Jurnal Pengembangan Teknologi Informasi dan Ilmu Komputer e-ISSN: 2548-964X
Vol. 3, No. 9, September 2019, hlm. 8599-8607 http://j-ptiik.ub.ac.id

Penerapan *Theta* Pathfinding* untuk Navigasi *Non-Player Character* pada Gim *Maze*

Akbar Ramadhan[1], Eriq Muhammad Adams Jonemaro[2], Muhammad Aminul Akbar[3]

Program Studi Teknik Informatika, Fakultas Ilmu Komputer, Universitas Brawijaya
Email: [1]akbrrmdhn@student.ub.ac.id, [2]eriq.adams@ub.ac.id, [3]muhammad.aminul@ub.ac.id

Abstrak

Perkembangan *Video Game* semakin maju seiring berkembangnya zaman, dengan seperti contoh *Computer Graphic* yang menghasilkan produk yang hampir setara dengan kenyataan, dan banyaknya *game engine* yang tersedia untuk pengembang, membantu perkembangan *game development* dan industri gim. Salah satu aspek dari *game development* yang turut berkembang adalah *Artificial Intelligence (AI)*. *AI* dapat membuat gim terasa nyata dengan membuat kecerdasan yang dapat mendekati pola pikir manusia. Salah satu penerapan dari *AI* yang sering dipakai adalah *pathfinding. Pathfinding* merupakan metode pencarian jalur dari suatu titik ke titik tertentu yang akan dijalankan oleh *AI*. Metode *pathfinding* yang paling sering digunakan adalah *A**, karena *A* pathfinding* dipastikan dapat memberikan hasil pencarian yang paling optimal. Namun, jalur yang dihasilkan belum tentu jalur terdekat, karena jalur alternatif lainnya yang dilewati masih dapat dioptimalkan dengan *post-processing technique.* Banyak metode alternatif selain *A**, salah satunya adalah *Theta* Pathfinding*, yang merupakan algoritma yang dikembangkan lebih lanjut dari *A* Pathfinding.* Penelitian ini membahas *Theta* Pathfinding* dan performanya dibandingkan *A* Pathfinding* dalam gim *maze.* Hasil dari pengujian menunjukkan bahwa *Theta* Pathfinding* menghasilkan pencarian jalur yang lebih optimal dengan waktu yang lebih singkat dibandingkan *A* Pathfinding.*

Kata kunci: *artificial intelligence, pathfinding, A* (A star), Theta*(Theta star), maze, Non-Player Character (NPC)*

Abstract

Video Game development has grown as time passes, with an example of Computer Graphic which resulted in products that are almost on par with reality. Varieties of game engine available for developers helps the growth of game development and game industries. One aspect of game development which has also grown is Artificial Intelligence (AI). AI has the ability to give the game a feeling of reality, with its intelligence that is almost like that of a human. One use of AI that is frequently used is Pathfinding. Pathfinding is a searching method from one point to another driven by AI. A is the most frequently used pathfinding, because A* is guaranteed to give the optimal path. However, the generated path is not always the closest path, as there are other omitted paths which can optimized further with post-processing technique. There are lots of alternative beside A*, one example is Theta* Pathfinding, which is an algorithm that was developed from A*. This research discusses Theta* Pathfinding and its performance compared to basic A* Pathfinding in a maze game. The result of this research shows that Theta* Pathfinding generates a shorter, more optimal route and a shorter time compared to that of A* Pathfinding.*

Keywords: *artificial intelligence, pathfinding, A* (A star), Theta*(Theta star), maze, Non-Player Character (NPC)*

1. PENDAHULUAN

Video game sudah mengalami perkembangan yang cukup pesat seiring berjalannya zaman, dari yang awalnya hanya semacam permainan memantulkan bola seperti bermain ping pong, yaitu Pong yang rilis pada tahun 1958, sampai dengan *video game* yang berjenis *battle royale* seperti PlayerUnknown's Battlegrounds, atau *Multiplayer Online Battle Arena (MOBA)* seperti DOTA 2. Tidak menutup

kemungkinan, saat ini masa depan *video game* masih dapat berkembang lebih jauh, dibuktikan dengan adanya teknologi *Augmented Reality* dan *Virtual Reality*.

Saat ini sudah banyak *genre* dari *video game* yang telah mengimplementasikan AI (*Artificial Intelligence*) sebagai komponen yang dapat membuat *game* tersebut terasa hidup. AI pada *game*, bersama dengan CI (*Computational Intelligence*) sudah mengalami perkembangan dan kesuksesan dalam penelitian dalam sepuluh tahun terakhir ini (Togelius, 2015). Pengaplikasian AI pada *game* secara umum dapat dikenali dengan bebeapa jenis penerapan, di antaranya adalah *pathfinding, decision making, neural network,* dan masih banyak lagi.

Salah satu contoh dari jenis penerapan *AI* pada *video game* adalah *Pathfinding*, di mana *AI* dirancang agar ia dapat menelusuri level dengan instruksi yang telah diberikan. *Pathfinding* memiliki banyak penerapan, di antaranya di antaranya *BFS* (*Breadth First Search*), *DFS* (*Depth First Search*), *A**, dan lain-lain. *Pathfinding* digunakan untuk menemukan rute tercepat yang dapat ditempuh oleh *AI*. *A** selalu mencari dan menemukan rute terbaik di antara titik awal dan titik akhir dalam waktu yang singkat.

A Pathfinding* sudah banyak diterapkan pada berbagai macam *video game*, terutama pada *Non-Player Character (NPC)*. *NPC* adalah karakter selain pemain yang ada di dalam gim. Peran *NPC* dapat berupa teman, musuh, *guide*, atau karakter yang tidak berhubungan dengan pemain. Jika membahas konteks *pathfinding* pada *NPC*, umumnya *pathfinding* digunakan untuk melihat dua titik yang ada dan mengejar *player* (I. Millington, 2009). Implementasi *pathfinding* pada *NPC* dapat digunakan sebagai pemandu pemain dalam menelusuri sebuah *level*.

Contohnya terdapat sebuah *maze* yang cukup luas dan tidak ada petunjuk, hanya tujuan akhir. *A** dapat dengan mudah melacak jalur yang optimal dengan menghitung *grid* yang ada di dalam *maze*, namun belum tentu hasilnya merupakan jalur yang terdekat. Maka dari itu, terdapat algoritme yang didasarkan dari *A** dan tidak dibatasi dengan *grid*, namun dengan menggunakan *edge* sebagai perantara, yaitu *Theta* Pathfinding*. Dikarenakan pencarian menggunakan *A** belum tentu merupakan hasil yang terdekat akibat pencariannya berbasis *node*, maka *Theta** dapat menghasilkan rute yang lebih singkat dengan metode pencariannya yang berbasis *edge* (Firmansyah, 2016). Lalu *NPC* digunakan sebagai media untuk melakukan pencarian jalur dalam sebuah *level*. Untuk membuktikan metode yang lebih efektif untuk menentukan jalan bagi *NPC*, *level* berbentuk *maze* akan digunakan sebagai media untuk pengujian. *Maze* merupakan sebuah teknik yang membingungkan, di mana *maze* akan ditelusuri oleh pemecah menggunakan rute yang paling efisien dalam waktu yang sesingkat mungkin untuk mencapai tujuan (Barnouti, 2016).

2. DASAR TEORI

2.1. A* Pathfinding

Algoritma *A* Pathfinding* merupakan metode *pathfinding* yang paling sering digunakan dalam pengembangan gim. *A* Pathfinding* merupakan hasil penggabungan algoritma Djikstra dengan Greedy Best First Search (Firmansyah, 2016). *A* Pathfinding* sering digunakan dalam gim karena sangat efisien dalam pencarian jalur, dengan menghitung *node-node* yang akan dilalui oleh *AI* dari posisi awal hingga posisi akhir. Mekanisme kerja dari *A* Pathfinding* cukup sederhana, dengan *pseudocode* yang terdiri dari tiga fungsi utama, yaitu *Main*, *UpdateVertex*, dan *ComputeCost*.

```
1  Main()
2      open := closed := ∅;
3      g(s_start) := 0;
4      parent(s_start) := s_start;
5      open.Insert(s_start, g(s_start) + h(s_start));
6      while open ≠ ∅ do
7          s := open.Pop();
8          if s = s_goal then
9              return "path found";
10         closed := closed ∪ {s};
11         foreach s' ∈ nghbr_vis(s) do
12             if s' ∉ closed then
13                 if s' ∉ open then
14                     g(s') := ∞;
15                     parent(s') := NULL;
16                 UpdateVertex(s, s');
17     return "no path found";
18 end
19 UpdateVertex(s, s')
20     g_old := g(s');
21     ComputeCost(s, s');
22     if g(s') < g_old then
23         if s' ∈ open then
24             open.Remove(s');
25         open.Insert(s', g(s') + h(s'));
26 end
27 ComputeCost(s, s')
28     /* Path 1 */
29     if g(s) + c(s, s') < g(s') then
30         parent(s') := s;
31         g(s') := g(s) + c(s, s');
32 end
```

Gambar 1. Pseudocode A*

Pseudocode A Pathfinding* dapat dilihat pada Gambar 1. Saat pertama kali memulai pencarian, terlebih dahulu diinisialisasikan dua *ArrayList* yang bernama *OpenSet* dan *ClosedSet*. *OpenSet* akan berisi *node-node* yang berpotensi menjadi *neighbor* yang nantinya akan menjadi jalur utama untuk *pathfinding*, dan *ClosedSet* akan berisi *node-node* yang tidak akan dilewati atau *node* yang sudah dilewati sebelumnya. Pencarian dimulai dari titik start, di mana *Non-Player Character (NPC)* berada. Titik awal dari *NPC* akan masuk ke dalam *OpenSet*, yang kemudian *AI* akan mengecek *node-node* di sekelilingnya, yang bernama *neighbor*, lalu menghitung masing-masing *cost* dari tiap *neighbor*. *UpdateVertex* dan *ComputeCost* berfungsi untuk membandingkan *cost* dari *node* yang sedang ditempati oleh *NPC* dengan *node-node neighbor*-nya. Jika diperoleh *cost* terkecil antara titik *NPC* saat ini dengan salah satu *neighbor*, maka *node* tersebut akan masuk ke *OpenSet* dan kemudian akan menjadi jalur utama dari *NPC* tersebut.

Tiga komponen utama yang digunakan untuk menghitung *cost* dari *node* ke *node* adalah *fCost*, *gCost*, dan *hCost*. *fCost* merupakan nilai total yang diperlukan untuk menempuh jalur dari titik *node* yang sedang ditempati ke titik akhir, *gCost* merupakan nilai dari *node* yang sedang ditempati menuju *neighbor*, dan *hCost* merupakan nilai dari *node neighbour* menuju titik akhir. Nilai *fCost* didapat dengan menjumlahkan nilai *gCost* dengan *hCost*.

A Pathfinding* memiliki banyak algoritme turunan yang dikembangkan lebih lanjut dan beberapa di antaranya dilakukan Teknik *Post-processing* untuk memperhalus jalur yang dibuat oleh *NPC*. Jalur yang dibuat oleh *A* Pathfinding* dapat dibilang paling optimal, namun jalurnya berbasis *node* dan *grid*, sehingga gerakan yang akan dilakukan oleh *NPC* akan terasa kaku dan tidak natural. Salah satu algoritme turunan *A* Pathfinding* yang memperbaiki masalah ini adalah *Theta* Pathfinding*.

2.2. Theta* Pathfinding

Algoritma *A* Pathfinding* memiliki algoritma turunan yang dikembangkan lebih jauh, dengan tiap algoritma memiliki keunikan dan perbedaannya masing-masing, sesuai kebutuhannya. Salah satu turunan dari algoritma tersebut adalah *Theta* Pathfinding*. Cara kerja algoritma ini kurang lebih sama dengan algoritma *A* Pathfinding*. Hal yang membedakan *Theta* Pathfinding* dengan *Basic A* Pathfinding* adalah jika *Basic A* Pathfinding* memperkirakan jalur melalui *node* yang bersebelahan dan berurutan, *Theta* Pathfinding* dapat menentukan jalur berdasarkan *node* yang terlihat di depan mata, dan tidak harus bersebelahan. Karena *Basic A* Pathfinding* memiliki kelemahan di mana jalur yang dihasilkan belum tentu jalur yang terdekat yang disebabkan oleh metode pencarian *A* Pathfinding* yang menghitung *cost* dan *neighbor* berdasarkan *grid*, *Theta* Pathfinding* digunakan untuk menemukan rute terdekat dengan menghitung ujung-ujung dari tiap sisi atau sudut (Firmansyah, 2016).

```
1  ComputeCost(s, s')
2      if LineofSight(parent(s), s') then
3          /* Path 2 */
4          if g(parent(s)) + c(parent(s), s') < g(s') then
5              parent(s') := parent(s);
6              g(s') := g(parent(s)) + c(parent(s), s');
7      else
8          /* Path 1 */
9          if g(s) + c(s, s') < g(s') then
10             parent(s') := s;
11             g(s') := g(s) + c(s, s');
12 end
```

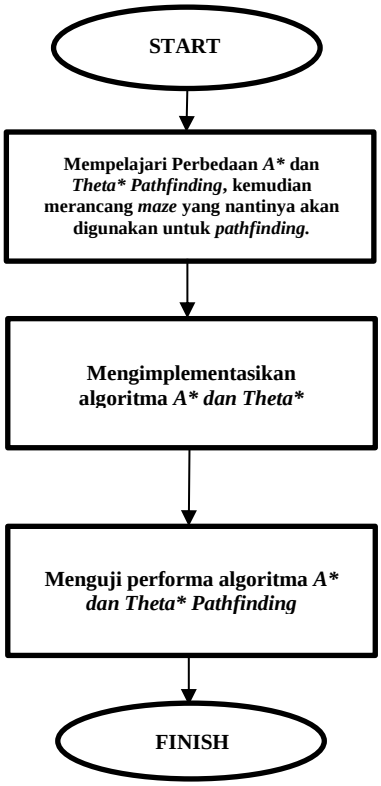
Gambar 2. Pseudocode Theta*

Perbedaan utama yang membedakan *Theta* Pathfinding dengan A* Pathfinding* adalah pada *method ComputeCost* milik *Theta**, seperti yang dapat dilihat pada Gambar 2. Jika pencarian *neighbor* pada *A** hanya dapat dilakukan dengan melihat *node-node* di sekitar titik yang sedang ditempati, *Theta** memperluas jangkauan *neighbor* yang dapat dihitung dengan menggunakan *Line-of-sight()* yang nantinya akan digunakan untuk melihat *edge* yang ada di depan jangkauan mata *NPC*, dan memasukkan *node* yang terdapat di *edge* tersebut ke dalam *neighbor*. Maka dari itu, *neighbor* pada *Theta** tidak terbatas pada *node* di sekitar titik yang ditempati. Selain itu, hasil dari *Line-of-sight* juga memberikan kemampuan bagi *NPC* untuk menempuh jalur tanpa dibatasi oleh *node*, sehingga membuat jalur yang ditempuh menjadi lebih halus dan tidak terasa kaku.

3. METODOLOGI

Pada bab ini akan menjelaskan tentang langkah-langkah yang dilakukan dalam penelitian ini. Penelitian ini bersifat Implementatif, dengan menggunakan algoritme *A* Pathfinding* dan *Theta* Pathfinding*. Metode

Penelitian yang digunakan adalah :



Gambar 3. Metodologi

Garis besar penelitian dapat dilihat pada Gambar 3. Tahapan pertama adalah studi literatur, yang di mana dilakukan pembelajaran mengenai *pathfinding*, yang nantinya akan berguna untuk menjadi dasar teori penelitian.

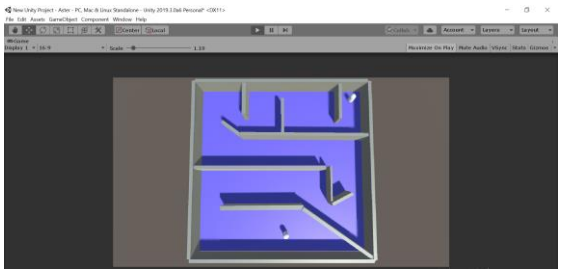
Kemudian penelitian dilanjutkan kepada tahapan kedua, yaitu perancangan dan implementasi. Perancangan dilakukan dengan membangun *level* yang di mana nantinya akan dilakukan pengujian. Lalu dilanjutkan dengan implementasi yang berupa pengaplikasian kode untuk *pathfinding.*

Lalu dilanjutkan ke tahapan terakhir, yaitu pengujian, yang di mana dilakukan pengujian terhadap kedua algoritme, untuk dibandingkan performanya.

4. PERANCANGAN DAN IMPLEMENTASI

4.1 Rancangan *Level*

Level yang akan dikembangkan menjadi *maze* berdimensi 3D. *Level* berisi *plane* dan *cube*. *Plane* berguna untuk tanah bagi objek, baik *Seeker*, *target*, ataupun *obstacles* yang berupa *Cube* berada. Sedangkan *Cube* berperan sebagai *obstacles* atau halangan yang berguna untuk menentukan jalur tempuh *NPC*. *Terrain*, atau keadaan tanah dari *maze* tidak memiliki perbedaan. Tidak ada kondisi yang dapat mempercepat atau memperlambat Gerakan *NPC*, seperti es, atau lumpur. Artinya bobot dari setiap *grid* adalah sama. Titik awal dan akhir dari *maze* ditentukan oleh posisi *NPC* yang dinamakan *seeker* dan *target*. Posisi *seeker* akan dipindahkan sesuai keperluan pengujian.



Gambar 4. Tampilan *Level Maze*

Gambar 4 menggambarkan *layout* dari *maze* yang akan digunakan untuk *pathfinding*. *Maze* tersebut hanya berfungsi untuk menampung objek-objek yang nantinya berperan untuk *pathfinding*. Setelah perancangan *maze* selesai, dilanjutkan ke tahap Implementasi Kode.

4.2 Implementasi Kode

Tahapan Implementasi akan membahas tentang *source code* dan *pseudocode* dari *class* yang diimplementasikan pada penelitian ini.

4.2.1 *A* Pathfinding*

Algoritma *A* Pathfinding*, seperti yang sudah dibahas sebelum-sebelumnya, melakukan pencarian jalur menggunakan data yang ada dalam grid, yaitu posisi awal, posisi akhir, dan *obstacles*. *Pseudocode A* Pathfinding* dapat dilihat pada Tabel 1 berikut.

*Pseudocode A**	
1	`Start`
2	
3	`Node start`
4	`Node finish`
5	
6	`ArrayList openList`
7	`ArrayList closedList`
8	`openList.Add(start)`
9	`start.parent = start`
10	
11	`While (openList > 0)`
12	`  currentNode = openList[0]`

13	For(int i=1;i<openList.count;i++)
14	If(openList[i].fCost < current.fCost)
15	currentNode = openList[i]
16	openList.Remove(currentNode)
17	closedList.Add(currentNode)
18	
19	If (currentNode = finish)
20	Retracing(start, finish)
21	
22	Foreach(neighbor in currentNode.neighbor)
23	If(neighbor not passable || closedList contains neighbor)
24	Continue
25	Float newCostToNeighbor = currenNode.gCost + getDistance(currentNode + neighbor)
26	If(newCostToNeighbor < neighbor.gCost || openList doesn't contain neighbor)
27	Neighbor.gCost = newCostToNeighbor
28	Neighbor.hCost = getDistance(neighbor, finish)
29	Neighbor.parent = currentNode
30	
31	If openList doesn't contain Neighbor
32	Neighbor.Add(openList)

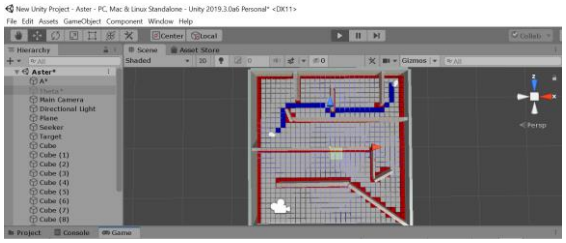
Tabel 1. *Pseudocode A* Pathfinding*

Pertama-tama, diinisialisasi terlebih dahulu posisi awal dan akhir pencarian, dinamakan *start* dan *finish*. Kemudian inisialisasi juga *ArrayList OpenList* dan *ClosedList*, untuk menampung data *node* yang berpotensi dilewati *NPC* dan data *node* yang sudah dilewati atau tidak perlu dilewati. Untuk memulai, *A** dimasukkan ke dalam *OpenList*, sebagai titik awal pencarian. Kemudian parent dari *start* bernilai *node* start itu sendiri.

Kemudian untuk melakukan pencarian *OpenList*, dilakukan perulangan. Sebelumnya, titik yang sedang ditempati, bernama *currentNode*, masuk ke dalam *OpenList[0]*. Dalam perulangan, jika ada *node* dalam *OpenList* yang memiliki *fCost* yang lebih kecil dibanding *currentNode*, maka pindahkan *currentNode* ke *node* tersebut. Kemudian keluarkan *currentNode* dari *OpenList*, dan masukkan *currentNode* ke dalam *ClosedList*.

Jika *currentNode* sudah sampai pada finish, maka hentikan pencarian dan panggil method *Retracing()* untuk melakukan pembuatan jalur.

Lalu untuk penghitungan *cost* pada neighbor, lakukan pencarian untuk setiap *neighbor* yang dimiliki *currentNode*. Jika *neighbor* tidak dapat dilewati, atau *ClosedList* berisi *neighbor*, *neighbor* tersebut dapat dihindari dan tidak perlu dihitung *cost*-nya. Kemudian buat variabel baru yang bernama *newCostToNeighbor*, yang berisi hasil penjumlahan *gCost currentNode* dan *GetDistance(currentNode, Neighbor)*. Kemudian bandingkan lagi, jika *newCostToNeighbor* memiliki *gCost* yang lebih kecil dibandingkan *gCost neighbor*, atau jika *neighbor* tidak ada dalam *OpenList*, ganti *gCost* miliki *neighbor* dengan *newCostToNeighbor*. Lalu hitung *hCost* milik *neighbor* dengan *method GetDistance(neighbor, finish)*. Kemudian, ganti *parent* milik *neighbor* dengan *currentNode*. Hasil akhirnya akan muncul berupa jalur yang menghubungkan antara titik start dan finish, seperti yang dapat dilihat pada Gambar 4.



Gambar 4. Hasil pencarian *A* Pathfinding*

3.2.2 *Theta* Pathfinding*

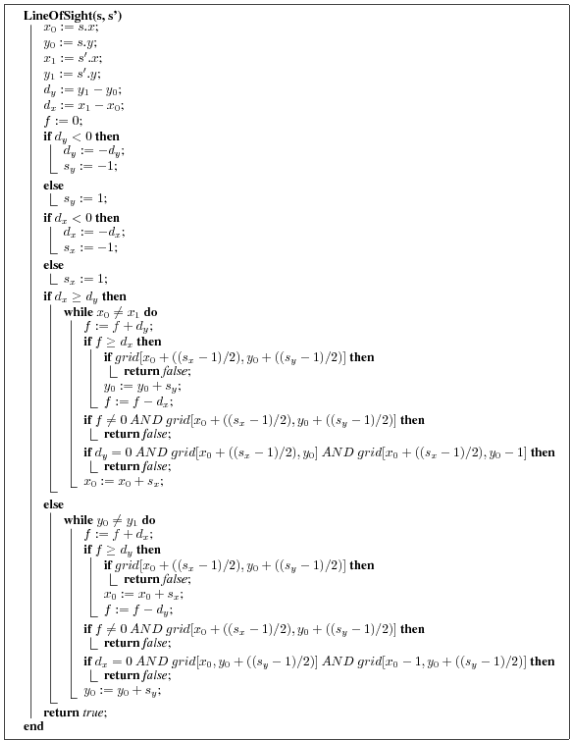
*Theta** memiliki alur utama yang sama dengan *A**, hanya saja terdapat beberapa perbedaan di dalamnya.

*Pseudocode Theta**	
1	Start
2	
3	Node start
4	Node finish
5	
6	ArrayList openList
7	ArrayList closedList
8	openList.Add(start)
9	start.parent = start
10	
11	While (openList > 0)

12	currentNode = openList[0]
13	For(int i=1;i<openList.count;i++)
14	If(openList[i].fCost < current.fCost)
15	currentNode = openList[i]
16	openList.Remove(currentNode)
17	closedList.Add(currentNode)
18	
19	If (currentNode = finish)
20	Retracing(start, finish)
21	
22	Foreach(neighbor in currentNode.neighbor)
23	If(neighbor not passable || closedList contains neighbor)
24	Continue
25	Float newCostToNeighbor
26	If(line of sight(current.parent.position, neighbor.position))
27	If(newCostToNeighbor < neighbor.gCost || openList doesn't contain neighbor)
28	Neighbor.gCost = newCostToNeighbor
29	Neighbor.hCost = getDistance(neighbor, finish)
30	Neighbor.parent = currentNode.parent
31	
32	If openList doesn't contain Neighbor
33	Neighbor.Add(openList)
34	
35	else
36	If(newCostToNeighbor < neighbor.gCost || openList doesn't contain neighbor)
37	Neighbor.gCost = newCostToNeighbor
38	Neighbor.hCost = getDistance(neighbor, finish)
39	Neighbor.parent = currentNode
40	
41	If openList doesn't contain Neighbor
42	Neighbor.Add(openList)

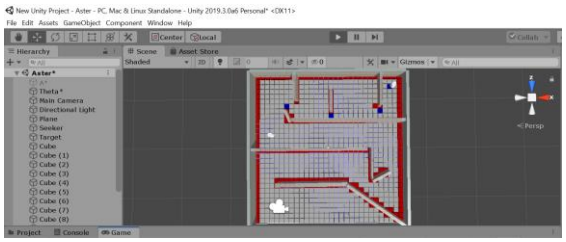
Tabel 2. *Pseudocode Theta**

Seperti yang dapat dilihat pada Tabel 2, struktur dari *Theta** kurang lebih sama seperti *A**. Satu hal yang menjadi pembeda antara *Theta** dan *A** adalah *Line-of-sight*. Dalam Unity3D, *Line-of-sight* sudah dapat diganti dengan fungsi bawaan *Physics.Linecast()*, jadi tidak perlu membuat kode untuk *Line-of-sight* dari awal, seperti yang dijelaskan oleh gambar 4.



Gambar 4. *Pseudocode Line-of-Sight*

Dengan fungsi bawaan Physics.Linecast, *pseudocode* di atas dapat disederhanakan menjadi satu fungsi. Linecast berguna untuk mengembalikan nilai bahwa terdapat jarak yang dapat dilihat kasat mata oleh titik awal ke titik tujuan, yang biasanya adalah *neighbor*. Linecast menghilangkan batasan bahwa *neighbor* hanya bisa berupa *node* yang bersebelahan dengan *node* awal, namun juga dapat berupa *node* yang jauh, selama masih dalam penglihatan. Hasil akhir dari *Theta* Pathfinding* dapat dilihat pada Gambar 5, dengan *node* yang ditempuh tidak sebanyak *A* pathfinding*.

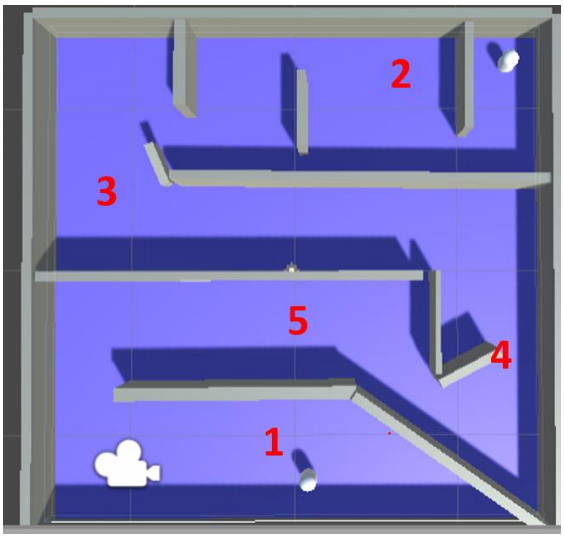


Gambar 5. Hasil pencarian *Theta* Pathfinding*

5. PENGUJIAN

Pada tahapan pengujian, akan dilakukan

pengujian antara dua algoritme yang sebelumnya sudah diimplementasikan. Pengujian dilakukan dengan meletakkan posisi awal pencari di lima titik yang berbeda, seperti yang dapat dilihat pada Gambar 5.1, lalu akan diukur *FPS*, waktu tempuh, dan *node* yang dicek selama pencarian berlangsung.



Gambar 6. Lima Titik Lokasi Pengujian Pencarian

5.1 Pengujian Waktu Tempuh

Pengujian terkait waktu tempuh yang dialami oleh kedua algoritme dilakukan dan dihitung dari awal *NPC* dijalankan dari titik mula sampai *NPC* menyentuh titik target. Keterangan mengenai lamanya waktu tempuh yang dialami oleh kedua algoritme dapat dilihat pada Tabel 3 berikut.

	A Pathfinding*	*Theta* Pathfinding*
Titik 1	1:05	0:13
Titik 2	0:07	0:04
Titik 3	0:23	0:07
Titik 4	0:45	0:09
Titik 5	0:53	0:11

Tabel 3. Hasil Pengujian Waktu Tempuh Kedua Algoritme *Pathfinding*

5.2 Pengujian *FPS Loss*

Pengujian terkait penurunan *FPS* yang dialami oleh kedua algoritme dapat dilihat pada Tabel 4. Hasil pengujian menggambarkan penurunan *FPS* yang dialami oleh kedua algoritme dari titik awal pencarian sampai ke tujuan.

	A Pathfinding*	*Theta* Pathfinding*
Titik 1	37.6→10.9	37.6→24.6
Titik 2	39.6→32.5	39.6→38.6
Titik 3	38.0→22.6	40.0→36.5
Titik 4	48.2-→16.2	48.3→34.3
Titik 5	43.2→13.6	46.5→30.6

Tabel 4. Hasil Pengujian *FPS Loss* Kedua Algoritme *Pathfinding*

5.3 Pengujian *Node Check*

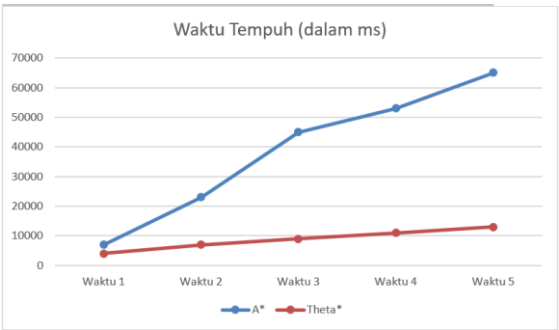
Pengujian *Node Check* pada kedua algoritme pencarian dapat dilihat pada Tabel 5. Pengujian ini bertujuan untuk menggambarkan banyaknya *node* yang dicek oleh kedua algoritme selama pencarian jalur berlangsung.

	A Pathfinding*	*Theta* Pathfinding*
Titik 1	276	289
Titik 2	30	29
Titik 3	162	168
Titik 4	206	212
Titik 5	2665	282

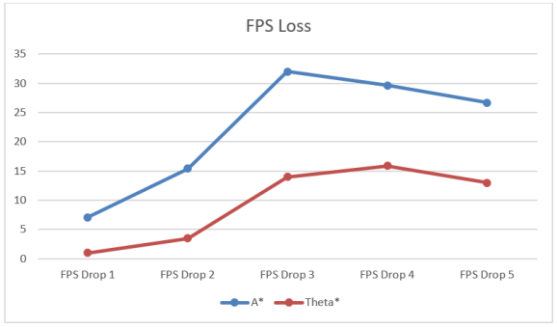
Tabel 5. Hasil Pengujian *Node Check* Kedua Algoritme *Pathfinding*

5.6 Hasil Akhir

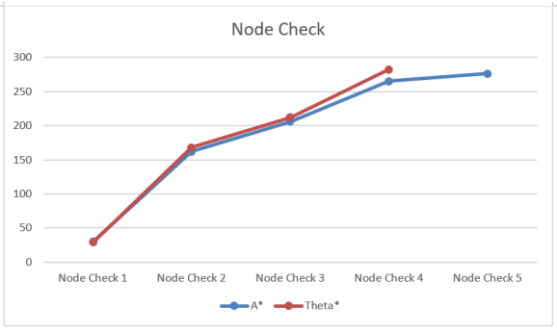
Dari ketiga percobaan, data-data yang diperoleh dapat dijadikan grafik yang dapat mendata hasil uji coba antara kedua algoritma *pathfinding.*



Gambar 7. Grafik Waktu Tempuh



Gambar 8. Grafik FPS Loss



Gambar 9. Grafik Node Check

Dari ketiga grafik di atas dapat disimpulkan bahwa *Theta** dapat menemukan jalur yang lebih optimal dibandingkan dengan *A**, meskipun semakin panjang jarak antara titik awal dan titik akhir *Theta** mengecek *node* yang lebih banyak. Waktu tempuh yang dialami oleh kedua *pathfinding* juga terlihat perbedaannya. *A** mengalami waktu tempuh yang lama karena *A** melewati *node* demi *node* yang ia temukan, sedangkan *Theta** hanya menemukan *node* di tiap ujung belokan dan melewati *node-node* tersebut. *FPS Loss* juga berhubungan dengan banyaknya *node* yang dicek pada tiap pencarian. *A** relatif mengalami penurunan *FPS* yang lebih banyak dan lebih sering dibanding *Theta** karena *A** memakan waktu lama dalam menempuh jalur dan rendering pun akan terus berjalan seiring dengan berjalannya *NPC*.

6. PENUTUP

6.1 Kesimpulan

Penelitian tentang Penerapan *Theta* Pathfinding* untuk Navigasi Non-Player Character pada gim *Maze* berhasil dilakukan berdasarkan studi literatur dan implementasi yang telah dibahas sebelumnya, dan telah dilakukan pengujian sesuai dengan rumusan masalah yang sebelumnya dibahas. Kesimpulan yang dapat ditarik dari pengujian antara lain sebagai berikut.

1. Algoritma *Theta* Pathfinding* berhasil diterapkan pada *Non-Player Character* dengan menambahkan *method Line-of-Sight()* pada algoritma *A* Pathfinding* yang sebelumnya telah diimplementasikan. Algoritma *Theta* Pathfinding* berjalan dengan sukses, dengan *AI* menempuh jalan antar *edge* yang ditemuinya melalui *Line-of-Sight.*
2. Perbandingan performa antara *Theta* Pathfinding dengan Basic A* Pathfinding* telah diuji pada Bab 5, dengan menggunakan tiga kriteria, yaitu waktu tempuh, *FPS Loss*, dan *Node* Check. Dari kelima percobaan dapat disimpulkan bahwa *Theta* Pathfinding* memiliki kinerja yang lebih baik dibandingkan dengan *A**, dibuktikan dengan waktu tempuh yang lebih singkat dan *FPS Loss* yang lebih sedikit, meskipun memakan resource yang lebih banyak dengan mengecek *node* yang lebih banyak disbanding *A* Pathfinding.*

6.2 Saran

Berdasarkan penelitian yang telah diterapkan sebelumnya, algoritma *Theta* Pathfinding* masih dapat dikembangkan lagi. Dari sisi pengembang, ada beberapa aspek yang dapat dikembangkan lebih lanjut.

Algoritma *Theta* Pathfinding* yang baru saja diterapkan belum menerapkan Heap Optimization, maka jalur yang diambil masih sebatas pencarian biasa saja, dan belum mengalami optimasi.

Selain Optimasi, dapat juga dikembangkan dengan menerapkan algoritma tersebut pada *map* yang strukturnya belum diketahui sebelumnya, atau *random-generated map.*

7. DAFTAR PUSTAKA

Yannakakis, G. N., & Togelius, J. (2015). *A Panorama of Artificial and Computational Intelligence in Games. IEEE Transactions on Computational Intelligence and AI in Games, 7(4), 317–335.* doi:10.1109/tciaig.2014.2339221

Firmansyah, E. R., Masruroh, S. U., & Fahrianto, F. (2016). *Comparative Analysis of A* and Basic Theta* Algorithm in Android-Based Pathfinding Games. 2016 6th International Conference on Information and Communication Technology for The*

Muslim World (ICT4M). doi:10.1109/ict4m.2016.063

I. Millington dan J. Funge. 2009. Artificial Intelligence for Games 2nd Edition. Elsevier, Oxford, UK.

N. H. Barnouti. (2016) Path finding in Strategy Games and Maze Solving Using A* Search Algorithm. *Journal of Computer and Communication*, 4, 15-25. http://dx.doi.org/10.4236/jcc.2016.411002