ISSN 2460-5662

Pergerakan Pasukan Untuk Mengejar Musuh Bergerak Menggunakan *D* Lite* Berbasis Algoritma *Pathfinding*

Mochamad Kholil

Program Studi Sistem Informasi, Fakultas Ilmu Komputer
Universitas Nadhlatul Ulama Sidoarjo
e-mail: kholil.si@unusida.ac.id

Abstrak

Pergerakan agen pada permainan *Real Time Strategy* dipengaruhi oleh beberapa faktor salah satunya adalah teknik pergerakan agen didalam lingkungan permainan. *Pathfinding* dalam *video game* merupakan algoritma kecerdasan buatan bagaimana cara sebuah agen bergerak menemukan jalan optimal dengan usaha minimal sampai pada tujuan. Hal ini bisa dicapai dengan mengimplementasikan suatu algoritma pathfinding pada game. Penelitian ini mengenai algoritma *D* Lite* yang mampu merencanakan pencarian jalur di lingkungan game dengan environment yang berubah sekaligus objek yang sebagai target bergerak dan menjadikan proses pengejaran target menjadi efisien bagi agen serta memberikan dasar yang kuat untuk penelitian lebih lanjut tentang metode pencarian ulang dalam kecerdasan buatan

Kata kunci: *Agent Movement, Real Time Strategy, Pathfinding, D* Lite*

Abstract

The movement of agents in the Real Time Strategy game influenced by several factors, one of which is the technique of agent movement within the game environment. Pathfinding in a video game is an artificial intelligence algorithm how to find an agent moves the optimal way in which there are obstacles in the environment. This can be achieved by implementing a pathfinding algorithm to the game. This study of the D Lite algorithm is able to plan a search path in an game environment, change the environment and moving target to be efficient, optimal and complete for agents and will describe some way of planning applications and provides a solid foundation for further research on methods of search re in artificial intelligence.*

Keywords: Agent Movement, Real Time Strategy, Pathfinding, D Lite*

Pendahuluan

Permainan komputer semakin kompleks dengan menyajikan grafis yang realistis dan *Non-Playable Character* (NPC) yang canggih. Selain pengembangan aspek grafis dalam permainan, para pengembang permainan menitikberatkan dalam aspek pengembangan kecerdasan buatan untuk membuat NPC terlihat canggih. Pergerakan agen merupakan salah satu elemen penting kecerdasan buatan dalam permainan komputer. Strategi dalam pathfinding merupakan bagian terpenting dalam pergerakan agen.

Pathfinding saat ini telah menjadi elemen penting dalam sebuah permainan, dimana semua permainan pasti memiliki metode pathfinding yang berbeda. Sesuai dengan tingkat kerumitan dari games yang diusung. Semakin rumit map dari suatu games maka semakin rumit pula metode pathfinding yang digunakan. Pada permainan *Real Time Strategy Clash of Clans* metode pathfinding merupakan metode yang otomatis dijalankan setiap kali NPC melakukan serangan. Pemain sangat dimudahkan, karena tidak perlu mengatur pergerakan karakter. Pemain hanya perlu menentukan titik awal ketika melakukan serangan.

Penelitian ini akan membahas bagaimana memecahkan suatu masalah dengan teknik searching atau pencarian dimana NPC berperan sebagai sebagai pasukan yang akan mengejar musuh bergerak dalam lingkungan sebuah game. Dalam hal ini, metode Pathfinding yang digunakan adalah Algoritma *D* Lite* yang diharapkan mampu merencanakan ulang pencarian jalur dari keadaan awal (*initial state*) menuju keadaan tujuan (*goal state*).

Dasar Teori

Video Game

Video games adalah salah satu jenis hiburan yang palingmodern, karena sangat banyak sehingga video games dapat ditemukan hampir di semua platform digital yang tersedia, mulai dari ponsel sampai game konsol. Game PC saja dapat menyumbang pendapatan lebih dari 13 miliar dollar untuk tahun 2009. Mobile dan jaringan *social gaming* juga mengalami peninggkatan, lebih dari 80 juta orang bermain Zynga's Farmville di Facebook pada tahun 2010.

Video games telah berubah secara dramatis selama dua dekade terakhir. Sebagai teknologi komputasi yang telah berkembang, pengembang *video game*

telah mengambil keuntungan dari pemrosesan yang tersedia dan membuat representasi yang lebih realistis dari dunia virtual game. Misalnya, *Need for Speed*, sebuah judul game balap arcade yang dirilis pada tahun 1994 menampilkan akselerasi *hardware* untuk visual, mesin fisika yang sangat sederhana dan tidak ada kerusakan kendaraan akibat tabrakan.

Sejak dirilis *Need for Speed* pada tahun 1994, peningkatan teknologi komputasi serta teknologi rendering telah memungkinkan untuk pengalaman bermain *game* yang lebih realistis. *Grand Turismo* 5 adalah *game* simulasi balap yang dirilis pada tahun 2010 untuk Playstation 3 dan menawarkan gambar visual realistis serta perhitungan fisika yang sangat akurat dan kerusakan model dalam *game* kendaraan.

Teknologi *video games* berkembang seiring dengan kemajuan *Central Processing Unit* (CPU) dan teknologi render grafis. Sebagai sebuah industri, pengembangan game adalah salah satu yang sangat inovatif dan kopetitif dengan pengembang game sering bekerja di ujung tombak dari grafis 3D dan tingkat *Artificial Intelligent* (AI). Setiap permainan baru yang dirilis, game *players (gamers)* berharap tingkat realistis dari visual, suara dan *Artificial Intelligent* (AI) *pada Non-Playable Characters* (NPC) lebih tinggi.

Pathfinding

Terdapat banyak metode pencarian yang telah diusulkan. Semua metode yang ada dapat dibedakan ke dalam dua jenis: pecarian buta/tanpa informasi (*blind* atau *un-informed search*). dan pencarian heuristik/dengan informasi (*heuristic* atau *informed search*). Setiap metode mempunyai karakteristik yang berbeda-beda dengan kelebihan dan kekurangan masing-masing.

Untuk mengukur performansi metode pencarian, terdapat empat kriteria yang dapat digunakan, yaitu (Russel & Norvig, 1995) :

1. *Completeness*: Apakah metode tersebut menjamin penemuan solusi jika solusinya memang ada?
2. *Time complexity*: Berapa lama waktu yang diperlukan?
3. *Space complexity*: Berapa banyak memori yang diperlukan?
4. *Optimality*: Apakah metode tersebut menjamin menemukan solusi yang terbaik jika terdapat beberapa solusi berbeda?

Algoritma *A**

Algoritma ini merupakan *Best-First Search* yang menggabungkan *Uniform Cost Search*. Biaya yang diperhitungkan didapat dari biaya sebenarnya ditambah dengan biaya perkiraan. Dalam notasi matematika dituliskan sebagai: f(n) = g(n) + h(n). Dengan perhitungan biaya seperti ini, algoritma *A** adalah complete dan optimal. Pembuktian secara matematis dapat dilihat di (Russel & Norvig, 1995).

Sama dengan algoritma dasar *Best-First Search*, algoritma *A** ini juga menggunakan dua senarai: *OPEN* dan *CLOSED*. Terdapat tiga kondisi bagi setiap suksesor yang dibangkitkan, yaitu: sudah berada di *OPEN*, sudah berada di *CLOSED*, dan tidak berada di *OPEN* maupun *CLOSED*. Pada ketiga kondisi tersebut diberikan penanganan yang berbeda-beda.

Jika susksesor sudah pernah berada di *OPEN*, maka dilakukan pengecekan apakah perlu pengubahan parent atau tidak tergantung pada nilai *g*-nya melalui parent lama atau parenti baru. Jika melalui parent baru memberikan nilai *g* yang lebih kecil, maka dilakukan pengubahan parent. Jika pengubahan parent dilakukan, maka dilakukan pula perbaruan (*update)* nilai *g* dan *f* pada suksesor tersebut. Dengan perbaruan ini, suksesor tersebut memiliki kesempatan yang lebih besar untuk terpilih sebagai simpul terbaik (*best node*).

Jika suksesor sudah pernah berada di *CLOSED*, maka dilakukan pengecekan apakah perlu pengubahan parent atau tidak. Jika ya, maka dilakukan perbaruan nilai *g* dan *f* pada suksesor tersebut serta pada semua “anak cucunya” yang sudah pernah berada di *OPEN*. Dengan perbaruan ini, maka semua anak cucunya tersebut memiliki kesempatan lebih besar untuk terpilih sebagai simpul terbaik (*best node*).

Jika suksesor tidak berada di *OPEN* maupun *CLOSED*, maka suksesor tersebut dimasukkan ke dalam *OPEN*. Tambahkan suksesor tersebut sebagai suksesornya best node. Hitung biaya suksesor tersebut dengan rumus berikut:

$$f(n) = g(n) + h(n) \quad (1)$$

dimana:

- h(n) = biaya estimasi dari titik n ke tujuan.
- g(n) = biaya path/perjalanan.

– f(n) = solusi biaya estimasi termurah titik n untuk mencapai tujuan.

*Lifelog Planning A** (*LPA**)

*LPA** adalah versi tambahan dari algoritma *A** yang menggunakan kembali dari pencarian sebelumnya di seluruh iterasi pencarian. Data nilai setiap node yang kira-kira sama dengan algoritma *A**, yaitu nilai CSF G dan nilai heuristik *H* dengan tambahan nilai *RHS* baru. *RH*S nomenklatur berasal dari algoritma *Dynamic SWSF-F*P dimana menurut aturan tata bahasa nilai *RHS* adalah nilai sisi kanan.

Nilai *RHS* merupakan langkah minimum *CSF* berikutnya dari node, kecuali untuk kasus khusus ketika node adalah node awal, dalam hal ini nilai *RHS* untuk node diatur menjadi 0. Perhitungan dari nilai simpul *N RHS* ditunjukkan pada rumus berikut:

$$RHS(N) = \begin{cases} 0 \\ min_{s\ \epsilon succ(N)}\big(CSF(S) + traversalCost(S, N)\big) \end{cases} \quad (2)$$

Dimana simpul *S* adalah penerus dari simpul *N* dan *succ(N)* adalah himpunan semua penerus simpul *N*. *CSF(N)* adalah nilai *cost-so-far* dari simpul N sementara *traversalCost* (S, N) adalah biaya yang bergerak dari node *S* ke node *N*. Literatur untuk algoritma *LPA** memanfaatkan node pendahulu untuk perhitungan nilai *RHS* tetapi dalam *navgraph* standar (diarahkan) sebuah node suksesor adalah pendahulunya dan lebih mudah menyebutkan pendahulu node dihilangkan.

Konsep penting berikutnya yang diperlukan untuk memahami pengoperasian algoritma *LPA** adalah dari konsistensi lokal untuk node. Sebuah node konsisten lokal jika nilai *G* adalah sama dengan nilai *RHS*. Ini berarti jika sebuah simpul konsisten lokal maka nilai G merupakan biaya jalan terpendek untuk node dari setiap node tetangganya.

Jika semua node dalam grafik konsisten lokal, nilai node *RHS* sama dengan jarak yang tepat dari simpul awal dan jalur terpendek ke tujuan dapat ditelusuri dengan memulai pada node tujuan dan bergerak ke node tetangga dengan nilai *RHS* terkecil sampai awal simpul tercapai. Ini berarti bahwa algoritma *LPA**

membangun jalan mundur dari node tujuan dengan memilih penerus *node* yang memiliki nilai *RHS* terkecil sebagai langkah berikutnya dalam sebuah jalan.

Ada dua jenis inkonsistensi lokal yang dapat terjadi. *Node* secara lokal lebih konsisten ketika nilai *G* lebih tinggi dari nilai *RHS* dan node secara lokal di bawah konsisten ketika nilai *G* lebih rendah dari nilai *RHS*-nya. Inkonsistensi ini terjadi sebagai akibat dari perubahan lingkungan dan sehingga perlu dikoreksi pada setiap iterasi dari algoritma. Seorang pembaca tertarik mungkin telah menyadari bahwa nilai *RHS* adalah minimum dari semua nilai *G* baru yang dihitung untuk setiap node pengganti selama *A** simpul eksplorasi.

Proses pencarian algoritma *LPA** memiliki cara yang sama dengan algoritma pencarian *A**. *Open List* yang berpotensi untuk mengeksplorasi terdapat dalam *Open List*. *Open List* diurutkan berdasarkan nilai node F dan hubungan yang rusak mendukung nilai *G* terendah. Ketika sebuah node dieksplorasi, nilai *RHS*-nya dibandingkan dengan nilai G. Dalam kasus overkonsistensi yang terjadi, bahwa jalan yang lebih pendek untuk node ditemukan.

Dalam hal ini nilai *G* diatur ke nilai RHS (jalur terpendek baru) dan semua node penerus diperbarui untuk mengambil perubahan ke account. Jika *node* konsisten atau bawah lokal konsisten maka nilai *G* yang diatur hingga tak terbatas (sehingga secara lokal lebih konsisten) dan simpul serta penerus diperbarui. Pembaruan simpul hanyalah sebuah perhitungan node *RHS, F* dan *G* nilai. Node diperbarui kemudian dimasukkan ke dalam daftar terbuka (atau posisinya dalam daftar terbuka diperbarui jika *node* sudah ada pada daftar terbuka).

Untuk penjelasan algoritma *LPA** di atas, pencarian awal yang dilakukan adalah persis sama seperti algoritma pencarian A*. Node yang tepat dieksplorasi dalam urutan yang sama persis seperti yang akan terjadi pada *A** (Sturtevant, Felner, Barrer, & Burch, 2009). Pencarian berikutnya hanya bertindak untuk memperbaiki yang terdeteksi lokal inkonsistensi dalam grafik yang mungkin dihasilkan dari perubahan lingkungan. Ketika perubahan lingkungan terdeteksi, simpul diperbarui (ditambahkan ke daftar terbuka) dan pencarian dilanjutkan. *Open List LPA** berfungsi untuk memprioritaskan perbaikan *node local* tidak konsisten

yang merupakan bagian dari solusi (seperti $A*$ mengeksplorasi node yang lebih mungkin untuk menjadi bagian dari solusi pertama).

Kondisi penghentian setiap pencarian $LPA*$ adalah sebagai berikut: pencarian berakhir setelah tujuan simpul konsisten lokal dan nilai F tujuan adalah lebih kecil dari atau sama dengan nilai F terendah di *Open List* (yaitu tidak ada simpul dengan potensi rute yang lebih baik ke node tujuan ada).

Dynamic A (D* Lite)*

Algoritma $D*$ *lite* disajikan sebagai sarana untuk memecahkan masalah *pathfinding real time* (agen bergerak) baik dalam lingkungan sebagian tidak diketahui dan dinamis (Koenig & Likhachev, 2002). Untuk menjaga informasi pencarian validasi bahkan di pencarian dengan simpul awal bervariasi, algoritma *D* Lite* hanya membalikkan pencarian arah $LPA*$ (pencarian sekarang dilakukan dari node tujuan ke node awal) sehingga nilai G sekarang mewakili biaya dari tujuan (*CSG*) nilai bukan nilai *CSF* per-*node.* Jalan yang dibangun ke depan dari node awal ke node tujuan dengan memilih node dengan *RHS* terendah sebagai langkah berikutnya.

Cukup membalikkan arah pencarian algoritma ini tidak cukup untuk menjamin optimalisasi dari Algoritma (Koenig & Likhachev, 2002), karena nilai-nilai heuristik untuk setiap simpul dalam grafik juga akan akurat sebagai agen yang mulai perubahan simpul selama traversal agen. Jika nilai-nilai heuristik berubah maka pemesanan *open list algoritma* $D*$ *lite* akan disegarkan dan *open list* akan perlu mengatur kembali dengan mempertimbangkan perubahan nilai heuristik. $D*$ *Lite* menggunakan metode algoritma D* untuk mencegah penataan open list kembali, dengan menggunakan prioritas simpul batas bawah dengan yang digunakan di $LPA*$ (Anthony Stentz, 1995). Untuk menghapus kebutuhan penataan kembali *open list*, fungsi heuristik yang digunakan harus memenuhi rumus berikut:

$$H(S,S') \leq traversalCost(S,S') and\ H(S,S) \leq H(S,S')+H(S',S) \quad (3)$$

Dimana *H (S, S')* adalah nilai heuristik antara titik *S* dan *S'* dan *traversalCost(S,S')* adalah biaya perpindahan dari node *S* to *S'*. Ini berarti bahwa fungsi heuristik harus memperkirakan dengan tepat dari biaya antara dua titik.

Ketika agent berpindah dari titik *S* ke titik *S'*, nilai heuristik akan menurun di sebagian besar *H (S,S')*. Ini berarti nilai *F* di open list akan juga menurun di sebagian besar *H(S,S')*.

Untuk memperbaiki nilai F pada titik di *open list,* setiap titik *F* dan nilai *H* harus memiliki *H(S, S'*) dikurangkan dari nilai *F*. Sejak *H(S,S')* konstan, ini berarti bahwa pemesanan antrian prioritas akan tetap tidak berubah dan open list akan memerlukan penataan kembali pada gerakan agen. Perbaikan perlu dilakukan pada setiap titik inkonsisten yang mungkin dihasilkan dari perubahan lingkungan.

Perbaikan ini mungkin perlu memasukkan titik baru ke dalam open list. Meskipun pemesanan *open list* sudah benar, semua nilai titik *H* (dan sebagai hasil nilai *F*) sekarang *H(S,S')* terlalu tinggi. Merperbarui setiap titik *F* dan *H* dari semua titik di dalam open list adalah sebuah biaya yang mahal dan dapat dihindari. Ketika sebuah perubahan lingkungan terjadi dan agent telah berubah lokasi (dari *S* ke *S'*), nilai *H(S,S')* ditambahkan ke sebuah variabel *Km* yang melacak perubahan pada nilai heuristik selama pergerakan agent. Sebuah agent mungkin telah berpindah beberapa langkah antara perubahan lingkungan. *S* dan *S'* tidak selau terhubung langsung.

Untuk mencegah mempebarui setiap titik pada open list, untuk mencerminkan penurunan nilai heuristik, variabel *Km* hanya ditambahkan ke nilai titik *F* baru ketika mereka memasukkan ke dalam *open list*. Menambahkan *Km* memiliki efek menjaga konsitensi nilai heuristik di agen bergerak. Penambahan nilai Km dilakukan secara otomatis oleh fungsi update titik (mengacu pada fungsi "*calculate key*") untuk titik baru yang dimasukkan dalam open list.

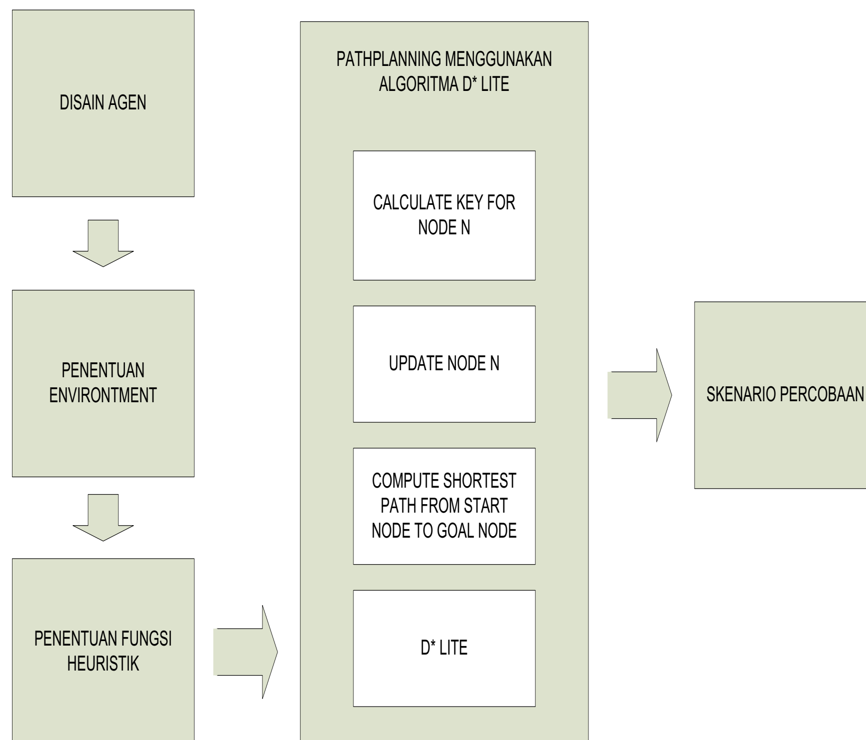
Selama tahap pencarian, titik dengan nilai *F* paling rendah dipilih dari open list dan diperluas untuk menjaga titik nilai kebenaran titik *F*. Pemeriksaan tambahan dilakukan sebelum titik yang dipilih dari open list. Titik dengan nilai *F* paling rendah dipilih dan nilai *F* baru dihitung (memperhitungkan nilai *Km* mungkin dirubah). Jika nilai F lama lebih rendah dari pada nilai *F* baru, maka nilai *F* lama diperbarui ke nilai *F* baru dan posisi simpul dimasukkan kembali dalam *open list*. Eksplorasi simpul terjadi dengan cara yang sama seperti di algoritma *LPA**.

Tingkat eksplorasi simpul *D** sangat mirip dengan *A** (Koenig & Likhachev, 2002) dan biaya memori per-simpul juga sama (nilai data per simpul

yang sama persis, hanya berbeda dalam link induk dari *A** diganti dengan nilai *RHS* di algoritma *D* Lite*). Kesamaan tersebut mengakibatkan biaya memori *D** Lite secara keseluruhan sangat mirip dengan *A**.

Metode Penelitian

Hasil dari penelitian ini dapat diterapkan untuk menentukan pergerakan Skeleton Traps maupun pasukan yang ada pada *Clan Castle*. Jika *Skeleton Traps* dan pasukan pada *Clan Castle* dapat bergerak bersama dari arah yang berbeda sekaligus mampu melakukan pencarian ulang jalur tercepat maka kemungkinan mereka mencapai tujuan semakin cepat dan menjadikan pertahanan semakin kuat.



Gambar 1. Skema Metodologi Penelitian

Gambar 1. merupakan diagram alur langkah-langkah pelaksanaan penelitian ini. Diawali dengan penentuan desain perilaku agen sampai dengan visualisasi pergerakan agen di dalam ruang 3 dimensi.

Desain Agen

Pada algoritma *D* Lite*, simulasi dilakukan pada agen pasukan yang mengejar musuh bergerak. Posisi *food source* atau sumber makanan ditentukan secara acak. Sumber makanan dianggap sebagai posisi baru agen ketika bergerak mendekati target atau lawan. Modifikasi algoritma *D* Lite* ditujukan untuk melakukan simulasi pergerakan serangan terhadap lawan.

Gambar 2. Desain Agen

Setiap agen memiliki *property speed, power, sensor dan efektor*. Tabel 1 adalah parameter serangan yang dimiliki oleh agen dan target. Setiap agen dianggap memiliki kecepatan 6 dan memiliki kekutatan rendah. Agen hanya dapat menyerang sekali kemudian mati. Sedangkan lawan dianggap sebagai target bergerak yang memiliki kecepatan tinggi kekuatan 5 kali lebih kuat dari agen. Sehingga butuh 5 kali serangan agen untuk menghancurkan.

Tabel 1. Parameter Serangan

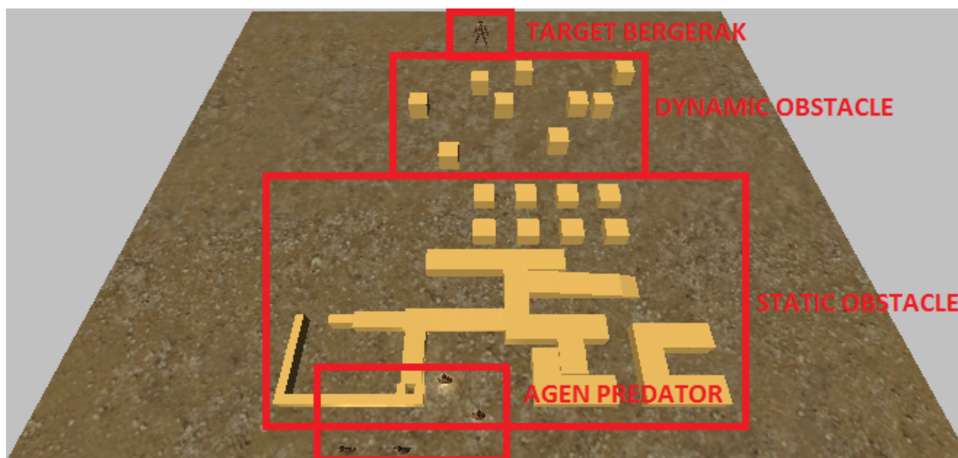
Parameter	Agen	Lawan	Keterangan
Speed	6	Tinggi	
Power	1	5	
Sensor	Barrier Beacon		Untuk mendeteksi keberadaan obstacle Untuk mendeteksi keberadaan
Effector	Direction Speed		Menentukan arah Menentukan kecepatan

ISSN 2460-5662

Behaviour	Track target Avoid barrier Avoid beacon		Mencari musuh Menghindari obstacle Menghindari agen lain

Setiap agen memiliki sensor barrier untuk mendeteksi keberadaan *obstacle* dan *sensor beacon* untuk mendeteksi keberadaan target didekatnya. Setiap agen juga memiliki 3 perilaku, yaitu *track target* (mencari musuh), *avoid barrier* (menghindari dinding) dan *avoid beacon* (menghindari agen lain).

Environment



Gambar 3. Ruang Gerak Agen

Gambar 3. adalah *environment* agen saat begerak menuju target atau musuh. Agen akan bergerak di ruang 3 dimensi di mana didalamnya terdapat *static obstacle* dan *dynamic obstacle* yang harus dihindari oleh agen. Gerakan menghindar dilakukan dengan mengitari *obstacle* melewati sebalah kiri dan sebelah kanan *obstacle*. Agen akan mengikuti dan mengejar dimana posisi target berada. Jalur optimal menuju target oleh masing-masing agen bisa berbeda tergantung jalur optimal setiap agen terhadap target.

Penentuan Fungsi Heuristik

Dalam masalah pencarian rute terpendek dalam sebuah grup agen, fungsi heuristik menggunakan metode pengukuran:

1. *Manhattan Distance*

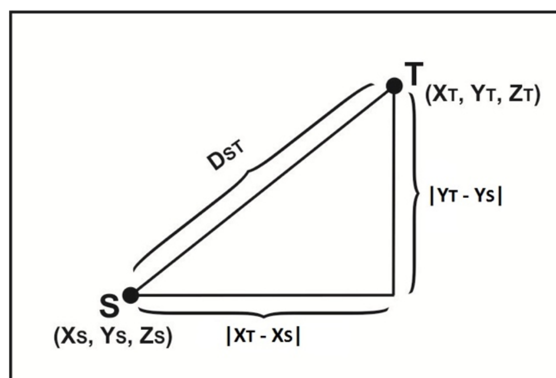
   Jarak antara dua vektor *p, q* dalam sebuah ruang nyata vector *n*-dimensional dengan sistem koordinat kartesian tetap, merupakan jumlah dari panjang proyeksi segmen garis antara titik ke sumbu koordinat. Untuk memenuhi fungsi heuristik *Manhattan Distance*, maka agen harus menghitung jarak posisi baru *(xs , ys, zs)* ke posisi target *(xt, yt, zt)*.

$$d_1\,(p,q) = \ \|p-q\|_1 = \sum_{i=1}^{n} |p_1 - q_1|, \qquad (4)$$

   dimana *p* dan *q* adalah vektor

   Dst adalah jarak antara posisi baru dengan posisi target, maka untuk mencari besarnya jarak antara posisi baru dengan posisi target diperoleh rumus sebagai berikut:

$$D_{st} = |x_t - x_s| + |y_t - y_s| + |z_t - z_s| \qquad (5)$$



Gambar 4. *Manhattan Distance*

2. *Euclidean Distance*

   Euclidean Distance antara titik *p* dan *q* merupakan panjang dari segmen garis yang menghubungkan antara *((pq)̅)*. Dalam koordinat kartesian, jika *p* =

ISSN 2460-5662

(p1, p2, ..., pn) dan q= *(q1, q2, ..., qn)* adalah 2 titik dalam n-*space Euclidean*, kemudian jarak (*d*) dari p ke q, atau dari q ke p disajikan dalam sebuah rumus *phytagoras*. Untuk memenuhi fungsi heuristik *Euclidean Distance*, maka agen harus menghitung jarak posisi baru *(xs , ys, zs)* ke posisi target *(xt, yt, zt)*.

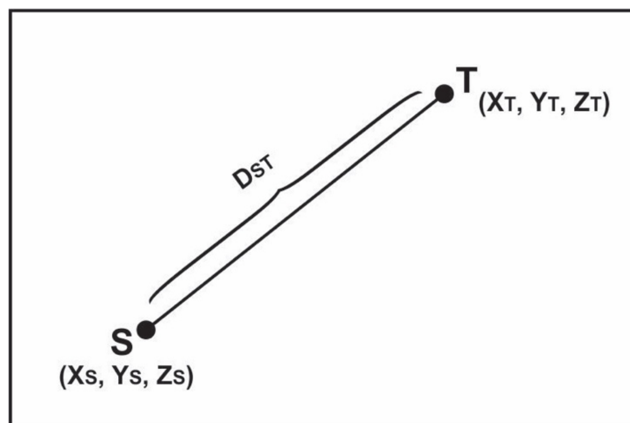
$$d(p,q) = d(p,q) = \sqrt{(q_1 - p_1)^2 + (q_2 - q_1)^2 + \cdots + (q_n - p_n)^2}$$

$$= \sqrt{\sum_{i=1}^{n} (q_i - p_i)^2} \quad (6)$$

Posisi titik dari *n-space Euclidean* adalah sebuah *Euclidean* vektor. Jadi, p dan q adalah *Euclidean* Vektor, mulai dari ruang asal dan mengindikasikan dua titik.

Dst adalah jarak antara posisi baru dengan posisi target, maka untuk mencari besarnya jarak antara posisi baru dengan posisi target diperoleh rumus sebagai berikut:

$$D_{st} = \sqrt{(x_t - x_s)^2 + (y_t - y_s)^2 + (z_t - z_s)^2} \quad (7)$$



Gambar 5. *Euclidean Distance*

Path Planning Menggunakan Algoritma *D* Lite*

Langkah 1 Menghitung kunci pada titik N

<u>Calculate Key for Node N</u>

1. G = nilai minimum G dan RHS
2. F = G + H + K_m
3. Retrun(F, G)

Gambar 6. Pseudocode Algoritma Proses Perhitungan Kunci pada Titik *N*

Langkah 2 Update pada Titik *N*.

<u>Update Node N</u>

1. If N bukan S_{goal} then hitung nilai RHS
2. If N berada di Open List, hapus N dari open list
3. If nilai G pada N != nilai RHS pada N (inkonsisten local)
   - 3.1 Calculate key for node N
   - 3.2 Masukkan N ke dalam open list

Gambar 7. Pseudocode Algoritma Proses Update pada Titik *N*

Langkah 3 Perhitungan Jalur Terpendek Dari Titik Awal ke Titik Akhir.

<u>Compute Shortest Path from Start Node to Goal Node</u>

1. While (S_{start} = inconsistent and S_{start}'s key > best open nodes's key)
   - 1.1 Set K_{old} to the best open node's key
   - 1.2 Remove the best node N dari open list
   - 1.3 If K_{old} < calculateKey(N)
     - 1.3.1 Masukkan N ke dalam open list
   - 1.4 Else if N is locally over-consistent (Nilai G pada N > Nilai RHS pada N)
     - 1.4.1 Set nilai G pada N to nilai RHS pada N
     - 1.4.2 Update titik suksesor N

1.1 Else
1.5.4 Set nilai G pada N to infinity
1.5.5 Update N
1.5.6 Update titik suksesor N

Gambar 8. Pseudocode Algoritma Proses *Compute Shortest Path*

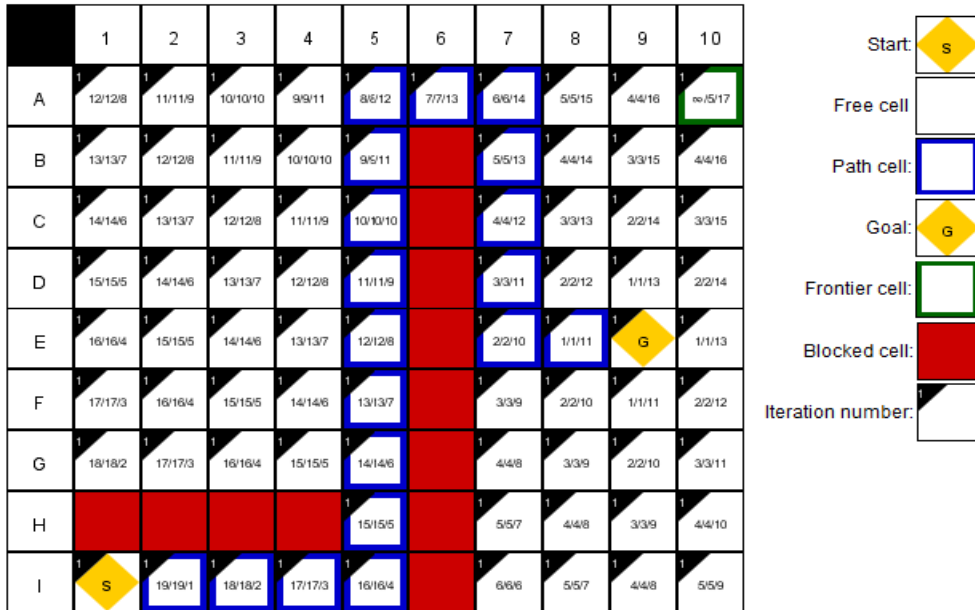
Langkah 4 D* Lite.

D* Lite

1. Set the last position S_{last} to the start node S_{start}
2. Set all RHS node dan Nilai G to infinity
3. Set RHS dari S_{goal} to 0
4. Calculate S_{goal}'s key and insert start node ke dalam open list
5. Set K_m to 0
6. Compute Shortest Path
7. While (S_{start} != S_{goal})
   7.1 S_{start} = minimum(G value + traversal cost) of all of S_{start}'s successor
   7.2 Move to S_{start}
   7.3 Check semua perubahan lingkungan

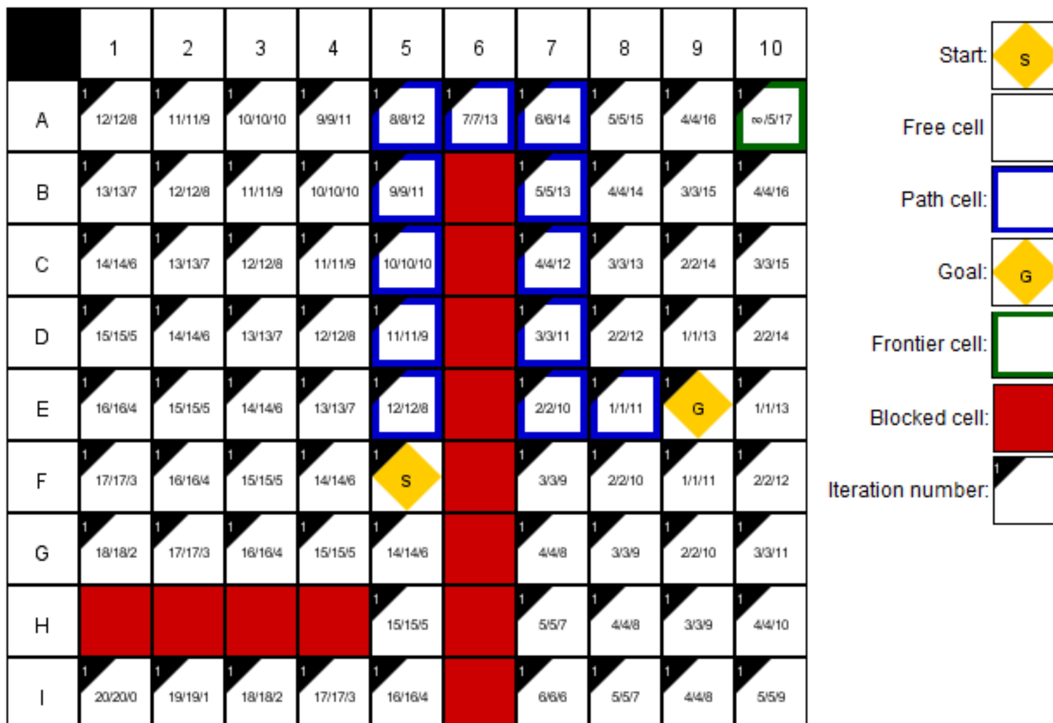
Gambar 9. Pseudocode Algoritma Proses *Compute Shortest Path*

Pada langkah ini agen pasukan akan memperhatikan setiap perubahan yang terjadi pada lingkungan game untuk mendapatkan jalur optimal dari posisi awal menuju posisi target (mengejar musuh). Gambar 10 merupakan keadaan awal dimana agen pasukan menemukan posisi musuh sebagai buruan.



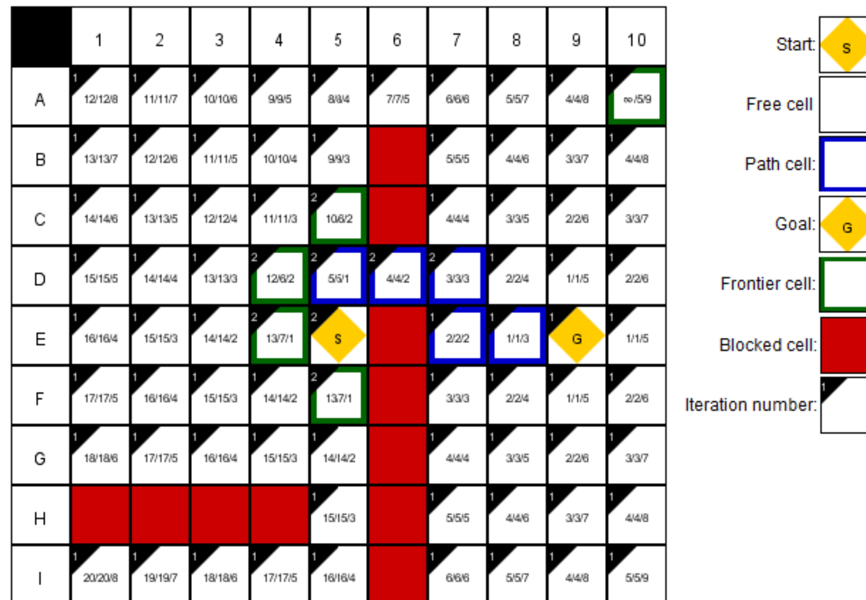
Gambar 10. Keadaaan Awal Pasukan Menemukan Musuh

Setelah pasukan berhasil menemukan posisi musuh, langkah selanjutnya pasukan akan bergerak mendekati target. Pasukan akan menghindari halangan yang ada pada peta dan akan melewati jalur optimal yang berhasil didapatkan. Gambar 11 merupakan pergerakan pasukan mendekati musuh.



Gambar 11. Pergerakan Pasukan Mendekati Musuh

Ketika agen pasukan sudah sampai pada posisi pertengahan jalan dan terjadi perubahan pada environment khususnya pada obstacle, maka pasukan akan melakukan tindakan menghitung ulang biaya perjalanan dari posisi agen yang baru terhadap posisi target. Agen akan membandingkan antara jalur yang baru dengan jalur sebelumnya apakah selalu sama atau lebih baik dari yang ada. Gambar 12 merupakan pergerakan pasukan saat terjadi perubahan lingkungan.



Gambar 12. Pergerakan Pasukan Saat Terjadi Perubahan Lingkungan

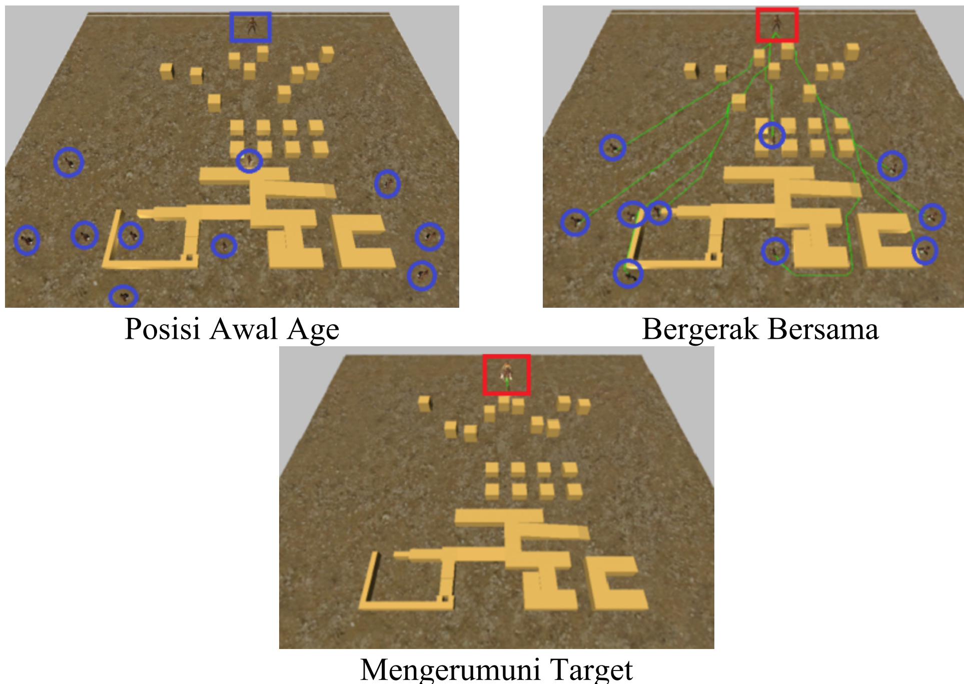
Skenario Percobaan

Tujuan dari penelitian ini adalah memperoleh sekelompok NPC cerdas yang mampu bergerak menuju target dengan formasi acak dari arah yang berbeda-beda tanpa menabrak agen lain dari kelompoknya, mampu menghindari *obstacle* dinamis dan mengejar target yang bergerak. Untuk memenuhi tujuan tersebut, penelitian ini melakukan percobaan dengan 6 skenario yaitu menciptakan pegerakan agen dari posisi asal menuju posisi target, pemindahan posisi koordinat target saat NPC bergerak, menghindari obstacle statis, menghindari obstacle dinamis, penambahan obstacle dinamis, dan pehitungan waktu ketika sampai pada target dengan variasi jumlah NPC lebih dari 10.

Percobaan dan Simulasi Komputer

Pergerakan kelompok agen yang telah diuraikan kemudian diimplementasikan dalam simulasi komputer dengan menggunakan Unity3D. Simulasi ini melibatkan 10 agen dengan posisi acak, satu target bergerak dengan koordinat awal (4, 0.05, 38), 10 obstacle dinamis dan beberapa obstacle statis. Input untuk setiap percobaan pada skenario 1-3 berupa posisi awal 10 agen secara acak dalam ruang 3 dimensi. Sedangkan outputnya iterasi, jarak dan waktu yang diperlukan agen dari posisi awal ke posisi target.

Untuk memudahkan analisa, percobaan dan simulasi ini dibagi menjadi 3 bagian dimana setiap bagian terdapat 3 sub bagian yaitu: pergerakan agen mencari jalur terpendek menuju target tanpa obstacle (fungsi heuristik menggunakan *Manhattan Distance* dan *Euclidean distance*), penambahan *obstacle* statis (fungsi heuristik menggunakan *Manhattan Distance* dan *Euclidean Distance*) dan terakhir penambahan *obstacle* dinamis (fungsi heuristic menggunakan *Manhattan Distance dan Euclidean Distance*) pada lingkungan *game*.



Posisi Awal Age — Bergerak Bersama

Mengerumuni Target

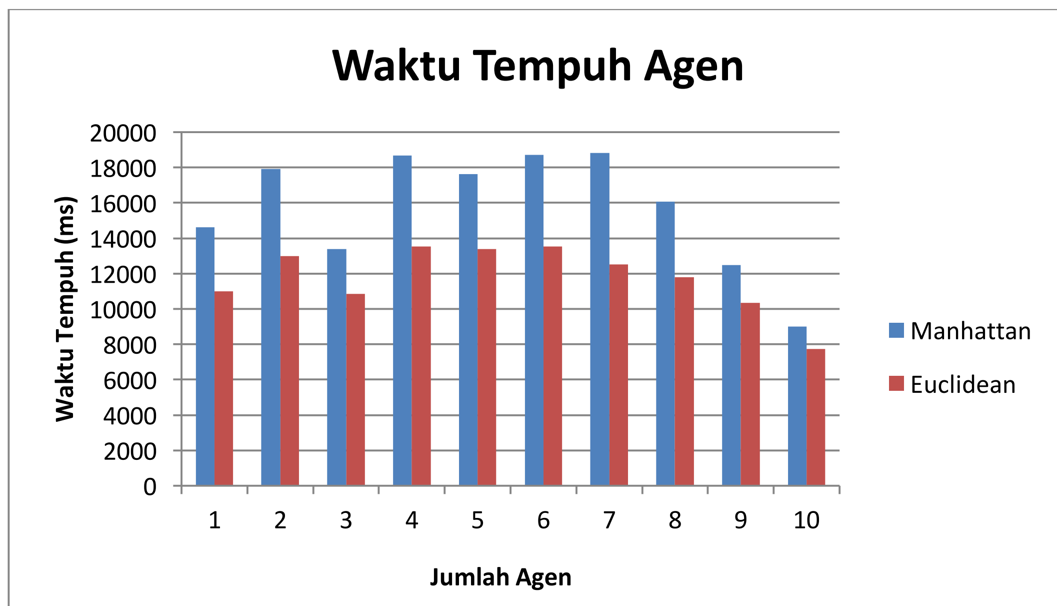
Gambar 13 Simulasi Pergerakan Agen

Perbandingan Tanpa *Obstacle*

Tabel 2. Perbandingan Waktu Pada Setiap Metode Pengukuran Tanpa *Obstacle.*

Agen	Manhattan	Euclidean
1	14615	11009
2	17932	12986
3	13372	10834
4	18667	13527
5	17634	13372
6	18711	13549
7	18807	12521
8	16070	11788
9	12474	10327
10	9009	7722
Rata-Rata	15729	11763

Dari data percobaan di atas diperoleh nilai rata-rata waktu tempuh pada seluruh agen dari masing-masing fungsi heuristik. Gambar 14 merupakan grafik perbandingan waktu tempuh pada semua agen.



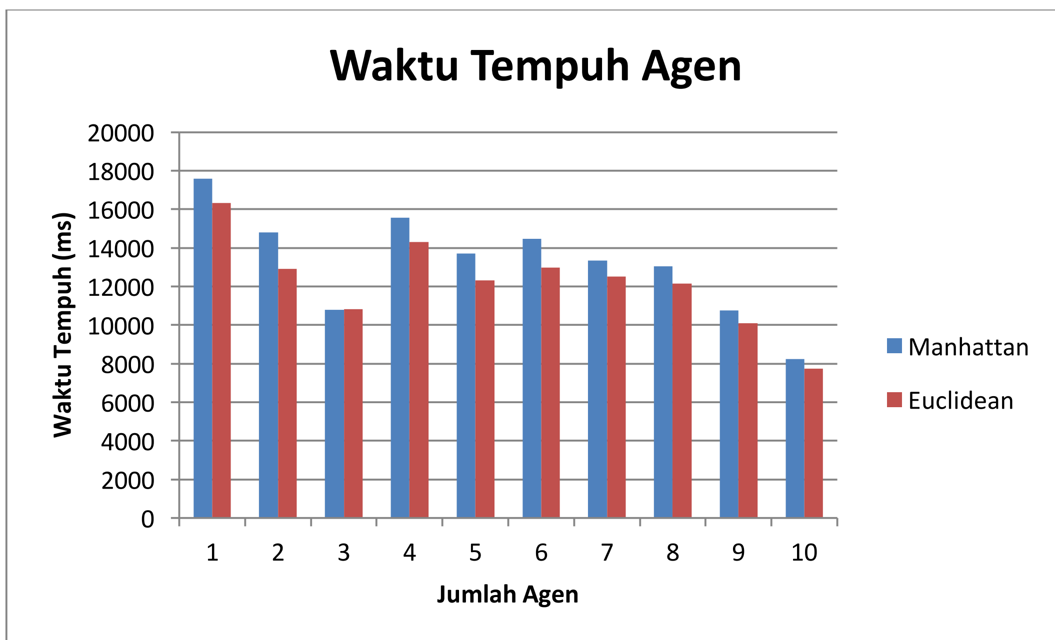
Gambar 14 Grafik Perbandingan Waktu Tempuh Agen Tanpa *Obstacle*

Perbandingan Dengan *Static Obstacle*

Tabel 3. Perbandingan Waktu Pada Setiap Metode Pengukuran Dengan *Static Obstacle*.

Agen	Manhattan	Euclidean
1	17592	16338
2	14794	12915
3	10807	10813
4	15577	14290
5	13700	12327
6	14483	12995
7	13338	12514
8	13037	12150
9	10758	10096
10	8238	7758
Rata-Rata	13232.4	12219.6

Dari data percobaan di atas diperoleh nilai rata-rata waktu tempuh pada seluruh agen dari masing-masing fungsi heuristik. Gambar 15 merupakan grafik perbandingan waktu tempuh pada semua agen.



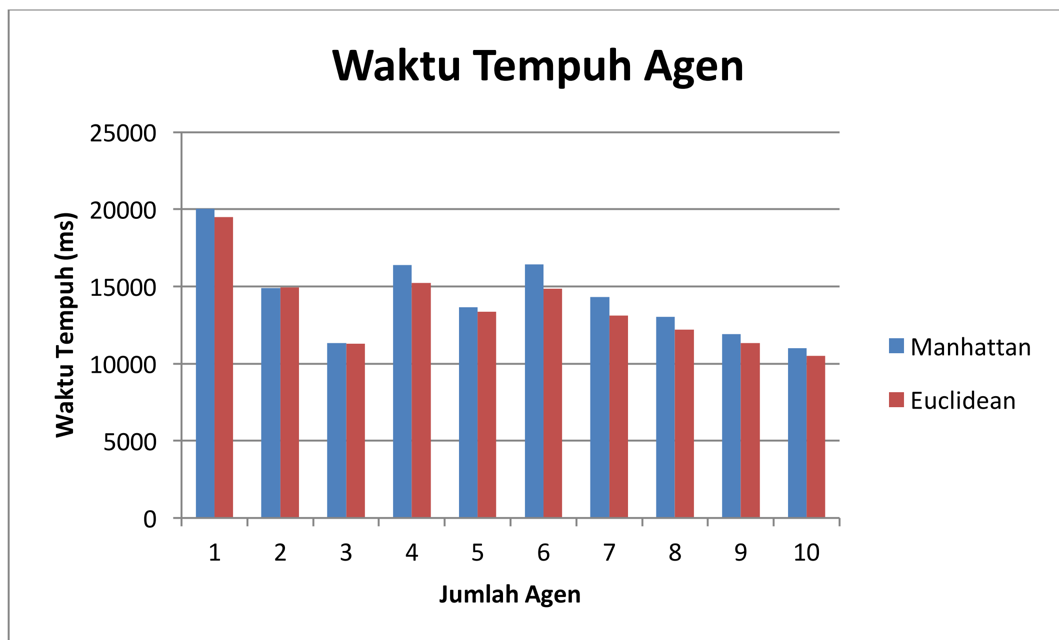
Gambar 15 Grafik Perbandingan Waktu Tempuh Agen Tanpa *Obstacle*

Perbandingan Dengan *Dynamic Obstacle*

Tabel 3. Perbandingan Waktu Pada Setiap Metode Pengukuran Dengan *Dynamic Obstacle.*

Agen	Manhattan	Euclidean
1	20021	19481
2	14893	14939
3	11324	11279
4	16373	15239
5	13644	13362
6	16429	14869
7	14315	13101
8	13038	12188
9	11901	11336
10	11025	10499
Rata-Rata	14296.3	13629.3

Dari data percobaan di atas diperoleh nilai rata-rata waktu tempuh pada seluruh agen dari masing-masing fungsi heuristik. Gambar 16 merupakan grafik perbandingan waktu tempuh pada semua agen.



Gambar 16 Grafik Perbandingan Waktu Tempuh Agen Tanpa *Obstacle*

Kesimpulan

Kajian telah dilakukan pada penelitian ini agar diperoleh proses pathfinding secara realtime terhadap musuh dinamis atau bergerak. Metode pengukuran *Manhattan Distance* dan *Euclidean Distance* digunakan untuk menunjukkan tingkat keandalan algoritma *D* Lite* pada skenario pengujian *static obstacle, dynamic obstacle* maupun tanpa *obstacle* dengan kecepatan agen 6 unit/detik.

Dari seluruh pengujian yang dilakukan metode pengukuran *Euclidean Distance* memberikan nilai rata-rata waktu tempuh agen menuju target paling cepat dibandingkan dengan *Manhattan Distance*.

Pengujian tanpa obstacle memberikan hasil pada metode pengukuran *Euclidean Distance*, agen dapat mencapai target dalam waktu 11.76 detik. Pada metode pengukuran *Manhattan Distance* memberikan hasil lebih lambat 14.42% dari *Euclidean Distance*.

Pengujian dengan *static obstacle* memberikan hasil pada metode pengukuran *Euclidean Distance*, agen dapat mencapai target dalam waktu 12.21 detik. Pada metode pengukuran *Manhattan Distance* memberikan hasil lebih lambat 3.97% dari *Euclidean Distance.*

Pengujian dengan *dynamic obstacle* memberikan hasil pada metode pengukuran *Euclidean Distance*, agen dapat mencapai target dalam waktu 13.62 detik. Pada metode pengukuran *Manhattan Distance* memberikan hasil lebih lambat 2.38% dari *Euclidean Distance*.

Referensi

Russel, S., & Norvig, P. (1995). *Artificial Intelligent: A Modern Approach.* Prentice Hall International, Inc.

Sturtevant, N. R., Felner, A., Barrer, M., & Burch, J. S. (2009). Memory-Based Heuritics for Explicit State Spaces. *Proceedings of the Twenty-First International Joint Conference on Artificial Intelligent (IJCAI-09)*.

Koenig, S., & Likhachev, M. (2002). D* Lite. *Proceedings of AAAI Conference of Artificial Intelligent (AAAI).*

Anthony Stentz. (1995). The Focussed D* Algorithm for Real-Time Replanning. *Proceedings of the International Joint Conference on Artificial Intelligence*, (hal. 1652–1659).

Millington, I., & Funge, J. (2009). *Artificial Intelligent for Games* (2nd ed.). Morgan Kaufmann.

Schwab, B. (2004). *AI Game Engine Programming* (Vol. Charles River Media).

Mateas, M. (2003). Expressive AI: Games and Artificial Intelligent. *Proceddings of Level Up: Digital Games Research ConferenceI.* Utrecht.

Coppin, B. (2004). *Artificial Intelligent Illuminated* (1st ed.). Sudbury: MA: Jones and Bartlett.

Bakie, R. T. (2010). Games and Society. Dalam S. Rabin, *Introduction to Game Development* (hal. 43-58). Boston, MA: Charles River Media.

Williams, R., & Clippinger, C. A. (2002). Aggression, Competition and Computer Games: Computer and Human Opponents. Dalam *Computers in Human Behavior* (Vol. 18, hal. 495-506).

Colwell, J. (2007, December). Needs Met Through Computer Game Play Among Adolescents. *43*, hal. 2072-2082.

Pagulayan, R. J., Keeker, K., Wixon, D., Romero, R., & Fuller, T. (2002). User-Centered Design in Games. Dalam H. f.-C. Systems. Mahmah, NJ: Lawrence Erlbaum Associates.

Michael, J. E. (2001). Human-Level AI's Killer Application: Interactive Computer Games. Dalam *AI Magazine* (Vol. 22).

Russel, S., & Norvig, P. (2002). Artificial Intelligent: A Modern Approach. Prentice Hall.

Rabin, S. (2010). Artificial Intelligent: Agents, Architecture and Techniques. Dalam S. Rabin, *Introduction to Game Development* (2nd ed., hal. 521-528). Charles River Media.

Newborn, M., & Newborn, M. (1997). *Kasparov Vs. Deep Blue: Computer Chess Comes of Age.* New York: Springer-Verlag.

Gregory, J. (2009). *Game Engine Architecture.* Wellesley MA: A K Peters Ltd.

[halaman ini sengaja dikosongkan]