Penggunaan QuadTree dalam optimasi berbagai persoalan pada game

Muhammad Afif Al-hawari (13510020)
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
afif.alhawari@s.itb.ac.id

***Abstrak*—QuadTree merupakan struktur pohon dimana setiap simpul dalam dari pohon tersebut memiliki tepat empat buah anak. QuadTree ini banyak dimanfaatkan untuk berbagai macam persoalan optimasi, terutama dalam persoalan seperti *pathfinding, Collision Detection* yang seringkali muncul dalam game. Pada makalah ini dibahas mengenai cara optimasinya, serta seberapa besar dampaknya terhadap optimasi persoalan tersebut.**

***Kata Kunci*—QuadTree, Collision Detection, Pathfinding algorithm.**

I. PENDAHULUAN

Pada game, persoalan optimasi merupakan hal yang sangat sering sekali dibahas. Banyak pertanyaan-pertanyaan muncul mengenai topik ini. Pada dasarnya game dibedakan menjadi dua kategori, yaitu game yang bersifat *real-time* serta yang bersifat *input-driven*. Game yang bersifat *input-driven* ini merupakan game yang menampilkan state game saat itu dan menunggu input user untuk melanjutkan ke state berikutnya. Contoh game yang termasuk tipe ini yaitu game-game puzzle, permainan kartu, serta beberapa game strategi. Untuk game yang bersifat *real-time* permainan akan terus berlangsung hingga selesai, tanpa menunggu input dari user. Game-game yang termasuk golongan ini biasanya merupakan game action yang biasanya membutuhkan banyak gambar, refleks mata serta tangan yang lincah serta membutuhkan banyak kalkulasi untuk setiap pergerakannya.

Untuk game yang bersifat *input-driven*, seringkali optimasi tidak terlalu diperlukan, namun untuk game-game action yang bersifat *real-time* optimasi merupakan sebuah keharusan. Game action yang bagus tentunya harus memiliki respon yang cepat, serta rate yang tinggi. Game yang lambat dan memiliki gambar yang patah-patah tentunya tidak nyaman serta menarik untuk dimainkan. Persoalan optimasi ini akan menjadi lebih penting lagi jika device yang digunakan untuk game itu memiliki kemampuan yang sangat terbatas, terutama kemampuan dalam *processing*. Hal ini tentunya membuat developer perlu memutar otak lebih keras untuk melakukan optimasi terhadap kode mereka agar game yang dibuat bisa berjalan dengan baik, dan nyaman dimainkan.

Beberapa persoalan klasik seperti *collision detection* serta *pathfinding* merupakan persoalan yang muncul hampir pada setiap game yang ada, khusunya yang bersifat *real-time*. Persoalan ini cukup menarik dan biasanya merupakan objek utama dalam melakukan optimasi.

Misal untuk kasus *collision detection.* Game-game biasanya memerlukan ini untuk mengecek apakah ada dua atau lebih objek yang bertabrakan. Misal pada pembuatan game *shooter*, kita tentu perlu mengecek apakah objek peluru itu mengenai objek lain seperti pesawat, dll. Bila ada 10 objek pada layar maka setiap objek perlu dicek apakah bertabrakan dengan 9 objek lainnya, sehingga total ada 90 pengecekan. Bayangkan jika objek pada layar jumlahnya mencapai ratusan, atau bahkan ribuan. Besarnya pengecekan yang pelu dilakukan pada setiap waktunya akan naik secara exponensial dan tentunya menyebabkan game menjadi lambat.

Begitu juga dengan masalah *pathfinding*. Untuk mencari jalan terpendek dari tempat asal menuju tempat tujuan, menggunakan algoritma apapun, biasanya besar kemungkinan perlu melakukan penelusuran terhadap jalan yang seharusnya tidak perlu kita lewati. Hal ini tidak menjadi masalah jika map yang ditelusuri tidak terlalu besar. Namun, jika map pada game tersebut sangatlah besar, proses pencarian ini tentunya akan berjalan lebih lambat. Tentunya perlu ditambahkan optimasi dengan metode tertentu untuk menyederhanan proses pencarian ini.

Berbagai metode bisa dilakukan dalam menyederhakan persoalan tersebut, contohnya dengan menggunakan QuadTree. QuadTree ini merupakan sebuah struktur pohon yang unik, dimana setiap simpul dalam pohon tersebut harus memiliki tempat empat buah anak. QuadTree ini digunakan untuk melakukan pengelompokan terhadap objek-objek yang ada pada suatu game.

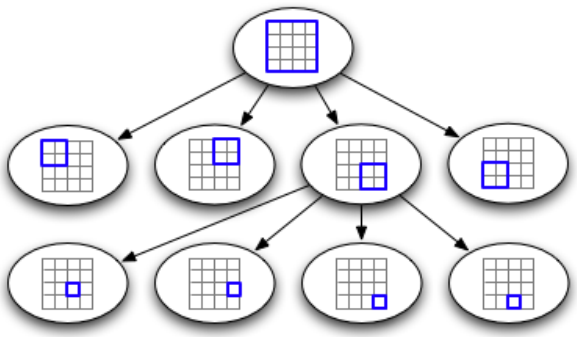
Pada Makalah ini, penulis akan membahas bagaimana menggunakan QuadTree ini untuk menyederhakan dua persoalan klasik diatas, yaitu *Collision Detection* serta *pathfinding.*

II. DASAR TEORI

1. QuadTree

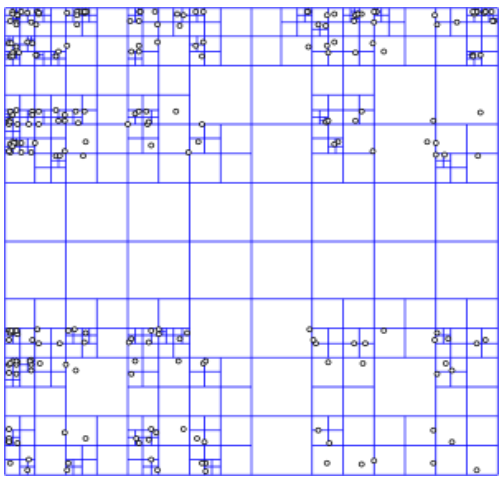
QuadTree merupakan struktur pohon yang unik dimana setiap simpul dalam dari pohon tersebut memiliki tepat empat buah anak. QuadTree ini umumnya digunakan untuk melakukan partisi terhadap ruang dua dimensi (*spatial indexing)* dengan cara membaginya secara rekusif menjadi empat buah ruang.

Pendekatan *spatial indexing* menggunakan QuadTree ini sangat *straightforward*. Setiap node pada QuadTree merepresentasikan sebuah bounding box yang merupakan sub ruang yang sedang di indeks, sedang akar dari QuadTree merupakan sebuah sub area yang dilingkupi dan tidak memiliki sub ruang lagi.



Gambar 3.3 Representasi QuadTree dalam bentuk pohon

Dalam melakukan *spatial indexing* pada game, biasanya pembagian dilakukan berdasarkan banyaknya objek yang ada pada suatu daerah tertentu. Bila pada ruang tersebut, jumlah objek yang ada telah melebihi jumlah maksimum objek yang telah kita tentukan, maka kita perlu membaginya menjadi empat bagian yang lebih kecil dan menempatkan kembali objek-objek tersebut para ruangan yang telah dibagi tersebut sesuai posisinya. Apabila sebuah objek tidak dapat masuk sepenuhnya pada suatu area tertentu, maka objek tersebut akan ditempatkan pada area / node parentnya.



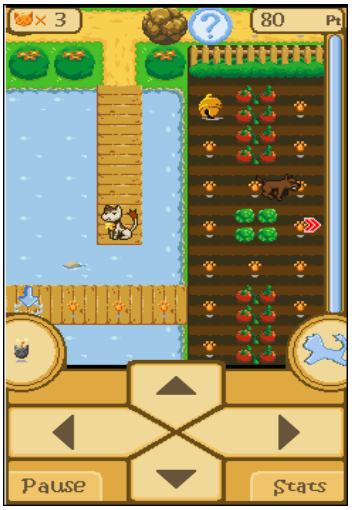
Gambar 3.3 Representasi Spatial Indexing menggunakan QuadTree(Source : Wikipedia)

2. Collision Detection

Collision Detection ini merupakan persoalan untuk mendeteksi dua atau lebih objek yang berpotongan/bertabrakan. Persoalan ini merupakan persoalan yang paling umum, dan muncul hampir pada setiap game yang ada.

Gambar 3.3 Touhou, contoh game shooter yang terdiri dari banyak sekali objek peluru

Pendeteksian tabrakan ini biasanya dilakukan dengan berbagai macam cara tergantung dengan implementasi sistem dari game tersebut. Untuk game yang berbasiskan grid/tile misalnya pendeteksian collision bisa dilakukan dengan mengecek apakah dua buah objek itu berada pada tile yang sama. Untuk game-game lain yang tidak berbasiskan tile, *collision detection* bisa dilakukan dengan membuat *bounding box* dari setiap objek. Tabrakan akan terjadi jika *bounding box* antar dua objek atau lebih ini saling berhimpit atau berpotongan.



Gambar 3.3 MiawVenture, contoh game yang berbasiskan tile

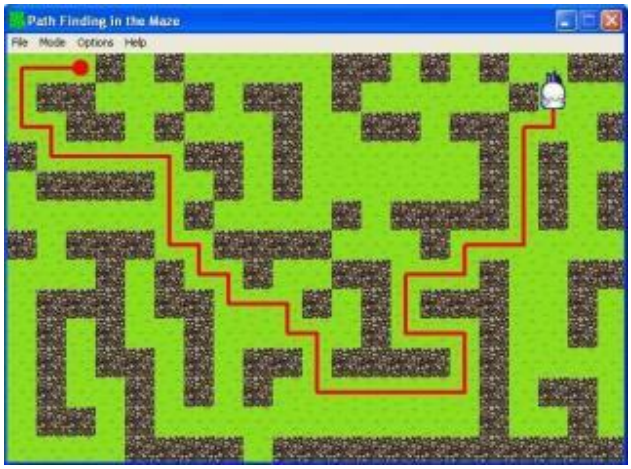
Persoalan *Collision Detection* merupakan salah satu proses yang paling banyak memakan waktu dalam setiap siklus update pada game. Developer game berlomba-lomba dalam melakukan optimasi terhadap persoalan yang satu ini. Berbagai metode optimasi dilakukan, dan salah satu yang dianggap paling efektif adalah melakukan *spatial indexing* menggunakan QuadTree, sehingga tidak

perlu melakukan pengecekan terhadap dua objek yang terletak sangat berjauhan. Metode lengkapnya akan dibahas lebih lanjut pada bagian III makalah ini.

3. Pathfinding

Pathfinding merupakan salah satu persoalan klasik lainnya yang sering muncul dalam berbagai game. Pokok permasalahan yang dibahas adalah menentukan jalan yang dapat dilalui dari posisi awal menuju posisi yang dinginkan. Biasanya solusi yang diinginkan merupakan solusi yang optimum/merupakan jalur terdekat yang menghubungkan dua titik tersebut.

Pada game biasanya persoalan ini muncul ketika membuat pola pergerakan musuh. Untuk musuh yang tipe pergerakannya selalu mengikuti karakter kita tentu pada setiap perlu dilakukan kalkulasi dengan algoritma *pathfinding* untuk menemukan jalur yang perlu dilewati musuh tersebut agar bisa menuju posisi karakter kita. Pada game-game tertentu juga terkadang terdapat sistem dimana player cukup mengklik di suatu titik tertentu pada layar permainan, dan kemudian karakter dari player tersebut akan berjalan secara otomatis ke titik yang diinginkan. Untuk menentukan jalur yang dilewati tersebut agar tidak terbentur halangan dan semacamnya tentunya perlu diterapkan algoritma *pathfinding*.



Gambar 3.3 Contoh persoalan yang diselesaikan menggunakan algoritma *pathfinding*

Persoalan *pathfinding* ini biasanya dimodelkan dalam bentuk graf. Dimana tempat yang bisa dikunjungi dimodelkan sebagai simpul serta jalur antara simpul-simpul tersebut dimodelkan sebagai sisinya, dimana bobot sisi merupakan jarak antar dua simpul tersebut. Persoalannya adalah menentukan sisi-sisi yang menghubungkan antara simpul awal dengan simpul tujuan dimana jumlah bobot sisi/jarak total dari kedua simpul tersebut harus minimal.

Banyak algoritma-algoritma yang bisa digunakan untuk menyelesaikan persoalan ini. Mulai dari Algoritma Brute Force, Algoritma Djikstra yang menerapkan prinsip Greedy, algoritma DFS, BFS ataupun ID, hingga algoritma yang lebih kompleks seperti algoritma A*. Masing-masing algoritma memiliki keunggulan serta kekurangan masing-masing, sehingga programmer perlu pintar-pintar memilih algoritma yang paling tepat untuk persoalan *pathfinding* yang dimiliki. Namun, khususnya untuk game, algoritma A* merupakan algoritma yang paling mangkus, karena mampu memberikan penghitungan *shortest-path* dengan waktu eksekusi yang paling kecil untuk hampir semua kasus yang muncul.

Dalam game tentunya tantangan utama dari pembuatnya adalah membuat seluruh eksekusi seoptimal mungkin. Algoritma-algoritma *pathfinding* ini bisa dioptimasi lagi dengan menggunakan pendekatan heuristik tertentu, ataupun dengan *spatial indexing* menggunakan QuadTree seperti yang dibahas pada makalah ini.

III. Analisis dan Pembahasan

1. Optimasi Collision Detection

Pada bagian sebelumnya sempat di singgung sedikit bagaiman cara melakukan *collision detection* dengan menggunakan pendekatan yang *straightforward* yaitu algoritma brute force. Menggunakan algoritma brute force ini seperti yang kita tahu, akan membutuhkan operasi pengecekan yang sangat besar. Bila ada 10 objek maka perlu dilakukan pengecekan sebanyak 90 kali. Bila jumlah objek ini sangat banyak sekali maka jumlah pengecekannya akan meningkat secara eksponensial sehingga membutuhkan waktu eksekusi yang tidak sedikit.

Menggunakan spatial indexing dengan QuadTree, kita akan mencoba meminimalisasi jumlah pengecekan tersebut. Ide utamanya adalah membagi objek-objek yang ada pada layar ke dalam ruang-ruang. Setiap objek hanya perlu di cek apakah berpotongan dengan objek yang berada satu ruang dengannya. Melalui metode ini kita akan mengurangi cukup banyak operasi pengecekan yang tidak perlu dilakukan sehingga waktu eksekusi akan jauh lebih cepat.

Sebelumnya kita akan membahas terlebih dahulu, bagaimana cara melakukan spatial indexing ini dengan cara mengimplementasikan QuadTree.

Pertama-tama kita perlu mendefinisikan dulu struktur QuadTree kita. Kelas QuadTree ini dibuat dengan memanfaatkan tutorial pada gamedev.tutsplus.com.

```
public class QuadTree {

  private int MAX_OBJECTS = 5;
  private int MAX_SUBDIVISION = 3;
  private int level;
  private ArrayList<Rectangle> objects;
  private Rectangle boundBox ;
  private QuadTree[] child;

  public QuadTree(int pLevel, Rectangle pBounds) {
   level = pLevel;
   objects = new ArrayList<Rectangle>();
   boundBox = pBounds;
   child = new QuadTree[4];
  }
}
```

Pada kelas QuadTree yang ditulis dalam bahasa java ini kita memiliki beberapa atribut-atribut yang penting. MAX_OBJECT merepresentasikan objek maksimal pada suatu ruang. Jika objek pada ruang tersebut telah melebihi MAX_OBJECT ini, maka perlu ruang tersebut perlu dibagi lagi menjadi empat ruang yang lebih kecil. MAX_LEVEL merepresentasikan jumlah level pembagian ruang yang diijinkan. Jika level telah mencapai MAX_LEVEL maka ruang sudah tidak dapat dibagi lagi. Level merepresentasikan level dari ruang QuadTree tersebut, sedangkan boundBox merepresentasikan batas ruangnya. Semua objek yang ada pada ruang QuadTree ini akan disimpan dalam list Objects. Setiap ruang akan memiliki empat buah sub ruang yang akan dibuat jika objek telah melebihi batas maksimum.

Untuk kita melakukan membagi objek-objek yang ada pada layar ke dalam struktur QuadTree, kita perlu beberapa operasi dasar, yaitu operasi *split*, operasi *insert* serta operasi *retrieve*.

```
private void split() {
    int subWidth = (int) (boundBox.getWidth() / 2);
    int subHeight = (int) (boundBox.getHeight() / 2);
    int x = (int) boundBox.getX();
    int y = (int) boundBox.getY();

    child[0] = new QuadTree(level + 1, new Rectangle(x +
subWidth, y, subWidth, subHeight));
    child[1] = new QuadTree(level + 1, new Rectangle(x,
y, subWidth, subHeight));
    child[2] = new QuadTree(level + 1, new Rectangle(x,
y + subHeight, subWidth, subHeight));
    child[3] = new QuadTree(level + 1, new Rectangle(x +
subWidth, y + subHeight, subWidth, subHeight));

}
```

Operasi split ini digunakan untuk membuat empat buah sub ruang baru dari sebuah ruang. Operasi ini dilakukan ketika objek yang ada telah memiliki batas maksimum dan jumlah level belum mencapai maksimal.

Untuk melakukan operasi *insert* dan *retrieve* kita perlu operasi bantuan tambahan yaitu *getIndex*.

```
private int getIndex(Rectangle pRect) {
        int index = -1;
        double verticalMidpoint = boundBox.getX() +
(boundBox.getWidth() / 2);
        double horizontalMidpoint = boundBox.getY() +
(boundBox.getHeight() / 2);

        // Object bisa masuk sepenuhnya ke sub ruang
bagian atas
        boolean topQuadrant = (pRect.getY() <
horizontalMidpoint && pRect.getY() + pRect.getHeight()
< horizontalMidpoint);
        // Object bisa masuk sepenuhnya ke sub ruang
bagian bawah
        boolean bottomQuadrant = (pRect.getY() >
horizontalMidpoint);

        // Object bisa masuk sepenuhnya ke sub ruang
bagian kiri
        if (pRect.getX() < verticalMidpoint &&
pRect.getX() + pRect.getWidth() < verticalMidpoint) {
                if (topQuadrant) {
                        index = 1;
                } else if (bottomQuadrant) {
                        index = 2;
                }
        } // Object bisa masuk sepenuhnya ke sub ruang
bagian kanan
        else if (pRect.getX() > verticalMidpoint) {
                if (topQuadrant) {
                        index = 0;
                } else if (bottomQuadrant) {
                        index = 3;
                }
        }

        return index;
}
```

Operasi getIndex ini akan mengembalikan indeks ruang yang tepat untuk suatu objek tertentu. Indeks 0 merupakan subruang bagian atas kanan, indeks 1 merepresentasikan subruang bagian atas kiri. Indeks 2 dan 3 masing-masing merepresentasikan subruang bagian kiri bawah dan kanan bawah. Jika hasil getIndex ini mengembalikan nilai -1, maka objek tersebut tidak bisa diletakkan sepenuhnya pada salah satu sub ruang, sehingga akan diletakkan sebagai bagian dari ruang parentnya.

```
public void insert(Rectangle pRect) {
    if (child[0] != null) {
        int index = getIndex(pRect);

        if (index != -1) {
            child[index].insert(pRect);
            return;
        }
    }

    objects.add(pRect);

    if (objects.size() > MAX_OBJECTS && level <
MAX_SUBDIVISION) {
        if (child[0] == null) {
            split();
        }

        int i = 0;
        while (i < objects.size()) {
            int index = getIndex(objects.get(i));
            if (index != -1) {
                child[index].insert(objects.remove(i));
            } else {
                i++;
            }
        }
    }
}
```

Operasi *insert* ini merupakan yang menyatukan operasi-operasi dasar sebelumnya. Operasi ini pertama-tama akan menentukan terlebih dahulu apakah ruang ini sudah memiliki sub ruang. Apabila ada sub ruang, maka objeknya akan dimasukkan ke sub ruang tersebut. Namun, apabila tidak ada sub ruang ataupun apabila objek tersebut tidak bisa diletakkan sepenuhnya di salah satu sub ruang, maka objek tersebut akan dimasukkan ke dalam ruang ini.

Ketika objek tersebut telah dimasukkan ke dalam ruang yang tepat, selanjutkan akan dilakukan pengecekan apakah jumlah objek yang ada pada ruang tersebut telah melebihi jumlah maksimal yang diizinkan. Jika jumlah objek telah melebihi jumlah maksimal serta level sub ruang belum mencapai maksimal, maka akan dilakukan operasi split. Untuk setiap objek yang bisa dimasukkan sepenuhnya ke dalam salah satu sub ruang tersebut, akan dilakukan operasi *insert*.

```
public ArrayList<Rectangle>
retrieve(ArrayList<Rectangle> returnObjects, Rectangle
pRect) {
    int index = getIndex(pRect);
    if (index != -1 && child[0] != null) {
```

```
            child[index].retrieve(returnObjects, pRect);
        }
        returnObjects.addAll(objects);

        return returnObjects;
    }
```

Operasi yang terakhir yaitu operasi *retrieve*. Operasi ini akan mengembalikan list dari seluruh objek yang mungkin bertabrakan dengan objek yang sedang kita cek.

Setelah selesai mengimplementasikan seluruh operasi dasar QuadTree diatas, maka kita sudah bisa menggunakannya untuk melakukan optimasi terhadap algoritma *collision detection* kita.

```
QuadTree space = new QuadTree(0, new Rectangle(0, 0,
MAP_WIDTH, MAP_HEIGHT));
for (int i = 0; i < ListObject.size(); i++) {
    space.insert(ListObject.get(i));
}
ArrayList<Rectangle> returnObjects = new ArrayList();
for (int i = 0; i < ListObject.size(); i++) {
    returnObjects.clear();
    space.retrieve(returnObjects, ListObject.get(i));

    for (int x = 0; x < returnObjects.size(); x++) {
        //melakukan pengecekan menggunakan algoritma
collision detection
    }
}
```

Menggunakan QuadTree ini, jumlah pengecekan yang diperlukan sudah jelas akan jauh lebih sedikit dibanding dengan menggunakan pendekatan *straightforward* menggunakan algoritma *bruteforce*. Walaupun setiap melakukan pengecekan kita perlu melakukan *generate* ulang terhadap struktur QuadTree kita, namun operasi ini lebih singkat dibanding melakukan perbandingan terhadap objek-objek secara *bruteforce*.

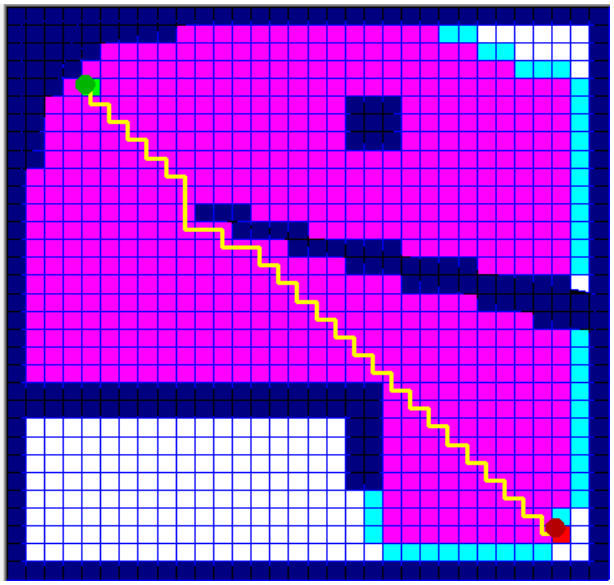
Sebagai pembanding, menggunakan struktur QuadTree yang telah dibahas diatas, untuk pengecekan *collision detection* terhadap 500 objek pada layar dibutuhkan 249.500 operasi perbandingan dan waktu eksekusi rata-rata selama 55ms. Sedangkan jika kita menggunakan struktur QuadTree rata-rata proses perbandingan yang dibutuhkan hanya 30.000 kali serta waktu eksekusi yang dibutuhkan hanya sekitar 25ms. Dari hasil tersebut tentu kita bisa menyimpulkan bahwa optimasi *collision detection* menggunakan QuadTree terbilang sangat efektif.

2. Optimasi Pathfinding

Pada dasarnya prinsip optimasi menggunakan QuadTree untuk kasus pathfinding mirip dengan kasus untuk optimasi Collision Detection. Pertama-tama kita perlu melakukan spatial indexing terhadap map kita. Struktur pembentukan QuadTree disini akan sedikit berbeda dengan kasus sebelumnya. Untuk kasus pathfinding ini, sebuah ruang yang tidak memiliki sub ruang, merupakan ruang kosong yang seluruh bagiannya bisa dilewati oleh karakter kita. Jika pada ruang tersebut ada objek yang tidak bisa dilewati oleh karakter kita, maka ruang tersebut dibagi menjadi 4 ruang sama besar.

Pada bagian ini penulis tidak menyertakan algoritma program untuk pembentukan QuadTree seperti pada bagian sebelumnya. Kali ini, penulis menggunakan ilustrasi dari bagaimana proses optimasi dilakukan dengan menggunakan sebuah program simulasi yang didapat dari http://www.ai-blog.net.

Pertama mari kita lihat salah satu contoh ilustrasi persoalan pathfinding berikut.



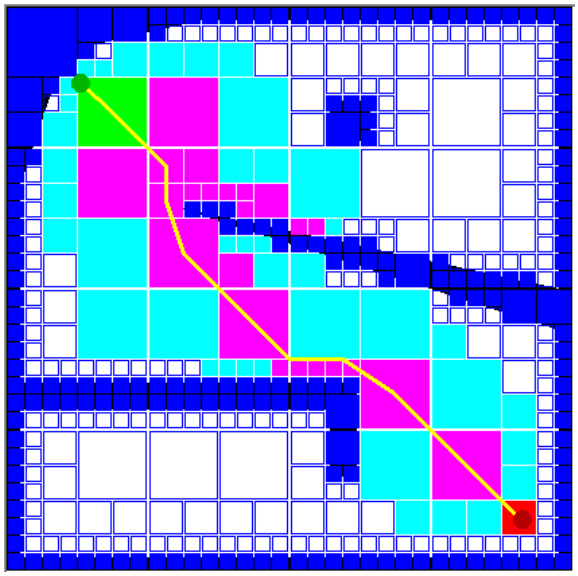
Gambar 3.3 Persoalan pathfinding yang diselesaikan dengan algoritma A* tanpa menggunakan QuadTree

Pada Gambar diatas, titik hijau merupakan melambangkan simpul asal, sedangkan titik merah melambangkan simpul tujuan. Bagian berwarna biru gelap merupakan bagian yang tidak bisa dilewati oleh karakter. Bagian yang berwarna ungu merupakan bagian yang perlu di periksa jika kita menggunakan algoritma A* dengan pola pergerakan 4 arah. Garis kuning melambangkan jalur terpendek yang perlu dilewati untuk mencapai ke simpul tujuan.

Pada contoh diatas, kita tidak menerapkan spatial indexing menggunakan QuadTree. Kita bisa lihat banyaknya simpul yang perlu periksa sebelum menemukan solusi. Walaupun algoritma A* merupakan algoritma yang paling mangkus, namun untuk kasus ini tetap saja proses pencarian berjalan sangat panjang dan hampir mengcover seluruh area.

Ide optimasi menggunakan QuadTree kali ini adalah mengurangi jumlah simpul tersebut dengan menggolongkan sebanyak-banyak simpul yang tidak memiliki objek yang tidak bisa dilewati menjadi satu ruang, sehingga ketika melakukan pengecekan, bisa dilakukan dengan cukup mengecek apakah ruang tersebut bisa dilewati. Metode pengelompokannya seperti pada yang sudah dijelaskan di awal bagian optimasi pathfinding ini.

Berikut ilustrasi dari persoalan sebelumnya, jika struktur QuadTree telah kita terapkan.



Gambar 3.3 Persoalan pathfinding yang diselesaikan dengan algoritma A* dengan menggunakan QuadTree

Pada gambar diatas, kita menerapkan QuadTree untuk mengelompokkan simpul-simpul pada map kita, sehingga kita bisa dapatkan node dengan jumlah yang lebih sedikit dibanding sebelumnya. Kita bisa perhatikan pada gambar diatas, jumlah kotak berwarna ungu, yang merupakan simpul yang perlu dikunjungi sebelum menemukan solusi jumlahnya menurun sangat drastis dibanding sebelumnya. Hal ini bisa membuktikan, bahwa optimasi dengan QuadTree ini sangatlah efektif untuk membuat persoalan pathfinding ini menjadi lebih mangkus.

IV. KESIMPULAN

Struktur QuadTree merupakan sebuah struktur yang sangat menarik, dan memiliki banyak sekali kegunaan. Pada bidang game, struktur ini bisa digunakan untuk melakukan optimasi terhadap persoalan melakukan *collision detection* serta *pathfinding*. Optimasi menggunakan QuadTree ini terbukti cukup efektif, dan mampu menghasilkan solusi yang jauh lebih mangkus dibanding solusi tanpa optimasi. Untuk itu penggunakan QuadTree untuk menyederhanakan persoalan-persoalan tersebut merupakan sebuah hal yang sangat menarik untuk dipertimbangkan.

REFERENSI.

[1] http://blog.notdot.net/2009/11/Damn-Cool-Algorithms-Spatial-indexing-with-Quadtrees-and-Hilbert-Curves, diakses pada tanggal 16 desember pukul 10.00 WIB

[2] http://www.ai-blog.net/archives/000091.html, diakses pada tanggal 16 desember pukul 10.00 WIB

[3] http://gamedev.tutsplus.com/tutorials/implementation/quick-tip-use-quadtrees-to-detect-likely-collisions-in-2d-space/, diakses pada tanggal 16 desember pukul 10.00 WIB

[4] http://www.ai-blog.net/archives/000091.html , diakses pada tanggal 20 desember pukul 22.00 WIB

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, 16 Desember 2012

Muhammad Afif Al-hawari (13510020)