# IMPLEMENTASI ALGORITMA *RECURSIVE DEPTH FIRST SEARCH* PADA GAME LABIRIN 3D BERBASIS DESKTOP

**Nabila Mahdiya Putri[1], Ilham Ramadhan Ali Umri[2], Muhammad Irsyad Nurul Azmi[3], Amalia Amriadi[4], Farros Shaffira[5], Ahmad Fahmi Karami[6], Fresy Nugroho[7]**
Teknik Informatika, Fakultas Sains dan Teknologi, Universitas Islam Negeri Maulana Malik Ibrahim
Jalan Gajayana No. 50 Malang, Telp: +62-341 551-354
e-mail: 210605110006@student.uin-malang.ac.id[1],210605110093@student.uin-malang.ac.id[2],210605110096@student.uin-malang.ac.id[3],210605110103@student.uin-malang.ac.id[4],210605110161@student.uin-malang.ac.id[5]

**Abstrak**

Penelitian ini menerapkan algoritma *Recursive Depth first search* (DFS) pada *game* labirin horor 3D berbasis desktop. Algoritma DFS memungkinkan pembuatan labirin dinamis dengan fitur yang dapat disesuaikan, termasuk penempatan musuh berlogika NPC untuk meningkatkan tingkat kesulitan pemain. Kriteria labirin sempurna dipenuhi dengan ukuran 30x30 untuk pengujian akurasi. Implementasi dalam Unity menghasilkan pengalaman bermain yang menarik dan menantang. Meskipun memperhatikan keterbatasan waktu, algoritma DFS memberikan fondasi yang kuat untuk pengaturan labirin yang dapat disesuaikan, menciptakan variasi dan kegembiraan dalam permainan.
***Kata kunci:*** *Depth first search*, Labirin, *Game*

***Abstract***

*This research implements the Recursive Depth first search (DFS) algorithm in a desktop-based 3D horror maze game. The DFS algorithm enables the creation of dynamic mazes with customizable features, including the logical placement of NPC enemies to enhance player difficulty levels. The criteria for a perfect maze are fulfilled with a size of 30x30 for accuracy testing. Implementation in Unity results in an engaging and challenging gaming experience. Despite considering time constraints, the DFS algorithm provides a solid foundation for customizable maze configurations, creating variation and excitement in the gameplay*

***Keywords: Depth first search, Labirin, Game***

## 1. Pendahuluan

Perkembangan teknologi informasi pada saat ini memberikan dampak signifikan terhadap berbagai aspek kehidupan, salah satunya dalam pengembangan permainan komputer. Permainan komputer, atau yang sering disebut sebagai *game*, telah menjadi bagian integral dari hiburan modern. Perkembangan ini tidak hanya mencakup aspek visual dan audio, tetapi juga berkaitan dengan tingkat kesulitan dan kekompleksan permainan itu sendiri.

Algoritma *Recursive Depth first search* (DFS) telah menjadi salah satu pendekatan yang signifikan, terutama pada *game* berbasis labirin. Labirin, sebagai struktur permainan, menawarkan potensi unik untuk tantangan eksplorasi dan navigasi. Penerapan algoritma DFS pada jalur labirin *game* labirin 3D membuka peluang untuk menciptakan pengalaman bermain yang lebih adaptif dan intens.

*Game* adalah salah satu hal penting dalam perkembangan teknologi, mulai dari yang sederhana dengan mekanika yang terbatas, hingga yang kompleks seperti kebanyakan *game* saat ini. Salah satu jenis mekanika permainan yang menarik adalah tema labirin, di mana pemain akan dituntut untuk memecahkan teka-teki mulai dari garis *start* hingga garis *finish*. Secara definisi, labirin adalah sebuah puzzle untuk mencari jalan keluar, di mana selama dalam perjalanan menelusuri labirin, pemain akan menghadapi banyak rintangan atau halangan untuk sampai pada tujuan. [1].

Pada penelitian ini dibuatlah sebuah *game* horor berbasis labirin yang mengimplementasikan algoritma *Recursive Depth first search*, untuk memberikan rintangan yang lebih menantang kepada *player* ditambahkan beberapa *enemy* (Succubus, Iblis, Wewe Gombel) yang memiliki logika NPC (*Non Playable Character*). Algoritma *Depth First Search* menciptakan dan mengelola labirin dengan ukuran dan fitur yang dapat dikonfigurasi, seperti lebar, kedalaman, jumlah ruangan, ukuran ruangan, jumlah musuh, dan skala elemen-elemen labirin. Setelah menginisialisasi dan menghasilkan labirin secara acak, Algoritma akan menambahkan ruangan, menempatkan *player* dan enemy di lokasi acak dalam labirin, dan membangun NavMesh untuk pengaturan navigasi *player*. selain itu, terdapat metode untuk menghitung jumlah tetangga persegi pada suatu posisi dalam labirin dan metode untuk menempatkan *player* dan enemy dalam labirin. Algoritma ini menyediakan dasar untuk menciptakan lingkungan labirin dalam permainan dengan elemen-elemen seperti *player*, enemy, dan ruangan[2].

Dengan demikian, pengembangan *game* horor berbasis labirin dengan implementasi algoritma *Recursive Depth first search* memberikan pengalaman bermain yang menarik dan menantang bagi pemain. Melalui kombinasi rintangan labirin dan kehadiran musuh dengan logika NPC, *game* ini memberikan kesempatan bagi pemain untuk menguji keterampilan pemecahan masalah dan keterampilan navigasi dalam lingkungan yang dinamis. Algoritma ini memberikan landasan yang kuat untuk menciptakan pengaturan labirin yang dapat disesuaikan, menjadikan pengalaman bermain semakin variatif dan menarik.

## 2. Metode Penelitian

Implementasi pada penelitian *game* labirin ini menetapkan aturan permainan yang sederhana sesuai dengan aturan umum permainan labirin. Pemain akan melakukan penelusuran labirin dari titik awal hingga titik akhir. Algoritma yang diadopsi adalah *Recursive Depth first search*, mengikuti syarat untuk menghasilkan labirin yang memenuhi kriteria sempurna, dimana labirin harus :

1. Memiliki titik awal serta titik akhir.
2. Tidak memiliki jalan yang berulang
3. Labirin sempurna dicapai dengan mengunjungi setiap node tepat satu kali dan tidak ada node yang terisolasi

Labirin yang dihasilkan akan memiliki dimensi sesuai dengan yang telah ditetapkan, sehingga hasil dari pengujian dapat terlihat lebih akurat. Dimensi labirin yang telah ditentukan adalah 30x30

## 3. Hasil dan Analisis

Algoritma pencarian mendalam, DFS, dikenal sebagai metode traversal graf yang menelusuri simpul secara mendalam. Secara informal, DFS mengunjungi simpul dari akar ke simpul anak (*child one*), terus menerus hingga ke daun terlebih dahulu. Jika tidak menemukan solusi, algoritma melakukan runut balik ke simpul bapak (*parent node*), lalu menelusuri kembali simpul anak yang belum dikunjungi. Begitu seterusnya hingga menemukan solusi.[3]

DFS menjadi dasar utama pada penelitian ini, mengoptimalkan penelusuran simpul dalam traversal graf dari akar ke simpul anak. Pemahaman konsep DFS dijadikan landasan untuk mengelola labirin 3D pada *game*. Langkah-langkah implementasi konkret algoritma DFS dilakukan melalui kode program yang dioptimalkan, menciptakan dan mengelola labirin *game* 3D.

Kode program di bawah ini mencerminkan aplikasi langsung dari algoritma DFS pada labirin *game* 3D:

```
Algorithm GenerateMaps():
   Generate(5, 5)
Algorithm Generate(x, z):
   if CountSquareNeighbours(x, z) >= 2:
      return
   map[x, z] = 0
```

```
directions.Shuffle()
Generate(x + directions[0].x,  z+ directions[0].z)
Generate(x + directions[1].x,  z+ directions[1].z)
Generate(x + directions[2].x,  z+ directions[2].z)
Generate(x + directions[3].x,  z+ directions[3].z)
```

Berdasarkan pseudocode diatas, labirin akan ter*generate* sesuai dengan alur dari algoritma DFS. Berikut adalah proses dari *generate* labirin sesuai dengan algoritma DFS:

1. *GenerateMaps*(): merupakan algoritma utama yang dipanggil untuk memulai proses penghasilan peta. Ini memanggil fungsi *Generate*(5, 5) dengan parameter awal x=5 dan z=5.
2. *Generate*(x, z): Ini adalah fungsi rekursif yang bertanggung jawab untuk menghasilkan peta. Proses ini dilakukan dengan langkah-langkah berikut:
    a. Pengecekan Tetangga: Melalui pemanggilan fungsi *CountSquareNeighbours*(x, z), dilakukan pengecekan jumlah tetangga yang memiliki nilai *map* sama dengan 1 pada posisi (x, z). Jika jumlah tetangga yang bernilai 1 lebih besar atau sama dengan 2, maka fungsi *Generate* berakhir (*return*) tanpa melakukan apa pun. Berikut adalah ilustrasi dari proses pengecekan tetangga.

Tabel 1. Peta awal

| | | | | |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 0 | 0 |
| 0 | 1 | 0 | 1 | 0 |
| 0 | 0 | 1 | 1 | 0 |
| 0 | 0 | 0 | 0 | 0 |

Pada proses ini, sebagai contoh, ketika sedang memproses node pada posisi (x, z) = (2, 2) (nilai *map* = 0), akan dilakukan pengecekan tetangga. Node yang memiliki nilai 1 dianggap sebagai tetangga.

Tabel 2. Peta dengan penanda sel yang sedang diproses (x, z) = (2, 2)

| | | | | |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 0 | 0 |
| 0 | 1 | X | 1 | 0 |
| 0 | 0 | 1 | 1 | 0 |
| 0 | 0 | 0 | 0 | 0 |

Selanjutnya, menghitung jumlah tetangga dengan nilai 1. Dalam konteks ini, node yang memiliki nilai 1 terletak di atas, bawah, kiri, dan kanan dari posisi (2, 2).

Tabel 3. Jumlah tetangga yang bernilai 1

| | | |
|---|---|---|
| 1 | 1 | 0 |
| 1 | X | 1 |
| 0 | 1 | 1 |

Dari proses pengecekan tetangga di atas, terhitung bahwa jumlah node yang bernilai 1 adalah 6. Kemudian, kita membandingkannya dengan syarat dalam pseudocode: Jika jumlah tetangga yang bernilai 1 lebih besar atau sama dengan 2, maka fungsi *Generate* berakhir (*return*) tanpa melakukan apa pun.

Dalam contoh proses ini, jumlah tetangga yang bernilai 1 adalah 6, dan karena 6 lebih besar atau sama dengan 2, fungsi *Generate* pada posisi (2, 2) berakhir tanpa melakukan perubahan apa pun. Sehingga nilai pada posisi (2, 2) tetap tidak berubah menjadi 0.

b. Pengaturan Nilai *Map*: Jika pengecekan tetangga tidak memenuhi syarat di atas, nilai *map* pada posisi (x, z) diatur menjadi 0.

Tabel 4. Pengaturan Nilai *Map*

| | | | | |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 0 | 0 |
| 0 | 1 | 0 | 1 | 0 |
| 0 | 0 | 0 | 1 | 0 |
| 0 | 0 | 0 | 0 | 0 |

Karena jumlah tetangga yang bernilai 1 berjumlah 3, tidak memenuhi syarat "lebih besar atau sama dengan 2". Oleh karena itu, langkah "Pengaturan Nilai *Map*" diaktifkan, mengakibatkan nilai pada posisi (3, 3) diubah menjadi 0. Dalam ilustrasi ini, terlihat bahwa nilai pada posisi (3, 3) diubah menjadi 0 karena jumlah tetangga yang bernilai 1 tidak memenuhi syarat yang tertera dalam pseudocode. Proses ini menciptakan perubahan pada peta, dan pengaturan nilai dapat terjadi pada setiap node di peta yang memenuhi syarat.

c. Pembalikan Arah Acak: Arah-arah pergerakan (*directions*) diacak (*Shuffle*()). Ini menjamin bahwa pergerakan ke tetangga-tetangga akan dilakukan dalam urutan acak.

Tabel 5. Hasil dari proses pembalikan acak

| | | | | |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 | 0 |
| 0 | 0 | 1 | 1 | 0 |
| 0 | 0 | 0 | 0 | 0 |

Adapun alur kerja dari proses pembalikan arah acak sehingga menghasilkan data yang ditampilkan pada tabel diatas, berikut cara kerjanya:

Langkah 1: Pertama-tama, kita mulai dari suatu posisi (x, z) di peta. Misalnya, kita mulai dari posisi (2, 2), yang merupakan elemen "0" pada peta.

Langkah 2: Kemudian, kita tentukan tetangga-tetangga dari posisi tersebut. Dalam contoh ini, tetangga-tetangga dari (2, 2) adalah (3, 2), (1, 2), (2, 3), dan (2, 1).
Langkah 3: Arah-arah pergerakan (*directions*) diacak menggunakan fungsi *Shuffle*(). Misalnya, hasil acak bisa menjadi (0, 1, 3, 2).
Langkah 4: Lakukan pergerakan ke tetangga-tetangga dalam urutan acak yang telah dihasilkan. Misalnya, kita mulai dengan (2, 2) dan pergi ke (3, 2).

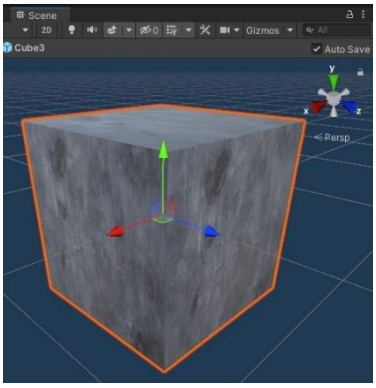
d. Rekursi untuk Tetangga: Kemudian, fungsi *Generate* dipanggil secara rekursif untuk keempat tetangga dari posisi saat ini (x, z), yaitu tetangga di arah acak. Ini dilakukan untuk terus memperluas area yang dihasilkan dalam peta.

Tabel 6. Rekursi untuk tetangga

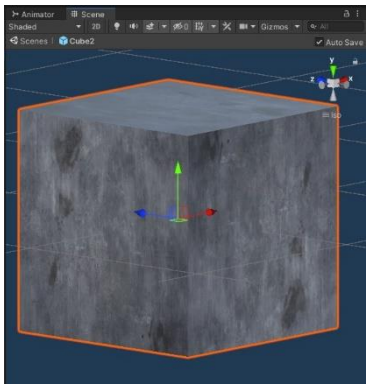
| | | | | |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 0 | 0 |
| 0 | 1 | X | 1 | 0 |
| 0 | 0 | 0 | 1 | 0 |
| 0 | 0 | 0 | 0 | 0 |

Proses ini akan berlanjut terus, dan rekursi untuk tetangga-tetangga dalam arah yang berbeda akan terus dilakukan. Setiap pemanggilan rekursif menciptakan cabang baru dalam proses, dan hal ini akan berlangsung hingga syarat pada 'Pengecekan Tetangga' terpenuhi pada suatu titik atau cabang tertentu. Akhirnya, peta dihasilkan melalui proses rekursif dan acak ini.

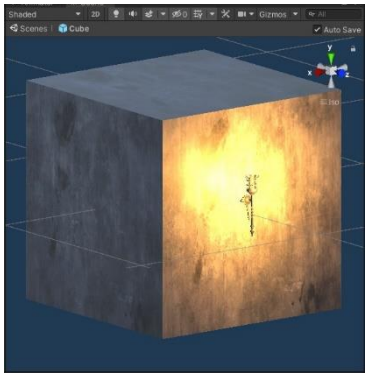
Berdasarkan tahapan-tahapan dari proses *generate* menggunakan algoritma DFS di atas, langkah selanjutnya adalah mengimplementasikannya ke dalam bentuk script untuk pembuatan labirin. Dilanjutkan dengan penambahan model 3 kubus sebagai representasi 3D untuk labirin seperti pada Gambar 1, Gambar 2, dan Gambar 3. Sedangkan hasil dari labirin terlihat pada Gambar 3 dan Gambar 4. Pembuatan labirin ini dilakukan secara otomatis melalui platform Unity.



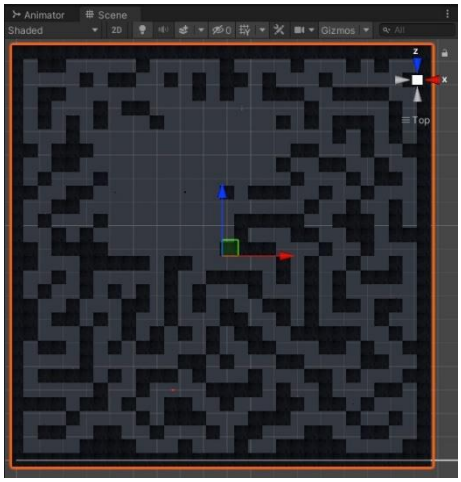

Gambar 2. Desain Cube 3




Gambar 1. Desain Cube 2




Gambar 3. Desain Cube 1




Gambar 5. Hasil *generate* labirin kedua


Gambar 4. Hasil *generate* labirin pertama

Ketika program dijalankan, labirin akan menampilkan keanekaragaman yang bergantung pada parameter X dan Z. Sehingga jumlah cube yang dibutuhkan menjadi beragam. Perbedaan bentuk yang terlihat pada Gambar 1 dan Gambar 2 menunjukkan variasi dalam jumlah cube yang dibutuhkan setiap kali proses *generate* dilakukan. Hasil implementasi labirin ini menggunakan ukuran 30x30. Berikut adalah tabel yang menunjukkan jumlah cube setelah melakukan 10 kali *generate*[4]:

Tabel 7. Jumlah Cube

| *Generate* ke- | Titik Awal (X, Z) | Titik Akhir (X, Z) | Jumlah Cube |
|---|---|---|---|
| 1 | 0,0 | 174, 174 | 391 |
| 2 | 0,0 | 174, 174 | 402 |
| 3 | 0,0 | 174, 174 | 382 |
| 4 | 0,0 | 174, 174 | 411 |
| 5 | 0,0 | 174, 174 | 390 |
| 6 | 0,0 | 174, 174 | 381 |

| | | | |
|---|---|---|---|
| 7 | 0,0 | 174, 174 | 381 |
| 8 | 0,0 | 174, 174 | 408 |
| 9 | 0,0 | 174, 174 | 387 |
| 10 | 0,0 | 174, 174 | 405 |

**4. Kesimpulan**

Dari penelitian di atas, implementasi algoritma *Recursive Depth first search* pada *game* labirin menghasilkan labirin yang sempurna dengan tambahan ruangan di dalamnya. Labirin dibentuk dari sejumlah cube yang beragam setiap kali proses *generate* dilakukan. Meskipun metode ini efektif, terdapat kelemahan terutama dari segi waktu. Proses pengecekan semua node membutuhkan waktu yang cukup lama. Setiap node harus ditandai sebagai visited, menambah jumlah langkah dibandingkan dengan algoritma lainnya.

**DAFTAR PUSTAKA**

[1] R. B. Sirait, "Perancangan Aplikasi Game Labirin Dengan Menggunakan Algoritma Backtracking," Pelita Inform. Budi Darma, vol. Volume 5, no. Nomor 2, p. Halaman 100-103, 2013.
[2] G. T. Nainggolan, A. Graf, and D. Graf, "Analisis Perbandingan Algoritma *Depth first search* dan Algoritma Breadth First Search dalam Memecahkan Permainan Labirin," 2022.
[3] Kleinberg, J., & Tardos, E. (2006). Algorithm Design. Massachussets: Pearson Education, Inc.
[4] E. Setiadharma, L. Husniah, and A. S. Kholimi, "Algoritma Maze Generator *Recursive* BacktrackingUntuk Membuat Prosedural Labirin Pada Game Petualangan Labirin 3D," J. Repos., vol. 2, no. 3, pp. 373–384, 2020, doi: 10.22219/repositor.v2i3.397.
[5] O. Pribadi, "Maze Generator Dengan Menggunakan Algoritma Depth-First-Search," *J. TIMES*, vol. 4, no. 1, pp. 1–5, 2015, [Online]. Available: http://www.stmik-time.ac.id/ejournal/index.php/jurnalTIMES/article/view/213.