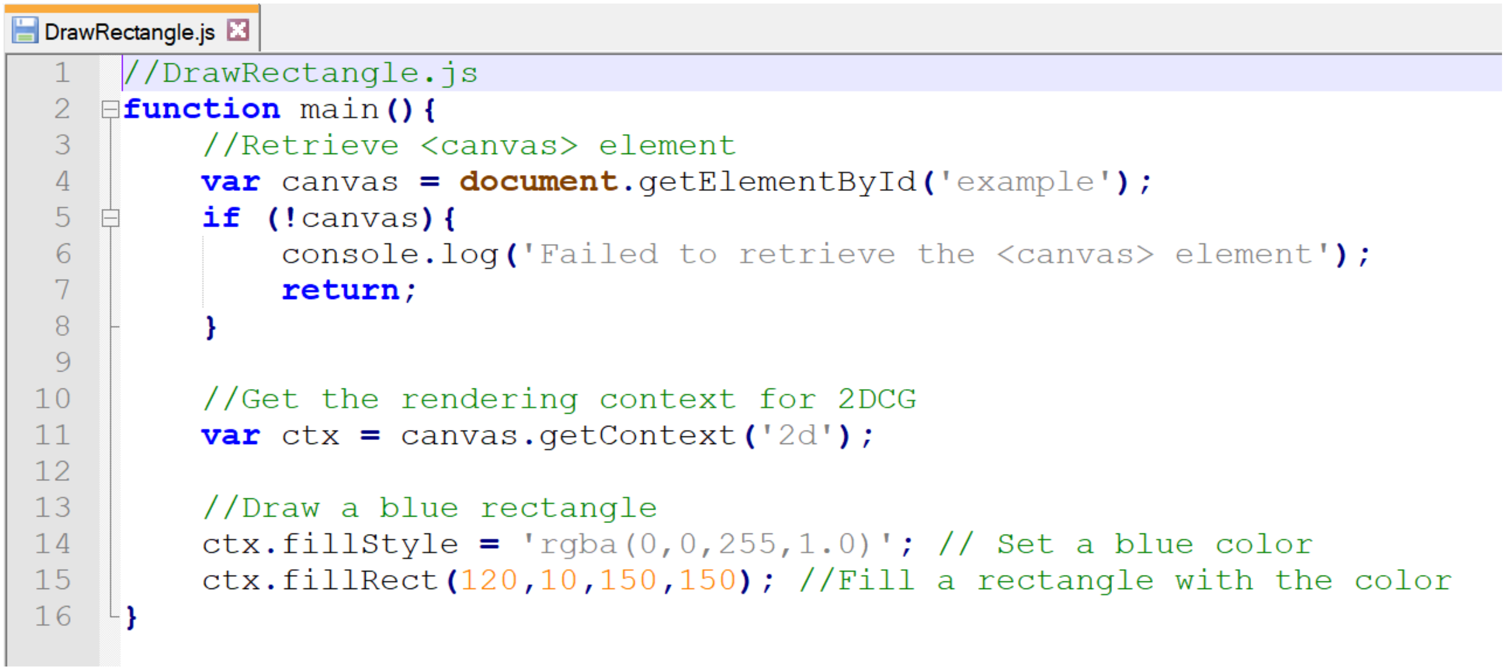
Draw a Rectangle



DrawRectangle.html

```html
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="utf-8" />
        <title>Draw a blue rectangle (canvas version) </title>
    </head>

    <body onload="main()">
    <canvas id="example" width="400" height="400">
    please use a browser that supports "canvas"
    </canvas>
    <script src="DrawRectangle.js"></script>
    </body>
</html>
```





DrawRectangle.js

```javascript
//DrawRectangle.js
function main(){
    //Retrieve <canvas> element
    var canvas = document.getElementById('example');
    if (!canvas){
        console.log('Failed to retrieve the <canvas> element');
        return;
    }

    //Get the rendering context for 2DCG
    var ctx = canvas.getContext('2d');

    //Draw a blue rectangle
    ctx.fillStyle = 'rgba(0,0,255,1.0)'; // Set a blue color
    ctx.fillRect(120,10,150,150); //Fill a rectangle with the color
}
```

Draw a Rectangle

- **DrawRectangle.html**

```
<!DOCTYPE html>
<html lang="en">
        <head>
                <meta charset="utf-8" />
                <title>Draw a blue rectangle (canvas version) </title>
        </head>

        <body onload="main()">
        <canvas id="example" width="400" height="400">
        please use a browser that supports "canvas"
        </canvas>
        <script src="DrawRectangle.js"></script>
        </body>
</html>
```

Draw a Rectangle

- **DrawRectangle.js**

```
function main(){
        //Retrieve <canvas> element
        var canvas = document.getElementById('example');
        if (!canvas){
                console.log('Failed to retrieve the <canvas> element');
                return;
        }

        //Get the rendering context for 2DCG
        var ctx = canvas.getContext('2d');

        //Draw a blue rectangle
        ctx.fillStyle = 'rgba(0,0,255,1.0)'; // Set a blue color
        ctx.fillRect(120,10,150,150); //Fill a rectangle with the color
}
```

Draw a Point - 1

- **HelloPoint1.html**

```
<!DOCTYPE html>
<html lang="en">
		<head>
				<meta charset="utf-8" />
				<title>Draw a point (1) </title>
		</head>

		<body onload="main()">
		<canvas id="webgl" width="400" height="400">
		please use a browser that supports "canvas"
		</canvas>

		<script src="../libs/webgl-utils.js"></script>
		<script src="../libs/webgl-debug.js"></script>
		<script src="../libs/cuon-utils.js"></script>
		<script src="../libs/initShaders.js"></script>
		<script src="HelloPoint1.js"></script>
		</body>
</html>
```

Draw a Point - 1

- **HelloPoint1.js**

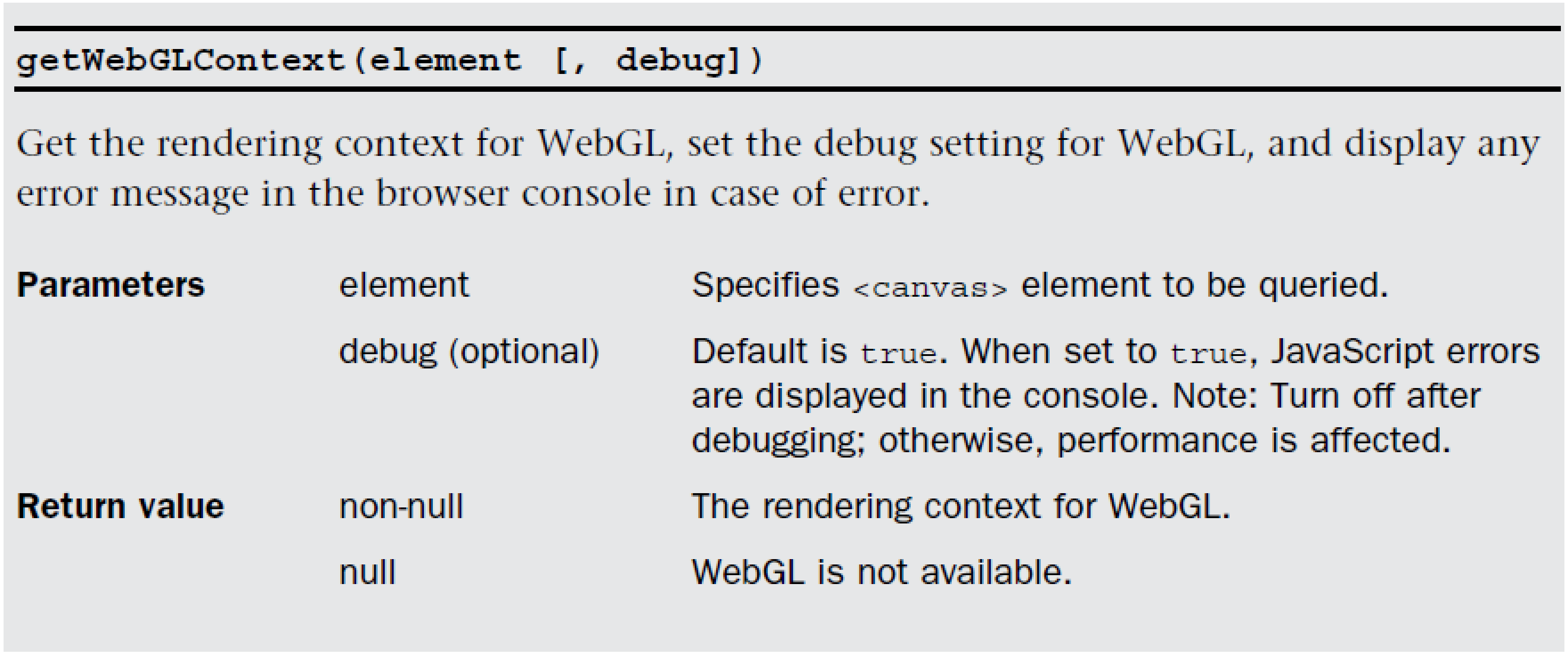
```
// HelloPoint1.js
// Vertex shader program
var VSHADER_SOURCE =
'void main(){\n'+
'gl_Position = vec4(0.0, 0.0, 0.0, 1.0); \n' + //Coordinates
'gl_PointSize = 10.0;\n' + // Set the point size
'}\n';

//Fragment shader program
var FSHADER_SOURCE =
'void main(){\n'+
'gl_FragColor = vec4(1.0, 0.0, 0.0, 1.0); \n' + //Set the color
'}\n';
```

Draw a Point - 1

- **HelloPoint1.js**

```
function main(){
        //Retrieve <canvas> element
        var canvas = document.getElementById('webgl');

        // Get the rendering context for WebGL
        var gl = getWebGLContext(canvas);
        // var gl = canvas.getContext('webgl');
        if (!gl){
                console.log('Failed to get the rendering context for WebGL');
                return;
        }

        //Initialize shaders
        if (!initShaders(gl,VSHADER_SOURCE,FSHADER_SOURCE)){
                console.log('Failed to initialize shaders.');
        return;
        }

        //Set the color for clearing <canvas>
        gl.clearColor(0.0,0.0,0.0,1.0);

        //Clear <canvas>
        gl.clear(gl.COLOR_BUFFER_BIT);

        //Draw a point
        gl.drawArrays(gl.POINTS,0,1);
}
```

Draw a Point - 1

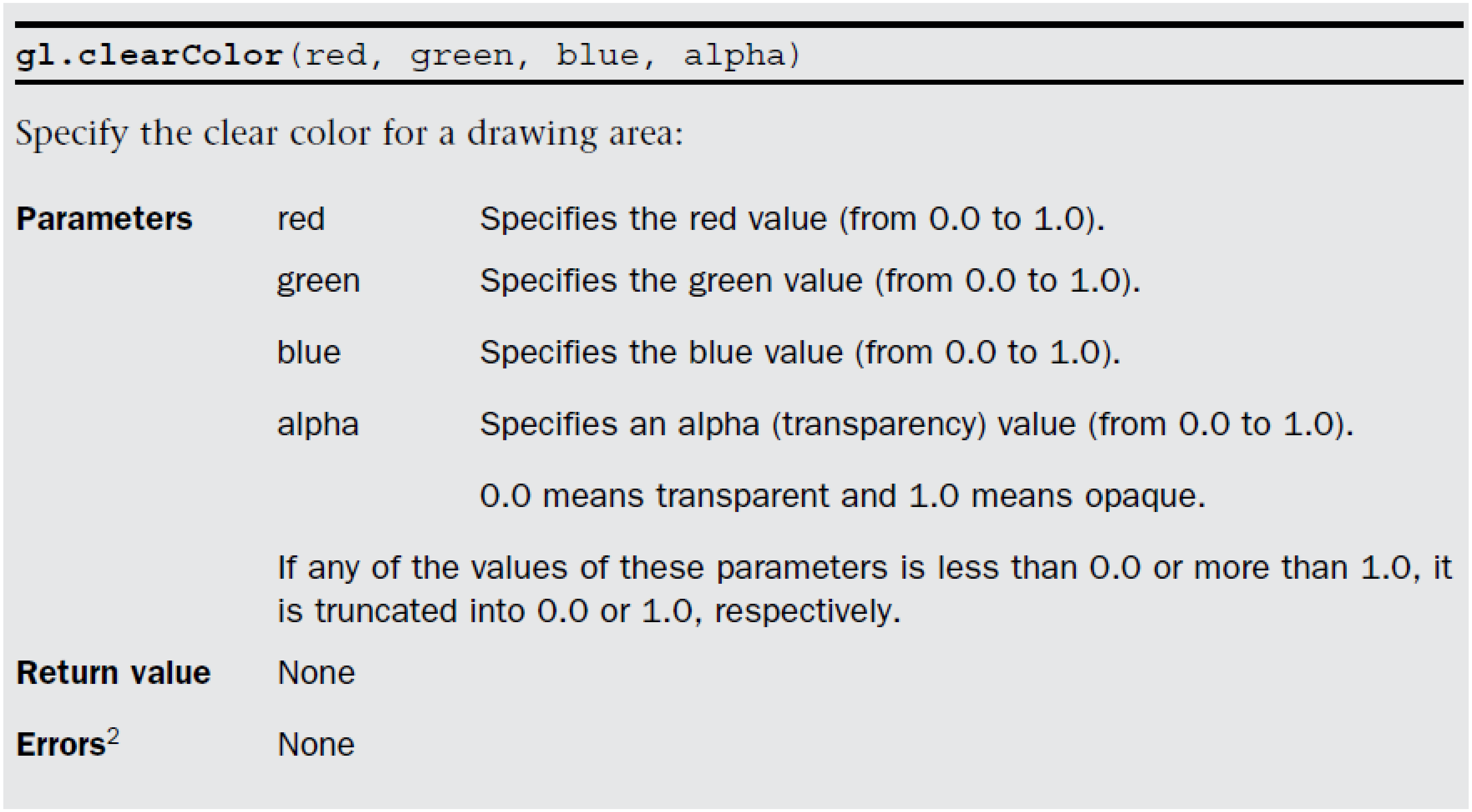


`getWebGLContext(element [, debug])`

Get the rendering context for WebGL, set the debug setting for WebGL, and display any error message in the browser console in case of error.

Parameters	element	Specifies `<canvas>` element to be queried.
	debug (optional)	Default is `true`. When set to `true`, JavaScript errors are displayed in the console. Note: Turn off after debugging; otherwise, performance is affected.
Return value	non-null	The rendering context for WebGL.
	null	WebGL is not available.

Draw a Point - 1

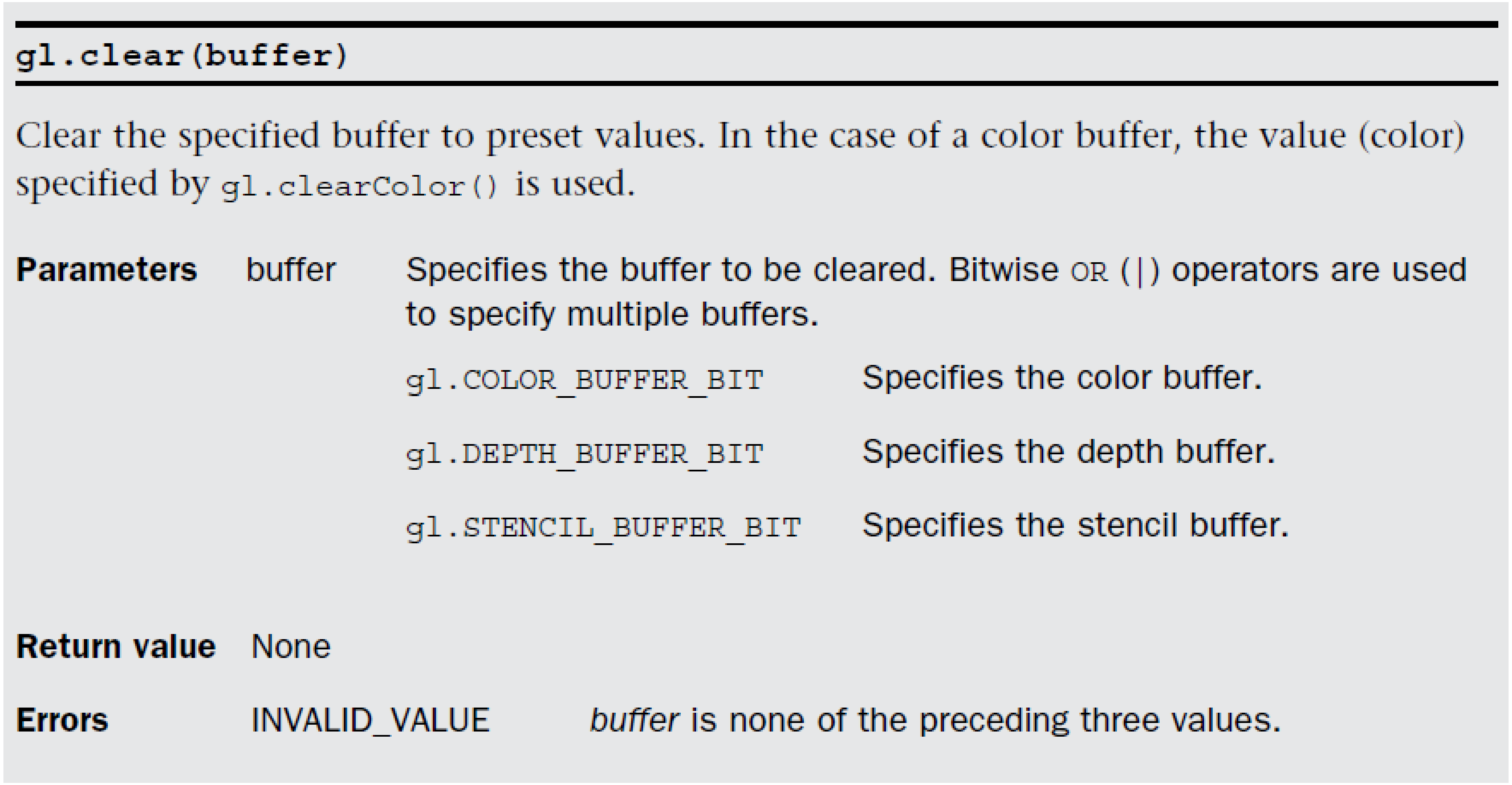


`gl.clearColor(red, green, blue, alpha)`

Specify the clear color for a drawing area:

Parameters	red	Specifies the red value (from 0.0 to 1.0).
	green	Specifies the green value (from 0.0 to 1.0).
	blue	Specifies the blue value (from 0.0 to 1.0).
	alpha	Specifies an alpha (transparency) value (from 0.0 to 1.0). 0.0 means transparent and 1.0 means opaque.
	If any of the values of these parameters is less than 0.0 or more than 1.0, it is truncated into 0.0 or 1.0, respectively.	
Return value	None	
Errors[2]	None	

Draw a Point - 1



`gl.clear(buffer)`

Clear the specified buffer to preset values. In the case of a color buffer, the value (color) specified by `gl.clearColor()` is used.

Parameters buffer Specifies the buffer to be cleared. Bitwise OR (|) operators are used to specify multiple buffers.

`gl.COLOR_BUFFER_BIT`	Specifies the color buffer.
`gl.DEPTH_BUFFER_BIT`	Specifies the depth buffer.
`gl.STENCIL_BUFFER_BIT`	Specifies the stencil buffer.

Return value None

Errors INVALID_VALUE *buffer* is none of the preceding three values.

What is a Shader?

- Shader programs are necessary when you want to draw something on the screen in WebGL.

What is a Shader?

- WebGL needs the following two types of shaders:
  - Vertex shader: program that describes the traits (position, colors,…) of a vertex. The vertex is a point in 2D/3D space, such as corner or intersection of a 2D/3D shape.
  - Fragment shader: program that deals with per-fragment processing such as lighting. The fragment is a WebGL term that you can consider as a kind of pixel (picture element).

What is a Shader?

- In a 3D scene, it is not enough just to draw graphics. We also account for how they are viewed as light sources hit them or the viewer’s perspective changes.
- Shading does this with a high degree of flexibility and is part of the reason that today’s 3D graphics are so realistic.

What is a Shader?

// Vertex shader program (Language: GLSL ES)

```
var VSHADER_SOURCE =
'void main(){\n'+
'gl_Position = vec4(0.0, 0.0, 0.0, 1.0); \n' + //Coordinates
'gl_PointSize = 100.0;\n' + // Set the point size
'}\n
```

//Fragment shader program (Language: GLSL ES)

```
var FSHADER_SOURCE =
'void main(){\n'+
'gl_FragColor = vec4(1.0, 0.0, 0.0, 1.0); \n' + //Set the color
'}\n';
```

These programs are actually the following shader language programs but written as a JavaScript string to make it possible to pass the shaders to the WebGL system.

What is a Shader?

```
//Vertex shader program
void main(){
gl_Position = vec4(0.0, 0.0, 0.0, 1.0);
gl_PointSize = 10.0;
}

//Fragment shader program
void main(){
gl_FragColor = vec4(1.0, 0.0, 0.0, 1.0);
}
```

What is a Shader?

Table 2.2 Built-In Variables Available in a Vertex Shader

Type and Varlable Name	**Descrlptlon**
vec4 `gl_Position`	Specifies the position of a vertex
float `gl_PointSize`	Specifies the size of a point (in pixels)

GLSL is a “typed” programming language.

Table 2.3 Data Types in GLSL ES

Type	**Descrlptlon**
float	Indicates a floating point number
vec4	Indicates a vector of four floating point numbers

`float`	`float`	`Float`	`float`

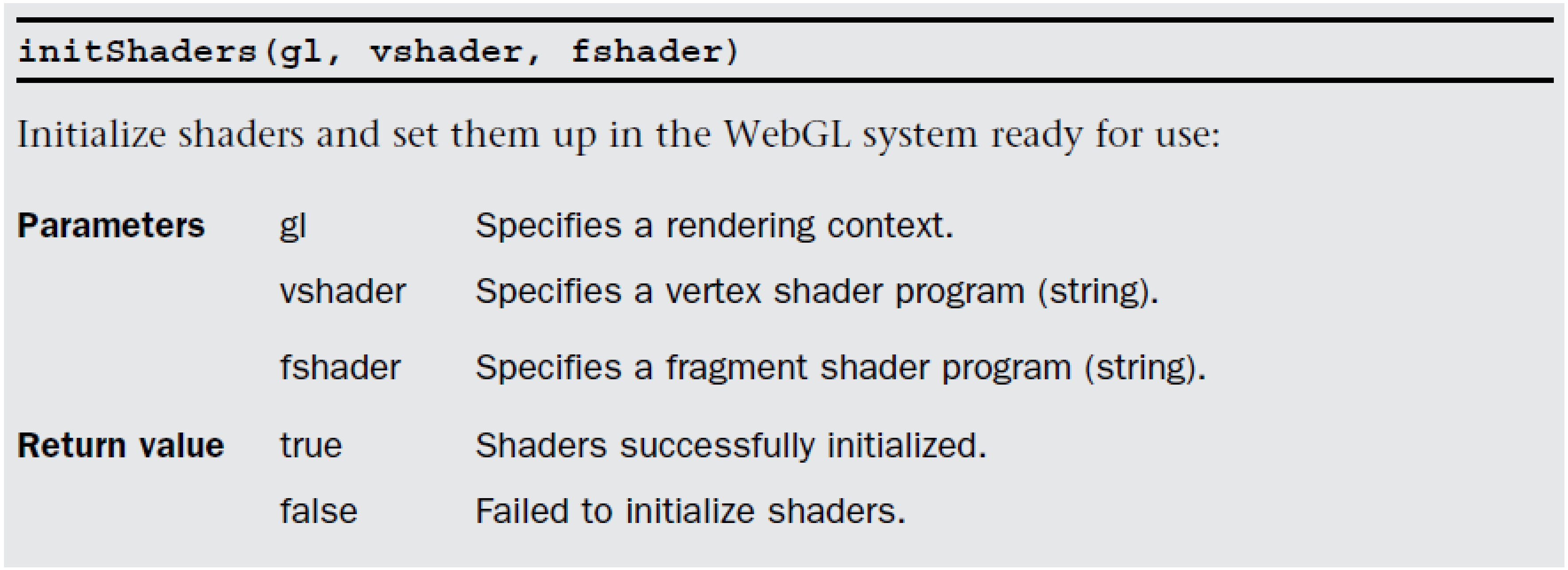
Table 2.4 The Built-In Value Available in a Fragment Shader

Type and Varlable Name	**Descrlptlon**
vec4 `gl_FragColor`	Specify the color of a fragment (in RGBA)

What is a Shader?

```
//Initialize shaders
         if
(!initShaders(gl,VSHADER_SOURCE,FSHADER_SOURCE))
{
                  console.log('Failed to initialize shaders.');
         return;
         }
```

What is a Shader?

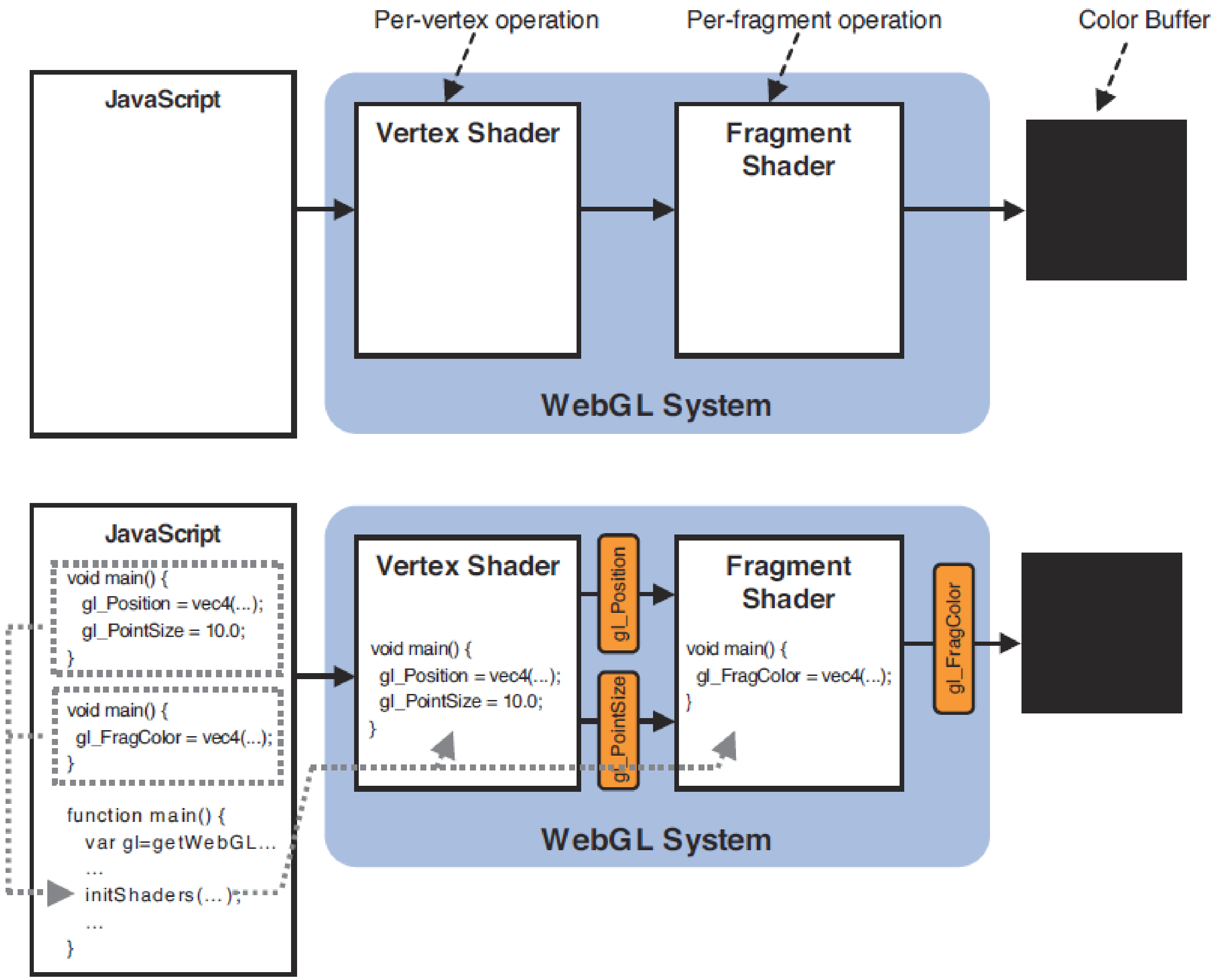


```
initShaders(gl, vshader, fshader)
```

Initialize shaders and set them up in the WebGL system ready for use:

Parameters	gl	Specifies a rendering context.
	vshader	Specifies a vertex shader program (string).
	fshader	Specifies a fragment shader program (string).
Return value	true	Shaders successfully initialized.
	false	Failed to initialize shaders.

What is a Shader?



```
void main() {
  gl_Position = vec4(...);
  gl_PointSize = 10.0;
}
```

```
void main() {
  gl_FragColor = vec4(...);
}
```

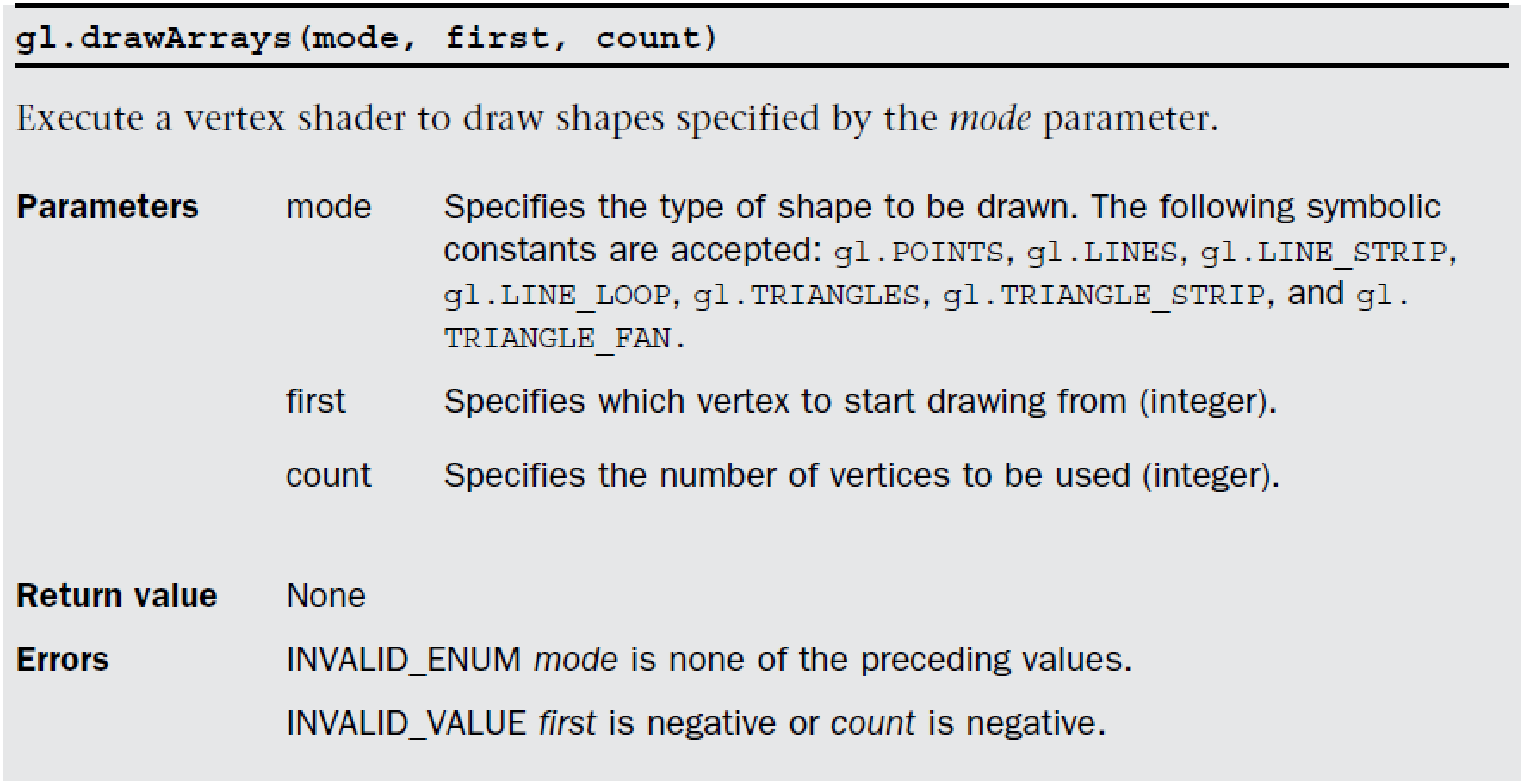
```
function main() {
  var gl=getWebGL...
  ...
  initShaders(...);
  ...
}
```

```
void main() {
 gl_Position = vec4(...);
 gl_PointSize = 10.0;
}
```

```
void main() {
 gl_FragColor = vec4(...);
}
```

Figure 2.14 Behavior of initShaders()

The Draw Operation



```
gl.drawArrays(mode, first, count)
```

Execute a vertex shader to draw shapes specified by the *mode* parameter.

Parameters	mode	Specifies the type of shape to be drawn. The following symbolic constants are accepted: `gl.POINTS`, `gl.LINES`, `gl.LINE_STRIP`, `gl.LINE_LOOP`, `gl.TRIANGLES`, `gl.TRIANGLE_STRIP`, and `gl.TRIANGLE_FAN`.
	first	Specifies which vertex to start drawing from (integer).
	count	Specifies the number of vertices to be used (integer).
Return value	None	
Errors	INVALID_ENUM *mode* is none of the preceding values.	
	INVALID_VALUE *first* is negative or *count* is negative.	

The Draw Operation

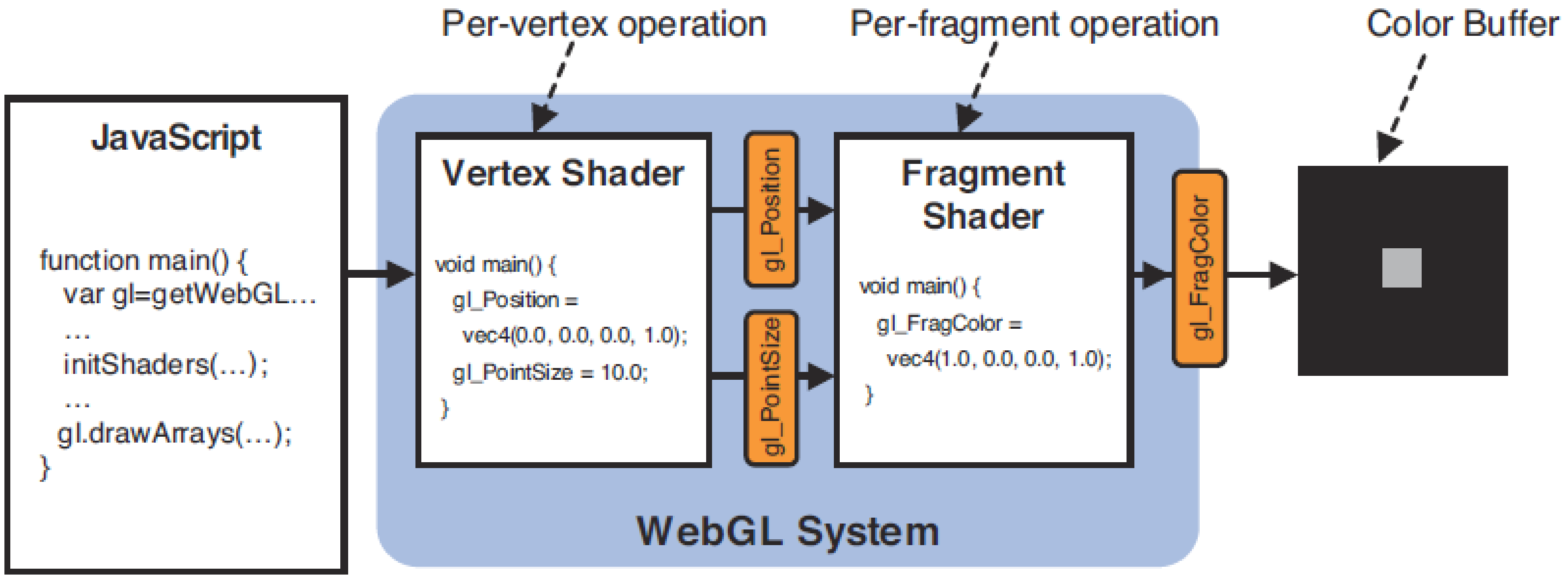
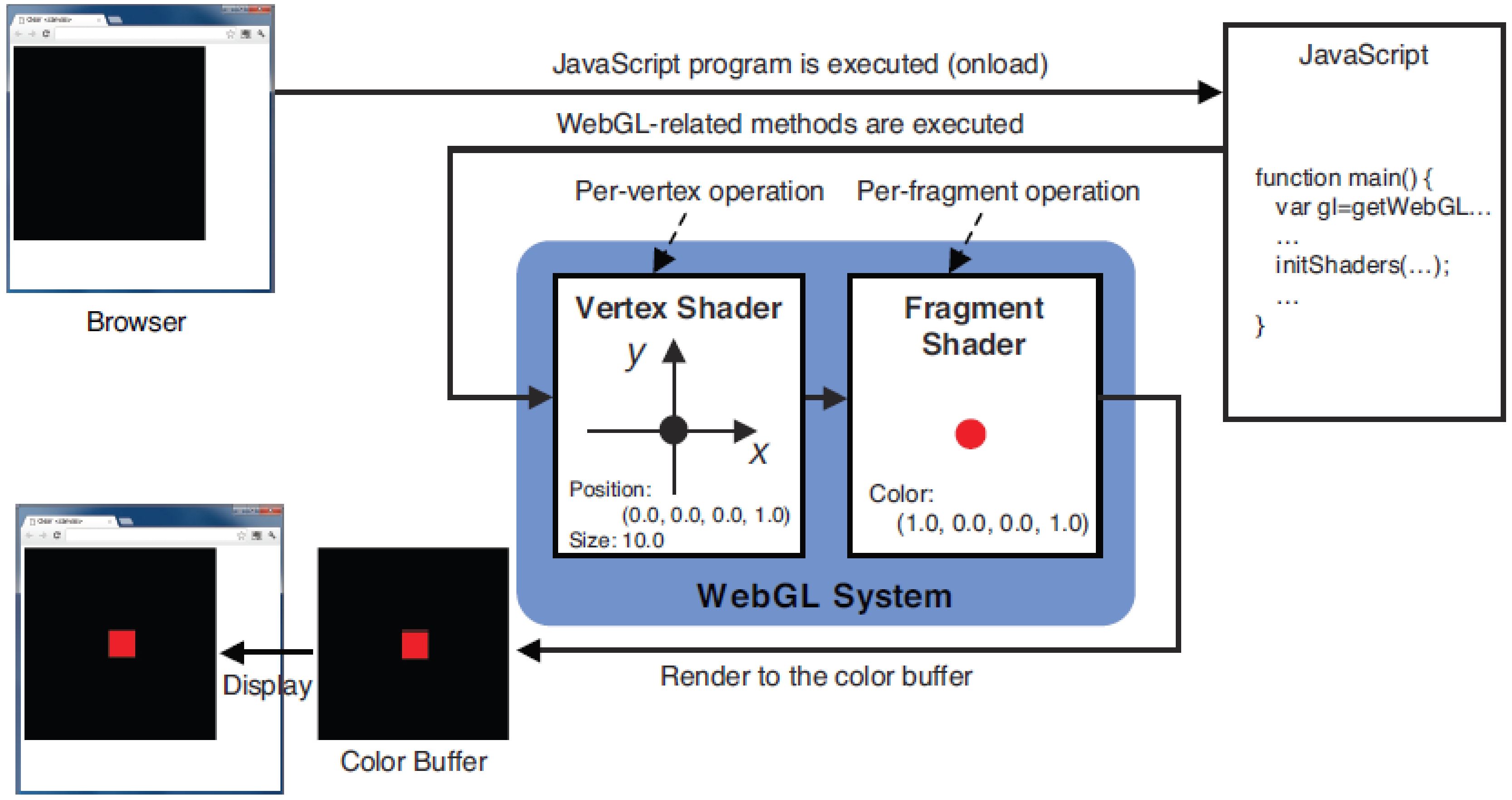


Figure 2.15 The behavior of shaders

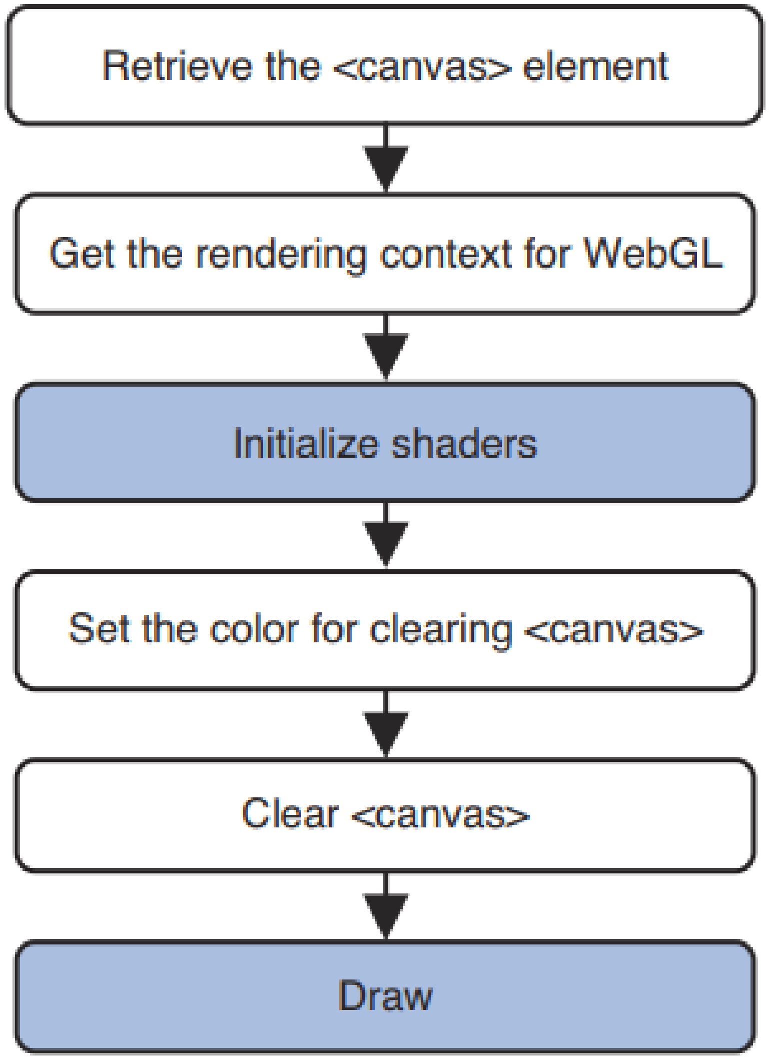
The Processing Flow of a WebGL Program



```
function main() {
  var gl=getWebGL...
  ...
  initShaders(...);
  ...
}
```

Figure 2.10 The processing flow from executing a JavaScript program to displaying the result in a browser

The Processing Flow of a WebGL Program


Retrieve the <canvas> element
Get the rendering context for WebGL
Initialize shaders
Set the color for clearing <canvas>
Clear <canvas>
Draw