COSC 414/519I: Computer Graphics

2023W2

Shan Du

Draw a Point - 2

- How to pass data between JavaScript and the shaders?
  - “HelloPoint1” directly writes (“hard-coded”) the point’s position in the vertex shader.
  - It always draws a point at the same position.
  - WebGL program can pass a vertex position from JavaScript to the vertex shader and then draw a point at that position.

Draw a Point - 2

- Using Attribute Variables
  - There are two ways to pass data to a vertex shader: attribute variable and uniform variable.
  - The attribute variable passes data that differs for each vertex, whereas the uniform variable passes data that is the same (or uniform) in each vertex.

Draw a Point - 2

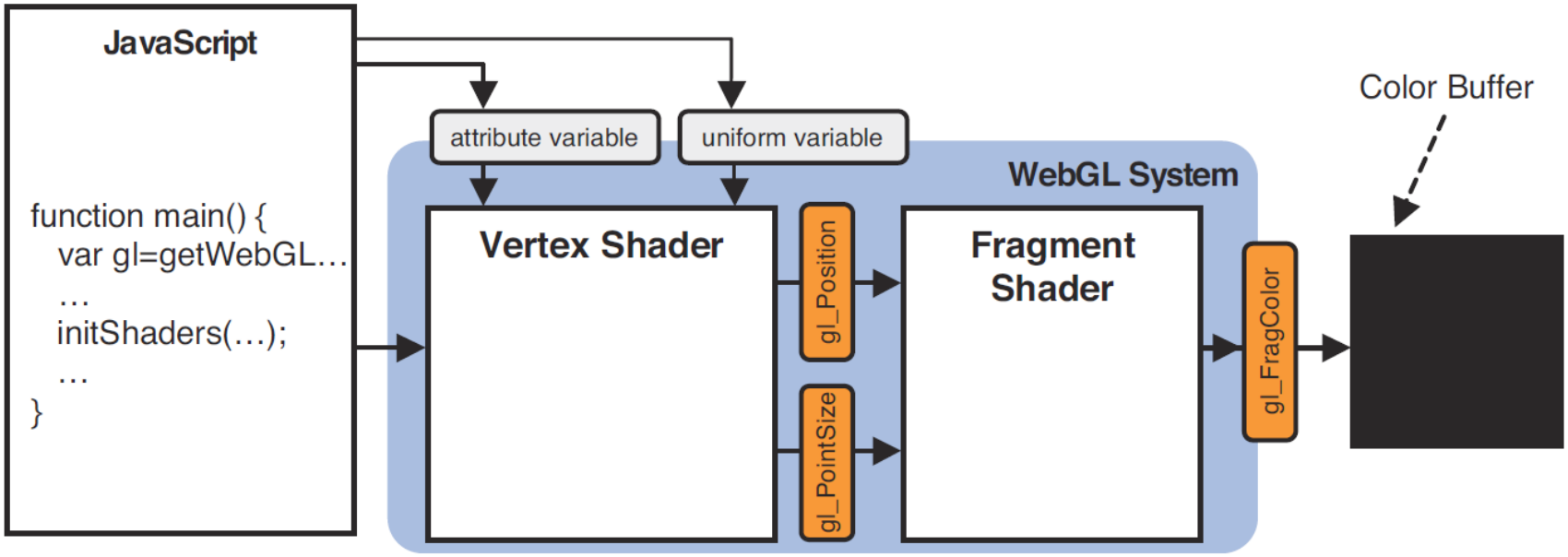


Figure 2.20 Two ways to pass data to a vertex shader

Draw a Point - 2

- The attribute variable is a GLSL variable which is used to pass data from outside a vertex shader into the shader and is only available to vertex shaders.
- To use the attribute variable, the sample program involves the following three steps:

  1. Prepare the attribute variable for the vertex position in the vertex shader.

  2. Assign the attribute variable to the *gl_Position* variable.

  3. Pass the data to the attribute variable.

Draw a Point - 2

```
// Vertex shader program
var VSHADER_SOURCE =
'attribute vec4 a_Position;\n' +
'void main() {\n' +
' gl_Position = a_Position;\n' +
' gl_PointSize = 10.0;\n' +
'}\n';
```

- `<Storage Qualifier> <Type> <Variable Name>`

Draw a Point - 2

```
// Get the storage location of attribute variable
var a_Position = gl.getAttribLocation(gl.program, 'a_Position');
if (a_Position < 0) {
console.log('Failed to get the storage location of a_Position');
return;
}
```

Draw a Point - 2

`gl.getAttribLocation(program, name)`

Retrieve the storage location of the attribute variable specified by the *name* parameter.

Parameters	program	Specifies the program object that holds a vertex shader and a fragment shader.
	name	Specifies the name of the attribute variable whose location is to be retrieved.
Return value	greater than or equal to 0	The location of the specified attribute variable.
	-1	The specified attribute variable does not exist or its name starts with the reserved prefix `gl_` or `webgl_`.

Errors	INVALID_OPERATION	*program* has not been successfully linked (See Chapter 9.)
	INVALID_VALUE	The length of *name* is more than the maximum length (256 by default) of an attribute variable name.

Draw a Point - 2

- *gl.program* specifies a **program object** that holds the vertex shader and the fragment shader.
- Note that you should use *gl.program* only after *initShaders()* has been called because *initShaders()* assigns the program object to the variable.
- The second parameter of *gl.getAttribLocation* specifies the attribute variable name (in this case '*a_Position*') whose location you want to know.

Draw a Point - 2

// Pass vertex position to attribute variable

gl.vertexAttrib3f(a_Position, 0.0, 0.0, 0.0);

`gl.vertexAttrib3f(location, v0, v1, v2)`

Assign the data (*v0*, *v1*, and *v2*) to the attribute variable specified by *location*.

Parameters	location	Specifies the storage location of an attribute variable to be modified.
	v0	Specifies the value to be used as the first element for the attribute variable.
	v1	Specifies the value to be used as the second element for the attribute variable.
	v2	Specifies the value to be used as the third element for the attribute variable.
Return value	None	
Errors	INVALID_OPERATION	There is no current program object.
	INVALID_VALUE	*location* is greater than or equal to the maximum number of attribute variables (8, by default).

Draw a Point - 2

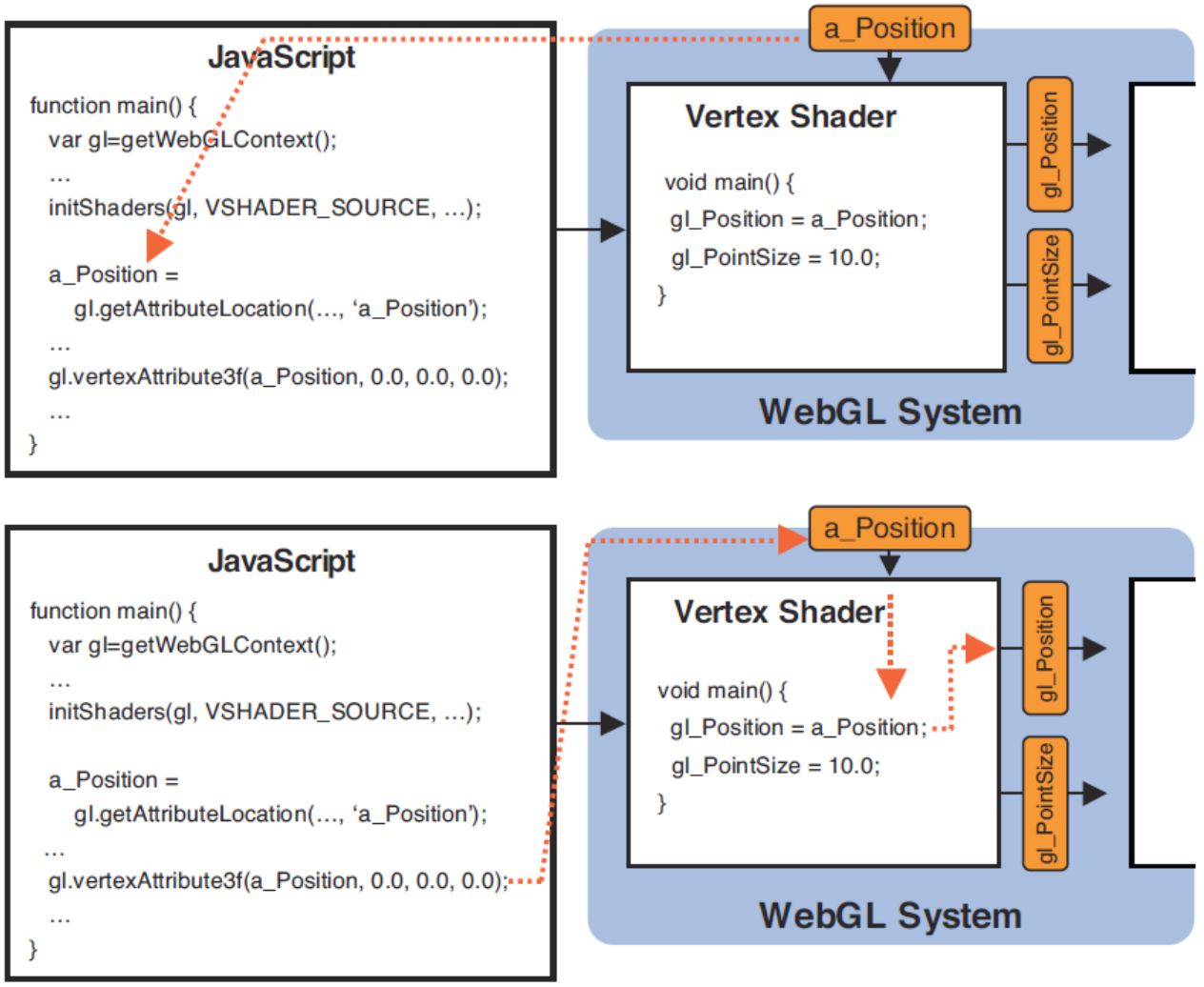


Figure 2.22 Getting the storage location of an attribute variable and then writing a value to the variable

Draw a Point - 2

- The Naming Rules for WebGL-Related Methods:

The function names in WebGL comprise the three components: *<base function name> <number of parameters> < parameter type>*.

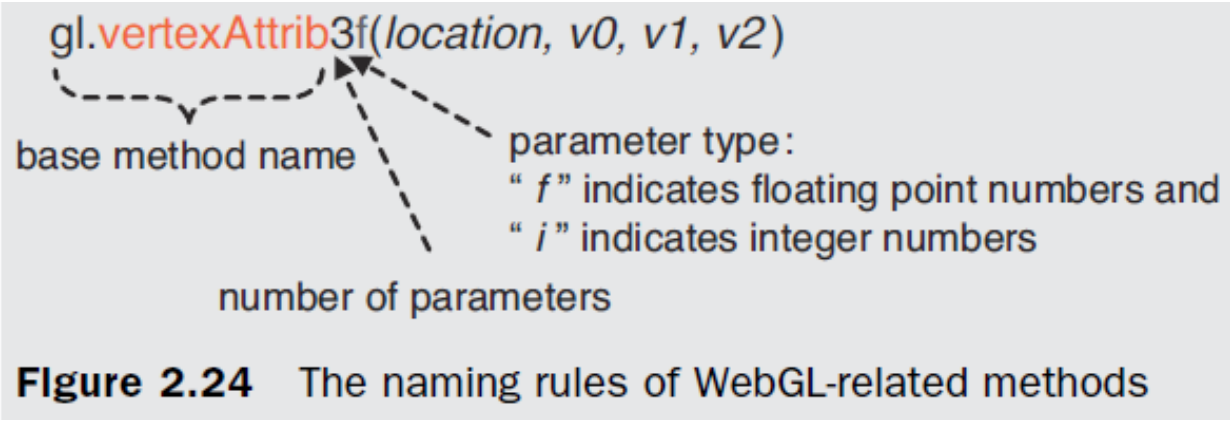


FIgure 2.24 The naming rules of WebGL-related methods

3D APIs

- The synthetic-camera model is the basis for many popular APIs, including OpenGL, WebGL and Direct3D.
- We use functions in the API to specify the following:
  - Objects
  - A viewer
  - Light sources
  - Material properties

3D APIs

- Objects are usually defined by sets of vertices.
- Most APIs provide similar sets of primitive objects for the user.
- We are specifying the geometry and leaving it to the APIs to determine which pixels to color in the framebuffer.

3D APIs

- We can define a viewer or camera by four specifications:
  1. Position: The camera location is usually given by the position of the center of the lens, COP.
  2. Orientation: We can place a camera coordinate system with its origin at COP. We can then rotate the camera independently around the three axes of this system.
  3. Focal Length: The focal length of the lens determines the size of the image on the film plane, or the portion of the world the camera sees.
  4. Film Plane: The orientation of the back of the camera can be adjusted independently of the orientation of the lens.

3D APIs

- Light sources are defined by their location, strength, color, and directionality.
- Material properties are attributes of the objects, and such properties can also be specified through APIs.

WebGL

- Performance is achieved by using GPU rather than CPU.
- Control GPU through programs called shaders. Both the vertex processor and fragment processor are now programmable by the application programmer by using OpenGL Shading Language (GLSL). The programmer can write their own shaders.
- Modern GPUs are unified shading engines that carry out both vertex and fragment shading concurrently.

Graphics Programming

- Use application programming interface (API) to program graphics.
- Regard 2D graphics as a special case of 3D graphics. 2D code will execute without modification on a 3D system.
- JavaScript code for WebGL can be easily converted to C/C++ code for desktop OpenGL.

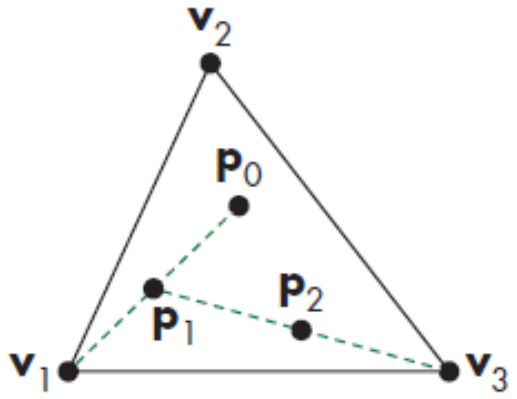
Sierpinski Gasket

- An interesting shape that has a long history and is of interest in areas such as fractal geometry.
- Suppose we have three points in space (not collinear), they are the vertices of a unique triangle and also define a unique plane. We assume this plane is the plane $z = 0$. The coordinates of these points are $(x_1, y_1, 0)$, $(x_2, y_2, 0)$, and $(x_3, y_3, 0)$.

Sierpinski Gasket

The construction is as follows:

1. Pick an initial point $p = (x, y, 0)$ randomly inside the triangle.
2. Select one of the three vertices at random.
3. Find the point *q* halfway between *p* and the randomly selected vertex.
4. Display *q* by putting some sort of marker (a small circle), at the corresponding location on the display.
5. Replace *p* with *q*.
6. Return to step 2.



Sierpinski Gasket

```
function sierpinski()
{
  initialize_the_system();
  p = find_initial_point();

  for (some_number_of_points) {
    q = generate_a_point(p);
    display_the_point(q);
    p = q;
  }

  cleanup();
}
```

Immediate mode graphics

Sierpinski Gasket

```
function sierpinski()
{
  initialize_the_system();
  p = find_initial_point();

  for (some_number_of_points) {
    q = generate_a_point(p);
    store_the_point(q);
    p = q;
  }

  display_all_points();
  cleanup();
}
```

Retained mode graphics

Sierpinski Gasket

```
function sierpinski()
{
  initialize_the_system();
  p = find_initial_point();

  for (some_number_of_points) {
    q = generate_a_point(p);
    store_the_point(q);
    p = q;
  }

  send_all_points_to_GPU();
  display_data_on_GPU();
  cleanup();
}
```

Sierpinski Gasket

```
const numPoints = 5000;

var vertices = [
  vec2(-1.0, -1.0),
  vec2(0.0,  1.0),
  vec2(1.0, -1.0)
];

var u = scale(0.5, add(vertices[0], vertices[1]));
var v = scale(0.5, add(vertices[0], vertices[2]));
var p = scale(0.5, add(u, v));

points = [ p ];

for (var i = 1; i < numPoints; ++i) {
  var j = Math.floor(Math.random() * 3);

  p = scale(0.5, add(points[i-1], vertices[j]));
  points.push(p);
}
```

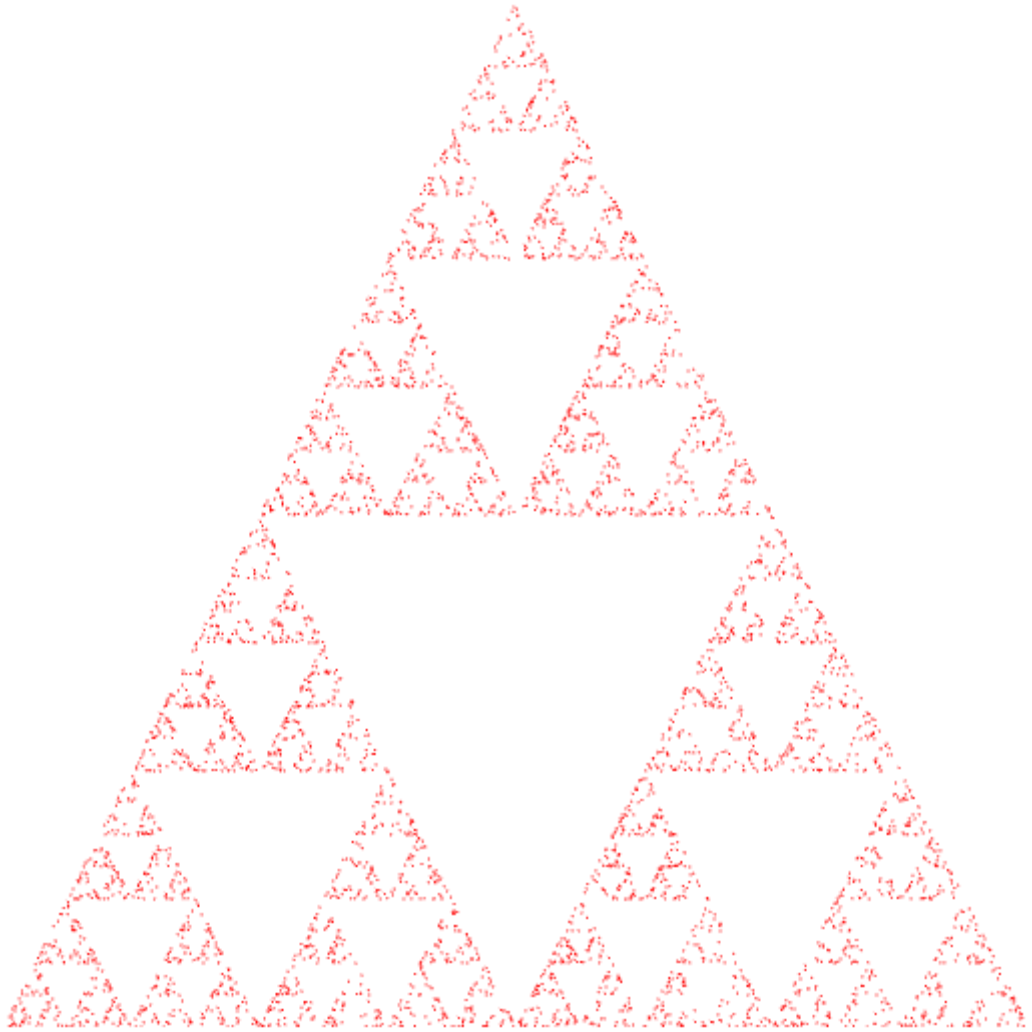
Sierpinski Gasket

```
const numPoints = 5000;

var vertices = [
  vec2(-1.0, -1.0),
  vec2(0.0,  1.0),
  vec2(1.0, -1.0)
];
```

```
var vertices = [
  vec3(-1.0, -1.0, 0.0),
  vec3(0.0, 1.0, 0.0),
  vec3(1.0, -1.0, 0.0)
];
```

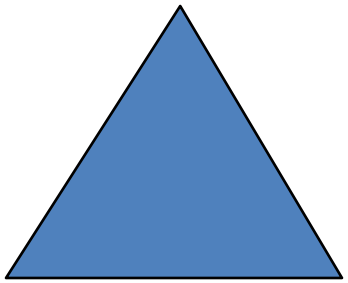
```
var u = scale(0.5, add(vertices[0], vertices[1]));
var v = scale(0.5, add(vertices[0], vertices[2]));
var p = scale(0.5, add(u, v));

points = [ p ];

for (var i = 1; i < numPoints; ++i) {
  var j = Math.floor(Math.random() * 3);

  p = scale(0.5, add(points[i-1], vertices[j]));
  points.push(p);
}
```

Sierpinski Gasket (2D)

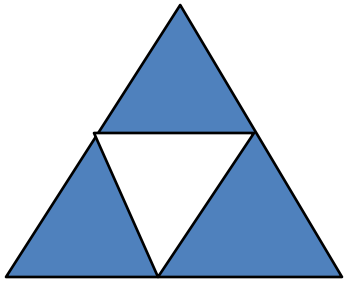


Sierpinski Gasket (2D)

- Start with a triangle.



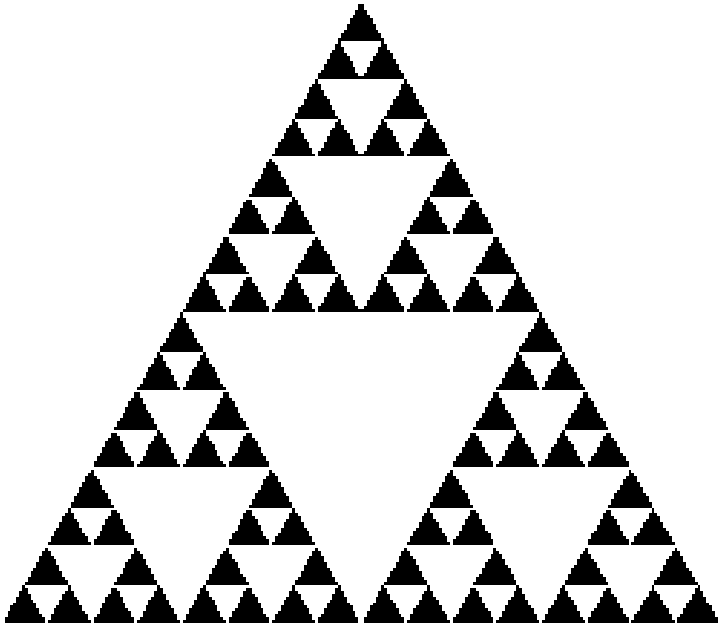
- Connect bisectors of sides and remove central triangle.



- Repeat.

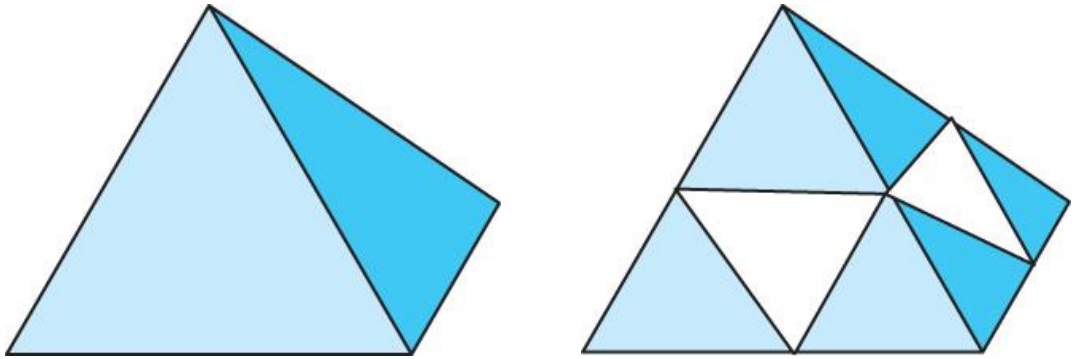
Sierpinski Gasket (2D)

- Five subdivisions.



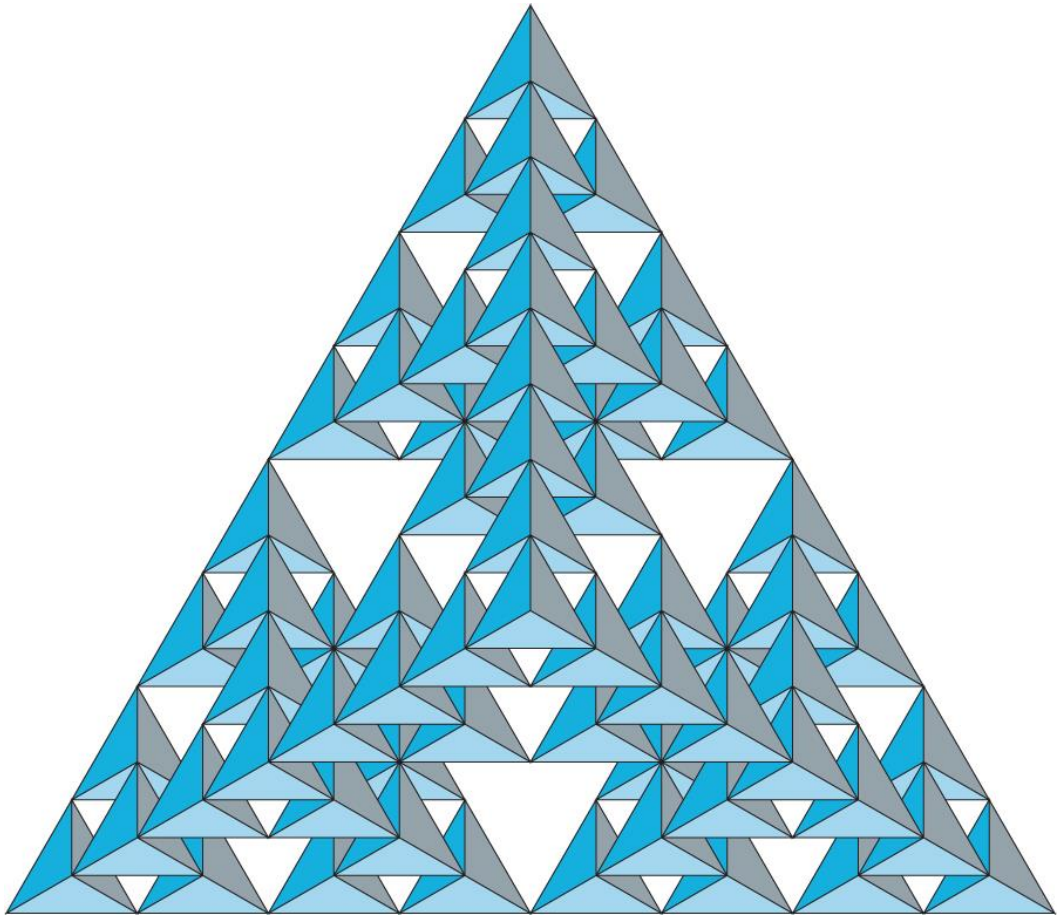
3D Gasket

- We can subdivide each of the four faces.



- Appears as if we remove a solid tetrahedron from the center leaving four smaller tetrahedra.
- Code almost identical to 2D example.

Volume Subdivision



Graphics Functions

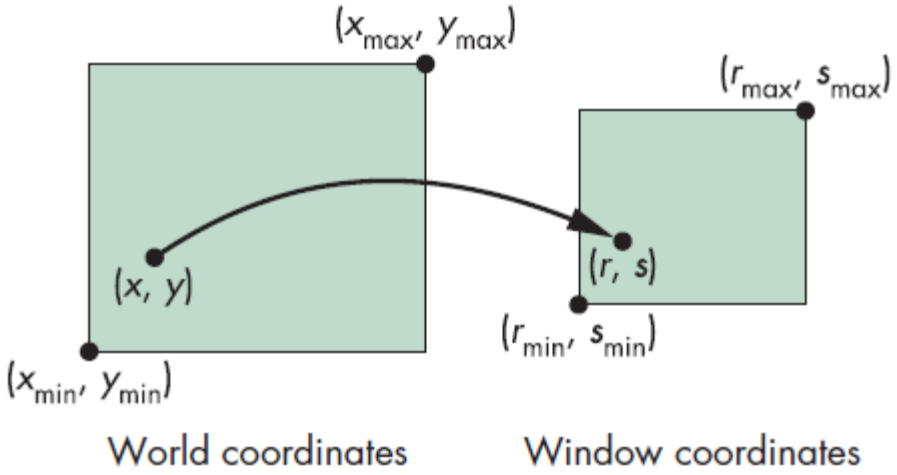
- Seven major groups:
  - Primitive functions
  - Attribute functions
  - Viewing functions
  - Transformation functions
  - Input functions
  - Control functions
  - Query functions

Coordinate Systems

- The application programmers do not need to worry about the details of input and output devices.
- The programmer can define the coordinate system, which is called the world coordinate system, or the application or object coordinate system.
- We will refer to the units that application program uses to specify vertex positions as vertex coordinates.

Coordinate Systems

- Units on the display were first called physical-device coordinates or just device coordinates. For raster devices, e.g., CRT, LCD, we use term window coordinates or screen coordinates.
- At some point, the values in vertex coordinates must be mapped to window coordinates. The mapping is performed automatically as part of the rendering process.

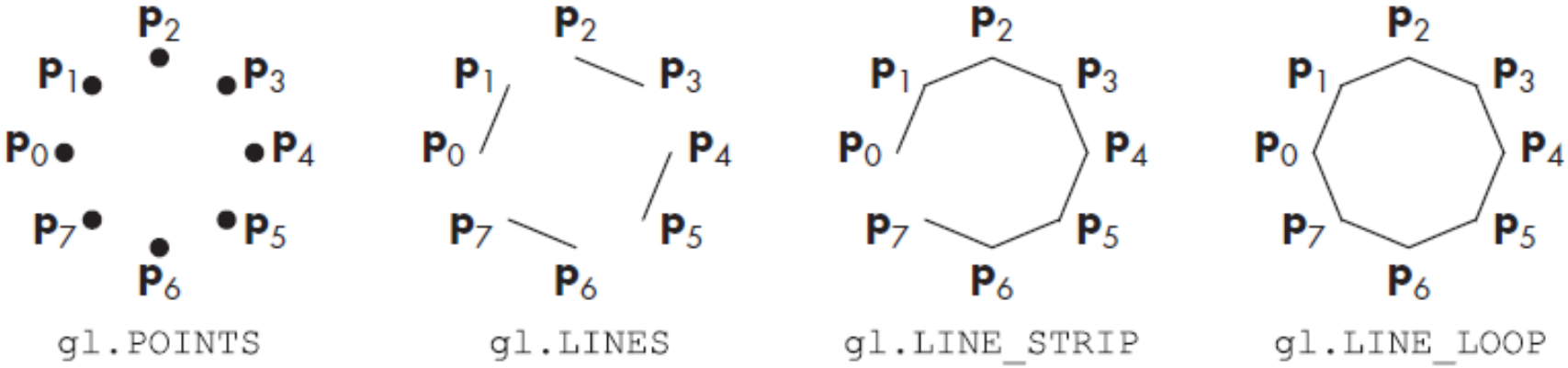


Primitives and Attributes

- We can separate primitives into two classes: geometric primitives and image, or raster, primitives.
  - Geometric primitives: points, line segments, polygon, ...
  - Raster primitives: arrays of pixels

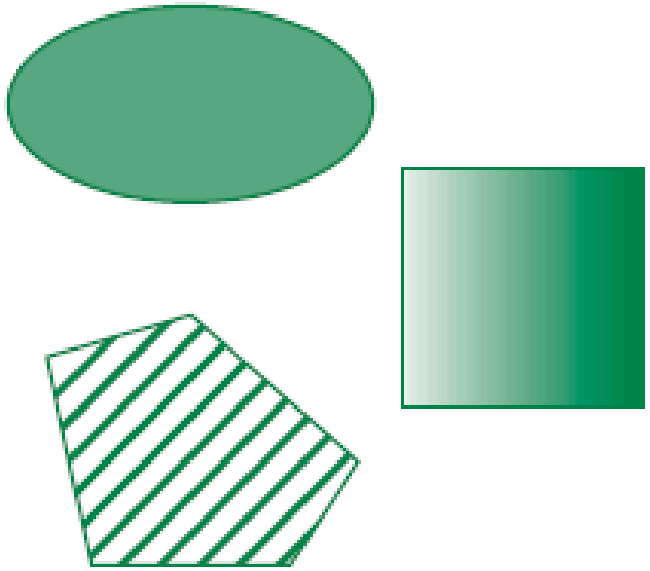
Geometric Primitives

- Point: a single pixel or a small group of pixels
- Line segments: finite sections of lines between two vertices.
  - Can be used for approximation of curves, a graph, edges of closed objects



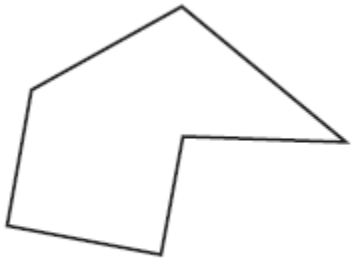
Geometric Primitives

- Polygon: an object that has a border that can be described by a line loop but also has a well-defined interior.
  - Can be used to approximate arbitrary surfaces.

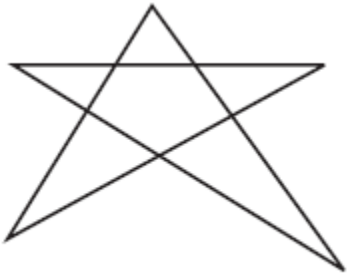


Geometric Primitives

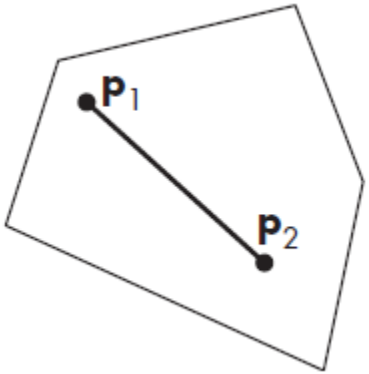
- A polygon must be simple, convex, and flat to be displayed correctly.
  - Simple: In 2D, no two edges of a polygon cross each other.
  - Convex: all points on the line segment between any two points inside the object, or on its boundary, are inside the object.



(a)

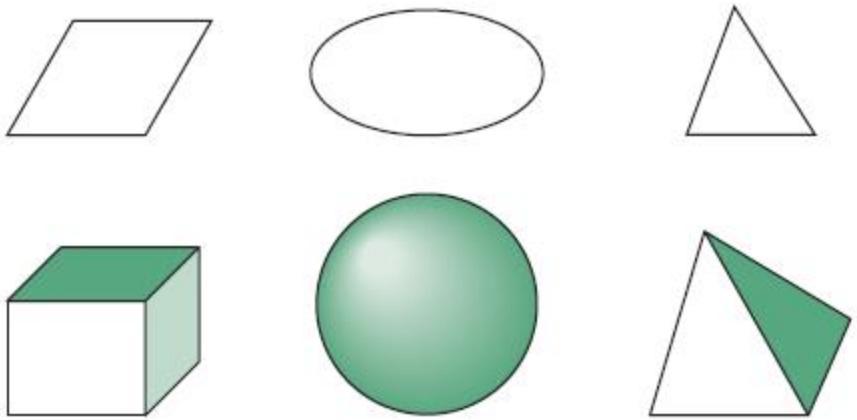


(b)



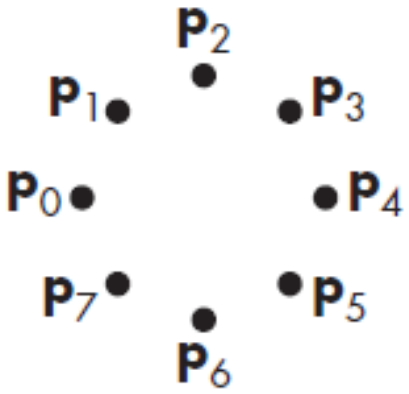
Geometric Primitives

– Convex objects include triangles, tetrahedra, rectangles, circles, spheres, and parallelepipeds.

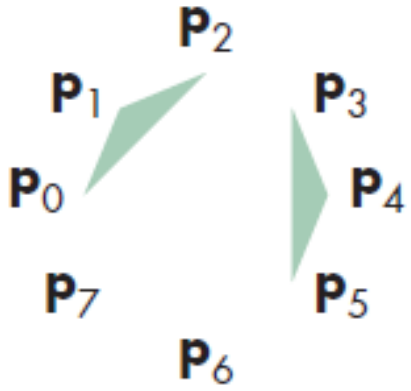


– In 3D, polygon may not lie in the same plane. If we are using triangles, we are safe.

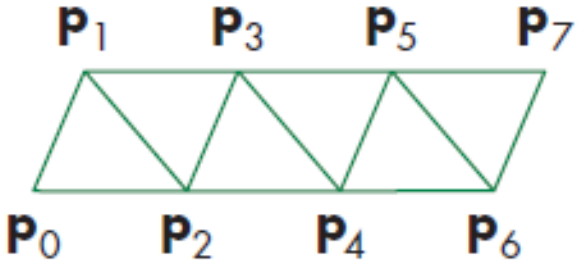
Geometric Primitives



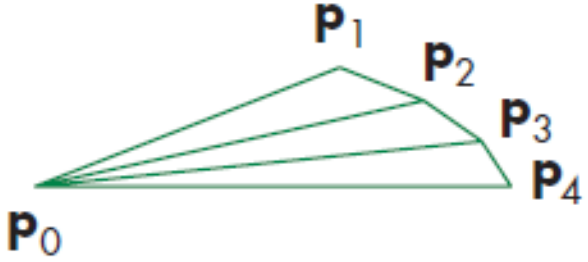
gl.POINTS



gl.TRIANGLES



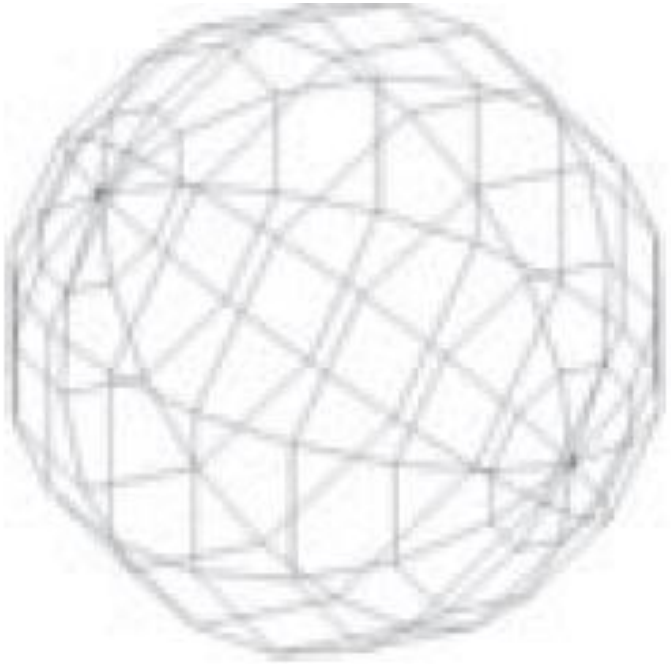
gl.TRIANGLE_STRIP



gl.TRIANGLE_FAN

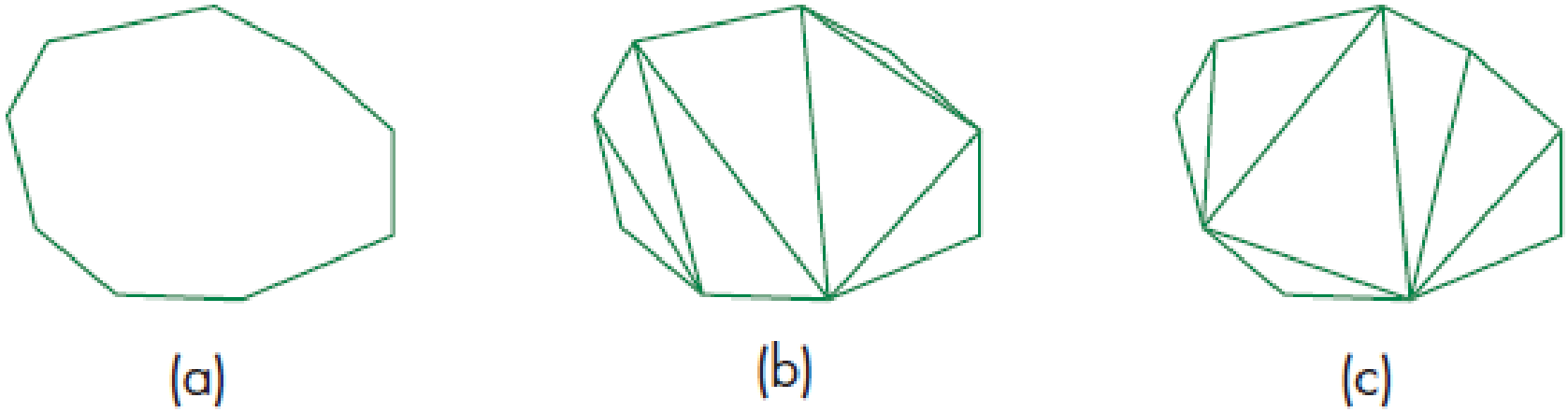
Approximating a Sphere

- Use a set of polygons defined by lines of longitude and latitude.



Triangulation

- A process that starts with a list of vertices and generates a list of triangles.



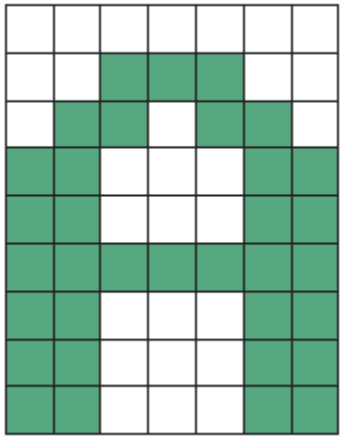
Delaunay triangulation: When we consider the circle determined by any triangle, no other vertex lies in this circle.

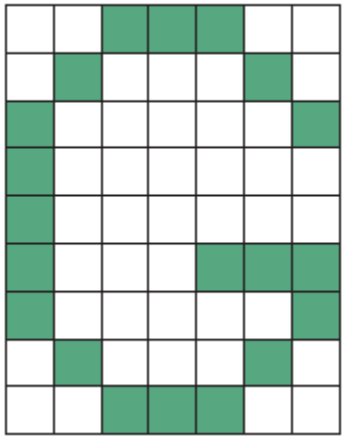
Text

- Stroke text: constructed like other geometric objects.

Computer
Graphics

- Raster text: use bit blocks.





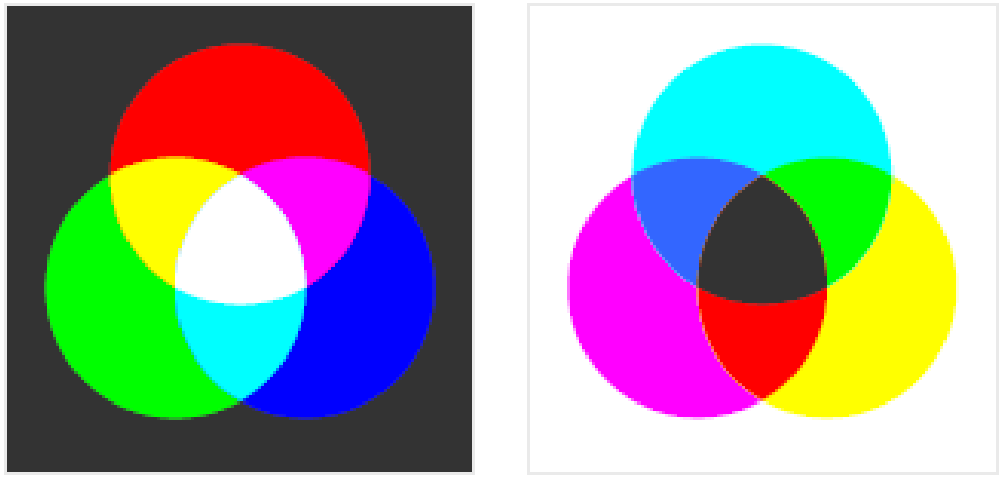
Attributes

- Properties that describe how an object should be rendered are called attributes.
  - Color: black or white
  - Pattern: solid or dashed
  - Transparency: opaque or transparent
  - ……

Color

- Additive color: where the primary colors add together to give the perceived color.
- Subtractive color: assume white light hits a surface, a particular point will be red if all components of the incoming light are absorbed by the surface except for wavelengths in the red part of the spectrum, which is reflected.

Primary and Secondary Colors



(a) (b)

(a) additive colors red, green, and blue can be mixed to produce cyan, magenta, yellow, and white;

(b) subtractive colors cyan, magenta, and yellow can be mixed to produce red, green, blue, and black.

Color

- In a three-primary-color, additive-color RGB system, there are conceptually separate buffers for red, green, and blue images.
- Each pixel has separate red, green, and blue components that correspond to locations in memory.
- In a typical system, there might be a 1280×1024 array of pixels, and each pixel might consist of 24 bits (3 bytes): 1 byte each for red, green, and blue.

Color

- For our 24-bit system, there are 2^{24} possible colors.
- For other systems, e.g., high dynamic range (HDR) device, use 12-bits per color; and some use as few as 4 bits per color.
- Our program should be independent of the particulars of the hardware, we would like to specify a color independently of the number of bits in the framebuffer and let the hardware match our specification.

Color

- A natural technique is to use the color cube and to specify color components as numbers between 0.0 and 1.0, where 1.0 denotes the maximum (or saturated value) of the corresponding primary, and 0.0 denotes a zero value of that primary.

Color

- In application, when we want to assign a color to each vertex, we can put colors into a separate object, such as

```
var colors = [
    vec3(1.0, 0.0, 0.0),
    vec3(0.0, 1.0, 0.0),
    vec3(0.0, 0.0. 1.0)
];
```

which holds the colors red, green, and blue.

Color

- Then put colors in an array as we did with vertex positions.

```
var colorsArray = [ ];

for (var index = 0; index < numPoints; ++index) {

  //determine which color[i] to assign to pointsArray[index]

  colorsArray.push(colors[i]);
}
```

Color

- Alternatively, we could create a single array that contains both vertex locations and vertex colors. These data can be sent to the shaders where colors will be applied to pixels in the framebuffer.

Color

- We can use a four-color (RGBA) system. The fourth color (A, or alpha) is stored in the framebuffer along with the RGB values.
- We can use alpha to create fog effects or combining images.
- Fully transparent A=0.0; fully opaque A=1.0.

```
gl.clearColor(1.0, 1.0, 1.0, 1.0);
```