Penerapan Operasi Citra Aras Titik dan Lokal pada Domain Spasial Berbasis WebGL

Jericho Russel Sebastian - 13521107
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
E-mail: 13521107@std.stei.itb.ac.id

***Abstrak*—Operasi citra, seperti perbaikan citra dan penapisan, menemukan banyak kegunaan dalam berbagai bidang, seperti pencitraan medis, astronomi, dan visi komputer. Di samping itu, di era informasi ini, pemrosesan citra digital masuk semakin dalam di kehidupan sehari-hari dan dimanfaatkan dalam perbaikan mutu citra secara visual. Teknologi yang mendukung pemrosesan citra secara efisien dan *real-time* juga semakin mudah untuk diakses. Hampir semua perangkat yang memiliki unit pemrosesan grafis, atau GPU, dapat digunakan sebagai alat komputasi paralel secara aksesibel oleh pengembang aplikasi menggunakan antarmuka pemrograman grafis yang tersebar luas, seperti WebGL. Dengan demikian, makalah ini disusun untuk menyelidiki dan mengkaji sebuah pendekatan untuk menuangkan konsep operasi citra ke dalam sebuah aplikasi berbasis *web browser* yang mampu menerapkan penapisan citra dengan menggunakan komputasi paralel yang ditawarkan oleh GPU, melalui antarmuka pustaka WebGL.**

Kata Kunci—Citra, Penapisan, Konvolusi, WebGL

I. Pendahuluan

Pemrosesan citra digital adalah proses mengolah sebuah citra menjadi citra lain dengan menggunakan komputer [1][2]. Pemrosesan citra merupakan operasi yang penting, baik untuk peningkatan kualitas visual citra, maupun ekstraksi fitur-fitur dan elemen-elemen dalam citra. Karena itu, pemrosesan citra digital kerap diterapkan dalam berbagai aplikasi yang mengolah atau memanfaatkan citra, seperti pengolahan citra dan video, visi komputer, dan pengenalan pola [3].

Lahirnya pemrosesan citra digital modern dipengaruhi oleh dikembangkannya perangkat komputer yang cukup kuat untuk menangani komputasi pemrosesan citra pada awal tahun 1960-an [2]. Teknik-teknik komputer digunakan untuk memperbaiki citra-citra yang dihasilkan oleh kamera pada kuar antariksa *Ranger 7* pada 1964. Kemudian, hingga awal 1970-an, berbagai teknik pemrosesan citra digital dikembangkan dan diterapkan pada bidang-bidang seperti pencitraan medis dan observasi Bumi.

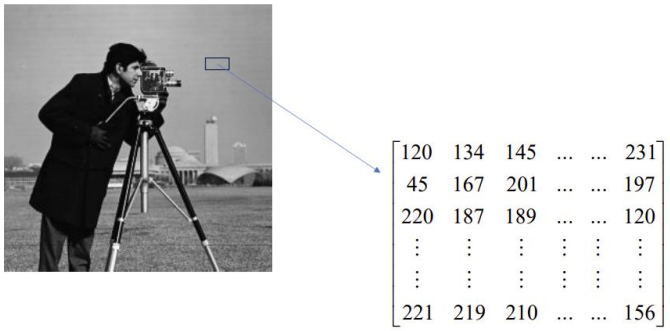
Seiring dengan berkembangnya teknologi komputer dan tersedianya kapasitas komputasi yang semakin besar, teknik-teknik pemrosesan citra digital dapat diterapkan pada semakin banyak perangkat, seperti laptop dan ponsel pintar. Pemrosesan citra digital kini memanfaatkan kemampuan paralelisasi dan percepatan perangkat keras yang disediakan oleh GPU [4]. Dengan demikian, pemrosesan dapat dilakukan dengan kecepatan tinggi, dan diterapkan dalam aplikasi interaktif ataupun latensi rendah, seperti video dan grafika *real-time*.

Dengan makalah ini, penulis bertujuan untuk mengkaji dan menerapkan teknik-teknik dasar pemrosesan citra digital, yaitu operasi aras titik dan lokal, dengan memanfaatkan salah satu antarmuka pemrograman GPU yang paling tersebar luas pada saat penulisan, yaitu WebGL. Penulis menuangkan hasil kajian dan penerapan teknik-teknik tersebut ke dalam sebuah aplikasi pengolahan citra berbasis *web browser*.

II. Dasar Teori

A. Citra Digital

Citra dapat direpresentasikan secara matematis sebagai sebuah fungsi dwimatra kontinu $f(x, y)$ yang memetakan sebuah titik dwimatra dengan nilai intensitas citra pada titik tersebut [1][2]. Ketika citra direpresentasikan ke dalam bentuk digital, fungsi citra dicuplik secara ruang dalam jangkauan titik-titik tertentu. Sehingga, citra digital diwakili oleh sebuah matriks I berukuran $M \times N$, di mana I_{xy} menyatakan nilai intensitas piksel pada titik (x, y).

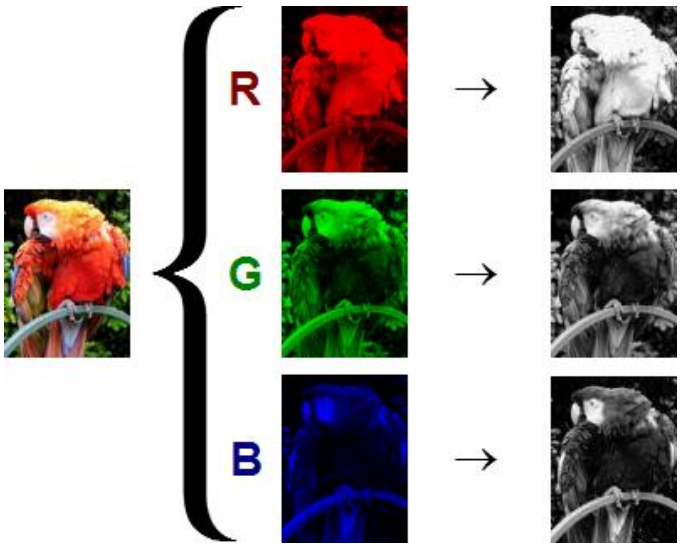


Gambar 1.1 Representasi sebuah citra dalam bentuk matriks nilai intensitas (Sumber: https://informatika.stei.itb.ac.id/~rinaldi.munir/Citra/2024-2025/01-Pengantar-Pemrosesan-Citra-Digital-Bag1-2024.pdf)

Nilai intensitas setiap piksel dalam citra digital dikuantisasi menjadi K nilai bilangan bulat dalam selang $[0, K-1]$, yang disebut sebagai skala keabuan. Karena citra digital disimpan dalam bentuk data biner oleh komputer, nilai K umumnya ditentukan sebagai sebuah perpangkatan dari 2 untuk

memaksimalkan kepadatan piksel: untuk nilai $K = 2^m$, setiap piksel membutuhkan ruang penyimpanan sebesar m bit. Jika nilai $K = 2$, citra digolongkan sebagai citra biner.

Untuk citra berwarna, setiap piksel terdiri dari 3 komponen yang berturut-turut menyatakan intensitas warna merah, hijau, dan biru. Setiap komponen dalam citra membentuk sebuah kanal warna yang merupakan sebuah citra keabuan (*grayscale*). Citra berwarna dapat pula memiliki sebuah komponen tambahan, yaitu *alpha*, yang menyatakan transparansi. Maka, kedalaman piksel pada citra berwarna adalah $3m$ bit untuk citra tanpa kanal *alpha*, dan $4m$ bit untuk citra dengan kanal *alpha*.



Gambar 1.2 Citra keabuan untuk setiap kanal warna pada sebuah citra (Sumber: https://commons.wikimedia.org/wiki/File:RGB_channels_separation.png)

B. *Operasi Citra*

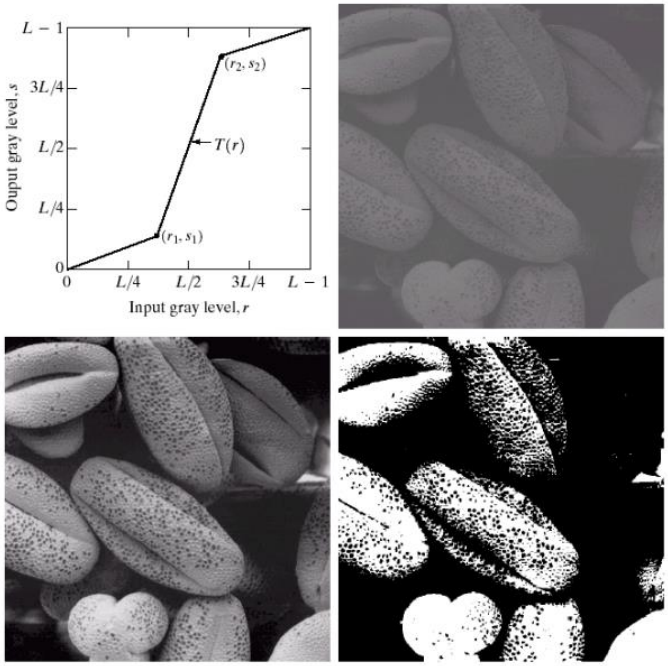
Operasi citra diterapkan untuk mengomputasi sebuah citra baru berdasarkan nilai intensitas pada sebuah citra mula. Ada banyak jenis operasi yang dapat diterapkan pada citra, yang dikelompokkan ke dalam empat aras komputasi [5], yaitu:

1. Aras titik, yaitu operasi citra yang hanya mengambil sebuah piksel tunggal sebagai masukan; didefinisikan sebagai:
$$f(x,y)' = O\{f(x,y)\}$$
2. Aras lokal, yaitu operasi citra yang memetakan intensitas sebuah piksel terhadap sebuah nilai intensitas yang bergantung kepada piksel-piksel tetangganya; didefinisikan sebagai:
$$f(x,y)' = O\{f(x_i,y_i); (x_i,y_i) \in N(x,y)\}$$
3. Aras global, yaitu operasi citra yang menggunakan nilai intensitas seluruh piksel dalam citra; didefinisikan sebagai:
$$f(x,y)' = O\{f(x_i,y_i); (x_i,y_i) \in I\}$$
4. Aras objek, yaitu operasi yang diterapkan hanya pada piksel-piksel dari objek tertentu di dalam citra.

Terdapat beberapa contoh operasi citra aras titik, yaitu:

a. Pencerahan citra (*image brightening*), yaitu menambahkan sebuah konstanta b ke nilai intensitas piksel pada citra.
$$f(x,y)' = f(x,y) + b$$
Dampak dari operasi ini bergantung kepada nilai b, di mana jika nilai b positif, maka kecerahan citra bertambah, sedangkan jika nilai b negatif, maka kecerahan citra berkurang.

b. Peregangan kontras (*contrast stretching*), yaitu meluaskan atau menyempitkan rentang intensitas pada citra. Umumnya, peregangan kontras dilakukan dengan memetakan rentang intensitas mula dari citra masukan terhadap sebuah rentang baru yang lebih luas.



Gambar 2.1 Operasi peregangan kontras terhadap citra kontras rendah (Sumber: https://informatika.stei.itb.ac.id/~rinaldi.munir/Citra/2024-2025/08-Image-Enhancement-Bagian1-2024.pdf)

Operasi ini mengubah rentang intensitas citra, sehingga jarak antara intensitas masing-masing piksel menjadi lebih lebar. Peregangan kontras dapat dihampiri dengan sebuah perkalian skalar:
$$f(x,y)' = cf(x,y)$$

c. Negasi citra (*image negation*), yaitu membalikkan nilai intensitas pada citra. Operasi ini mensimulasikan film negatif pada bidang fotografi. Negasi citra dapat dilakukan dengan mengurangi intensitas masukan dengan nilai intensitas maksimum dari skala keabuan citra.
$$f(x,y)' = (K-1) - f(x,y)$$

d. Koreksi *gamma* (*gamma correction*), yaitu operasi untuk mengoreksi dampak hukum pangkat pada citra. Umumnya, perangkat yang menangkap ataupun

menampilkan citra tidak merespon terhadap sinyal citra secara linear, namun mengikuti hukum pangkat.

$$f(x,y)' = cf(x,y)^{\gamma}$$

Setiap perangkat memiliki nilai γ (*gamma*) yang berbeda, dan menghasilkan respons yang berbeda pula, yang mengakibatkan intensitas citra yang dihasilkan tidak sesuai dengan intensitas aslinya. Sebagai contoh, nilai $\gamma > 1$ mengakibatkan citra tampak lebih gelap dari seharusnya. Untuk mengatasi fenomena ini, diterapkan koreksi *gamma* yang membalikkan dampak respons tersebut.

$$f(x,y)' = c'f(x,y)^{\frac{1}{\gamma}}$$

e. Citra hitam-putih (*black-and-white image*) mengubah citra berwarna menjadi citra keabuan. Nilai piksel pada citra hasil merupakan tingkat kecerahan piksel mula, dan dapat dihitung sebagai kombinasi linear dari komponen-komponen warna dari piksel mula.

$$f(x,y)' = [P \quad Q \quad R]\begin{bmatrix} f_R(x,y) \\ f_G(x,y) \\ f_B(x,y) \end{bmatrix}$$

Nilai (P, Q, R) menentukan tingkat kontribusi setiap komponen warna terhadap kecerahan piksel. Ketiga nilai tersebut dipilih sehingga $P + Q + R = 1$. Umumnya, nilai (P, Q, R) yang dipilih adalah $(0.299, 0.587, 0.114)$.

f. Pengambangan citra (*image thresholding*) mengubah citra keabuan menjadi citra biner berdasarkan sebuah nilai ambang m.

$$f(x,y)' = \begin{cases} 0, & f(x,y) < m \\ 1, & f(x,y) \geq m \end{cases}$$

g. Kuantisasi piksel (*pixel quantization*), dikenal juga sebagai posterisasi, bertujuan menyempitkan skala keabuan dari citra masukan, dengan memetakan nilai intensitas dalam sebuah rentang ke sebuah nilai diskrit tunggal. Skala keabuan mula citra dibagi menjadi K' jangkauan dengan ukuran $\frac{K}{K'}$. Kemudian, nilai intensitas baru hasil pemetaan adalah:

$$f(x,y)' = \frac{1}{K'-1}\lfloor Kf(x,y) \rfloor$$

Jika $K' = 2$, maka kuantisasi piksel ekuivalen dengan operasi pengambangan dengan $m = \frac{K-1}{2}$. Kuantisasi piksel pada citra berwarna dapat dilakukan menggunakan nilai K' yang berbeda untuk setiap kanal warna, sehingga memiliki parameter (K'_R, K'_G, K'_B).

h. Pengaturan HSL (*HSL adjustment*) bertujuan mengubah nilai intensitas piksel pada citra berwarna pada ruang warna HSL (*hue-saturation-luminance*). Setiap piksel berwarna dengan komponen (R, G, B) pada citra diubah menjadi representasi dalam ruang HSL sebagai berikut [6]:

$$M = \max\{R,G,B\}\,; m = \min\{R,G,B\}$$

$$\Delta = M - m$$

$$L = \frac{M+m}{2}$$

$$S = \begin{cases} 0, & \Delta = 0 \\ \dfrac{C}{1-|2L-1|}, & \Delta \neq 0 \end{cases}$$

$$H = \begin{cases} 60° \dfrac{G-B}{\Delta} \bmod 360°, & M = R \\ 60° \left(\dfrac{B-R}{\Delta} + 2\right) \bmod 360°, & M = G \\ 60° \left(\dfrac{R-G}{\Delta} + 4\right) \bmod 360°, & M = B \end{cases}$$

Kemudian, nilai komponen HSL digeser dengan parameter (h, s, l), yang menghasilkan komponen baru $(H', S', L') = (H + h \bmod 360°, S + s, L + l)$. Nilai komponen warna akhir dari piksel dihitung berdasarkan komponen HSL baru: jika $S = 0$, maka $R = G = B = L$; jika tidak, maka:

$$\Delta = S(1 - |2L - 1|)$$

$$H' = \frac{H}{60°}$$

$$X = \Delta(1 - |H' \bmod 2 - 1|)$$

$$m = L - \frac{\Delta}{2}$$

$$(R-m, G-m, B-m) = \begin{cases} (\Delta, X, 0) & 0 \leq H' \leq 1 \\ (X, \Delta, 0) & 1 \leq H' \leq 2 \\ (0, \Delta, X) & 2 \leq H' \leq 3 \\ (0, X, \Delta) & 3 \leq H' \leq 4 \\ (X, 0, \Delta) & 4 \leq H' \leq 5 \\ (\Delta, 0, X) & 5 \leq H' \leq 6 \end{cases}$$

Operasi citra pada aras lokal umumnya berupa sebuah konvolusi terhadap citra dengan sebuah matriks kernel berukuran lebih kecil dari citra. Konvolusi didefinisikan sebagai [5]:

$$f(x,y) * g(x,y) = \sum_{a=-\infty}^{\infty} \sum_{b=-\infty}^{\infty} f(a,b)g(x-a, y-b)$$

C. WebGL

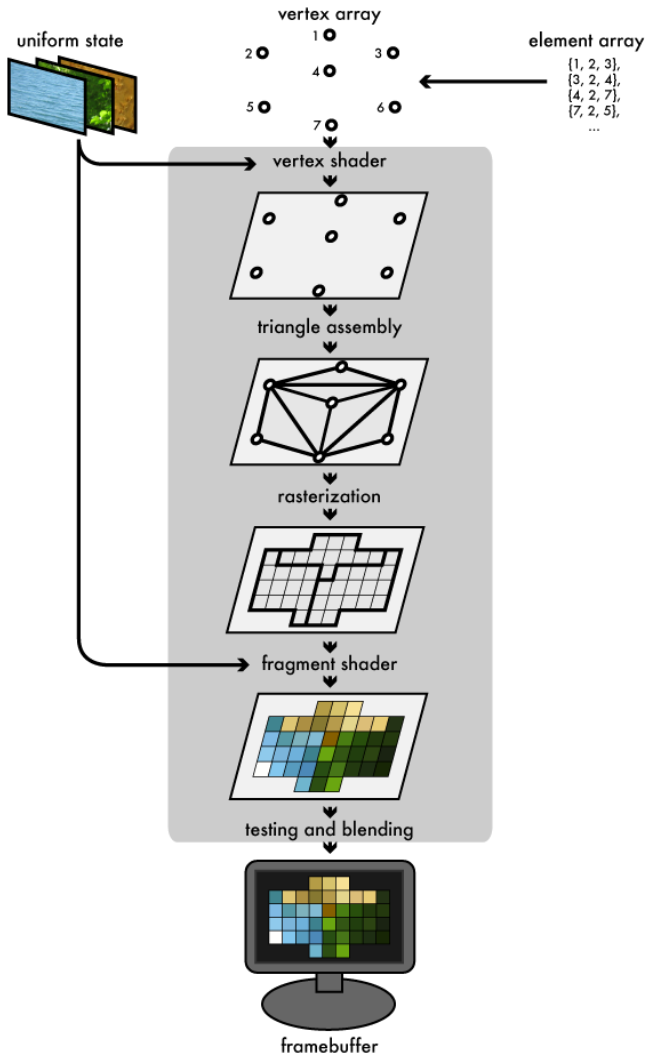
WebGL merupakan antarmuka pemrograman aplikasi berbahasa JavaScript yang menyediakan fungsionalitas interaksi dengan GPU perangkat menggunakan pustaka OpenGL ES [7]. Dengan memanfaatkan pustaka WebGL, pengembang aplikasi berbasis *web browser* dapat memanfaatkan kapabilitas GPU dengan menulis kode program berbahasa GLSL, yang disebut sebagai *shader*.

WebGL dapat dimanfaatkan untuk menghasilkan grafika dwimatra maupun trimatra, menggunakan spesifikasi antarmuka pemrograman yang mirip dengan OpenGL ES. Mayoritas *web browser* seperti Chrome, Edge, Safari, dan Firefox sudah menerapkan spesifikasi WebGL dalam pustaka JavaScript standarnya.

OpenGL pada dasarnya adalah pustaka penggambaran grafika trimatra. OpenGL berfungsi dengan menerapkan sebuah *pipeline* penggambaran yang menerima sebuah program GL dari aplikasi. Kemudian, *pipeline* akan menerima data atribut simpul (*vertex*) dari aplikasi, yang mewakili objek-objek yang akan digambar, beserta dengan data seragam (*uniform*) yang diterapkan pada semua objek. Setiap program GL terdiri dari sepasang *shader*, yaitu:

- *Vertex shader*, yang menerima atribut simpul (*vertex attribute*) dan menentukan letak dari simpul pada ruang trimatra sebelum primitif objek dibentuk. *Vertex shader* juga bertanggung jawab untuk mengolah sebagian atribut dan mengirimkan hasilnya ke *pipeline* berupa data *varying* untuk diolah lebih lanjut.
- *Fragment shader*, yang menerima fragmen-fragmen hasil rasterisasi dari primitif objek dan menentukan warna dari setiap fragmen tersebut. *Fragment shader* dapat menerima data *varying* yang telah diinterpolasi oleh *pipeline* untuk melakukan komputasi.

Luaran dari program GL dituliskan pada sebuah *framebuffer*, yang kemudian ditampilkan dalam kanvas.



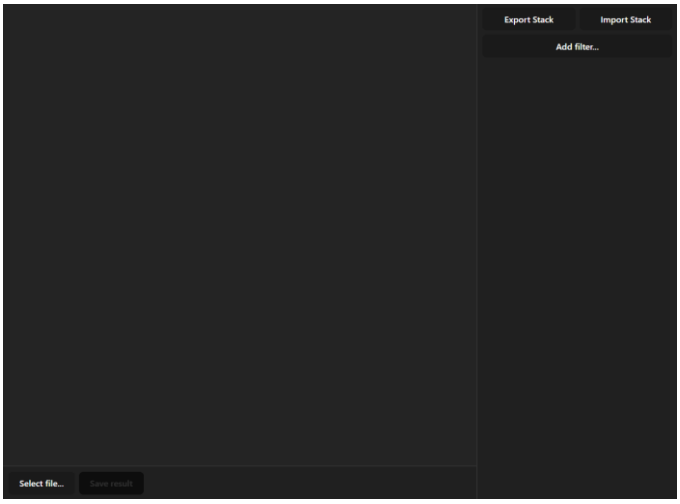
Gambar 2.2 *Pipeline* grafika umum dari sebuah aplikasi WebGL (Sumber: https://duriansoftware.com/joe/an-intro-to-modern-opengl.-chapter-1:-the-graphics-pipeline)

WebGL memanfaatkan paralelisasi GPU untuk memproses banyak simpul dan fragmen secara sekaligus. Hal ini menyebabkan penggambaran grafika menggunakan WebGL memakan waktu yang relatif lebih pendek dibandingkan dengan pemrosesan berbasis CPU.

III. Penerapan Operasi Citra Aras Titik dan Lokal pada Domain Spasial Berbasis WebGL

Penulis menerapkan operasi-operasi dasar citra aras titik dan lokal dalam wujud GLFX, yaitu sebuah aplikasi berbasis *web browser* dengan memanfaatkan antarmuka pemrograman WebGL untuk memungkinkan pengolahan piksel citra secara paralel dan latensi rendah.

Berikut adalah tampilan utama dari aplikasi GLFX:



Gambar 3.1 Tampilan utama aplikasi GLFX (Sumber: dokumen pribadi)

Tampilan utama GLFX terdiri dari dua bagian, yaitu:

- Kanvas citra, yang digunakan untuk menampilkan citra hasil penapisan, dan
- Rak penapis, yang menampilkan susunan penapis yang diterapkan pada citra.

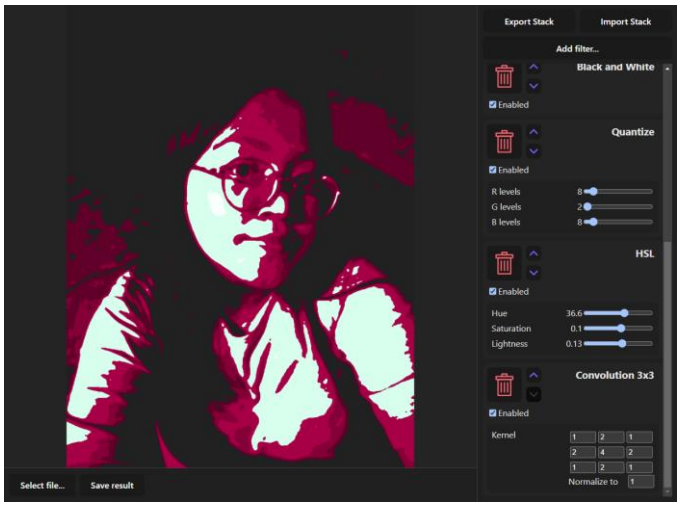
Alur penggunaan aplikasi adalah sebagai berikut:

1. Mengklik tombol *Select file...* pada bagian bawah kanvas citra untuk memuat citra ke aplikasi.
2. Mengklik tombol *Add filter...* pada bagian atas rak penapis untuk menambahkan penapis yang akan digunakan. Penapis ditambahkan pada bagian akhir rak penapis, dan citra akan ditapis secara otomatis menggunakan seluruh rangkaian penapis dari atas ke bawah.
3. Mengubah parameter masing-masing penapis dengan menggunakan moda masukan yang disediakan pada rak penapis hingga mendapatkan hasil yang sesuai.

4. Mengklik tombol *Save result* pada bagian bawah kanvas citra untuk menyimpan citra yang sudah ditapis ke perangkat.

Masing-masing penapis pada rak memiliki tombol *Delete* (dengan ikon tempat sampah) untuk menghapus penapis dari rak, dan sepasang tombol panah untuk memindahkan urutan penapis. Di samping itu, penapis dapat dimatikan dan dihidupkan menggunakan kotak centang *Enabled*. Ketika penapis dimatikan, aplikasi akan mengabaikan penapis tersebut tanpa menghapusnya ketika memproses rak, sehingga fitur ini penting untuk membandingkan dampak dari sebuah penapis terhadap citra.

Rak penapis dapat disimpan ke sebuah *file* dengan format JSON dengan mengklik tombol *Export stack* pada bagian atas rak penapis. *File* akan menyimpan urutan penapis dalam rak beserta nilai-nilai parameternya, namun tidak akan menyimpan citra yang telah dimuat. Rak yang telah disimpan dapat digunakan kembali dengan memuat *file* dari rak dengan mengklik tombol *Import stack*.



Gambar 3.2 Contoh penggunaan aplikasi GLFX (Sumber: dokumen pribadi)

Aplikasi GLFX menyediakan pilihan-pilihan penapis sebagai berikut:

- *Invert*: melakukan negasi citra.
- *Black and White*: mengubah citra menjadi hitam-putih.
- *Brightness/Contrast*: menerapkan pencerahan citra dan peregangan kontras secara sekaligus. Penapis ini menerima dua parameter, yaitu *Brightness* (B) dan *Contrast* (C), yang berada dalam rentang $[-1, 1]$, dan mentransformasikan citra menggunakan fungsi berikut:

$$f(x,y)' = Cf(x,y) + B$$

- *Gamma correction*: menerapkan koreksi *gamma* menggunakan parameter *Gamma* (γ). Nilai *Gamma* dibatasi dalam rentang $[0.01, 7.99]$, namun nilai yang lebih kecil atau besar dapat dicapai dengan menerapkan penapis ini secara berulang, dengan nilai *gamma* efektif dari sebuah rantai penapis *Gamma correction* adalah hasil perkalian nilai *gamma* setiap penapis tunggal:

$$\gamma_{\text{eff}} = \prod \gamma_i$$

- *HSL*: mentransformasikan citra dalam ruang warna HSL. Penapis menerima parameter *Hue* (h), *Saturation* (s), dan *Luminance* (l).
- *Binarize*: mengambangkan citra dengan sebuah nilai ambang *Threshold* (m).
- *Quantize*: mengkuantisasi nilai piksel citra ke dalam skala keabuan yang lebih kecil. Penapis menerima parameter ukuran skala keabuan baru untuk ketiga kanal warna (K'_R, K'_G, dan K'_B).
- *Convolution 3×3* dan *Convolution 5×5*: melakukan konvolusi terhadap citra dengan menggunakan sebuah matriks kernel yang nilainya didapatkan dari masukan aplikasi. Nilai matriks kernel dinormalisasi sedemikian hingga jumlah setiap bobot sama dengan nilai yang dimasukkan pada isian *Normalize to*, kecuali jika bobot matriks berjumlah nol.
- *Gaussian blur*: menerapkan pengaburan Gauss terhadap citra. Penapis menerima sebuah parameter *Radius* (r), dan bekerja secara *double-pass*, yaitu menerapkan konvolusi dua kali dengan kernel bobot Gauss ekamatra, yaitu konvolusi secara horizontal, diikuti dengan konvolusi secara vertikal. Akibat keterbatasan WebGL, penapis *Gaussian blur* hanya mendukung nilai r dalam rentang $[3, 24]$, namun nilai r yang lebih tinggi dapat dicapai dengan menerapkan beberapa penapis *Gaussian blur* secara berulang dengan nilai r efektif sebagai berikut:

$$r_{\text{eff}} = \sqrt{\sum r_i^2}$$

Setiap penapis diterapkan dalam bentuk program GL yang ditulis dalam bahasa GLSL. Setiap program GL terdiri dari dua buah *shader*, yaitu:

- *Vertex shader*, yang bertugas menempatkan segiempat citra dan menentukan acuan koordinat piksel. Pada penapis konvolusi dan pengaburan Gauss, *vertex shader* juga bertanggung jawab menentukan koordinat piksel tetangga yang digunakan dalam tahap konvolusi. Di samping itu, pada penapis pengaburan Gauss, *vertex shader* juga berfungsi untuk menghitung bobot kernel menggunakan fungsi distribusi Gauss, dengan standar deviasi bernilai $\sigma = \frac{r-1}{6}$.
- *Fragment shader*, yang bertugas menghitung nilai intensitas baru berdasarkan nilai piksel pada citra. *Shader* ini merupakan kode program utama bagi setiap penapis. Untuk operasi citra aras titik, *fragment shader* mendefinisikan pemetaan nilai intensitas menggunakan parameter yang diterima oleh aplikasi. Pada operasi yang melibatkan konvolusi, *fragment shader* melakukan operasi konvolusi terhadap masing-masing fragmen tunggal dengan menggunakan nilai

intensitas piksel tetangga yang bersesuaian, dengan memanfaatkan nilai koordinat piksel yang telah dihitung oleh *vertex shader*.

Berikut adalah kode sumber dari *vertex shader* yang digunakan untuk penapis operasi aras titik:

```
#version 100

attribute vec2 a_position;
attribute vec2 a_uv;

varying vec2 v_uv;

void main(){
  v_uv = a_uv;
  gl_Position = vec4(a_position, 0.0, 1.0);
}
```

Berikut adalah kode sumber dari *fragment shader* untuk salah satu penapis operasi aras titik sederhana, yaitu *Brightness/Contrast*:

```
#version 100
precision mediump float;

varying vec2 v_uv;

uniform sampler2D u_texture;
uniform float u_brightness;
uniform float u_contrast;

void main(){
  float contrastFactor = 259.0 * (u_contrast +
255.0) / (255.0 * (259.0 - u_contrast));
  vec4 color = texture2D(u_texture, v_uv);
  gl_FragColor = clamp(contrastFactor * (color -
0.5) + 0.5 + u_brightness / 256.0, 0.0, 1.0);
}
```

Setiap penapis operasi aras titik memiliki *fragment shader* dengan bentuk serupa, namun dengan masukan *uniform* yang berbeda dan operasi komputasi yang disesuaikan dengan metode perhitungan masing-masing penapis.

Berikut adalah kode sumber dari *vertex shader* untuk penapis konvolusi 3×3. *Vertex shader* di bawah memberikan luaran koordinat 9 piksel yang relevan dalam operasi konvolusi.

```
#version 100

attribute vec2 a_position;
attribute vec2 a_uv;

varying vec2 v_uvLU;
varying vec2 v_uvMU;
varying vec2 v_uvRU;

varying vec2 v_uvLM;
varying vec2 v_uvMM;
varying vec2 v_uvRM;

varying vec2 v_uvLD;
varying vec2 v_uvMD;
varying vec2 v_uvRD;

uniform vec2 u_resolution;

void main(){
  vec2 stepXY = 1.0 / u_resolution;
  vec2 stepX = vec2(stepXY.x, 0.0);
  vec2 stepY = vec2(0.0, stepXY.y);
  vec2 stepXnY = vec2(stepXY.x, -stepXY.y);

  v_uvLU = a_uv - stepXnY;
  v_uvMU = a_uv + stepY;
  v_uvRU = a_uv + stepXY;

  v_uvLM = a_uv - stepX;
  v_uvMM = a_uv;
  v_uvRM = a_uv + stepX;

  v_uvLD = a_uv - stepXY;
  v_uvMD = a_uv - stepY;
  v_uvRD = a_uv + stepXnY;

  gl_Position = vec4(a_position, 0.0, 1.0);
}
```

Pada kode di atas, *vertex shader* menghitung koordinat piksel tetangga dari masing-masing titik sudut citra, dan menyimpannya dalam bentuk *varying*. Program memanfaatkan interpolasi *varying* yang diterapkan oleh OpenGL untuk menentukan nilai *varying* yang bersesuaian pada setiap fragmen, sedemikian hingga nilai setiap koordinat piksel *varying* yang diterima oleh *fragment shader* adalah koordinat piksel tetangga dari fragmen yang bersesuaian. Pendekatan yang serupa digunakan oleh *vertex shader* dari penapis *Gaussian blur*, namun dengan menggunakan *varying* dalam bentuk *array*, seperti berikut:

```
#version 100
precision mediump float;
precision mediump int;

#define MAX_RADIUS 24
#define PI 3.1415928

attribute vec2 a_position;
attribute vec2 a_uv;

varying vec2 v_uv[MAX_RADIUS];
varying float v_weights[MAX_RADIUS];

uniform vec2 u_resolution;
uniform int u_radius;

void main(){
  vec2 stepXY = 1.0 / u_resolution;
  vec2 stepX = vec2(stepXY.x, 0.0);
  float r = float(u_radius);

  for(int i = 0; i < MAX_RADIUS; i++){
    if(i >= u_radius)break;

    float idx = float(i);
    float offset = idx - floor(r / 2.0);
    v_uv[i] = a_uv + offset * stepX;

    float s = float(u_radius - 1) / 6.0;
    v_weights[i] = exp(-float(offset * offset) /
```

```
      (2.0 * s * s));
  }

  gl_Position = vec4(a_position, 0.0, 1.0);
}
```

Kode di atas merupakan *vertex shader* untuk *pass* pengaburan Gauss dengan kernel horizontal. Untuk *pass* vertikal, koordinat piksel dihitung dengan melangkah di arah sumbu Y. Berikut adalah *fragment shader* untuk pengaburan Gauss yang menggunakan nilai bobot di atas:

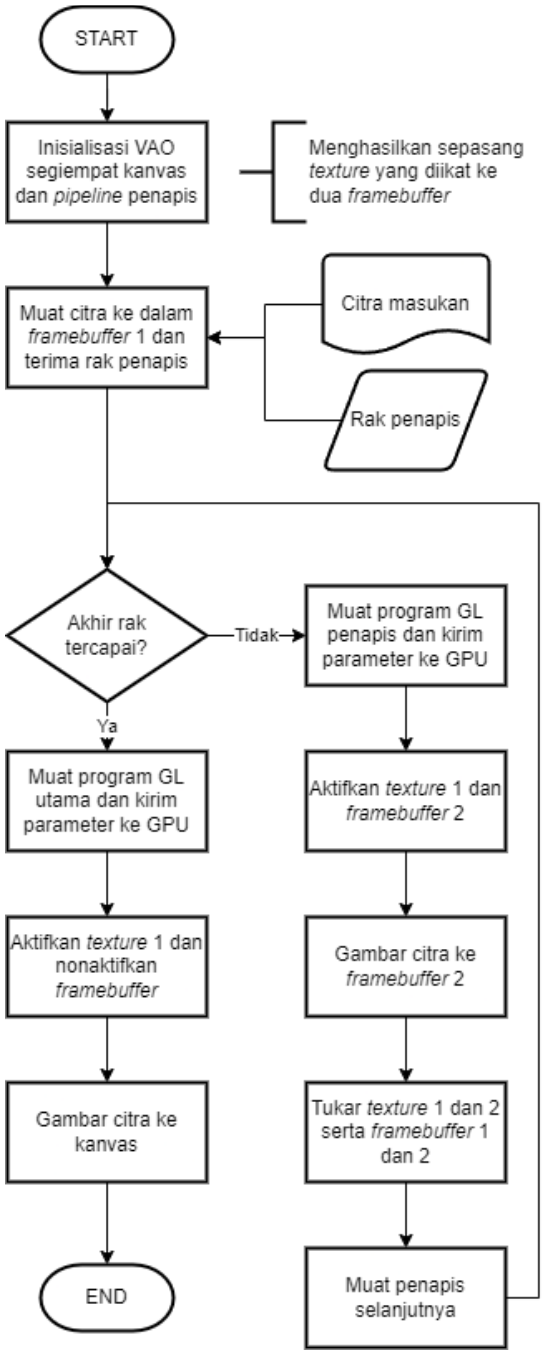
```
#version 100
precision mediump float;
precision mediump int;

#define MAX_RADIUS 24

varying vec2 v_uv[MAX_RADIUS];
varying float v_weights[MAX_RADIUS];

uniform sampler2D u_texture;
uniform int u_radius;

void main(){
  vec4 color = vec4(0.0);
  float weightSum = 0.0;

  for(int i = 0; i < MAX_RADIUS; i++){
    if(i >= u_radius)break;

    color += texture2D(u_texture, v_uv[i]) *
v_weights[i];
    weightSum += v_weights[i];
  }

  gl_FragColor = color / weightSum;
}
```

Setelah seluruh penapis pada rak sudah diterapkan, *framebuffer* yang memuat citra akhir akan ditampilkan pada kanvas aplikasi GLFX menggunakan program GL dengan *vertex shader* sebagai berikut:

```
#version 100

attribute vec2 a_position;
attribute vec2 a_uv;

varying vec2 v_uv;

uniform vec2 u_scale;

void main(){
  v_uv = a_uv;
  gl_Position = vec4(u_scale * a_position, 0.0,
1.0);
}
```

Nilai *uniform* `u_scale` adalah faktor skala untuk mengubah ukuran segiempat citra agar memiliki rasio aspek yang sama dengan citra semula, untuk menghindari terjadinya peregangan atau penyempitan visual dari citra akhir.

Di bawah ini adalah *flowchart* yang menampilkan alur kerja aplikasi GLFX secara umum. Pemroses rak penapis bekerja dengan melakukan penggambaran bergantian antara dua *framebuffer*.



Gambar 3.3 *Flowchart* aplikasi GLFX (Sumber: dokumen pribadi)

Berikut adalah beberapa hasil pemrosesan citra dengan menggunakan beberapa rak penapis berbeda pada aplikasi GLFX:

Gambar 3.4 Perbandingan citra sebelum dan sesudah pemrosesan menggunakan rak perbaikan citra, yang disusun dengan perbaikan kecerahan, kontras, koreksi *gamma*, dan peningkatan saturasi warna (Sumber: dokumen pribadi)

Gambar 3.5 Perbandingan citra sebelum dan sesudah pemrosesan menggunakan rak posterisasi, yang berpusat pada *Gaussian Blur – Black and White – Quantize – HSL* (Sumber: dokumen pribadi)

Gambar 3.6 Perbandingan citra hasil rak pendeteksian tepi menggunakan penapis Sobel (kiri) dan *Laplacian of Gaussian* (kanan), diterapkan dengan penapis *Convolution 3×3* (Sumber: dokumen pribadi)

IV. Kesimpulan dan Saran

Pengujian menunjukkan bahwa aplikasi GLFX mampu menerapkan teknik-teknik operasi citra menggunakan pustaka WebGL dengan baik. Aplikasi dinilai responsif terhadap perubahan parameter dengan latensi yang tidak berarti.

Keterbatasan terbesar dari aplikasi GLFX adalah jumlah pilihan penapisan yang masih relatif sedikit. Namun, arsitektur GLFX didesain untuk kemudahan pengembangan lebih lanjut. Pembuatan penapis baru hanya membutuhkan penambahan entri pada daftar penapis, serta penulisan *shader* yang bersesuaian.

Dalam versi yang dipublikasikan pada saat penulisan makalah ini, aplikasi juga belum mendukung penapisan dan perbaikan citra yang berbasis histogram. Pembentukan histogram dengan memanfaatkan GPU sulit untuk diterapkan akibat keterbatasan WebGL 1.0. Salah satu pendekatan yang dapat digunakan untuk mensimulasikan pembentukan histogram berbasis WebGL 1.0 adalah dengan menggunakan *point rendering*, untuk memetakan intensitas piksel ke sebuah *bin* dalam histogram, dan *additive blending*, untuk mengakumulasi jumlah piksel dalam setiap *bin* tersebut. Ke depannya, perlu dilakukan kajian lebih dalam terhadap metode komputasi histogram berbasis GPU yang efisien dan kompatibel terhadap WebGL 1.0.

Beberapa pengembangan lebih lanjut yang dapat dilakukan terhadap aplikasi GLFX melingkupi:

1. Pemrosesan citra secara *batch* untuk mengolah beberapa citra secara sekaligus. Dengan pemroses rak penapis yang diterapkan pada saat penulisan makalah ini, aplikasi hanya dapat memproses satu citra dalam satu waktu. Pengembangan lebih lanjut dapat dilakukan dengan mengasosiasikan sebuah *framebuffer* untuk setiap instansi penapis, dan mengolah citra-citra masukan secara *pipelining*.
2. Pemrosesan video. Meskipun latensi pemrosesan citra dapat diabaikan untuk citra tunggal, pemrosesan video membutuhkan frekuensi pemrosesan yang tidak kurang dari jumlah *frame* per detik video. Menerapkan sistem yang mampu memenuhi batasan ini dengan berbasis pada teknologi *web browser* merupakan tantangan yang cukup besar. Besar harapan penulis bahwa masalah ini dapat dipecahkan lewat kajian yang lebih luas dan iterasi berlanjut dari pengembangan aplikasi ini.

Ucapan Terima Kasih

Puji dan syukur penulis panjatkan kepada Tuhan Yang Maha Esa; oleh karena kasih karunia dan penyertaan-Nya penulis dapat menyelesaikan makalah berjudul “Penerapan Operasi Citra Aras Titik dan Lokal pada Domain Spasial Berbasis WebGL” ini dengan baik. Penulis mengucapkan terima kasih kepada semua pihak yang terlibat dalam proses penyusunan makalah ini, yaitu:

1. Bapak Dr. Rinaldi Munir, M.T. selaku dosen pengajar mata kuliah IF4073, atas bimbingan dan ilmu yang sarat akan manfaat yang telah dibagikan kepada penulis melalui kegiatan perkuliahan,

2. Kedua orang tua penulis, atas dorongan dan dukungan kepada penulis senantiasa di sepanjang umur hidupnya, yang olehnya penulis mampu menjalani hidup, baik perkuliahan maupun di luarnya, dengan baik dan tidak kekurangan suatu apapun,
3. Kezia Helena P. Naibaho, sebagai seorang partner, sahabat, dan pendukung setia, yang menemani penulis dalam proses menyelesaikan makalah ini, yang memberikan masukan dan saran terkait topik dan penyusunan, dan yang dengan senang hati menyediakan citra untuk diuji dan ditampilkan dalam makalah ini,
4. Para sahabat, teman, dan rekan penulis, atas dukungan dan kontribusi dalam penyusunan dan penyempurnaan makalah ini, dan yang penulis juga doakan agar mampu menyelesaikan tugas-tugas mereka dengan hasil yang memuaskan, serta
5. Para penulis jurnal, artikel, dan buku, yang karyanya penulis jadikan sebagai sumber informasi di sepanjang proses penulisan.

LAMPIRAN

Aplikasi GLFX didesain untuk kompatibel dengan *web browser* apapun yang menyediakan dukungan pustaka WebGL, namun belum memiliki pengaturan khusus untuk antarmuka layar kecil ataupun *portrait* seperti pada perangkat *mobile*. Disarankan untuk mengakses aplikasi GLFX menggunakan perangkat komputer atau laptop. Aplikasi GLFX dapat diakses lewat *web browser* pada tautan berikut:

https://jerichofletcher.github.io/glfx

Di samping itu, kode sumber program untuk aplikasi GLFX dapat dilihat dan diunduh pada *repository* yang diacu oleh tautan berikut:

https://github.com/JerichoFletcher/glfx

REFERENCES

[1] Munir, Rinaldi. 2024. *Pengantar Pemrosesan Citra Digital.* https://informatika.stei.itb.ac.id/~rinaldi.munir/Citra/2024-2025/01-Pengantar-Pemrosesan-Citra-Digital-Bag1-2024.pdf diakses pada 15 Januari 2025, pukul 13:53 WIB.

[2] Gonzales, R. C. dan Woods, R. E. 2008. *Digital Image Processing, 3rd Edition*. Prentice Hall.

[3] *Applications of Digital Image Processing*. https://www.javatpoint.com/applications-of-digital-image-processing diakses pada 15 Januari 2025, pukul 14:05 WIB.

[4] Fernano, R. 2004. *GPU Gems: Programming Techniques, Tips, and Tricks for Real-Time Graphics*. Pearson Higher Education.

[5] Munir, Rinaldi. 2024. *Penapisan Citra dan Konvolusi*. https://informatika.stei.itb.ac.id/~rinaldi.munir/Citra/2024-2025/07-Konvolusi-2024.pdf diakses pada 15 Januari 2025, pukul 17:24 WIB.

[6] Waldman, N. 2013. *Math Behind Colorspace Conversions, RGB-HSL*. https://www.niwa.nu/2013/05/math-behind-colorspace-conversions-rgb-hsl/ diakses pada 15 Januari 2025, pukul 20:01 WIB.

[7] *WebGL: Low-Level 3D Graphics API Based on OpenGL ES.* https://www.khronos.org/webgl/ diakses pada 15 Januari 2025, pukul 20:30 WIB.

PERNYATAAN

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 15 Januari 2025

Jericho Russel Sebastian – 13521107