Implementasi Algoritma *Divide and Conquer* untuk Optimasi Generasi Medan Prosedural

Satrio Dewantono - 13509051[1]
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
[1]13509051@std.stei.itb.ac.id

***Abstrak*—Makalah ini membahas optimasi metode generasi medan prosedural menggunakan fungsi Riuh Perlin (*Perlin Noise*), menjelaskan prinsipnya, menganalisis kompleksitas dan performanya, dan membahas mengenai metode *divide and conquer* sebagai salah satu solusi optimasi dan memberi analisis hasil uji coba.**

Index Terms*—Generasi medan, *Perlin noise*, *divide and conquer

I. PENDAHULUAN

Emulasi medan tiga dimensi yang realistis pada komputer seringkali dibutuhkan khususnya pada bidang hiburan, seperti perfilman dan industri *game*. Tidak jarang dibutuhkan visualisasi variasi topologi medan seperti gunung, lembah, sungai dan sebagainya untuk memberi kesan dunia artifisial yang realistis. Topologi ini mungkin masih mudah untuk dibuat secara manual, petak demi petak, apabila medan yang dibutuhkan memiliki luas yang relatif kecil, namun ketika misalnya dibutuhkan pembuatan medan berukuran ratusan kilometer persegi maka pembuatan manual akan menjadi suatu proses yang melelahkan atau bahkan tidak praktis.

Maka dari itu, untuk memenuhi kebutuhan generasi medan dalam skala masif dapat menggunakan suatu algoritma yang dapat menghasilkan nilai ketinggian untuk suatu nilai x dan y spesifik pada peta medan. Penggunaan suatu algoritma yang tepat untuk kebutuhan ini akhirnya akan memungkinkan pembuatan medan yang luasnya tidak terhingga dan memiliki variasi topologi yang cukup realistis. Terkadang juga dibutuhkan bahwa metode yang digunakan dapat menghasilkan nilai ketinggian yang konsisten untuk suatu x dan y dalam setiap penggunaannya. Nilai-nilai elevasi pada setiap x dan y ini kemudian diinterpolasi dengan tetangga-tetangganya untuk menghasilkan regresi yang halus.

Untuk menghasilkan nilai ketinggian yang konsisten namun acak ini, salah satu metode yang populer adalah pemanfaatan fungsi gangguan (*noise function*), di mana prinsipnya adalah fungsi ini akan menghasilkan bilangan *pseudo-random* yang selalu sama untuk setiap masukan, namun pola regresinya tidak dapat dilihat dengan mudah apabila dipetakan dalam grafik sehingga memberi kesan acak. Metode ini disebut **Riuh Perlin** (***Perlin Noise***), dinamakan setelah penggagas pertamanya, Ken Perlin.

Metodologi spesifik algoritma ini akan dibahas pada bab selanjutnya, namun sekarang perlu diketahui bahwa implementasi fungsi ini bukanlah hal yang ringan, bahkan untuk CPU modern, karena baik dalam proses generasi nilai ketinggian maupun interpolasi dibutuhkan beberapa operasi aritmatika dengan bilangan yang besar dan non-integer untuk setiap titik yang ada pada peta medan.

Masalah ini menjadi semakin terasa ketika tidak adanya waktu tersedia bagi program untuk melakukan *rendering* sebelum hasil ditampilkan ke pengguna, seperti pada program yang harus dapat menghasilkan bagian medan baru secara dinamis ketika pengguna mencoba melihat bagian tersebut.

II. RIUH PERLIN

A. Sejarah

Pada tahun 1985, Prof. Ken Perlin dari Departemen Ilmu Komputer New York University (NYU) mengembangkan suatu algoritma yang mampu menghasilkan grafik acak menggunakan bilangan-bilangan *pseudo-random* yang dapat diatur baik frekuensi maupun amplitudo grafiknya. Untuk penemuan ini, Prof. Perlin telah memenangkan berbagai penghargaan seperti *Academy Award for Technical Achievement* dari *Academy of Motion Picture Arts and Sciences* pada tahun 1997 untuk penggunaan algoritmanya dalam generasi konten prosedural dalam berbagai film.

B. Prinsip

Prinsip dasar dari algoritma riuh Perlin adalah pemetaan berbagai nilai dalam *x* pada *y* menggunakan suatu **fungsi riuh (*noise function*)**. Fungsi riuh yang dapat menghasilkan bilangan *pseudo-random* yang konsisten untuk setiap nilai *x* ini merupakan "jiwa" dari algoritma riuh Perlin karena fungsi inilah yang akan

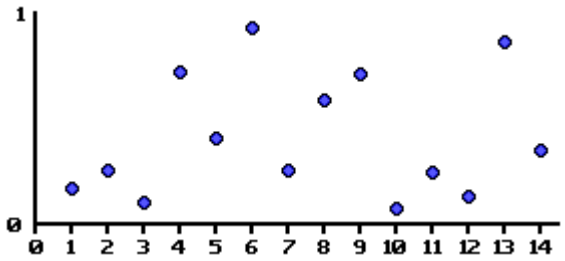
menentukan bentuk umum dari pemetaan riuh yang dihasilkan. Salah satu contoh fungsi riuh ini menggunakan operasi dengan bilangan-bilangan prima dan operasi *bitwise* untuk membatasi nilai pada suatu *range*.

```
function IntNoise(integer: x):float

    x ← (x<<13) ^ x;
    return ( 1.0 - ( (x * (x * x *
15731 + 789221) + 1376312589) &
7fffffff) / 1073741824.0);
```

persamaan 2.2.1: contoh pseudocode *fungsi riuh yang mengembalikan bilangan* pseudo-random

Pseudocode 2.2.1 memberi salah satu contoh penggunaan lima bilangan prima yang sangat besar untuk memberi kesan acak pada keluaran fungsi. Bilangan prima yang digunakan masing-masing dapat diganti dengan bilangan prima lain untuk merubah keluaran fungsi. Hal penting lainnya yang perlu diperhatikan di sini adalah bahwa selama nilai masukan, bilangan-bilangan prima yang digunakan, dan urutan operasi yang dilakukan sama, fungsi ini akan menghasilkan keluaran yang sama untuk setiap pemanggilannya. Ini berarti bahwa grafik hasil generasi fungsi ini akan selalu dapat direproduksi.

Variasi dari fungsi 2.2.1 akan menghasilkan grafik seperti berikut:

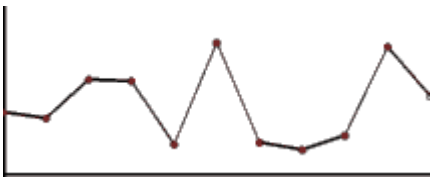


Gambar 2.2.1: contoh grafik hasil fungsi pseudo-random, *dengan masukan sebagai sumbu horizontal*

Masukan pada kebanyakan kasus (seperti koordinat pada peta) akan terbatas pada *range* tertentu. Seperti terlihat, dengan masukan-masukan diskrit seperti bilangan bulat maka hasilnya pun akan berupa grafik diskrit. Oleh karena itu titik-titik ini kemudian **diinterpolasi** untuk menentukan nilai-nilai di antaranya agar menghasilkan fungsi yang kontinyu.

Ada beberapa metode interpolasi yang dapat diurutkan berdasarkan kompleksitas dari yang paling sederhana ke yang paling rumit:

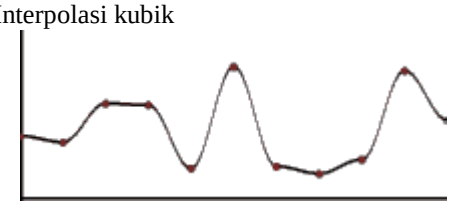
- Interpolasi linier



Gambar 2.2.2: contoh fungsi yang diinterpolasi linier

```
function Linear_Interpolate(a, b,
x):float
        return  a*(1-x) + b*x
```

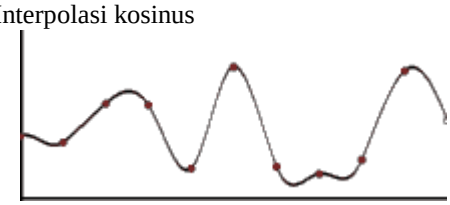
- Interpolasi kubik



Gambar 2.2.3: contoh fungsi yang diinterpolasi kubik

```
function Cubic_Interpolate(v0, v1, v2,
v3,x):float
        P = (v3 - v2) - (v0 - v1)
        Q = (v0 - v1) - P
        R = v2 - v0
        S = v1

        return Px^3 + Qx^2 + Rx + S
```

- Interpolasi kosinus

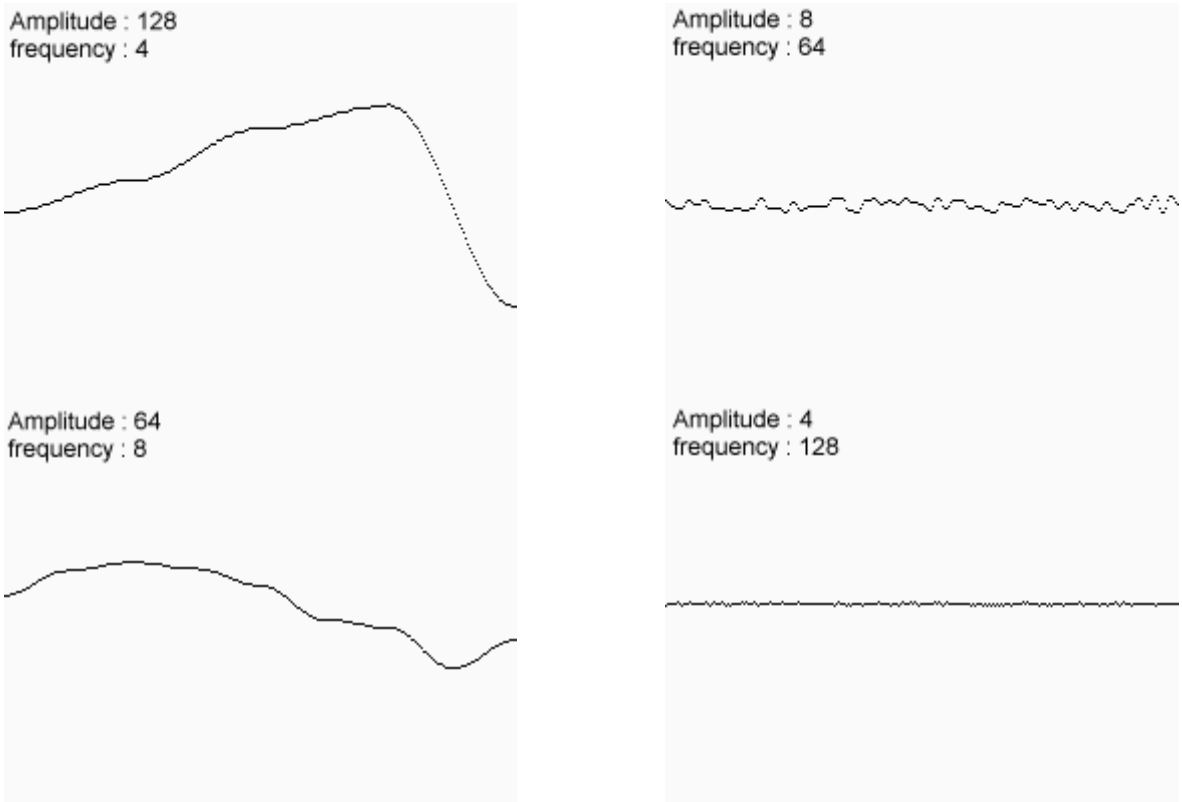


Gambar 2.2.4: contoh fungsi yang diinterpolasi kosinus

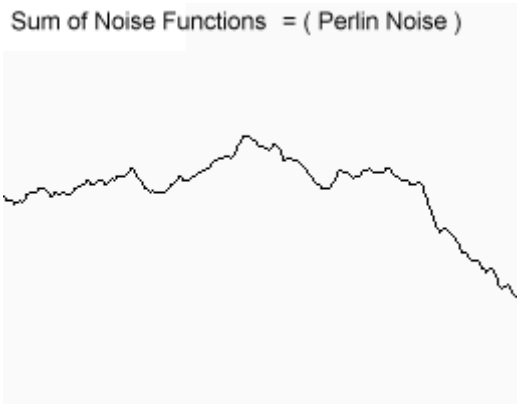
```
function Cosine_Interpolate(a, b,
x):float
        ft = x * 3.1415927
        f = (1 - cos(ft)) * .5

        return  a*(1-f) + b*f
```

Dapat dilihat bahwa kompleksitas fungsi interpolasi yang digunakan akan berpengaruh pada kemulusan grafik yang dihasilkan. Khususnya pada generasi medan, kemulusan ini sangat diinginkan untuk hasil yang realistis. Namun perlu diketahui juga bahwa kompleksitas akan mempengaruhi performa dari fungsi yang dijalankan.

Unsur penting selanjutnya dari Riuh Perlin adalah grafik hasil interpolasi selanjutnya dianggap sebagai gelombang, di mana grafik tersebut memiliki frekuensi dan amplitudo sedemikian hingga berbagai variasi grafik dengan frekuensi dan amplitudo yang berbeda-beda dapat ditambahkan satu sama lain layaknya fungsi sinus. Variasi frekuensi dan amplitudo dari suatu grafik disebut **oktaf**.

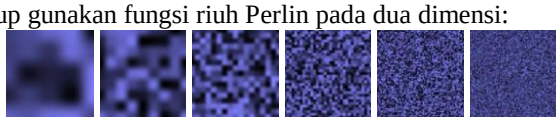


Gambar 2.2.3: berbagai variasi frekuensi dan amplitudo dari suatu fungsi riuh

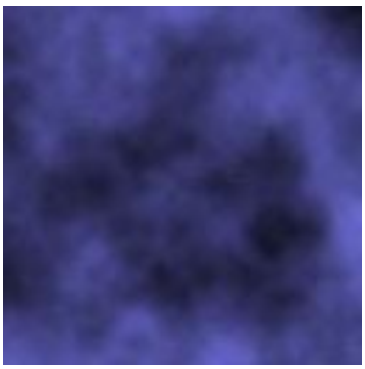


Gambar 2.2.4: hasil penjumlahan berbagai oktaf grafik di atas

Pada akhirnya, untuk membuat suatu peta elevasi, cukup gunakan fungsi riuh Perlin pada dua dimensi:



Gambar 2.2.4: implementasi fungsi Riuh Perlin dengan berbagai frekuensi dan amplitudo, dalam dua dimensi



Gambar 2.2.5: hasil penjumlahan enam oktaf di atas, menghasilkan peta elevasi final yang dapat digunakan untuk generasi medan

Singkatnya, algoritma fungsi riuh Perlin dalam dua dimensi adalah sebagai berikut:

```
function Noise1(integer x, integer
y):float
    n = x + y * 57
    n = (n<<13) ^ n;
    return ( 1.0 - ( (n * (n * n *
15731 + 789221) + 1376312589) &
7fffffff) / 1073741824.0);
  end function

function Interpolate(float a, float b,
float x):float
       return  a*(1-x) + b*x

  function InterpolatedNoise_1(float
x, float y):float

      integer_X    = int(x)
      fractional_X = x - integer_X

      integer_Y    = int(y)
      fractional_Y = y - integer_Y

      v1 = Noise1(integer_X,
integer_Y)
      v2 = Noise1(integer_X + 1,
integer_Y)
      v3 = Noise1(integer_X,
integer_Y + 1)
      v4 = Noise1(integer_X + 1,
integer_Y + 1)

      i1 = Interpolate(v1 , v2 ,
fractional_X)
      i2 = Interpolate(v3 , v4 ,
fractional_X)

      return Interpolate(i1 , i2 ,
fractional_Y)

  end function


  function PerlinNoise_2D(float x,
float y):float

      total = 0
      p = persistence
      n = Number_Of_Octaves - 1

      loop i from 0 to n

          frequency = 2^i
          amplitude = p^i

          total = total +
InterpolatedNoise_i(x * frequency, y *
frequency) * amplitude

      end of i loop

      return total
```

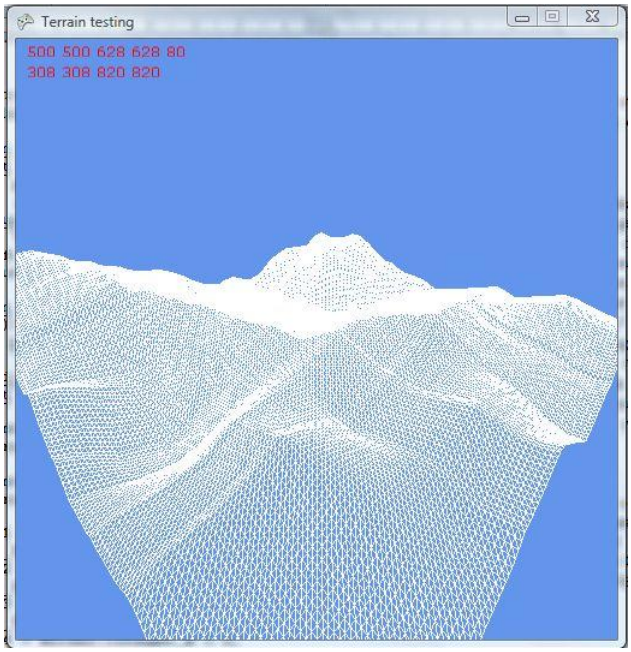
Fungsi **PerlinNoise_2D** di atas akan mengembalikan nilai elevasi pada titik x dan y yang dijadikan masukan.

C. Performa

Selanjutnya perlu diperhatikan kompleksitas algoritma riuh ini. Dapat kita asumsikan operasi generasi nilai *pseudo-random* dan interpolasi sebagai operasi inti. Apabila diketahui terdapat *n* bilangan pada satu sumbu di koordinat, sehingga terdapat n^2 titik pada peta yang ingin digenerasi, maka dilihat dari fungsi **InterpolatedNoise_1** terdapat $(7n)^2$ (3 operasi generasi + 4 operasi interpolasi) operasi yang diperlukan. Fungsi utama **PerlinNoise_2D** juga memerlukan generasi dalam beberapa oktaf untuk kemudian dijumlahkan. Dengan menyebut *i* sebagai jumlah oktaf, didapat total operasi yang diperlukan, yaitu $(7in)^2$, sehingga:

$$T(n) = (7in)^2 = O((in)^2)$$

Fungsi Perlin ini kemudian diuji coba untuk menghasilkan medan tiga dimensi pada komputer berprosesor Intel Core 2 Duo 2.00 GHz dengan RAM 2 GB dan memori VGA 256 MB.



Gambar 2.3.1: medan tiga dimensi hasil generasi fungsi Perlin

Hasilnya, dibutuhkan waktu sekitar **7,4 detik** untuk menghasilkan nilai elevasi untuk medan berukuran 512x512 *vertex* dengan 6 oktaf (iterasi 6 kali).

III. Optimasi

A. Teori

Untuk generasi medan dinamis, waktu yang dibutuhkan ini masih belum mencukupi. Program uji coba di atas memungkinkan penggunanya untuk melakukan *scrolling* terhadap medan yang ditampilkan, sehingga nilai elevasi untuk koordinat-koordinat yang akan ditampilkan harus dihitung secara konstan. Pada program uji coba, yang terjadi adalah program mengalami *fatal error* ketika medan terus di-*scrolling*, karena pada akhirnya harus ditampilkan koordinat-koordinat tertentu yang nilai ketinggiannya belum sempat dihitung.

Salah satu ide optimasi adalah dengan membagi-bagi pekerjaan penghitungan medan ini dalam beberapa *thread*, sehingga penghitungan berbagai sektor dapat dilakukan secara konkuren. Contohnya suatu medan 512x512 dapat dibagi menjadi 64x64 per *thread*. Maka dari hal inilah digunakan metode *divide and conquer*.

Pada harapannya, berdasarkan perhitungan kompleksitas ini berarti bahwa setiap *thread* hanya akan mengerjakan $(n/4)^2$ titik apabila terdapat 4 *thread*, dan seterusnya.

Sementara itu, batasan dari solusi ini adalah karena penggunaan *thread* yang terlalu banyak bukanlah praktek yang baik dan tidak akan memberi perubahan drastis apabila tetap diproses pada CPU yang sama, jumlah *thread* yang akan dibuat pun harus ditentukan dari awal sebagai basis dari algoritma *divide and conquer*.

Karena implementasi *threading* berbeda-beda antar bahasa, berikut adalah contoh implementasi *divide and conquer* untuk fungsi riuh dalam bahasa C#:

```
public void calculateHeightBatch()
        {
            #region THREADED
            Thread[] T = new Thread[jobnum];
            for (int i = 0; i <
(int)Math.Sqrt(jobnum); i++)
            {
                for (int j = 0; j <
(int)Math.Sqrt(jobnum); j++)
                {
                    W[i *
(int)Math.Sqrt(jobnum) + j] = new Work(this, i
* (512 / (int)Math.Sqrt(jobnum)),
                                (i + 1) * (512 /
(int)Math.Sqrt(jobnum)), j * (512 /
(int)Math.Sqrt(jobnum)),
                                (j + 1) * (512 /
(int)Math.Sqrt(jobnum)));
                    T[i *
(int)Math.Sqrt(jobnum) + j] = new Thread(W[i *
(int)Math.Sqrt(jobnum) + j].workBatch);
                    T[i *
(int)Math.Sqrt(jobnum) + j].IsBackground =
true;
                    T[i *
(int)Math.Sqrt(jobnum) + j].Start();
                }
            }
            #endregion
        }
```

Di mana `jobnum` adalah jumlah *thread*, dan setiap *thread* menjalankan fungsi riuh Perlin seperti *pseudocode* pada bab II.

B. Hasil

Hasil dari penggunaan *threading* dan *divide and conquer* ini, penghitungan peta elevasi pada kondisi yang sama dengan percobaan pada bab II (ukuran 512x512 *vertex*, 6 oktaf) dan dengan perangkat keras yang juga sama (prosesor Intel Core 2 Duo 2.00 GHz, RAM 2 GB, memori VGA 256 MB) menjadi memerlukan waktu sekitar **4,7 detik** dengan penggunaan 4 *thread*. Walaupun masih belum memadai untuk *scrolling* medan dinamis, peningkatan yang besar sudah dapat dilihat.

IV. Kesimpulan

Penggunaan pendekatan *divide and conquer* disertai dengan *thread* dapat digunakan untuk optimasi untuk pekerjaan yang melakukan operasi yang serupa pada suatu senarai besar seperti generasi medan prosedural. Dalam implementasinya, masih dibutuhkan optimasi lebih lanjut

lagi seperti menggunakan suatu algoritma untuk memperkirakan koordinat mana saja yang akan diperlukan untuk ditampilkan, agar jumlah koordinat baru yang dikerjakan menjadi lebih sedikit dan fungsi riuh menjadi lebih cepat. Perlu diketahui juga bahwa menggunakan *thread* yang banyak tidak selalu berarti akan lebih cepat, karena pada dasarnya *thread-thread* yang ada tetap diproses satu persatu oleh CPU, namun tidak menunggu satu sama lain untuk selesai sehingga memberikan kesan konkurensi.

REFERENSI

[1] Munir, Rinaldi. 2007. *Strategi Algoritmik*. Bandung : Teknik Informatika ITB.

[2] http://freespace.virgin.net/hugo.elias/models/m_perlin.htm

[3] http://mrl.nyu.edu/perlin/

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, 29 April 2010

ttd

Satrio Dewantono
13509051