Đồ họa

Tuần 2

Giảng viên: Trần Đức Minh

Nội dung bài giảng

- Tìm hiểu về Shaders
- Giới thiệu về GLSL
- Hoạt động của WebGL
- Vẽ các đối tượng cơ bản
- Vẽ đa giác
- Các ví dụ minh họa

Shaders

- **Shaders** là các chương trình chạy bên trong GPU nhằm xử lý một nhiệm vụ cụ thể nào đó.
- Mã nguồn lập trình shader sử dụng **ngôn ngữ tựa C** được gọi là **OpenGL Shading Language (GLSL).**
- Có 2 shaders
  - Vertex shader
  - Fragment shader
- Tương ứng với mỗi đỉnh của đối tượng, Vertex shader sẽ được thực hiện trước, sau đó đến lượt Fragment shader.

Vertex Shader

- **Vertex shader** chứa đoạn mã chương trình dùng để chạy trên mỗi đỉnh của đối tượng truyền vào.
- Sau khi nhận dữ liệu được truyền vào từ bên ngoài (dữ liệu đỉnh, màu sắc, các hệ số, …), điểm bắt đầu thực thi của chương trình là **hàm main()**
  - Ví dụ: Đối tượng được hình thành bởi n đỉnh thì hàm main() bên trong vertex shader cũng được gọi n lần tương ứng với mỗi đỉnh truyền vào.
- **Vertex shader xử lý dữ liệu của mỗi đỉnh** như tính toán biến đổi tọa độ đỉnh, chuẩn hóa, xử lý màu, …
  - Ví dụ: Tịnh tiến đỉnh, chuyển đổi giá trị từ trục tọa độ người dùng sang trục tọa độ WebGL.

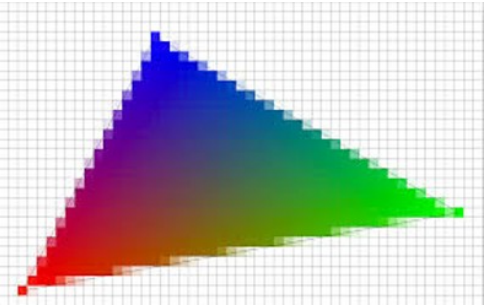
```
<script id="vertex-shader" type="x-shader/x-vertex">
#version 300 es

in vec2 aCoordinates;

void main()
{
    gl_Position = vec4(aCoordinates, 0.0, 1.0);;
}
</script>
```

Vertex Shader

- **gl_Position**
  - là một biến được định nghĩa sẵn bên trong vertex shader, **dùng để chứa vị trí đỉnh** mà giá trị của nó xác định trên trục tọa độ WebGL.
  - Sau khi được truyền giá trị, gl_Position sẽ được sử dụng bởi các công việc tiếp theo bên trong đường ống đồ họa.

Fragment Shader

- Tương ứng với mỗi đỉnh, **Fragment shader** chứa đoạn mã chương trình được thực hiện ngay sau khi hàm main() bên trong Vertex shader kết thúc.
- **Fragment shader** có nhiệm vụ tính toán và xác định màu sắc của mỗi pixel liên quan đến đỉnh hay một tổ hợp đỉnh.
- Phụ thuộc vào loại đối tượng cơ bản cần vẽ mà Fragment shader sẽ tô màu cho phù hợp. Ta có các đối tượng cơ bản cần vẽ như sau:
  - **Điểm**: Tô pixel theo màu được truyền cho Fragment shader
  - **Đoạn thẳng**: Các pixels nằm trên đường thẳng được tô màu theo dạng nội suy dựa trên hai màu truyền cho 2 đầu mút đoạn thẳng.
  - **Tam giác**: Các pixels nằm bên trong tam giác được tô màu theo dạng nội suy dựa trên màu truyền cho các đỉnh tam giác.



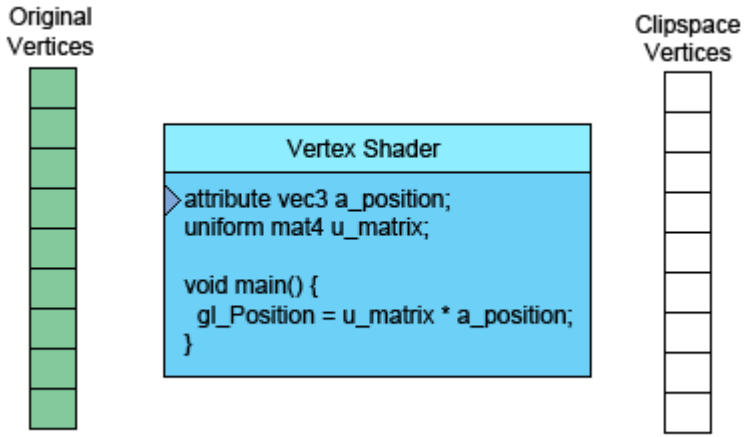
Fragment Shader

- Sau khi nhận dữ liệu được truyền đến từ Vertex shader (có thể có hoặc không), điểm bắt đầu thực thi của Fragment shader là **hàm main()**
- Đầu ra của hàm main() trong Fragment shader là **màu sắc của đỉnh** mà Vertex shader vừa xử lý.
- Ví dụ:

```
<script id="fragment-shader" type="x-shader/x-fragment">
#version 300 es
precision mediump float;

out vec4 fColor;

void main()
{
    fColor = vec4( 1.0, 0.0, 0.0, 1.0 );
}
</script>
```

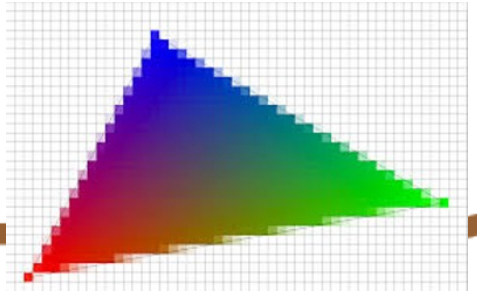
Ví dụ về hoạt động của các Shaders

- Giả sử ta muốn vẽ 3 tam giác, mỗi tam giác có 3 đỉnh. Vậy tổng số đỉnh ta cần đưa vào Vertex shader để xử lý là 3x3=9 đỉnh.
  - **Vertex Shader** được thực hiện lặp đi lặp lại 9 lần tương ứng với 9 đỉnh.
  - gl_Position sẽ chứa giá trị đỉnh mới sau khi tính toán. GPU sẽ lưu giữ lại giá trị trong gl_Position để xử lý tiếp.



Ví dụ về hoạt động của các Shaders

- Hoạt động của Fragment Shader
  - Để vẽ một hình tam giác, ta phải dựa trên **tọa độ** và **màu sắc** của 3 pixels tương ứng với 3 đỉnh của hình tam giác.
  - Hệ thống sẽ tự động rasterizer (sinh ra fragment) cho tam giác.
  - **Fragment Shader** được dùng để tô màu cho fragment, cụ thể là xác định màu cho từng pixel bên trong fragment
    - Trường hợp **3 đỉnh tam giác cùng màu** thì màu của các pixel bên trong tam giác sẽ trùng màu với đỉnh.
    - Trường hợp **3 đỉnh tam giác khác màu** thì màu của các pixel bên trong tam giác sẽ được **nội suy**.



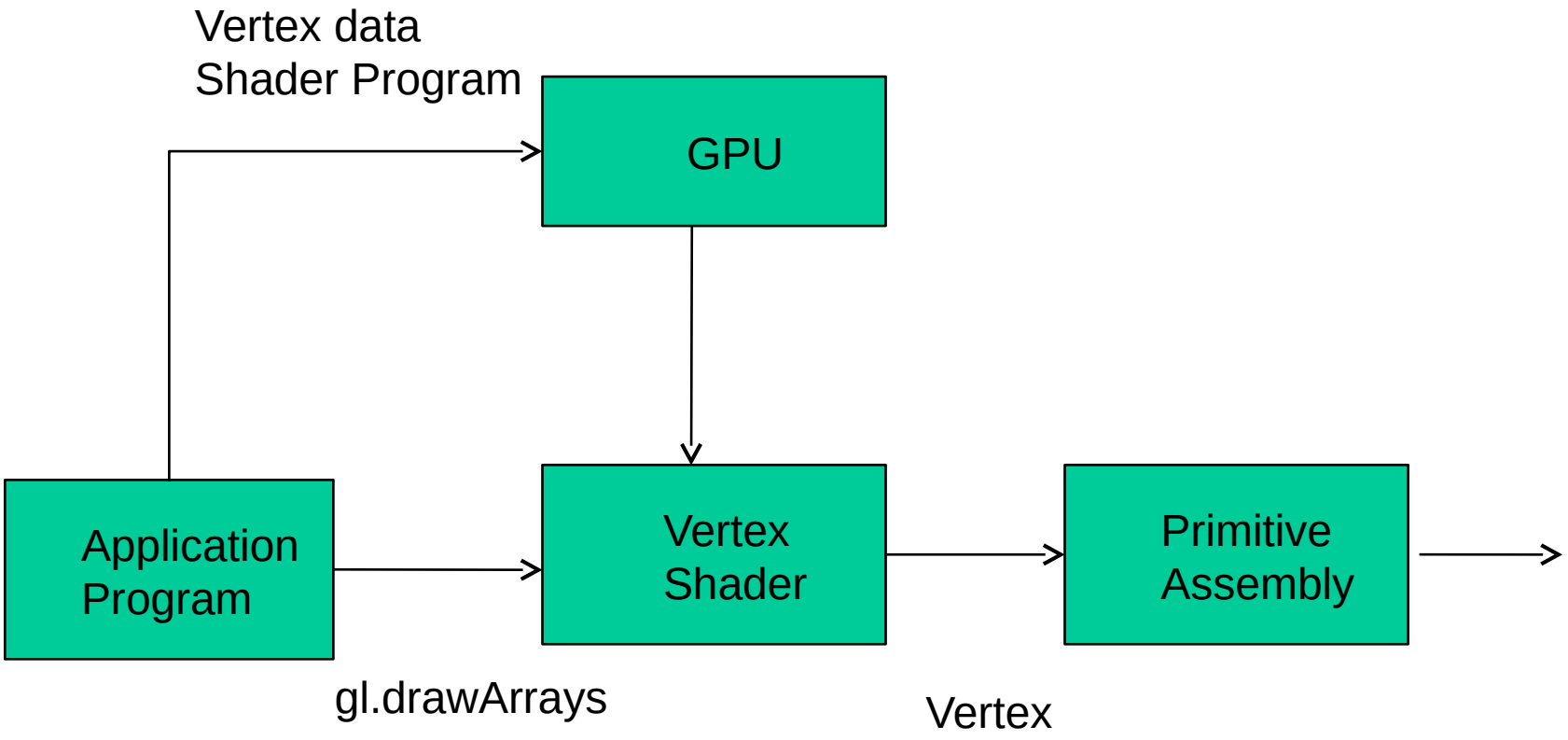
Liên kết thư viện WebGL với các shaders

- Mã nguồn JavaScript

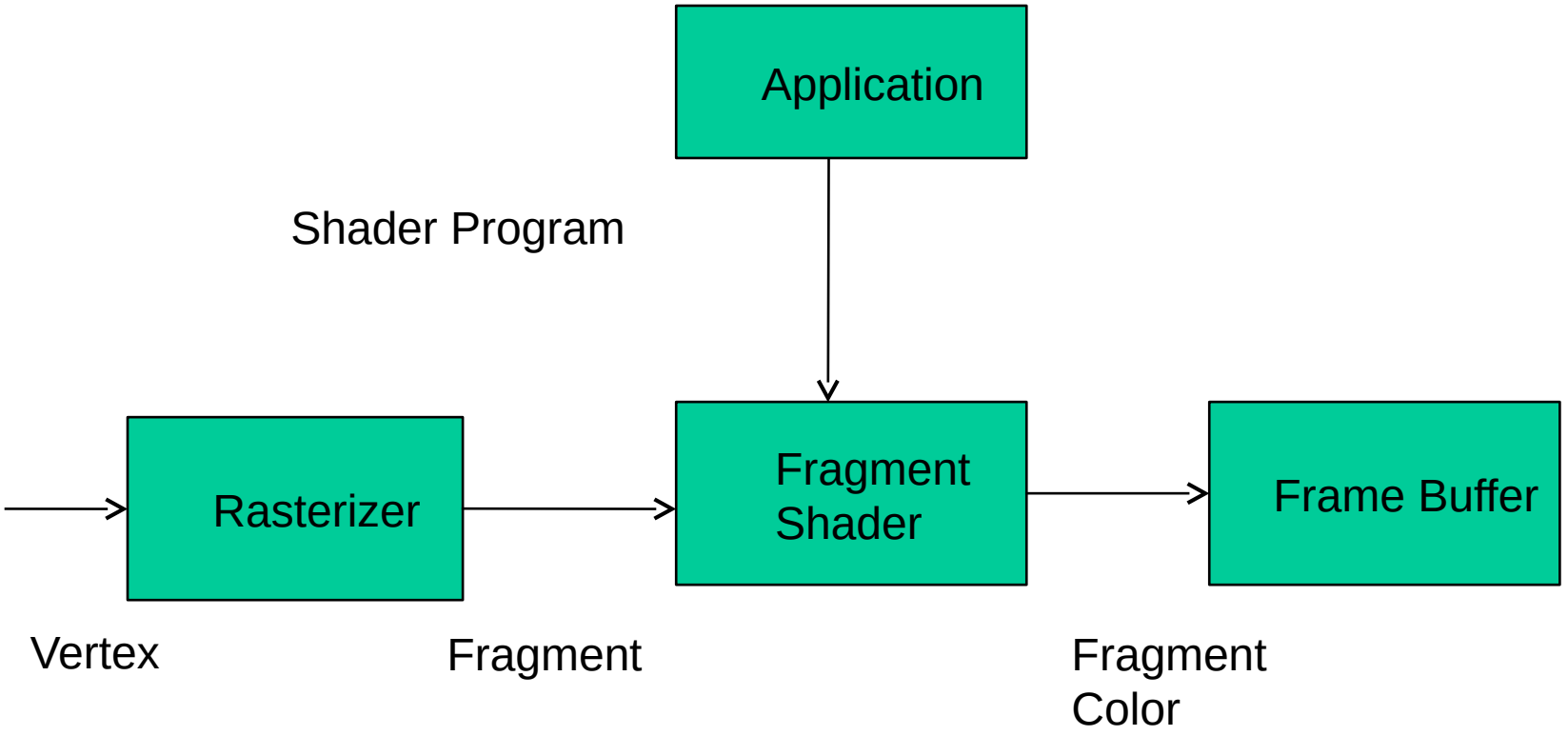
```
canvas = document.getElementById("gl-canvas");
gl = canvas.getContext('webgl2');
if (!gl) {
    alert("WebGL 2.0 isn't available");
}
var program = initShaders(gl, "vertex-shader", "fragment-shader");
gl.useProgram(program);
```

- Trong đó:
  - **gl-canvas** là tên id của đối tượng canvas
  - **vertex-shader** là tên id của Vertex shader
  - **fragment-shader** là tên id của Fragment shader
  - Hàm **getContext** với tham số **“webgl2”** dùng để trả về một đối tượng **WebGL2RenderingContext** mà cho phép vẽ lên canvas các hình ảnh 3 chiều có sử dụng thư viện đồ họa WebGL 2.0
  - **initShaders** là hàm trong file thư viện **/Common/initShaders.js**, hỗ trợ việc khởi tạo và liên kết các shaders với đối tượng **WebGL2RenderingContext** để tính toán và vẽ lên canvas.
    - Hàm initShaders trả về đối tượng được tạo ra bởi hàm **gl.createProgram()**, đây là đối tượng có kiểu **WebGLProgram** dùng để kết hợp hai shaders sau khi đã được biên dịch với mục đích **liên kết các shaders thành một chương trình**.

Mô hình thực thi



Mô hình thực thi



Ngôn ngữ GLSL

- Ngôn ngữ GLSL (OpenGL Shader Language) được đưa ra từ phiên bản OpenGL 2.0 vào năm 2004.
- Một số đặc điểm cơ bản của GLSL
  - Là một **ngôn ngữ biên dịch.**
  - Cú pháp **tương tự ngôn ngữ C/C++**.
  - Được dùng để tạo ra các shader **thực thi bên trong GPU**.

Các kiểu dữ liệu (KDL) của GLSL

KDL GLSL	KDL trong C	Mô tả
void	**void**	Đại diện cho một giá trị rỗng
bool	**int**	**true** hoặc **false**
int	**int**	Số nguyên có dấu
float	**float**	Số thực dấu chấm động
vec2, vec3, vec4	**float[n]**	véc-tơ chứa n (2, 3, 4) phần tử số thực
bvec2, bvec3, bvec4	**int[n]**	véc-tơ chứa n (2, 3, 4) phần tử kiểu bool
ivec2, ivec3, ivec4	**int[n]**	véc-tơ chứa n (2, 3, 4) phần tử kiểu số nguyên có dấu
mat2, mat3, mat4	**float[n*n]**	ma trận n x n (2 x 2, 3 x 3, 4 x 4) số thực
sampler1D, sampler2D, sampler3D	**int**	Truy cập một texture 1 chiều, 2 chiều, 3 chiều
samplerCube	**int**	Truy cập một texture ánh xạ hình khối

GLSL không có con trỏ

Truy cập đến dữ liệu véc-tơ

Chỉ số	0	1	2	3
Truy cập thông qua chỉ số	[0]	[1]	[2]	[3]
Truy cập thông qua tên trục tọa độ	x	y	z	w
Truy cập thông qua tên tọa độ texture	s	t	p	q
Truy cập thông qua tên màu sắc	r	g	b	a

- Ví dụ:
  - Giả sử:

    vec4 v4;

    v4[2] = 8.9;
  - Ta có:

    v4.z bằng **v4.p** bằng **v4.b** và bằng **8.9**

Truy cập đến dữ liệu véc-tơ

- Ví dụ:
  - Giả sử

    vec2 v2;
  - Ta có:

    v2.x là cách truy cập hợp lệ

    ~~**v2.z**~~ là cách truy cập không hợp lệ

Khởi tạo dữ liệu véc-tơ

- Ví dụ:
  - Khởi tạo giá trị ban đầu cho một véc-tơ

    vec4 v4 = vec4(1.0, 2.0, 3.0, 4.0)
- Ví dụ:
  - Giả sử:

    vec2 v2_1 = vec2(1.0, 3.0)

    vec2 v2_2 = vec2(2.0, 4.0)

    vec4 v4 = vec4(v2_1, v2_2)
  - Ta có:

    v4 bằng (1.0, 3.0, 2.0, 4.0)
- Ví dụ:
  - Giả sử: vec4 v4 = vec4(1.0)
  - Ta có: v4 bằng (1.0, 1.0, 1.0, 1.0)

Khởi tạo dữ liệu véc-tơ

- Ví dụ:
  - Giả sử: vec4 v4 = vec4(1.0, 2.0, 3.0, 4.0)
  - Tạo một véc-tơ từ một véc-tơ có sẵn

    vec4 v4_1 = v4.wzyx;

    vec4 v4_2 = v4.xxzz;
  - Ta có:

    v4_1 bằng (4.0, 3.0, 2.0, 1.0)

    v4_2 bằng (1.0, 1.0, 3.0, 3.0)

Khởi tạo dữ liệu véc-tơ

- Ví dụ:
  - Tạo một véc-tơ có số phần tử khác với số phần tử của một véc-tơ có sẵn

    vec3 v3 = v4.wzy;

    vec2 v2 = v4.xy;
  - Ta có:

    v3 bằng (4.0, 3.0, 2.0)

    v2 bằng (1.0, 2.0)

Thay đổi dữ liệu véc-tơ

- Ví dụ:
  - Giả sử

    vec4 v4 = vec4(1.0, 2.0, 3.0, 4.0)

    - v4.xw = vec2(5.0, 6.0)
      - Ta có: v4 bằng (5.0, 2.0, 3.0, 6.0)
    - v4.xx = vec2(7.0, 8.0)
      - Cách viết này không hợp lệ vì x bị lặp 2 lần
    - v4.xy = vec3(9.0, 10.0, 11.0)
      - Cách viết này không hợp lệ vì sự không phù hợp giữa số lượng phần tử trong vec3 và số lượng phần tử trong v4.xy

Khởi tạo dữ liệu ma trận n x n

- Sử dụng một trong các cú pháp sau
  - mat2 m2 = mat2(vec2, vec2);
  - mat3 m3 = mat3(vec3, vec3, vec3);
  - mat4 m4 = mat4(vec4, vec4, vec4, vec4);
  - mat2 m2 = mat2(float, float,
    float, float);
  - mat3 m3 = mat3(float, float, float,
    float, float, float,
    float, float, float);
  - mat4 m4 = mat4(float, float, float, float,
    float, float, float, float,
    float, float, float, float,
    float, float, float, float);

Cấu trúc trong GLSL

- Ví dụ về cấu trúc (structure)
  - Khởi tạo kiểu dữ liệu cấu trúc và định nghĩa biến cấu trúc **myObject**

```
struct DoiTuong {
    vec3 toa_do_dinh;
    float he_so;
} myObject;
```

  - Định nghĩa biến cấu trúc **yourObject**

```
DoiTuong yourObject;
```

Mảng trong GLSL

- Ví dụ về mảng

```
float so_thuc[5];
vec4 myVector[10];

const int number = 8;
DoiTuong myObject[number];
```

Qualifier

- **Qualifier** là một dạng từ khóa thể hiện chức năng của một loại dữ liệu nào đó trong shader.
- Có 3 qualifier chính trong GLSL
  - attribute
  - uniform
  - varying

Attribute Qualifier

- **Attribute (thuộc tính) qualifier** hoạt động như một **liên kết giữa vertex shader** và **WebGL** đối với **dữ liệu mỗi đỉnh**.
- Các biến thuộc dạng attribute có những đặc điểm sau:
  - Nhận giá trị được gửi từ bên ngoài vertex shader.
  - Có thể thay đổi giá trị trong quá trình thực thi vertex shader.
- Ví dụ: Định nghĩa biến có dạng qualifier là attribute bên trong shader

```
in float aDiChuyen;

in vec2 aToaDo;
```

Uniform Qualified

- **Uniform qualifier** hoạt động như một liên kết giữa shader, WebGL và ứng dụng.
- Các biến thuộc dạng uniform có những đặc điểm sau:
  - Nhận giá trị được gửi từ bên ngoài vertex shader.
  - **Không thể thay đổi giá trị** trong quá trình vertex shader đang xử lý.
- Cú pháp và ví dụ
  - **uniform** float uTyLe;

Varying Qualified

- **Varying qualifier** hình thành một liên kết giữa **vertex shader** và **fragment shader**.
- Các biến được **gửi từ vertex shader đến fragment shader** sử dụng 2 từ khóa **out** và **in**.
- Cú pháp và ví dụ
  - Trên vertex shader: **out vec4 vMauSac**;
  - Trên fragment shader: **in vec4 vMauSac**;
  - ***Chú ý: Tên biến dùng out và in buộc phải giống nhau thì mới truyền được giá trị.***

Quy ước đặt tên

- Thuộc tính được gửi từ chương trình WebGL đến vertex shader có tiếp đầu từ là **a**
  - Ví dụ: **aPosition, aColor**
- Biến định nghĩa trong vertex shader có tiếp đầu từ là **v**
  - Ví dụ: **vSize**
- Biến định nghĩa trong fragment shader có tiếp đầu từ là **f**
  - Ví dụ: **fColor**
- Các biến uniform có tiếp đầu từ là **u**
  - Ví dụ: **uScale**

Hoạt động của WebGL

- Truyền dữ liệu vào biến attribute
  - Ví dụ:
    - Truyền dữ liệu đỉnh vào biến attribute có tên là **aToaDo**

```
var vertices = [ vec2(-1, -1), vec2(0, 1), vec2(1, -1) ];

var vBuffer = gl.createBuffer();
gl.bindBuffer(gl.ARRAY_BUFFER, vBuffer);
gl.bufferData(gl.ARRAY_BUFFER, flatten(vertices), gl.STATIC_DRAW);

// flatten là hàm trong file MV.js, được dùng để chuyển đổi các giá trị trong vertices sang số thực

var aPosition = gl.getAttribLocation(program, "aToaDo");
gl.vertexAttribPointer(aPosition, 2, gl.FLOAT, false, 0, 0);
gl.enableVertexAttribArray(aPosition);
```

Hoạt động của WebGL

- Truyền dữ liệu vào biến attribute
  - **gl.createBuffer()**: Khởi tạo một đối tượng WebGLBuffer (bộ đệm WebGL) dùng để lưu trữ các dữ liệu liên quan đến đỉnh hoặc màu sắc.
  - **gl.bindBuffer(target, WebGLBuffer)**: Xác định mục tiêu dùng WebGLBuffer vào mục đích gì
    - **target**:
      - **gl.ARRAY_BUFFER**: Mục tiêu bộ đệm được dùng để **lưu trữ các thuộc tính đỉnh** như **tọa độ đỉnh**, **dữ liệu tọa độ texture** hoặc **dữ liệu màu của đỉnh**.
      - **gl.ELEMENT_ARRAY_BUFFER**: Mục tiêu bộ đệm được dùng để lưu trữ chỉ số của các phần tử.
    - **WebGLBuffer**: Tham chiếu đến bộ đệm mà WebGL sẽ được sử dụng với mục tiêu target.

Hoạt động của WebGL

- Truyền dữ liệu vào biến attribute
  - **gl.bufferData(target, ArrayBufferView srcData, usage)**: Khởi tạo và nạp dữ liệu cho đối tượng buffer
    - **target**: Giống với hàm ***gl.bindBuffer***.
    - **srcData**: Nguồn dữ liệu sẽ được sao chép vào buffer. Nếu để là **null** thì dữ liệu trong buffer không được khởi tạo.
    - **usage**: Xác định loại hình dữ liệu dự kiến sử dụng nhằm mục đích tối ưu việc thao tác trên buffer
      - **gl.STATIC_DRAW**: Dữ liệu dự kiến được xác định chỉ một lần nhưng sử dụng nhiều lần cho các lệnh liên quan đến đặc tả và vẽ hình ảnh.
      - **gl.DYNAMIC_DRAW**: Dữ liệu dự kiến thay đổi nhiều lần và cũng được sử dụng nhiều lần cho các lệnh liên quan đến đặc tả và vẽ hình ảnh.
      - **gl.STREAM_DRAW**: Dữ liệu dự kiến được xác định một lần nhưng chỉ được sử dụng vài lần cho các lệnh liên quan đến đặc tả và vẽ hình ảnh.
    - ***Chú ý: Bản chất của hàm này là copy dữ liệu từ bộ nhớ của ứng dụng vào bộ nhớ của GPU để sử dụng cho shader.***

Hoạt động của WebGL

- Truyền dữ liệu vào biến attribute
  - **gl.getAttribLocation(program, variableName)**: Trả về một tham chiếu đến biến attribute bên trong chương trình WebGLProgram.
    - **program**: Chương trình WebGLProgram mà chứa biến attribute.
    - **variableName**: Tên của biến attribute cần tham chiếu đến.
    - Trường hợp không tìm thấy biến attribute, hàm sẽ trả về giá trị -1.

Hoạt động của WebGL

- Truyền dữ liệu vào biến attribute
  - **gl.vertexAttribPointer(index, size, type, normalized, stride, offset)**: liên kết bộ đệm (kho dữ liệu đỉnh gl.ARRAY_BUFFER) với biến attribute chứa đỉnh ở bên trong vertex shader.
    - **index**: tham chiếu đến biến attribute mà sẽ nhận dữ liệu đỉnh từ buffer.
    - **size**: Xác định số lượng phần tử của tham chiếu đến biến attribute. Chỉ có thể là 1, 2, 3, hoặc 4.
    - **type**: Xác định kiểu dữ liệu của mỗi phần tử trong buffer
      - **gl.BYTE** hoặc **gl.SHORT** hoặc **gl.UNSIGNED_BYTE** hoặc **gl.UNSIGNED_SHORT** hoặc **gl.FLOAT** hoặc **gl.HALF_FLOAT**
    - **normalized:** Xác định xem liệu các giá trị có kiểu dữ liệu số nguyên có nên được chuẩn hóa vào một phạm vi nào đó khi bị ép kiểu sang số thực hay không
      - Đối với kiểu dữ liệu **gl.BYTE** và **gl.SHORT**, chuẩn hóa giá trị vào phạm vi [-1, 1] **nếu normalized được thiết lập là true**.
      - Đối với kiểu dữ liệu gl.UNSIGNED_BYTE và gl.UNSIGNED_SHORT, chuẩn hóa giá trị vào phạm vi [0, 1] **nếu normalized được thiết lập là true**.
      - Đối với kiểu dữ liệu gl.FLOAT và gl.HALF_FLOAT, tham số này không có ý nghĩa.
    - **stride**: Xác định độ lệch (theo bytes) tính từ điểm bắt đầu của các thuộc tính đỉnh. Giá trị này không được phép lớn hơn 255. Nếu stride = 0 tức là đỉnh tiếp theo nằm ngay sau đỉnh hiện tại.
    - **offset**: Xác định độ lệch (theo bytes) của phần tử đầu tiên trong kho thuộc tính đỉnh. Giá trị này phải là bội số (tính theo byte) đối với độ lớn của kiểu dữ liệu.
    - ***Nếu không phải xử lý đặc biệt đối với đỉnh, ta luôn đặt hai tham số stride và offset đều = 0.***

Hoạt động của WebGL

- Truyền dữ liệu vào biến attribute
  - **gl.enableVertexAttribArray(index)**: Cho phép sử dụng dữ liệu từ buffer chứa đỉnh liên kết với tham chiếu đến biến attribute xác định bởi index.
    - **Index**: Tham chiếu đến biến có dạng attribute mà nhận dữ liệu từ buffer chứa đỉnh.
  - **gl.disableVertexAttribArray(index)**: Không cho phép sử dụng dữ liệu từ buffer liên kết với tham chiếu biến attribute xác định bởi index.

Ví dụ 1

- Vẽ một hình tam giác màu đỏ
  - Phân tích từng câu lệnh đã trình bày ở trên.
  - Phương pháp xác định màu trong ví dụ 1 là **xác định màu trực tiếp**.
  - Phân biệt giữa kiểu dữ liệu **vecX**, **matX** (tự định nghĩa trong file MV.js) trong file .js và **vecX**, **matX** trong shaders.
- Các hàm bổ trợ
  - **gl.clearColor(red, green, blue, alpha)**: Hàm này dùng để xác định màu được dùng để xóa màn hình. Giá trị red, green, blue nằm trong khoảng [0, 1]
  - **gl.clear(mask)**: Xóa bộ đệm với giá trị cho trước.
    - **mask**:
      - **gl.COLOR_BUFFER_BIT**: Xóa màn hình với màu được thiết lập bởi hàm **gl.clearColor()**
      - **gl.DEPTH_BUFFER_BIT**: Xóa màn hình với giá trị được thiết lập bởi hàm **gl.clearDepth()** (Sử dụng khi muốn vẽ hình 3D)

Ví dụ 1

- Vẽ một hình tam giác màu đỏ
- Các hàm bổ trợ
  - **gl.drawArrays(mode, first, count)**: Vẽ đối tượng theo cách được lựa chọn trước
    - **mode**: Chọn một trong các cách vẽ sau:
      - **gl.POINTS**
      - **gl.LINE_STRIP**
      - **gl.LINE_LOOP**
      - **gl.LINES**
      - **gl.TRIANGLE_STRIP**
      - **gl.TRIANGLE_FAN**
      - **gl.TRIANGLES**
    - **first**: Chỉ số bắt đầu của phần tử trong mảng kho dữ liệu được dùng để vẽ.
    - **count**: Số lượng chỉ số được dùng để vẽ.

Ví dụ 2

- Vẽ hình tam giác với màu 3 đỉnh được truyền vào từ chương trình.
  - Thử nghiệm với 3 đỉnh có màu giống nhau.
  - Thử nghiệm với 3 đỉnh có màu khác nhau (quan sát nội suy).
  - ***Chú ý:***
    - Chương trình sử dụng 2 mảng dữ liệu, 1 mảng chứa dữ liệu đỉnh, 1 mảng chứa dữ liệu màu.
      - Vertex shader sẽ lấy **cùng một lúc** dữ liệu của từng phần tử trong 2 mảng để xử lý
    - Phương pháp xác định màu trong ví dụ 2 là **xác định màu gián tiếp** (truyền từ bên ngoài).
    - Cách truyền dữ liệu từ vertex shader sang fragment shader (dạng varying qualifier)

Hoạt động của WebGL

- Truyền dữ liệu vào biến uniform
  - Ví dụ:
    - Truyền dữ liệu vào biến số thực uniform có tên là **uToaDo**

```
var vec2 = vec2(0.5, 0.5);

var uToaDo = gl.getUniformLocation(program, "uToaDo");

gl.uniform2f(uToaDo, vec2);
```

Hoạt động của WebGL

- Truyền dữ liệu vào biến uniform
  - **gl.getUniformLocation(program, variableName)**: Trả về tham chiếu đến biến uniform bên trong chương trình WebGLProgram.
    - **program**: Chương trình WebGLProgram mà chứa biến uniform.
    - **variableName**: Tên của biến uniform cần tham chiếu đến.

Hoạt động của WebGL

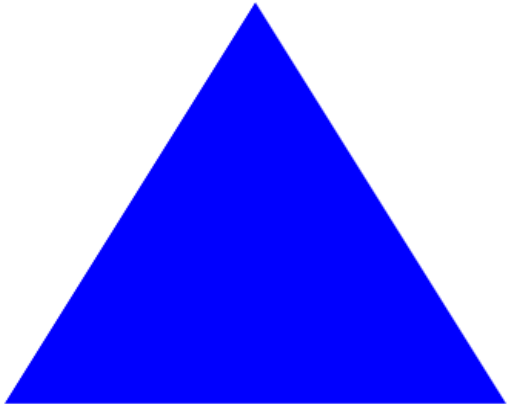
- Truyền dữ liệu vào biến uniform: Tùy thuộc vào dữ liệu cần truyền vào mà ta lựa chọn cú pháp cho phù hợp

– **gl.uniform[1234][fi][v]()**: Đặt giá trị cho biến uniform. (f: float; i: integer; v: vector)

- void gl.uniform1f(location, v0);
- void gl.uniform1fv(location, value); // value phải là một mảng
- void gl.uniform1i(location, v0);
- void gl.uniform1iv(location, value);
-
- void gl.uniform2f(location, v0, v1);
- void gl.uniform2fv(location, value);
- void gl.uniform2i(location, v0, v1);
- void gl.uniform2iv(location, value); // value là một vec2
-
- void gl.uniform3f(location, v0, v1, v2);
- void gl.uniform3fv(location, value); // value là một vec3
- void gl.uniform3i(location, v0, v1, v2);
- void gl.uniform3iv(location, value);
-
- void gl.uniform4f(location, v0, v1, v2, v3);
- void gl.uniform4fv(location, value); // value là một vec4
- void gl.uniform4i(location, v0, v1, v2, v3);
- void gl.uniform4iv(location, value);

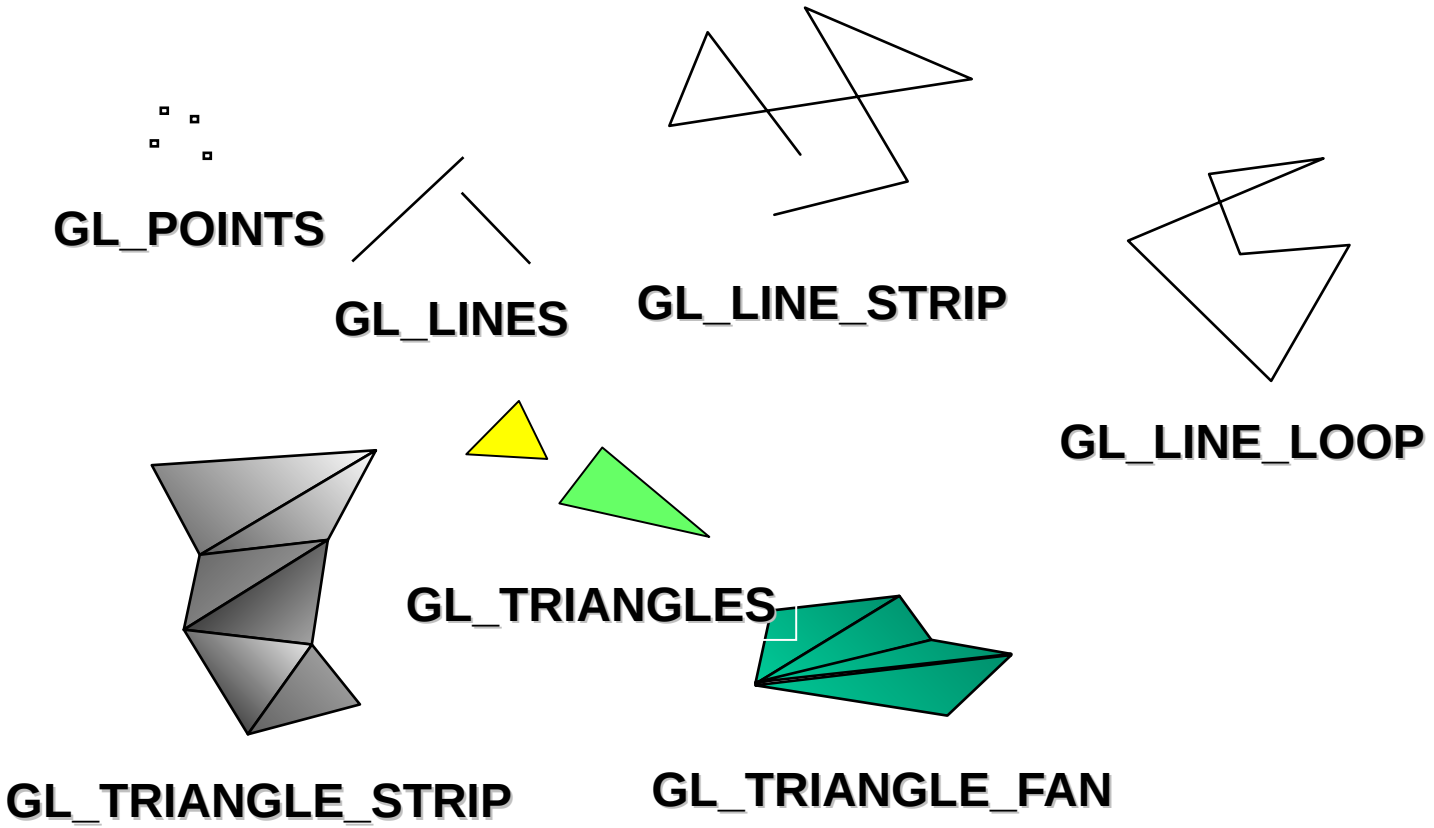
Ví dụ 3

- Vẽ tam giác phóng to, thu nhỏ theo tỉ lệ được truyền vào từ chương trình.
  - Giá trị tọa độ được tính toán trước khi truyền vào cho biến **gl_Position**



Vẽ các đối tượng cơ bản

- WebGL chỉ vẽ điểm, đường và tam giác.



Ví dụ 4

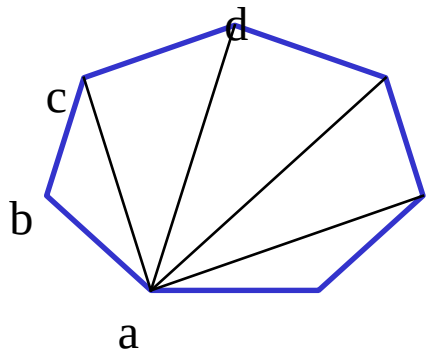
- Sử dụng WebGL vẽ điểm (**gl.POINTS**)
  - Chú ý sử dụng biến **gl_PointSize** trong Vertex Shader để đặt kích thước điểm.
- Sử dụng WebGL vẽ đường thẳng (**gl.LINES**)
  - Chú ý sử dụng hàm gl.lineWidth() để đặt kích thước đường.
  - Số đỉnh đưa vào nếu không chia hết cho 2 thì đỉnh dư sẽ bị cắt bỏ.
- Sử dụng WebGL vẽ nối các điểm nhưng không nối về điểm đầu (**gl.LINE_STRIP**)
- Sử dụng WebGL vẽ nối các điểm và có nối về điểm đầu (**gl.LINE_LOOP**)
- Sử dụng WebGL vẽ các tam giác (**gl.TRIANGLES**)
  - Số đỉnh đưa vào nếu không chia hết cho 3 thì các đỉnh dư sẽ bị cắt bỏ.

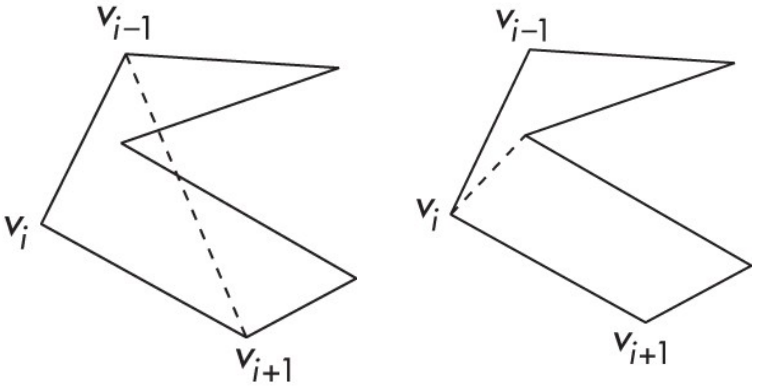
Ví dụ 5

- Sử dụng WebGL vẽ dải tam giác kề nhau (gl.TRIANGLE_STRIP)
  - Chú ý thứ tự các đỉnh để vẽ tam giác
    - (1, 2, 3) – (2, 3, 4) – (3, 4, 5) …..
- Sử dụng WebGL vẽ các tam giác hình quạt (gl.TRIANGLE_FAN)
  - Chú ý thứ tự các đỉnh để vẽ tam giác
    - (1, 2, 3) – (1, 3, 4) – (1, 4, 5) …..

Vấn đề về đa giác

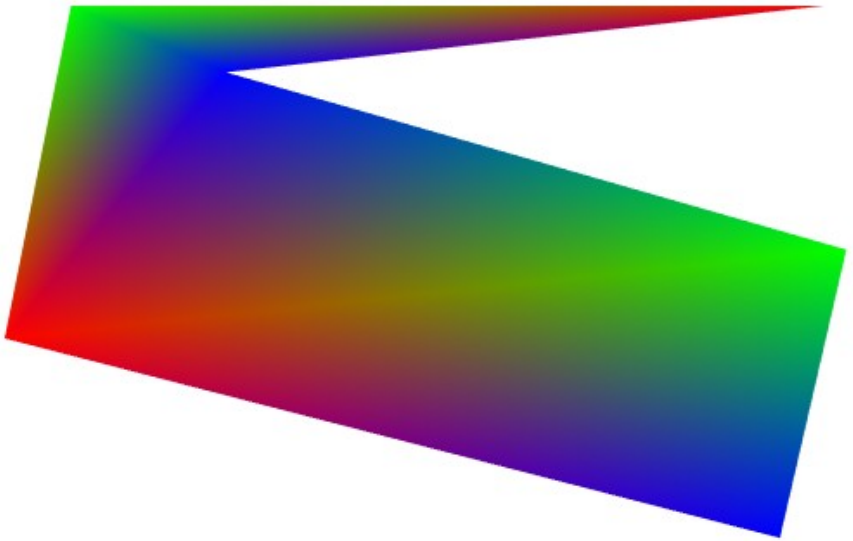
- Để vẽ một đa giác trong WebGL ta buộc phải **chia đa giác đó thành các tam giác nhỏ hơn**.
- Việc phân chia này sẽ làm đơn giản hóa đa giác. Tuy nhiên ta cần phải có một **thuật toán** để làm công việc này.
- Với **đa giác lồi** thì việc vẽ tam giác sẽ dễ dàng hơn **đa giác lõm**.
- Với đa giác lõm, ta cần tìm cách chia tam giác sao cho các tam giác khi vẽ **không bị chồng lấn**.





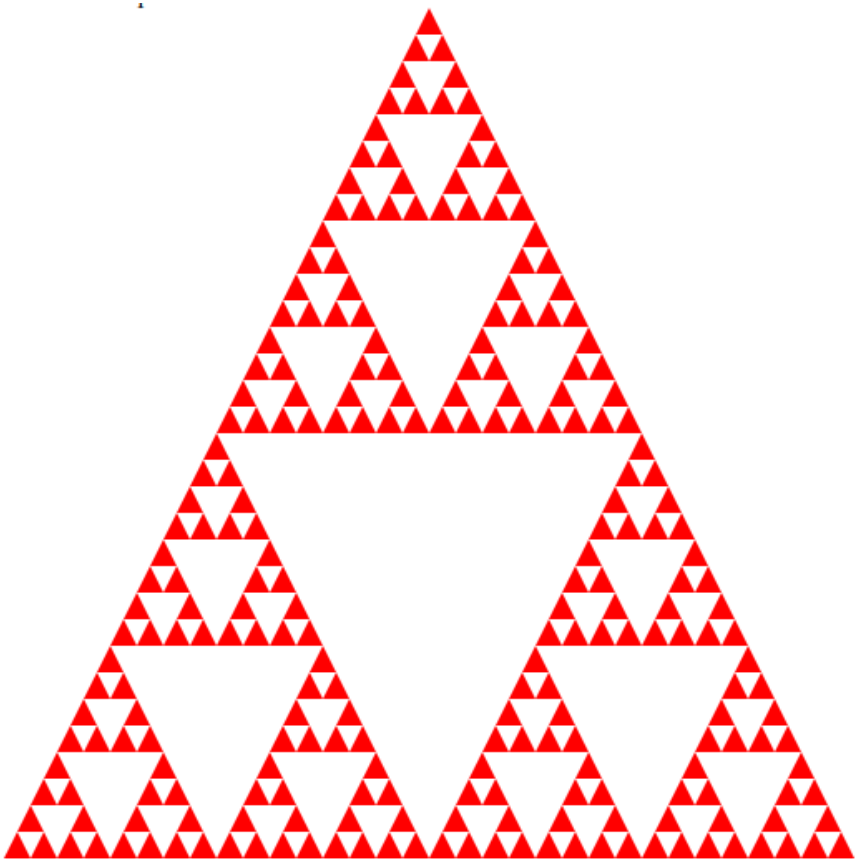
Ví dụ

- **Ví dụ 6**: Vẽ một đa giác lồi
- **Ví dụ 7**: Vẽ một đa giác lõm



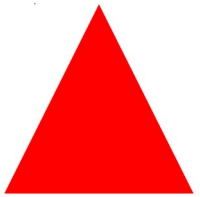
Ví dụ 8

- Vẽ tam giác 2D Sierpinski

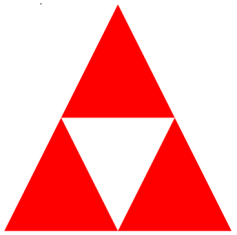


Ví dụ 8

- Vẽ tam giác Sierpinski
  - Tính toán tọa độ đỉnh của các tam giác nhỏ, sau đó nạp dần dần cứ 3 đỉnh một lần tương ứng với 3 đỉnh của tam giác nhỏ vào mảng dữ liệu đỉnh.
- Ý tưởng thuật toán
  - Bắt đầu với một tam giác



  - Nối trung điểm của các cạnh rồi loại bỏ tam giác ở trung tâm.



  - Cứ lặp lại như vậy đối với các tam giác ở bên trong cho đến n lần thì dừng.

Ví dụ 8

- Vẽ tam giác 2D Sierpinski
  - Thuật toán (xây dựng hàm đệ quy)

chiaTamGiac(đỉnh A, đỉnh B, đỉnh C, số lần lặp) {

Nếu (số lần lặp = 0) **Thì**

veTamGiac(đỉnh A, đỉnh B, đỉnh C);

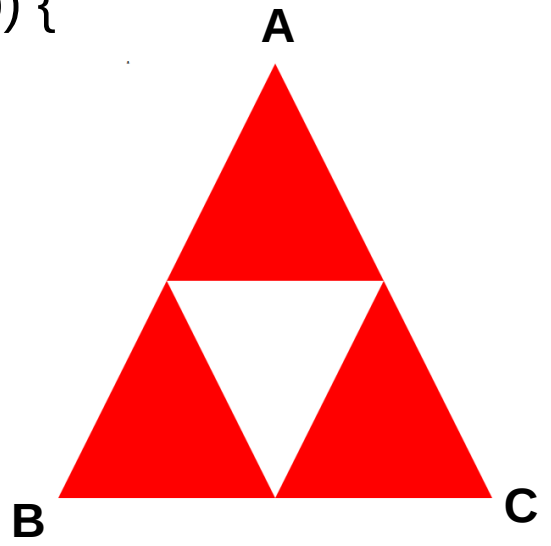
Ngược lại

trung điểm AB = tinhTrungDiem(đỉnh A, đỉnh B);
trung điểm BC = tinhTrungDiem(đỉnh B, đỉnh C);
trung điểm AC = tinhTrungDiem(đỉnh A, đỉnh C);

số lần lặp = số lần lặp – 1;

chiaTamGiac(đỉnh A, trung điểm AB, trung điểm AC, số lần lặp);
chiaTamGiac(đỉnh B, trung điểm AB, trung điểm BC, số lần lặp);
chiaTamGiac(đỉnh C, trung điểm AC, trung điểm BC, số lần lặp);

}



Ví dụ 8

- Vẽ tam giác 2D Sierpinski
  - Khởi tạo một **mảng V** được dùng để chứa đỉnh của tất cả các tam giác.
    - Cứ mỗi khi gặp số lần lặp = 0 thì hàm **veTamGiac** sẽ đưa bộ đỉnh tạo lập tam giác cuối cùng cần vẽ vào mảng V này.
  - Sau khi hàm đệ quy **chiaTamGiac** thực hiện xong
    - Vẽ các tam giác dựa trên mảng V.

Ví dụ 8

- Hàm **mix()** trong thư viện MV.js
  - **Cú pháp**: **mix(u, v, s)** dùng để lấy tọa độ của điểm nằm trên đoạn thẳng nối u với v với **tỉ lệ s**.
    - **u, v**: Tọa độ của 2 điểm trên không gian 2 hoặc 3 chiều. Số chiều của u và v bắt buộc phải bằng nhau.
    - Công thức:

$$\mathbf{m_i = (1 - s) * u_i + s*v_i}$$

$\mathbf{m_i}$ là tọa độ của điểm cần xác định

Hết Tuần 2

Cảm ơn các bạn đã chú ý lắng nghe !!!