IMPLEMENTASI *REAL-TIME PATHFINDING* MENGGUNAKAN A* DAN REYNOLDS *STEERING OBSTACLE AVOIDANCE* PADA PERMAINAN KOMPUTER

SKRIPSI

Untuk memenuhi sebagian persyaratan untuk mencapai gelar Sarjana Komputer

Disusun Oleh :

Ivan Ananda Harsono

NIM. 115060800111057

KEMENTRIAN RISET TEKNOLOGI DAN PENDIDIKAN TINGGI

PROGRAM TEKNOLOGI INFORMASI DAN ILMU KOMPUTER

PROGRAM STUDI INFORMATIKA/ILMU KOMPUTER

UNIVERSITAS BRAWIJAYA

MALANG

2015

KATA PENGANTAR

Puji syukur saya ucapkan kepada Allah SWT yang senantiasa melimpahkan rahmat dan hidayah-Nya sehingga penulis dapat menyelesaikan Proposal Skripsi yang berjudul, "Implementasi *Real-Time Pathfinding* Menggunakan A* Dan Reynolds *Steering Obstacle Avoidance* Pada Permainan Komputer".

Skripsi ini diajukan sebagai salah satu syarat untuk mendapatkan gelar sarjana S-1 Program Teknologi Informasi dan Ilmu Komputer Universitas Brawijaya. Penyusun menyelesaikan Skripsi ini berdasarkan teori-teori yang telah di peroleh dalam perkuliahan, literature dari beberapa buku dan paper dan bimbingan dari dosen pembimbing serta pihat-pihak lain yang telah banyak memberikan semangat dan bantuan. Penyusun ingin menyampaikan penghargaan dan ucapan terima kasih yang sedalam-dalamnya kepada:

1. Bapak Eriq M. Adams J, S.T., M.Kom dan Bapak Denny Sagita R., S.Kom, M.Kom selaku dosen pembimbing Penulis. Terima kasih atas semua bimbingan dan dorongan semangatnya.
2. Bapak Drs. Marji, M.Si. dan Issa Arwani, ST., MT. selaku Ketua dan Sekertaris Program Studi Informatika serta segenap Bapak/Ibu Dosen, Staff Administrasi dan Perpustakaan Program Studi Teknik Informatika Universitas Brawijaya.
3. Kedua orang tua, Bapak Iwan Harsono dan Ibu Astuti yang selalu memberikan dukungan moril dan materil, semangat, kasih sayang, serta doa yang tidak pernah ada habisnya. Terima kasih kepada Ika Caesarina Rahmawati, kakak saya yang selalu memberikan kasih sayang, bimbingan dan dorongan semangat untuk saya.
4. Seluruh Dosen Teknologi Informasi dan Ilmu Komputer Universitas Brawijaya yang telah membekali penulis dengan ilmu-ilmu yang bermanfaat.

5. Kepada Siti Roslinda Rohman S.Kep. yang selalu memberi do'a, semangat, motivasi dan kasih sayang kepada penulis dalam pengerjaan Skripsi ini.
6. Sahabat saya Ahdiyat Gunamandi S.T yang selalu memberi semangat dan motivasi dalam pengerjaan Skripsi ini.
7. Teman terdekat yang selalu ada dan selalu mendukung saya : Rusliawan Santoso S.Kom., Briandana Riznov S.Kom., Faizal Abdi S.Kom..
8. Teman-teman Asrama seperjuangan terutama Fikri Ibrahim Arif S.S., Ronald T R S.Pi, Erapasha Garry S.S, Ulul, Adi.
9. Teman-teman angkatan 2011 terutama TIF C : Fahmi, Wiely, Angga, Ian, Imam, Indra, Reza, Fadhil, Agus, Elliya, Dina, Berlian, dll.
10. Teman-teman Raion Community yang telah memberikan saya banyak ilmu-ilmu dan pengalaman yang tidak saya dapatkan diperkuliahan.
11. Serta semua pihak yang namanya tidak bisa penulis sebutkan satu-persatu. Terima kasih atas do'a dan dukungannya.

Penyusun sadar bahwa masih banyak kesalahan dan kekurangan dalam penyusunan Skripsi ini, untuk itu penyusun mohon maaf dan mengharapkan kritik dan saran guna penyempurnaan selanjutnya.

Malang, 1 Januari 2015

Penulis

ABSTRAK

Ivan Ananda Harsono. 2014. Implementasi *Real-time Pathfinding* menggunakan A* dan Reynolds *Steering Obstacle Avoidance* pada Permainan Komputer. Skripsi Program Studi Informatika/ Ilmu Komputer, Program Teknologi Informasi dan Ilmu Komputer, Universitas Brawijaya. Pembimbing : Eriq M. Adams J, S.T., M.Kom dan Denny Sagita R., S.Kom, M.Kom.

Algoritma A* merupakan algoritma tradisional yang sering digunakan untuk menyelesaikan masalah *pathfinding*. Algoritma *long steering* tradisional seperti A* tidak mampu untuk dipakai untuk menyelesaikan permasalahan *real-time pathfinding*. *Real-time Pathfinding* merupakan permasalahan *pathfinding* dalam lingkungan *real-time* dimana terdapat *dynamic obstacles* yang menambah kompleksitas permasalahan *pathfinding*. Dengan A* tradisional dalam lingkungan *real-time*, jika terdapat halangan dinamis akan dilakukan kalkulasi ulang yang akan memakan banyak sumber daya CPU. Salah satu algoritma *short steering* yang dapat diterapkan untuk menyelesaikan permasalahan *real-time pathfinding* adalah Reynolds *Steering Obstacle Avoidance* yang mampu menghindari adanya tabrakan dengan *dynamic obstacles*. Oleh karena itu, dalam penelitian ini dilakukan implementasi *realtime pathfinding* dengan mengintegrasikan antara algoritma *long steering* A* untuk pathfinding dan algoritma *short steering* Reynolds *Steering Obstacle Avoidance* untuk menghindari halangan dinamis sehingga tidak perlu dilakukan kalkulasi jalur ulang untuk mengoptimalkan kinerja dari CPU.

Implementasi algoritma dilakukan dengan melakukan simulasi pada peta permainan 3D. Representasi ruang pencarian dari peta permainan yang digunakan berupa *regular grids* berbentuk persegi. Pada peta akan diletakan halangan dinamis dan statis lalu agen akan berjalan dari titik awal ke titik tujuan yang telah di perhitungkan jalurnya dengan menghindari halangan. Simulasi implementasi algoritma dibedakan menjadi 2 skenario. Simulasi dilakukan dengan *game engine* Unity versi 4.6. Pengujian panjang jalur pada 30 uji coba dalam skenario 1, menunjukkan bahwa implementasi *real-time pathfinding* dengan A* dan Reynolds *Steering Obstacle Avoidance* akan menghasilkan jalur yang lebih panjang dibandingkan dengan penghitungan jalur ulang dengan perbandingan 8:7. Sedangkan untuk pengujian FPS pada 30 uji coba dalam skenario 2, menunjukkan bahwa kinerja CPU pada implementasi *real-time pathfinding* dengan A* dan Reynolds *Steering Obstacle Avoidance* akan 2 kali optimal dibandingkan dengan menggunakan penghitungan jalur ulang.

Kata Kunci – pencarian jalur, *real-time pathfinding*, algoritma A*, Reynolds Steering, permainan komputer.

ABSTRACT

Ivan Ananda Harsono. 2014. ***Implementation of Realtime pathfinding using A * and*** **Reynolds** ***Steering Obstacle Avoidance on Computer Games*****.** ***Undergraduate Thesis of Informatic Study Program, Information Technology and Computer Science Program, Brawijaya University,*** **Malang.** ***Advisor*** **: Eriq M. Adams J, S.T., M.Kom** ***and*** **Denny Sagita R., S.Kom, M.Kom.**

*A * algorithm is a traditional algorithms that are often used to solve the problem of pathfinding. Traditional long steering algorithms such as A* is not able to be used to solve real-time pathfinding problems. Real-time pathfinding is a pathfinding problems in real-time environment where there is a dynamic obstacle that increase the complexity of pathfinding. With traditional A* in real-time environment, if there is a dynamic obstacle there will be a recalculation path that will take a lot of CPU resources. One of the short steering algorithms that can be applied to solve real-time problems pathfinding is Reynolds Steering Obstacle Avoidance who was able to avoid any collision with dynamic obstacles. Therefore, in this study implemented a real-time pathfinding algorithm that integrates long steering A* algorithm for pathfinding and short steering Reynolds Steering Obstacle Avoidance algorithm to avoid any collision with dynamic obstacles so it does not need to re-calculate path to optimize the performance of the CPU.*

The algorithm implemented by doing a simulation on a 3D game map. Representation of the search space of a game map is using a square-shaped regular grids. On the map, will be placed some dynamic and static obstacle then there's will be an agent that will run from the starting point to a destination point on a path that has been calculated and agen will avoid obstacles. Simulation of algorithm implementations will be divided into 2 scenarios. Simulations done with the game engine Unity version 4.6. Testing the path length in 30 trials in scenario 1, shows that the real-time pathfinding implementation using A and Reynolds Steering Obstacle Avoidance will result in a longer path than path re-calculation with a ratio of 8:7. As for FPS testing on 30 trials in scenario 2, shows that the performance of the CPU in the real-time pathfinding implementation using A* and Reynolds Steering Obstacle Avoidance would be 2 times more optimal than path re-calculation.*

Keywords **-** *pathfinding, real-time pathfinding, A * algorithm,* Reynolds *Steering, computer games.*

DAFTAR ISI

DAFTAR GAMBAR

DAFTAR TABEL

BAB I
PENDAHULUAN

1.1 Latar Belakang

Perkembangan *game* saat ini telah didukung dengan kemajuan grafis yang membuat *game environment* dan jalannya permainan menjadi lebih realis. Namun grafis yang telah dibangun tidak akan menghasilkan sebuah *game* yang realistis jika tidak didukung oleh implementasi *behaviors* agen cerdas/ *player* / NPC (*Non Player Character*) dalam *game*. Untuk menghidupkan perilaku dari agen cerdas pada *game* simulasi perlu diterapkan *Artificial Intelligence* [THU-05]. Penggunaan *Artificial Intelligence* dalam *game* untuk mengimplementasikan perilaku NPC meliputi *movement, pathfinding, strategy,* dan *decision making*. *Pathfinding* merupakan teknik kecerdasan buatan dalam *game* untuk pencarian jalur antara 2 titik.

Algoritma A* merupakan algoritma tradisional yang sering digunakan untuk menyelesaikan masalah *pathfinding.* Algoritma *long steering* tradisional seperti A* tidak mampu untuk dipakai untuk menyelesaikan permasalahan *real-time pathfinding*. *Real-time Pathfinding* merupakan permasalahan *pathfinding* dalam lingkungan *real-time* dimana terdapat *dynamic obstacles* (halangan dinamis) yang menambah kompleksitas permasalahan *pathfinding.* Dengan A* tradisional dalam lingkungan *real-time,* jika terdapat halangan dinamis yang menutupi jalur dari agen cerdas, akan dilakukan kalkukasi jalur ulang. Jika peta permainan terus berubah-ubah maka akan dilakukan kalkulasi ulang terus-menerus. Dengan kalkukasi ulang yang terus menerus akan memakan banyak sumber daya CPU yang menyebabkan penurunan kinerja dari CPU. Sebelumnya, sudah terdapat penelitian sejenis [FOU-09], dengan meneliti tabrakan antara aktor dengan menggunakan Reynolds Steering *Collision Avoidance*.

Salah satu algoritma *short steering* yang dapat diterapkan untuk menyelesaikan permasalahan *real-time pathfinding* adalah Reynolds Steering *Obstacle Avoidance*. Reynolds Steering *Obstacle Avoidance* merupakan algoritma untuk menghindari adanya tabrakan dengan halangan yang dinamis (*dynamic obstacles*). Reynolds Steering *Obstacle Avoidance* biasanya diterapkan pada agen

yang hanya berjalan lurus tanpa mengikuti sebuah jalur dan hanya akan merubah arah jika terdapat halangan didepannya.

Oleh karena itu, dalam skripsi ini dilakukan implementasi untuk *real-time pathfinding* dengan mengintegrasikan antara algoritma *long steering* A* untuk *pathfinding* dan algoritma *short steering* Reynolds *Steering Obstacle Avoidance* untuk menghindari halangan yang dinamis sehingga tidak perlu dilakukan kalkulasi jalur ulang pada lingkungan yang *real-time* untuk mengoptimalkan kinerja dari CPU.

1.2 Rumusan Masalah

Berdasarkan latar belakang diatas, penulis merumuskan beberapa masalah sebagai berikut :

1. Bagaimana mengimplementasikan *Real-time Pathfinding* dengan A* dan Reynold *Steering Obstacle Avoidance* ?
2. Bagaimana menguji kinerja *Real-time Pathfinding* dengan A* dan Reynold *Steering Obstacle Avoidance* ?

1.3 Batasan Masalah

Agar permasalahan yang dirumuskan lebih terfokus, maka penelitian ini dibatasi oleh hal-hal sebagai berikut :

1. Tabrakan antara satu aktor dengan aktor yang lain tidak diperhatikan.
2. Representasi ruang pencarian dari peta permainan yang digunakan berupa *regular grids* berbentuk persegi.
3. Target *pathfinding* yang digunakan dalam penelitian bukan merupakan *moving target* (*static target*).
4. Rintangan dinamis (*dynamic obstacle*) yang digunakan berupa rintangan yang tiba-tiba muncul dan rintangan yang bergerak lambat.
5. Faktor ancaman pada peta permainan tidak diperhatikan (*non-tactical*).
6. Pada peta permainan tidak terdapat banyak halangan (*non-crowded environment*).

1.4 Tujuan

Tujuan yang ingin dicapai dalam implementasi skripsi ini adalah untuk membangun solusi untuk menyelesaikan permasalahan *real-time pathfinding* menggunakan A* sebagai algoritma *long steering* dan Reynold *Steering Obstacle Avoidance* sebagai algoritma *short steering*.

1.5 Manfaat

Manfaat yang ingin dicapai dalam implementasi skripsi ini adalah:

1. Bagi Penulis :
   a. Mengaplikasikan ilmu yang didapat selama mengikuti perkuliahan di Teknik Informatika Universitas Brawijaya.
   b. Mendapatkan pemahaman tentang perancangan dan pengembangan *real-time patfinding* menggunakkan A* dan Reynolds *Steering Obstacle Avoidance*.
2. Bagi Pengembang *Game* :

   Memberikan alternatif baru terhadap pengembang permainan untuk mengatasi pencarian jalur dalam lingkungan dinamis (*real-time pathfinding*).

1.6 Sistematika Penulisan

Sistematika isi dan penulisan dalam skripsi ini antara lain :

BAB I **Pendahuluan**

Berisi tentang latar belakang masalah, rumusan masalah, batasan masalah, tujuan dan manfaat dari penelitian serta sistematika penulisan.

BAB II **Dasar Teori**

Menguraikan tentang dasar teori dan teori penunjang yang berkaitan dengan *real-time pathfinding*, A*, dan Reynolds *steering obstacle avoidance*.

BAB III **Metodologi Penelitian dan Perancangan**

Berisi tentang gambaran umum langkah – langkah dalam mengimplementasi *real-time pathfinding* dimulai dari tahap perancangan dan implementasi serta pengujian secara umum.

BAB IV **Implementasi**

Membahas tentang implementasi dari *real-time pathfinding* menggunakan A* dan Reynolds *steering obstacle avoidance*.

BAB V **Pengujian dan Analisis**

Memuat hasil pengujian dan analisisi terhadap *real-time pathfinding* yang telah diimplementasikan.

BAB VI **Penutup**

Memuat kesimpulan yang diperoleh dari pembuatan dan pengujian *real-time pathfinding*, serta saran – saran untuk pengembangan lebih lanjut.

BAB II
DASAR TEORI

Bab ini membahas dasar teori yang digunakan untuk menunjang penulisan skripsi mengenai Implementasi *Real-time Pathfinding* menggunakan A* dan Reynolds *Steering Obstacle Avoidance.* Beberapa dasar teori uang dimaksud adalah *Pathfinding*, Algoritma A*, *Regular Grids*, *Obstacle Avoidance* dan Parameter Pengujian.

2.1 *Pathfinding* pada Permainan Komputer

Pathfinding biasanya merupakan inti dari setiap sistem pergerakan AI. *Pathfinding* memiliki tanggung jawab untuk mencari jalur dari sebuah titik dalam *game world* ke titik lainnya. *Pathfinding* akan menggunakan titik awal dan titik tujuan kemudian akan menemukan serangkaian titik-titik yang membentuk jalur dari titik awal ke titik tujuan [GRA-03].

Ada tiga tahap umum dalam melakukan *pathfinding* dalam permainan komputer [GRA-06] yaitu (1) melakukan *pre-processing* peta permainan (2) membuat sebuah *graph* beserta datanya (3) menelusuri *graph* untuk mendapatkan solusi *path* menggunakan algoritma pencarian.

Pre-processing pada peta permainan dilakukan dengan cara membagi-bagi peta permainan ke dalam bagian-bagian kecil. Setelah peta permainan dibagi kedalam bagian-bagian kecil sehingga didapatkan representasi ruang pencarian dari peta permainan yang telah disederhanakan. Representasi ruang pencarian dari peta permainan yang digunakan dalam permainan komputer diantaranya adalah *binary space partition trees* (BSP), *corner graph*, *waypoints graph*, *circle-based waypoint graph*, *navigation meshes*, *area awarness system*, dan *regular grids*.

Dalam melakukan penelusuran *graph* dapat digunakan algoritma *blind search* dan algoritma *informed search*. Algoritma *blind search* yang dapat digunakan diantaranya adalah BFS (*breadth first search*), dan DFS (*depth first search*). Algoritma *informed search* yang dapat digunakan diantaranya adalah Djikstra dan A*.

2.1.1 Real-time *Pathfinding*

Secara tradisional, *pathfinding* pada permainan komputer dilakukan pada representasi dari peta permainan yang statis [GRA-05]. Ini bekerja dengan baik jika tidak ada perubahan pada peta permainan. Pengunaan *physics engine* dalam permainan komputer merupakan salah satu penyebab adanya halangan dinamis dalam peta permainan sehingga menyebabkan algoritma *pathfinding* tradisional tidak mampu bekerja dalam lingkungan dinamis. Real-time *pathfinding* merupakan permasalahan *pathfinding* pada peta permainan yang dinamis. Lingkungan dinamis merupakan lingkungan dimana terdapat obstacle yang dinamis yang meningkatkan kompleksitas *pathfinding*.

Objektif dalam *real-time pathfinding* dapat dibagi menjadi dua, (i) mencari *path* dari posisi awal ke posisi tujuan dan (ii) menghindari halangan dinamis [GRA-05]. Objektif (i) dapat diselesaikan menggunakan algoritma *long steering* dan objektif (ii) dapat diselesaikan menggunakan algoritma *short steering* [KORF-90].

2.2 Algoritma A*

Algoritma A* [HAR-68] digunakan untuk menemukan sebuah path terpendek diantara dua titik dalam sebuah peta permainan. Dalam algoritma A*, peta permainan terdiri dari *node-node*. Setiap *node-node* tersebut memiliki tiga atribut yaitu F, G, dan H yang seringkali disebut berurutan sebagai nilai *fitness*, *goal* dan heuristik. Nilai G adalah biaya dari node awal sampai ke node yang ditelurusi sekarang (*current node*). Nilai H adalah biaya estimasi dari *current node* sampai ke node tujuan. Nilai F adalah penjumlahan dari nilai F dan G. *Node-node* dalam A* disimpan dalam *open list* dan *closed list*. *Open list* berisi *node-node* yang belum dijelajahi sedangkan *closed list* berisi *node-node* yang telah dijelajahi. *Parent node* merupakan referensi dari *node* yang menambahkan *node* ke *Open list*. Sebuah node dikatakan telah dijelajahi jika algoritma A* telah memeriksa semua *node* yang terhubung , menghitung nilai F, G, dan H, serta menyimpannya dalam *open list* untuk dijelajahi kemudian. *Open list* dan *closed list* dibutuhkan untuk

menyimpan jejak penelusuran *node-node* tersebut. Berikut merupakan *pseudo-code* algoritma A* [LES-05] :

1) Tambahkan *node* awal ke dalam *open list*.

2) Mengulangi langkah berikut:

a) Mencari biaya F terendah pada setiap *node* dalam *open list*. *Node* dengan biaya F terendah kemudian disebut *current node*.

b) Masukkan ke dalam *closed list*.

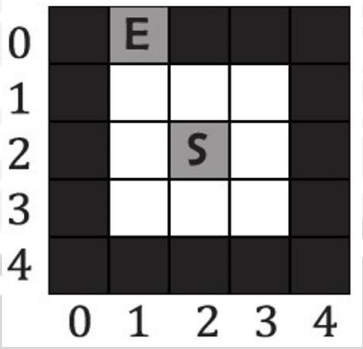
c) Untuk setiap 8 simpul (*neighbor node*) yang berdekatan dengan current node:

- Jika tidak *walkable* atau jika termasuk *closed list*, maka abaikan.
- Jika tidak ada pada *open list*, tambahkan ke *open list*.
- Jika sudah ada pada *open list*, periksa apakah ini jalan dari *node* ini ke *current node* yang lebih baik dengan menggunakan biaya G sebagai ukurannya. *Node* dengan biaya G yang lebih rendah berarti bahwa ini adalah jalan yang lebih baik. Jika demikian, buatlah *node* ini (*neighbor node*) sebagai *parent node* dari *current node*, dan menghitung ulang nilai G dan F dari simpul ini.

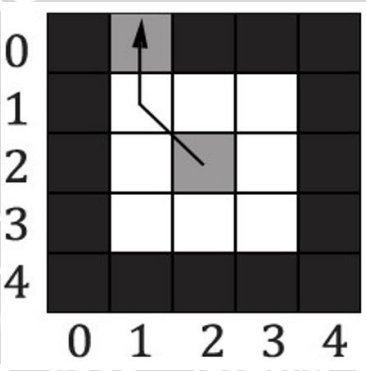
d) Pencarian jalur dihentikan ketika :

- Menambahkan target point ke dalam *closed list*, dalam hal ini jalan telah ditemukan, atau
- Gagal untuk menemukan titik tujuan, dan *open list* kosong. Dalam kasus ini, tidak ada jalan.

3) Simpan jalan. Bekerja mundur dari titik tujuan, pergi dari masing-masing *node* ke *parent node* sampai mencapai titik awal. Dan menghasilkan jalur yang akan diikuti.



Gambar 2.1 Peta permainan sederhana
Sumber : [MAT-02]



Gambar 2.2 Solusi *path* dari peta permainan
Sumber : [MAT-02]

Contoh sederhana berikut akan memperjelas *pseudo-code* algoritma A* diatas. Sebagai contoh, misal dalam peta permainan yang ditunjukkan dalam Gambar 2.1 dan Gambar 2.2 akan ditentukan jalur dari *node S*(2,2) ke *node E*(1,0). Penelusuran diawali dari node *S* dan kemudian dilakukan penghitungan nilai *f, g,* dan *h*. Nilai *g* sama dengan nol dikarenakan *node* yang ditelusuri hanya node *S*. Nilai *h* dapat dihitung menggunakan Manhattan Distance. Misal (*dx*,*dy*) merupakan *node* tujuan dan (*sx, sy*) merupakan *node* awal maka nilai *h* dapat dihitung menggunakan Manhattan Distance dengan persamaan berikut,

$$h = |dx - sx| + |dy - sy|$$

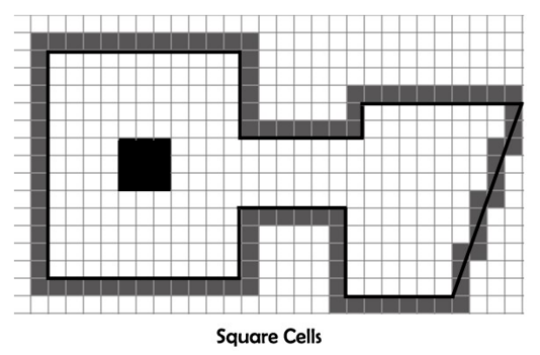
Jadi nilai *h* dari *node* yang ditelusuri sekarang *S*(2,2) ke *node E*(1,0) sama dengan 3 dan nilai *f* sama dengan 3. Setelah nilai *f, g,* dan *h* dari *node S* selesai dihitung maka *node* yang ditelusuri sekarang (*node S*) akan dimasukkan ke dalam *open list*. Kemudian semua *node* anak yang berdekatan dengan *node* yang ditelusuri

sekarang (*node S*) akan dimasukkan ke dalam *open list*. Nilai *g* untuk masing-masing *node* anak tersebut sama dengan 1. Sedangkan nilai *h* masing-masing *node* anak berbeda. Nilai *h* paling kecil adalah nilai *node* pada posisi (1,1) dan nilai *node* tersebut akan ditelusuri selanjutnya.

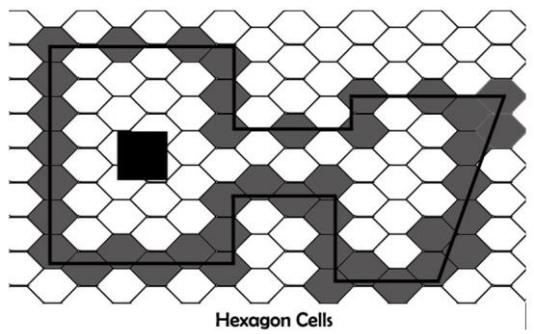
Node (1,1) mempunyai empat buah *node* anak : (1,0), (1,2), (2,1), dan (2,2). *Node* (2,2) ada dalam *closed list*, sedangkan node (1,0), (1,2), (2,1) ada dalam *open list*. Kemudian setelah dihitung nilai *f, g, h* dari masing-masing *node* anak didapatkan *node* (1,0) mempunyai nilai terbaik dan merupakan *node* tujuan.

2.3 *Regular Grids*

Cara paling sederhana untuk merepresentasikan sebuah ruang pencarian dengan menggunakan sebuah *regular grids* yang terbuat dari bentuk persegi, persegi panjang, *hexagons*, dan segitiga. Gambar 2.3 dan Gambar 2.4 menunjukkan contoh *regular grids* yang terbuat dari persegi dan *hexagons* [TOZ-04].



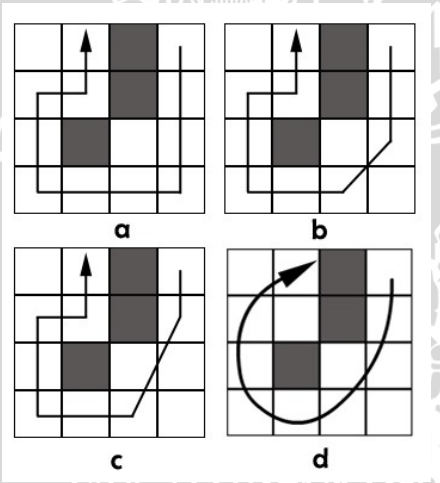
Gambar 2.3 Representasi *grid* dalam bentuk persegi
Sumber : [TOZ-04]



Gambar 2.4 Representasi *grid* dalam bentuk persegi
Sumber : [TOZ-04]

Ruang pencarian berdasarkan *grids* sangat populer digunakan dalam permainan komputer dua dimensi. Untuk memakai ruang pencarian berdasarkan *grids* dalam permainan tiga dimensi diperlukan modifikasi. Biasanya diperlukan *grid cells* dengan jumlah yang banyak untuk merepresentasikan sebuah peta permainan. Semakin luas peta permainan semakin banyak pula jumlah *grid cells* yang dipakai sehingga kebutuhan penyimpanan data *grid* dalam memori semakin banyak dan memperlambat *pathfinding*.

Kelebihan dari representasi ruang pencarian berbasis *grids* adalah mendukung adanya *random-access lookup*. *Random-access lookup* dalam representasi ruang pencarian berbasis *grids* memungkinkan kita untuk menentukan *tile* atau *grid cell* mana yang sesuai dengan koordinat peta permainan (*word-space coordinate*) dengan kompleksitas waktu O(1). Representasi ruang pencarian lain



Gambar 2.5 (a) pergerakan agen menggunakan empat arah (b) pergerakan agen menggunakan arah diagonal (c) pergerakan agen menggunakan stright pulling (d) pergerakan agen menggunakan catmull-rom splin.

Sumber : [TOZ-04]

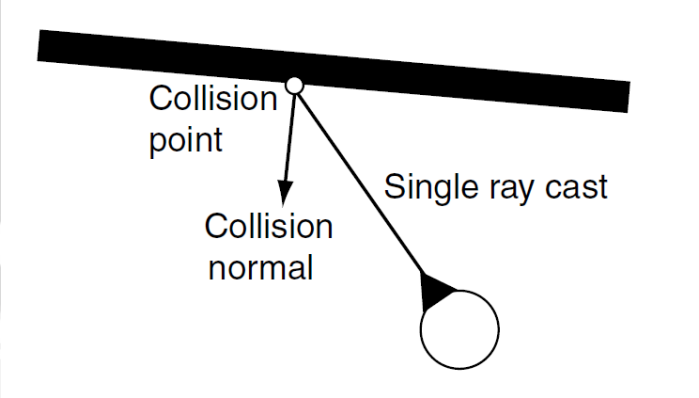
seperti *corner graph, waypoints graph, circle-based waypoint graph, navigation meshes* tidak mendukung *random-access lookup* sehingga diperlukan sebuah penelusuran *graph* untuk menentukan *node* terdekat sesuai dengan koordinat (*X,Y*) atau (*X,Y,Z*) yang diberikan.

Jika sebuah agen AI hanya mempunyai empat arah pergerakan dalam *grid* yang terbentuk dari persegi maka pergerakan agen terlihat aneh, seperti yang

ditunjukkan dalam Gambar 2.5a. Untuk meningkatkan pergerakan agen ini mungkin dapat ditambahkan arah gerakan diagonal seperti ditunjukkan dalam gambar 2.5b, namun *path* yang dihasilkan belum begitu optimal. *Stright-pulling* dapat digunakan untuk meningkatkan kualitas *path* seperti ditunjukkan dalam gambar 2.5c. *Catmull-Rom splines* dapat digunakan untuk membuat *path* terlihat halus. Seperti yang ditunjukkan dalam gambar 2.5d *splines* mengubah sederetan titik menjadi sebuah *path* yang halus.

2.4 *Obstacle Avoidance*

Obstacle Avoidance merupakan salah satu Steering Behaviours dari Craig Reynolds yang digunakan untuk menghindar dari halangan atau dinding yang ada didepan karakter. Karakter akan melemparkan satu atau lebih *ray* (sinar) keluar ke arah gerakannya. Jika *ray* (sinar) ini bertabrakan atau mengenai *obstacle* (halangan), akan dibuat target baru dan akan dilakukan *basic seek* terhadap target tersebut untuk menghindari tabrakan. Biasanya, sinar yang dilemparkan tidak *infinite* (tak terbatas). Biasanya disesuaikan dengan beberapa detik dari gerakan karakter tersebut [MIL-09].



Gambar 2.6 Collision ray menghindari dinding.

Sumber : [MIL-09]

2.5 Parameter Pengujian

Penelitian ini menggunakan 2 parameter dalam pengujian yaitu Frame per second dan Jarak Tempuh.

2.5.1 Frame per Second (FPS)

Frame per second (FPS) pada tampilan simulasi adalah jumlah gambar fragmen gambar-gambar berbeda ditampilkan tiap detik untuk mengelaui mata penonton menjadi objek yang bergerak [ALD-09]. FPS biasa disebut juga *frame rate*, FPS merupakan sebuah metrik fluiditas [ALD-09]. Acara pada TV NTSC sidiarkan pada 30 FPS, sedangkan PAL adalah 25 FPS, dan film pada bioskop ditampilkan pada 24 FPS [ALD-09].

Kapasitas 20 FPS dianggap oleh sebagian besar pihak sebagai batas minimum untuk mengelabui mata [ALD-09]. *Frame rate* yang lebih tinggi (berkisar antara 60 FPS) menghasilkan animasi yang lebih memanjakan, dan beberapa konsol permainan mengorbankan elemen lain untuk menjaga level ini [ALD-09].

FPS yang lebih rendah disebabkan oleh keterbatasan sistem, pemrograman yang buruk, dan aplikasi lain yang menghabiskan sumber daya sistem (seperti kecerdasan buatan) [ALD-09]. *Detail* dari gambar, seperti resolusi layar, kompleksitas objek, dan pencahayaan juga menurunkan jumlah FPS [ALD-09].

2.5.2 Panjang Jalur

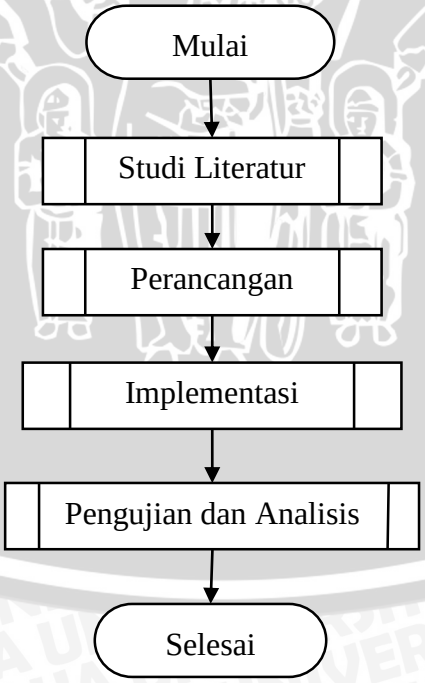
Panjang Jalur yang dimaksud adalah berapa panjang Jalur yang ditempuh masing-masing aktor. Terdapat 2 panjang Jalur yang diukur untuk pengujian, yaitu Panjang Jalur dengan perhitungan A* dengan menghindari halangan dinamis menggunakan Reynolds *Steering Obstacle Avoidance* dan Panjang Jalur yang dihitung ulang setelah munculnya halangan yang dinamis. Lalu akan dibandingkan mana panjang Jalur yang lebih pendek.

BAB III
METODE PENELITIAN DAN PERANCANGAN

Bab ini menjelaskan langkah-langkah metodologi penelitian dan perancangan. Dalam metode penelitian dijabarkan mengenai metode atau langkah-langkah yang digunakan dalam penelitian skripsi. Dalam subbab perancangan menjelaskan perancangan implementasi dari penelitian ini.

3.1 Metodologi Penelitian

Subbab ini menjelaskan langkah-langkah yang ditempuh dalam penelitian. Penelitian ini akan dilakukan dalam beberapa tahap meliputi perancangan, implementasi dan pengujian. Kesimpulan dan saran disertakan sebagai catatan atas metode yang diterapkan dan kemungkinan arah pengembangan selanjutnya. Gambar 3.1 merupakan diagram alir metode yang digunakan dalam penelitian ini diantaranya dimulai dari perancangan *real-time pathfinding* kemudian implementasi dan yang terakhir melakukan pengujian dan analisis.



Gambar 3.1 Diagram Alir Runtutan Metode Penelitian

Sumber : [Metode Penelitian]

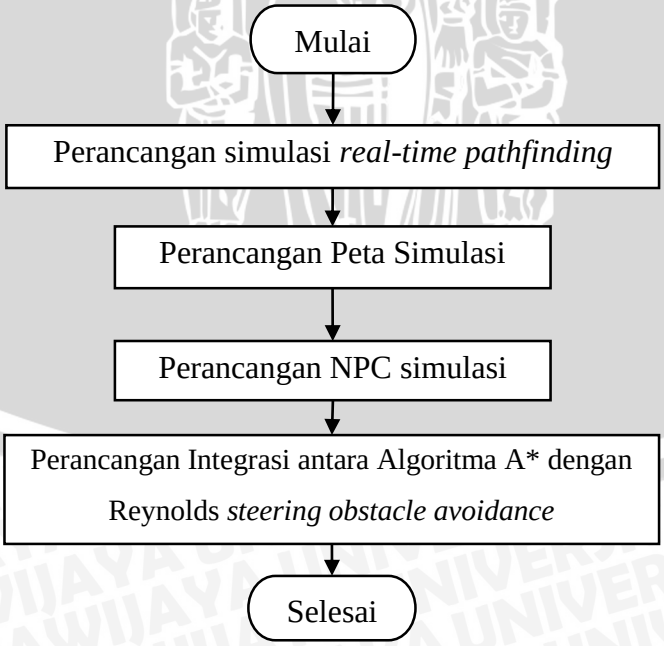
3.1 Studi Literatur

Studi literatur merupakan cara yang digunakan dalam pengumpulan informasi-informasi yang diperlukan dengan sumber yang digunakan berupa data-data yang terkait. Dalam studi literatur dikumpulkan data-data dari sumber yang terpercaya yang menjadikan sumber tersebut menjadi dasar dari teori-teori yang akan diimplementasikan lebih lanjut pada aplikasi yang dibuat. Studi literatur yang digunakan adalah sebagai berikut :

1. Algoritma *pathfinding* A*
2. Reynolds Steering Obstacle Avoidance
3. Bahasa Pemrograman C#
4. Parameter Pengujian
   a. *Frame per secont (FPS)*

3.2 Perancangan

Bab perancangan terdiri dari empat tahap, yaitu perancangan peta simulasi *real-time pathfinding*, perancangan peta simulasi, perancangan NPC simulasi, dan perancangan integrasi antara algoritma A* dengan Reynolds *steering obstacle avoidance*. Tahap-tahap perancangan dijabarkan dalam diagram alir Gambar 3.2.



Gambar 3.2 Diagram Alir Perancangan Implementasi

Sumber : [Metode Penelitian]

3.2.1 Perancangan Simulasi *real-time pathfinding*

Pada awal perancangan akan dirancang simulasi untuk pathfinding dinamis (*real-time pathfinding*) pada lingkungan 3D. Akan ada 2 simulasi yang dirancang untuk *real-time pathfinding.*

3.2.1.1 Perancangan Simulasi 1

Pada awal simulasi 1 sudah disediakan beberapa halangan statis lalu akan dimuncukan beberapa aktor yang akan berjalan dari titik awal ke titik akhir yang sudah ditentukan jalurnya. Setelah itu akan dimunculkan beberapa halangan dinamis yang akan menutupi jalur. Aktor akan mendeteksi halangan dinamis yang ada didepannya dan akan keluar dari jalurnya untuk menghindari halangan dinamis tersebut dan akan kembali kejalurnya setelah menghindari halangan yang dinamis. Skenario ini bertujuan mengetahui valid atau tidaknya *behaviour* yang akan dirancang dan diimplementasikan dan untuk menguji jarak tempuh yang dilalui jika dibandingkan antara A* dengan *obstacle avoidance* dan penghitungan ulang jalur A* setelah sudah ada halangan dinamis dengan melakukan uji coba dalam 3 peta permainan yang berbeda-beda.

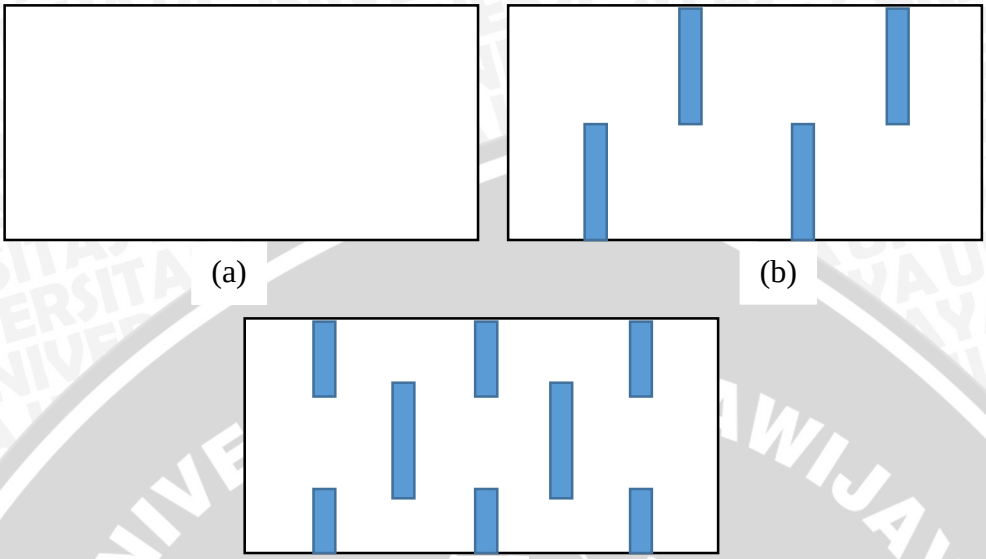
3.2.1.2 Perancangan Simulasi 2

Pada awal simulasi 2 sudah disediakan beberapa halangan dinamis yang bergerak lambat lalu akan dimuncukan beberapa aktor yang akan berjalan dari titik awal ke titik akhir yang sudah ditentukan jalurnya. Aktor akan mendeteksi halangan dinamis yang ada didepannya dan akan keluar dari jalurnya untuk menghindari halangan dinamis tersebut dan akan kembali kejalurnya setelah menghindari halangan yang dinamis. Skenario ini bertujuan untuk menguji algoritma yang lebih efisien diantara A* dengan *obstacle avoidance* dan kalkulasi jalur ulang A* berdasarkan rata-rata FPS dengan melakukan uji coba dalam 3 peta permainan yang berbeda-beda

3.2.2 Perancangan Peta Simulasi

Pada simulasi *real-time pathfinding* akan ada beberapa peta yang dirancang. Terdapat 3 peta untuk simulasi 1 dan 3 peta untuk simulasi 2. Berikut rancangan dari masing-masing peta simulasi dapat dilihat pada gambar 3.3 dan gambar 3.4.

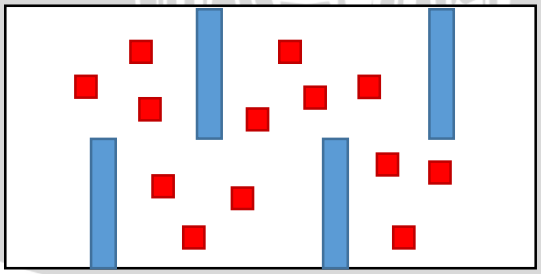
1. Peta Simulasi 1



Gambar 3.3 (a) Peta 1.1 (b) Peta 1.2 (c) Peta 1.3

Sumber : [Perancangan]

Dapat dilihat pada gambar 3.3, Kotak berwarna biru pada peta merupakan halangan statis. Pada peta 1.1 belum terdapat halangan dan pada peta 1.2 dan peta 1.3 halangan statis pada peta semakin banyak. Setelah memunculkan beberapa aktor pada peta permainan dan telah ditentukan jalurnya masing-masing, akan dimunculkan beberapa halangan dinamis seperti pada gambar 3.4.

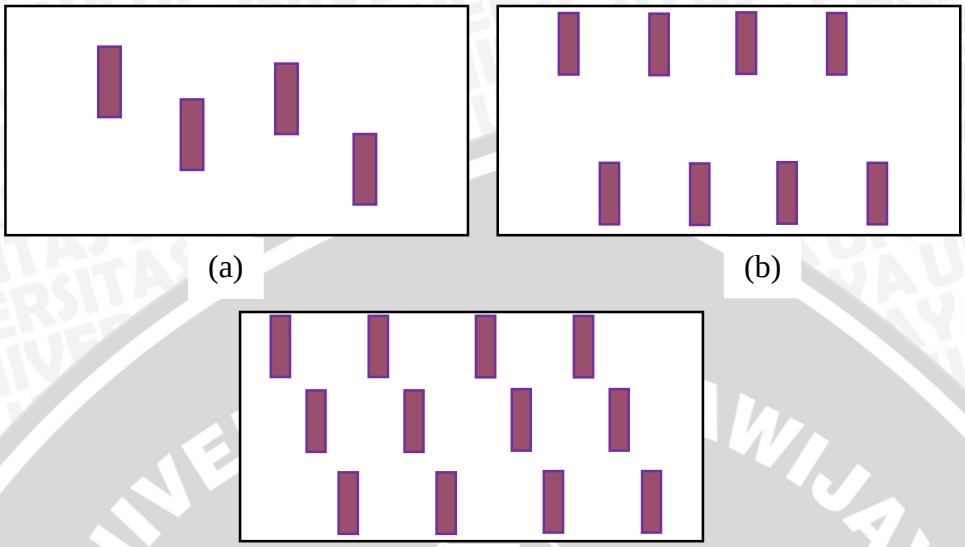


Gambar 3.4 Contoh peta yang telah dimunculkan halangan dinamis

Sumber : [Perancangan]

Kotak berwarna merah pada peta dalam gambar 3.4 merupakan halangan dinamis yang dimunculkan setelah memunculkan aktor.

2. Peta Simulasi 2



Gambar 3.5 (a) Peta 2.1 (b) Peta 2.2 (c) Peta 2.3

Sumber : [Perancangan]

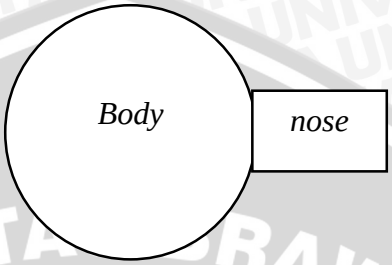
Pada gambar 3.5, Kotak berwarna ungu pada peta merupakan halangan dinamis yang akan bergerak lambat.

3.2.3 Perancangan NPC Simulasi

Pada simulasi *real-time pathfinding* hanya menggunakan satu jenis NPC yaitu Aktor. Aktor akan memiliki titik awal dan titik akhir yang diacak. Aktor akan mencari jalur dari titik awal ke titik akhir. Setelah menemukan jalurnya, Aktor akan berjalan pada jalur yang telah ditemukan. Jika ada halangan dinamis yang menutupi jalurnya, aktor akan keluar jalur untuk menghindari halangan tersebut lalu akan kembali ke jalurnya.

Behaviour dari NPC Aktor diimplementasikan menggunakan algoritma A* untuk mencari rute terpendek pada peta permainan dan algoritma Reynold *Steering Path Following* untuk menelusuri jalur yang dihasilkan dan algoritma Reynold *Steering Obstacle Avoidance* untuk menghindari halangan-halangan dinamis yang menghalangi jalur yang ditelusuri aktor. Rangkuman *behaviour* dari aktor ditunjukkan dalam Tabel 3.1.

Tabel 3.1 *Behaviour* NPC Aktor

Attribut	***Detail***
Nama NPC	Aktor
Rancangan Tampilan	 (Tampak Atas)
AI *Behaviour*	***Find Path & Path Follow* :** mencari rute terpendek dengan algoritma A* dan berjalan menelusuri jalur menggunakan algoritma Reynolds *Steering Path Following.* ***Avoid Obstacle* :** menghindari halangan dengan menggunakkan algoritma Reynolds *Steering Path Following.*

Sumber : [Perancangan]

3.2.3.1 Perancangan Implementasi *Behaviour Find Path & Path Follow*

Algoritma A* memiliki tiga tahapan yaitu, mendapatkan koordinat titik tujuan, mencari jalur terdekat dan membuat jalurnya, dan mengikuti jalur yang sudah tercipta. Representasi ruang pada peta permainan menggunakkan *Regular Grid* berbentuk persegi, karena representasi ruang pencarian berbasis *grids* mendukung adanya *random-access lookup* .Perancangan algoritma *pathfinding* A* dalan NPC akan dijabarkan pada diagram alir gambar 3.6.

Gambar 3.6 Diagram Alir Implementasi Behaviour Find Path & Path Follow

Sumber : [Perancangan]

Adapun langkah-langkah pada diagram alir implementasi *behaviour* dari aktor yang akan dijabarkan sebagai berikut :

1. Mendapatkan Titik Tujuan

   Aktor akan dimunculkan pada posisi yang diacak dan titik tujuan dari masing-masing aktor akan diacak.

2. Mencari Jalur Terdekat

   Pencarian jalur dilakukan dengan algoritma A* dengan langkah-langkah berikut:

   1) Tambahkan *node* awal ke dalam *open list*.

   2) Mengulangi langkah berikut:

a) Mencari biaya F terendah pada setiap *node* dalam *open list*. *Node* dengan biaya F terendah kemudian disebut *current node*.

b) Masukkan ke dalam *closed list*.

c) Untuk setiap 8 simpul (*neighbor node*) yang berdekatan dengan current node:

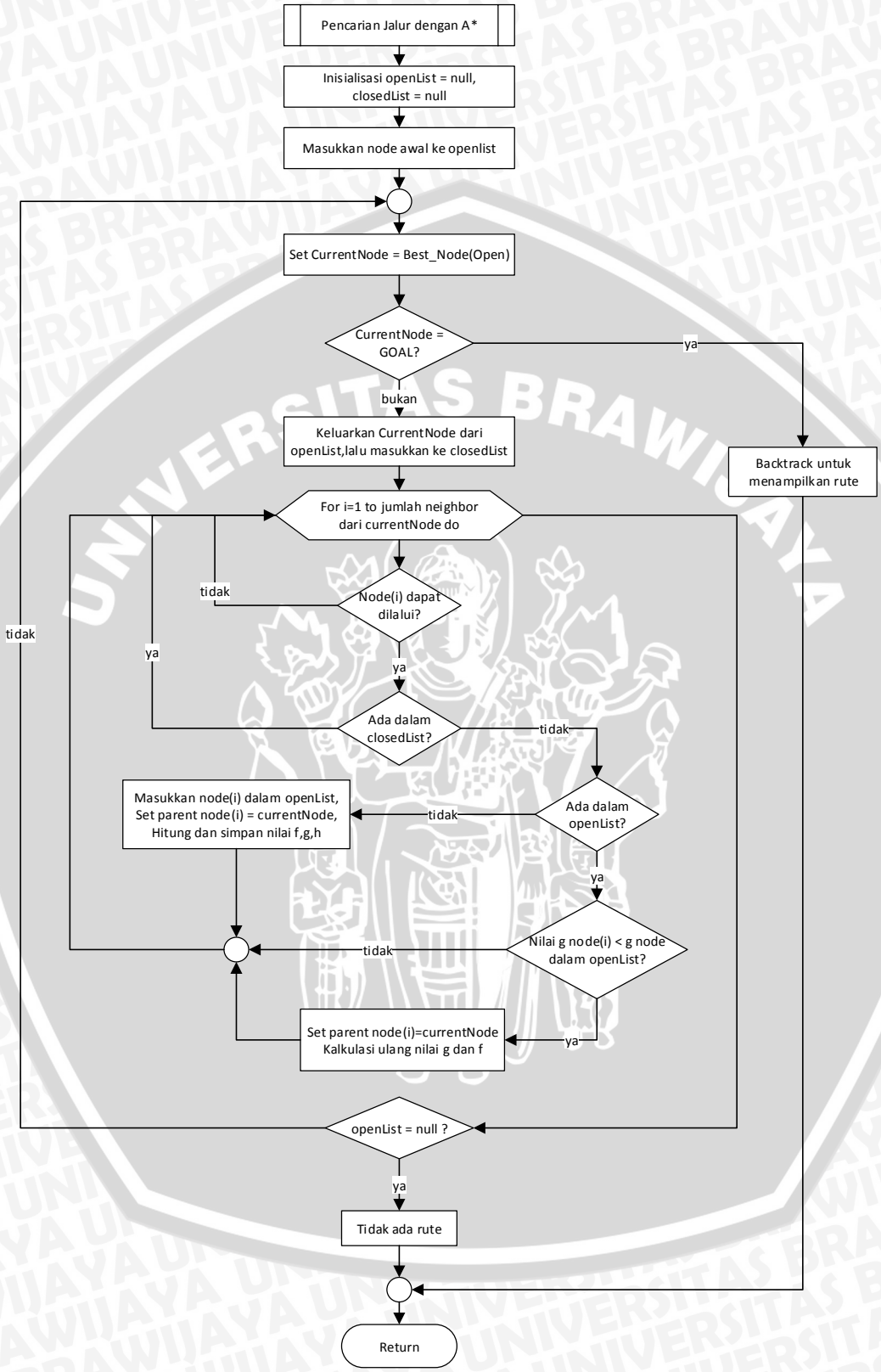
- Jika tidak *walkable* atau jika termasuk *closed list*, maka abaikan.
- Jika tidak ada pada *open list*, tambahkan ke *open list*.
- Jika sudah ada pada *open list*, periksa apakah ini jalan dari *node* ini ke *current node* yang lebih baik dengan menggunakan biaya G sebagai ukurannya. *Node* dengan biaya G yang lebih rendah berarti bahwa ini adalah jalan yang lebih baik. Jika demikian, buatlah *node* ini (*neighbor node*) sebagai *parent node* dari *current node*, dan menghitung ulang nilai G dan F dari simpul ini.

d) Pencarian jalur dihentikan ketika :

- Menambahkan target point ke dalam *closed list*, dalam hal ini jalan telah ditemukan, atau
- Gagal untuk menemukan titik tujuan, dan *open list* kosong. Dalam kasus ini, tidak ada jalan.

3) Simpan jalan. Bekerja mundur dari titik tujuan, pergi dari masing-masing *node* ke *parent node* sampai mencapai titik awal. Dan menghasilkan jalur yang akan diikuti.

Cara bekerja dari algoritma A* akan dijabarkan dalam diagram alir pada gambar 3.7.



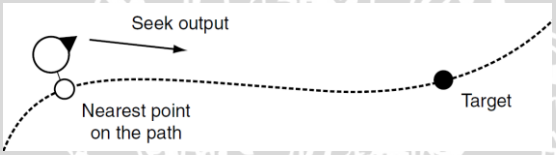
Gambar 3.7 Diagram alir algoritma A*

Sumber : [Perancangan]

3. Mengikuti Jalur

Hasil jalur berupa kumpulan hubungan *node-node* yang sudah tercipta dari pencarian berdasarkan bobot terkecil, akan diikuti oleh AI Agen untuk menuju ke titik tujuan dari titik awal. Berdasarkan penghitungan tersebut, kemudian AI aktor akan mencari tiap-tiap ujung *node*, yang disebut *corner*, yang sudah terbentuk. Proses pencarian *corner* oleh AI Agen disebut *seek*, dimana proses *seek* berusaha mengarahkan agen menuju titik tujuan yang dicapai bukan merupakan titik tujuan akhir, melainkan *corner* atau ujung dari *node*.

AI Agen akan berjalan menuju ke arah corner dan akan merubah tujuan ke *corner* berikutnya hingga mencapai titik tujuan akhir seperti pada gambar 3.8.

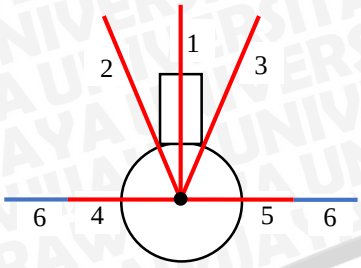


Gambar 3.8 Agen mengikuti jalur tujuan

Sumber : [MIL-09]

3.2.3.2 Perancangan Implementasi *Behaviour Avoid Obstacle*

Untuk mengimplementasikan Reynolds *Steering Obstacle Avoidance behaviour*, yang pertama harus dilakukan adalah memasang *ray* (sinar) pada aktor yang akan digunakan untuk mendeteksi adanya halangan. Terdapat 6 *ray* pada aktor. Jika *Center Ray* mengenai halangan, maka aktor akan mengurangi kecepatan. Jika *UpperLeft Ray* atau *Left Ray* mengenai halangan, maka aktor akan berbelok ke kanan. Jika *UpperRight Ray* atau *Right Ray* mengenai halangan, maka aktor akan berbelok ke kiri. *Extra Ray* digunakan untuk menghindari halangan yang bergerak. Jika *Extra Ray* mengenai halangan yang bergerak, maka aktor akan menambah kecepatan untuk menghindari halangan. Perancangan *ray* pada aktor dapat dilihat pada gambar 3.9.



Keterangan :

1. *Center Ray*
2. *UpperLeft Ray*
3. *UpperRight Ray*
4. *Left Ray*
5. *Right Ray*
6. *Extra Ray*

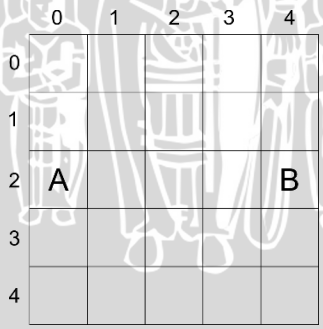
Gambar 3.9 Perancangan ray pada aktor
Sumber : [Perancangan]

3.2.3.3 Perhitungan panjang langkah

Berikut akan dilakukan perhitungan untuk menentukan jarak node panjang langkah menuju node yang tegak lurus dan yang diagonal. Panjang map adalah 200 dan Lebar map adalah 80. Jumlah node adalah 40 x 16. Sehingga setiap node memiliki panjang 200 / 40 = 5 dan memiliki lebar 80 / 16 = 5. Berdasarkan perhitungan tersebut maka dapat diketahui bahwa panjang langkah menuju node yang tegak lurus adalah 5 dan panjang langkah menuju node yang diagonal adalah $\sqrt{5^2 + 5^2} = \sqrt{50} = 7.071$ dan dibulatkan menjadi 7.

3.2.3.4 Perhitungan manual

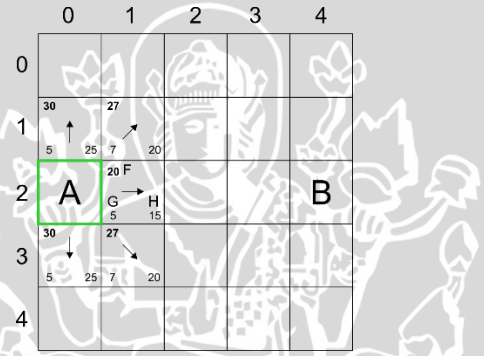
- Contoh Kasus 1 : Tidak terdapat halangan pada peta permainan



Gambar 3.10 Contoh Kasus 1 *pathfinding* pada peta permainan sederhana
Sumber : [Perancangan]

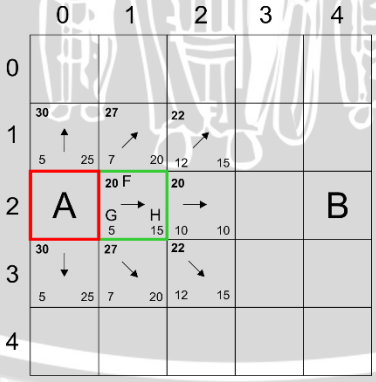
Pada Gambar 3.10 merupakan contoh kasus *pathfinding* pada peta permainan yang tidak terdapat halangan. Pencarian jalur dari titik A(2,0) ke titik B(2,4) akan dilakukan dengan menggunakan algoritma A*. Langkah pertama yang dilakukan adalah memasukan *node* A(2,0) kedalam *open list*, lalu menghitung nilai F dari masing-masing *node* yang ada pada *open list* dan menjadikan *node* dengan

nilai F yang paling rendah sebagai *current node,* karena hanya ada satu *node,* maka titik A akan dijadikan sebagai *current node* dan akan dipindahkan dari *open list* ke *closed list.* Selanjutnya akan ditelusuri semua *node* yang berdekatan dengan *current node* yaitu node (1,0),(1,1),(2,1),(3,0) dan (3,0). Dari masing-masing *node* tetangga tersebut, akan dilakukan pengecekan, jika *node* sudah ada dalam *closed list* maka akan diabaikan, jika *node* belum ada dalam *open list* maka akan dimasukan ke dalam *open list* dan *current node* akan menjadi *parent node* dari *node* tersebut, jika sudah ada dalam *open list* maka akan diperiksa apakah panjang langkah dari node sebelumnya lebih optimal berdasarkan nilai G. Jika iya maka akan dilakukan penghitungan nilai F ulang dan *parent node*-nya diubah menjadi *current node.* Langkah-langkah diatas akan dijabarkan pada gambar 3.11 dan gambar 3.12.



Gambar 3.11 Contoh Kasus 1 langkah awal

Sumber : [Perancangan]

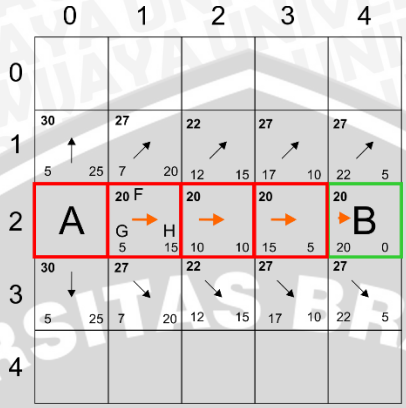


Gambar 3.12 Contoh Kasus 1 langkah selanjutnya

Sumber : [Perancangan]

Proses diatas akan diulang terus-menerus hingga *open list* kosong (jalur tidak ditemukan) atau titik akhir dimasukan kedalam *closed list* (jalur ditemukan).

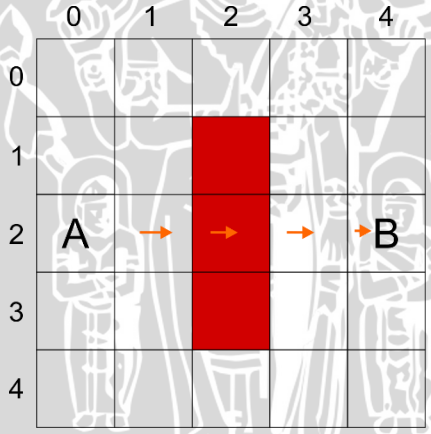
Jika jalur ditemukan maka akan dilakukan *reverse* dari titik akhir hingga titik awal mengikuti *parent node* yang menghasilkan jalur akhir yang akan diikuti pemain seperti pada gambar 3.13. Panjang jalur yang dihasilkan adalah 20.



Gambar 3.13 Contoh Kasus 1 solusi sederhana ditemukan

Sumber : [Perancangan]

- Contoh Kasus 2 : Terdapat halangan dinamis yang tiba-tiba muncul



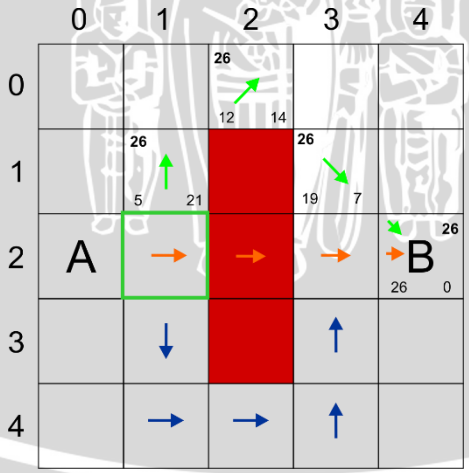
Gambar 3.14 Contoh Kasus 2

Sumber : [Perancangan]

Pada kasus kedua memiliki titik awal dan titik akhir yang sama dengan kasus pertama yaitu titik A(2,0) dan titik B(2,4). Pada kasus ini, telah dihitung jalur optimal dengan menggunakan A* lalu akan muncul halangan dinamis yang tiba-tiba muncul yang akan menutupi jalur yang telah dihitung seperti yang dapat dilihat pada gambar 3.14. Jika menggunakan kalkulasi ulang A*, maka ketika ada halangan dinamis yang menutupi jalur maka akan dilakukan kalkulasi ulang untuk menghitung jalur optimal lainnya. Pada kasus ini, aktor akan mendeteksi

keberadaan obstacle dinamis ketika berada pada *node* (2,1), maka akan dilakukan kalkulasi ulang dan *node* terdekat dengan aktor yaitu *node* (2,1) akan dijadikan sebagai titik awal untuk mencari jalur lalu akan ditemukan jalur optimal yang baru seperti pada panah berwarna hijau pada gambar 3.15.

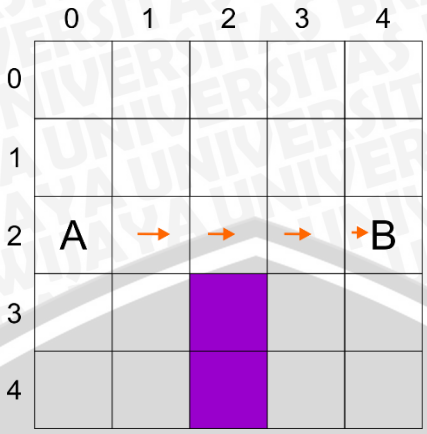
Sedangkan jika menggunakan A* dengan *obstacle avoidance* maka tidak akan dilakukan kalkulasi ulang. Jika aktor mendeteksi halangan dinamis yaitu pada saat berada dekat pada *node* (2,1), maka akan diperiksa *current direction* dari status variabel *obstacle avoidance* dari aktor. *Current direction* ini akan berubah tiap 4 detik yang memiliki nilai antara *left* dan *right*. Misalkan *current direction* dari aktor adalah dan *right*. Maka aktor akan berbelok ke kanan untuk menhindari halangan tanpa melakukan kalkulasi ulang, ketika aktor sudah menghindari halangan, maka aktor akan kembali ke jalur sebelumnya dan akan mengabaikan *node* dari jalur yang terkena halangan dinamis yaitu *node* (2,2). Jalur untuk menghindari halangan ditunjukkan pada panah berwarna biru pada gambar 3.15. Panjang jalur yang dihasilkan menggunakan perhitungan ulang adalah 26+5(ditambah langkah dari titik A) = 31. Sedangkan panjang jalur yang dihasilkan dengan A* dan obstacle avoidance adalah 5*8 (karena 8 langkah tegak lurus) = 40.



Gambar 3.15 Contoh Kasus 2 perhitungan ulang dan dengan *obstacle avoidance*

Sumber : [Perancangan]

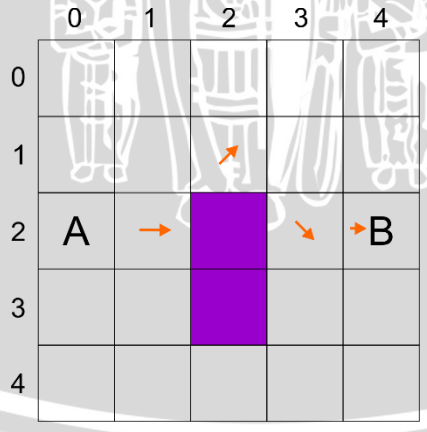
- Contoh Kasus 3 : Terdapat halangan dinamis yang bergerak



Gambar 3.16 Contoh Kasus 3

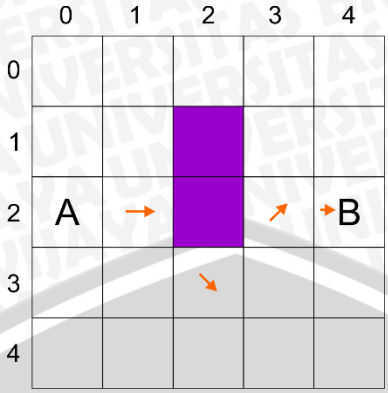
Sumber : [Perancangan]

Pada kasus ini, jalur optimal telah ditemukan yaitu dengan titik awal dan titik akhir yang sama dengan kasus 1 yaitu A(2,0) dan B(2,4) seperti pada gambar 3.16. Tetapi pada peta permainan terdapat halangan yang bergerak yang akan menutupi jalur. Ketika halangan menutupi jalur, dilakukan kalkulasi ulang untuk menghindari jalur maka aktor akan menghitung jalur baru seperti pada gambar 3.17. Selanjutnya ketika jalur yang telah diperbarui ditutupi lagi dengan halangan dinamis, maka akan dilakukan kalkulasi ulang lagi yang ditunjukan pada gambar 3.18.



Gambar 3.17 Contoh Kasus 3 perubahan map pertama

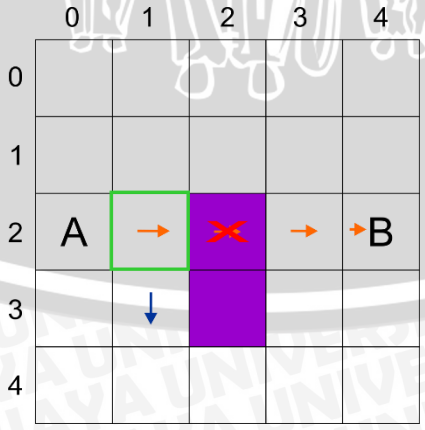
Sumber : [Perancangan]



Gambar 3.18 Contoh Kasus 3 perubahan map kedua

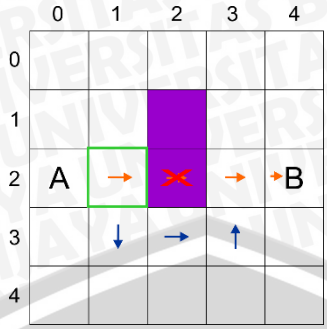
Sumber : [Perancangan]

Sebaliknya jika kasus ini diselesaikan dengan cara menghindari dengan obstacle avoidance tanpa perhitungan ulang, ketika terdapat halangan dinamis bergerak yang menutupi jalur maka akan menghindari halangan ke kiri atau ke kanan. Misalkan *current direction* dari aktor adalah *right*, maka aktor akan berbelok ke kanan untuk menghindari halangan bergerak. Ketika sudah menghindari halangan aktor akan kembali ke jalur utama dengan mengabaikan *node* yang telah terkena halangan dinamis seperti yang dapat dilihat pada gambar 3.19 dan 3.20. Panjang jalur yang dihasilkan dengan perhitungan ulang adalah 5*2(2 langkah tegak lurus)+7*2(2langkah diagonal)=24. Sedangkan panjang jalur yang dihasilkan dengan A* dan obstacle avoidance adalah 5*8 (karena 8 langkah tegak lurus) = 40.



Gambar 3.19 Contoh Kasus 3 perubahan map pertama

Sumber : [Perancangan]



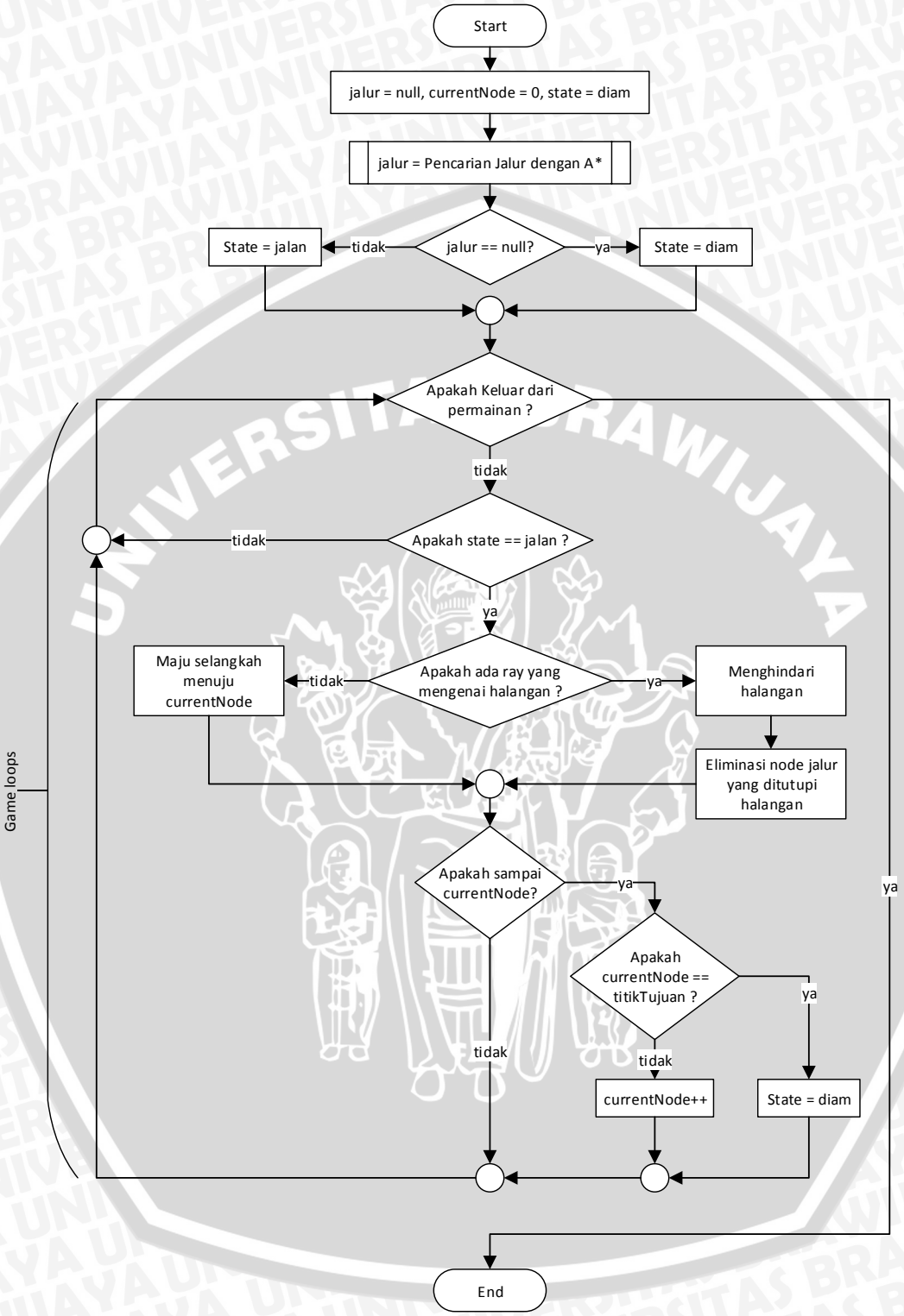
Gambar 3.20 Contoh Kasus 3 perubahan map kedua

Sumber : [Perancangan]

3.2.4 Perancangan Integrasi antara Algoritma A* dengan Reynolds steering obstacle avoidance

Dalam menyelesaikan permasalahan *real-time pathfinding* diperlukan integrasi antara algoritma *long steering* A* untuk *pathfinding* dan algoritma *short steering* Reynolds *Steering Obstacle Avoidance* untuk menghindari halangan yang dinamis sehingga NPC dapat berjalan lancar dari titik awal ke titik tujuan pada peta permainan yang terdapat halangan dinamis di dalamnya.

Untuk mengintegrasikan algoritma *long steering* A* dan algoritma *short steering* Reynolds *Steering Obstacle Avoidance* yang pertama harus dilakukan adalah mengimplementasikan A* yang digunakkan mencari jalur dari titik awal ke titik tujuan. Jika jalur sudah ditemukan, NPC akan berjalan menelusuri jalur. Selanjutnya akan diimplementasikan Reynold *Steering* Obstacle Avoidance pada NPC. Reynold *Steering* Obstacle Avoidance akan memasang beberapa *ray* (sinar) pada NPC untuk mendeteksi adanya halangan didepan NPC. Jika *ray* yang dipancarkan NPC mengenai halangan (terdapat halangan dinamis yang menutupi jalur), maka NPC akan berhenti mengikuti jalur lalu akan berjalan keluar jalur untuk menghindari halangan dinamis yang menghalangi jalan NPC. Semua *node-node* dari jalur yang ditutupi atau terkena dengan halangan dinamis akan dieliminasi dari jalur. Ketika *ray* pada NPC tidak mengenai atau mendeteksi halangan lagi, maka NPC akan kembali masuk ke jalurnya dan akan lanjut menelusuri jalurnya hingga ke titik tujuan. Integrasi antara algoritma A* dan algoritma Reynolds *Steering Obstacle Avoidance* akan dijabarkan dalam diagram alir pada gambar 3.21.

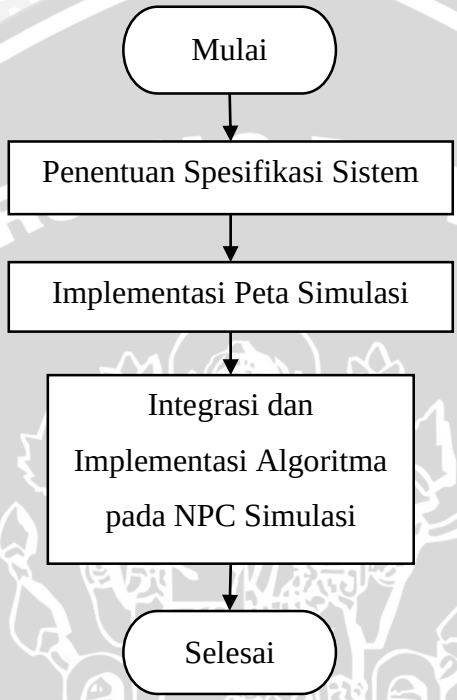


Gambar 3.21 Diagram Alir Perancangan Integrasi antara Algoritma A* dengan Reynolds steering obstacle avoidance

Sumber : [Perancangan]

3.3 Implementasi

Bab implementasi terdiri dari empat tahap, yaitu spesifikasi sitem, implementasi peta simulasi, implementasi NPC simulasi dan integrasi antara algoritma A* dengan Reynolds *steering obstacle avoidance*. Tahap-tahap implementasi dijabarkan dalam diagram alir dalam Gambar 3.22.



Gambar 3.22 Diagram Alir Implementasi

Sumber : [Metode Penelitian]

3.3.1 Penentuan Spesifikasi Sistem

Subbab ini menjabarkan spesifikasi dari lingkungan sistem tempat simulasi dijalanakan. Spesifikasi sitem yang dimaksud adalah lingkungan perangkat keras dan perangkat lunak yang digunakan dalam menjalankan simulasi. Subbab ini bertujuan untuk mengetahui dalam spesifikasi tersebut, seberapa baik simulasi dijalankan untuk pengolahan data selanjutnya.

3.3.2 Implementasi Peta Simulasi

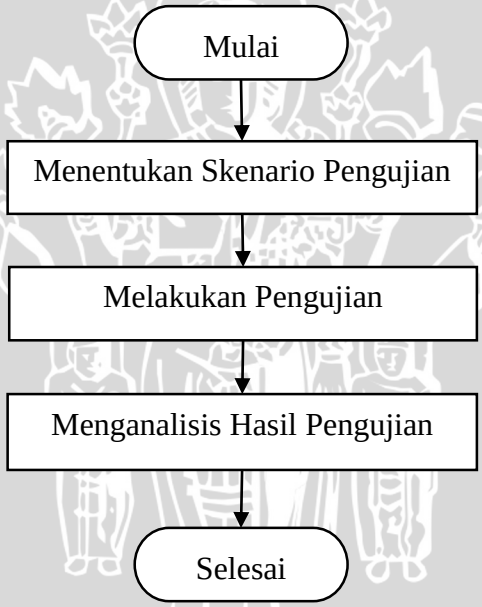
Pada subbab ini akan diimplementasikan peta yang telah dirancang sebelumnya yang akan digunakan dalam simulasi.

3.3.3 Integrasi dan Implementasi Algoritma pada NPC Simulasi

Subbab ini berisi integrasi antara algoritma *pathfinding* A* dan Reynolds *steering behaviours obstacle avoidance* untuk menyelesaikan permasalahan real-time pathfinding serta penerapan dari algoritma yang telah dirancang dan akan dipakai dalam simulasi. Metode-metode yang sudah dirancang dalam bab perancangan juga akan dijabarkan dalam subbab ini.

3.4 Pengujian dan Analisis

Pengujian yang dilakukan berdasarkan implementasi yang telah dibuat berdasarkan metriks pengujian yang telah ditentukan, yaitu FPS dan panjang lintasan. Alur dari pengujian dan analisis dijabarkan dalam diagram alir pada gambar 3.23.



Gambar 3.23 Diagram alir Pengujian dan Analisis

Sumber : [Metodologi Penelitian]

3.4.1 Menentukan Skenario Pengujian

Skenario pengujian algoritma dilakukan dengan 2 alternatif, yang pertama adalah pengujian FPS (*Frame per Second*) dengan membandinegkan antara FPS saat menggunakan *obstacle avoidance* dengan FPS saat menggunakan perhitungan jalur ulang (*recalculate path)* tanpa *obstacle avoidance*, yang kedua adalah dengan memunculkan beberapa aktor dan menentukan dan mencari jalur dari titik awal ke

titik akhir yang diacak lalu meletakan beberapa halangan dinamis (*dynamic obstacle*) pada posisi yang diacak setelah masing-masing aktor sudah ditentukan lintasannya dengan perhitungan A*, lalu mengukur panjang lintasan dengan membandingkan antara panjang lintasan dengan perhitungan A* yang menghindari halangan dinamis tersebut menggunakan Reynolds *Steering Obstacle Avoidance* dan Panjang lintasan yang dihitung ulang setelah munculnya halangan yang dinamis.

3.4.2 Melakukan Pengujian

Pada skenario pertama akan direkam FPS dan Jumlah aktor yang ada. Pada skenario yang kedua akan direkam panjang lintasan dari masing-masing aktor.

3.4.3 Menganalisis Hasil Pengujian

Hasil pengujian skenario pertama akan dianalisis performa antara saat menggunakan *obstacle avoidance* dengan saat menggunakan perhitungan jalur ulang (*recalculate path)* tanpa *obstacle avoidance* berdasarkan hasil rekaman FPS. Sedangkan hasil pengujian skenario akan dianalisis lintasan yang optimal antara panjang lintasan dengan perhitungan A* yang menghindari halangan dinamis tersebut menggunakan Reynolds *Steering Obstacle Avoidance* dan Panjang lintasan yang dihitung ulang setelah munculnya halangan yang dinamis.

BAB IV
IMPLEMENTASI

Bab ini membahas mengenai implementasi algoritma berdasarkan hasil yang telah didapatkan dari analisis kebutuhan dan proses perancangan. Pembahasan terdiri dari penjelasan tentang spesifikasi sistem, batasan-batasan dalam implementasi, implementasi dan implementasi rancangan algoritma.

4.1 Penentuan Spesifikasi

Hasil analisis kebutuhan dan perancangan perangkat lunak yang telah dijelaskan pada bab 3 menjadi acuan untuk melakukan implementasi *real-time pathfinding* menggunakan A* dan Reynolds steering *obstacle avoidance* pada permainan komputer yang dapat berfungsi sesuai dengan kebutuhan. Spesifikasi sistem untuk implementasi algoritma dijelaskan pada spesifikasi perangkat keras dan perangkat lunak.

4.1.1 Spesifikasi Perangkat Keras

Spesifikasi perangkat keras yang dipakai dalam proses pengembangan dijelaskan dalam Tabel 4.1.

Tabel 4.1 Spesifikasi Perangkat Keras

Nama Komponen	Spesifikasi
Processor	Intel® Core™ i7-2670QM @ 2.20GHz (8 CPUs), ~2.2GHz
Memory	4096MB RAM
Hardisk	750GB
Graphic Card	NVIDIA GeForce GTX 560M, 2GB

Sumber : [Implementasi]

4.1.2 Spesifikasi Perangkat Lunak

Spesifikasi perangkat lunak yang dipakai dalam proses pengembangan dijelaskan dalam Tabel 4.2.

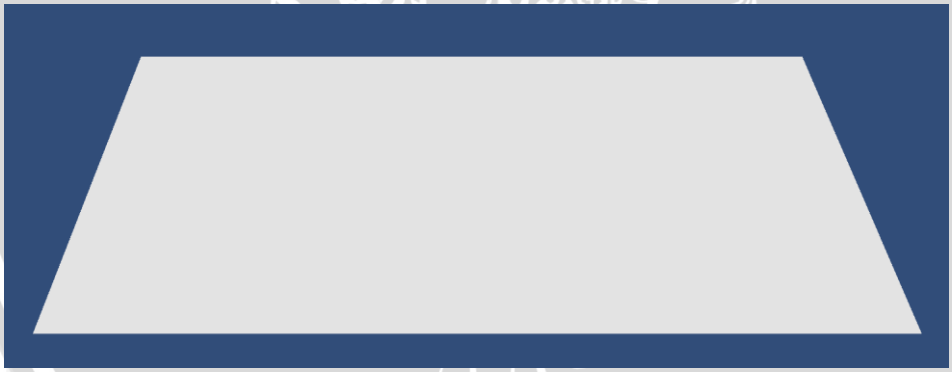
Tabel 4.2 Spesifikasi Perangkat Lunak

Perangkat Lunak	Spesifikasi
Operating System	Microsoft Windows 8.1 Pro 64-bit (6.3, Build 9600)
DirectX Version	DirectX 11
Programming Language	C#
Software Development Kit	Unity Version 4.6.0f3

Sumber : [Implementasi]

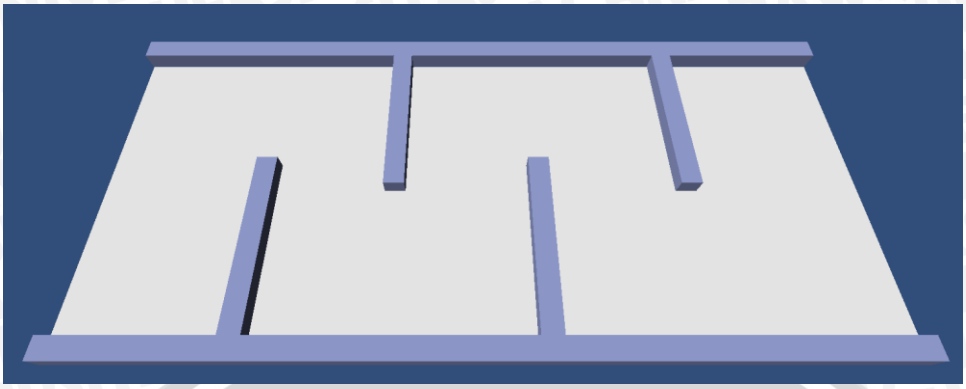
4.2 Implementasi Peta Simulasi

Implementasi peta permainan untuk simulasi *real-time pathfinding* akan lansung dirancang dalam Unity3D pada *editor scene view*. Objek-objek yang akan digunakkan antara lain adalah *terrain* dan *cube*. *Terrain* akan digunakan sebagai lantai. *Cube* akan digunakan sebagai halangan statis dan halangan dinamis. Hasil implementasi dari masing-masing peta ditunjukkan pada gambar 4.1, 4.2, 4.3, 4.4, 4.5 dan 4.6.



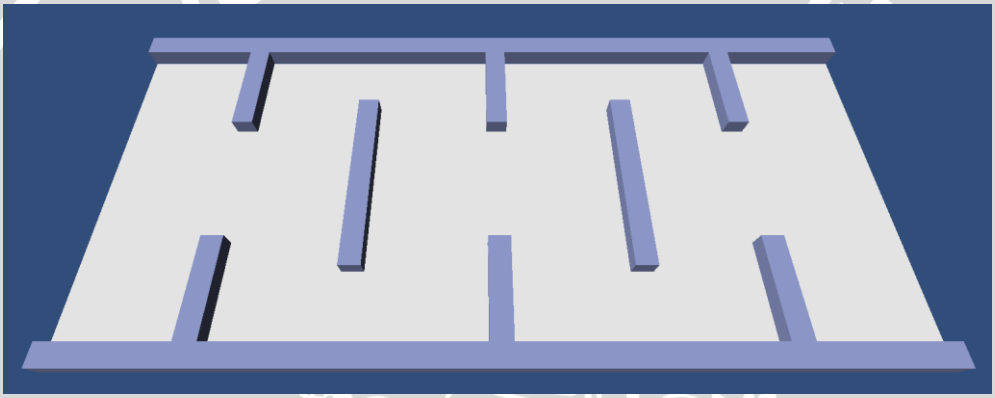
Gambar 4.1 Implementasi Peta 1 Simulasi 1

Sumber : [Implementasi]



Gambar 4.2 Implementasi Peta 2 Simulasi 1

Sumber : [Implementasi]

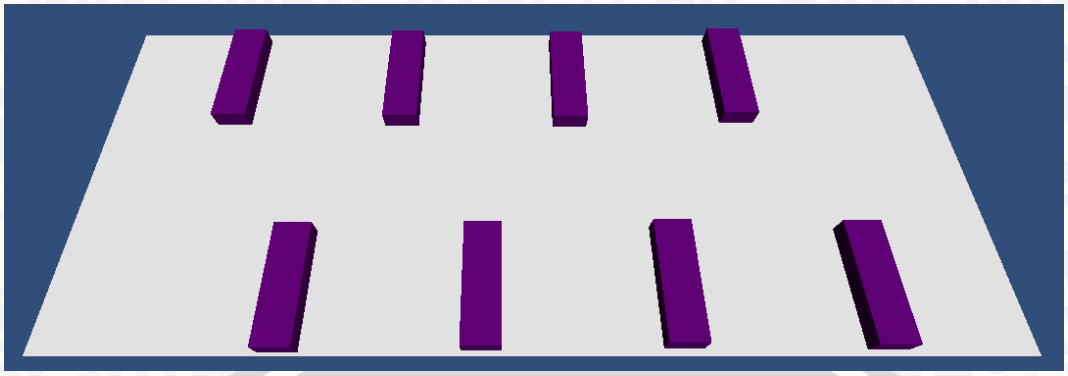


Gambar 4.3 Implementasi Peta 3 Simulasi 1

Sumber : [Implementasi]

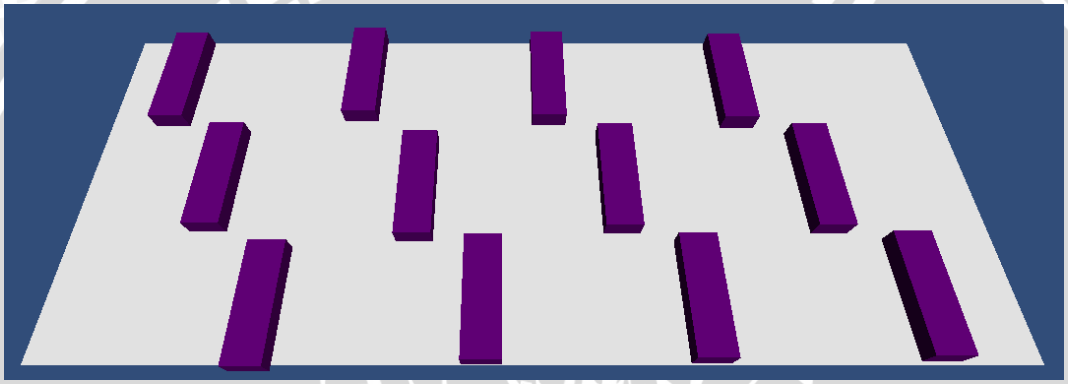
Gambar 4.4 Implementasi Peta 1 Simulasi 2

Sumber : [Implementasi]



Gambar 4.5 Implementasi Peta 2 Simulasi 2

Sumber : [Implementasi]



Gambar 4.6 Implementasi Peta 3 Simulasi 2

Sumber : [Implementasi]

Pada *terrain* yang terdapat pada masing-masing peta akan dilakukan generateGrid untuk menghasilkan *node-node* berupa *regular grids* berbentuk persegi yang nantinya akan digunakkan untuk melakukan pencarian jalur. Fungsi untuk *generate grid* akan dijelaskan pada tabel 4.3.

Tabel 4.3 Fungsi generateGrid()

Fungsi generateGrid()	
1	for i = 0 to width do
2	_tmpList = new List<Transform>();
3	_nodeStatesList = new List<NodeState>();
4	
5	GameObject parentGO = new GameObject("ROW_" + i.ToString());
6	parentGO.transform.parent = _parentTransform;
7	_parentTransforms.Insert(i, parentGO.transform);

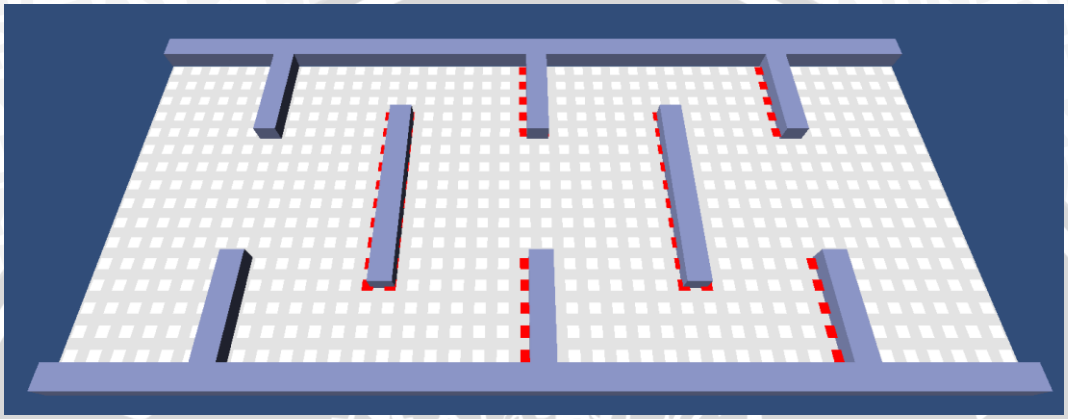
```
8
9     for j = 0 to height do
10        GameObject go = GameObject.Instantiate(nodePrefab) as
GameObject;
11        go.transform.name =
go.transform.name.Replace("(Clone)", "[" + i.ToString() + ",
" + j.ToString() + "]");
12        go.transform.parent = _parentTransforms[i];
13
14        Vector3 pos = GetWorldPosition(new Vector2(i, j));
15        pos += new Vector3(CellSize / distance,
terrain.SampleHeight(GetWorldPosition(new Vector2(i, j))) +
1, CellSize / distance);
16        go.transform.position = pos;
17
18        go.transform.localScale = new Vector3(CellSize /
distance, 0.1f, CellSize / distance);
19
20        _tmpList.Insert(j, go.transform);
21        _nodeStatesList.Insert(j,
go.GetComponent<NodeState>());
22    end for
23    _gridTransforms.Insert(i, _tmpList);
24    _nodeStatesGrid.Insert(i, _nodeStatesList);
25 end for
```

Sumber : [Implementasi]

Penjelasan :

1. Pada baris 1, merupakan perulangan sebanyak width. Nilai width merupakan hasil bagi antara lebar *terrain* dengan ukuran *grid / node.*
2. Pada baris 2-3, merupakan inisialisasi untuk tmpList dan nodeStateList. tmpList sebagai list sementara yang akan menyimpan *node* dari masing-masing baris dan nodeStateList akan menyimpan nodeState dari masing-masing *node.*
3. Pada baris 5-7, membuat object parent agar semua gameObject tertata rapi pada *Hierarcy Panel.*
4. Pada baris 9, merupakan perulangan sebanyak *height*. Nilai *height* merupakan hasil bagi antara panjang *terrain* dengan ukuran *node*.
5. Pada baris 10-12, pembuatan dan penamaan *node*.
6. Pada baris 14-16, penentuan posisi terhadap *node.*
7. Pada baris 18, merupakan menentuan ukuran dari *node.*

8. Pada baris 20-21, menambahkan *node* kedalam tmpList dan *node state* kedalam nodeStateList.
9. Pada baris 23-24, menambahkan tmpList kedalam gridTransforms dan nodeStateList kedalam nodeStateGrid.



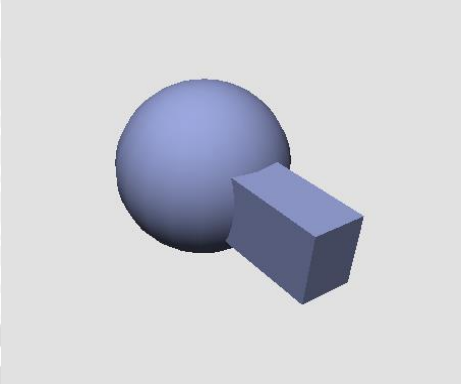
Gambar 4.7 Peta yang sudah memiliki node-node

Sumber : [Implementasi]

Pada gambar 4.7 akan ditunjukkan contoh peta yang sudah memiliki *node-node* yang didapatkan dengan menjalankan fungsi generateGrid. Pada peta dapat dilihat bahwa terdapat *node* yang berwarna putih dan merah. Warna merah menandakan bahwa *node* tidak dapat dilalui dan warna putih menandakan bahwa *node* dapat dilalui.

4.3 Integrasi dan Implementasi Algoritma pada NPC Simulasi

Implementasi algoritma AI kedalam NPC diperlukan untuk menjadikan NPC bergerak sesuai dengan perilaku yang sudah dirancang sebelumnya. Oleh karena itu, algoritma AI yang sudah dirancang akan diterapkan ke Aktor. Pada gambar 4.8 akan ditunjukan model 3D dari aktor.



Gambar 4.8 Model 3D dari Aktor

Sumber : [Implementasi]

4.3.1 Integrasi antara Algoritma A* dengan Reynolds *steering obstacle avoidance*

Integrasi algoritma A* dengan Reynold Steering *Obstacle Avoidance* untuk akan ditunjukkan dalam tabel 4.4.

Tabel 4.4 Pseudocode untuk integrasi A* dengan Reynolds Steering Obstacle Avoidance

Pseudocode untuk integrasi A* dengan Reynolds Steering Obstacle Avoidance	
1	`def Start() :`
2	`  path = FindPath();`
3	
4	`def Update() :`
5	`  PathFollow();`
6	`  AvoidObstacle();`
7	

Sumber : [Implementasi]

Penjelasan :

Pada baris 1, method Start merupakan method yang hanya dijalankan sekali pada awal *load Scene*. Pertama-tama akan dipanggil FindPath untuk mencari jalur dengan algoritma A*. Lalu pada baris 4, method Update akan dipanggil terus menerus. Pada method update akan dipanggil PathFollow() dan AvoidObstacle() untuk mengikuti jalur dan menghindari halangan.

4.3.2 Implementasi algoritma A* *Behaviour Find Path & Path Follow*

Implementasi algoritma A* untuk *behaviour Find Path* pada NPC akan ditunjukkan dalam tabel 4.5. Implementasi algoritma Reynold *Steering Path Following* untuk *behaviour Path Follow* pada NPC akan ditunjukkan dalam tabel 4.6.

Tabel 4.5 Pseudocode algoritma A*

Pseudocode algoritma A*	
Deklarasi openList, closedList, nodeAdjacents Deskripsi Input : startIndex, endIndex	
	Proses
1	Start
2	If startIndex == endIndex then
3	return emptyPath
4	end if
5	openList.Add(startIndex)
6	do
7	currentNode = FindLowestF()
8	closedList.Add(currentNode)
9	openList.Remove(currentNode)
10	foreach n in nodeAdjacents do
11	if n not walkable || ContainsNodeIndex(openList, n) || ContainsNodeIndex(closedList, n) then
12	nodeAdjacents.remove(n)
13	end if
14	end foreach
15	foreach indx in nodeAdjacents do
16	node = GetNodeByIndex(openList, indx);
17	if node == null then
18	node = indx
19	openList.Add(node)
20	if node.nodeIndex.i != currentNode.nodeIndex.i && node.nodeIndex.j != currentNode.nodeIndex.j then
21	node.G = 14 + currentNode.G
22	else then
23	node.G = 10 + currentNode.G
24	end if
25	node.H = CalcHeuristicPrice(node)
26	node.F = node.G + node.H
27	else then

```
28          if node.nodeIndex.i != currentNode.nodeIndex.i &&
node.nodeIndex.j != currentNode.nodeIndex.j then
29              if node.G > 14 + currentNode.G then
30                  node.G = 14 + currentNode.G
31                  node.F = node.G + node.H
32                  node.parent = currentNode
33              end if
34          else then
35              if node.G > 14 + currentNode.G then
36                  node.G = 14 + currentNode.G
37                  node.F = node.G + node.H
38                  node.parent = currentNode
39              end if
40          end if
41       end if
42    end foreach
43  end while openList.Count > 0 && !ContainsNodeIndex(openList,
endIndex)
44  if !ContainsNodeIndex(openList, endIndex) then
45     return emptyPath
46  end if
47  node = GetNodeByIndex(openList, endIndex)
48  do
49     pathNodes.Add(node)
50     node = node.parent
51  end while node!=null
52  pathNodes.Reverse()
53  return pathNodes
```

Sumber : [Implementasi]

Penjelasan :

1. Pada baris 2-4, jika titik awal sama dengan titik akhir maka akan mengembalikan Path kosong.
2. Pada baris 5, menambahkan titik awal ke openlist.
3. Pada baris 6, merupakan awal dari pengulangan do while.
4. Pada baris 7, memanggil method FindLowestF() yang akan mengembalikan node yang memiliki nilai F terendah lalu dijadikan currentNode.
5. Pada baris 8-9, memindahkan currentNode dari openList ke closedList.
6. Pada baris 10-14, menghapus semua node yang berdekatan yang tidak bisa dilalui atau terdapat di openList atau terdapat di closedList.

7. Pada baris 15-42, menghitung nilai F, G, dan H dari semua node yang berdekatan dengan currentNode.
8. Pada baris 43, merupakan penutup dari do while yang akan dijalankan selama openList tidak kosong dan selama tidak ada *node* tujuan di dalam open list.
9. Pada baris 44-46, jika tidak ada *node* tujuan di dalam openList maka akan mengembalikan Path kosong.
10. Pada baris 47, mencari *node* akhir di dalam openList lalu disimpan didalam variabel node.
11. Pada baris 48-51, dilakukan penelusuran *parent node* dari node tujuan hingga node awal lalu disimpan kedalan variable pathNodes.
12. Pada baris 52, membalik urutan pathNodes.
13. Pada baris 53, mengembalikan nilai pathNodes.

Tabel 4.6 Pseudocode untuk path following

Pseudocode untuk path following	
Deklarasi state, currentNodeID, blockedPath Deskripsi Input : path	
1	**def Update() :**
2	if state == IDLE then
3	do nothing
4	else if state == MOVING then
5	MoveToward()
6	end if
7	
8	**def MoveToward() :**
9	if currentNodeID > path.Length then
10	currentNodeID = 0
11	state = IDLE
12	return
13	end if
14	if blockedPath.Contains(path[currentNodeID])
15	currentNodeID++;
16	end if
17	dest = path[currentNodeID].position
18	offset = dest - transform.position

```
19   if offset > reachDistance then
20     offset = offset.normalized
21     transform.Translate(transform.forward * speed *
         Time.deltaTime)
22     traveledDistanceOB += speed * Time.deltaTime
23     if !isAvoiding && !isCoolingDown then
24       lookRot = Quaternion.LookRotation(offset)
25       transform.rotation =
         Quaternion.Slerp(transform.rotation, lookRot, rotSpeed
         * Time.deltaTime)
26     end if
27   else then
28     currentNodeID++;
29   end if
30   if isAvoiding then
31     if !isCoolingDown then
32       Invoke("stopCoolingDown", coolingDownTime)
33       isCoolingDown = true
34     end if
35     foreach obstacle in obstacles do
36       for i = currentNodeID to path.Length - 1 do
37         if path[i] intersects with obstacle then
38           if !blockedPath.Contains(path[i]) then
39             blockedPath.Add(path[i]);
40           end if
41         end if
42       end if
43     end foreach
44   end if
```

Sumber : [Implementasi]

Penjelasan :

1. Pada baris 1, merupakan method update yang akan dijalankan berulang-ulang.
2. Pada baris 2-6 , jika state sama dengan IDLE maka tidak akan melakukan apa-apa. Jika state sama dengan MOVING maka akan dipanggil method MoveToward()
3. Pada baris 9-13 dilakukan pengecekan apakah currentNodeID lebih besar dari panjang lintasan. Jika iya, nilai currentNodeID akan diubah menjadi 0 dan state akan diubah menjadi IDLE. CurrentNodeID menyimpan index dari *node* path yang akan ditelusuri.

4. Pada baris 14-16, akan dilakukan pengecekan apakan *node* dari path yang ingin ditelusuri terdapat didalang blockedPath. Jika iya, maka *node* ini akan dilewati atau tidak ditelusuri.
5. Pada baris 17, mengambil posisi dari currentNode dan disimpan kedalam variabel dest.
6. Pada baris 18, menghitung offset antara posisi node dan posisi aktor.
7. Pada baris 19-29, jika offset lebih kecil dari reachDistance maka akan lanjut menelusuri *node* selanjutnya. Jika tidak maka aktor akan berjalan menuju currentNode selama aktor tidak mendeteksi halangan dan tidak sedang *cooling down*.
8. Pada baris 30-44, akan dilakukan pengecekan apakah aktor mendeteksi adanya tabrakan. Jika iya, maka semua *node* yang mengenai halangan dinamis akan dimasukkan ke blockedPath dan variabel isCoolingDown diganti nilainya menjadi true. Variabel IsCoolingDown akan kembali mejadi false sesuai dengan waktu coolingDownTime.

4.3.3 Implementasi *Behaviour Avoid Obstacle*

Implementasi algoritma Reynold Steering *Obstacle Avoidance* untuk *behaviour Avoid Obstacle* pada NPC akan ditunjukkan dalam tabel 4.7.

Tabel 4.7 Pseudocode untuk Obstacle Avoidance

Pseudocode untuk obstacle avoidance	
Deklarasi hitCenter, hitUpperLeft, hitUpperRight, hitLeft, hitRight, distance	
1	**def update() :**
2	checkRayCast()
3	
4	**def checkRayCast():**
5	hitCenter = Physics.Raycast(transform.position, transform.forward, out hitCenterRay, distance)
6	hitUpperLeft = Physics.Raycast(transform.position, transform.forward - transform.right / 2, out hitUpperLeftRay, distance)
7	hitUpperRight = Physics.Raycast(transform.position, transform.forward + transform.right / 2, out hitUpperRightRay, distance)
8	hitLeft = Physics.Raycast(transform.position, -transform.right, out hitLeftRay, distance)
9	hitRight = Physics.Raycast(transform.position, transform.right, out hitRightRay, distance)

```
10
11    if hitCenter|| hitUpperLeft || hitUpperRight || hitLeft
      || hitRight then
12      bool isHitMovingWallCast = false;
13      if (hitCenter) then
14        float resist = Mathf.Abs((hitCenterRay.distance -
      distance) / distance ) * speed)
15        transform.Translate(transform.forward * -resist *
      Time.deltaTime)
16      end if
17      if hitUpperLeft then
18        transform.eulerAngles += new Vector3(0, 200 *
      Time.deltaTime, 0)
19      else if hitUpperRight then
20        transform.eulerAngles -= new Vector3(0, 200 *
      Time.deltaTime, 0)
21      end if
22      if hitLeft && !hitUpperRight then
23        if hitLeftRay.distance < (distance / 2 ) then
24          transform.eulerAngles += new Vector3(0, 200 *
      Time.deltaTime, 0)
25        else then
26          if hitLeftRay.transform.name == "MovingWall" then
27            float resist = Mathf.Abs((hitLeftRay.distance -
      distance) / distance) * speed * 2
28            transform.Translate(transform.forward * resist *
      Time.deltaTime)
29          end if
30          isHitMovingWallCast = true
31        end if
32      end if
33      if hitRight && !hitUpperLeft then
34        if hitRightRay.distance < (distance / 2 ) then
35          transform.eulerAngles -= new Vector3(0, 200 *
      Time.deltaTime, 0)
36        else then
37          if hitRightRay.transform.name == "MovingWall" then
38            float resist = Mathf.Abs((hitRightRay.distance -
      distance) / distance) * speed * 2
39            transform.Translate(transform.forward * resist *
      Time.deltaTime)
40          end if
41          isHitMovingWallCast = true
42        end if
43      end if
44      if !isHitMovingWallCast then
45        isAvoiding = true
46      else then
47        isAvoiding = false
48      end if
49    else then
50      isAvoiding = false
51    end if
```

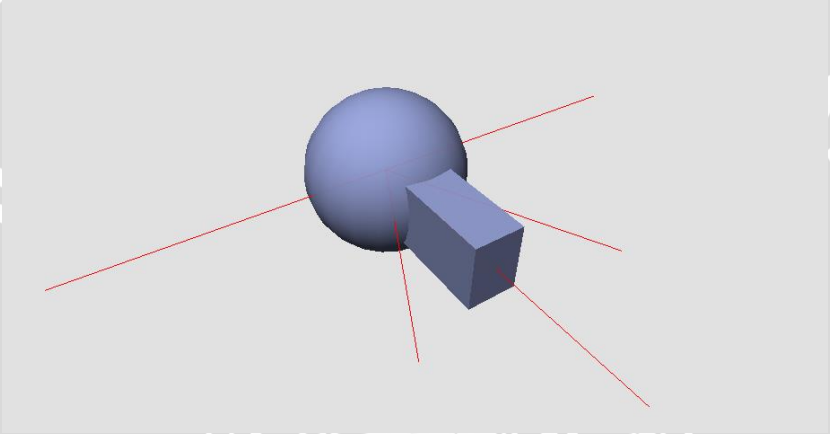
Sumber : [Implementasi]

Penjelasan :

1. Pada baris 1, merupakan method update yang akan dijalankan berulang-ulang.
2. Pada baris 4-9 , akan dipasangkan raycast pada bagian depan, kiri depan, kanan depan, kiri, kanan aktor untuk mendeteksi halangan.
3. Pada baris 11, dilakukan pengecekan apakah ada halangan yang mengenai raycast. Jika iya, maka akan dijalankan baris 12-47.
4. Pada baris 12, merupakan inisialisasi dan deklarasi variabel isHitMovingWallCast = false. Variabel ini digunakan untuk menetukan apakah raycast mendeteksi halangan dinamis yang bergerak atau tidak.
5. Pada baris 13-16, jika raycast depan yang mendeteksi halangan maka akan aktor akan mundur.
6. Pada baris 17-21, jika raycast kiri depan mendeteksi halangan maka aktor akan berbelok ke kanan. Sebaliknya, jika raycast kanan depan mendeteksi halangan maka aktor akan berbelok ke kiri.
7. Pada baris 22-32, jika raycast kiri mendeteksi halangan dan raycast kanan depan tidak mendeteksi halangan maka akan dilakukan mengecekan lagi apakah raycast yang terkena halangan melebihi setengah dari panjang ray. Jika iya, maka aktor akan berbelok ke kanan. Jika tidak, maka variabel isHitMovingWallCast akan diubah nilainya menjadi true dan jika halangan yang dideteksi merupakan halangan bergerak maka aktor akan menambah kecepatan.
8. Pada baris 33-43, jika raycast kanan mendeteksi halangan dan raycast kiri depan tidak mendeteksi halangan maka akan dilakukan mengecekan lagi apakah raycast yang terkena halangan melebihi setengah dari panjang ray. Jika iya, maka aktor akan berbelok ke kiri. Jika tidak, maka variabel isHitMovingWallCast akan diubah nilainya menjadi true dan jika halangan yang dideteksi merupakan halangan bergerak maka aktor akan menambah kecepatan.

9. Pada baris 44-48, jika ray tidak terkena halangan dinamis yang bergerak maka isAvoiding bernilai true. Jika sebaliknya, maka isAvoiding bernilai false.
10. Pada baris 49-51, jika tidak ada halangan yang terdeteksi maka isAvoiding bernilai false.

Pada gambar 4.9 ditunjukkan model 3D dari aktor dengan *ray* yang akan digunakan untuk mendeteksi halangan dinamis.



Gambar 4.9 Model 3D dari Aktor dengan ray untuk mendeteksi halangan

Sumber : [Implementasi]

BAB V
PENGUJIAN DAN ANALISIS

Pada bab ini dilakukan proses pengujian dan analisis terhadap implementasi *real-time pathfinding* menggunakkan A* dan Reynold *Steering Obstacle Avoidance.* Pengujian dilakukan dengan menguji tiap skenario dan pencatatan hasil berdasarkan parameter pengujian. Parameter pengujian berupa jarak yang ditempuh NPC dan juga rata-rata FPS yang dibutuhkan saat menjalankan *game*.

5.1 Spesifikasi Pengujian

Spesifikasi perangkat keras yang digunakan dalam uji coba adalah prosesor Intel Core i7, harddisk 750 GB, memori 4GB RAM, kartu grafis NVIDIA GeForce GTX 560M, 2GB. Sedangkan spesifikasi perangkat lunak yang digunakan adalah Microsoft Windows 8.1 Pro 64-bit, *game engine* Unity v4.6.

5.2 Menentukan Skenario Pengujian

Pengujian dilakukan untuk mengetahui valid atau tidaknya *behaviour* yang sudah dirancang dengan implementasinya. Selain itu, pengujian juga dilakukan untuk mengetahui performa dari sistem saat menjalankan *real-time pathfinding.* Performa dari sistem dapat ditinjau dari rata-rata FPS per simulasi. Uji coba dibagi menjadi 2 skenario :

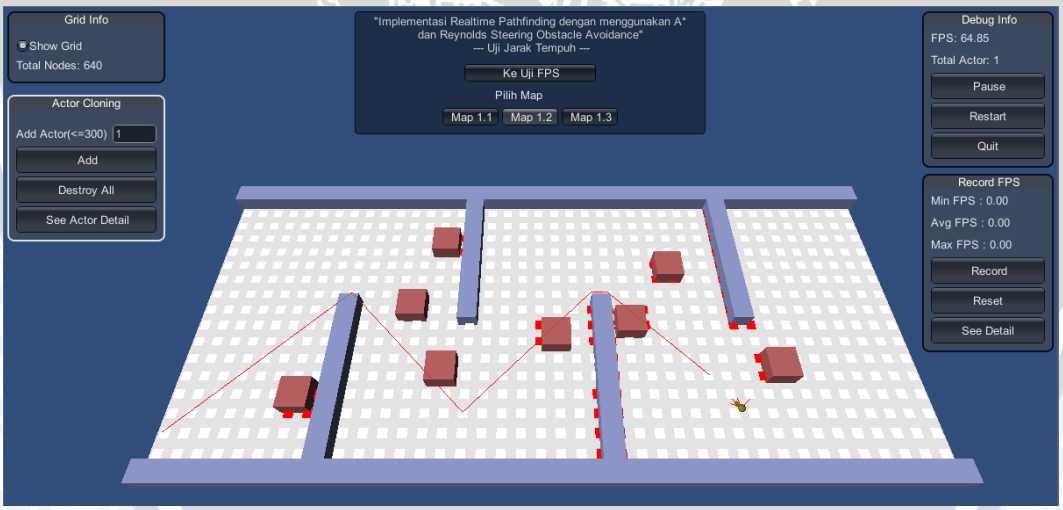
1. Uji coba skenario 1, dilakukan untuk mengetahui valid atau tidaknya *behaviour* yang sudah dirancang dengan implementasinya dan untuk menguji jarak tempuh yang dilalui jika dibandingkan antara *obstacle avoidance* dan penghitungan ulang jalur setelah sudah ada halangan dinamis dengan melakukan uji coba dalam peta permainan yang berbeda-beda.
2. Uji coba skenario 2, dilakukan untuk menguji algoritma yang lebih efisien diantara A* dengan *obstacle avoidance* dan kalkulasi jalur ulang A* berdasarkan rata-rata FPS dengan melakukan uji coba dalam peta permainan yang berbeda-beda.

5.3 Melakukan Pengujian

Pada subbab ini akan dijabarkan uji coba dan hasil pengujian dari skenario 1 dan skenario 2.

5.3.1 Pengujian Skenario 1

Uji coba skenario 1 dilakukan dengan cara meletakan beberapa halangan statis pada peta lalu akan dimuncukan beberapa aktor yang akan berjalan dari titik awal ke titik akhir. Setelah itu akan dimunculkan beberapa halangan dinamis yang akan menutupi jalur. Tujuan dari skenario ini untuk untuk mengetahui valid atau tidaknya *behaviour* yang sudah dirancang dengan implementasinya dan untuk menguji jarak tempuh yang dilalui jika dibandingkan antara *obstacle avoidance* dan penghitungan ulang jalur setelah sudah ada halangan dinamis. Akan dilakukan 10 kali percobaan dari masing-masing map yang akan merekam koordinat awal, koordinat akhir,apakah sampai tujuan, panjang jalur dengan obstacle avoidance, panjang jalur dengan penghitungan ulang dari aktor. Contoh pengujian skenario 1 dapat dilihat pada gambar 5.1.



Gambar 5.1 Contoh Pengujian Skenario 1

Sumber : [Pengujian dan Analisis]

Hasil dari pengujian skenari 1 pada masing-masing peta dapat dilihat pada tabel 5.1, 5.2 dan 5.3.

Tabel 5.1 Tabel Hasil Pengujian Skenario 1 Peta 1

No. Tes	Koordinat awal	Koordinat tujuan	Sampai tujuan	Panjang Jalur		Selisih panjang jalur antara OB dengan PU
				dengan Obstacle Avoidance (OB)	dengan Penghitungan jalur ulang (PU)	
1	7,0	4,39	Ya	219.24	209.50	9.74
2	15,0	0,39	Ya	311.09	237.78	73.31
3	1,0	8,39	Ya	235.06	230.21	4.85
4	3,0	12,39	Ya	246.94	221.92	25.01
5	0,39	4,0	Ya	261.58	219.85	41.72
6	0,0	3,39	Ya	234.51	221.92	12.58
7	12,0	3,39	Ya	255.83	221.92	33.90
8	4,0	4,39	Ya	222.86	215.71	7.15
9	8,39	6,0	Ya	280.24	211.57	68.67
10	14,39	8,0	Ya	241.60	219.85	21.75

Sumber : [Pengujian dan Analisis]

Tabel 5.2 Tabel Hasil Pengujian Skenario 1 Peta 2

No. Tes	Koordinat awal	Koordinat tujuan	Sampai tujuan	Panjang Jalur		Selisih panjang jalur antara OB dengan PU
				dengan Obstacle Avoidance (OB)	dengan Penghitungan jalur ulang (PU)	
1	7,0	12,39	Ya	274.46	257.99	16.47
2	0,39	15,0	Ya	380.95	293.35	87.60
3	3,0	2,39	Ya	275.69	249.71	25.98
4	7,0	4,39	Ya	267.07	257.99	9.08
5	10,0	8,39	Ya	237.19	232.28	4.91
6	15,0	15,39	Ya	338.52	260.06	78.46
7	7,39	4,0	Ya	265.77	262.63	3.14
8	13,39	3,0	Ya	270.11	248.85	21.26
9	11,39	3,0	Ya	306.52	252.99	53.53
10	11,0	8,39	Ya	269.98	248.49	21.49

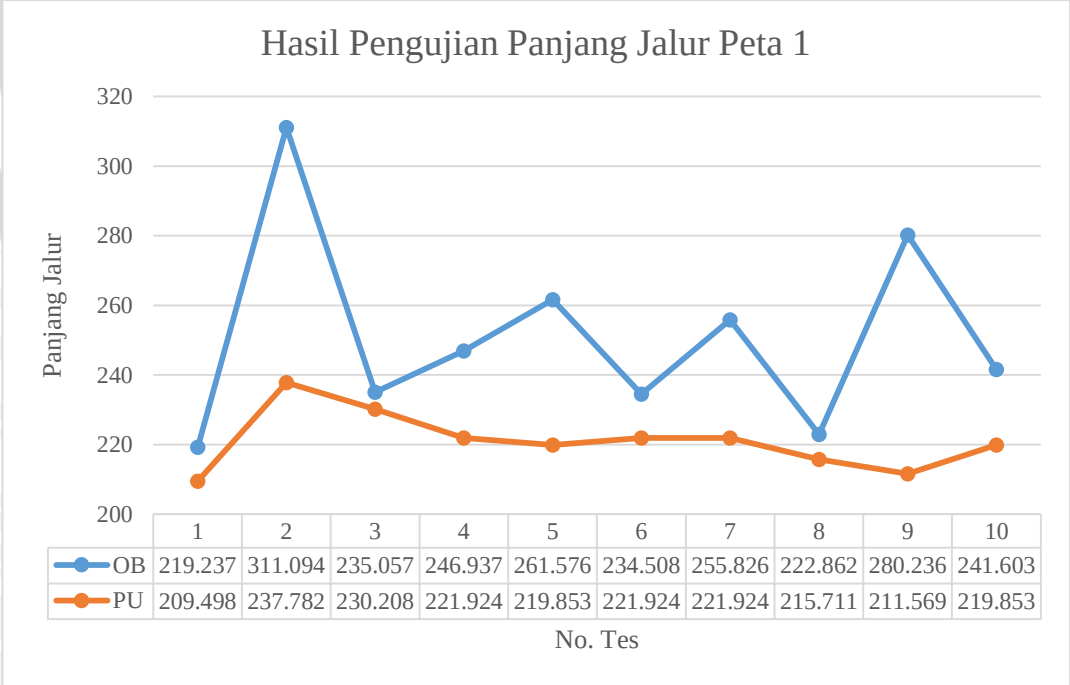
Sumber : [Pengujian dan Analisis]

Tabel 5.3 Tabel Hasil Pengujian Skenario 1 Peta 3

No. Tes	Koordinat awal	Koordinat tujuan	Sampai tujuan	Panjang Jalur		Selisih panjang jalur antara OB dengan PU
				dengan Obstacle Avoidance (OB)	dengan Penghitungan jalur ulang (PU)	
1	8,0	6,39	Ya	259.46	248.85	10.61
2	14,0	3,39	Ya	265.86	254.35	11.51
3	14,39	9,0	Ya	278.01	252.63	25.38
4	3,0	12,39	Ya	292.77	246.78	45.99
5	7,0	3,39	Ya	274.71	243.49	31.22
6	7,39	13,0	Ya	275.78	247.63	28.14
7	10,39	4,0	Ya	251.27	243.49	7.78
8	14,39	4,0	Ya	386.72	254.71	132.01
9	7,39	0,0	Ya	296.16	266.78	29.38
10	12,0	10,39	Ya	374.50	250.56	123.94

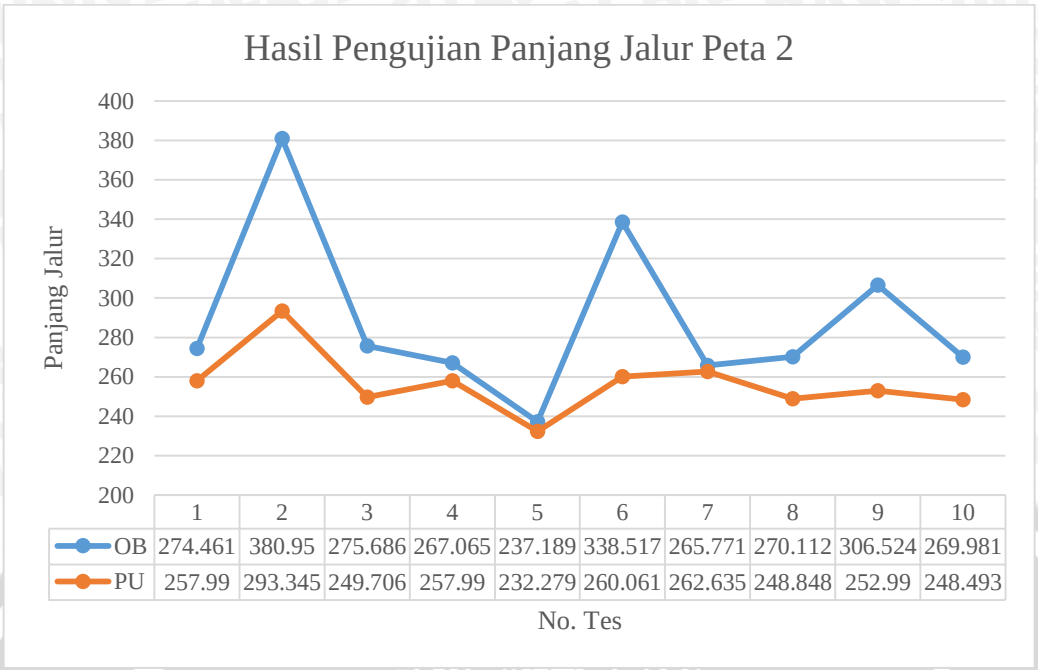
Sumber : [Pengujian dan Analisis]

Agar lebih mudah untuk melihat data maka tampilan grafik dari hasil pengujian skenario 1 ditampilkan pada gambar 5.2, 5.3 dan 5.4.



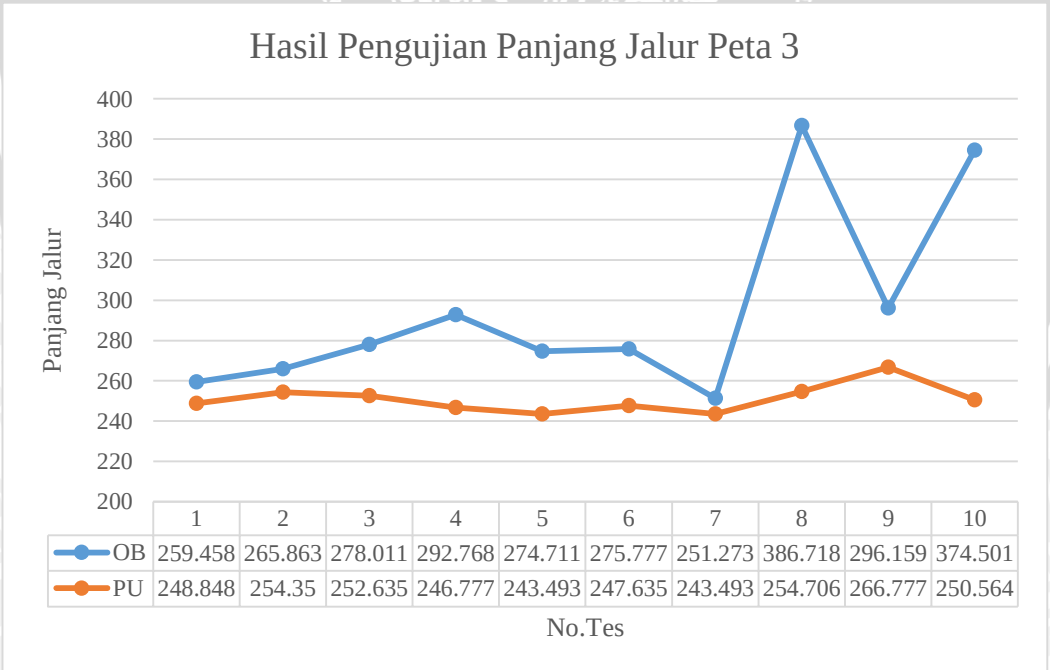
Gambar 5.2 Grafik Hasil Pengujian Skenario 1 Peta 1

Sumber : [Pengujian dan Analisis]



Gambar 5.3 Grafik Hasil Pengujian Skenario 1 Peta 2

Sumber : [Pengujian dan Analisis]

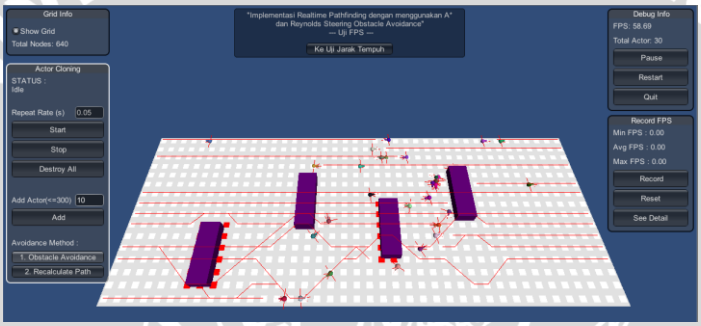


Gambar 5.4 Grafik Hasil Pengujian Skenario 1 Peta 3

Sumber : [Pengujian dan Analisis]

5.3.2 Pengujian Skenario 2

Uji coba skenario 2 dengan cara meletakan beberapa halangan dinamis bergerak yang akan bergerak lambat lalu akan dimuncukan beberapa aktor yang akan berjalan dari titik awal ke titik akhir sesuai hasil perhitungan jalurnya. Ada 2 jenis aktor yang akan dimunculkan yaitu aktor yang menggunakan *obstacle avoidance* dan aktor yang menggunakan kalkulasi jalur ulang. Tujuan dari skenario ini untuk menguji algoritma yang lebih efisien diantara keduanya berdasarkan rata-rata FPS. Akan dilakukan 10 kali percobaan dari masing-masing map dengan memunculkan jumlah aktor yang berbeda-beda. Contoh pengujian skenario 1 dapat dilihat pada gambar 5.5.



Gambar 5.5 Contoh Pengujian Skenario 2

Sumber : [Pengujian dan Analisis]

Hasil dari pengujian skenari 2 dapat dilihat pada tabel 5.4, 5.5, dan 5.6. Dan agar lebih mudah untuk melihat data maka tampilan grafik dari hasil pengujian skenario 2 ditampilkan pada gambar 5.6, 5.7, dan 5.8.

Tabel 5.4 Tabel Hasil Pengujian Skenario 2 Peta 1

No. Tes	Jumlah Aktor	Rata-rata FPS	
		dengan Obstacle Avoidance (OB)	dengan Penghitungan jalur ulang (PU)
1	20	60.39	60.28
2	40	60.23	59.98
3	60	60.21	55.65
4	80	60.13	54.72
5	100	60.07	51.99
6	200	53.09	12.87
7	400	34.25	9.88

8	600	24.05	7.66
9	800	18.23	3.83
10	1000	14.7	3.62
Rata-rata		**44.535**	**32.048**

Sumber : [Pengujian dan Analisis]

Tabel 5.5 Tabel Hasil Pengujian Skenario 2 Peta 2

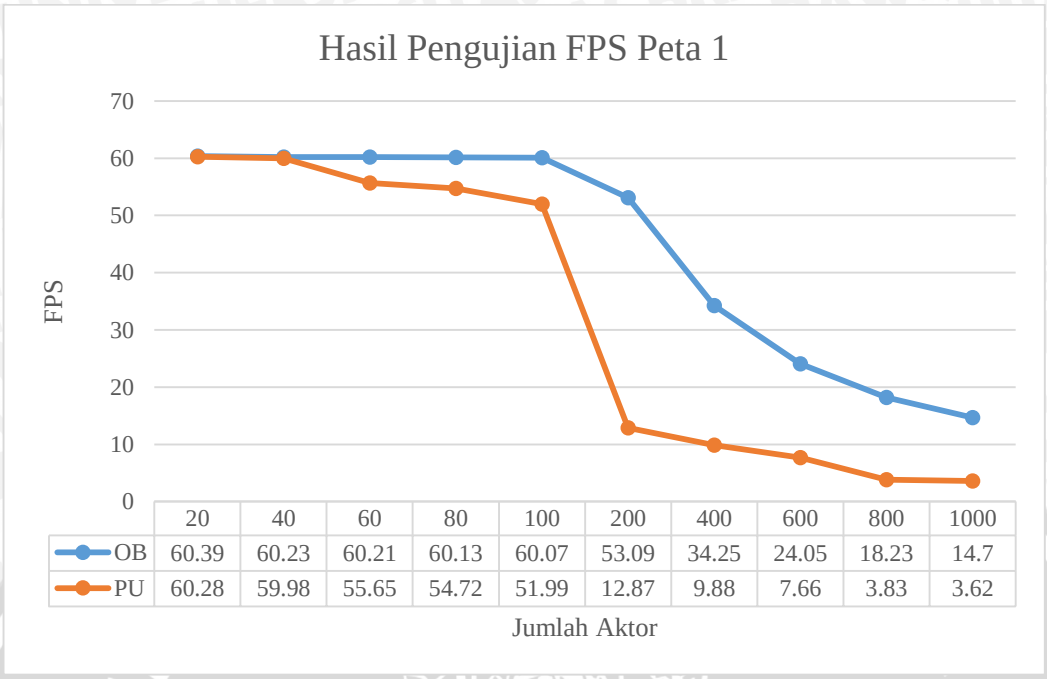
No. Tes	Jumlah Aktor	Rata-rata FPS	
		dengan Obstacle Avoidance (OB)	dengan Penghitungan jalur ulang (PU)
1	20	60.28	58.98
2	40	60.2	58.78
3	60	59.97	39.26
4	80	56.08	20.91
5	100	55.31	16.53
6	200	45.7	8.56
7	400	29.52	4.22
8	600	24.94	3
9	800	13.76	3
10	1000	11.38	3
Rata-rata		**41.714**	**21.624**

Sumber : [Pengujian dan Analisis]

Tabel 5.6 Tabel Hasil Pengujian Skenario 2 Peta 3

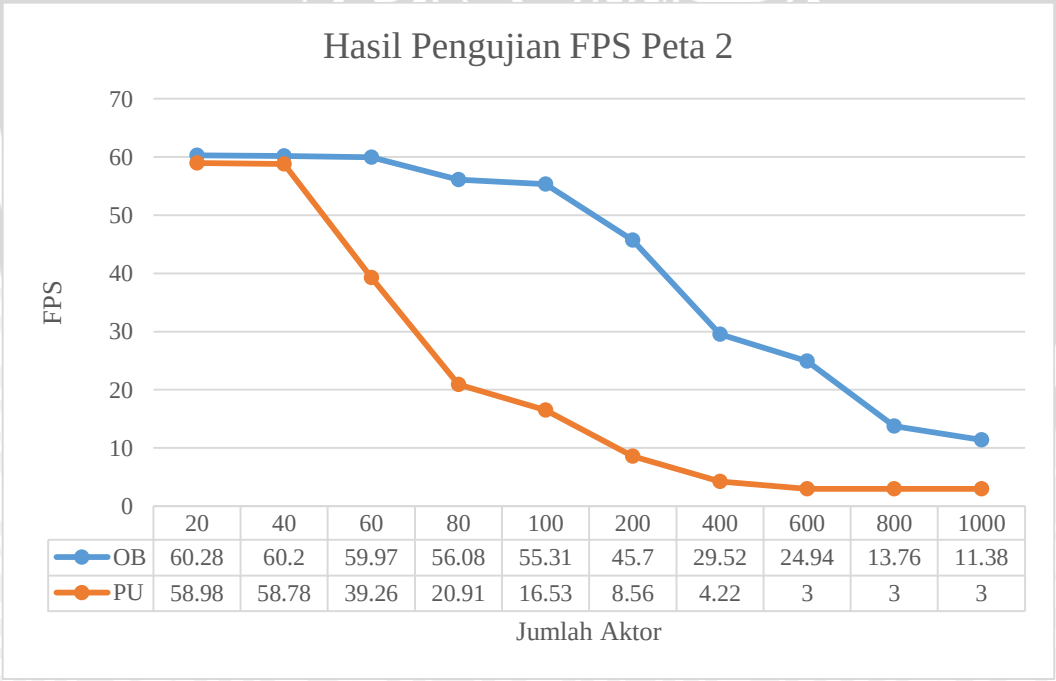
No. Tes	Jumlah Aktor	Rata-rata FPS	
		dengan Obstacle Avoidance (OB)	dengan Penghitungan jalur ulang (PU)
1	20	60.28	59.23
2	40	60.15	49.88
3	60	60.13	10.85
4	80	58.98	6.25
5	100	58.09	4.69
6	200	36.33	3
7	400	21.32	3
8	600	14.09	3
9	800	11.01	3
10	1000	8.8	3
Rata-rata		**38.918**	**14.59**

Sumber : [Pengujian dan Analisis]



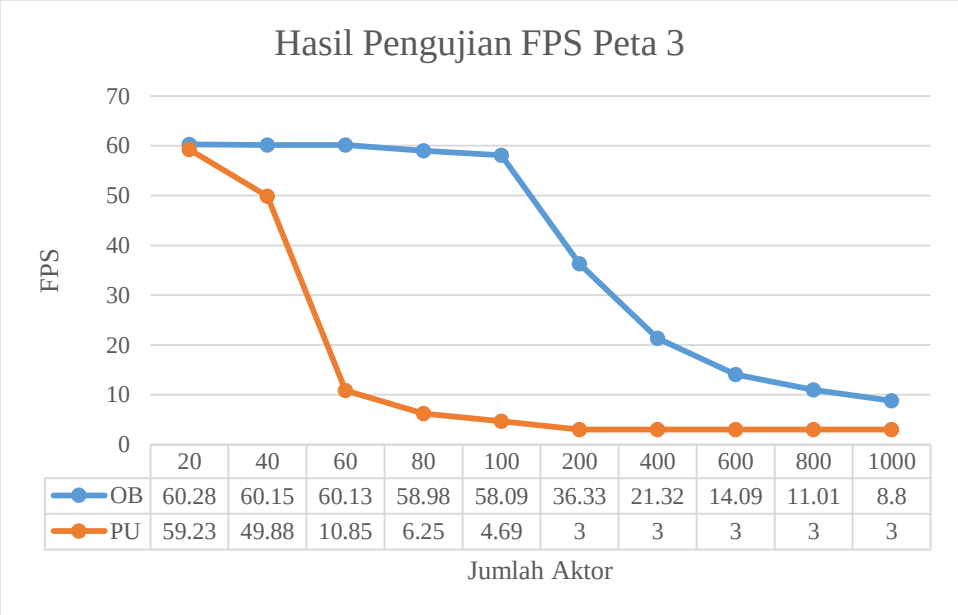
	20	40	60	80	100	200	400	600	800	1000
OB	60.39	60.23	60.21	60.13	60.07	53.09	34.25	24.05	18.23	14.7
PU	60.28	59.98	55.65	54.72	51.99	12.87	9.88	7.66	3.83	3.62

Gambar 5.6 Grafik Hasil Pengujian Skenario 2 Peta 1
Sumber : [Pengujian dan Analisis]



	20	40	60	80	100	200	400	600	800	1000
OB	60.28	60.2	59.97	56.08	55.31	45.7	29.52	24.94	13.76	11.38
PU	58.98	58.78	39.26	20.91	16.53	8.56	4.22	3	3	3

Gambar 5.7 Grafik Hasil Pengujian Skenario 2 Peta 2
Sumber : [Pengujian dan Analisis]



	20	40	60	80	100	200	400	600	800	1000
OB	60.28	60.15	60.13	58.98	58.09	36.33	21.32	14.09	11.01	8.8
PU	59.23	49.88	10.85	6.25	4.69	3	3	3	3	3

Gambar 5.8 Grafik Hasil Pengujian Skenario 2 Peta 3
Sumber : [Pengujian dan Analisis]

5.4 Analisis Hasil Pengujian

Analisis bertujuan untuk mendapatkan kesimpulan dari hasil pengujian implementasi *real-time pathfinding* menggunakkan A* dan Reynold *Steering Obstacle Avoidance.* Proses analisis terhadap hasil pengujian untuk skenario 1 dilihat dari aktor yang sampai ke tujuan dan panjang jalur yang dihasilkan dan untuk skenario 2 dilihat dari rata-rata FPS yang dihasilkan.

Pada uji coba skenario 1, dari hasil pengujian panjang jalur peta 1 yang merupakan peta tanpa halangan statis dapat dilihat pada tabel 5.1 dan grafik 5.2 bahwa 10 dari 10 uji coba yang dijalankan, semua aktor berhasil mencapai titik tujuan dan panjang jalur yang dihasilkan dengan A* dan *obstacle avoidance* (OB) lebih panjang dibandingkan dengan panjang jalur dengan perhitungan ulang A* (PU). Selisih antara panjang jalur yang dihasilkan dengan OB dan PU tidak menentu, ada yang memiliki selisih yang pendek dan ada yang memiliki selisih yang panjang. Selisih yang terpendek terdapat pada uji coba ke-3 yaitu 4.848 dan selisih yang terpanjang terdapat pada uji coba ke-2 yaitu 73.312. Rata-rata dari selisih antara panjang jalur yang dihasilkan dengan OB dan PU dari 10 uji coba pada peta 1 adalah 29.86.

Dari hasil pengujian panjang jalur peta 2 pada skenario 1 yang pada peta sudah terdapat beberapa halangan statis dapat dilihat pada tabel 5.2 dan grafik 5.3 bahwa 10 dari 10 uji coba yang dijalankan, semua aktor berhasil mencapai titik tujuan dan panjang jalur yang dihasilkan dengan A* dan *obstacle avoidance* (OB) lebih panjang dibandingkan dengan panjang jalur dengan perhitungan ulang A* (PU). Selisih antara panjang jalur yang dihasilkan dengan OB dan PU tidak menentu seperti pada hasil pengujian skenario 1 peta 1. Selisih yang terpendek terdapat pada uji coba ke-7 yaitu 3.136 dan selisih yang terpanjang terdapat pada uji coba ke-2 yaitu 87.604. Rata-rata dari selisih antara panjang jalur yang dihasilkan dengan OB dan PU dari 10 uji coba pada peta 2 adalah 32.19.

Dari hasil pengujian panjang jalur peta 3 pada skenario 1 yang merupakan peta dengan halangan statis terbanyak dapat dilihat pada tabel 5.3 dan grafik 5.4 bahwa 10 dari 10 uji coba yang dijalankan, semua aktor berhasil mencapai titik tujuan dan panjang jalur yang dihasilkan dengan A* dan *obstacle avoidance* (OB) lebih panjang dibandingkan dengan panjang jalur dengan perhitungan ulang A* (PU). Selisih antara panjang jalur yang dihasilkan dengan OB dan PU tidak menentu seperti pada hasil pengujian skenario 1 peta 1 dan peta 2. Selisih yang terpendek terdapat pada uji coba ke-7 yaitu 7.78 dan selisih yang terpanjang terdapat pada uji coba ke-8 yaitu 132.01. Rata-rata dari selisih antara panjang jalur yang dihasilkan dengan OB dan PU dari 10 uji coba pada peta 3 adalah 44.59.

Dari data-data pada hasil pengujian skenario 1 dapat diketahui bahwa dari semua hasil pengujian (30 uji coba), semua aktor berhasil mencapai titik tujuan sehingga dapat disimpulkan bahwa implementasi *real-time pathfinding* dengan menggunakan A* dan Reynold *Steering Obstacle Avoidance* berhasil dilakukan. Selain itu, dari semua hasil pengujian dapat dilihat bahwa panjang jalur yang dihasilkan menggunakan A* dan *obstacle avoidance* lebih panjang dibandingkan dengan perhitungan ulang A*. Dan rata-rata terendah dari selisih antara panjang jalur yang dihasilkan antara dengan OB dan PU yaitu 29.86 terdapat pada peta 1 yang tidak terdapat halangan statis didalamnya sedangkan rata-rata tertinggi yaitu 44.59 terdapat pada peta 3 yang memiliki halangan statis terbanyak didalamnya.

Rata-rata dari panjang jalur dari semua hasil pengujian (30 uji coba) adalah 278.3 (OB) dan 242.8 (PU).

Pada uji coba skenario 2, dari hasil pengujian FPS peta 1 yang merupakan peta dengan halangan dinamis yang paling sedikit dapat dilihat pada tabel 5.4 dan grafik pada gambar 5.6 bahwa 10 dari 10 uji coba yang dilakukan dengan memunculkan jumlah aktor yang berbeda-beda, rata-rata FPS yang dihasilkan lebih tinggi saat menggunakan A* dengan *obstacle avoidance* dibandingkan dengan menggunakan perhitungan jalur ulang dengan A*. Rata-rata FPS yang dihasilkan dengan menggunakan A* dengan *obstacle avoidance* berada diatas batas minimum untuk mengelabui mata manusia yaitu diatas 20 FPS ketika aktor berjumlah 20 sampai dengan 600. Sedangkan rata-rata FPS yang dihasilkan dengan menggunakan perhitungan jalur ulang dengan A* berada diatas batas minimum ketika aktor berjumlah 20 sampai dengan 100. Rata-rata dari rata-rata FPS dari 10 hasil uji coba pada hasil pengujian FPS peta 1 dengan menggunakan A* dengan *obstacle avoidance* adalah 44.535, dan dengan mengunakan perhitungan jalur ulang dengan A* adalah 32.04.

Dari hasil pengujian FPS peta 2 pada skenario 2 yang merupakan peta dengan jumlah halangan dinamis yang lebih banyak dibanding peta 1 dapat dilihat pada tabel 5.5 dan grafik pada gambar 5.7 bahwa 10 dari 10 uji coba yang dilakukan dengan memunculkan jumlah aktor yang berbeda-beda, rata-rata FPS yang dihasilkan lebih tinggi saat menggunakan A* dengan *obstacle avoidance* dibandingkan dengan menggunakan perhitungan jalur ulang dengan A*. Rata-rata FPS yang dihasilkan dengan menggunakan A* dengan *obstacle avoidance* berada diatas batas minimum untuk mengelabui mata manusia yaitu diatas 20 FPS ketika aktor berjumlah 20 sampai dengan 600. Sedangkan rata-rata FPS yang dihasilkan dengan menggunakan perhitungan jalur ulang dengan A* berada diatas batas minimum ketika aktor berjumlah 20 sampai dengan 80. Rata-rata dari rata-rata FPS dari 10 hasil uji coba pada hasil pengujian FPS peta 2 dengan menggunakan A* dengan *obstacle avoidance* adalah 41.714, dan dengan mengunakan perhitungan jalur ulang dengan A* adalah 21.624.

Dari hasil pengujian FPS peta 3 pada skenario 2 yang merupakan peta dengan jumlah halangan dinamis yang paling banyak dibanding peta-peta sebelumnya dapat dilihat pada tabel 5.6 dan grafik pada gambar 5.8 bahwa 10 dari 10 uji coba yang dilakukan dengan memunculkan jumlah aktor yang berbeda-beda, rata-rata FPS yang dihasilkan lebih tinggi saat menggunakan A* dengan *obstacle avoidance* dibandingkan dengan menggunakan perhitungan jalur ulang dengan A*. Rata-rata FPS yang dihasilkan dengan menggunakan A* dengan *obstacle avoidance* berada diatas batas minimum untuk mengelabui mata manusia yaitu diatas 20 FPS ketika aktor berjumlah 20 sampai dengan 400. Sedangkan rata-rata FPS yang dihasilkan dengan menggunakan perhitungan jalur ulang dengan A* berada diatas batas minimum ketika aktor berjumlah 20 sampai dengan 40. Rata-rata dari rata-rata FPS dari 10 hasil uji coba pada hasil pengujian FPS peta 3 dengan menggunakan A* dengan *obstacle avoidance* adalah 38.918, dan dengan mengunakan perhitungan jalur ulang dengan A* adalah 14.59.

Dari data-data pada hasil pengujian skenario 2 dapat diketahui bahwa dari semua hasil pengujian (30 uji coba), dapat diketahui bahwa kinerja CPU dalam *real-time pathfinding* lebih optimal jika menggunakan A* dan *obstacle avoidance* dibandingkan dengan menggunakan perhitungan jalur ulang dengan A* berdasarkan rata-rata FPS. Selain itu, rata-rata dari rata-rata FPS dari hasil pengujian pada masing-masing peta yaitu 44.535 FPS (OB) dan 32.048 FPS (PU) pada peta 1, 41.714 FPS (OB) dan 21.624 (PU) pada peta 2, 38.92 FPS (OB) dan 14.59 (PU) pada peta 3 dapat dilihat terdapat penurunan FPS sehingga dapat disimpulkan semakin banyak halangan pada peta permainan maka semakin rendah kinerja CPU berdasarkan rata-rata FPS. Rata-rata keseluruhuan pada semua pengujian pada skenario 2 adalah 41.7 (OB) dan 22,7 (PU).

BAB VI
PENUTUP

6.1 Kesimpulan

Berdasarkan hasil pengujian dan analisis yang dilakukan terhadap implementasi dari *real-time pathfinding* menggunakan A* dan Reynolds *Steering obstacle avoidance* pada permainan komputer, dapat diambil kesimpulan sebagai berikut :

1. Implementasi *real-time pathfinding* berhasil dilakukan dengan integrasi antara algoritma *long steering* A* untuk *pathfinding* dan algoritma *short steering* Reynolds *Steering Obstacle Avoidance* untuk menghindari halangan dinamis.
2. Dengan *real-time pathfinding* menggunakan A* dan Reynolds *steering obstacle avoidance*, panjang jalur yang lebih panjang jika dibandingkan dengan menggunakan penghitungan jalur ulang dengan perbandingan 8: 7.
3. Dengan *real-time pathfinding* menggunakan A* dan Reynolds *steering obstacle avoidance*, kinerja CPU 2 kali lebih optimal dibandingkan dengan menggunakan penghitungan jalur ulang.
4. Banyaknya halangan pada peta permainan berpengaruh terhadap menurunnya kinerja CPU saat menggunakan A* dan Reynold *steering obstacle avoidance* untuk *real-time pathfinding.*

6.2 Saran

Untuk meningkatkan hasil yang telah dicapai dari penelitian ini dapat dilakukan beberapa perbaikan sebagai berikut :

1. Diharapkan pada penelitian selanjutnya, penggunaan *pathfinding* dapat dikembangkan menjadi *tactical pathfinding* agar dapat mendeteksi faktor ancaman.

2. Modifikasi algoritma Reynold *steering obstacle avoidance* untuk menciptakan suatu algoritma baru yang lebih efisien dari segi panjang jalur yang dihasilkan.
3. Penambahan Algoritma *short steering* Reynolds *Steering Collision Avoidance* [MIL-09] agar dapat menghindari tabrakan antar aktor.

DAFTAR PUSTAKA

[ALD-09] Aldrich, Clark. 2009, *The Complete Guide to Simulations and Serious Games: How the Most Valuable Content Will be Created in the Age Beyond Gutenberg to Google*, John Wiley and Sons, Hoboken, New Jersey.

[FOU-09] Foudil, C., Noureddine, D., Sanza, C., dan Duthen, Y. 2009, "Path Finding and Colision Avoidance in Crowd Simulation", Journal of Computing and Information Technology.

[GRA-03] Graham, R., McCabe, H., Sheridan, S. 2003, "Pathfinding in Computer Games", Institute of Technology Blanchardstown.

[GRA-05] Graham, R., McCabe, H., Sheridan, S. 2005, "Neural Pathways for Real-Time Dynamic Computer Games", The Eurographics Association.

[GRA-06] Graham, Ross & McCabe, Hugh. 2006, "Real-time Agent Navigation with Neural Networks for Computer Games", MSc. Thesis.

[HAR-68] Hart, P., Nilsson, N., & Raphael, B. 1968, "A Formal Basis for Heuristic Determination of minimum cost paths", IEEE Trans on Systems Science and Cybern, 4:100-107.

[KOR-90] Korf, R. 1990, "*Real-time Heuristic Search*", Artificial Intelligence 42(2-3):189-211.

[LES-05] Lester, Patrick. 2005, "A* Pathfinding for Beginners". http://www.policyalmanac.org/games/aStarTutorial.htm. [15 Desember 2014].

[MAT-02] Matthews, James. 2002, *Basic A* Pathfinding Made Simple*, AI Game Programming Wisdom, Charles River Media.

[MIL-09] Millington, Ian And John Funge. 2009, *Artificial Intelligence For Games*, Second Edition, Morgan Kaufmann Publishers Inc., San Francisco, California.

[THU-05] Thurau, Christian., Bauckhage, Christian., dan Sageger, gerhard,. 2005, "*Learning Human Like Movement Behavior for Computer Games"*, ACM International Conference Proceeding Series Vol 265.

[TOM-04] Tomlinson, S. L. 2004, "The Long and Short of Steering in Computer Games*"*, International Journal of Simulation Vol 1-2 No5.

[TOZ-04] Tozour, Paul. 2004, *Search Space Representation*, *AI Game Programming Wisdom 2*, Charles River Media.