BAB 3
ANALISIS DAN KEBUTUHAN ALGORITMA

Tahap analisis dan kebutuhan algoritma diperlukan sebagai tahapan sistematis terhadap penerapan metode algoritma yang digunakan dalam sistem yang dibangun. Tahap pertama dari analisis adalah melakukan analisis masalah, analisis *game*, analisis masukan, analisis kebutuhan fungsional dan analisis non-fungsional.

3.1. Analisis Masalah

Analisis masalah penjabaran dari masalah yang ada, yaitu dengan pertumbuhan jumlah pengguna jaringan atau internet diseluruh dunia yang kian hari kian bertambah dengan pesat. Hal tersebut didorong oleh tingginya tingkat kebutuhan akan komunikasi melalui jaringan internet dan ditambah dengan pergeseran gaya hidup kebanyakan orang yang pada setiap harinya tidak lepas dari penggunaan jaringan atau internet.

Faktanya, ketersediaan jumlah IPv4 sendiri terbatas yaitu setara dengan 2 pangkat 32 atau sebanyak 4.294.967.296 atau jika kita bulatkan ada sekitar 4.3 miliar host yang dapat diberikan pengalamatan oleh IPv4 [11]. sehingga kebutuhan akan hadirnya IPv6 sudah sangat perlu diberikan pada pembangunan infrastruktur yang baru. Maka dengan menerapkan konsep IPv6 pada permainan fire shooter multiplayer 2D dengan catatan baik infrastruktur yang lama maupun yang baru tetap dapat saling berkomunikasi. Dengan demikian proses peralihan protokol jaringan dapat diimplemetasikan secara berangsur-angsur yang pada akhirnya seluruh infrastruktur yang ada dapat menggunakan protokol jaringan terbaru yaitu IPv6.

pada penelitian yang dilakukan, masalah yang diangkat dibuat dalam bentuk rancangan server game menggunakan protokol jaringan IPv6 dengan cara mengimplementasikan kedua protokol jaringan IPv4 dan IPv6 yang dapat digunakan secara bersama-sama.

3.1.1. Mekanisme Transisi IPv4 Ke IPv6

Teknologi Internet saat ini menggunakan protokol IPv4. Kenyataannya bahwa infrastruktur digunakan sekarang sangat menyulitkan transisi protokol dari IPv4 ke IPv6 sekaligus. Sangat tidak relistis untuk mengharuskan semua node menggunakan IPv6 pada suatu saat yang ditentukan, misalkan tengah malam pada tanggal tertentu [11]. Yang

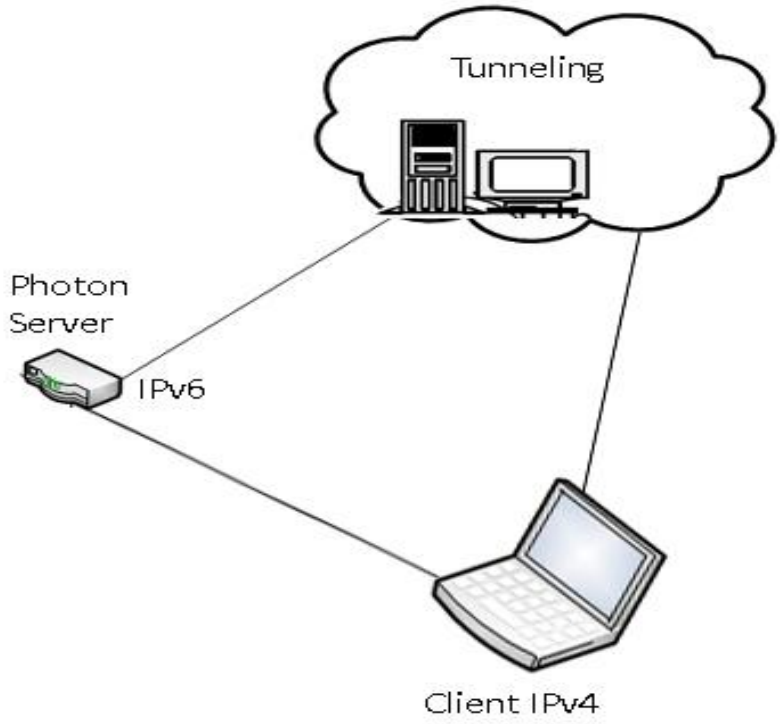
dibutuhkan adalah sebuah mekanisme transisi. Mekanisme yang dibahas di sini adalah kondisi saat mesin IPv6 harus berhubungan dengan mesin IPv6 dan menggunakan infrastruktur *routing* IPv4. Cara termudah proses transisi IPv6 adalah dengan menyediakan implementasi IPv4 dan IPv6 sekaligus pada satu mesin, sehingga dapat berhubungan baik dengan mesin IPv4 maupun IPv6. Mesin dengan kemampuan demikian disebut mesin IPv6 / IPv4.

3.1.2. Tunneling Otomatis (*Autimatic Tunneling*)

Tunneling demikian disebut tunneling secara otomatis. Mekanisme *Automatic Tunneling* berfungsi melewatkan paket IPv6 melalui jaringan IPv4 yang telah ada, tanpa merubah infrastruktur jaringan IPv4. Mekanisme Automatic Tunneling mempunyai prinsip kerja mengenkapsulasi paket IPv6 dengan header IPv4, kemudian paket tersebut langsung dikirimkan ke jaringan IPv4 Untuk menghubungkan dua network IPv6 agar bisa melakukan komunikasi melalui infrastruktur jaringan IPv4, maka penulis memilih *automatic tunneling* sebagai salah satu jalan keluar. Salah satu pertimbangannya adalah untuk melihat bagaimana dalam sebuah lingkungan IPv4 bisa dijalankan IPv6. Fungsi *automatic tunneling* adalah melewatkan paket IPv6 pada jaringan IPv4. *Automatic tunneling* mempunyai prinsip kerja mengenkapsulasi paket IPv6 dengan header IPv4, kemudian paket tersebut dikirimkan ke tujuannya melalui jaringan IPv4.

3.1.3. Implementasi *Automatic Tunelling*

Untuk mempermudah pemahaman tentang konfigurasi dan implementasi mekanisme automatic tunneling dapat dilihat pada gambar berikut



Gambar 3. 1 Proses *Tunneling Automatic*

pada gambar 3.1, digambarkan terdapat *client* dengan menggunakan ip versi 4 yang kemudian melewati proses tunneling otomatis untuk bisa berkomunikasi dengan photon server atau locallhost yang digunakan pada *game fire shooter* dengan menggunakan *default server* IPv6, setelah proses tunneling kemudian dilanjutkan melakukan komunikasi data dengan photon server yang menggunakan ip versi 6 dan di kembalikan kembali kepada client yang menggunakan ip versi 4 dengan tanpa merubah infrastruktur jaringan IPv4.

3.2. Analisis Game

Pada bagian analisa *game* ini akan menjelaskan analisis permainan dan *gameplay* pada permainan *fire shooter 2D*

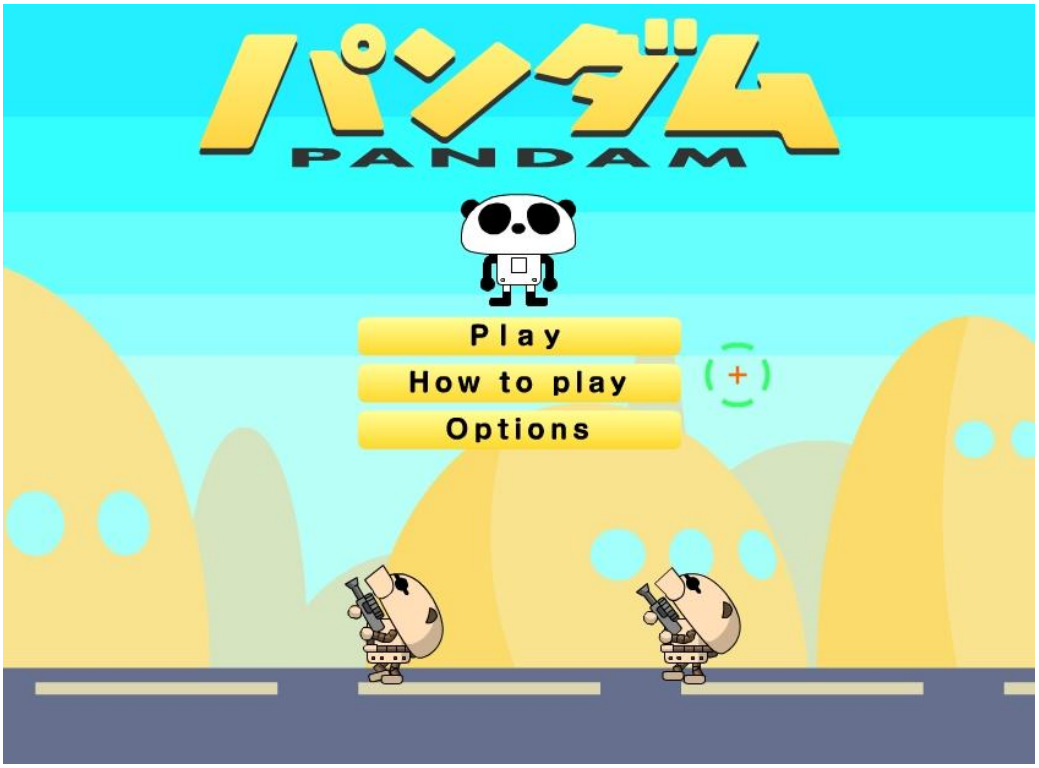
3.2.1. Analisis Permainan Sejenis

Analisis *game* sejenis dilakukan untuk memberikan gambaran sistem agar lebih mudah dalam pembangunan dan sebagai pembanding untuk kesempurnaan *game* yang akan dicapai.

3.2.2. *Game Panda Wars*

Panda wars adalah suatu permainan *game online* dan berbasis flash, permainan ini ber genre FPS *game* atau biasa disebut *First Person Shooter* dan hanya bisa dimainkan oleh satu pemain,[3] skenario pada *game* ini pemain akan mendapati dirinya sebagai robot panda dalam misi bertahan dan menyelamatkan diri serta harus menghancurkan atau mengalahkan robot panda lainnya.

Musuh *panda wars* pada *game* ini adalah robot panda pengkhianat, yang dikenal sebagai *cyborg panda*, pada alur permainan kita hanya menunggu *cyborg* tersebut muncul kemudian menembaknya sebelum *cyborg* tersebut menembak pemain.



Gambar 3. 2 Panda *Wars* [3]

Tabel 3. 1 Game Panda Wars Overview

Pemain	*Single Player*
Desain Grafis	2D
Karakter Utama	*Panda*
Kekurangan	- Musuh dalam *game panda wars* hanya bisa berjalan secara *random* hanya mampu bergerak maju atau mundur dan tidak dapat mengejar pemain.

	- Tidak adanya parameter informasi mengenai berapa lama lagi pemain bisa bertahan menghadapi musuh. - Hanya bisa dimainkan oleh satu pemain. - Tidak dimainkan secara *Online*

3.2.3. *Gameplay Fire Shooter 2D*

Menurut kamus oxford, *gameplay* adalah sebuah fitur dari permainan komputer, seperti *plot* dan cara permainan dimainkan. [2] Tugas utama pemain dalam memainkan g*ame Fire Shooter Multiplayer 2D* ini pemain akan menjadi seorang prajurit yang harus berjuang membasmi semua makhluk *alien* yang telah datang masuk ke Bumi. Ada dua jenis musuh *alien* yang bergerak menyerang player dengan jarak dekat, musuh pertama *alien* yang menyerang pemain dengan menggunakan *UFO* dan musuh kedua *alien* yang berjalan seperti ulat untuk menyerang pemain, pemain harus bisa memusnahkan semua musuh yang datang sampai batas waktu yang telah ditentukan tujuan misi untuk mendapatkan score terbanyak dengan pemain yang lain.

Di dalam *game* ini *player* (pemain) dituntut untuk menyelesaikan misi yang ada dengan peralatan dan senjata yang telah diberikan. Didalam *game* juga tersedia menu *health bar* sebagai informasi *health* (darah) *player*, *health bar* akan berkurang jika *player* terkena kontak langsung oleh npc musuh dan jika *health* bar habis, *player* akan mati dan otomatis misi akan gagal. Akan tetapi di sekitar lokasi *game* juga akan dapat ditemukan *health pack* yang berguna untuk menambah *healt bar player*.

3.3.1. Metode Algoritma

Agar permainan *game* menjadi lebih menarik dapat dilakukan dengan meningkatkan kualitas *gameplay* bagi pengguna yang memainkannya. AI merupakan faktor yang penting dalam meningkatkan *gameplay* pada semua jenis permainan *game*, termasuk pada *game* yang akan dibangun. *Pathfinding* dalam *game* ini digunakan diterapkan pada objek musuh dalam melakukan pencarian karakter pemain di arena permainan.

Dengan *pathfinding*, objek musuh saat menuju ke karakter pemain menjadi lebih agresif. Algoritma A* (dibaca “A bintang”/”A star) adalah algoritma pencarian graft/pohon yang mencari jalur dari satu titik akhir yang telah ditentukan. Algoritma A* menggunakan pendekatan heuristik h(x) yang memberikan peringkat ke tiap-tiap titik x dengan cara memperkirakan rute terbaik yang dapat dilalui dari titik tersebut. di dalam

game akan diterapkan pada NPC *(Non Playable Character)* sebagai penentuan jalur terpendek atau pathfinding menuju target. Cara kerja algoritma A* di dalam *game Fire Shooter 2D* lebih jelasnya sebagai berikut.

(0.0)	(0.1)	(0.2)	(0.3)	(0.4)	(0.5)	(0.6)
(1.0)	(1.1) A	(1.2)	(1.3)	(1.4)	(1.5) B	(1.6)
(2.0)	(2.1)	(2.2)	(2.3)	(2.4)	(2.5)	(2.6)
F (3.0) G H	(3.1)	(3.2)	(3.3)	(3.4)	(3.5)	(3.6)

Gambar 3. 3 Ilustrasi A*

pada gambar ilustrasi di atas diketahui bahwa wilayah pencarian dibagi menjadi beberapa bagian kotak yang disebut dengan *node, node* A yang berwarna merah (1.1) merupakan *node* awal dan *node* B yang berwarna biru (1.5) merupakan *node* tujuan, sedangkan *node* yang berwarna hijau (0.3), (0.4), (1.3), (2.3), (2.4) merupakan dinding penghalang atau disebut *collision*. Diasumsikan pencarian jalan terbaik dimulai dari *node* A menuju node B dengan *node* hijau sebagai *collision*nya.

Sedangkan untuk nilai F yang akan dihitung untuk setiap *node* terletak disudut kiri atas kotak, nilai G terletak di sudut kiri bawah kotak dan nilai H terletak disudut kanan bawah kotak. Perhitungan nilai F didapat dari persamaan F= G+H. Sedangkan untuk nilai G digunakan nilai 10 untuk gerakan *vertical* atau *horizontal*, angka 10 digunakan hanya untuk memudahkan perhitungan. Sedangkan untuk gerakan *diagonal* maka akan dikalikan dengan √2 yang berarti 10x√2 = 14,14. Untuk memudahkan perhitungan angka 14,14 dibulatkan menjadi 14.

Untuk perhitungan nilai H digunakan fungsi *heuristic*, metode yang digunakan di dalam contoh ini adalah metode *Manhattan* dimana perhitungan jumlah *node* yang bergerak secara *vertical* dan *horizontal* menuju tujuannya serta mengabaikan penghalang atau *collision*, yang kemudian nilainya dikalikan dengan 10. Atau dirumuskan dengan:

H = 10*(abs(*current*X-*target*X) + abs(*current*Y-*target*Y))

Inisialisasi nilai graph dan nilai matrix untuk menentukan posisi koordinat dari *node-node* yang digunakan. Dan inisialisasi awal untuk *open list* dan *closed list* berjumlah kosong atau *null*.

Tahap berikutnya adalah melakukan perhitungan untuk menemukan jalan dengan cara menghitung *node-node* yang berdekatan dari *node* A sampai *node* B. Langkah-langkahnya adalah sebagai berikut:

1. Tahap pertama adalah, dimulai dari *node* A dan di tambahkan ke dalam *open list*. Sedangkan *open list* itu sendiri merupakan *list* untuk *node-node* yang harus diperiksa.

2. Tahap kedua ambil *node* A (1,1) sebagai *current node*, dan *set current node = best node*. Atau dalam artian *node* A merupakan bagian dari jalan terpendek yang sudah didapatkan

3. Langkah berikutnya, periksa apakah *current node* sama dengan *node* B, atau *current node* yang sudah didapatkan merupakan *node* tujuan. Jika iya maka jalan sudah ditemukan, jika tidak maka hapus *node* A dari *open list*, kemudian tambahkan ke dalam *closed list.*

4. Periksa semua *node* yang berdekatan dengan *node* A apakah *walkable* atau tidak, abaikan *node* dengan *collision* atau penghalang lalu tambahkan semua *node* tersebut ke dalam *open list*. Dan untuk setiap *node* yang baru ditambahkan, simpan *node* A sebagai *"parent"* yang nantinya akan digunakan untuk menelusuri jalan.

Untuk lebih jelasnya proses perhitungan dilakukan per-iterasi dan dijelaskan sebagai berikut.

74 (0.0) 14 ↘ 60	60 (0.1) 10 ↓ 50	54 (0.2) 14 ↙ 50	(0.3)	(0.4)	(0.5)	(0.6)
60 (1.0) 10 → 50	40 (1.1) 0 40	40 (1.2) 10 ← 30	(1.3)	(1.4)	(1.5)	(1.6)
74 (2.0) 14 ↗ 60	60 (2.1) 10 ↑ 50	54 (2.2) 14 ↖ 40	(2.3)	(2.4)	(2.5)	(2.6)
F (3.0) G H	(3.1)	(3.2)	(3.3)	(3.4)	(3.5)	(3.6)

Gambar 3. 4 Ilustrasi Iterasi Pertama

Pada gambar ilustrasi iterasi pertama di atas, dapat dilihat bahwa *node* (1,1) dimasukkan ke dalam *open list*, dan selanjutnya memeriksa *neighbor node* atau *node* yang berdekatan dengan *node* (1,1) yaitu *node* {(0,0), (0,1), (0,2), (1,0), (1,2), (2,0), (2,1), (2,2)} dan dimasukkan ke dalam *open list.* Kemudian hapus *node* (1,1) dari *open list* dan masukkan ke dalam *closed list*. Ambil *node* dengan nilai F paling kecil sebagai *current node* berikutnya dalam perhitungan diatas didapat *node* (1,2) memiliki nilai F paling rendah dan diberikan tanda dengan kotak berwarna abu-abu.

Node yang menjadi *current node* atau *node* (1,2) diberikan nilai H = 30 yang didapatkan dari perhitungan 3 langkah atau 3 *node* menuju node B dan dikalikan 10, 3*10 = 30.

Pada perhitungan iterasi pertama menghasilkan *node-node* di dalam *open list* dan *closed list* yang akan dijelaskan dalam tabel 3.2 berikut.

Tabel 3. 2 Perhitungan Iterasi Pertama

Open List	*Parent*	*Closed List*	*Current ode*
(0,0)	(1,1)	(1,1)	(1,2)
(0,1)		(1,1)	
(0,2)		(1,1)	
(1,0)		(1,1)	
(1,2)		(1,1)	
(2,0)		(1,1)	
(2,1)		(1,1)	
(2,2)		(1,1)	

Lakukan proses ini berulang kali, dimulai dari *open list* dan menemukan *node* dengan nilai F paling rendah. Dengan melakukan langkah berikut.

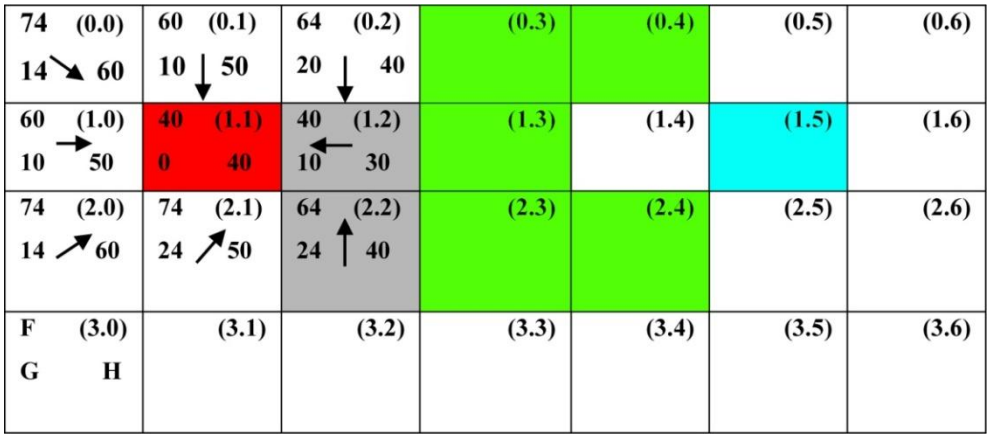
1. Hapus *current node* tersebut dari *open list* dan masukkan ke dalam *closed list.*

2. Periksa semua *node* yang berdekatan dengan *current node*, lalu abaikan *node* yang *unwalkable* atau dalam contoh ini adalah *node* penghalang. Jika pada *open list* tidak ada, maka tambahkan ke dalam *open list*. Dan tambahkan *"parent"* untuk *node* tersebut.

3. Apabila *node* sudah ada pada *open list*, periksa apakah ini jalan yang lebih baik dari *node* awal, dengan kata lain periksa apakah nilai G-nya lebih rendah apabila

menggunakan *node* ini, jika lebih rendah maka hitung ulang nilai F dan G serta merubah arah pointer, jika tidak maka abaikan saja.

Selanjutnya perhitungan pada iterasi kedua dengan *current node* berada pada node (1,2).



Gambar 3. 5 Ilustrasi Iterasi Kedua

Periksa *node* yang berdekatan dengan *current node*, disebelah kanan, kanan bawah dan kanan atas dari *node* ini adalah *node* penghalang atau *unwalkable* (gambar 3.5) berarti diabaikan, dan disebelah kanan adalah *node* (1,1) yang sudah berada di dalam *closed list,* yang berarti juga diabaikan.

Sedangkan *node* lainnya yang berdekatan sudah berada di dalam *open list,* jadi perlu diperiksa apakah *node* tersebut mempunyai jalan yang lebih baik atau tidak dengan menggunakan nilai G sebagai acuannya. Setelah diperiksa *node-node* tersebut ternyata tidak ada jalan yang lebih baik, maka tidak ada yang perlu diubah dan proses untuk *current node* ini selesai.

Node (0,2) dan *node* (2,2) memiliki nilai F yang sama, dapat dilihat pada gambar 3.5 diatas, maka sistem akan memilih *node* (2,2) karena yang terakhir masuk ke dalam open list. Yang berarti *current node* selanjutnya adalah *node* (2,2).

Iterasi kedua menghasilkan *node-node* di dalam *open list* dan *closed list* dijelaskan dalam tabel 3.3.

Tabel 3. 3 Perhitungan Iterasi Kedua

Open List	*Parent*	*Closed List*	*Current ode*
(0,0)	(1,1)	(1,1)	(2,2)
(0,1)	(1,1)	(1,2)	

(0,2)	(1,2)		
(1,0)	(1,1)		
(1,2)	(1,1)		
(2,0)	(1,2)		
(2,1)	(1,2)		
(2,2)	(1,2)		

Selanjutnya perhitungan pada iterasi ketiga dengan *current node* berada pada *node* (3,2).

74 (0.0) 14 ↘ 60	60 (0.1) 10 ↓ 50	64 (0.2) 20 ↓ 40	(0.3)	(0.4)	(0.5)	(0.6)
60 (1.0) 10 → 50	40 (1.1) 0 40	40 (1.2) 10 ← 30	(1.3)	(1.4)	(1.5)	(1.6)
74 (2.0) 14 ↗ 60	84 (2.1) 34 → 50	64 (2.2) 24 ↑ 40	(2.3)	(2.4)	(2.5)	(2.6)
F (3.0) G H	98 (3.1) 38 ↗ 60	84 (3.2) 34 ↑ 50	(3.3)	(3.4)	(3.5)	(3.6)

Gambar 3. 6 Ilustrasi Iterasi Ketiga

Pada gambar 3.6 di atas *node* (3,3) tidak dihitung atau tidak masuk ke dalam *open list*, karena karakter tidak akan bisa bergerak secara *diagonal* menembus dinding atau *collision* sehingga *node* (3,3) otomatis menjadi *unwalkable*.

Kemudian ulang kembali seperti langkah sebelumnya, hapus dari *open list* dan tambahkan ke dalam *closed list*, periksa semua *node* yang berdekatan, masukkan ke dalam *open list* dan tambahkan parentnya, dan periksa lagi jalan terbaik dan *parent*nya. Pada *node* (2,1) *parent*nya mengalami perubahan karena setelah diperiksa nilai G yang dihasilkan dari *current node* lebih kecil, maka nilai F nya juga dirubah.

Nilai F pada *node* (2,1) juga sama dengan *node* (3,2), sama seperti contoh kasus sebelumnya maka *node* (3,2) dipilih sebagai *current node.*

Iterasi ketiga menghasilkan *node-node* di dalam *open list* dan *closed list* dijelaskan dalam tabel 3.4.

Tabel 3. 4 Perhitungan Iterasi Ketiga

Open List	*Parent*	*Closed List*	*Current ode*
(0,0)	(1,1)	(1,1)	(3,2)
(0,1)	(1,1)	(1,2)	
(0,2)	(1,2)	(2,2)	
(1,0)	(1,1)		
(1,2)	(1,1)		
(2,0)	(1,1)		
(2,1)	(2,2)		
(3,1)	(2,2)		
(3,2)	(2,2)		

Selanjutnya perhitungan pada iterasi keempat dengan *current node* berada pada *node* (3,2).

74 (0.0) 14 ↘ 60	60 (0.1) 10 ↓ 50	64 (0.2) 20 ↓ 40	(0.3)	(0.4)	(0.5)	(0.6)
60 (1.0) 10 → 50	40 (1.1) 0 40	40 (1.2) 10 ← 30	(1.3)	(1.4)	(1.5)	(1.6)
74 (2.0) 14 ↗ 60	94 (2.1) 44 ↘ 50	64 (2.2) 24 ↑ 40	(2.3)	(2.4)	(2.5)	(2.6)
F (3.0) G H	104 (3.1) 44 → 60	84 (3.2) 34 ↑ 50	84 (3.3) 44 ↖ 40	(3.4)	(3.5)	(3.6)

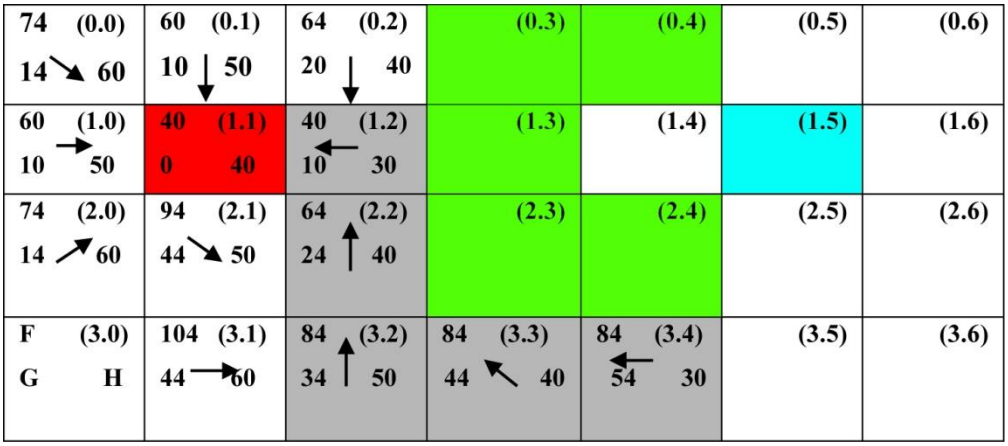
Gambar 3. 7 Ilustrasi Iterasi Keempat

Setelah melakukan langkah yang sama seperti sebelumnya pada iterasi keempat menghasilkan *node-node* di dalam *open list* dan *closed list* yang akan dijelaskan dalam tabel 3.5 berikut

Tabel 3. 5 Perhitungan Iterasi Kelima

Open List	*Parent*	*Closed List*	*Current ode*
(0,0)	(1,1)	(1,1)	(3,3)
(0,1)	(1,1)	(1,2)	
(0,2)	(1,2)	(2,2)	
(1,0)	(1,1)	(3,2)	
(1,2)	(1,1)		
(2,0)	(1,1)		
(2,1)	(3,2)		
(3,1)	(3,2)		
(3,3)	(3,2)		

Berikutnyan perhitungan pada iterasi kelima dengan *current node* berada pada *node* (3,3).



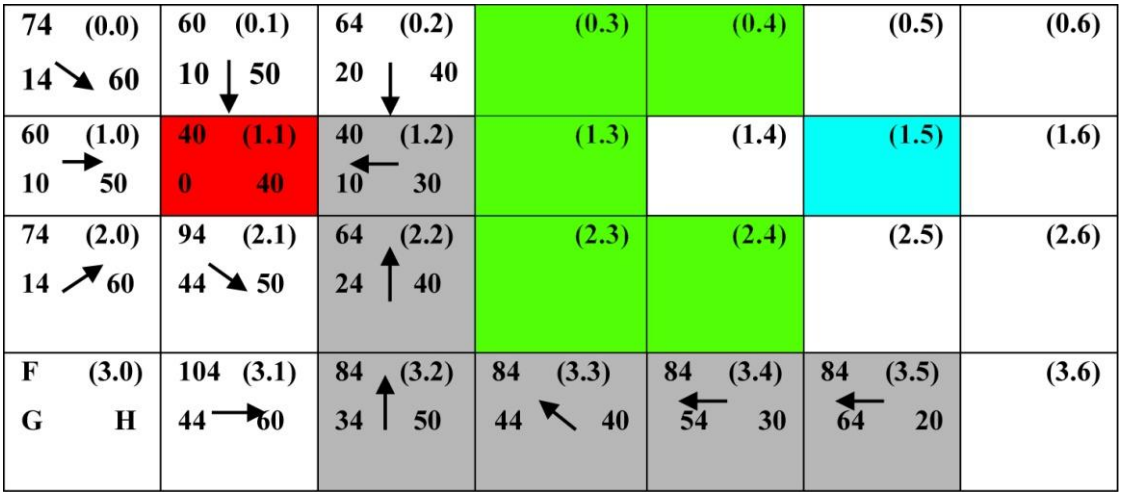
Gambar 3. 8 Ilustrasi Iterasi Kelima

Iterasi kelima menghasilkan *node-node* di dalam *open list* dan *closed list* yang akan dijelaskan dalam tabel 3.6 berikut ini

Tabel 3. 6 Perhitungan Iterasi Kelima

Open List	*Parent*	*Closed List*	*Current ode*
(0,0)	(1,1)	(1,1)	(3,4)
(0,1)	(1,1)	(1,2)	
(0,2)	(1,2)	(2,2)	
(1,0)	(1,1)	(3,2)	
(1,2)	(1,1)	(3,3)	
(2,0)	(1,1)		
(2,1)	(3,2)		
(3,1)	(3,2)		
(3,4)	(3,3)		

Selanjutnya perhitungan pada iterasi keenam dengan *current node* berada pada *node* (3,4).



Gambar 3. 9 Ilustrasi Iterasi Keenam

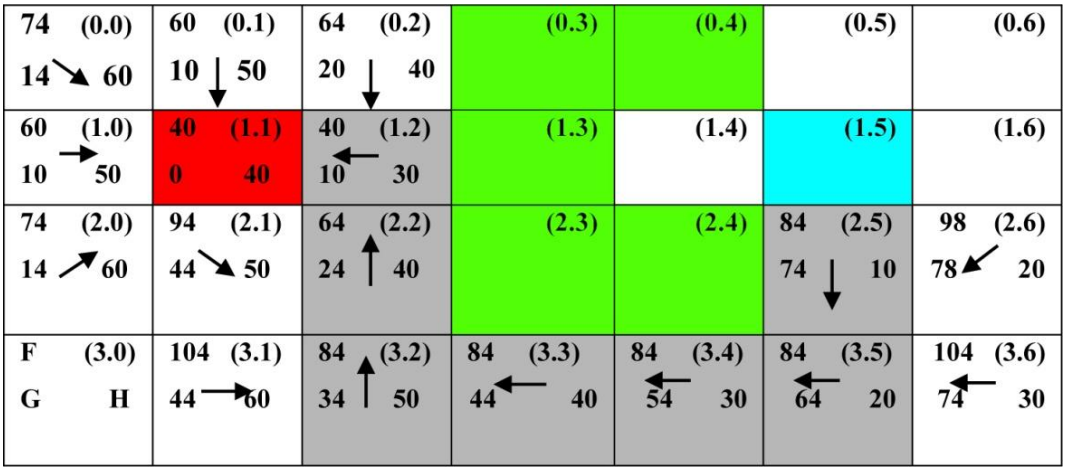
Node (2,5) menjadi *unwalkable* karena karakter tidak bisa langsung bergerak ke *node* tersebut, sehingga tidak masuk ke dalam *open list.*

Iterasi keenam menghasilkan *node-node* di dalam *open list* dan *closed list* dijelaskan dalam tabel 3.7.

Tabel 3. 7 Perhitungan Iterasi Keenam

Open List	*Parent*	*Closed List*	*Current ode*
(0,0)	(1,1)	(1,1)	(3,5)
(0,1)	(1,1)	(1,2)	
(0,2)	(1,2)	(2,2)	
(1,0)	(1,1)	(3,2)	
(1,2)	(1,1)	(3,3)	
(2,0)	(1,1)	(3,4)	
(2,1)	(3,2)		
(3,1)	(3,2)		
(3,5)	(3,4)		

Perhitungan pada iterasi ketujuh dengan *current node* berada pada *node* (3,5).



Gambar 3. 10 Ilustrasi Iterasi Ketujuh

Iterasi ketujuh menghasilkan *node-node* di dalam *open list* dan *closed list* dijelaskan dalam tabel 3.8.

Tabel 3. 8 Perhitungan Iterasi Ketujuh

Open List	*Parent*	*Closed List*	*Current ode*
(0,0)	(1,1)	(1,1)	(2,5)
(0,1)	(1,1)	(1,2)	
(0,2)	(1,2)	(2,2)	
(1,0)	(1,1)	(3,2)	
(1,2)	(1,1)	(3,3)	
(2,0)	(1,1)	(3,4)	
(2,1)	(3,2)	(3,5)	
(3,1)	(3,2)		
(3,5)	(3,4)		
(2,6)	(3,5)		
(3,6)	(3,5)		
(2,5)	(3,5)		

Pada iterasi kedelapan sudah dapat dipastikan target *point* masuk ke dalam *closed list* (gambar 3.10) maka proses perhitungan selesai.

Penentuan jalan terbaik dengan melakukan pemeriksaan dimulai dari target *point* atau *node* (1,5) dengan mengikuti arah *pointer* yang berarti akan membawanya kembali ke *node* awal, maka akan didapatkan jalan terbaiknya.

3.3. Analisis Kebutuhan Non-Fungsional

Analisis non fungsional merupakan analisis yang dibutuhkan untuk menentukan spesifikasi kebutuhan sistem. Spesifikasi ini juga meliputi elemen atau komponen-komponen apa saja yang akan dibutuhkan sistem yang akan dibangun, sampai sistem tersebut diimplementasikan. Analisis kebutuhan ini juga menentukan spesifikasi masukan yang diperlukan sistem, keluaran yang akan dihasilkan sistem dan proses yang dibutuhkan untuk mengolah masukan sehingga menghasilkan suatu keluaran yang diinginkan.

Analisis kebutuhan sistem non fungsional ini menjelaskan analisis kebutuhan perangkat lunak, analisis kebutuhan perangkat keras dan analisis pengguna.

3.3.2. Analisis Kebutuhan Perangkat Keras

Perangkat keras atau hardware merupakan salah satu hal yang penting karena tanpa hardware yang memenuhi syarat, program yang akan dibuat tidak akan dapat berjalan. Berikut spesifikasi standar perangkat keras yang dapat dipergunakan untuk membangun aplikasi game ini dan spesifikasi perangkat keras yang dapat dipergunakan untuk menjalankan alplikasi game ini yaitu :

Spesifikasi standar perangkat keras untuk pembangunan aplikasi game tersebut adalah :

Tabel 3. 9 Spesifikasi Perangakat Keras Pembangun

Nama Perangkat	Spesifikasi
Prosesor	Intel ® Pentium ® P6200 2.13GHz
RAM	3.00 GB
VGA	1275 MB
Hard disk	320 GB HDD
Display	Resolusi 1366 x 768
Keyboard	*Standard*
Mouse/Touch pad	*Standard*

3.3.3. Analisis Kebutuhan Perangkat Lunak

Dari hasil pengamatan perangkat lunak, terdapat beberapa hal penting yang harus diperhatikan agar sistem yang dirancang dapat berjalan dengan baik dan sesuai dengan yang diharapkan. Pengguna diharapkan dapat memenuhi kebutuhan perangkat lunak yang telah direkomendasikan agar aplikasi dapat berjalan dengan baik.

Perangkat lunak atau *software* merupakan hal yang terpenting dalam mendukung kinerja sebuah sistem. Perangkat lunak digunakan dalam sebuah sistem merupakan perintah-perintah yang diberikan kepada perangkat keras agar dapat salung berinteraksi diantara keduanya.

Perangkat lunak yang dibutuhkan untuk membangun aplikasi *game fire shooter* ini adalah sebagai berikut :

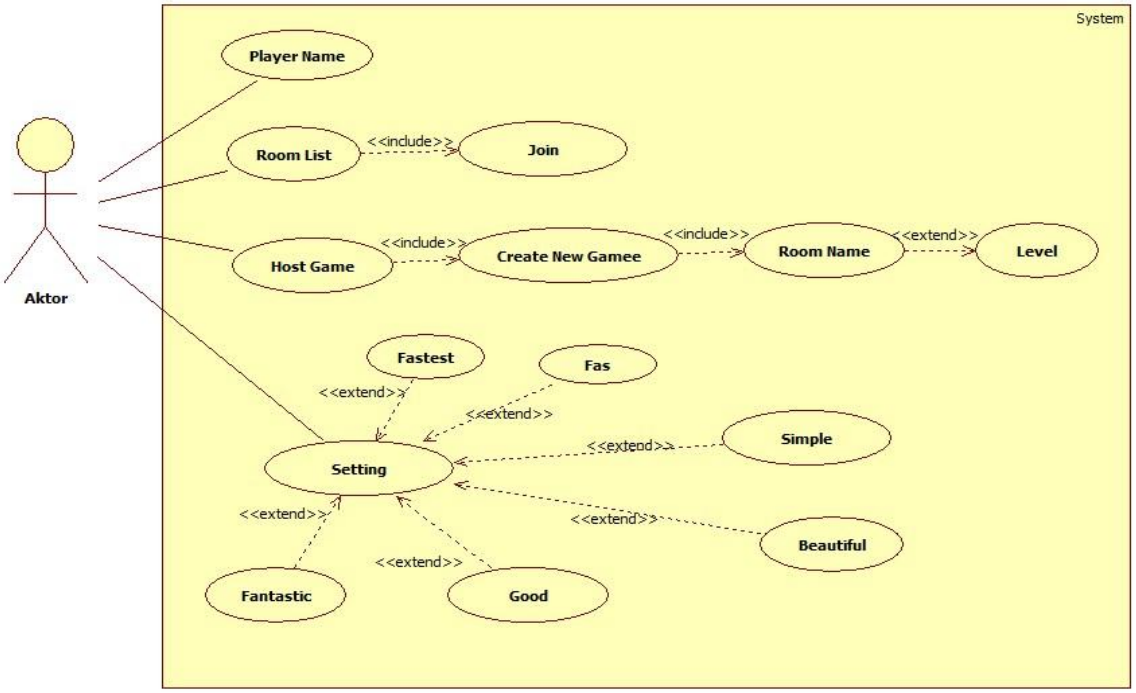
1. *System* operasi *Windows 7*
2. *Unity 3D*
3. *Adobe Photoshop*

3.4. Analisis Kebutuhan Fungsional

Analisis kebutuhan fungsional merupakan tindak lanjut dari analisis sistem. Tujuannya adalah untuk memberikan gambaran mengenai sistem yang akan dibuat. Tahap awal yang dilakukan dalam melakukan analisis berorientasi objek mengunakan UML (Unified Modeling Language). Dimana tahap-tahap perancangan yang dilakukan dalam membangun aplikasi *game fire shooter* antara lain *Use case Diagram, Class Diagram, Sequence Diagram, dan Activity Diagram.*

3.4.1. Use Case Diagram

Usecase atau diagram *Usecase* merupakan pemodelan untuk kelakuan *(behavior)* sistem informasi yang akan dibuat. *Usecase* mendeskripsikan sebuah interaksi antara satu atau lebih aktor dengan sistem yang akan dibuat dapat dilihat pada gambar berikut.



Gambar 3. 11 *Use Case Diagram*

3.4.1.1. *Actor definition*

Actor Definition berfungsi untuk menjelaskan Actor yang terdapat pada use case diagram. Actor Definition diterangkan pada Tabel 3.10

Tabel 3. 10 Actor Definition

No	*Actor*	Deskripsi
1	*User*/Pegguna	Orang yang memainkan permainan

3.4.1.2. *Use Case Definition*

Use case Definiton berfungsi untuk menjelaskan fungsi use case yang terdapat pada use case diagram. Use case definition diterangkan pada Tabel 3.10 di bawah ini

Tabel 3. 11 Use Case Definition

No	*Use Case*	Deskripsi
1	Room List	Proses untuk menampilkan tempat dan daftar ruangan atau *server* pada permainan yang telah tersedia

2	Join	Proses untuk masuk dan bergabung kedalam *Room List* yang tersedia
3	Host Game	Proses untuk membuat ruangan baru atau membuat *server* baru
4	Level	Proses untuk memilih tempat pada permainan yang akan dimainkan
5	Create New Game	Proses untuk melakukan memulainya permainan
6	Player Name	Proses untuk mengisi nama *player* (pemain)
7	Room Name	Fungsional untuk mengisi nama pada Room yang akan di bikin
8	List Host	Fungsional untuk menampilkan detail ruangan yang tersedia
9	Room Name	Fungsional untuk menampilkan nama ruangan pada List Host
10	Map	Fungsional untuk menampilkan level berapa yang ada pada List Host
11	Players	Fungsional untuk menampilkan jumlah pemain yang ada pada List Host

3.5. Skenario Use Case

Skenario Use Case berisi tentang alur kejadian yang didalamnya mendeskripsikan urutan langkah-langkah dalam proses yang dilakukan aktor, berdasarkan use case diagram. Skenario use case untuk permainan *Fire Shooter 2D* dijelaskan sebagai berikut.

A. Skenario use case menampilkan *Room List*

Use case skenario ini menjelaskan interaksi aktor, dengan *use case* menampilkan *Room List* yang akan dijelaskan pada tabel berikut.

Tabel 3. 12 Skenario Use Case Menampilkan Room List

Identifikasi	
Nomor	1
Nama	Menampilkan *Room List*
Tujuan	Menampilkan tampilan *Room List* dan ketersediaan ruangan dalam permainan
Deskripsi	Proses untuk menampilkan ruangan utama dan memulai permainan

Aktor	*User /* Pengguna
Skenario Utama	
Kondisi Awal	Program *Fire Shooter 2D* dibuka
Aksi Aktor	**Reaksi Sistem**
1. Membuka aplikasi *Fire Shooter 2D*	2. Menampilkan *Player Name Room List, Host Game, dan Setting*
3. Membuka *Room List*	4. Menampilkan ruangan atau *server* yang tersedia
Kondisi Akhir	Aplikasi menampilkan tampilan *Room List* dan informasi *server* yang tersedia.

B. Skenario use case menampilkan *Join*

Use case skenario ini menjelaskan interaksi aktor, dengan *use case* menampilkan *Join* ketika pengguna telah masuk *Room List* yang akan dijelaskan pada tabel berikut.

Tabel 3. 13 Skenario Use Case Menampilkan Join

Identifikasi	
Nomor	2
Nama	Menampilkan *Join*
Tujuan	Menampilkan tampilan *Room List* dan ketersediaan ruangan dalam permainan kemudian *join*
Deskripsi	Proses untuk masuk dan bergabung kedalam *Room List* yang tersedia
Aktor	*User /* Pengguna
Skenario Utama	
Kondisi Awal	Program *Fire Shooter 2D* sedang berjalan
Aksi Aktor	**Reaksi Sistem**
1. Membuka *Room List*	2. Menampilkan ruangan atau *server* yang tersedia
3. Menentukan *Room List* yang dipilih dan *join*	4. Masuk pada *server* yang dipilih
5. Klik *Deplay*	

Kondisi Akhir	Pengguna memulai permainan

C. Skenario Use Case Menampilkan *Host Game*

Use case skenario ini menjelaskan interaksi aktor, dengan *use case* menampilkan *Host Game* yang akan dijelaskan pada tabel berikut.

Tabel 3. 14 Skenario Use Case Menampilkan Host Game

Identifikasi	
Nomor	3
Nama	Menampilkan *Host Game*
Tujuan	Menampilkan tampilan *Host Game* dan cara membuat *server game*
Deskripsi	Proses untuk membuat ruangan baru atau membuat *server* baru
Aktor	*User* / Pengguna
Skenario Utama	
Kondisi Awal	Program *Fire Shooter 2D* dibuka
Aksi Aktor	**Reaksi Sistem**
1. Membuka aplikasi *Fire Shooter 2D*	2. Menampilkan *Room List, Host Game, dan Setting*
3. Membuka *Host Game*	4. Menampilkan nama ruangan atau *server* yang akan dibuat, serta menentukan level yang akan digunakan,
5. *Create a new game*	
Kondisi Akhir	Aplikasi menampilkan ruangan yang telah dibuat dan mulai permainan

D. Skenario Use Case Menampilkan *Level*

Use case skenario ini menjelaskan interaksi aktor, dengan *use case* menampilkan *Level* yang akan dijelaskan pada tabel berikut.

Tabel 3. 15 Skenario Use Case Menampilkan Level

Identifikasi

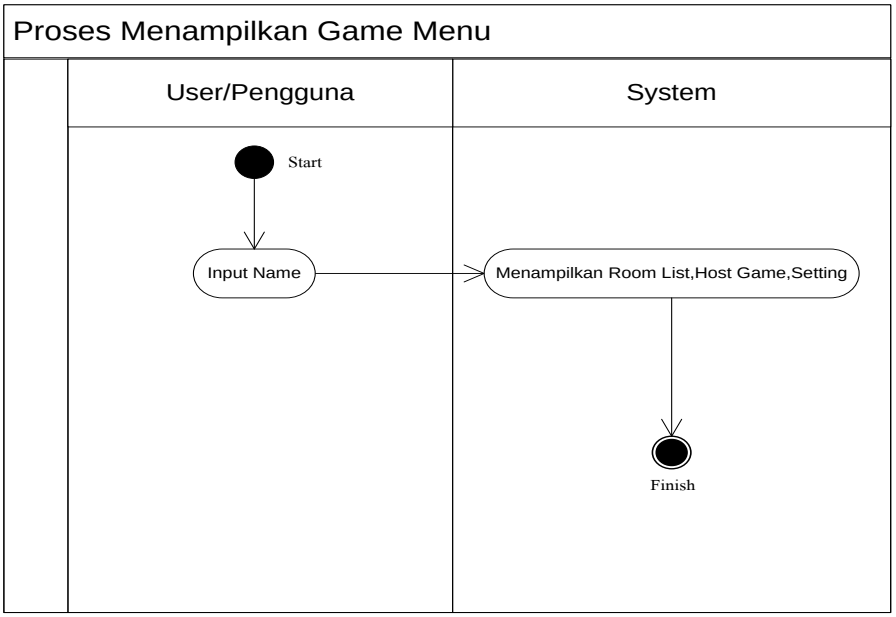
Nomor	4
Nama	Menampilkan *Level*
Tujuan	Menampilkan tampilan *Level*
Deskripsi	Proses untuk membuat ruangan baru atau membuat *server* baru
Aktor	*User /* Pengguna
Skenario Utama	
Kondisi Awal	Program *Fire Shooter 2D* dibuka
Aksi Aktor	**Reaksi Sistem**
1. Membuka aplikasi *Fire Shooter 2D*	2. Menampilkan *Room List, Host Game, dan Setting*
3. Membuka *Host Game*	4. Menampilkan *level* yang sebelumnya pernah digunakan
5. Memilih *Level*	
Kondisi Akhir	Aplikasi menampilkan ruangan yang telah dibuat dan mulai permainan

3.5.1. Activity Diagram

Activity Diagram menggambarkan berbagai aliran aktivitas dari setiap proses dalam sistem yang sedang dirancang, bagaimana masing-masing alur berawal, *decision* yang mungkin terjadi, dan bagaimana berakhir. Gambaran *Activity diagram* pada permainan *Fire Shooter 2D* dijelaskan sebagai berikut.

3.5.1.1. *Activity Diagram Menampilkan Game Menu*

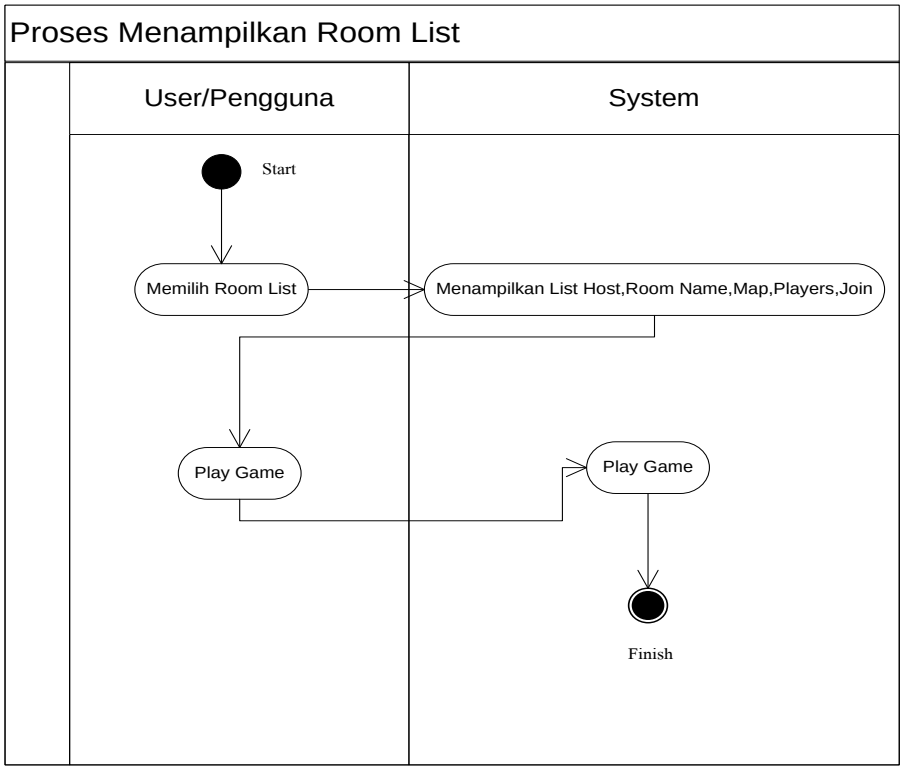
Berikut ini adalah diagram yang menunjukan alur aksi pada aktivitas menampilkan *game menu*, pada permainan *Fire Shooter 2D* yang dapat dilihat pada gambar 3.11 di bawah ini.



Gambar 3. 12 *Actifity Diagram* Menampilkan *Menu* Utama

3.5.1.2. A*ctivity Diagram* Menampilkan *Room List*

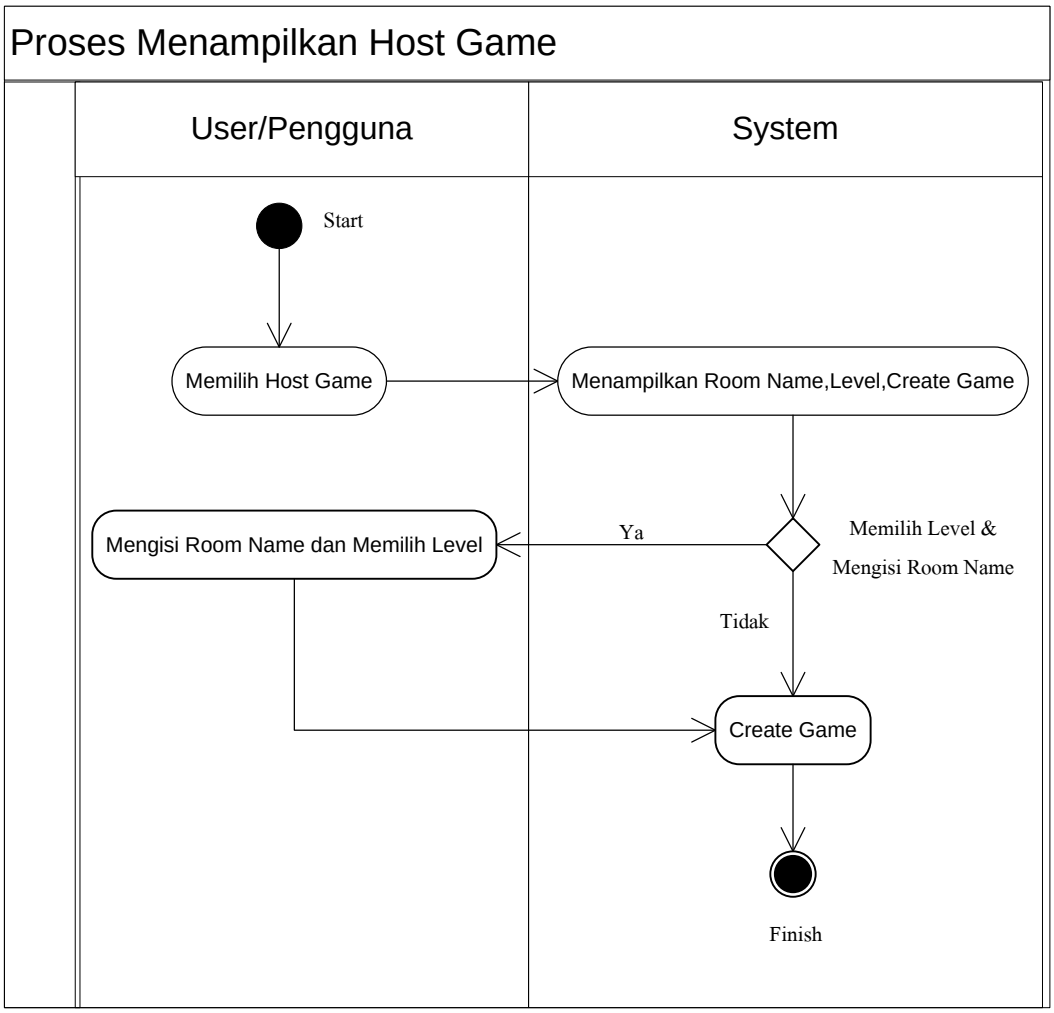
Berikut ini adalah diagram yang menunjukan alur aksi pada aktivitas menampilkan *Room List*, pada permainan *Fire Shooter 2D* yang dapat dilihat pada gambar 3.12 di bawah ini.



Gambar 3. 13 *Actifity Diagram* Menampilkan *Room List*

3.5.1.3. A*ctivity Diagram* Menampilkan *Host Game*

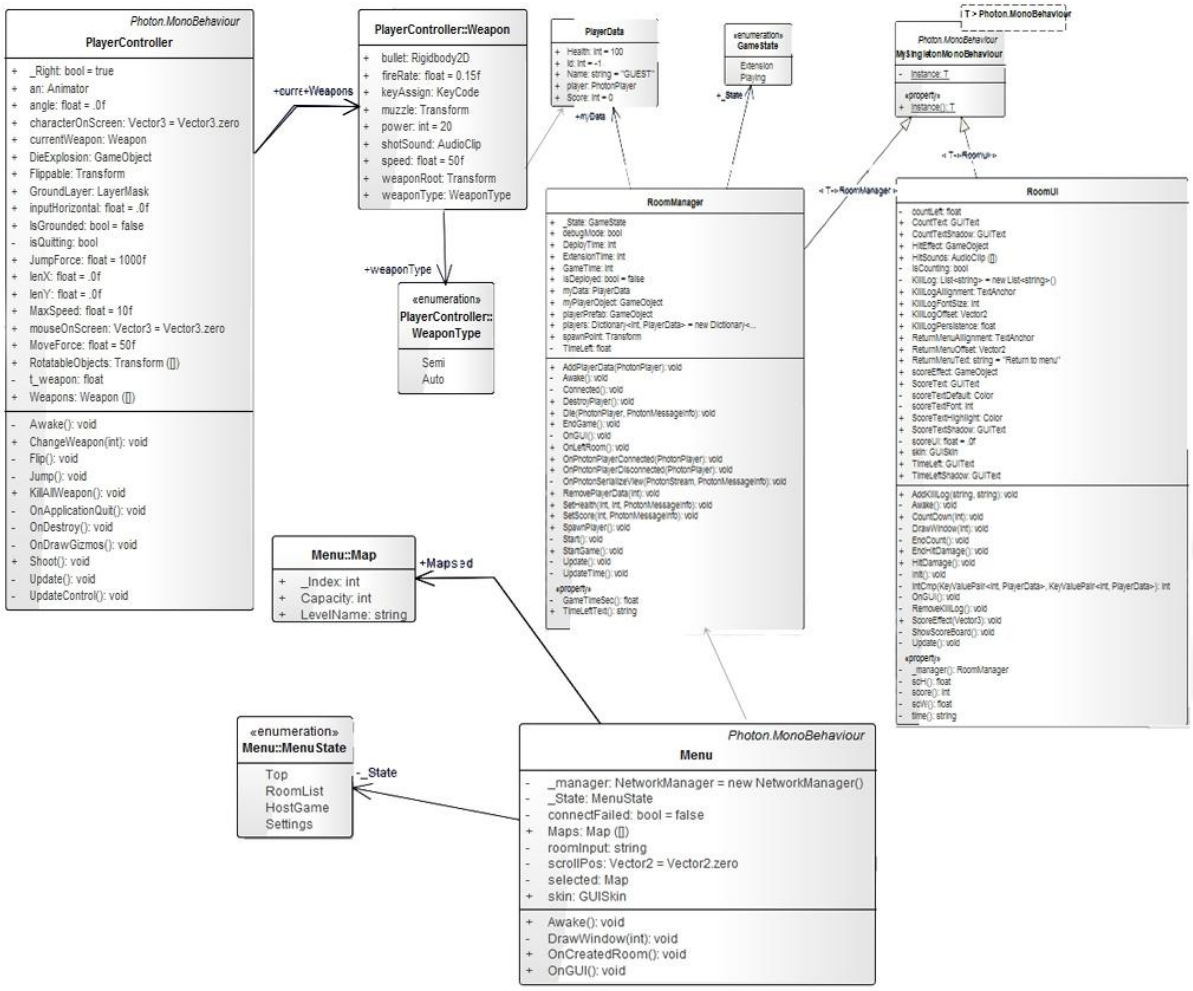
Berikut ini adalah diagram yang menunjukan alur aksi pada aktivitas menampilkan *Host Game*, pada permainan *Fire Shooter 2D* yang dapat dilihat pada gambar 3.13 di bawah ini.



Gambar 3. 14 *Actifity Diagram* Menampilkan *Host Game*

3.5.2. *Class* Diagram

Class diagram menggambarkan struktur dan hubungan antar kelas-kelas yang terdapat pada sistem. Struktur pada *class* digram meliputi atribut-atribut dan metode-metode yang ada pada masing-masing *class*. *Class* diagram pada permainan *Fire Shooter 2D* dapat dilihat pada Gambar 3.14 berikut ini.



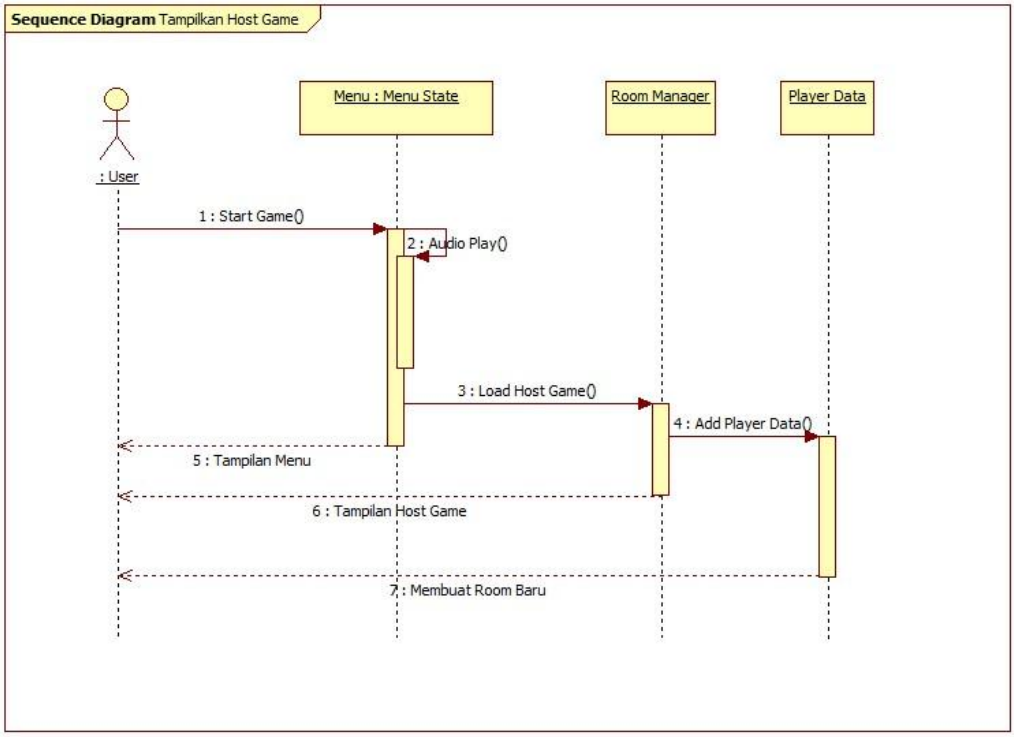
Gambar 3. 15 *Class Diagram*

3.5.3. Sequence Diagram

Sequence diagram menggambarkan interaksi antar masing-masing objek pada setiap *use case* dalam urutan waktu. Interaksi ini berupa pengiriman serangkaian data antar objek-objek yang saling berinteraksi. *Sequence diagram* pada permainan *Fire Shooter 2D* dijelaskan sebagai berikut.

3.5.3.1. *Sequence Diagram Game Fire Shooter*

Berikut ini adalah gambar *sequence diagram Game Fire Shooter Multiplayer* yang, akan dijelaskan pada gambar berikut



Gambar 3. 16 *Sequence* Diagram *Game Fire Shooter*

3.6. Perancangan Sistem

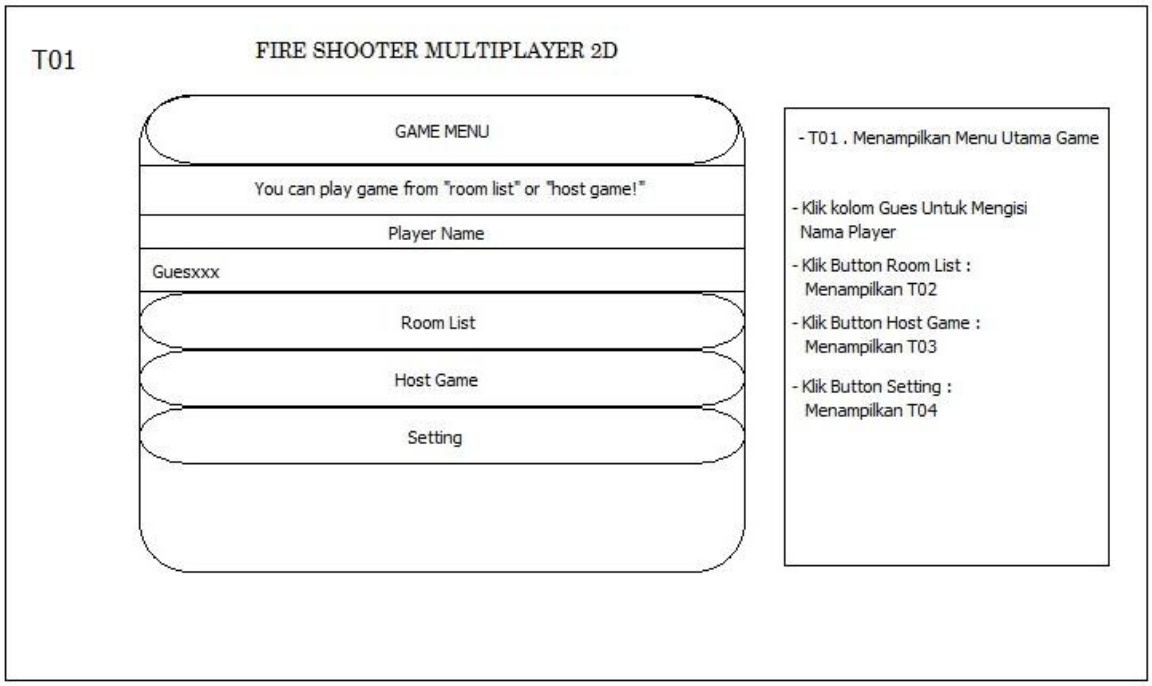
Adapun tahapan yang dilakukan dalam perancangan sistem ini membahas mengenai tujuan perancangan antarmuka, dan jaringan simantik

3.6.1. Perancangan Antar Muka

Antar muka merupakan tampilan dari suatu program aplikasi *Fire Shooter 2D* Perancangan antarmuka dibutuhkan untuk mawakili keadaan sebenarnya dari aplikasi yang akan dibangun. Berikut ini beberapa perancangan antarmuka dari aplikasi yang akan dibangun yaitu :

3.6.1.1. perancangan Antar Muka Menu Utama

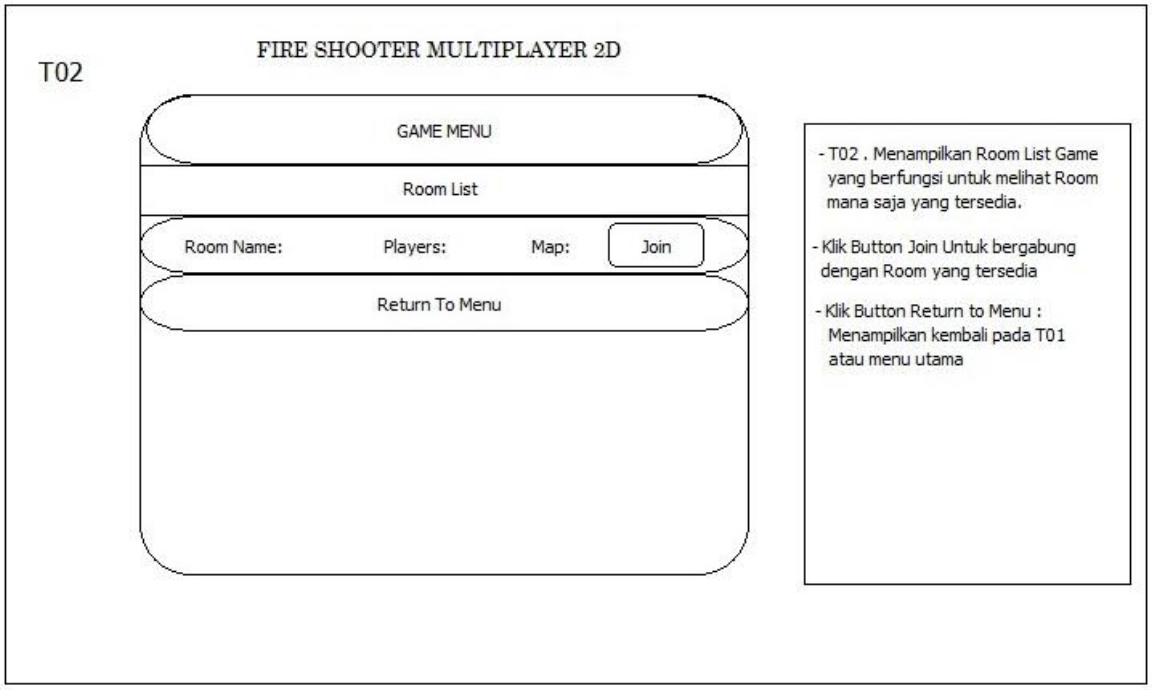
Perancangan menu utama mendeskripsikan rencana tampilan dari awal aplikasi *Fire Shooter 2D* dibuka. Berikut merupakan gambaran dari perancangan antarmuka aplikasi *Fire Shooter 2D*



Gambar 3. 17 Perancangan Antar Muka Menu Utama

3.6.1.2. Perancangan Antar Muka Room List

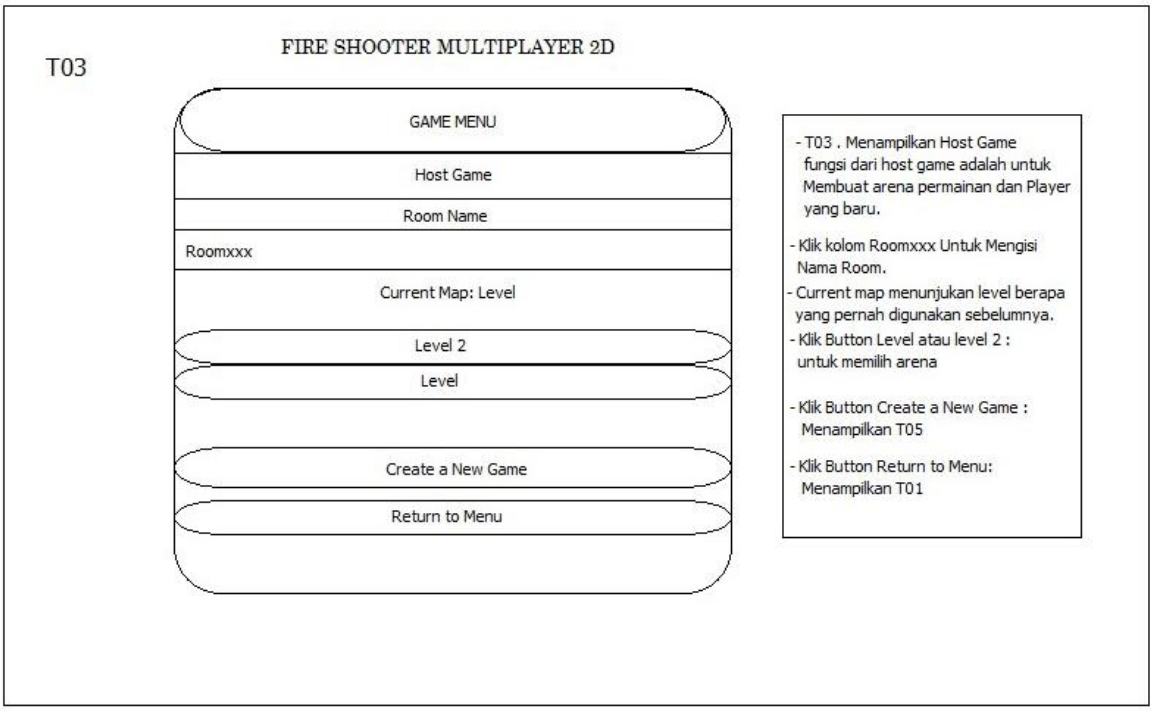
Perancangan *Room List* mendeskripsikan rencana tampilan ketika user akan memasuki Room yang sudah dibangun oleh *player* atau pemain lain, Berikut merupakan gambaran dari perancangan antar muka *Room List* pada aplikasi *Fire Shooter 2D.*



Gambar 3. 18 Perancangan Antar Muka Room List

3.6.1.3. Perancangan Antar Muka Host Game

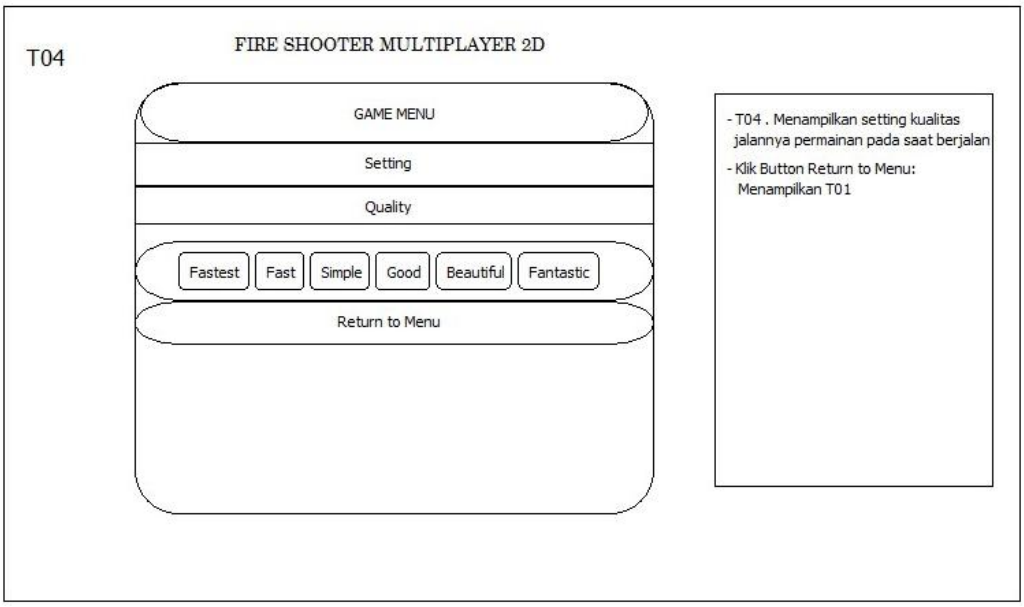
Perancangan *Host Game* mendeskripsikan rencana tampilan ketika user akan membangun Room baru, Berikut merupakan gambaran dari perancangan antar muka *Host Game* pada aplikasi *Fire Shooter 2D.*



Gambar 3. 19 Perancangan Antar Muka Host Game

3.6.1.4. Perancangan Antar Muka *Setting*

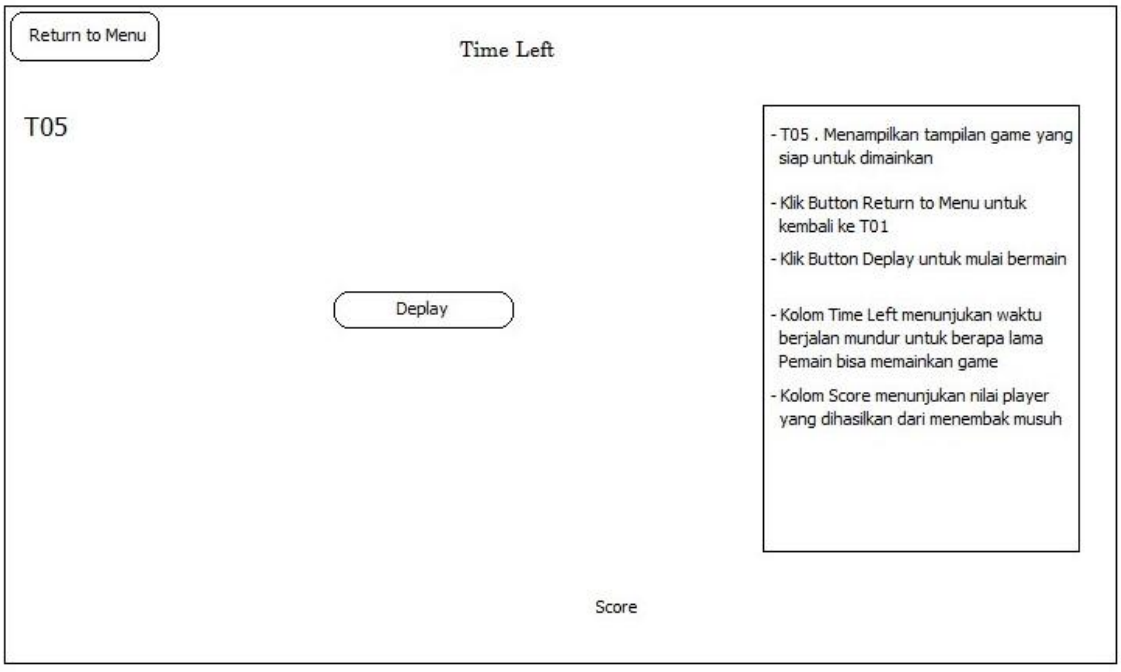
Perancangan *Setting* mendeskripsikan rencana tampilan untuk user kecepatan permainan yang di inginkan, Berikut merupakan gambaran dari perancangan antar muka *Setting* pada aplikasi *Fire Shooter 2D.*



Gambar 3. 20 Perancangan Antar Muka Setting

3.6.1.5. Perancangan Antar Muka *Game*

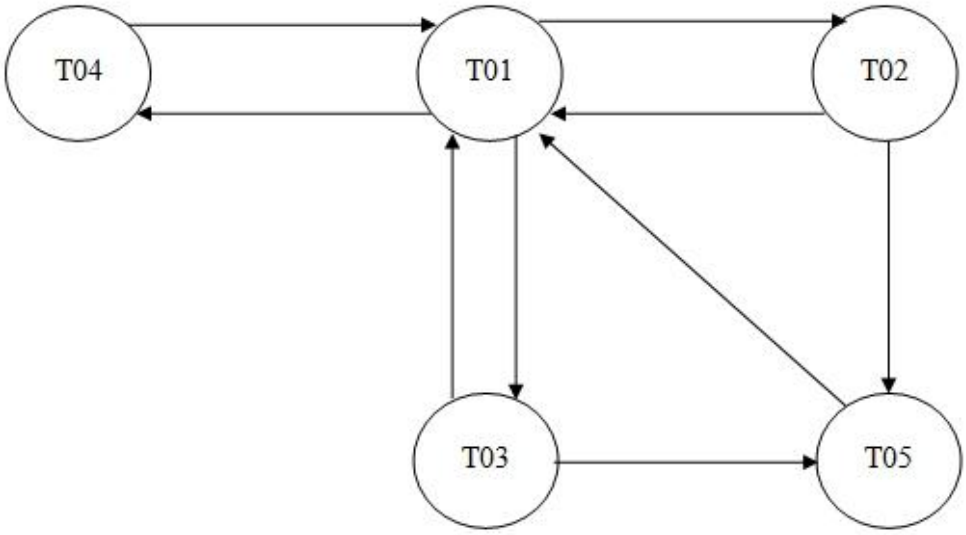
Perancangan *Game* mendeskripsikan rencana tampilan untuk menampilkan tampilan game yang akan dimainkan, Berikut merupakan gambaran dari perancangan antar muka *Game* pada aplikasi *Fire Shooter 2D.*



Gambar 3. 21 Perancangan Antar Muka Game

3.6.2. Jaringan Semantik

Jaringan semantik merupakan gambaran grafis yang menunjukkan antar objek (lingkaran) dan garis yang menggambarkan formasi antar objek tersebut. Adapun jaringan semantik untuk aplikasi *Fire Shooter 2D* dapat dilihat pada gambar dibawah:



Gambar 3. 22 Jaringan Semantik