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
- WebGL Primitives
- Các ví dụ minh họa

Shaders

- **Shaders** là các chương trình chạy bên trong GPU để xử lý một nhiệm vụ nào đó.
- Chương trình shader sử dụng **ngôn ngữ tựa C** được gọi là **OpenGL Shading Language (GLSL).**
- Có 2 shaders
  - Vertex shader
  - Fragment shader

Vertex Shader

- Vertex shader là một đoạn mã chương trình được gọi trên mỗi đỉnh.
- Nó xử lý dữ liệu của mỗi đỉnh như xác định tọa độ đỉnh, chuẩn hóa, xử lý màu, …
- Ví dụ:

```
<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

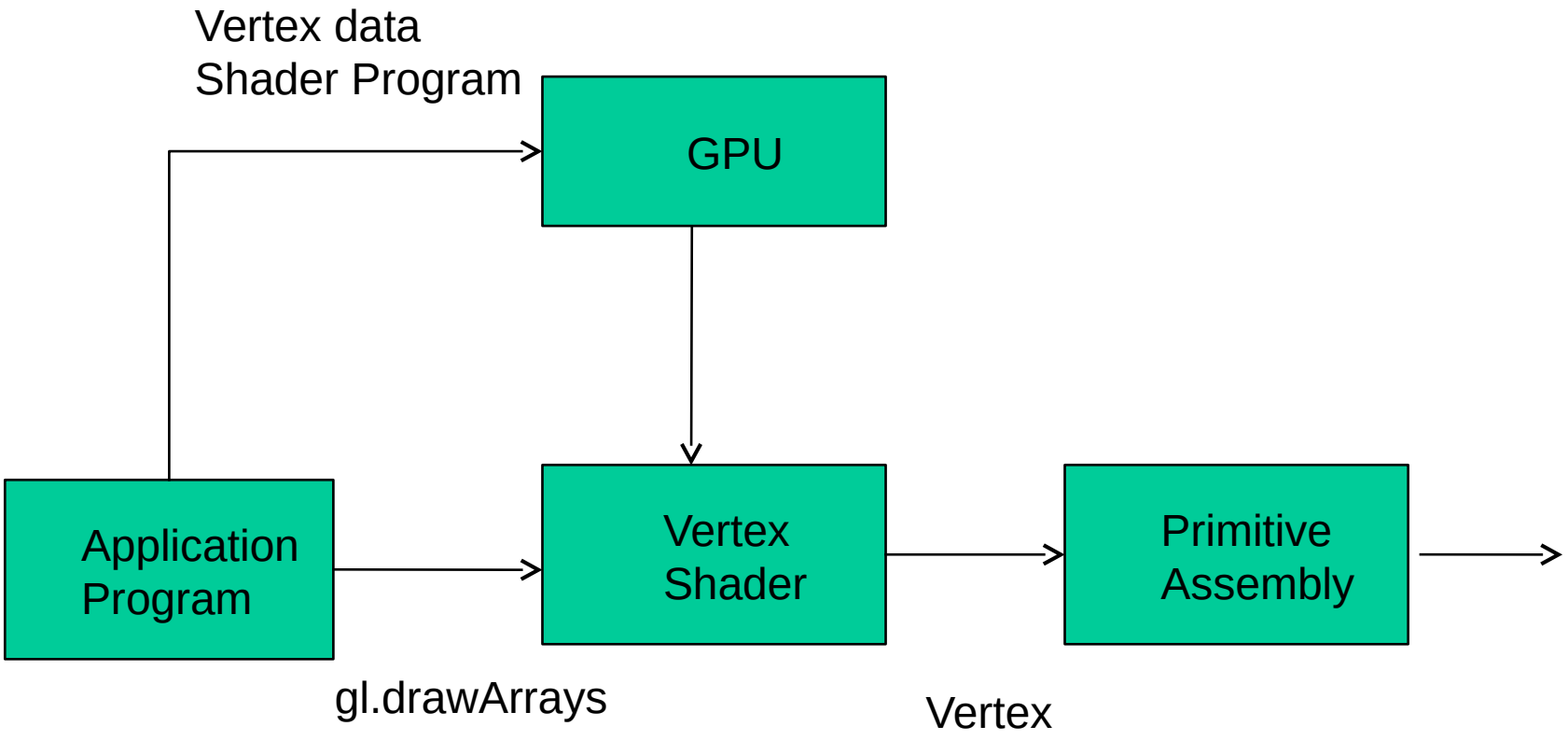
- **gl_Position**
  - là một biến được định nghĩa sẵn bên trong chương trình vertex shader **dùng để chứa vị trí đỉnh**.
  - Sau ghi được truyền giá trị, giá trị trong gl_Position sẽ được thực hiện bởi primitive assembly, clipping,... (các công việc trong đường ống đồ họa)

Fragment Shader

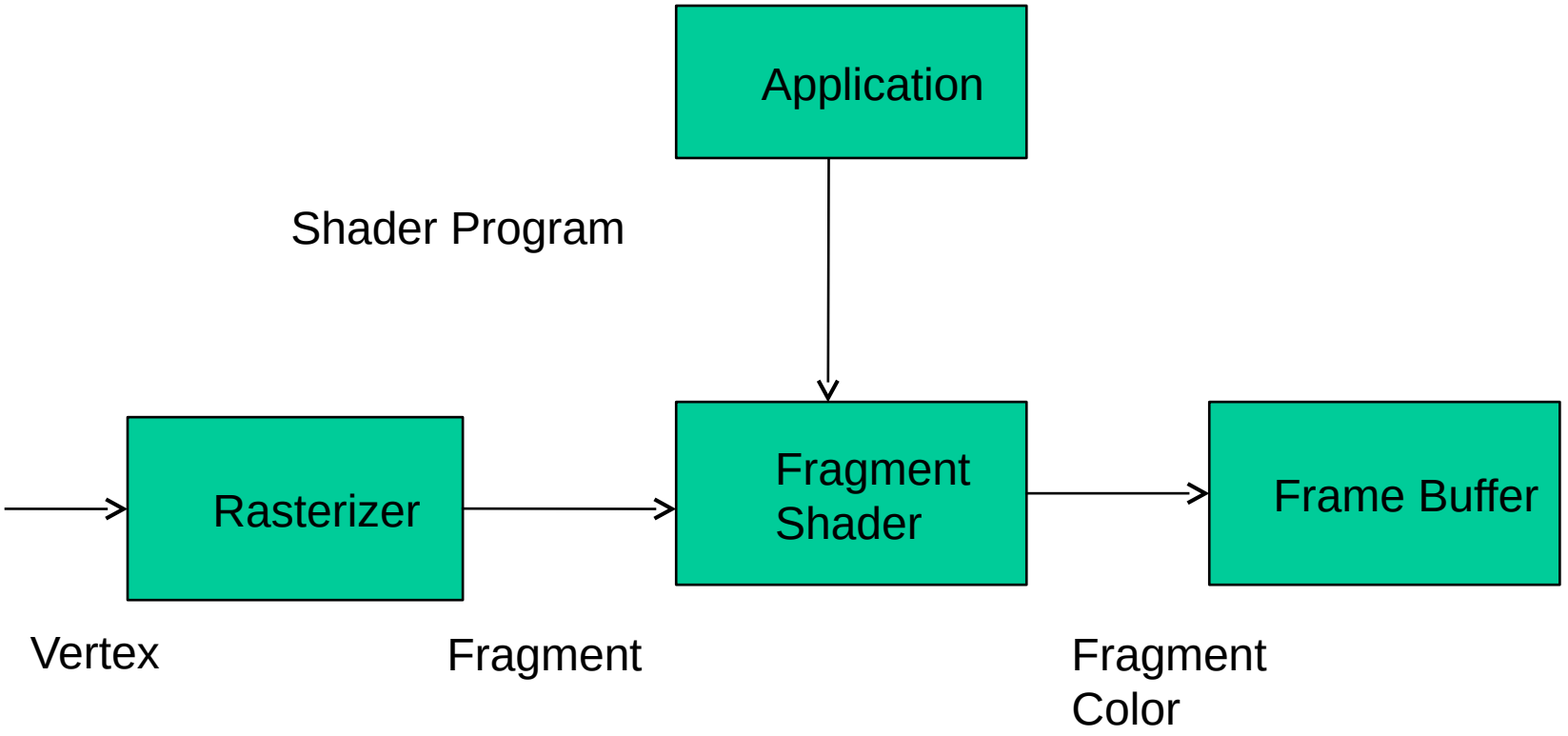
- Một lưới được tạo bởi các hình tam giác và bề mặt của mỗi hình tam giác được gọi là một **fragment**.
- Fragment shader là một mã chương trình được chạy trên mỗi pixel của mỗi fragment. Nó có nhiệm vụ tính toán và xác định màu sắc của mỗi pixel.
- Ví dụ:

```
<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>
```

Mô hình thực thi



Mô hình thực thi



Các kiểu dữ liệu của GLSL

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

GLSL không có con trỏ

Qualifier

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

Attribute Qualifier

- Qualifier này hoạt động như một **liên kết giữa vertex shader** và **OpenGL ES** đối với **dữ liệu mỗi đỉnh**.
- Giá trị của thuộc tính này thay đổi mỗi khi vertex shader thực thi.
- Cú pháp và ví dụ
  - **in** float aDiChuyen
  - **in** vec2 aToaDo

Uniform Qualified

- Qualifier này hoạt động như một liên kết giữa các chương trình shader với ứng dụng WebGL.
- Qualifier này không thể thay đổi giá trị bên trong shader.
- Thường được dùng để gửi thông tin từ ứng dụng đến shader.
- Cú pháp và ví dụ
  - **uniform** float uTyLe;

Ví dụ 1

- Vẽ tam giác phóng to, thu nhỏ theo tỉ lệ được truyền vào từ chương trình.

Varying Qualified

- Qualifier này hình thành một liên kết giữa **vertex shader** và **fragment shader** để dữ liệu nội suy.
- Các biến được gửi từ vertex shader đến fragment shader.
- Cú pháp và ví dụ
  - Trên vertex shader: **out vec4 vMauSac**;
  - Trên fragment shader: **in vec4 vMauSac**;

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

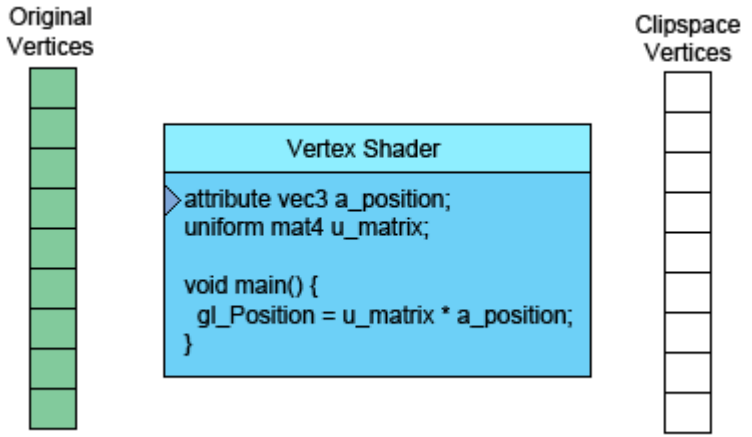
- Hoạt động của Vertex Shader
  - Giả sử ta muốn vẽ 3 tam giác, mỗi tam giác có 3 đỉnh. Ta thực hiện câu lệnh sau:

gl.drawArrays(gl.TRIANGLES, 0, 9);

    - Tham số 1: Dạng hình muốn vẽ
    - Tham số 2: Vị trí của dữ liệu bắt đầu được xử lý bên trong bộ nhớ đệm.
    - Tham số 3: Số lượng đỉnh muốn xử lý

Hoạt động của WebGL

- Hoạt động của Vertex Shader
  - Cột bên trái là dữ liệu 9 đỉnh mà ta đưa vào bởi lệnh **gl.drawArrays**
  - **Vertex Shader là một hàm** và được thực hiện lặp đi lặp lại một lần cho mỗi 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.



Hoạt động của WebGL

- Hoạt động của Fragment Shader
  - Giả sử ta vẽ một hình tam giác, điều này có nghĩa là ta sẽ có 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 cũng là một hàm** và với mỗi pixel nằm trong fragment, hệ thống sẽ gọi đến hàm này để lấy màu của pixel.

Hoạt động của WebGL

- Hoạt động của Fragment Shader
  - 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**.

Hoạt động của WebGL

- Đặt màu cho các đỉnh
  - Cách 1: Đặt trực tiếp màu tại Fragment Shader

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

    out vec4 fColor;

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

Hoạt động của WebGL

- Đặt màu cho các đỉnh
  - Cách 2:
    - **Thêm một thuộc tính đến Vertex Shader** để có thể nhận dữ liệu từ chương trình.
    - Sau đó dữ liệu của thuộc tính này sẽ được **truyền đến Fragment Shader**.

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

    in vec2 aPosition;
    in vec3 aColor;

    out vec4 vColor;

    void main()
    {
        gl_Position = vec4(aPosition, 0.0, 1.0);
        vColor = vec4(aColor, 1.0);
    }
</script>

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

    in vec4 vColor;
    out vec4 fColor;

    void main()
    {
        fColor = vColor;
    }
</script>
```

Hoạt động của WebGL

- Thiết lập thuộc tính màu
  - Sử dụng hàm với các tham số kèm theo:
    - gl.vertexAttribPointer(colorLocation, size, type, nomalize, stride, offset)
      - **colorLocation**: Trỏ đến thuộc tính trong Vertex Shader.
      - **size**: số lượng phần tử trong thuộc tính.
      - **type**: Kiểu dữ liệu của mỗi phần tử trong bộ đệm.
      - **normalize**: Dùng để chuẩn hóa dữ liệu với ***type có kiểu số nguyên***, nếu ***type là số thực*** thì tham số này không có ý nghĩa.
      - **stride**: Nếu = 0 thì sau mỗi lần lặp, vị trí tiếp theo trong bộ đệm sẽ là **size * sizeof(type)**
      - **offset**: Điểm bắt đầu trong bộ đệm
    - Ví dụ:

```
var aColor = gl.getAttribLocation(program, "aColor");
gl.vertexAttribPointer(aColor, 3, gl.FLOAT, false, 0, 0);
```

Hoạt động của WebGL

- Thiết lập thuộc tính màu
  - Sử dụng hàm với các tham số kèm theo:
    - Hàm **gl.enableVertexAttribArray** dùng để cho phép sử dụng thuộc tính trong Vertex Shader.
    - Ví dụ:

```
gl.enableVertexAttribArray(aColor);
```

Hoạt động của WebGL

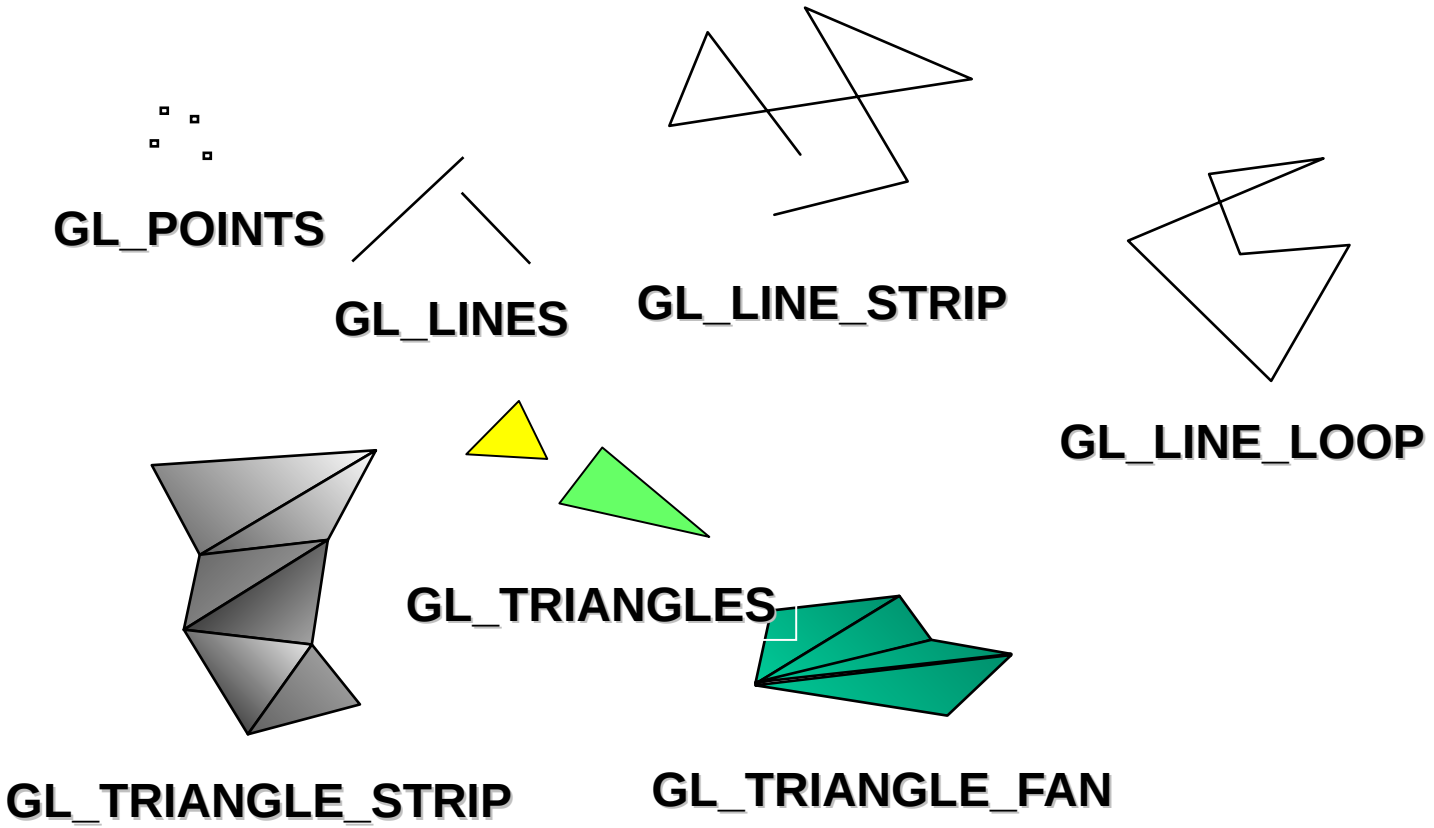
- Các hàm làm việc với Buffer (Bộ đệm)
  - gl.createBuffer() : Tạo ra một buffer mới.
  - gl.bindBuffer: Thiết lập buffer vừa tạo như buffer đang trong xử lý.
  - gl.bufferData: Dùng để copy dữ liệu vào buffer.

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).

WebGL Primitives

- WebGL chỉ vẽ điểm, đường và tam giác.



Ví dụ 3

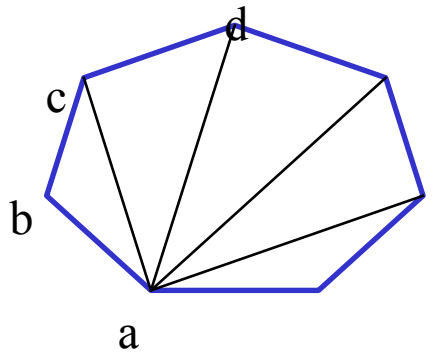
- Sử dụng WebGL vẽ điểm (gl.POINTS)
  - Chú ý sử dụng biến gl_PointSize trong Vertex Shader để đặt kích thước điểm.
- Sử dụng WebGL vẽ đường thẳng (gl.LINES)
  - Chú ý sử dụng hàm gl.lineWidth() để đặt kích thước đường.
  - Số đỉnh đưa vào nếu không chia hết cho 2 thì đỉnh dư sẽ bị cắt bỏ.
- Sử dụng 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.

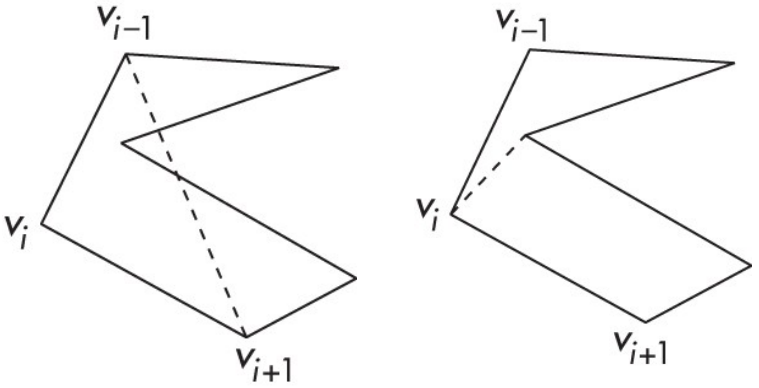
Ví dụ 4

- 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

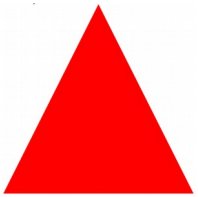
- Để vẽ một đa giác trong WebGL ta buộc phải **chia đa giác đó thành các tam giác nhỏ hơn**.
- Việc phân chia này sẽ làm đơn giản hóa đa giác. Tuy nhiên ta cần phải có một **thuật toán** để làm công việc này.
- Với **đa giác lồi** thì việc vẽ tam giác sẽ dễ dàng hơn **đa giác lõm**.
- Với đa giác lõm,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



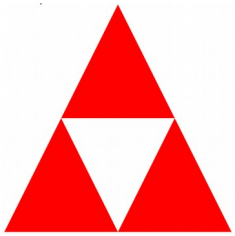


Ví dụ 5

- Vẽ tam giác Sierpinski
  - Bắt đầu với một tam giác



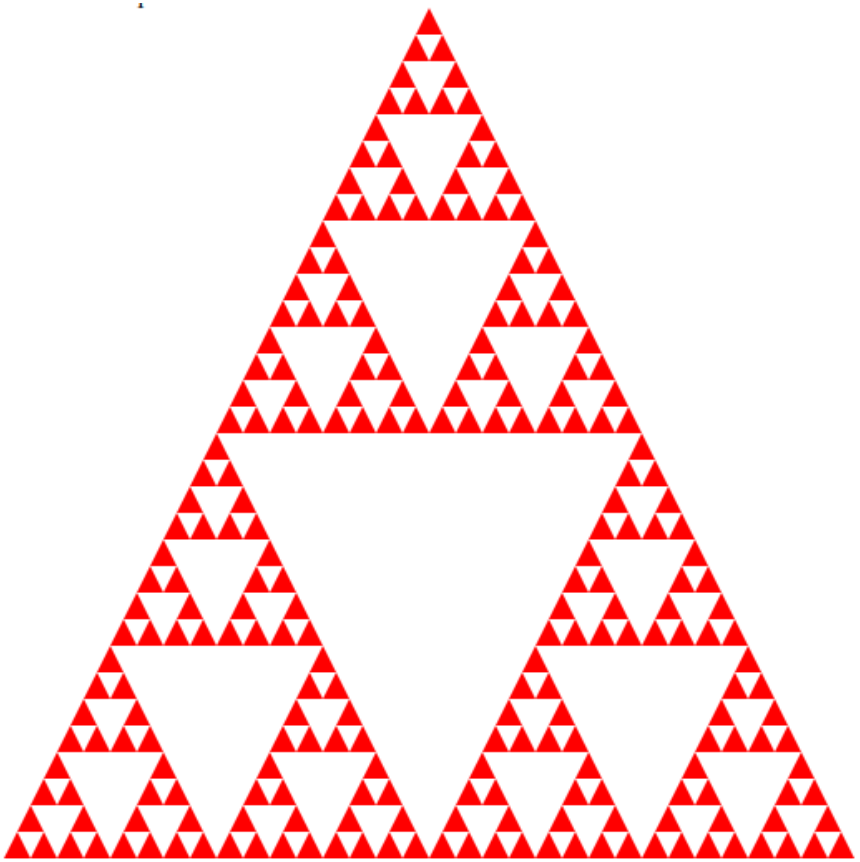
  - Nối trung điểm của các cạnh rồi loại bỏ tam giác ở trung tâm.



  - Cứ lặp lại như vậy cho đến n lần

Ví dụ 5

- Vẽ tam giác 2D Sierpinski



Ví dụ 5

- 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);

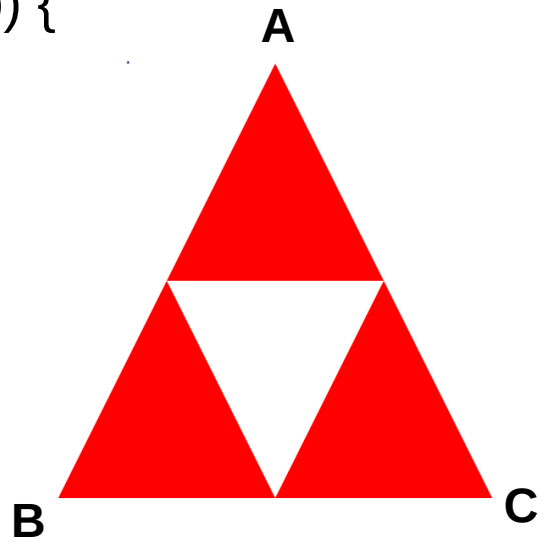
Ngược lại

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);

}

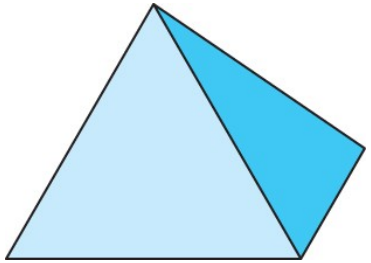


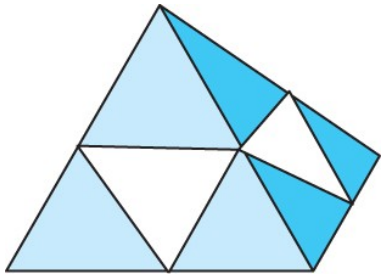
Ví dụ 5

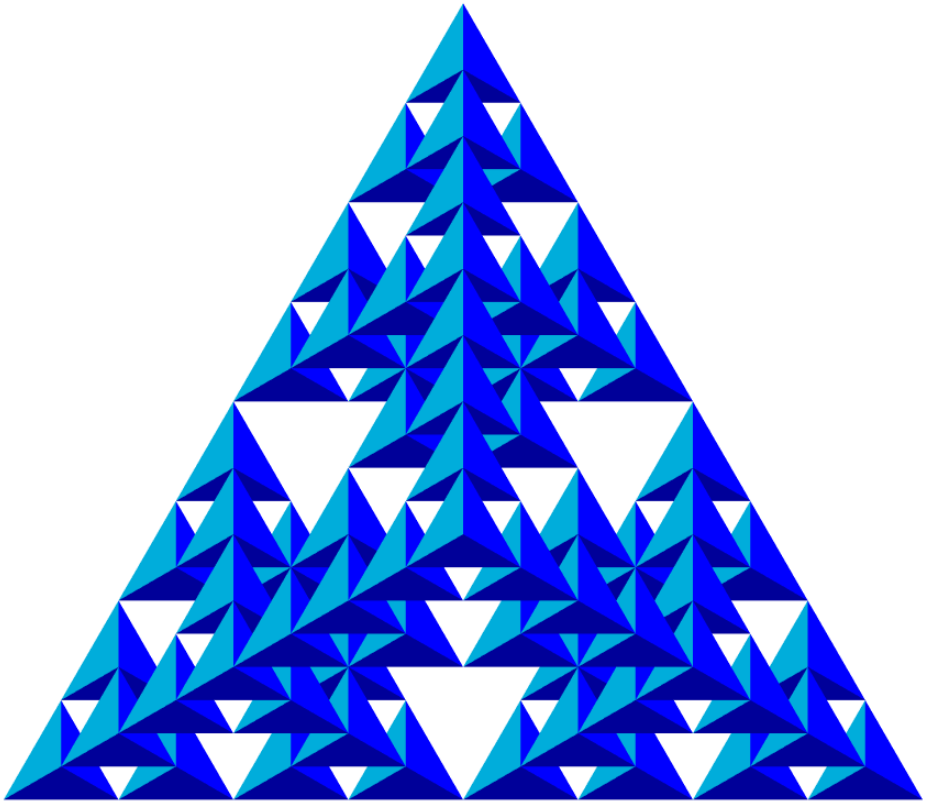
- Vẽ tam giác 2D Sierpinski
  - Khởi tạo một **mảng V** được dùng để chứa đỉnh của tất cả các tam giác.
    - Cứ mỗi khi gặp số lần lặp = 0 thì hàm **veTamGiac** sẽ đưa bộ đỉnh tạo lập tam giác cuối cùng cần vẽ vào mảng V này.
  - Sau khi hàm đệ quy **chiaTamGiac** thực hiện xong
    - Thực hiện đưa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

Ví dụ 6

- Vẽ hình chóp Sierpinski







Ví dụ 6

- Vẽ hình chóp Sierpinski
  - Thuật toán (xây dựng hàm đệ quy)

chiaHinhChop(đỉnh A, đỉnh B, đỉnh C, đỉnh D, số lần lặp) {

Nếu (số lần lặp = 0) **Thì**

veHinhChop(đỉnh A, đỉnh B, đỉnh C, đỉnh D);

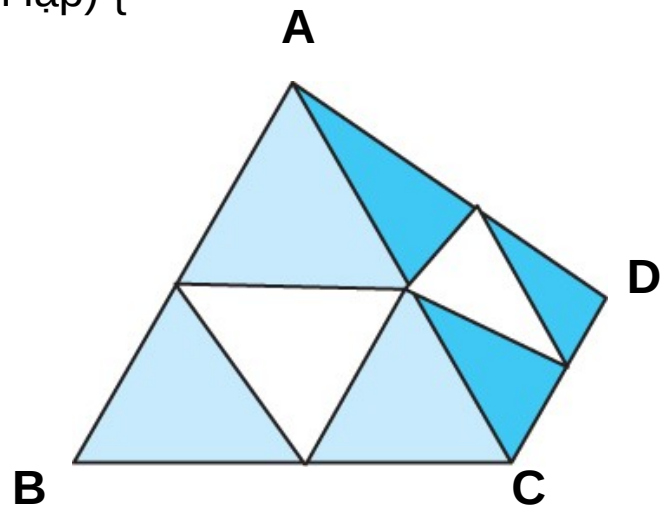
Ngược lại

trung điểm AB = tinhTrungDiem(đỉnh A, đỉnh B);
trung điểm AC = tinhTrungDiem(đỉnh A, đỉnh C);
trung điểm AD = tinhTrungDiem(đỉnh A, đỉnh D);
trung điểm BC = tinhTrungDiem(đỉnh B, đỉnh C);
trung điểm BD = tinhTrungDiem(đỉnh B, đỉnh D);
trung điểm CD = tinhTrungDiem(đỉnh C, đỉnh D);

số lần lặp = số lần lặp – 1;

chiaHinhChop(đỉnh A, trung điểm AB, trung điểm AC, trung điểm AD, số lần lặp);
chiaHinhChop(đỉnh B, trung điểm AB, trung điểm BC, trung điểm BD, số lần lặp);
chiaHinhChop(đỉnh C, trung điểm AC, trung điểm BC, trung điểm CD, số lần lặp);
chiaHinhChop(đỉnh D, trung điểm AD, trung điểm BD, trung điểm CD, số lần lặp);

}



Ví dụ 6

- Vẽ hình chóp Sierpinski
  - Thuật toán

```
veHinhChop(đỉnh A, đỉnh B, đỉnh C, đỉnh D) {
    veTamGiac(đỉnh A, đỉnh B, đỉnh C);
    veTamGiac(đỉnh A, đỉnh B, đỉnh D);
    veTamGiac(đỉnh A, đỉnh C, đỉnh D);
    veTamGiac(đỉnh B, đỉnh C, đỉnh D);
}
```

Ví dụ 6

- Vẽ hình chóp Sierpinski
  - Khởi tạo một **mảng V** được dùng để chứa đỉnh của tất cả các hình chóp.
    - Cứ mỗi khi gặp số lần lặp = 0 thì hàm **veHinhChop** sẽ gọi đến các hàm **veTamGiac** để đưa toàn bộ các đỉnh vẽ tam giác ở bên trong hình chóp được tạo lập cuối cùng vào mảng V này.
  - Sau khi hàm đệ quy **chiaHinhChop** thực hiện xong
    - Thực hiện đưa toàn bộ các đỉnh cần vẽ vào Vertex Shader để bắt đầu thực hiện vẽ các tam giác lên màn hình.

Hết Tuần 2

Cảm ơn các bạn đã chú ý lắng nghe !!!