Web GL Fundamentals – Manipulation and Image Processing
IF3260 Computer Graphics

Oleh

NIM : 13519183
Nama : Afifah Fathimah Qur'ani
Kelas : K-04

PROGRAM STUDI TEKNIK INFORMATIKA
SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA
INSTITUT TEKNOLOGI BANDUNG

2021

Project I – How It Works

Pada bagian awal project dijelaskan mengenai bagian dari GPU. Bagian pertama melakukan pemrosesan terhadap vertex dan memasukkannya ke clip space yang disediakan, dan bagian kedua menggambar pixel berdasarkan hasil pemrosesan bagian pertama. Bagian pertama tersebut dilakukan oleh fungsi vertex shader yang akan dipanggil untuk setiap vertex.

```
<!-- vertex shader -->
<script  id="vertex-shader-2d" type="x-shader/x-vertex">
attribute vec2 a_position;
attribute vec4 a_color;

uniform mat3 u_matrix;

varying vec4 v_color;

void main() {
  // Multiply the position by the matrix.
  gl_Position = vec4((u_matrix * vec3(a_position, 1)).xy, 0, 1);

  // Copy the color from the attribute to the varying.
  v_color = a_color;
}
</script>
```

Setiap terbentuk 3 vertex, GPU akan membentuk segitiga dan menghasilkan pixel yang berkorespondensi dengan 3 vertex tersebut. Untuk setiap pixel akan dipanggil fungsi fragment shader untuk menentukan warna pada pixel tersebut.

```
<!-- fragment shader -->
<script  id="fragment-shader-2d" type="x-shader/x-fragment">
precision mediump float;

varying vec4 v_color;

void main() {
  gl_FragColor = v_color;
}
</script>
```

Fungsi fragment shader dapat menerima data yang di pass dari fungsi vertex shader dengan menggunakan varying. Dapat dilihat pada 2 potongan kode diatas bahwa dilakukan deklarasi varying v_color pada fungsi vertex shader, kemudian dideklarasikan varying yang sama di fungsi fragment shader. WebGL akan secara otomatis menghubungkan varying dengan nama dan tipe yang sama.

Hasil dari kedua fungsi diatas adalah segitiga dengan warna interpolasi antara 3 warna yang dikomputasi, dan warna yang ditampilkan bersifat relative terhadap background (karena merupakan hasil komputasi dari clipspace). WebGL secara otomatis akan menampilkan warna interpolasi karena fungsi tersebut menggunakan varying. Hasil dapat dilihat pada file project 01.

Fungsi vertex shader dapat melakukan passing data yang lebih banyak kepada fungsi fragment shader, dengan menambahkan attribute baru dan menyalin warna dari attribute ke varying.

Untuk menentukan warna yang digunakan oleh WebGL, pada kode javascript akan dibuat buffer untuk warna-warna, kemudian buffer tersebut akan diisi dengan warna yang dipilih. Pada waktu render dilakukan binding terhadap buffer warna dan set up atribut warna untuk mengambil data dari hasil binding tersebut.

Buffer adalah cara WebGL untuk menyimpan vertex dan data yang berhubungan ke GPU, dan buffer ini biasanya dibuat dan di bind pada waktu inisialisasi.

```
  // look up where the vertex data needs to go.
  var positionLocation = gl.getAttribLocation(program, "a_position");
  var colorLocation = gl.getAttribLocation(program, "a_color");

  // Create a buffer for the colors.
  var colorBuffer = gl.createBuffer();
  gl.bindBuffer(gl.ARRAY_BUFFER, colorBuffer);
  // Set the colors.
  setColors(gl);

// Fill the buffer with colors for the 2 triangles
// that make the rectangle.
function setColors(gl) {
  // Pick 2 random colors.
  var r1 = Math.random();
  var b1 = Math.random();
  var g1 = Math.random();
  var r2 = Math.random();
  var b2 = Math.random();
  var g2 = Math.random();

  gl.bufferData(
      gl.ARRAY_BUFFER,
      new Float32Array(
        [ r1, b1, g1, 1,
          r1, b1, g1, 1,
          r1, b1, g1, 1,
          r2, b2, g2, 1,
          r2, b2, g2, 1,
          r2, b2, g2, 1]),
      gl.STATIC_DRAW);
}
```

Kode diatas akan menghasilkan sebuah persegi yang terdiri dari 2 segitiga, dimana kedua segitiga tersebut memiliki 2 warna solid yang berbeda. Pada kasus ini, warna yang muncul adalah warna solid padahal pada fungsi digunakan varying, sedangkan varying seharusnya melakukan interpolasi warna. Hasil ini dapat dilihat pada file project 02.

Agar dihasilkan warna interpolasi, harus dipastikan bahwa untuk tiap 3 vertex di setiap segitiga memiliki warna yang berbeda agar varying dapat menghasilkan interpolasi warna. Dilakukan modifikasi kode dan dihasilkan sebuah persegi dengan 2 segitiga yang masing-masing memiliki warna interpolasi, dapat dilihat pada file project 03.

Passing data dari vertex shader ke fragment shader juga dapat dilakukan untuk memberikan koordinat tekstur.

Hal menarik lain adalah proses normalisasi untuk tipe data non floating point. Misalnya jika normalize flag di set true, maka yang awalnya nilai byte adalah -128 hingga 127 akan dikonversi menjadi representasi -1.0 hingga +1.0, nilai unsigned_byte 0 hingga 255 menjadi 0.0 hingga +1.0. Proses normalisasi ini sangat sering dilakukan untuk memproses data warna. Perlunya normalisasi disebabkan penggunaan tipe full float untuk tiap variasi warna akan menghabiskan space sebanyak

16 byte untuk tiap vertex dan warna. Jika dilakukan normalisasi, maka data warna akan di konversi sehingga hanya memerlukan 4 byte untuk tiap vertex dan warna.

Proses normalisasi dapat dilakukan dengan potongan kode berikut

```
// Tell the position attribute how to get data out of positionBuffer (ARRAY_BUFFER)
var size = 2;          // 2 components per iteration
var type = gl.FLOAT;   // the data is 32bit floats
var normalize = false; // don't normalize the data
var stride = 0;        // 0 = move forward size * sizeof(type) each iteration to get the next position
var offset = 0;        // start at the beginning of the buffer
gl.vertexAttribPointer(
    positionLocation, size, type, normalize, stride, offset);
```

dan hasilnya dapat dilihat pada file project 04.

Project II – Image Processing

Pada project ini dijelaskan cara menghasilkan gambar dengan tekstur. Penggambaran tekstur ini menggunakan koordinat seperti halnya clip space. Koordinat tekstur bernilai mulai dari 0.0 hingga 0.1 tanpa bergantung ke dimensi.

Dengan memanfaatkan vertex shader dan fragment shader di project sebelumnya, maka kini tinggal me load gambar dan membuat tekstur. Berikut ini kode yang digunakan

```
function main() {
  var image = new Image();
  image.src = "http://someimage/on/our/server";  // MUST BE SAME DOMAIN!!!
  image.onload = function() {
    render(image);
  }
}

function render(image) {
  ...
  // all the code we had before.
  ...
  // look up where the texture coordinates need to go.
  var texCoordLocation = gl.getAttribLocation(program, "a_texCoord");

  // provide texture coordinates for the rectangle.
  var texCoordBuffer = gl.createBuffer();
  gl.bindBuffer(gl.ARRAY_BUFFER, texCoordBuffer);
  gl.bufferData(gl.ARRAY_BUFFER, new Float32Array([
      0.0,  0.0,
      1.0,  0.0,
      0.0,  1.0,
      0.0,  1.0,
      1.0,  0.0,
      1.0,  1.0]), gl.STATIC_DRAW);
  gl.enableVertexAttribArray(texCoordLocation);
  gl.vertexAttribPointer(texCoordLocation, 2, gl.FLOAT, false, 0, 0);

  // Create a texture.
  var texture = gl.createTexture();
  gl.bindTexture(gl.TEXTURE_2D, texture);

  // Set the parameters so we can render any size image.
  gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_WRAP_S, gl.CLAMP_TO_EDGE);
  gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_WRAP_T, gl.CLAMP_TO_EDGE);
  gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER, gl.NEAREST);
  gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MAG_FILTER, gl.NEAREST);

  // Upload the image into the texture.
  gl.texImage2D(gl.TEXTURE_2D, 0, gl.RGBA, gl.RGBA, gl.UNSIGNED_BYTE, image);
  ...
}
```

Hasil dari kode tersebut dapat dilihat di file project 05.

WebGL juga dapat melakukan manipulasi pada gambar, misalnya dengan merubah warna. Pada file project 06 dilakukan contoh merubah warna merah dengan biru.

WebGL juga memungkinkan melakukan referensi antar pixel. Proses referensi ini dapat dilakukan dengan matematika sederhana yaitu jumlah perpindahan yang dibutuhkan untuk satu pixel adalah 1 per ukuran tekstur.

Hasil dari modifikasi tekstur misalnya berupa perubahan tingkat blur pada gambar. Pada file project 07 dapat dilihat modifikasi tingkat blur dibandingkan dengan file project 05.

Untuk melakukan image processing juga dapat memanfaatkan convolution kernel yang direpresentasikan dalam bentuk matriks 3x3 dimana setiap entri adalah jumlah hasil perkalian antar pixel yang akan direnderdengan 8 pixel disekitarnya, kemudian dibagi dengan berat atau total nilai di kernel. Contoh kode kernel ini dapat dilihat pada file project 08.

Project III – WebGL 2D Matrices

Sebelum memulai project dijelaskan bahwa terdapat perbedaan antara matriks pada matematika dan matriks pada WebGL. Pada WebGL, penempatan kolom dan baris ditukar sehingga jika ada contoh matriks matematika sebagai berikut

$$\begin{bmatrix} 1 & 0 & 0 & tx \\ 0 & 1 & 0 & ty \\ 0 & 0 & 1 & tz \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

matriks diatas tidak akan dituliskan dengan cara umum pada bahasa pemrograman

```
const some4x4TranslationMatrix = [
   1,  0,  0,  tx,
   0,  1,  0,  ty,
   0,  0,  1,  tx,
   0,  0,  0,  1,
];
```

tetapi akan dituliskan sebagai berikut

```
const some4x4TranslationMatrix = [
   1,  0,  0,  0,   // this is column 0
   0,  1,  0,  0,   // this is column 1
   0,  0,  1,  0,   // this is column 2
  tx, ty, tz,  1,   // this is column 3
];
```

perubahan cara penulisan diatas dilakukan dengan tujuan agar mudah mengakses kolom tertentu pada matriks.

Pada kode yang akan ditulis di WebGL, matriks akan di proses dari kanan ke kiri. Sehingga misalnya ada baris kode sebagai berikut

```
projectionMat * translationMat * rotationMat * scaleMat * position
```

maka akan dilakukan perhitungan scaleMat * position, dilanjutkan komputasi dengan rotationMat, dan seterusnya hingga projectionMat.

Pada project kedua ini akan dilakukan transformasi pada data matriks, mencakup translasi, rotasi, dan perubahan skala. Transformasi yang dilakukan akan merubah juga shader yang sudah ada, atau bahkan jadi diperlukan komputasi shader baru. Untuk menghindari pembuatan shader baru, dapat dilakukan perhitungan pada matriks seperti biasa dilakukan pada matematika.

Berikut ini tabel yang menampilkan kode yang diperlukan untuk tiap jenis transformasi

Jenis Transformasi	Rumus
Translasi	`newX = x + tx;` `newY = y + ty;`

Rotasi	`s = Math.sin(angleToRotateInRadians);`
	`c = Math.cos(angleToRotateInRadians);`
	disederhanakan menjadi

	newX = x * c + y * s; newY = x * -s + y * c;
Merubah skala	newX = x * sx; newY = y * sy;

Sehingga jika dikonversi ke WebGL akan menghasilkan kode sebagai berikut

```
var m3 = {
  translation: function(tx, ty) {
    return [
      1, 0, 0,
      0, 1, 0,
      tx, ty, 1,
    ];
  },

  rotation: function(angleInRadians) {
    var c = Math.cos(angleInRadians);
    var s = Math.sin(angleInRadians);
    return [
      c,-s, 0,
      s, c, 0,
      0, 0, 1,
    ];
  },

  scaling: function(sx, sy) {
    return [
      sx, 0, 0,
      0, sy, 0,
      0, 0, 1,
    ];
  },
```

Dengan adanya kode diatas maka fungsi vertex shader yang awalnya melakukan langkah scaling, rotasi, dan translasi dapat disederhanakan menjadi hanya melakukan perkalian dengan matriks hasil perhitungan. Berikut ini fungsi vertex shader yang baru.

```
<script  id="vertex-shader-2d" type="x-shader/x-vertex">
attribute vec2 a_position;

uniform vec2 u_resolution;
uniform mat3 u_matrix;

void main() {
  // Multiply the position by the matrix.
  vec2 position = (u_matrix * vec3(a_position, 1)).xy;
```

Hasil perhitungan yang lebih sederhana ini dapat dilihat pada file project 05 dan project 06, dimana perbedaan kedua project tersebut adalah urutan transformasi yang dilakukan terhadap matriks. Urutan transformasi tersebut sangat penting pada aplikasi animasi, seperti dicontohkan pada project 07.

Pada project 07 tersebut dapat dilihat bahwa titik tumpu rotasi ada pada bagian kiri atas. Hal tersebut dikarenakan default pada rotasi bertumpu pada titik origin (0,0). Dengan fungsi translasi yang sudah dibuat sebelumnya, dapat dilakukan perpindahan titik tumpu rotasi. Hasil modifikasi ini dapat dilihat pada project 08, dimana titik tumpu nya kini ada di tengah.

Aplikasi yang lebih penting dari transformasi matriks diatas adalah penyederhanaan kode yang akan mengonversi pixel ke clip space (refer ke Project I – How It Works). Untuk itu, dibuat fungsi tambahan sebagai berikut

```
var m3 = {
  projection: function(width, height) {
    // Note: This matrix flips the Y axis so that 0 is at the top.
    return [
      2 / width, 0, 0,
      0, -2 / height, 0,
      -1, 1, 1
    ];
  },
```

sehingga kode awal yang memerlukan 6-7 langkah kini dapat disederhanakan menjadi hanya 1 langkah. Hasil penyederhanaan dapat dilihat pada file project 09.

Atau alih-alih pemampatan menjadi 1 langkah, dapat digunakan ulang ketiga fungsi awal translate, rotate, dan scale sehingga hasil akhir menjadi 4 langkah sederhana tanpa perlu membuat fungsi projection. Hasil penyederhanaan ini dapat dilihat di file project 10.

Catatan :

Link repository github : https://github.com/afifahfq/if3160grafkom

Link video youtube :

1. How It Works : https://www.youtube.com/watch?v=Y7y204kgcvE
2. WebGL Image Processing : https://www.youtube.com/watch?v=na1oHjsXHdk
3. WebGL 2D Matrices : https://www.youtube.com/watch?v=DWL4HUOfoOM