Optimasi *Collision Detection* Menggunakan Quadtree

Ecky Putrady 13508004
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
if18004@students.if.itb.ac.id

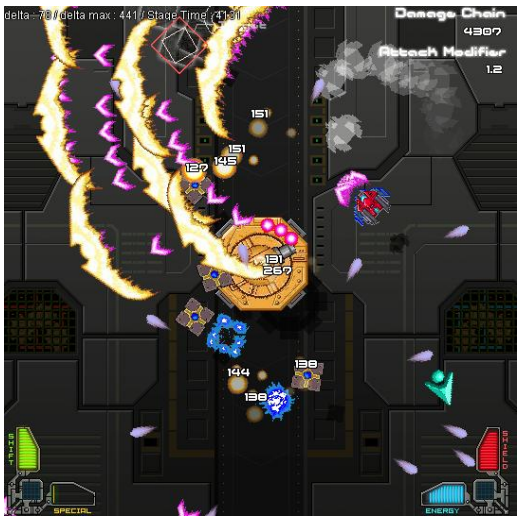
***Abstract*—*Collision Detection* atau deteksi tumbukan adalah suatu hal yang mendasar dalam bidang *game programming*. Hampir sebagian besar game membutuhkan hal ini sebagai inti dari *game mechanic*-nya.**

Pada karya tulis ini, penulis membahas bagaimana memodelkan masalah *collision detection* pada ruang dua dimensi dan menyelesaikannya menggunakan algoritma brute force. Selain itu, penulis juga membahas bagaimana cara melakukan optimasi dengan menggunakan quadtree pada masalah di atas sehingga dapat menyelesaikan dengan lebih baik.

***Index Terms*—Brute Force, Collision Detection, Divide and Conquer, Quadtree, Game Programming**

I. PENDAHULUAN

Collision Detection adalah suatu topik yang membahas tentang bagaimana cara mengetahui objek-objek apa saja yang bersentuhan satu sama lain dalam bidang koordinat 2 dimensi ataupun 3 dimensi. Aplikasi dari *collision detection* ini sangat banyak dijumpai pada game programming. Banyak sekali game yang menggunakan *collision detection* sebagai penunjang dari mekanika game. Sebagai contoh, pada game tembak-tembakan pesawat, atau yang umumnya disebut sebagai SHMUP (*Shoot 'em Up*), *collision detection* digunakan untuk mengetahui apakah peluru lawan mengenai pemain.



Gambar 1. Contoh game SHMUP, Cube Colossus

Dengan mengetahui apakah ada peluru yang bersentuhan dengan pesawat pemain, darah dari pemain dapat berkurang dan apabila pemain kehabisan darah, permainan akan berakhir.

Selain pada game, *collision detection* juga dapat diterapkan pada program simulasi fisika. Dengan mengetahui adanya objek-objek yang bersentuhan atau bertumbukan, maka pergerakan objek-objek tersebut berikutnya dapat dihitung dengan menggunakan rumus tumbukan.

Pada makalah ini, penulis memfokuskan masalah *collision detection* pada ruang dua dimensi.

II. PEMODELAN MASALAH

Seperti yang telah dijelaskan sebelumnya, *collision detection* adalah topik yang membahas bagaimana cara mengetahui objek-objek apa saja yang bersentuhan dalam bidang koordinat tertentu. Objek-objek ini bisa saja memiliki bentuk yang sangat bervariasi. Pada gambar 1 di atas terlihat bahwa objek-objek pada game memiliki bentuk yang bervariasi, ada yang berbentuk kotak, segi-n, sampai bentuk pesawat pemain yang sangat mendetail. Untuk mempercepat proses pada *collision detection*, umumnya objek-objek ini direpresentasikan secara logik dengan bentuk primitif seperti segiempat dan lingkaran (jika pada koordinat dua dimensi), atau kubus dan bola (jika pada koordinat tiga dimensi). Bentuk primitif yang merepresentasikan objek ini biasa disebut sebagai **Bounding Box** atau **Bounding Circle**.

Gambar 2. Representasi sederhana dengan menggunakan segiempat dan lingkaran.

Suatu objek dapat dikatakan bersentuhan apabila representasi sederhana ini saling bersentuhan. Pada representasi dengan lingkaran, pengecekan dapat dilakukan dengan sangat mudah. Untuk mengetahui apakah kedua objek bersentuhan, cukup menghitung jarak antara kedua objek itu, jika jaraknya kurang dari jumlah radius masing-masing objek, maka objek-objek tersebut

dapat dikatakan saling bersentuhan. Pengecekan dengan menggunakan representasi segiempat juga tidak sulit untuk dilakukan.

Gambar 3. Objek dalam keadaan bersentuhan (*collision*)

Pengecekan dengan bentuk primitif ini tidak selalu menjamin bahwa kedua objek tersebut memang benar bersentuhan, karena tidak semua objek memiliki bentuk yang mendekati bentuk primitif tersebut. Seringkali hal ini diabaikan, sebab dalam game pada dunia nyata, pemain jarang sekali menyadari hal yang mendetail seperti itu. Namun, jika diperlukan akurasi yang tinggi, suatu metode yang berkaitan dengan perbandingan nilai *alpha* pada gambar objek dapat digunakan agar akurasi mencapai 100%. Metode ini juga masih tetap menggunakan representasi primitif seperti yang telah dijelaskan sebelumnya. Metode ini tidak akan dibahas pada makalah ini, jika tertarik, silahkan mencari tentang **pixel-perfect collision detection**.

Jadi, setiap objek, serumit apapun bentuknya, dapat direpresentasikan dengan bentuk primitif. Dengan menggunakan bentuk primitif ini, pencarian objek-objek mana yang sedang bersentuhan dapat dengan mudah dilakukan.

III. METODE BRUTE FORCE

Brute force adalah suatu bentuk algoritma yang lempang (straightforward) dalam memecahkan suatu masalah. Brute force biasanya didasarkan pada pernyataan masalah dan pemecahan masalah secara intuitif ataupun sesuai konsep.

Algoritma brute force seringkali dikatakan tidak cerdas, karena caranya yang naif dan membutuhkan banyak langkah untuk dapat menyelesaikan suatu masalah. Namun, metode brute force hampir dapat menyelesaikan semua masalah, walaupun kadang membutuhkan waktu yang tidak dapat diterima. Sulit untuk menunjukkan masalah yang tidak dapat diselesaikan dengan brute force. Selain itu, brute force juga mudah untuk dimengerti dan diimplementasikan.

Pada masalah yang sedang dibahas ini, pencarian objek-objek mana saja yang saling bersentuhan dapat kita temukan langkahnya secara intuitif : untuk setiap objek yang ada, lakukan pengecekan bounding box (atau bounding circle) dengan setiap objek lainnya. Atau dengan pseudo-code, dapat dituliskan seperti ini :

```
for each(obj1:object in world) :
      for each(obj2:object in world) :
            if(obj1.isCollide(obj2)) :
                  do_something()
```

Kompleksitas dari algoritma ini adalah polynomial, yaitu n^2, untuk n adalah jumlah dari objek yang ada. Kompleksitas dengan tipe seperti ini sangat tidak diharapkan, karena semakin banyaknya jumlah objek, maka jumlah komputasi yang diperlukan akan bertambah secara eksponensial. Untuk suatu aplikasi yang real-time atau cepat mengalami perubahan, seperti game, hal ini dapat menjadi suatu masalah.

Tentu saja perhitungan ini masih dapat dioptimasi. Pertama, kita tidak perlu melakukan *collision detection* pada objek yang sama. Sehingga, pada masalah dengan jumlah objek sama dengan n, kita mendapati bahwa hasil pengecekan pasti tetap valid walaupun pengecekan dilakukan sebanyak n^2-n kali.

Selanjutnya kita dapat melakukan optimasi lagi, yakni, jika objek A menyentuh objek B, maka objek B juga dapat dikatakan menyentuh A. Sehingga kita dapat mengurangi jumlah perbandingan mencapai setengahnya. Sekarang, jumlah pengecekan dilakukan sebanyak $(n^2-n)/2$ kali. Namun, tetap saja berkisar pada $O(n^2)$. Algoritma brute force hanya dapat dioptimasi sampai sini.

IV. METODE QUADTREE

Collision detection menggunakan brute force, seperti yang telah dibahas di atas, membutuhkan waktu komputasi yang semakin meningkat seiring dengan bertambahnya jumlah objek yang akan dicek. Untuk mengurangi jumlah pengecekan, kita harus sebisa mungkin melakukan pengecekan hanya pada objek-objek yang mungkin bersentuhan sehingga objek-objek yang sudah pasti tidak bersentuhan tidak perlu dicek. Quadtree adalah salah satu metode yang menggunakan konsep ini.

Quadtree pada dasarnya adalah struktur data pohon yang setiap simpulnya memiliki anak berjumlah tepat empat. Quadtree digunakan untuk membagi ruang dua dimensi menjadi empat region. Pada ruang tiga dimensi, struktur yang mirip dengan quadtree digunakan, yakni octree. Quadtree pada dasarnya menggunakan strategi *divide and conquer* dalam prosesnya.

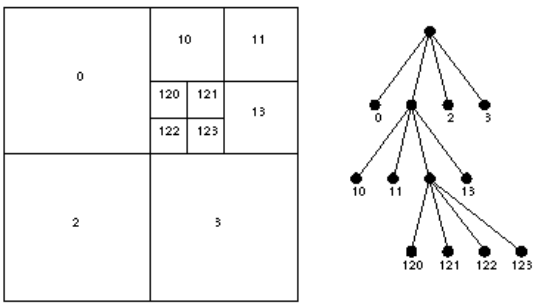


Figure 3.6 Quadtree.

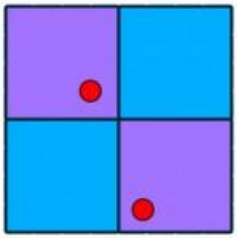
Gambar 4. Quadtree

Pada *collision detection* menggunakan quadtree digunakan sebagai mekanisme awal untuk memastikan secara akurat objek-objek mana saja yang mungkin bersentuhan. Setelah mengetahui hal tersebut, selanjutnya digunakan algoritma brute force seperti di atas hanya pada objek-objek yang mungkin bersentuhan. Dengan cara ini, jumlah pengecekan dapat ditekan menjadi lebih sedikit.

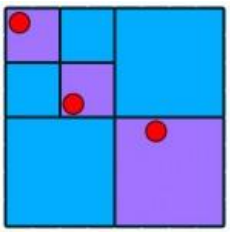
Quadtree bekerja dengan cara membagi ruang dua dimensi menjadi empat buah region (yaitu simpul) yang sama besarnya ketika suatu objek memasuki ruangan tersebut, dengan aturan yang sederhana : Untuk setiap simpul pada quadtree, jika terdapat lebih dari k buah objek di dalam simpul, simpul itu akan dibagi lagi. Dengan besar dari k biasanya angka kecil, yakni 1 sampai 5.

Ketika sebuah simpul dibagi, ia terbagi menjadi empat buah simpul lagi dengan cara membagi ruang secara horizontal dan vertikal secara merata. Objek pada simpul yang tepat berada dalam ruang yang dibentuk simpul baru, akan dipindahkan ke simpul baru tersebut. Objek pada simpul yang tidak tepat berada di dalam simpul yang baru, misalnya berada pada perpotongan atau sisi, akan tetap berada simpul induk. Dengan cara inilah setiap objek didaftarkan pada region-region (simpul-simpul) pada quadtree.

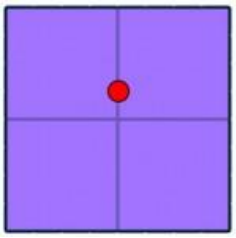
Berikut ini adalah ilustrasi dari pembagian ruang dalam quadtree :



Gambar 4. Objek yang tepat berada di dalam region (simpul) akan dimasukkan ke dalam region tersebut.



Gambar 5. Region yang memiliki objek lebih dari n (n = 1) akan terbagi lagi menjadi region yang lebih kecil



Gambar 6. Objek yang berada tidak tepat di dalam region akan tetap berada ada simpul induknya.

Pada saat waktunya ada permintaan untuk *collision detection*, quadtree akan menerima input koordinat bounding box. Dari koordinat ini akan ditentukan di simpul mana sajakah bounding box ini terletak. Setelah didapatkan simpulnya, barulah *collision detection* dilakukan pada semua objek yang terkandung di dalam simpul tersebut. Dengan cara seperti ini, pengecekan hanya akan dilakukan pada objek-objek yang mungkin bersentuhan dan akan mengurangi jumlah pengecekan secara signifikan.

Secara ringkas, alur penggunaan quadtree dalam *collision detection* adalah seperti ini :

1. Membangun quadtree dengan cara mengindex setiap objek
2. Untuk setiap objek, carilah simpul-simpul quadtree yang bersentuhan dengan boundig box objek tersebut, ambillah setiap objek yang terdapat simpul-simpul tersebut
3. Lakukan *collision detection* dengan brute force pada objek-objek yang didapat

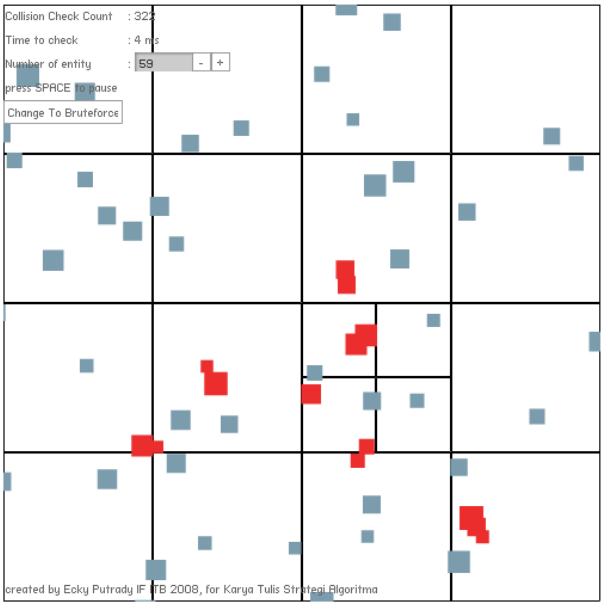
```
for each(obj:Object in world) :
  quadtree.index(obj)

for each(obj1:Object in world) :
  colObjects = quadtree.query(obj1.getBoundingBox())
  for each(obj2:Object in colObjects) :
    if(obj1.isCollide(obj2)) :
      do_something()
```

Jika kita menggunakan quadtree dan melakukan *collision detection* terhadap seluruh objek yang ada di dalam quadtree tersebut, dengan jumlah objek n. Setiap pencarian simpul menghasilkan log n. Maka kompleksitas *collision detection* turun dari n^2 menjadi n log n.

V. IMPLEMENTASI

Untuk menguji dan memperlihatkan kemangkusan penggunaan quadtree, maka penulis membuat sebuah demonstrasi sederhana menggunakan flash. Dalam demonstrasi ini, pengguna dapat melihat perbedaan antara penggunaan quadtree dan brute force. demonstrasi ini dapat dijalankan secara online melalui pranala berikut : http://megaswf.com/serve/80821/ . Untuk menjalankan demonstrasi ini, diperlukan adobe player versi 9 ke atas. Secara teknis, jika anda bisa menonton video dari youtube, biasanya anda dapat menjalankan demonstrasi ini.



Gambar 7. Demonstrasi quadtree

Program juga dilengkapi dengan ilustrasi pembuatan quadtre sebagai informasi tentang quadtree yang sedang dibentuk. Quadtree yang digunakan pada program ini mendefinisikan 5 sebagai banyaknya objek yang dimiliki agar mulai membagi lagi. Jumlah 5 dipilih karena setelah beberapa kali mencoba, menggunakan 5 menghasilkan optimasi yang baik untuk masalah *collision detection*.

Objek-objek yang menjadi perhatian akan berwarna merah apabila ia bersentuhan dengan objek lainnya. Jumlah dari objek-objek ini dapat diatur jumlahnya sehingga pengguna dapat bereksperimen dengan jumlah yang berbeda-beda.

Dari hasil demonstrasi ini, rata-rata jumlah pengecekan dan waktu pengecekan yang didapat adalah sebagai berikut :

Tabel Jumlah Rata-Rata Pengecekan *Collision*

Jumlah Objek	*Brute Force*	*Quadtree*
50	2500	400
100	10000	1400
500	250000	32000
1000	1000000	120000

Tabel Waktu Rata-Rata Pengecekan *Collision*

Jumlah Objek	*Brute Force*	*Quadtree*
50	2 ms	3 ms
100	8 ms	8 ms
500	150 ms	48 ms
1000	800 ms	150 ms

VI. Analisis

Berdasarkan penjabaran di atas dapat diambil kesimpulan bahwa penggunaan quadtree menghasilkan kompleksitas algoritma (notasi big O) yang lebih baik. Pada algoritma brute force kompleksitasnya adalah $O(n^2)$, sedangkan dengan penggunaan quadtree, kompleksitasnya adalah O(n log n).

Berdasarkan hasil demonstrasi, diperoleh bahwa menggunakan quadtree memang dapat secara signifikan mempercepat waktu proses *collision detection* serta mengurangi jumlah pengecekan namun memberikan hasil (objek-objek yang bersentuhan) tetap valid. Penggunaan quadtree, berdasarkan table, mampu memangkas jumlah perbandingan hingga mencapai 10% sampai 20% dari jumlah perhitungan brute force. Dari segi waktu komputasi, quadtree-pun mampu menguranginya mencapai 20% dari waktu yang dibutuhkan oleh brute force. Namun, pada jumlah objek yang kecil, seperti 50, penggunaan quadtree justru memperlama waktu komputasi. Pada kasus terburuk quadtree, yakni seluruh objek berada pada simpul yang sama, waktu perhitungannya juga menjadi lebih buruk daripada brute force. Kedua hal ini terjadi karena quadtree memiliki *overhead* pada pembangunan pohonnya.

Kompleksitas yang baik tidak selalu menghasilkan komputasi yang lebih cepat pada praktiknya, karena masih ada faktor-faktor lain yang mempengaruhi kecepatan, termasuk pengimplementasian algoritma itu sendiri. Quadtree memiliki *overhead* komputasi pada pembangunan pohonnya, ditambah lagi *effort* untuk mengimplementasikan quadtree lebih besar daripada brute force saja.

Karena adanya *overhead* ini, waktu pemrosesan akan menjadi lebih lama dibandingkan menggunakan brute force biasa pada jumlah objek yang sedikit. Namun, tentu saja jika objek yang diperhitungkan pada *collision detection* mencapai jumlah yang banyak, penggunaan quadtree bisa sangat membantu performa dari *collision detection*, sesuai yang ditunjukkan dari hasil demonstrasi.

VII. Kesimpulan

Dari pemaparan di atas, dapat disimpulkan bahwa :

1. Objek-objek yang menjadi perhatian untuk *collision detection*, serumit apapun bentuk yang terlihat, dapat dimodelkan menjadi bentuk yang primitif untuk mempermudah *collision detection*.
2. Algoritma brute force dapat dilakukan untuk menyelesaikan masalah *collision detection*, sekali lagi brute force mempertegas bahwa tidak ada masalah yang tidak bisa diselesaikan oleh brute force.
3. Algoritma brute force memakan waktu komputasi yang besar.
4. Optimasi pada *collision detection* dapat dilakukan dengan hanya mengecek objek-objek mana saja yang masih mungkin bersentuhan, mengeliminasi pengecekan dengan objek-objek yang tidak mungkin

bersentuhan.

5. Quadtree adalah salah satu metode untuk mengoptimasi *collision detection* dengan mengurangi waktu komputasi menjadi O(n log n) dan memiliki *overhead* pada pembangunan quadtree itu sendiri.
6. Quadtree memang benar dapat mengoptimasi *collision detection* secara signifikan, berdasarkan hasil percobaan, sebsar 80% - 90%.
7. Gunakan quadtree hanya jika jumlah objek yang akan dicek banyak, di mana waktu komputasi brute force saja tidak dapat diterima untuk memproses *collision detection.*

REFERENCES

[1] http://en.wikipedia.org/wiki/Quadtree diakses pada 2/12/2010 20:47.

[2] http://www.drdobbs.com/184409694;jsessionid=YLTUPB4YTOJ3LQE1GHPSKHWATMY32JVN?pgno=3 diakses pada 2/12/2010 20:47.

[3] http://www.kyleschouviller.com/wsuxna/quadtree-source-included/ diakses pada 2/12/2010 20:47.

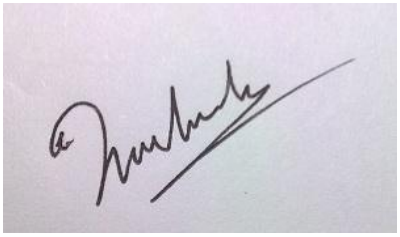
[4] http://www.informatika.org/~rinaldi/Stmik/2006-2007/Algoritma%20Brute%20Force%20(Bagian%201).pdf diakses pada 2/12/2010 20:47.

[5] http://www.informatika.org/~rinaldi/Stmik/2006-2007/Algoritma%20Brute%20Force%20(Bagian%202).pdf diakses pada 2/12/2010 20:47.

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, 5 Desember 2010



Ecky Putrady 13508004