Penerapan Event-Driven Behavior Tree dalam Permainan Berbasis Kecerdasan Buatan

Holy Lovenia / 13515113
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
13515113@std.stei.itb.ac.id
holy.lovenia@gmail.com

***Abstrak*—Seiring dengan perkembangan zaman, semakin banyak permainan yang menggunakan intelejensia buatan sebagai basis koordinasi dari sistemnya. Komponen utama dari permainan adalah *Non Playable Character* (NPC) dan objek spesial yang dapat memberikan respon terhadap suatu kondisi tertentu secara otomatis. Kedua hal ini kerap ditemukan dalam mode penjelajahan maupun mode pertarungan permainan. *Event-Driven Behavior Tree* merupakan salah satu implementasi pohon yang terbaru dan bertujuan untuk mengontrol perilaku dan reaksi sesuai dengan situasi yang sedang berlangsung.**

***Kata Kunci*—*Artificial Intelligence*, Behavior Tree, Event-Driven, Kecerdasan Buatan.**

I. PENDAHULUAN

Intelejensia buatan merupakan sebuah model komputasi untuk menyelesaikan masalah, di mana masalah tersebut memiliki tingkat kompleksitas yang setara dengan persoalan yang dihadapi oleh manusia pada umumnya. Kecerdasan buatan kemudian diintegrasikan dalam berbagai bidang seperti kesehatan, transportasi, hiburan dan lainnya. Aplikasi dari intelejensia buatan dapat ditemukan sehari-hari, contohnya mobil pintar, analisis detak jantung, dan humanoid.

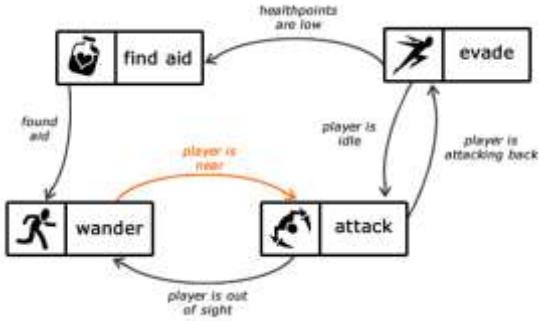
Tidak ketinggalan, simulasi kecerdasan mirip manusia ini kemudian diimplementasikan pada hampir semua permainan untuk mengarahkan jalannya permainan. Beberapa contoh permainan yang menggunakan intelejensia buatan adalah *The Sims*, *Halo: Combat Evolved*, dan *Black & White*. Intelejensia buatan mengorrdinasikan semuanya, mulai dari mengatur grafis, suara, naskah, alur cerita, hingga tindakan-tindakan NPC dan respon terhadap objek spesial. NPC tentu harus bisa menyaring dan menganalisis lingkungan sekitarnya agar dapat memberikan reaksi yang spesifik. Contohnya adalah mengetahui di mana posisi pemain, kemampuan bertarung yang ia sendiri miliki, dan status NPC.

Ada berbagai cara yang dapat digunakan untuk mengontrol perilaku NPC pada suatu permainan, misalnya *Hierarchical Finite State Machine* (HFSM), *Behavior Tree*, *Event-Driven Behavior Tree*, dan *Decision Tree*. Masing-masing cara tentu memiliki kekurangan dan potensi tersendiri. Pada makalah ini, penulis akan menjabarkan aplikasi *Event-Driven Behavior Tree* sebagai intelejensia buatan pada permainan.

II. TEORI *FINITE STATE MACHINE*

A. *Finite State Machine* (FSM)

Finite State Machine merupakan mesin berbasis hipotesis yang terdiri dari berbagai *state* yang terdefinisi dan berbeda satu sama lain. *Finite State Machine* juga dapat mendefinisikan kondisi-kondisi di mana *state* harus berubah. Pada dasarnya, *state* terdiri dari dua hal, yaitu set aksi yang berlangsung pada saat yang sama dan set transisi untuk memeriksa kondisi dan menentukan perlu untuk maju ke *state* berikutnya atau tidak.



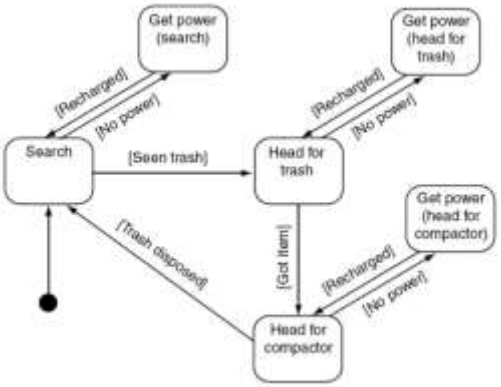
Gambar 1. Contoh Implementasi FSM [1]

Umumnya, *Finite State Machine* digunakan untuk mengatur dan merepresentasikan alur eksekusi dari program. Dalam representasinya sebagai graf, setiap simpul merupakan state atau aksi dari NPC (dalam konteks permainan) dan setiap sisi melambangkan syarat transisi ke state selanjutnya.

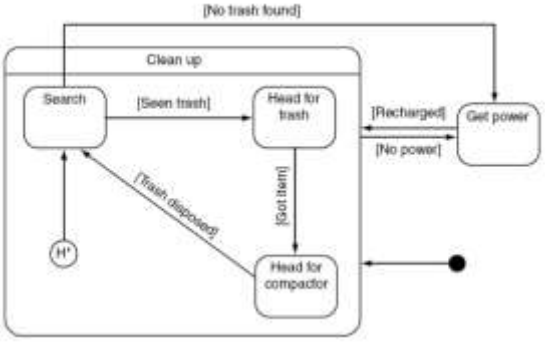
B. *Hierarchical Finite State Machine* (HFSM)

Hierarchical Finite State Machine merupakan pengembangan dari bentuk *Finite State Machine* yang sudah ada. Pada HFSM, *state-state* yang ada dapat dibentuk menjadi beberapa kelompok (*super-state*).

Super-state ini juga dapat memiliki transisi yang berlaku untuk seluruh *state* yang tergabung dalam *super-state* tersebut. Transisi antar *super-state* dikenal sebagai transisi umum (*generalized transition*). Masing-masing *state* hanya boleh masuk ke dalam satu *super-state*.



Gambar 2. Contoh Perbandingan FSM [2]



Gambar 3. Contoh Perbandingan HFSM [2]

Dengan membandingkan Gambar 2. dan Gambar 3., perbedaan FSM dan HFSM dapat jelas terlihat. Aksi *Search, Head for Trash,* dan *Head for Compactor* tergabung dalam satu super-state. Sedangkan, transisi umum adalah [*Recharged*] dan [*No power*].

III. TEORI POHON

A. Pohon

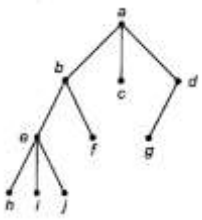
Pohon adalah sebuah struktur data yang merupakan implementasi dari graf tidak berarah yang terhubung dan asiklik (tidak mengandung sirkuit). Karakteristik dari pohon:

1. Simpul berderajat-masuk sama dengan nol disebut sebagai akar.
2. Simpul selain akar akan memiliki derajat-masuk sama dengan satu.
3. Daun atau simpul terminal adalah simpul dengan derajat-keluar sama dengan nol.
4. Simpul atau simpul dalam adalah simpul dengan derajat-keluar lebih dari nol.
5. Setiap simpul pada pohon dicapai dengan lintasan yang tunggal dan unik.
6. Setiap lintasan pada pohon memiliki arah dari akar menuju daun.
7. Arah panah pada lintasan dapat ditiadakan.

B. Terminologi Pohon

Berikut ini adalah terminologi yang umum digunakan:

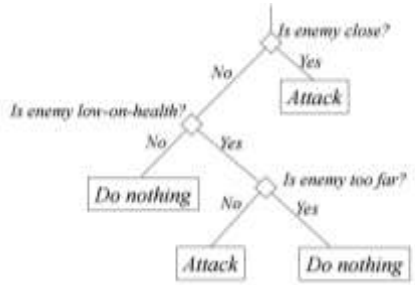
1. Anak (*child*)
   Simpul yang posisinya lebih jauh dari akar dan terhubung ke simpul yang posisinya lebih dekat ke akar. Merujuk ke Gambar 4., anak dari b adalah e dan f.
2. Orang tua (*parent*)
   Simpul yang posisinya lebih dekat ke akar dan terhubung ke simpul yang posisinya lebih jauh dari akar. Merujuk ke Gambar 4., orang tua dari c adalah a.
3. Lintasan (*path*)
   Penghubung antara satu simpul dengan simpul lainnya. Panjang lintasan adalah banyaknya perpindahan yang dilakukan untuk mencapai simpul target. Merujuk ke Gambar 4., lintasan dari a ke i adalah a, b, e, I dan panjang lintasannya adalah 3.
4. Keturunan (*descendant*)
   Simpul yang posisinya lebih jauh dari akar dan terdapat lintasan untuk pergi ke simpul yang lebih dekat ke akar. Merujuk ke Gambar 4., j adalah keturunan dari b.
5. Leluhur (*ancestor*)
   Simpul yang posisinya lebih dekat ke akar dan terdapat lintasan untuk pergi ke simpul lebih jauh dari akar. Merujuk ke Gambar 4., a adalah leluhur g.
6. Saudara kandung (*sibling*)
   Memiliki orang tua yang sama. Merujuk ke Gambar 4., i adalah saudara kandung j.
7. Derajat (*degree*)
   Jumlah simpul yang terhubung langsung dengan simpul tersebut. Merujuk ke Gambar 4., derajat e adalah 4.
8. Aras atau tingkat (*level*)
   Akar memiliki aras yang sama dengan nol, sedangkan simpul lainnya memiliki aras yang senilai dengan 1 + panjang lintasan dari akar ke simpul tersebut. Merujuk ke gambar 4., aras dari f adalah 2.
9. Tinggi (*height*) atau kedalaman (*depth*)
   Aras maksimum dari sebuah pohon, atau merupakan panjang maksimum lintasan dari akar ke daun. Merujuk ke Gambar 4., tinggi pohon adalah 3.
10. Upapohon (*subtree*)
    Sebuah simpul yang berada dalam pohon tersebut beserta keturunannya. Merujuk ke Gambar 4., terdapat upapohon yang mengandung simpul b, e, f, h, i, dan j.



Gambar 4. Contoh Pohon [3]

C. *Decision Tree*

Decision Tree merupakan metode pengambilan keputusan dengan representasi pohon. Simpul dalam melambangkan kondisi yang menjadi pertimbangan, sedangkan daun merupakan aksi yang akan dilakukan apabila semua prasyaratnya terpenuhi. Lintasan pada pohon adalah hasil yang diperoleh dari simpul. Pada umumnya, hasil tersebut berupa nilai benar atau salah.

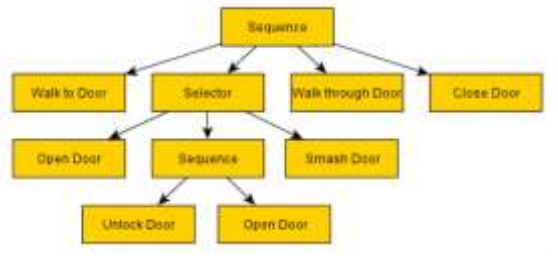


Gambar 5. Contoh Implementasi *Decision Tree* [2]

Mengacu pada *Decision Tree* dalam Gambar 5., aksi Attack hanya terjadi ketika posisi musuh dekat atau (musuh sekarat dan posisi musuh terjangkau), sedangkan NPC akan tidak melakukan apa-apa apabila (posisi musuh jauh dan musuh tidak sekarat) atau (posisi musuh jauh dan posisi musuh tidak terjangkau).

D. *Behavior Tree*

Behavior Tree merupakan pohon yang terdiri dari sekumpulan simpul yang tersusun menurut hirarki dan mengatur proses pengambilan keputusan. Dalam implementasi *Behavior Tree*, setiap simpul memiliki logika dan aturan tersendiri (seperti *state* pada HFSM), dan membatasi transisinya untuk tetap berada pada lingkup *state-state* tersebut. Daun pada *Behavior Tree* mendefinisikan *task* atau *behavior* yang digunakan untuk membentuk *main behavior* atau *root task.*



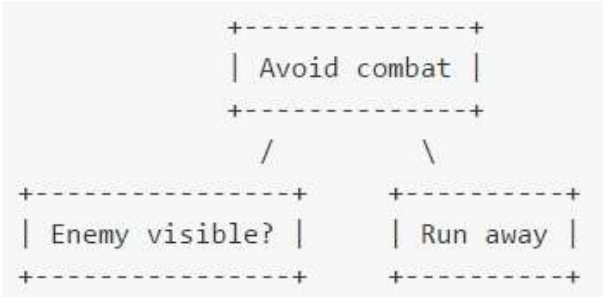
Gambar 6. Contoh *Behavior Tree* [4]

Berdasarkan fungsinya, *task* pada *Behavior Tree* dibagi menjadi dua bagian:

1. *Leaf Task*

   Leaf Task merupakan simpul terminal bertujuan untuk mengeksekusi bagian paling primitif yang dimiliki oleh *main behavior* dan mengembalikan nilai dari suatu *state*.

   a. Simpul Kondisi
      Digunakan untuk mengecek apakah suatu kondisi tertentu sudah terpenuhi.
   b. Simpul Aksi
      Digunakan untuk melakukan komputasi terhadap suatu state pada permainan tersebut untuk mengubahnya.



Gambar 7. Contoh *Leaf Task* [5]

Pada Gambar 7., *root task* terbagi menjadi dua *leaf task*, yaitu simpul kondisi yang mengecek apakah musuh terlihat dan simpul aksi yang membuat NPC melarikan diri. Setiap simpul aksi dan simpul kondisi dapat berhasil ataupun gagal, di mana hasil dari simpul-simpul tersebut akan memengaruhi keberhasilan dari simpul yang merupakan orang tuanya.

Nilai yang dapat dikembalikan oleh simpul pada *Behavior Tree* antara lain adalah:

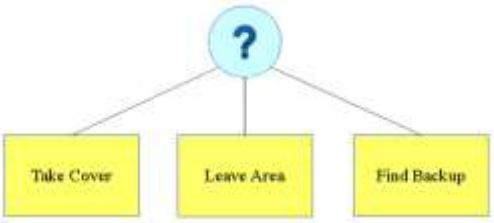
   a. *Success*
      Ketika persyaratan yang diajukan oleh simpul kondisi terpenuhi atau eksekusi dari simpul aksi telah diselesaikan.
   b. *Failure*
      Ketika persyaratan yang diajukan oleh simpul kondisi tidak terpenuhi atau eksekusi dari simpul aksi tidak bisa diselesaikan.
   c. *Running*
      Ketika eksekusi dari simpul aksi sudah dimulai, tetapi program belum dapat menarik kesimpulan untuk hasilnya.

2. *Composite Task*

Composite Task digunakan untuk mendefinisikan dan mengatur hubungan antara simpul-simpul lainnya, seperti bagaimana atau kapan simpul tersebut harus diproses.

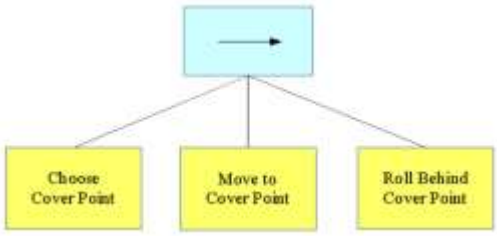
   a. *Selector (or)*
      Selector memproses setiap anaknya secara bergantian. Ketika salah satu dari anaknya mengembalikan nilai *success*, maka *selector* akan segera mengembalikan nilai *success*

tersebut. Apabila anak yang sedang diproses mengembalikan nilai *failure*, maka selector akan memproses anak berikutnya hingga tidak ada anak lagi yang tersisa dan mengembalikan nilai *failure*. Selector direpresentasikan dengan simbol tanda tanya dan kadang disebut sebagai *priority*.



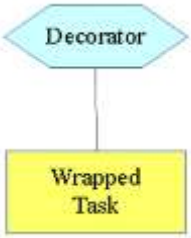
Gambar 8. Contoh *Selector* [5]

b. *Sequence (and)*
Sequence memproses setiap anaknya secara bergantian. Ketika salah satu dari anaknya mengembalikan nilai *failure*, maka *selector* akan segera mengembalikan nilai *failure* tersebut. Apabila anak yang sedang diproses mengembalikan nilai *success*, maka *selector* akan memproses anak berikutnya hingga tidak ada anak lagi yang tersisa dan mengembalikan nilai *success*. *Sequence* direpresentasikan dengan simbol anak panah.



Gambar 9. Contoh *Sequence* [5]

c. *Decorator*
Setiap *decorator* hanya memiliki anak tunggal dan berfungsi untuk memodifikasi *behavior* anak tersebut (*wrapped task*) dengan cara memanipulasi nilai yang dikembalikan atau mengubah frekuensi.

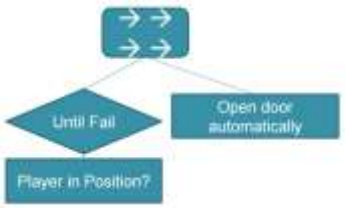


Gambar 10. Contoh *Decorator* [5]

Tipe-tipe *decorator* yang dapat digunakan:

i. *AlwaysFail*
Selalu mengembalikan nilai *failure* tanpa menghiraukan nilai kembali yang sesungguhnya.
ii. *AlwaysSuceed*
Selalu mengembalikan nilai *success* tanpa menghiraukan nilai kembali yang sesungguhnya.
iii. *Include*
Mencantumkan sebuah upapohon eksternal.
iv. *Invert (negation)*
Mengembalikan nilai *succeed* apabila wrapped task gagal dan mengembalikan nilai *failure* apabila *wrapped task* berhasil.
v. *Limit*
Membatasi berapa kali *wrapped task* dapat diproses.
vi. *Repeat*
Mengulangi pemrosesan *wrapped task* sebanyak n kali.
vii. *UntilFail*
Mengulangi pemrosesan *wrapped task* hingga *wrapped task* bernilai *failure*, agar *decorator* dapat mengembalikan nilai *success*.
vii. *UntilSuccess*
Mengulangi pemrosesan *wrapped task* hingga *wrapped task* bernilai *success*, agar *decorator* dapat mengembalikan nilai *success*.

d. *Parallel*
Parallel adalah sebuah *composite task* yang menangani *behavior* yang berlangsung secara bersamaan. *Parallel* memulai atau melanjutkan pemrosesan seluruh anaknya pada satu waktu. Terdapat dua pengaturan yang dapat dimiliki oleh *parallel*, yaitu *sequence policy* (*parallel task* langsung gagal ketika salah satu anaknya mengembalikan nilai *failure*) dan *selector policy* (*parallel task* langsung berhasil ketika salah satu anaknya mengembalikan nilai *succeed*). *Parallel* dilambangkan dengan simbol banyak panah atau banyak *guard* (“|”).



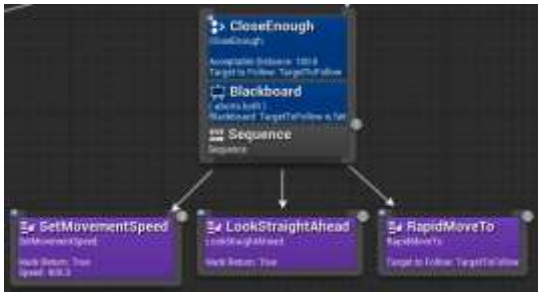
Gambar 10. Contoh *Parallel* [6]

E. *Event-Driven Behavior Tree*

Event-Driven Behavior Tree merupakan modifikasi dan pengembangan dari *Behavior Tree*. Salah satu perubahannya adalah mengenai cara pohon tersebut mengenali *event* pada permainan. *Event* dideskripsikan sebagai suatu kondisi khusus yang menyebabkan perubahan signifikan terhadap *state-state* yang ada. Pada *Behavior Tree* biasa, pohon akan mengecek apakah prasyarat pohon tersebut sesuai dengan kondisi saat ini. Sedangkan, *Event-Driven Behavior Tree* hanya perlu membaca event yang berpotensi untuk membuat perubahan dalam pohon tersebut.

Modifikasi lain yang dilakukan oleh *Event-Driven Behavior Tree* adalah lokasi *condition task*. Dalam pengaturan standar, *condition task* merupakan bagian dari *leaf task*. Dalam pengaturan *Event-Driven Behavior Tree*, *condition task* tidak lagi tergabung di dalam *leaf task*, melainkan menggunakan *decorator* sebagai *condition task*. Perubahan ini membuat pohon lebih mudah dibaca sehingga tampak jelas aksi apa yang sebenarnya akan dieksekusi oleh pohon, karena pembaca tidak lagi perlu memilah *condition task* dan *action task*. Selain itu, representasi *condition task* sebagai *decorator task* juga dapat digunakan pada simpul-simpul penting untuk menunggu *event* yang dapat menginisialisasi pemrosesan selanjutnya.

Untuk melakukan pemrosesan pada simpul anak secara bersamaan, *Event-Driven Behavior Tree* mengalihkan fungsi yang dimiliki oleh simpul *parallel* menjadi simpul *simple parallel* dan *service*. *Simple parallel* hanya memperbolehkan dua anak, yaitu simpul *task* tunggal (dapat disertai dengan *decorator*) dan sebuah upapohon. Pada dasarnya, salah satu simpulnya merupakan *primary task* dan yang lainnya adalah *secondary* atau *filler task*. Sedangkan, *service* adalah simpul spesial yang diasosiasikan dengan *composite node* lainnya. *Service* dapat melakukan pemanggilan kembali setiap n detik dan pembaharuan terhadap setiap aspek yang harus muncul secara periodik. *Service* hanya aktif pada saat eksekusi berada pada upapohon di mana *composite node*-nya diasosiasikan dengan *service* [7].



Gambar 11. Contoh *Event-Driven Behavior Tree* [7]

IV. *EVENT-DRIVEN BEHAVIOR TREE* SEBAGAI SOLUSI TERBAIK

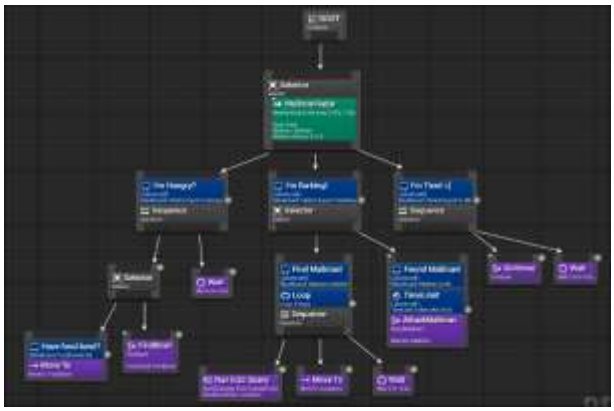
Setelah melakukan perbandingan pada empat alternatif solusi yang tersedia, maka didapatkan hasil analisis sebagai berikut.

1. *Hierarchical Finite State Machine* memberi penggunanya kesempatan untuk menggunakan ulang transisi. Meskipun begitu, dalam skala besar yang kompleks, pengontrolan perilaku dengan HFSM membutuhkan banyak rancangan untuk logika-logika yang akan dipakai dan sulit dipelihara. Selain itu, penambahan atau pengurangan *state* akan mempersulit perubahan pada setiap transisinya.
2. *Decision Tree* memiliki kekurangan dalam menangani cakupan nilai dan masukan yang besar atau kurang jelas, juga berpotensi untuk menghasilkan keluaran yang terhubung satu sama lain (tidak *disjoint*). Hal ini disebabkan karena kalkulasi yang dilakukan akan menjadi sangat kompleks.

Karena itu, *Behavior Tree* merupakan alternatif yang cukup baik bila dibandingkan dengan HFSM dan *Decision Tree*. Setelah dilakukan sedikit modifikasi untuk mengoptimasi cara kerja pohon agar lebih mangkus dan stabil, *Event-Driven Behavior Tree* dapat terbentuk dari *Behavior Tree* dan menjadi solusi terbaik untuk menentukan perilaku dari NPC pada permainan.

V. IMPLEMENTASI *EVENT-DRIVEN BEHAVIOR TREE*

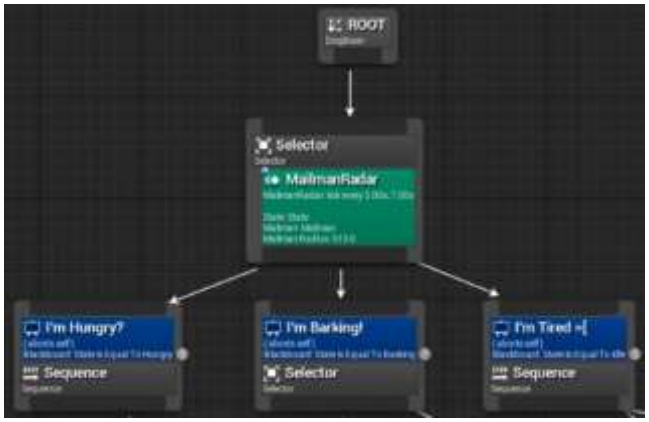
Hampir seluruh bagian dari sebuah permainan diatur dan dikendalikan oleh kecerdasan buatan. Bentuk implementasi utama *Event-Driven Behavior Tree* dalam sebuah permainan adalah untuk mengontrol pergerakan dan kelakuan NPC. Berikut ini adalah penjelasan contoh kasus mengenai penerapan pohon untuk NPC sebagai hewan dalam mode penjelajahan dalam sebuah permainan.



Gambar 12. Penerapan *Event-Driven Behavior Tree* (Keseluruhan) [8]

Bagian teratas dari pohon tersebut merupakan akar yang memiliki anak bernama *MailmanRadar*. Fungsi dari *MailmanRadar* ini adalah untuk memeriksa apakah situasi memenuhi syarat agar dapat memicu munculnya sebuah event dan membuka akses kepada keturunannya pada pohon tersebut. *MailmanRadar* juga merupakan sebuah *composite task* berbentuk *selector* yang memiliki 3 anak

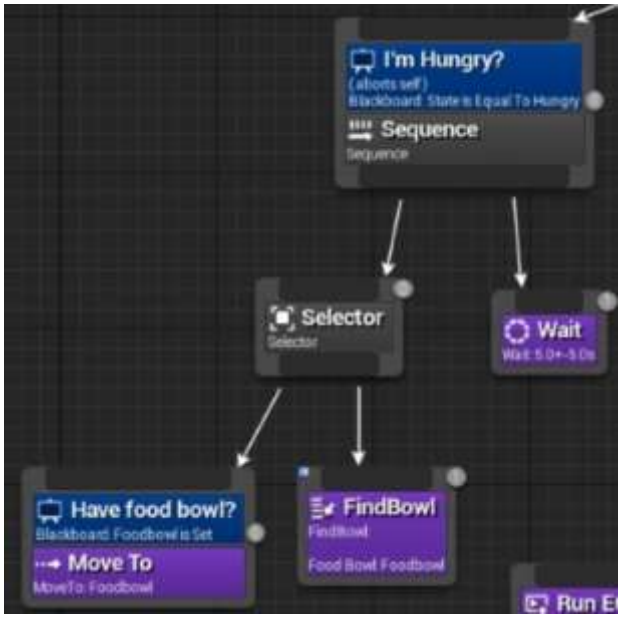
berupa *decorator*. Ketiga *decorator* tersebut adalah *I'm Hungry?*, *I'm Barking!*, dan *I'm Tired =[*. Kegunaan ketiga *decorator* tersebut adalah untuk mengaktifkan *behavior* yang sesuai dengan *event*-nya.



Gambar 13. Penerapan *Event-Driven Behavior Tree* (Bagian Atas) [8]

Setelah prasyarat dari *MailmanRadar* terpenuhi, hal pertama yang dilakukan oleh pohon adalah melanjutkan proses ke *I'm Hungry?* dan melakukan perbandingan *state* untuk mengetahui apakah ada *event* memanggil upapohon *I'm Hungry?*. Bila ternyata *I'm Hungry?* bernilai *success*, maka pemrosesan akan berlanjut ke anak dari *I'm Hungry?*. Bila tidak, pohon akan mencoba melakukan pengecekan ke saudara kandung dari *I'm Hungry?*, yaitu *I'm Barking!* dan *I'm Tired =[*.

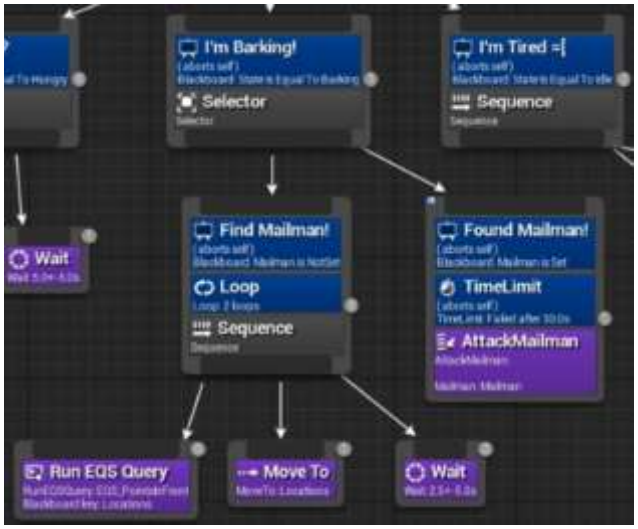
A. *I'm Hungry?*



Gambar 14. Upapohon *I'm Hungry?* [8]

Setelah upapohon *I'm Hungry?* berhasil dipicu, maka pemrosesan akan berlanjut ke anak-anak dari *I'm Hungry?*. Anak-anak tersebut memiliki *composite task* berbentuk *sequence*, sehingga upapohon anak pertama dan upapohon anak kedua akan dioperasikan secara sekuensial. Pada upapohon anak pertama, terdapat selector yang mengarah ke *decorator Have food bowl?*. Apabila *Have food bowl?* bernilai *success*, maka NPC tersebut akan bergerak mendekati *Foodbowl* dan melanjutkan pemrosesan ke *Wait*. Sebaliknya jika *Have food bowl?* bernilai *failure*, maka NPC akan melakukan aksi *FindBowl*, baru melanjutkan ke aksi *Wait*.

B. *I'm Barking!*

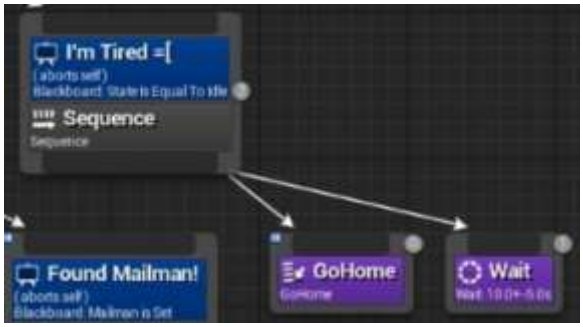


Gambar 15. Upapohon *I'm Barking!* [8]

Ketika *I'm Hungry?* bernilai *failure*, pohon akan melakukan pengecekan pada anak selanjutnya, yaitu *I'm Barking!*. Apabila *I'm Barking!* bernilai *success*, maka pemrosesan akan diteruskan ke anaknya, yang memiliki *selector* sebagai *composite task*-nya. Proses akan dilanjutkan ke upapohon anak pertama terlebih dahulu, yaitu *Find Mailman!*. *Find Mailman!* merupakan sebuah *decorator* yang berfungsi sebagai penanda mengenai lokasi *Mailman*. Apabila lokasi *Mailman* belum diketahui (pada kasus ini *Mailman* belum diset), maka pemrosesan akan dilanjutkan ke *sequence* dari aksi *Run EQS Query*, berpindah lokasi, dan menunggu. Karena terdapat *decorator loop*, maka *sequence* tersebut akan diulang sebanyak angka yang diberikan pada *decorator* (pada kasus ini angkanya adalah dua).

Apabila *Find Mailman!* sebelumnya bernilai *failure*, maka pemrosesan akan diteruskan ke upapohon anak kedua yang dijaga oleh sebuah *decorator*, yaitu *Found Mailman!*. *Found Mailman!* akan bernilai *success* jika posisi *Mailman* sudah diketahui (pada kasus ini artinya *Mailman* sudah diset). Kemudian, NPC akan melanjutkan operasi dan melakukan aksi *AttackMailman*, yang akan digagalkan secara otomatis setelah 30 detik berlalu oleh decorator *TimeLimit*.

C. *I'm Tired =[*



Gambar 16. Upapohon *I'm Tired =[* [8]

Pada saat *I'm Hungry?* dan *I'm Barking!* bernilai *failure*, pemrosesan akan dilanjutkan ke *I'm Tired =[*. Setelah mengecek *I'm Tired =[* dan bernilai *success*, maka anak dari I'm Tired =[akan dioperasikan. Lalu, NPC akan melakukan aksi dari anak *I'm Tired =[*, yakni *sequence* dari *GoHome* dan *Wait*.

VI. KESIMPULAN

Pohon dapat diimplementasikan dalam berbagai persoalan, salah satunya adalah kecerdasan buatan. Penggunaan pohon pada permainan dapat diaplikasikan untuk berbagai hal, terutama untuk mengendalikan perilaku dari NPC. Ada beberapa metode yang dapat dipakai untuk mengatur kelakuan dari NPC, yaitu *Hierarchical Finite State Machine* (HFSM), *Decision Tree*, *Behavior Tree*, dan *Event-Driven Behavior Tree*. Untuk saat ini, *Event-Driven Behavior Tree* merupakan representasi yang paling tepat karena fleksibel, mudah diintegrasikan, responsif, dan berorientasi tujuan.

VII. UCAPAN TERIMA KASIH

Saya mengucapkan terima kasih pada Tuhan Yang Maha Esa (YME) untuk setiap berkatNya dalam kehidupan saya. Saya juga ingin berterima kasih pada orang tua saya untuk setiap dukungan yang mereka beri. Tak ketinggalan, saya berterima kasih pada Dr. Ir. Rinaldi Munir, MT. dan Dra. Harlili S., M.Sc. sebagai dosen IF2120 Matematika Diskrit.

REFERENSI

[1] https://gamedevelopment.tutsplus.com/tutorials/finite-state-machines-theory-and-implementation--gamedev-11867
Waktu akses: 6 Desember 2016 pukul 21.50.
[2] http://www.cs.cmu.edu/~maxim/classes/CIS15466_Fall11/lectures/intelligenceI_cis15466.pdf
Waktu akses: 6 Desember 2016 pukul 21.53.
[3] R. Munir, "Pohon," in *Matematika Diskrit*, Edisi Ketiga Revisi Keempat, Bandung: INFORMATIKA, 2010, hlm. 457–461.
[4] https://web.stanford.edu/class/cs123/lectures/CS123_lec08_HFSM_BT.pdf
Waktu akses: 6 Desember 2016 pukul 21.56.
[5] https://github.com/libgdx/gdx-ai/wiki/Behavior-Trees
Waktu akses: 6 Desember 2016 pukul 21.57.
[6] http://lecturer.ukdw.ac.id/~mahas/dossier/gameng_04_read.pdf
Waktu akses: 6 Desember 2016 pukul 22.03.
[7] https://docs.unrealengine.com/latest/INT/Engine/AI/BehaviorTrees/HowUE4BehaviorTreesDiffer/
Waktu akses: 6 Desember 2016 pukul 22.05.
[8] https://forums.unrealengine.com/showthread.php?125060-Behaviour-Trees-Event-driven
Waktu akses: 6 Desember 2016 pukul 22.07.
[9] http://www.gameaipro.com/GameAIPro/GameAIPro_Chapter06_The_Behavior_Tree_Starter_Kit.pdf
Waktu akses: 6 Desember 2016 pukul 22.10.
[10] http://aigamedev.com/open/tutorial/asynchronous-event-driven-conditions/
Waktu akses: 6 Desember 2016 pukul 22.11.
[11] http://aigamedev.com/open/tutorial/monitoring-assumptions-polling-conditions/
Waktu akses: 6 Desember 2016 pukul 22.11.
[12] https://www.cs.hmc.edu/~pmawhorter/research/slides/behavior_trees-Mawhorter-2010.pdf
Waktu akses: 6 Desember 2016 pukul 22.12.
[13] http://leamonde.net/posts/11232015.html
Waktu akses: 6 Desember 2016 pukul 22.12.
[14] http://aigamedev.com/open/article/event-handling-purposeful-behaviors/
Waktu akses: 6 Desember 2016 pukul 22.12.
[15] http://aigamedev.com/open/article/popular-behavior-tree-design/
Waktu akses: 6 Desember 2016 pukul 22.13.
[16] http://aiandgames.com/outsmarting-the-covenant/
Waktu akses: 6 Desember 2016 pukul 22.13.
[17] http://www.gamasutra.com/blogs/JakobRasmussen/20160427/271188/Are_Behavior_Trees_a_Thing_of_the_Past.php
Waktu akses: 6 Desember 2016 pukul 22.13.
[18] http://www.gamasutra.com/blogs/ChrisSimpson/20140717/221339/Behavior_trees_for_AI_How_they_work.php
Waktu akses: 6 Desember 2016 pukul 22.14.
[19] https://gamedevdaily.io/advanced-behavior-tree-structures-4b9dc0516f92#.f20eva86a
Waktu akses: 6 Desember 2016 pukul 22.15.
[20] http://aigamedev.com/open/article/hierarchical-or-nested-fsm/
Waktu akses: 6 Desember 2016 pukul 22.15.
[21] http://web.cs.wpi.edu/~rich/courses/imgd4000-d10/lectures/B-AI.pdf
Waktu akses: 6 Desember 2016 pukul 22.16.
[22] https://www.cs.utexas.edu/~fussell/courses/cs378/lectures/cs378-13.pdf
Waktu akses: 6 Desember 2016 pukul 22.16.

PERNYATAAN

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 6 Desember 2016

Holy Lovenia - 13515113