



Introduction to GLSL (WebGL/OpenGL)

CSU0021: Computer Graphics

What is OpenGL

- A low-level graphics rendering API (application programming interface)
- Generate high-quality images from geometric and image primitives
- Portability
 - Display/window/OS independent

History of OpenGL

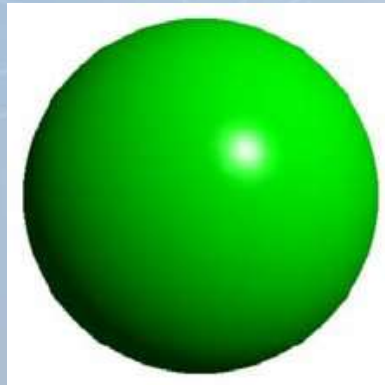
- Iris GL is developed by Silicon Graphics Inc.
- OpenGL 1.0 API is finalized in 1992 and first release in 1993
- OpenGL 1.0 (1993) ~ 1.5(2003)
 - Texture objects, 3D textures, cubemap textures, mipmap generation, shadow map, vertex buffer object ...

History of OpenGL

- OpenGL 2.0 (2004) ~ 2.1 (2006)
 - Vertex and fragment shading (GLSL), pixel buffer objects ...
- OpenGL 3.0 (2008) ~ 3.3 (2010)
 - Framebuffer object, texture buffer objects ...
- OpenGL 4.0 (2010) ~ 4.6 (2017)
 - More modern modules
- https://en.wikipedia.org/wiki/OpenGL#Version_history

Brief Introduction of OpenGL

- Main function of OpenGL – Rendering
- What is rendering?
 - Converting **geometric object description** into **frame buffer values** (convert model to image)
 - A 3D model: a green surface sphere, radius is 3, center at (0, 0, 0), and a light source at (1,1,1)
 - “Render” the above model



OpenGL Example



Cube with flat color



Cube with lighting



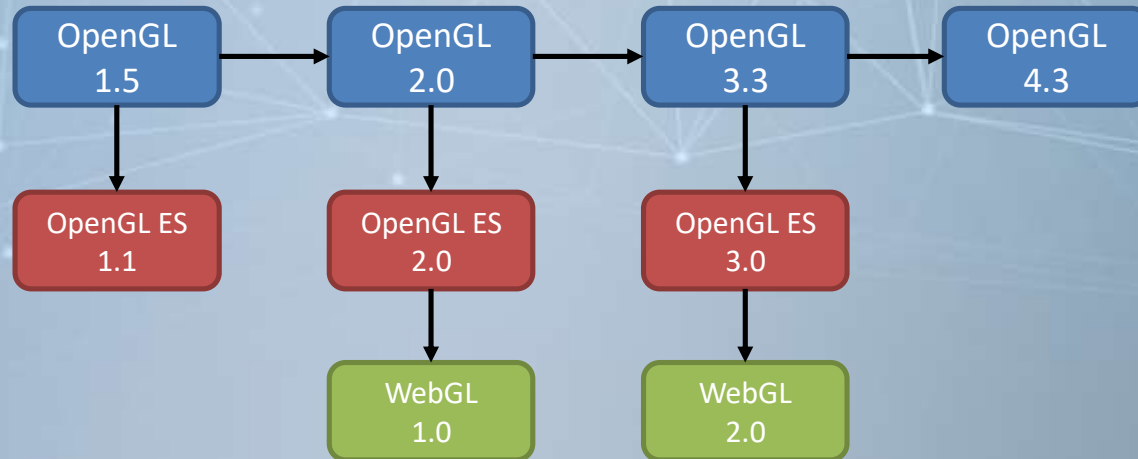
Cube with textures

WebGL

- Javascript implementation of OpenGL
 - Run on web environment
- What you have to know/learn
 - OpenGL basics
 - pipeline architecture, OpenGL Shading Language ...
 - Web basics
 - Javascript

Brief History of WebGL

- OpenGL ES
 - Lightweight OpenGL for embedded system/mobile phone
 - A subset of OpenGL 3.1 API
- WebGL
 - Javascript implementation of ES 2.0
 - Run on browsers
 - <https://en.wikipedia.org/wiki/WebGL>



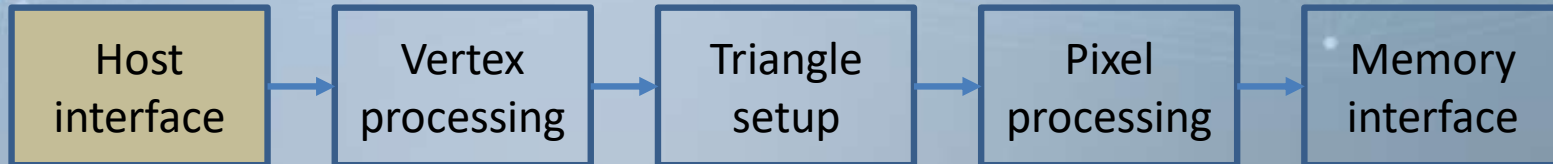
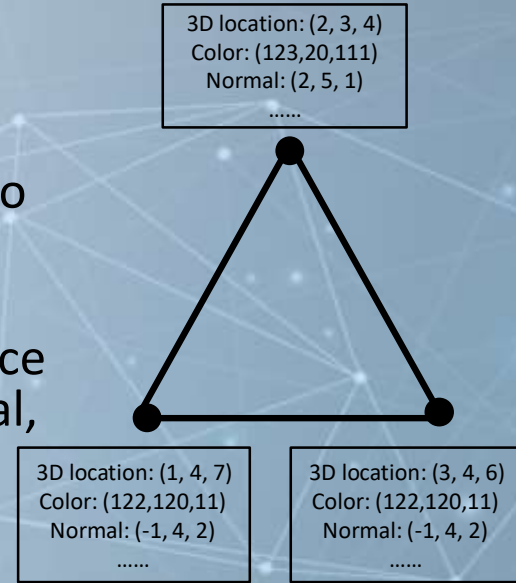
WebGL Rendering Pipeline

- Host interface: move data from CPU to GPU
- Vertex processing: transform vertex from object to screen space
- Triangle setup: rasterization
- Pixel processing: color pixels
- Memory interface: produce final image



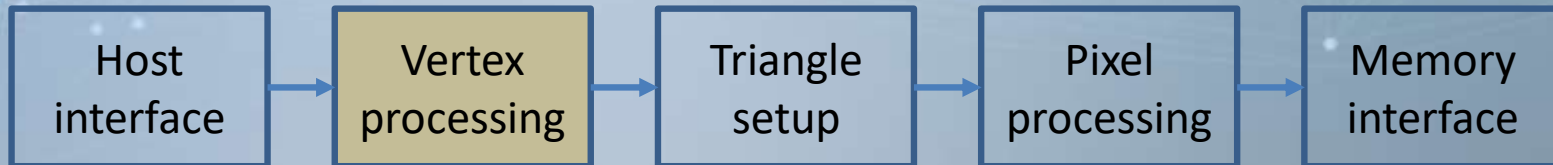
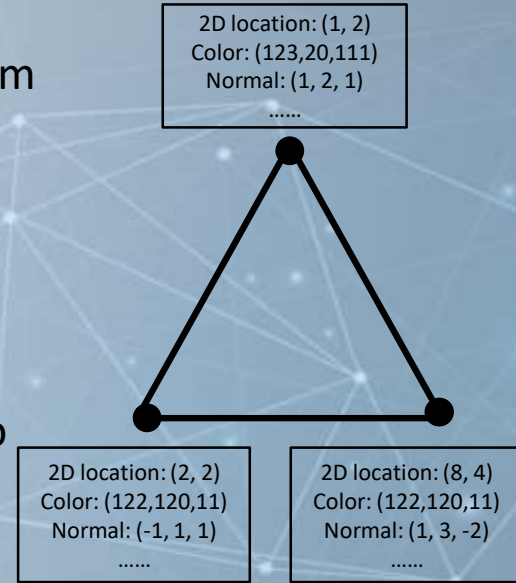
Host Interface

- The host interface is the communication bridge between the CPU and the GPU
- It receives commands from the CPU and also pulls geometry information from system memory
- It outputs a stream of vertices in object space with all their associated information (normal, texture coordinates, per vertex color etc.)



Vertex Processing

- The vertex processing stage receives vertices from the host interface in object space and outputs them in **screen space**
- This may be a simple linear transformation, or a complex operation
- **Normals, texture coordinates etc. are also transformed**
- No new vertices are created in this stage, and no vertices are discarded (input/output has 1: 1 mapping)

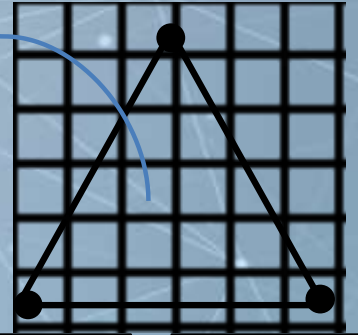


Triangle setup

- In this stage geometry information becomes raster information (screen space geometry is the input, pixel are the output)
- Prior to rasterization, triangles that are backfacing or are located outside the viewing frustum are rejected
- Some GPUs also do some hidden surface removal at this stage

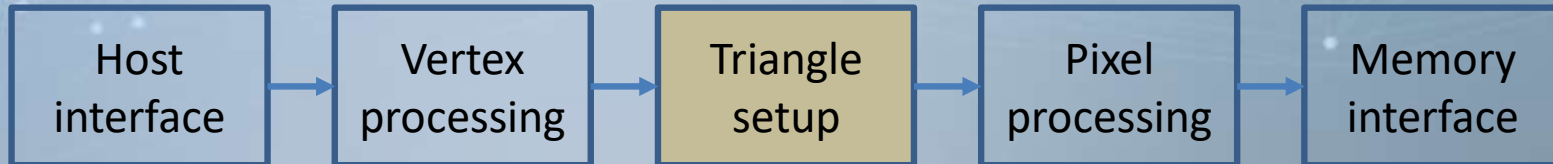
By interpolation from vertices
2D location: (1.4, 3.2)
Color: (122,50,60)
Normal: (-1.5, 2, 1)
.....

2D location: (1, 2)
Color: (123,20,111)
Normal: (1, 2, 1)
.....



2D location: (2, 2)
Color: (122,120,11)
Normal: (-1, 1, 1)
.....

2D location: (8, 4)
Color: (122,120,11)
Normal: (1, 3, -2)
.....

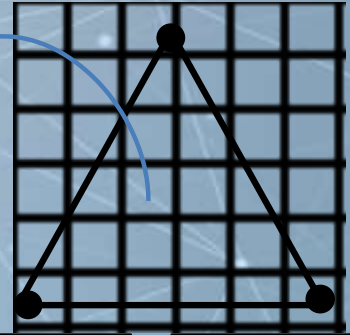


Triangle setup

- A fragment is generated if and only if its center is inside the triangle
- Every fragment generated has its attributes computed to be the perspective correct interpolation of the three vertices that make up the triangle

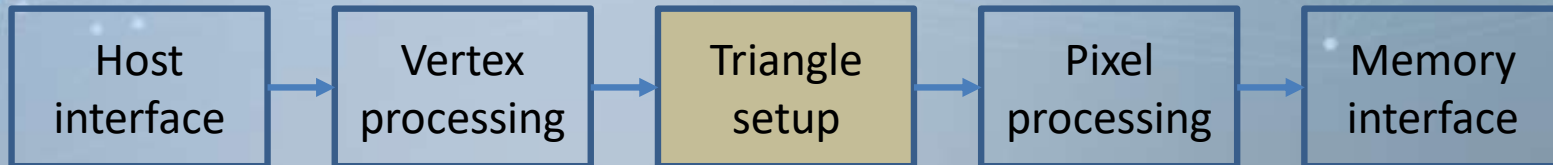
By interpolation from vertices
2D location: (1.4, 3.2)
Color: (122,50,60)
Normal: (-1.5, 2, 1)
.....

2D location: (1, 2)
Color: (123,20,111)
Normal: (1, 2, 1)
.....



2D location: (2, 2)
Color: (122,120,11)
Normal: (-1, 1, 1)
.....

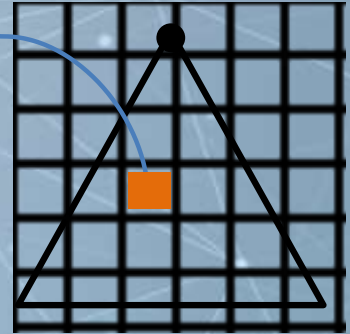
2D location: (8, 4)
Color: (122,120,11)
Normal: (1, 3, -2)
.....



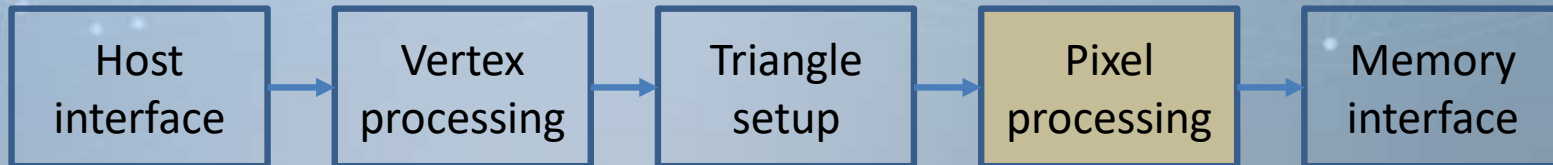
Fragment Processing

- Each fragment provided by triangle setup is fed into fragment processing as a set of attributes (position, normal, texture coordinates etc.), which are used to compute the final color for this pixel
- The computations taking place here include texture mapping and math operations
- Typically, the bottleneck in modern applications

By interpolation from vertices
2D location: (1.4, 3.2)
Color: (122,50,60)
Normal: (-1.5, 2, 1)
.....

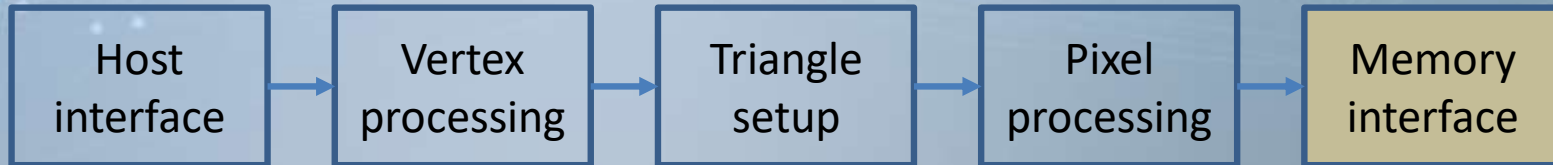


Color every pixel
by the information of the pixel



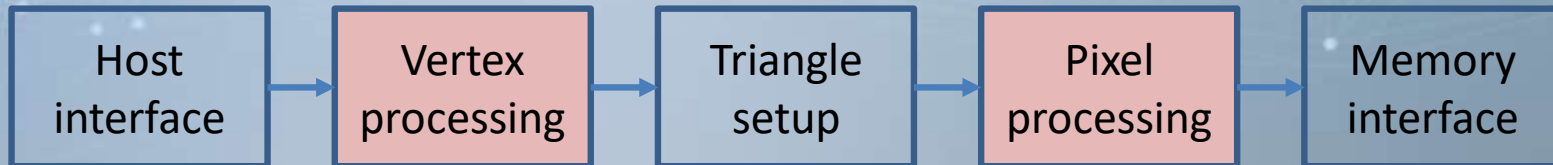
Memory Interface

- Fragment colors provided by the previous stage are written to the framebuffer
- Used to be the biggest bottleneck before fragment processing took over
- Before the final write occurs, some fragments are rejected by the z-buffer, stencil and alpha tests
- On modern GPUs, z and color are compressed to reduce framebuffer bandwidth (but not size)



Programmability in the GPU

- Vertex and fragment processing, and now triangle set-up, are programmable
- The programmer can write programs that are executed for every vertex as well as for every fragment
- This allows fully customizable geometry and shading effects that go well beyond the generic look and feel of older 3D applications

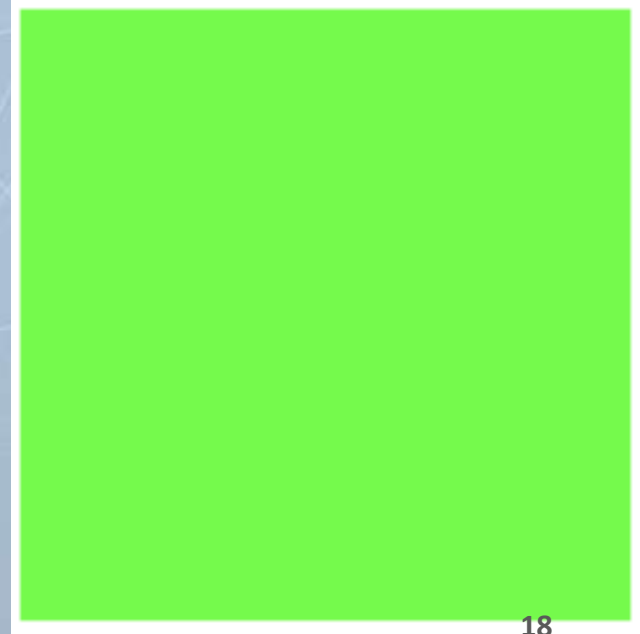


WebGL Programming

- HTML + Javascript + GLSL
- HTML:
 - WebGL renders image on HTML5 “canvas” element
- Javascript:
 - prepare GLSL, data for rendering, and user interface
- GLSL:
 - rendering

WebGL Programming (Ex1-1)

- Just clear the background by a specific color
- Two files in this example
 - Index.html
 - WebGL.js



WebGL Programming (Ex1-1)

index.html

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8" />
<title>WebPage Title </title>
</head>

<body onload="main()">
<canvas id="webgl" width = "400" height = "400">
Please use a browser that support "canvas"
</canvas>
<script src="webGL.js"></script>
</body>
</html>
```

3. Call "main()" in our javascript after <body> is loaded

1. The canvas tag
(where WebGL draws)

2. Javascript for data
and GLSL preparation

WebGL Programming (Ex1-1)

WebGL.js

1. Get the canvas

2. Get the context (gl).
We will use "gl" to setup
everything about
WebGL

```
function main(){  
    ///// get the canvas  
    var canvas = document.getElementById('webgl');  
  
    ///// get the context for draw  
    var gl = canvas.getContext('webgl2');  
    if(!gl){  
        console.log('Failed to get the rendering context for  
        WebGL');  
        return ;  
    }  
  
    ///// clear the screen by designated background color  
    gl.clearColor(0.0, 1.0, 0.0, 1.0); //background color  
    gl.clear(gl.COLOR_BUFFER_BIT); // clear  
}
```

3. Just set the background
color (no clear action here)
(R, G, B, A)

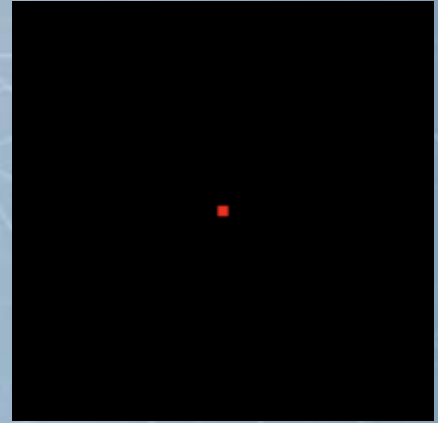
4. Clear screen
by the
background color

Let's Try (5 mins)

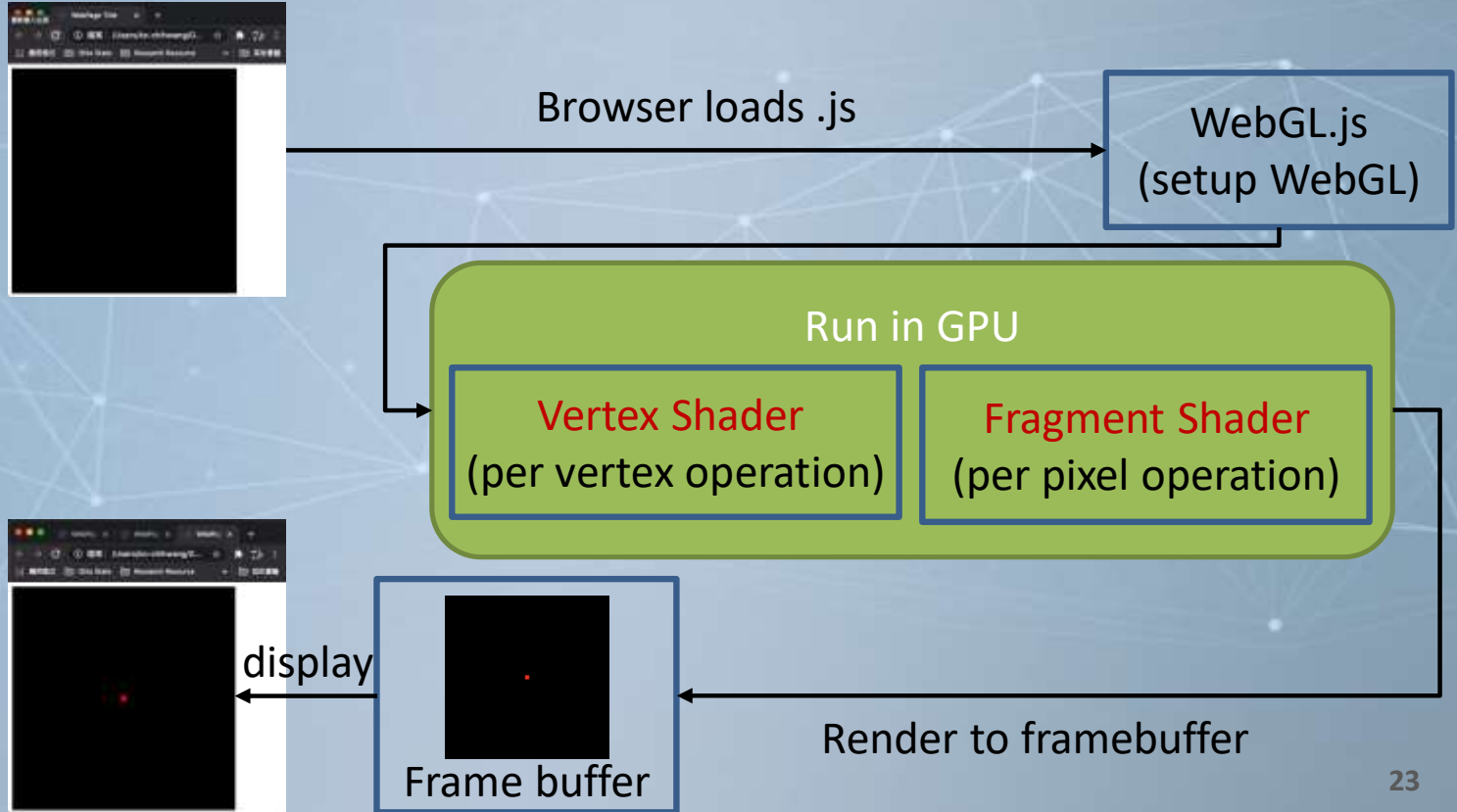
- Download Ex1-1 from Moodle
 - Run it on browser first
- What you can try
 - Change to other background color
 - What if you remove “onload=main()” in index.html
 - What happen if you comment the line “gl.clear()”

WebGL Programming (Ex1-2)

- Draw a point
- Two files in this example
 - Index.html (same as Ex1-1)
 - WebGL.js (javascript and **GLSL**)



WebGL Programming (Ex1-2)



WebGL Programming (Ex1-2)

WebGL.js: main()

```
function main(){
    var canvas = document.getElementById('webgl');

    var gl = canvas.getContext('webgl2');
    if(!gl){
        console.log('Failed to get the rendering context for WebGL');
        return ;
    }

    let renderProgram = compileShader(gl, VSHADER_SOURCE, FSHADER_SOURCE);
    gl.useProgram(renderProgram);

    gl.clearColor(0.0, 0.0, 0.0, 1.0);
    gl.clear(gl.COLOR_BUFFER_BIT);

    gl.drawArrays(gl.POINTS, 0, 1);
}
```

1. Compile vertex and fragment shader

2. Say "I want to use renderProgram" to draw

3. Draw a point

WebGL Programming (Ex1-2)

- `gl.drawArrays()`
 - `drawArrays()` is usually called after you setup everything
 - Call `drawArrays()` to really render a frame
 - We will introduce the details later
- `gl.drawArrays(mode, first, count)`
 - mode: how to draw
 - `GL.POINTS`, `gl.LINES`, `gl.LINE_STRIP`, `gl.LINE_LOOP`, `gl.TRIANGLES`, `gl.TRIANGLE_STRIP`, `gl.TRIANGLE_FAN`
 - first (int): the first vertex for drawing in vertex array
 - count (int): how many vertices to draw

WebGL Programming (Ex1-2)

To draw...we need vertex and fragment shaders and compile it

```
function main(){
    var canvas = document.getElementById('webgl');

    var gl = canvas.getContext('webgl2');
    if(!gl){
        console.log('Failed to get the rendering context for WebGL');
        return ;
    }

    let renderProgram = compileShader(gl, VSHADER_SOURCE, FSHADER_SOURCE);
    gl.useProgram(renderProgram);

    gl.clearColor(0.0, 0.0, 0.0, 1.0);
    gl.clear(gl.COLOR_BUFFER_BIT);

    gl.drawArrays(gl.POINTS, 0, 1);
}
```


WebGL Programming (Ex1-2)

- WebGL.js: shaders
 - The code of shaders can be string variables in .js
 - Use ``` for shader code

```
var VSHADER_SOURCE = `  
void main(){  
    gl_Position = vec4(0.0, 0.0, 0.0, 1.0);  
    gl_PointSize = 10.0;
```

```
}
```

```
`;  
`
```

```
var FSHADER_SOURCE = `  
void main(){  
    gl_FragColor = vec4(1.0, 0.0, 0.0, 1.0);
```

```
}
```

```
`;  
`
```



WebGL Programming (Ex1-2)

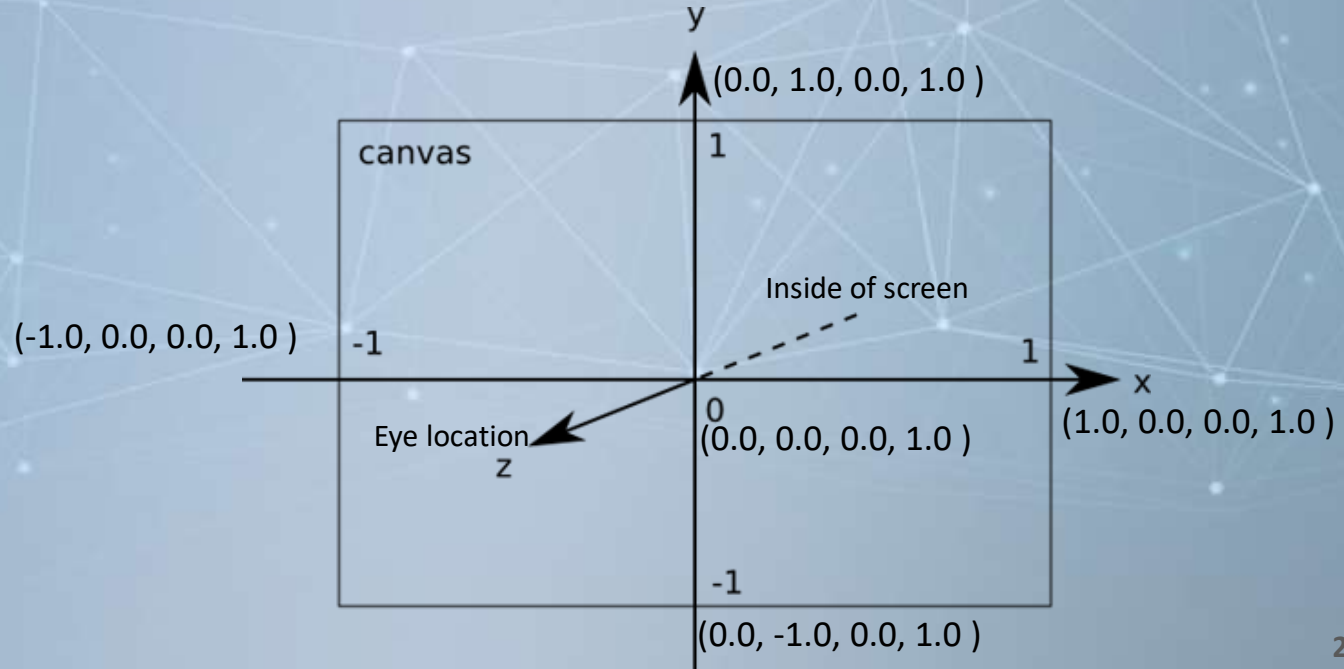
- GLSL build-in variable for vertex shader
 - vec4 gl_Position (vertex location)
 - vec4 is a type in GLSL, a vector with 4 floats
 - float gl_PointSize (vertex size)
 - This is not necessary to assign a value (default: 1.0)
 - float is also a type in GLSL
 - GLSL does not do auto-type conversion.
 - You will get error if you write gl_PointSize = 10

```
var VSHADER_SOURCE = `
void main(){
    gl_Position = vec4(0.0, 0.0, 0.0, 1.0);
    gl_PointSize = 10.0;
}
`;

var FSHADER_SOURCE = `
void main(){
    gl_FragColor = vec4(1.0, 0.0, 0.0, 1.0);
}
`;
```

WebGL Programming (Ex1-2)

- Coordinate system
 - Homogenous system (this is why we use 4 numbers to define a 3D point)
 - $(x, y, z, w) = (x/w, y/w, z/w)$
 - We will introduce homogenous system in detail later



WebGL Programming (Ex1-2)

- GLSL build-in variable for fragment shader
 - `vec4 gl_FragColor` (pixel color)
- In this example, we assign red color to pixels

```
var VSHADER_SOURCE = `
void main(){
    gl_Position = vec4(0.0, 0.0, 0.0, 1.0);
    gl_PointSize = 10.0;
}
`;

var FSHADER_SOURCE = `
void main(){
    gl_FragColor = vec4(1.0, 0.0, 0.0, 1.0);
}
`;
```

WebGL Programming (Ex1-2)

- WebGL.js: function to compile shaders (VSHADER_SOURCE, FSHADER_SOURCE)
 - We ignore details of this utility function here

```
function compileShader(gl, vShaderText, fShaderText){
    //Build vertex and fragment shader objects
    var vertexShader = gl.createShader(gl.VERTEX_SHADER)
    var fragmentShader = gl.createShader(gl.FRAGMENT_SHADER)
    //The way to set up shader text source
    gl.shaderSource(vertexShader, vShaderText)
    gl.shaderSource(fragmentShader, fShaderText)
    //compile vertex shader
    gl.compileShader(vertexShader)
    if(!gl.getShaderParameter(vertexShader, gl.COMPILE_STATUS)){
        console.log('vertex shader error');
        var message = gl.getShaderInfoLog(vertexShader);
        console.log(message);//print shader compiling error message
    }
    //compile fragment shader
    gl.compileShader(fragmentShader)
    if(!gl.getShaderParameter(fragmentShader, gl.COMPILE_STATUS)){
        console.log('fragment shader error');
        var message = gl.getShaderInfoLog(fragmentShader);
        console.log(message);//print shader compiling error message
    }

    //Link shader to program (by a self-define function)
    var program = gl.createProgram();
    gl.attachShader(program, vertexShader);
    gl.attachShader(program, fragmentShader);
    gl.linkProgram(program);
    //if not success, log the program info, and delete it.
    if(!gl.getProgramParameter(program, gl.LINK_STATUS)){
        alert(gl.getProgramInfoLog(program) + "");
        gl.deleteProgram(program);
    }

    return program;
}
```

WebGL Programming (Ex1-2)

- Let's illustrate what the shader is doing in this example
- When `gl.drawArrays(gl.POINTS, 0, 1)` is called, the the shader starts to run
 - In this example, we just say “hey shader, help me drawing one point”, and do not pass any vertex information to shader because we hardcode the information in the shader
- Note: WebGL runs “vertex shader” first, then pass information produced by vertex shader to “fragment shader” to run
 - The vertex shader process all vertices parallelly (but, we only have one vertex here. Let's keep the concept of penalization for later)

```
var VSHADER_SOURCE = `
void main(){
    gl_Position = vec4(0.0, 0.0, 0.0, 1.0);
    gl_PointSize = 10.0;
}
`;

var FSHADER_SOURCE = `
void main(){
    gl_FragColor = vec4(1.0, 0.0, 0.0, 1.0);
}
`;
```

Set center of the
point at the
giving location



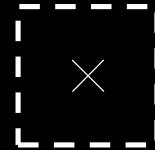
WebGL Programming (Ex1-2)

- Let's illustrate what the shader is doing in this example
- When `gl.drawArrays(gl.POINTS, 0, 1)` is called, the the shader starts to run
 - In this example, we just say “hey shader, help me drawing one point”, and do not pass any vertex information to shader because we hardcode the information in the shader
- Note: WebGL runs “vertex shader” first, then pass information produced by vertex shader to “fragment shader” to run

```
var VSHADER_SOURCE = `
void main(){
    gl_Position = vec4(0.0, 0.0, 0.0, 1.0);
    gl_PointSize = 10.0;
}
`;

var FSHADER_SOURCE = `
void main(){
    gl_FragColor = vec4(1.0, 0.0, 0.0, 1.0);
}
`;
```

Set the region of
the point



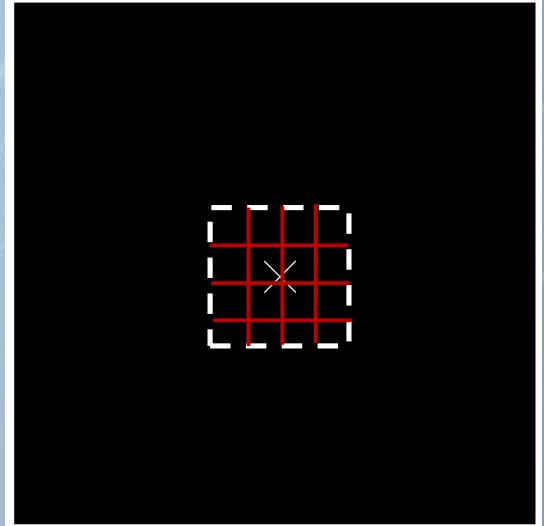
These are all information
produced by vertex shader and
pass to fragment shader

WebGL Programming (Ex1-2)

- **Between** vertex and fragment shader, WebGL **automatically** do many things for you
 - One of them is “rasterization”
 - Subdivide the regions which should be colored into pixels

```
var VSHADER_SOURCE = `
void main(){
    gl_Position = vec4(0.0, 0.0, 0.0, 1.0);
    gl_PointSize = 10.0;
}
`;

var FSHADER_SOURCE = `
void main(){
    gl_FragColor = vec4(1.0, 0.0, 0.0, 1.0);
}
`;
```

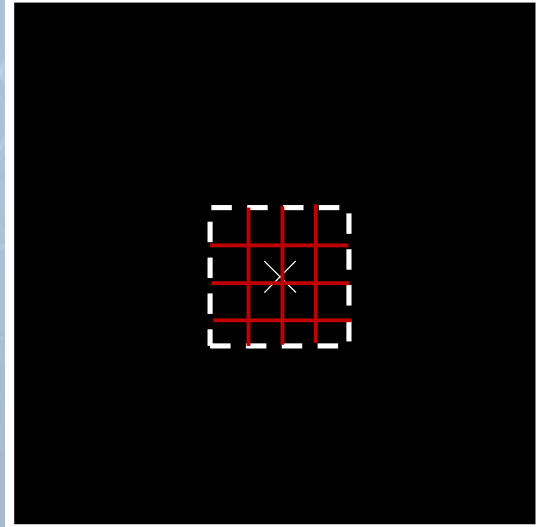


WebGL Programming (Ex1-2)

- Fragment shader
 - The fragment shader process all “pixels” parallely
 - In this example, we have $4*4=16$ pixels after rasterization
 - So, the fragment shader will have 16 copies on 16 threads and each one colors one pixel

```
var VSHADER_SOURCE = `
void main(){
    gl_Position = vec4(0.0, 0.0, 0.0, 1.0);
    gl_PointSize = 10.0;
}
`;

var FSHADER_SOURCE = `
void main(){
    gl_FragColor = vec4(1.0, 0.0, 0.0, 1.0);
}
`;
```

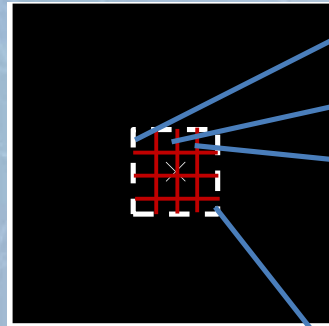


WebGL Programming (Ex1-2)

- Illustration of parallelization on graphics card



GPU threads



```
void main(){  
    gl_FragColor = vec4(1.0, 0.0, 0.0, 1.0);  
}
```

```
void main(){  
    gl_FragColor = vec4(1.0, 0.0, 0.0, 1.0);  
}
```

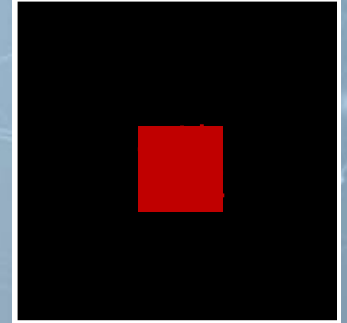
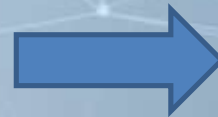
```
void main(){  
    gl_FragColor = vec4(1.0, 0.0, 0.0, 1.0);  
}
```

•

•

•

```
void main(){  
    gl_FragColor = vec4(1.0, 0.0, 0.0, 1.0);  
}
```



Some Variable Types in GLSL

- float, int, bool
 - No double, no long int, no short int.....
 - GLSL uses additional keyboard to control the variable precision. We will see it soon
- Type conversion: newType(oldType)
 - `int a = 10;`
 - `float b = float(a)`
- Other data types
 - float vector: `vec2`, `vec3`, `vec4`
 - Integer vector: `ivec2`, `ivec3`, `ivec4`
 - Boolean vector: `bvec2`, `bvec3`, `bvec4`
 - Matrix: `mat2`, `mat3`, `mat4`
 - `mat2`: 2x2matrix, `mat3`: 3x3 matrix, `mat4`: 4x4 matrix

Let's Try (5 mins)

- Download Ex1-2 from Moodle
 - Try it on browser
- What you can try
 - Draw the point at different location
 - Draw the point with different color
 - What if you change the last element of `gl_Position` to 0.0
 - What if you change the last element of `gl_FragColor` to 0.0
 - What if you change “`gl_PointSize = 10.0`” to “`gl_PointSize = 10`”
 - What if you change “`gl_PointSize = 10.0`” to “`gl_PointSize = float(10)`”

“attribute”, “uniform” and “varying”

- “attribute”, “uniform” and “varying” are keywords for GLSL variables

— ex:

- attribute vec4 a_Position;
- uniform float u_Factor;
- varying vec3 v_Color;

keyword

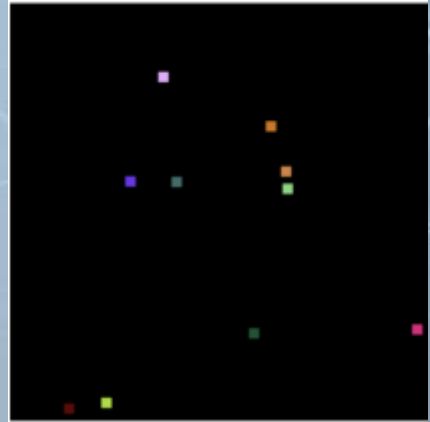
type

variable name

- attribute, uniform variables: pass data from outside of graphics card (javascript) to inside (GLSL)
- varying variables: pass data from vertex and fragment shaders
- Let's introduce “uniform” here, but keep attribute and varying for later

WebGL Programming (Ex1-3)

- Draw multiple points with random color at random locations
- Files
 - index.html
 - WebGL.js



WebGL Programming (Ex1-3)

- WebGL.js: shaders

```
var VSHADER_SOURCE = `
    uniform vec4 u_Position;
    void main(){
        gl_Position = u_Position;
        gl_PointSize = 10.0;
    }
`;
```

Receive a position vector from outside
(javascript)

Assign it to gl_Position (where to draw
a point)

```
var FSHADER_SOURCE = `
    precision mediump float;
    uniform vec4 u_FragColor;
    void main(){
        gl_FragColor = u_FragColor;
    }
`;
```

Receive a color from outside
(javascript)

Use it to draw a point

WebGL Programming (Ex1-3)

- WebGL.js: shaders

```
var VSHADER_SOURCE = `
    uniform vec4 u_Position;
    void main(){
        gl_Position = u_Position;
        gl_PointSize = 10.0;
    }
`;

var FSHADER_SOURCE = `
    precision mediump float;
    uniform vec4 u_FragColor;
    void main(){
        gl_FragColor = u_FragColor;
    }
`;
```

- Determine how much precision the GPU uses when calculating “floats” (this statement is necessary in fragment shader. Otherwise, the shader will not compile)
 - “highp”: high precision
 - “mediump”: medium precision
 - “lowp”: low precision
- Check here for definition of each precision:
http://learnwebgl.brown37.net/12_shader_language/gsl_data_types.html
- Tradeoff: you will see a performance hit if you always use “highp”

WebGL Programming (Ex1-3)

- WebGL.js: shaders

```
var VSHADER_SOURCE = `
    uniform vec4 u_Position;
    void main(){
        gl_Position = u_Position;
        gl_PointSize = 10.0;
    }
`;

var FSHADER_SOURCE = `
    precision mediump float;
    uniform vec4 u_FragColor;
    void main(){
        gl_FragColor = u_FragColor;
    }
`;
```

- Suggestion:
 - “highp” for vertex position
 - “mediump” for texture coordinate
 - “lowp” for colors
- This statements says that “use MEDIUM PRECISION to calculate ALL floats in this fragment shader”
- Can we assign different variables to different precision?
 - Yes
 - Ex: uniform highp vec4 u_FragColor;

WebGL Programming (Ex1-3)

- How to pass data to the uniform variable in shaders?
 - 1. get reference of the uniform variable
 - `gl.getUniformLocation()`
 - 2. pass data
 - `gl.uniform4f(location, v0, v1, v2, v3)`: to a uniform **vec4**
 - `gl.uniform3f(location, v0, v1, v2)`: to a uniform **vec3**
 - `gl.uniform2f(location, v0, v1)`: to a uniform **vec2**
 - `gl.uniform1f(location, v0)`: to a uniform **float**

WebGL Programming (Ex1-3)

- WebGL.js: main()

1. Compile shaders and use the compiled program

2. Get reference of “u_Position” from the program, “renderProgram”

3. Keep the reference for use later (you can just use any javascript variable to keep it)

```
function main(){
  var canvas = document.getElementById('webgl');
  var gl = canvas.getContext('webgl2');
  if(!gl){
    console.log('Failed to get the rendering context for WebGL');
    return ;
  }

  let renderProgram = compileShader(gl, VSHADER_SOURCE, FSHADER_SOURCE);
  gl.useProgram(renderProgram);

  renderProgram.u_Position = gl.getUniformLocation(renderProgram, 'u_Position');
  renderProgram.u_FragColor = gl.getUniformLocation(renderProgram, 'u_FragColor');

  gl.clearColor(0.0, 0.0, 0.0, 1.0);
  gl.clear(gl.COLOR_BUFFER_BIT);

  for(let i = 0; i < 10; i++){
    gl.uniform4f(renderProgram.u_Position, Math.random() * 2.0 - 1.0, Math.random() * 2.0 - 1.0, 0.0, 1.0);
    gl.uniform4f(renderProgram.u_FragColor, Math.random(), Math.random(), Math.random(), 1.0);
    gl.drawArrays(gl.POINTS, 0, 1);
  }
}
```

WebGL Programming (Ex1-3)

- WebGL.js: main()

```
function main(){
  var canvas = document.getElementById('webgl');
  var gl = canvas.getContext('webgl2');
  if(!gl){
    console.log('Failed to get the rendering context for WebGL');
    return ;
  }

  let renderProgram = compileShader(gl, VSHADER_SOURCE, FSHADER_SOURCE);
  gl.useProgram(renderProgram);

  renderProgram.u_Position = gl.getUniformLocation(renderProgram, 'u_Position');
  renderProgram.u_FragColor = gl.getUniformLocation(renderProgram, 'u_FragColor');

  gl.clearColor(0.0, 0.0, 0.0, 1.0);
  gl.clear(gl.COLOR_BUFFER_BIT);

  for(let i = 0; i < 10; i++){
    gl.uniform4f(renderProgram.u_Position, Math.random() * 2.0 - 1.0, Math.random() * 2.0 - 1.0, 0.0, 1.0);
    gl.uniform4f(renderProgram.u_FragColor, Math.random(), Math.random(), Math.random(), 1.0);
    gl.drawArrays(gl.POINTS, 0, 1);
  }
}
```

Use "gl.uniform4f" to pass data to a uniform vec4 variable in shader

Four floats because we use gl.uniform4f

Let's Try

- Download Ex1-3 from Moodle
 - Try it on browser
- What you can try
 - What if you comment “gl.useProgram()”
 - Change ‘renderProgram.u_Position’ at Line 66 to any variable name, such as “abc”. Also, change ‘renderProgram.u_Position’ to “abc” at Line 73. Then, run it.
 - What if you change all “u_Position” in vertex shader to “position”. Do you get error message? If so, change ‘u_Position’ at Line 66 to ‘position’. Is that fixed?

Compile the Shaders

- The compilation functions in the example code
 - compileShader()

Take the context, vertex shader source and fragment shader source

Create shader object and tell what type (vertex or fragment) shader you want

Assign shader source code to shader objects

```
function compileShader(gl, vShaderText, fShaderText){
    //Build vertex and fragment shader objects
    var vertexShader = gl.createShader(gl.VERTEX_SHADER)
    var fragmentShader = gl.createShader(gl.FRAGMENT_SHADER)
    //The way to set up shader text source
    gl.shaderSource(vertexShader, vShaderText)
    gl.shaderSource(fragmentShader, fShaderText)
    //compile vertex shader
    gl.compileShader(vertexShader)
    if(!gl.getShaderParameter(vertexShader, gl.COMPILE_STATUS)){
        //print shader compiling error message
    }
    //compile fragment shader
    gl.compileShader(fragmentShader)
    if(!gl.getShaderParameter(fragmentShader, gl.COMPILE_STATUS)){
        //print shader compiling error message
    }

    //link shader to program (by a self-define function)
    var program = gl.createProgram();
    gl.attachShader(program, vertexShader);
    gl.attachShader(program, fragmentShader);
    gl.linkProgram(program);
    //if not success, log the program info, and delete it.
    if(!gl.getProgramParameter(program, gl.LINK_STATUS)){
        alert(gl.getProgramInfoLog(program) + "");
        gl.deleteProgram(program);
    }

    return program;
}
```

Compile the Shaders

- The compilation functions in the example code
 - compileShader()

Compile shaders

```
function compileShader(gl, vShaderText, fShaderText){
    //Build vertex and fragment shader objects
    var vertexShader = gl.createShader(gl.VERTEX_SHADER)
    var fragmentShader = gl.createShader(gl.FRAGMENT_SHADER)
    //The way to set up shader text source
    gl.shaderSource(vertexShader, vShaderText)
    gl.shaderSource(fragmentShader, fShaderText)
    //compile vertex shader
    gl.compileShader(vertexShader)
    if(!gl.getShaderParameter(vertexShader, gl.COMPILE_STATUS)){
        //print shader compiling error message
    }
    //compile fragment shader
    gl.compileShader(fragmentShader)
    if(!gl.getShaderParameter(fragmentShader, gl.COMPILE_STATUS)){
        //print shader compiling error message
    }

    //link shader to program (by a self-define function)
    var program = gl.createProgram();
    gl.attachShader(program, vertexShader);
    gl.attachShader(program, fragmentShader);
    gl.linkProgram(program);
    //if not success, log the program info, and delete it.
    if(!gl.getProgramParameter(program, gl.LINK_STATUS)){
        alert(gl.getProgramInfoLog(program) + "");
        gl.deleteProgram(program);
    }

    return program;
}
```

Compile the Shaders

- The compilation functions in the example code
 - `compileShader()`

```
function compileShader(gl, vShaderText, fShaderText){
    //Build vertex and fragment shader objects
    var vertexShader = gl.createShader(gl.VERTEX_SHADER)
    var fragmentShader = gl.createShader(gl.FRAGMENT_SHADER)
    //The way to set up shader text source
    gl.shaderSource(vertexShader, vShaderText)
    gl.shaderSource(fragmentShader, fShaderText)
    //compile vertex shader
    gl.compileShader(vertexShader)
    if(!gl.getShaderParameter(vertexShader, gl.COMPILE_STATUS)){
        //print shader compiling error message
    }
    //compile fragment shader
    gl.compileShader(fragmentShader)
    if(!gl.getShaderParameter(fragmentShader, gl.COMPILE_STATUS)){
        //print shader compiling error message
    }

    //link shader to program (by a self-define function)
    var program = gl.createProgram();
    gl.attachShader(program, vertexShader);
    gl.attachShader(program, fragmentShader);
    gl.linkProgram(program);
    //if not success, log the program info, and delete it.
    if(!gl.getProgramParameter(program, gl.LINK_STATUS)){
        alert(gl.getProgramInfoLog(program) + "");
        gl.deleteProgram(program);
    }

    return program;
}
```

Handle the error if the compilation fails

Compile the Shaders

- The compilation functions in the example code
 - compileShader()

```
function compileShader(gl, vShaderText, fShaderText){
    //Build vertex and fragment shader objects
    var vertexShader = gl.createShader(gl.VERTEX_SHADER)
    var fragmentShader = gl.createShader(gl.FRAGMENT_SHADER)
    //The way to set up shader text source
    gl.shaderSource(vertexShader, vShaderText)
    gl.shaderSource(fragmentShader, fShaderText)
    //compile vertex shader
    gl.compileShader(vertexShader)
    if(!gl.getShaderParameter(vertexShader, gl.COMPILE_STATUS)){
        //print shader compiling error message
    }
    //compile fragment shader
    gl.compileShader(fragmentShader)
    if(!gl.getShaderParameter(fragmentShader, gl.COMPILE_STATUS)){
        //print shader compiling error message
    }

    //link shader to program (by a self-define function)
    var program = gl.createProgram();
    gl.attachShader(program, vertexShader);
    gl.attachShader(program, fragmentShader);
    gl.linkProgram(program);
    //if not success, log the program info, and delete it.
    if(!gl.getProgramParameter(program, gl.LINK_STATUS)){
        alert(gl.getProgramInfoLog(program) + "");
        gl.deleteProgram(program);
    }

    return program;
}
```

Create a GPU program and link compiled shader objects to it

Compile the Shaders

- The compilation functions in the example code
 - `compileShader()`

```
function compileShader(gl, vShaderText, fShaderText){
    /////Build vertex and fragment shader objects
    var vertexShader = gl.createShader(gl.VERTEX_SHADER)
    var fragmentShader = gl.createShader(gl.FRAGMENT_SHADER)
    //The way to set up shader text source
    gl.shaderSource(vertexShader, vShaderText)
    gl.shaderSource(fragmentShader, fShaderText)
    //compile vertex shader
    gl.compileShader(vertexShader)
    if(!gl.getShaderParameter(vertexShader, gl.COMPILE_STATUS)){
        //print shader compiling error message
    }
    //compile fragment shader
    gl.compileShader(fragmentShader)
    if(!gl.getShaderParameter(fragmentShader, gl.COMPILE_STATUS)){
        //print shader compiling error message
    }

    /////link shader to program (by a self-define function)
    var program = gl.createProgram();
    gl.attachShader(program, vertexShader);
    gl.attachShader(program, fragmentShader);
    gl.linkProgram(program);
    //if not success, log the program info, and delete it.
    if(!gl.getProgramParameter(program, gl.LINK_STATUS)){
        alert(gl.getProgramInfoLog(program) + "");
        gl.deleteProgram(program);
    }

    return program;
}
```

Check and handle linking error