Pendekatan Program Dinamis dalam Simulasi Pencahayaan Voxel Berbasis Floodfill untuk Global Illumination Rendering

Fachriza Ahmad Setiyono - 13523162
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
E-mail: rizacal.mamen@gmail.com , 13523162@std.stei.itb.ac.id

***Abstrak*—Penelitian ini mengusulkan pendekatan baru untuk *global illumination* (GI) rendering menggunakan program dinamis melalui simulasi pencahayaan berbasis voxel dan algoritma floodfill. Adegan 3D direpresentasikan dalam model voxel, di mana informasi pencahayaan (irradiance) dipropagasi dari voxel sumber cahaya ke voxel tetangga menggunakan floodfill. Untuk mengoptimalkan biaya komputasi, propagasi irradiance dilakukan secara iteratif, dengan satu iterasi floodfill per frame, memanfaatkan data dari iterasi floodfill frame sebelumnya. Pendekatan ini secara signifikan mengurangi beban komputasi sambil tetap mencapai GI. Data *irradiance* yang dihasilkan dan disimpan dalam voxel ini secara khusus digunakan untuk mengambil sampel *secondary rays* dalam proses *ray tracing*. Strategi ini bertujuan untuk mengurangi jumlah sampel dan *noise* yang signifikan setelah pantulan pertama. Implementasi ini menunjukkan efektivitas dalam menyeimbangkan kualitas visual dan efisiensi komputasi untuk rendering GI secara *real-time*.**

Keywords—Program Dinamis, Floodfill, Global Illumination, Voxel

I. Pendahuluan

Simulasi pencahayaan realistis *real-time* dalam bidang *computer graphics* (CG) merupakan salah satu tantangan utama yang membutuhkan pendekatan algoritma yang efisien. Teknik seperti path tracing dan physically based rendering (PBR) menawarkan hasil visual yang sangat akurat, namun memiliki biaya komputasi yang tinggi. Untuk mengurangi kompleksitas tersebut, digunakan pendekatan estimasi seperti *irradiance caching*, di mana pencahayaan tidak langsung (*indirect lighting*) diaproksimasi dari hasil perhitungan sebelumnya.

Global Illumination (GI) adalah aspek fundamental dalam grafika komputer yang bertujuan untuk mensimulasikan bagaimana cahaya berinteraksi dengan objek dalam suatu adegan, termasuk pantulan, refraksi, dan penyebaran, untuk menghasilkan citra yang realistis dan imersif. Tantangan utama dalam mencapai GI yang akurat terletak pada sifat komputasinya yang sangat intensif. Secara tradisional, metode seperti *path tracing* atau radiosity mampu menghasilkan hasil yang akurat, namun seringkali membutuhkan waktu render yang sangat lama, sehingga tidak cocok untuk aplikasi *real-time* seperti video game atau simulasi interaktif. Kebutuhan akan kualitas visual yang tinggi berbanding terbalik dengan tuntutan kinerja real-time.

Gambar 1. Ilustrasi global illumination pada Cornell box.

(Source: https://en.wikipedia.org/wiki/Cornell_box)

Penelitian ini mengeksplorasi pendekatan alternatif menggunakan voxel-based floodfill lighting, di mana informasi cahaya dipropagasi dari sumber cahaya ke seluruh ruang voxel menggunakan metode floodfill. Pendekatan ini secara struktur memenuhi prinsip-prinsip algoritma program dinamis, terutama dalam hal pemecahan masalah menjadi submasalah yang saling tumpang tindih, serta penggunaan penyimpanan hasil (memoisasi) untuk menghindari perhitungan yang redundan.

II. Dasar Teori

A. Program Dinamis

Program dinamis (*dynamic programming*) adalah strategi algoritmik yang digunakan untuk menyelesaikan masalah dengan cara memecah masalah menjadi submasalah yang lebih kecil, menyimpan hasil dari submasalah tersebut, dan membangunnya kembali menjadi solusi akhir. Dengan

program dinamis, solusi persoalan dapat dipandang sebagai serangkaian keputusan yang saling berkaitan.

Pada program dinamis, rangkaian keputusan yang optimal dibuat dengan menggunakan prinsip optimalitas. Prinsip optimalitas menyatakan bahwa jika kita bekerja dari tahap k ke tahap $k + 1$, kita dapat menggunakan hasil optimal dari tahap k tanpa harus kembali melakukan proses di tahap awal [1].

Persoalan program dinamis memiliki beberapa karakteristik:

- Persoalan dapat dibagi menjadi beberapa tahap, yang pada setiap tahap hanya diambil satu keputusan.
- Masing-masing tahap terdiri dari sejumlah status.
- Hasil dari keputusan yang diambil pada setiap tahap ditransformasi ke status berikutnya pada tahap berikutnya.
- Hubungan rekursif yang mengidentifikasikan keputusan terbaik tahap k memberikan keputusan terbaik untuk status pada tahap $k + 1$.
- Prinsip optimalitas berlaku pada persoalan.

Terdapat dua pendekatan pada persoalan program dinamis, yaitu pendekatan maju dan mundur.

1) Program dinamis maju *(forward* atau *top-down*)

Pendekatan dinamis maju bergerak mulai dari tahap 1, terus maju ke tahap 2, 3, dan seterusnya sampai tahap *n*.

Rangkaian peubah keputusan adalah $x_1, x_2, ..., x_n$.

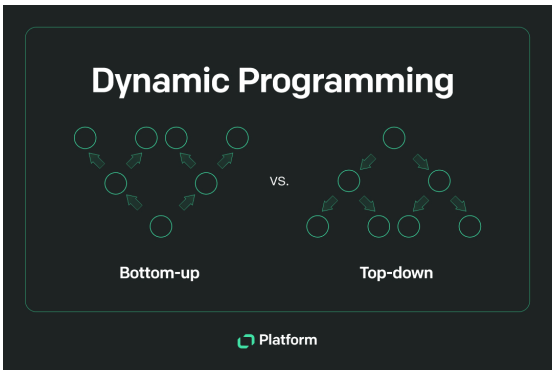
2) Program dinamis mundur *(backward* atau *bottom-up*)

Pendekatan dinamis mundur bergerak mulai dari tahap *n*, terus mundur ke tahap $n - 1$, $n - 2$, dan seterusnya sampai tahap 1.

Rangkaian peubah keputusan adalah $x_n, x_{n-1}, ..., x_1$.

Untuk mengembangkan algoritma program dinamis, dapat dilakukan langkah-langkah berikut:

- Tentukan karakteristik struktur solusi optimal.
- Definisikan nilai solusi optimal secara rekursif.
- Hitung nilai solusi optimal secara maju atau mundur
- Rekonstruksi solusi optimal (opsional).



Gambar 2. Dua pendekatan berbeda pada program dinamis.

(Source: https://platform.text.com/resource-center/updates/what-is-dynamic-programming)

B. Global Illumination

Global Illumination mengacu pada simulasi interaksi cahaya yang lebih kompleks dalam suatu adegan 3D dibandingkan dengan model pencahayaan langsung (*direct lighting*). Pencahayaan langsung hanya mempertimbangkan cahaya langsung dari sumber cahaya ke permukaan. GI memperhitungkan bagaimana cahaya memantul secara tidak langsung dari satu permukaan ke permukaan lain (pantulan diffuse, specular, refraction, subsurface scattering), sehingga menciptakan efek pencahayaan yang lebih realistis seperti warna yang dipantulkan, bayangan akurat, dan kaustik.

Persamaan dasar yang sering digunakan untuk mendeskripsikan transfer cahaya dalam GI adalah "The Rendering Equation", yang dirumuskan oleh Kajiya [2]:

$$L_o(x, \omega_o) = L_e(x, \omega_o) + \int_{\Omega} f_r(x, \omega_i, \omega_o) L_i(x, \omega_i)(n \cdot \omega_i) d\omega_i \quad (1)$$

Di mana:

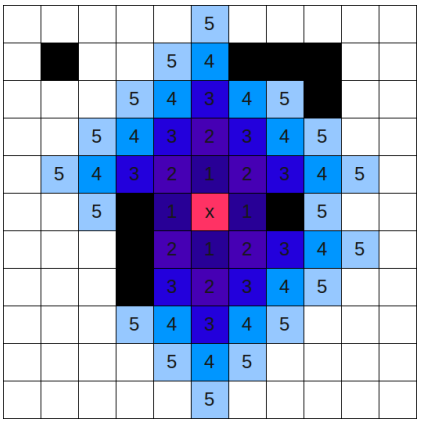
x : posisi di ruang 3D
ω_o : arah keluar cahaya
ω_i : arah masuk cahaya
n : normal bidang di x
L_o : *radiance* ke arah ω_o
L_i : *radiance* dari arah ω_i
f_r : Bidirectional Reflectance Distribution Function (BRDF)

Perhatikan bahwa L_i di dalam bagian integrasi merupakan cahaya yang dikeluarkan oleh posisi lain *y* di ruang 3D dengan arah ω_i, yang harus dicari dengan $L_o(y, \omega_i)$. Perhitungan integral yang berulang ini yang membuat kalkulasi GI secara *real-time* sulit dicapai.

C. Floodfill

Algoritma floodfill digambarkan dengan konsep air yang selalu mengalir dari tempat yang lebih tinggi ke tempat yang

lebih rendah [3]. Konsep ini diterapkan dengan memberikan setiap unit sel suatu nilai, Kemudian, sel dengan nilai yang lebih rendah dianggap memiliki ketinggian lebih rendah, dan sebaliknya sel dengan nilai tinggi memiliki ketinggian yang lebih tinggi. Representasi nilai ini bisa dibalik, tergantung pada kebutuhan dan tujuan algoritma. Secara alami, air akan mengalir dari tempat yang tinggi ke tempat yang lebih rendah. Ini adalah konsep dasar algoritma flood fill.



Gambar 3. Algoritma flood fill pada unit sel grid 2D.

(Source: https://stackoverflow.com/questions/8120179/how-many-moves-to-reach-a-destination-efficient-flood-filling)

Algoritma ini cocok dimodelkan dengan persamaan rekursif, dimana bila fungsi flood fill dilakukan pada unit *x*, maka akan memanggil fungsi floodfill pada unit-unit tetangganya. Algoritma floodfill umumnya dapat dinotasikan sebagai berikut:

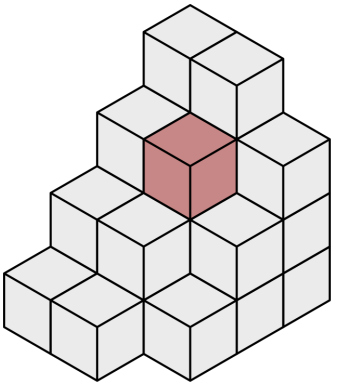
```
FloodFill (node):
    if node is not Inside then return.
    set the node.
    for each neighboring unit i:
        perform FloodFill(i)
    return.
```

Penggunaan algoritma ini umumnya dipakai pada permasalah pencarian jalur keluar pada maze. Dengan algoritma floodfill, akan terlihat jalur terpendek yang berujung pada jalur keluar maze. Namun pada konteks ini, flood fill dapat digunakan untuk memprogasi data cahaya.

Algoritma flood fill yang biasa diimplementasi dalam 2 dimensi dapat dengan mudah dikembangkan menjadi 3 dimensi. Untuk keperluan 3 dimensi, algoritma flood fill dapat dilakukan pada unit satuan voxel. Tentunya dengan menambah dimensionalitas berpengaruh pada jumlah unit tetangga yang ada.

D. Voxel

Voxel (*volumetric pixel*) adalah model tiga dimensi yang merepresentasikan informasi volume, berbeda dengan *mesh* yang hanya merepresentasikan permukaan suatu objek [4]. Proses pengkodean suatu ruang tiga dimensi menjadi voxel disebut *voxelization*.



Gambar 4. Representasi ruang 3D dengan voxel.

(Source: https://en.wikipedia.org/wiki/Voxel)

Representasi ruang berbasis voxel digunakan untuk menyimpan informasi spasial dan pencahayaan dalam grid 3D diskrit. Setiap voxel adalah unit volume kecil dalam ruang 3D, mirip dengan piksel dalam gambar 2D. Dalam sistem kami, setiap voxel menyimpan warna cahaya yang akan dipropagasi.

Keuntungan penggunaan voxel mencakup:

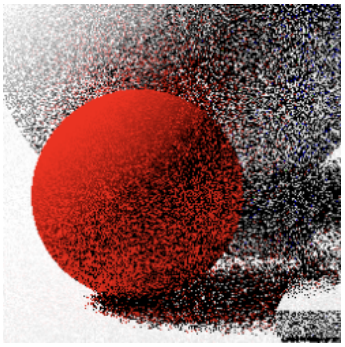
- Struktur yang seragam
  Memungkinkan traversal dan akses data yang cepat untuk algoritma berbasis grid seperti floodfill.
- Data volumetrik
  Memungkinkan penyimpanan informasi cahaya di dalam volume ruang, bukan hanya di permukaan, yang berguna untuk efek seperti scattering atau pantulan di ruang tertutup (*multibounce*).
- Diskrit
  Menyederhanakan masalah propagasi cahaya menjadi perhitungan antara sel-sel diskrit.

III. Algoritma

Sebelum mengembangkan algoritma, mari pahami dulu apa yang ingin kita selesaikan dari permasalahan yang ada. Ingat kembali bahwa pada "The Rendering Equation", nilai cahaya atau *radiance* L_o memerlukan integrasi semua nilai L_o dari titik yang terlihat dari *x* dan terarah ke *x*. Algoritma yang umum digunakan untuk menyelesaikan persamaan tersebut adalah dengan Monte Carlo path tracing, dimana untuk tiap titik *x*, dipilih sampel ω_i acak di belahan *hemisphere* atas normal bidang *x* untuk mencari nilai L_o di titik lain *y*.

Pada implementasi yang membutuhkan kinerja *real-time*, biasanya hanya digunakan 1 sampel per *frame* dan sekitar 2 pantulan yang dihitung. Ada juga algoritma *russian-roulette* yang dapat memotong *path* secara acak untuk menghemat waktu komputasi. Namun. pada dasarnya, penggunaan sampel

yang sangat kecil ini berpengaruh buruk pada kualitas GI yang dihasilkan, yaitu terhadap banyaknya *noise* yang muncul.



Gambar 5. Noise pada Monte Carlo path tracing.

(Source: https://en.wikipedia.org/wiki/Path_tracing)

Dalam konteks ini, algoritma propagasi seperti floodfill bekerja jauh lebih efektif daripada algoritma *gathering* atau sampling (seperti Monte Carlo path tracing), sehingga secara efektif dapat mengurangi *noise* yang muncul di citra akhir.

Algoritma floodfill yang dikembangkan dapat mengurangi *noise* dengan melakukan penyimpanan aproksimasi *irradiance* pada setiap **voxel udara/kosong** atau **transparan** yang ada di ruang. Nantinya saat dilakukan proses Monte Carlo ray tracing untuk pantulan kedua (*secondary ray*), *irradiance* voxel yang ada di *hemisphere* atas titik interseksi akan digunakan sebagai L_o.

Algoritma floodfill yang dikembangkan dapat dijelaskan dengan pendekatan program dinamis sebagai berikut:

A. *Karakteristik Struktur Solusi Optimal*

"Solusi optimal" yang dicari bukanlah nilai tunggal, melainkan distribusi *irradiance* yang stabil di representasi voxel ruang 3D. Setiap voxel akan menyimpan nilai *irradiance* terakumulasi yang menggambarkan cahaya yang telah sampai padanya dari semua sumber, baik langsung maupun tidak langsung.

Solusi dibangun secara bertahap melalui iterasi. Setiap iterasi (atau frame) akan membawa distribusi *irradiance* lebih dekat ke keadaan konvergen. *Irradiance* yang optimal adalah keadaan di mana perubahan *irradiance* antar-frame menjadi sangat minimal.

Pada persoalan ini, tahap (k) adalah proses menghitung *irradiance* untuk digunakan sebagai perhitungan di tahap (frame) berikutnya. Status (s) adalah *irradiance* yang dipertahankan tiap voxel.

B. *Definisi Rekursif Solusi Optimal*

Misal $I(x,t)$ adalah nilai *irradiance* voxel di x pada frame/tahap t, α adalah suatu faktor *falloff* cahaya, dan $V(x,i)$ adalah nilai visibilitas antar voxel di x dan i, maka dapat dirumuskan solusi rekursif sebagai berikut:

$$I(x,t+1) = \alpha \times \frac{\sum_{i \in neighbor}^{N} I(i,t) \times V(x,i)}{N},\ 0 < \alpha$$

Nilai visibilitas $V(x,i)$ mendefinisikan apakah voxel di x dan i bisa saling melihat dan memprogasi *irradiance* atau tidak. Sedangkan faktor redaman cahaya α dapat digunakan untuk mengatur intensitas propagasi.

C. *Perhitungan Nilai Solusi Optimal Secara Maju*

Karena perhitungan GI dimulai dari frame 0 hingga n, maka masuk akal jika perhitungan dilakukan secara maju. Setiap *irradiance* voxel pada frame saat ini bergantung pada *irradiance* voxel tetangga dari frame sebelumnya.

1) Inisiasi $(t = 0)$

Untuk tiap voxel, tetapkan nilai *irradiance* menjadi nilai emisivitas voxel tersebut. Nilai emisivitas bergantung pada material voxel (misal lampu memiliki nilai emisivitas, sedangkan batu tidak).

2) Iterasi tiap tahap *k* hingga *n*

Gunakan dua *buffer*, satu untuk menyimpan data *irradiance* tahap k dan satunya untuk menyimpan *irradiance* tahap $k-1$.

Untuk tiap voxel, hitung nilai *irradiance* baru (k) menggunakan rumus rekursif yang telah didefinisikan sebelumnya. Pastikan nilai *irradiance* voxel tetangga diambil dari *buffer* tahap $k-1$. Kemudian simpan nilai *irradiance* di *buffer* tahap k. Pada iterasi selanjutnya, nilai *irradiance* baru ini akan menjadi nilai *irradiance* yang lama.

D. *Rekonstruksi Solusi Optimal*

Di saat yang bersamaan dengan propagasi, dilakukan proses ray tracing satu sampel di ruang 3D. Namun, ketika *ray* pantulan pertama mengenai suatu objek, maka diambil nilai *irradiance* voxel yang ada di sekitar (*hemisphere* atas) objek tersebut. Proses sampling nilai *irradiance* ini menggantikan dan mengaproksimasi bagian integrasi dalam "The Rendering Equation".

Rekonstruksi ini bukanlah langkah terpisah dalam perhitungan program dinamis, melainkan bagaimana hasil dari program dinamis (yaitu, grid *irradiance* yang terisi) dimanfaatkan oleh bagian lain dari GI rendering.

IV. Implementasi

Algoritma yang dikembangkan diimplementasikan sebagai shader dalam permainan Minecraft menggunakan Iris Shaders mod. Pilihan platform ini memungkinkan eksplorasi propagasi cahaya dinamis dalam dunia voxel yang masif dan berubah secara interaktif, sekaligus memanfaatkan kapabilitas rendering GPU secara langsung. Implementasi ini akan merinci struktur dan proses voxelisasi, logika algoritma floodfill yang berjalan per frame, serta bagaimana hasil *irradiance* yang terkomputasi digunakan oleh ray tracing untuk *secondary rays* untuk menghasilkan efek GI yang meyakinkan dengan performa yang optimal.

Sebagai pengantar singkat, *pipeline* shader Minecraft menggunakan Iris Shader pada umumnya sama dengan *pipeline* shader lain. Iris Shader akan menyediakan fungsionalitas (atau API) bagi penulis shader untuk memodifikasi metode rendering Minecraft. Tipe shader yang digunakan untuk implementasi ini adalah vertex dan geometry shader untuk proses voxelisasi, dan fragment shader untuk algoritma GI.

A. *Voxelisasi Ruang*

Walaupun Minecraft adalah game voxel, tetapi penulis harus tetap merubah data vertex yang disediakan menjadi representasi voxel. Karena keterbatasan versi Iris Shader yang digunakan, voxel harus direpresentasikan dalam sebuah *buffer* 2D. Sehingga perlu dilakukan *mapping* posisi 3D → 2D.

```
ivec2 GetVoxelStoragePos(ivec3 voxelIndex) { // in pixels/texels
    int offset = (320 - int(cameraPosition.y)) / 2 - 64;
    return (voxelIndex.xz + (int(MC_SHADOW_QUALITY * shadowMapResolution) / 32)) * ivec2(8, 16) + ivec2((voxelIndex.y + offset) / 16,
           (voxelIndex.y + offset) % 16);
}
```

Gambar 6. Kode mapping 3D → 2D.

(Source: dokumentasi penulis)

B. *Voxel Floodfill*

Di setiap kelipatan n frame, sebuah fragment shader akan melakukan iterasi floodfill. Kelipatan n dipilih sesuai keinginan untuk menyeimbangkan responsivitas floodfill dengan performa. Untuk implementasi ini, dipilih $n = 3$.

Proses floodfill diimplementasi sesuai rumus rekursif yang telah dijabarkan sebelumnya. `sampleVoxelContribution` adalah data *irradiance* dari suatu voxel tetangga, dan `visibility` adalah fungsi visibilitas $V(x, i)$. Faktor *falloff* α yang dipilih setelah dilakukan beberapa percobaan adalah 2.43902.

Metode floodfill yang digunakan mengikut implementasi milik Balint pada situs *shadertoy* [5], dimana sampel warna tetangga diubah dari *linear space* ke *gamma space* terlebih dahulu, sebelum diubah kembali di akhir.

```
vec3 samples = vec3(0.0);

//balint floodfill method
for(int i = 0; i < 6; ++i) {

    if(hitNoncubic(id, offset[i])) continue;

    ivec2 previousSampleVoxelUV = GetPreviousVoxelStoragePos(prevVoxelPos+offset[i], previousCameraPosition.y);
    ivec2 sampleVoxelUV = GetVoxelStoragePos(voxelPos+offset[i]);

    vec4 sampleVoxel = unpackUnorm4x8(texelFetch(shadowcolor1, sampleVoxelUV, 0).x);

    sampleVoxel.rgb /= PI;

    int sampleVoxelID = int(0.5 + 255.0 * sampleVoxel.a);

    bool nonCubic = (sampleVoxelID >= 4 && sampleVoxelID <= 66)||(sampleVoxelID >= 249 && sampleVoxelID <= 252);
    bool translucent = (sampleVoxelID == 201);

    vec3 previousSampleVoxel = (texelFetch(colortex2, previousSampleVoxelUV, 0).rgb);

    vec3 sunContribution = sunVisibility * (length(sampleVoxel.rgb) <= 0.0 ? vec3(0.0) : sampleVoxel.rgb) * sun
                         * float(max0(dot(normalize(-offset[i]), sunVector))) * sunTransmittance;
    vec3 skyContribution = float(upVis) * (length(sampleVoxel.rgb) <= 0.0 ? vec3(1.0) : sampleVoxel.rgb) * sky;

    vec3 visibility = mix(vec3(1.0), sampleVoxel.rgb, float(translucent)) * float(!hitNoncubic(sampleVoxelID, -offset[i]));
    vec3 sampleVoxelContribution = previousSampleVoxel * visibility;

    samples += pow(sunContribution + skyContribution + sampleVoxelContribution, vec3(2.2));
}
floodFillData.rgb =  (pow(samples, vec3(1.0/2.2)) / 6.0) * falloff;
```

Gambar 7. Kode sampling voxel tetangga untuk floodfill.

(Source: dokumentasi penulis)

C. *Irradiance Sampling*

```
// dalam loop ray tracing
getMaterial(hitMat, hitPos, hitNormal, voxel);

#ifdef IrradianceCaching
    vec3 cache = texelFetch(colortex2, GetVoxelStoragePos(ivec3(floor(hitPos+randomHemisphereVector(hitNormal)+0.5))), 0).rgb  ;
#else
    vec3 cache = vec3(0.0);
#endif

vec3 feetHitPos = VoxelSpaceToSceneSpace(hitPos);

vec3 sunLight = (calculateShadow(shadowtex1, feetToShadowScreen(feetHitPos+hitMat.normal*EPS ), 0.0003)
              * sun * sunTransmittance) * max0(dot(hitMat.normal, normalize(sunPosition)));

float dist = distance(hitPos, diffuseOrigin);
diffuse += ( (hitMat.albedo * ((hitMat.emission+sunLight)/max(1.0, pow2(dist)) + (cache))) ) * t * accColor;
```

Gambar 8. Kode sampling irradiance hasil floodfill.

(Source: dokumentasi penulis)

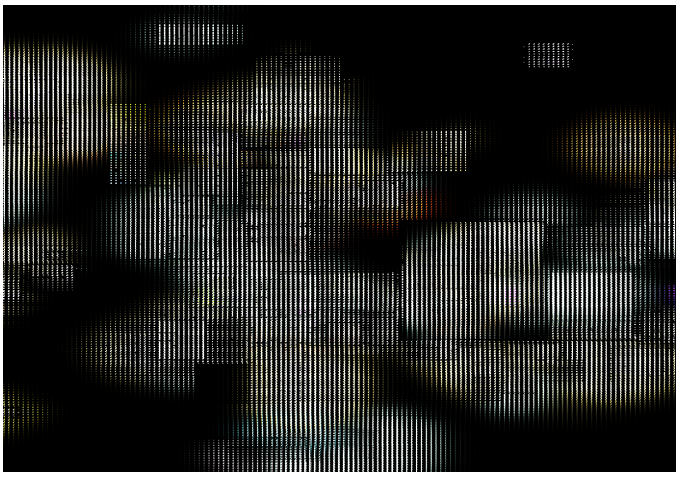
Pada satu *fullscreen pass*, dilakukan proses ray tracing yang dimulai langsung di ruang 3D. Ini berarti tidak dilakukan proses tracing dari kamera (*primary rays*), melainkan langsung dari *secondary rays* menggunakan data yang disediakan *GBuffer*.Jika algoritma digunakan (dinyalakan melalui opsi shader), maka diambil nilai `cache`, yaitu nilai *irradiance* di sekitar titik sampel yang diambil dari grid *irradiance* hasil floodfill. Digunakan sampel acak arah *hemisphere* berorientasi normal untuk mengurangi efek "*blocky*".

V. Hasil dan Analisis

Implementasi diuji dalam Minecraft Java versi 1.20.1 menggunakan Iris Shader. Spesifikasi sistem uji adalah platform Windows 11 dengan AMD Ryzen 9 6900HS 8 Core CPU dan NVIDIA GeForce RTX 3070 Ti mobile GPU.

A. *Hasil*

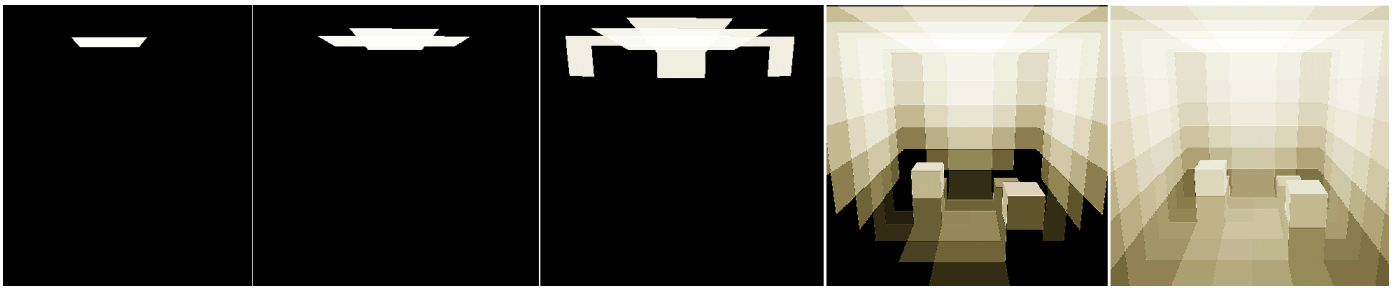
Pertama, mari kita lihat apakah algoritma yang dikembangkan berhasil menghitung dan mengaproksimasi *irradiance* pada setiap voxel. Hasil floodfill yang dilakukan dalam 2D terlihat berhasil dalam mempropagasi data cahaya ke voxel sekitarnya, seperti di gambar 9.



Gambar 9. Buffer 2D penyimpanan irradiance voxel

(Source: dokumentasi penulis)

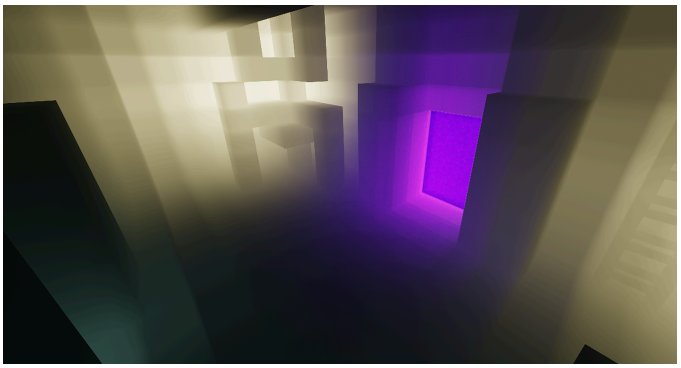
Melihat kumpulan *slice* 2D dari suatu representasi ruang 3D memang relatif sulit. Untuk lebih mudah melihat hasilnya, kita coba lihat grid *irradiance* di ruang 3D dengan



Gambar 10. Visualisasi iterasi floodfill pada berbagai tahap di Cornell Box, mulai dari tahap 1.

(Source: dokumentasi penulis)

men-sampel *irradiance* voxel yang dilalui oleh *ray* dari kamera ke sepanjang ruang.



Gambar 11. Grid irradiance dalam ruang 3D.

(Source: dokumentasi penulis)

Terlihat lebih jelas bahwa cahaya terpropagasi dari sumber cahaya ke ruang 3D, tepatnya ke voxel-voxel udara dan transparan.

Untuk melihat bagaimana iterasi floodfill bekerja, dapat dilakukan analisis di tiap tahapan iterasi floodfill seperti yang terlihat di gambar 10 Pada tahap 1, terlihat hanya voxel sumber cahaya tampak memiliki nilai cahaya. Di beberapa tahap selanjutnya, nilai cahaya terpropagasi ke voxel sekitar, hingga akhirnya di tahap yang kesekian, distribusi *irradiance* mencapai konvergensi dan menjadi stabil. Ini menunjukkan bahwa pendekatan program dinamis untuk perhitungan *irradiance* berhasil bekerja.

Hal yang perlu diperhatikan adalah kita tidak melakukan sampel material dari voxel non-udara yang disentuh oleh propagasi floodfill. Algoritma ini hanya menghitung intensitas cahaya yang diterima suatu voxel udara, bukan cahaya yang dikeluarkan oleh voxel tersebut. Pen-sampelan material dilakukan pada proses ray tracing, dengan mengalikan albedo dengan *irradiance* yang didapat dari grid *irradiance*.

Selanjutnya, kita lihat apakah algoritma yang dikembangkan mampu menghasilkan kualitas GI yang *believable* dengan performa yang optimal. Untuk pengecekan kualitas GI, dilakukan uji coba pada adegan Cornell Box. Ruang Cornell Box terdapat pada map “le_ultimate_rtx” yang dibuat oleh komunitas Minecraft ShaderLABS.

Perhatikan pada gambar 12. tepatnya pada gambar a). Gambar a) adalah citra yang dihasilkan hanya dengan *direct lighting*. Terlihat pada inset gambar a) bahwa tidak ada pantulan warna hijau dan merah yang tampak pada objek. Hal ini membuat adegan tampak tidak realistis. Kualitas ini dicapai dengan 30 fps.

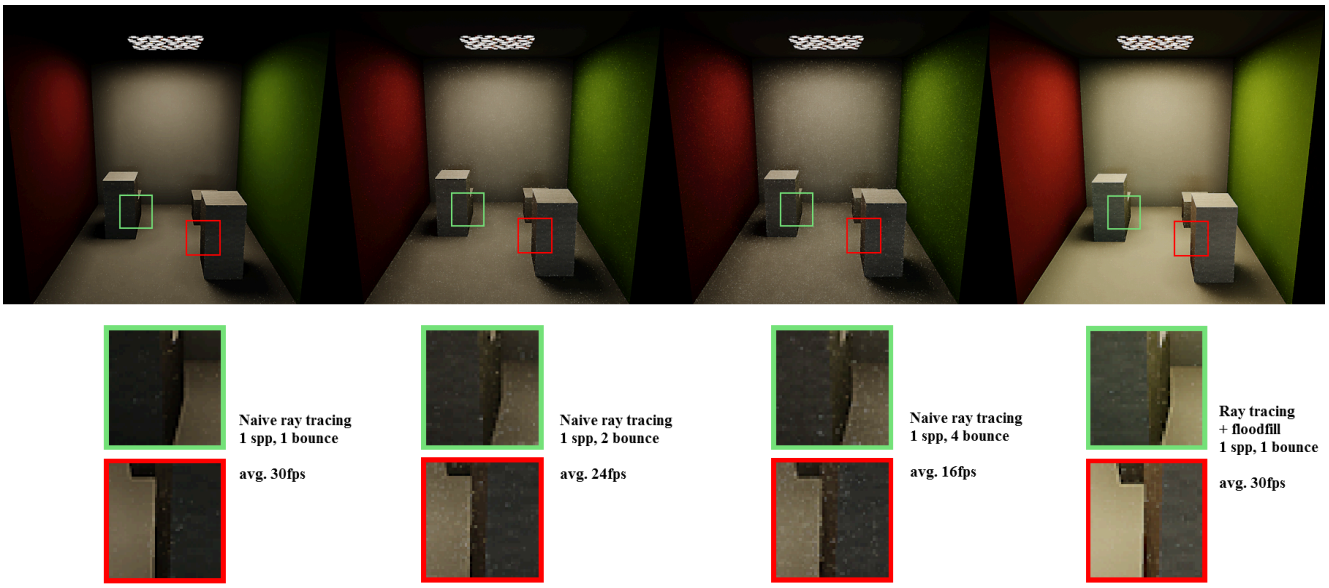
Selanjutnya perhatikan pada gambar b) dan c), terlihat pada inset bahwa pantulan warna merah dan hijau tampak pada objek. Untuk mencapai GI tersebut, dibutuhkan jumlah pantulan yang lebih tinggi, yaitu 2 dan 4 pantulan. Namun, seiring naiknya parameter pantulan, performa menjadi semakin buruk, yaitu 24 fps dan 16 fps. Jadi penurunan performa bisa mencapai setengah dari jumlah fps yang didapat pada gambar a). Tidak lupa juga *noise* yang lebih terlihat seiring naiknya jumlah pantulan. Hal ini tentunya tidak diinginkan untuk kinerja *real-time*.

Terakhir pada gambar d), dengan algoritma floodfill yang dikembangkan, program mampu mencapai citra GI yang meyakinkan dengan performa yang serupa dengan 1 sampel per piksel dan 1 pantulan, yaitu 30 fps. Pantulan warna hijau dan merah terlihat jelas pada inset. *Noise* yang terlihat juga relatif lebih sedikit dari gambar-gambar sebelumnya. Walaupun voxel yang digunakan cukup besar sebagai representasi ruang, tidak terlihat efek “blocky” pada gambar.

B. Analisis

Dari hasil pengujian yang telah dilakukan, algoritma yang dikembangkan berhasil mencapai kualitas GI yang menyakinkan dengan performa yang optimal. Namun, perlu diingat bahwa pada metode ini, diperlukan memori yang lebih besar untuk menyimpan representasi ruang dengan voxel. Pada konteks ini, penambahan memori hanya terjadi saat menambahkan *buffer* untuk penyimpanan *irradiance*. Ini karena voxelisasi sudah dilakukan walau tanpa menggunakan metode ini.

Pembagian tahap ke dalam kelipatan frame yang berbeda sangat menghemat waktu komputasi yang dibutuhkan. Proses propagasi terjadi dengan cukup cepat, mengingat bahwa



Gambar 12. Perbandingan hasil citra rendering Cornell Box di Minecraft serta performa dengan variasi a) naive ray tracing 1 spp, 1 bounce, b) 2 spp, 1 bounce, c) 1 spp, 4 bounce, dan d) metode ray tracing + floodfill 1 spp, 1 bounce.

(Source: dokumentasi penulis)

representasi voxel cukup besar. Metode yang digunakan juga bersifat cukup reaktif terhadap perubahan yang terjadi pada ruang, sehingga metode ini cukup efektif digunakan sebagai solusi GI *real-time*.

Sejauh ini, hanya propagasi *diffuse* yang disimulasikan oleh metode ini. Metode ini belum mampu untuk memprogasi cahaya *specular* yang lebih rumit.

Ada beberapa ruang untuk peningkatan yang terlihat pada implementasi algoritma ini. Penggunaan *buffer* 3D serta compute shader secara teori dapat meningkatkan efisiensi penggunaan GPU. Hal tersebut juga dapat mengurangi waktu komputasi *mapping* antar 2D ↔ 3D. Dapat dilakukan juga penelitian lebih lanjut untuk membuat propagasi lebih akurat dan mengikuti aturan *Physically Based Rendering* (PBR).

Kesimpulannya, jika keakuratan GI yang ingin dicapai, maka algoritma ini bukan pilihan yang cocok. Namun, jika pengguna hanya ingin GI yang cukup meyakinkan untuk kinerja *real-time*, maka algoritma ini bisa digunakan sebagai salah satu alternatif.

Appendix

Kode sumber shader untuk Minecraft dapat ditemukan di sini https://github.com/L4mbads/stima-floodfill-gi

Video Link At Youtube

https://youtu.be/egwCTtVx0Jg

Acknowledgment

Penulis mengucapkan rasa syukur kepada Tuhan Yang Maha Esa yang telah memberikan rahmat-Nya, sehingga penulis dapat menyelesaikan makalah yang berjudul "Pendekatan Program Dinamis dalam Simulasi Pencahayaan Voxel Berbasis Floodfill untuk Global Illumination Rendering" ini dengan tepat waktu. Penulis juga mengucapkan terima kasih kepada Monterico Adrian, S.T., M.T. selaku dosen pengampu mata kuliah IF2211 Strategi Algoritma Kelas 3 atas bimbingan dan pengajaran yang telah diberikan pada mata kuliah ini. Penulis juga mengucapkan terima kasih kepada Dr. Ir. Rinaldi Munir, MT., selaku salah satu dosen Strategi Algoritma yang telah memberikan referensi dan sumber belajar Strategi Algoritma melalui website-nya. Akhirnya, penulis mengucapkan terima kasih kepada kedua orang tua, keluarga, dan semua pihak yang telah membantu penulis dalam menyelesaikan makalah ini.

References

[1] Munir, Rinaldi. 2025. "Program Dinamis (*Dynamic Programming*) Bagian 1". https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/25-Program-Dinamis-(2025)-Bagian1.pdf (accessed on 24 Juni2025).

[2] Kajiya, J. T. (1986, August). The rendering equation. In Proceedings of the 13th annual conference on Computer graphics and interactive techniques (pp. 143-150).

[3] Law, G. (2013). Quantitative comparison of flood fill and modified flood fill algorithms. International Journal of Computer Theory and Engineering, 5(3), 503.

[4] Eisemann, E., & Décoret, X. (2006, March). Fast scene voxelization and applications. In Proceedings of the 2006 symposium on Interactive 3D graphics and games (pp. 71-78).

[5] Csala. Balint. 2021. "Colored Light Floodfill" https://www.shadertoy.com/view/NtcSD2 (accessed on 24 Juni2025)

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, 1 Juni 2025

Fachriza Ahmad Setiyono 13523162