Penggunaan k-d *Tree* dalam *Ray Tracing*

Stefanus Ardi Mulia 13517119
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
stefanus.ardimulia@students.itb.ac.id

***Abstrak— Pengguanaan ray tracing* pada aplikasi-aplikasi yang membutuhkan *real-time rendering* seperti gim video selalu dihindari karena penggunaannya membutuhkan sumber daya komputasi yang tidak kecil sehingga perlu waktu yang cukup lama untuk menghasilkan hanya satu *frame* yang dapat ditampilkan di layar. Makalah ini akan membahas salah satu cara untuk meminimalkan sumber daya komputasi yang dibutuhkan hingga sekitar $O(\log n)$ untuk mengaplikasikan *ray tracing* sehingga dapat mengurangi waktu yang dibutuhkan untuk melakukan hal tersebut juga.**

***Kata Kunci—ray tracing*, pohon, k-d *tree*, grafika.**

I. PENDAHULUAN

Untuk menghasilkan gambar yang realistis, kita perlu menyimulasikan dengan akurat semua interaksi antara cahaya dengan permukaan obyek. Komputasi dari interaksi-interaksi ini disebut *light transport,* atau *global illumination,* dalam grafika komputer. Menyimulasikan *light transport* telah menjadi area penelitian aktif beberapa dekade ini [6]. Metode populer untuk menyelesaikan masalah *light transport* adalah dengan menggunakan *ray tracing*. Prosedur dasar dalam *ray tracing* adalah dengan melakukan komputasi titik potong antara berkas cahaya dan obyek. Refleksi maupun refraksi disimulasikan dengan membuat jenis-jenis berkas cahaya yang berbeda tergantung dengan sifat material dari benda tersebut.

Permasalahan dari penggunaan *ray tracing* untuk memecahkan masalah *global illumination* adalah komputasi yang perlu di lakukan sangat berat. Sebagai contoh, pembuatan suatu gambar menggunakan *ray tracing* dapat menghabiskan beberapa jam. Oleh karena hal tersebut, kebanyakan aplikasi komputer interaktif yang menampilkan grafik seperti gim video tidak menggunakan *ray tracing* karena pengaplikasiannya terlalu lambat untuk penggunaan yang interaktif [7]. Sebagai gantinya, aplikasi-aplikasi tersebut menggunakan teknik *rasterization*. *Rasterization* membuat suatu gambar dengan memproyeksikan obyek ke layar dan teknik ini biasanya lebih cepat dari *ray tracing* karena dukungan percepatan langsung oleh perangkat keras yang melakukan *rendering*, dalam hal ini *GPU* atau *Graphics Processing Unit*. Meskipun begitu *rasterization* tidak cocok digunakan untuk memecahkan masalah *global illumination.* Dengan mulai berkembangnya perangkat keras yang mendukung percepatan proses komputasi *ray tracing* seperti lini produk RTX oleh perusahaan pengembang perangkat keras dan lunak grafis NVIDIA, penggunaan *ray tracing* sebagai cara membuat gambar yang realistis dalam waktu *real-time* bukan lagi impian belaka.

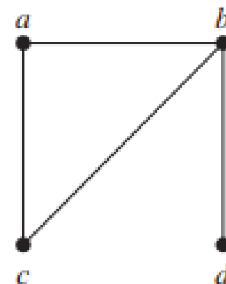
Untuk mendukung penggunaan *ray tracing* pada *rendering* gambar yang realistis diperlukan juga suatu algoritma dan struktur data yang dapat mengoptimalkan komputasi dari *ray tracing* itu sendiri. Makalah ini akan membahas salah satu cara untuk meminimalkan sumber daya komputasi yang dibutuhkan untuk *ray tracing*.

Gambar 1: Contoh gambar hasil *render* menggunakan *ray tracing [8].*

II. DASAR TEORI

A. *Graf*

Sebuah graf (*graph*) G didefinisikan sebagai pasangan himpunan (V, E), ditulis dengan notasi $G = (V, E)$, dengan V merupakan himpunan tidak kosong dari simpul-simpul (*vertices atau nodes*) dan E adalah himpunan sisi (*edges* atau *arcs*) yang menghubungkan sepasang simpul [1].



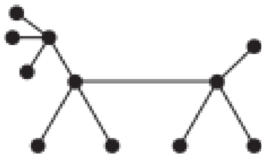
Gambar 2: Contoh representasi graf. [2]

B. *Pohon*

Sebuah pohon (*tree*) didefinisikan sebagai graf terhubung tak-berarah yang tidak mengandung sirkuit.

Misalkan sebuah graf tak-berarah sederhana $G = (V, E)$ yang memiliki simpul berjumlah n, maka semua pernyataan di bawah ini adalah ekuivalen:

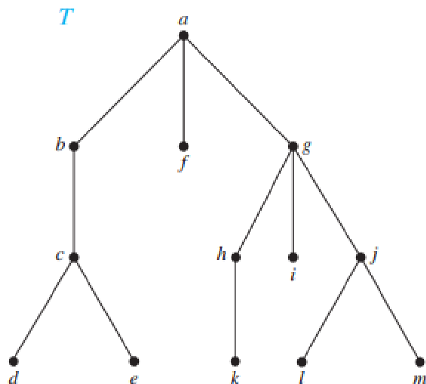
1. G merupakan sebuah pohon.
2. Setiap pasang simpul di dalam G terhubung dengan lintasan tunggal.
3. G terhubung dan memiliki $m = n - 1$ buah sisi.
4. G tidak mengandung sirkuit dan memiliki $m = n - 1$ buah sisi.
5. G tidak mengandung sirkuit dan penambahan satu sisi pada graf akan membuat hanya satu sirkuit.
6. G terhubung dan semua sisinya adalah jembatan (jembatan adalah sisi yang bila dihapus menyebabkan graf terpecah menjadi dua komponen). [1]



Gambar 3: Contoh representasi pohon. [2]

C. *Pohon Berakar*

Sebuah pohon dapat merupakan sebuah pohon berakar (*rooted tree*) jika salah satu simpulnya diperlakukan sebagai akar dan simpul-simpul lain yang terhubung dengannya diberi arah menjauh dari akar tersebut. [1]



Gambar 4: Contoh representasi pohon berakar. [3]

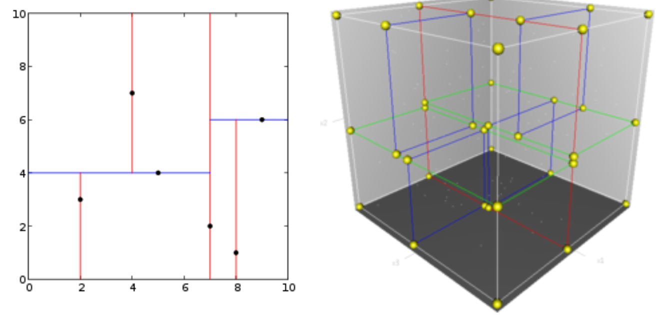
Misalkan sebuah pohon T dan v adalah sebuah simpul di T selain akarnya, **orang tua (*parent*)** dari v adalah simpul unik u sehingga ada sisi berarah dari u ke v. Jika u adalah orang tua dari v, maka v merupakan **anak (*child*)** dari u. Simpul yang memiliki orang tua yang sama disebut **saudara kandung (*siblings*)**. **Leluhur (*ancestors*)** dari sebuah simpul selain akar adalah simpul-simpul yang berada di lintasan dari akar ke simpul ini, tidak termasuk simpul itu sendiri dan termasuk akar. **Keturunan (*descenant*)** dari sebuah simpul v adalah simpul-simpul yang memiliki v sebagai leluhurnya.

Simpul dari sebuah pohon berakar disebut **daun (*leaf*)** jika simpul tersebut tidak mempunyai anak. Simpul-simpul yang mempunyai anak disebut **simpul dalam (*internal vertices*)**. Akar merupakan simpul dalam kecuali simpul tersebut merupakan simpul satu-satunya di pohon graf tersebut, dalam hal ini merupakan daun.

Jika a merupakan sebuah simpul pada sebuah pohon, sebuah upapohon dengan a sebagai akarnya adalah upagraf dari pohon yang terdiri dari a dan semua keturunannya dan semua sisi yang terhubung dengan keturunan-keturunan tersebut. [2]

D. *k-d Tree*

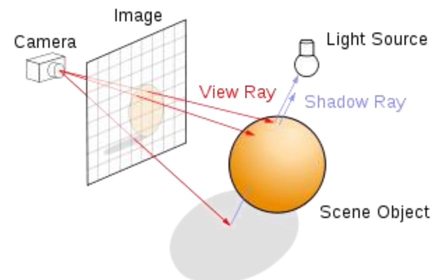
k-d Tree atau *k-dimensional tree* merupakan suatu pohon biner rekursif dengan tiap daunnya merupakan suatu titik k-dimensi. Setiap simpul bukan daun dapat dianggap sebagai sebuah *hyperplane,* bidang k-1 dimensi, yang membagi suatu ruang k-dimensi menjadi dua bagian yang disebut *half-space*. Titik yang berada di sebelah kiri dari *hyperplane* ini direpresentasikan oleh upapohon kiri dari simpul tersebut dan titik yang berada di sebelah kanan dari *hyperplane* ini direpresentasikan oleh upapohon kanan. [3]



Gambar 5: Visualisasi k-d *tree* 2 dimensi (kiri) dan 3 dimensi (kanan). *Sumber: https://en.wikipedia.org/wiki/K-d_tree.*

E. *Ray Tracing*

Ray tracing merupakan salah satu teknik *rendering* untuk membentuk suatu gambar dengan menelusuri lintasan simulasi berkas cahaya antara bidang gambar dengan suatu objek dalam suatu *scene* [4]. *Ray tracing* membutuhkan komputasi akan perpotongan antara berkas cahaya yang dipancarkan dengan suatu obyek. Performa relatif dari berbagai struktur terakselerasi telah diteliti secara luas. Havran membandingkan banyak jenis struktur terakselerasi pada berbagai *scene* dan menyimpulkan bahwa k-d *tree* merupakan struktur terakselerasi serbaguna terbaik untuk *ray tracing* menggunakan CPU. [5]

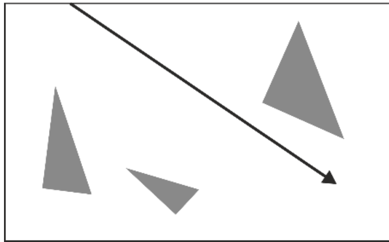


Gambar 6: Algoritma *ray tracing* membangun sebuah gambar dengan membentangkan berkas cahaya ke dalam *scene*. *Sumber: https://en.wikipedia.org/wiki/Ray_tracing_(graphics).*

III. METODE

Misalkan suatu *scene* 2 dimensi yang terdiri atas bidang

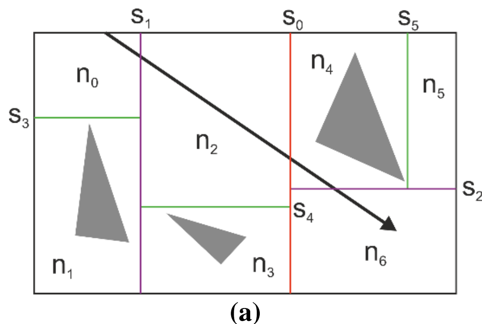
kosong yang luas dan beberapa segitiga yang merepresentasikan obyek terletak pada bidang tersebut. Jika dilakukan *ray tracing* sederhana, maka setiap berkas cahaya yang melewati *scene* tersebut akan mengecek perpotongan dengan semua obyek (segitiga) yang berada di *scene* tersebut seperti pada gambar 7.



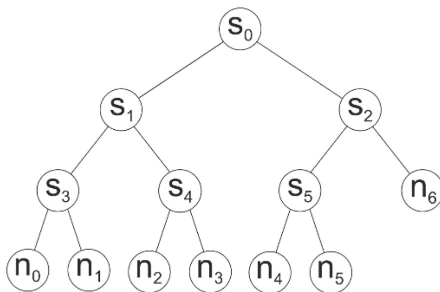
Gambar 7: Panah merepresentasi berkas cahaya. Segitiga yang diarsir merepresentasikan obyek yang dicek perpotongannya oleh berkas cahaya.

Algoritma *ray* tracing yang mengecek semua obyek untuk setiap berkas cahaya yang dipancarkan ini akan menjadi semakin boros sumber daya komputasi jika kita memutuskan untuk menambah jumlah berkas cahaya atau obyek karena algoritma ini memerlukan kompleksitas $O(n^2)$.

Untuk mempercepat pengecekan perpotongan antara berkas cahaya dengan objek kita dapat mengurangi obyek yang diperiksa oleh tiap berkas cahaya yang melintas. Pada kasus inilah k-d *tree* dapat digunakan untuk memartisi obyek pada *scene* sesuai dengan posisinya.



(a)



(b)

Gambar 8: **(a)** contoh *scene* yang telah dipartisi dengan menggunakan algoritma k-d *tree.* **(b)** tree yang dihasilkan dengan algoritma k-d *tree*.

Pada gambar 8(a) kita dapat melihat bahwa *scene* pada gambar 7 dapat dipartisi menggunakan algoritma k-d *tree* dengan setiap daun pada pohon di gambar 8(b) merepresentasikan potongan area berbentuk persegi panjang yang selaras dengan sumbu bidang tersebut dan setiap simpul dalam pada pohon tersebut merepresentasikan bidang *hyperplane* yang memotong area *scene* tersebut menjadi 2 bagian anak pohon.

A. *Pendekatan standar*

Penggunaan algoritma pengunjung (*traversal*) k-d *tree* yang rata-rata digunakan untuk *ray tracing* ditunjukkan pada gambar 9. Algoritma tersebut menerima masukan sebuah pohon dan sebuah berkas cahaya kemudian mencari primitif pertama dalam pohon yang berpotongan dengan berkas cahaya tersebut. Pohon ini dikunjungi mulai dari akarnya dan sebuah tumpukan (*stack*) digunakan sebagai lis simpul yang belum dikunjungi terurut berdasarkan prioritas. Setiap simpul pada tumpukan lebih dekat dengan sumber berkas cahaya dibandingkan simpul lain di bawahnya dan simpul yang sedang dikunjungi lebih dekat dari pada semua simpul di dalam tumpukan. Sebuah interval (*tmin,tmax*) membatasi bagian berkas cahaya berdasarkan bagian berkas yang berpotongan dengan simpul yang sedang dikunjungi. [9]

```
kd-search( tree, ray )
   (global-tmin, global-tmax) = intersect( tree.bounds, ray )
   search-node( tree.root, ray, global-tmin, global-tmax )

search-node( node, ray, tmin, tmax )
   if( node.is-leaf )
      search-leaf( node, ray, tmin, tmax )
   else
      search-split( node, ray, tmin, tmax )

search-split( split, ray, tmin, tmax )
   a = split.axis
   thit = ( split.value - ray.origin[a] ) / ray.direction[a]
   (first, second) = order( ray.direction[a], split.left, split.right )

   if( thit >= tmax or thit < 0 )
      search-node( first, ray, tmin, tmax )
   else if( thit <= tmin )
      search-node( second, ray, tmin, tmax )
   else
      stack.push( second, thit, tmax )
      search-node( first, ray, tmin, thit )

search-leaf( leaf, ray, tmin, tmax )
   // search for a hit in this leaf
   if( found-hit and hit.t < tmax )
      succeed( hit )
   else
      continue-search( leaf, ray, tmin, tmax )

continue-search( leaf, ray, tmin, tmax )
   if( stack.is-empty )
      fail()
   else
      (n, tmin, tmax) = stack.pop()
      search-node( n, ray, tmin, tmax )
```

Gambar 9: *Pseudocode* dari algoritma *traversal* k-d *tree* standar. Perhatikan semua pemanggilan fungsi lain dilakukan pada akhir dari suatu fungsi sehingga algoritma tersebut dapat dilakukan secara berulang (*iteratively*) [9].

Ketika sebuah simpul dalam ditemui ketika proses *traversal* simpul, interval (*tmin,tmax*) dari berkas cahaya tersebut di tentukan dengan bidang pemotong yang direpresentasikan oleh simpul tersebut. Jika seluruh interval terletak pada salah satu bagian dari bidang yang terpotong, maka *traversal* yang

dilakukan akan berpindah ke anak dari simpul tersebut. Namun, jika interval terletak diantara 2 bidang yang berbeda maka *traversal* akan dilanjutkan pada anak pertama yang dilewati oleh berkas cahaya sementara anak kedua dimasukkan (*push*) ke dalam tumpukan beserta interval (*tmin,tmax*) yang sesuai. Dengan begini *traversal* berlanjut menurun pada pohon dengan sekali-sekali memasukkan simpul serta intervalnya ke dalam tumpukan hingga suatu simpul daun tercapai.

Jika berkas cahaya berpotongan dengan salah satu primitif di dalam daun pada interval (*tmin,tmax*), maka perpotongan yang terdekat di dalam daun dipastikan merupakan perpotongan pertama sepanjang berkas cahaya dan *traversal* dihentikan, mengembalikan hasil ini. Jika tidak ada perpotongan yang ditemukan, maka ambil (*pop)* sebuah simpul serta intervalnya dari tumpukan dan lanjutkan pencarian. [9]

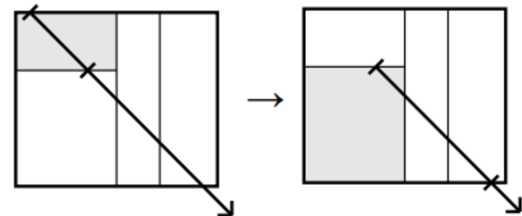
Walaupun performa terburuk (*worst case*) dari algoritma ini adalah $O(n)$ dengan n adalah jumlah daun pada pohon, performa harapan dari implementasi algoritma ini pada *scene* nyata adalah $O(\log n)$. Pada kemungkinan terburuk, sebuah berkas cahaya mengunjungi jumlah simpul linear dengan besar pohon tersebut. Namun pada kenyataannya dapat diasumsikan bahwa kebanyakan berkas cahaya akan menemukan sebuah perpotongan pada simpul-simpul daun pertama yang perlu dikunjungi [5], sehingga performa harapannya proporsional dengan tinggi dari pohon tersebut yang biasanya logaritmik.

B. *Algoritma kd-Restart*

Algoritma *traversal* kd-*restart* mengubah algoritma *traversal* kd-*treei* standar dengan menghapus semua operasi yang melibatkan tumpukan. Jika kita menghapus operasi *push* dari fungsi *search-split* pada gambar 9, maka proses *traversal* akan dilanjutkan langsung ke daun pertama yang dilewati oleh berkas cahaya. Dapat kita lihat dengan perubahan tersebut proses *traversal* hanya mengubah nilai dari *tmax* ketika sebelumnya satu simpul dan intervalnya dimasukkan ke tumpukan. Bahkan, nilai baru intuk *tmax* pada kasus ini persis sama dengan nilai yang kita masukkan ke dalam tumpukan sebagai *tmin*. Ini menunjukkan bahwa ketika kita mencapai sebuah daun, nilai dari *tmax* adalah tepat *tmax* global (ujung dari berkas cahaya) atau tepat nilai *tmin* (posisi berkas cahaya masuk ke daun selanjutnya). Kita dapat memanfaatkan fakta ini dengan memodifikasi fungsi *continue-search* seperti berikut:

```
continue-search( leaf, ray, tmin, tmax )
  if( tmax == global-tmax )
    fail()
  else
    tmin = tmax
    tmax = global-tmax
    search-node( tree.root, ray, tmin, tmax )
```

Ketika *traversal* dari algoritma kd-*restart* mencapai suatu simpul daun dan gagal menemukan perpotongan, fungsi ini akan memulai kembali pencarian pada akar dari pohon dengan nilai *tmin* yang dimajukan ke ujung dari daun sebelumnya. Pada gambar 10 dapat kita lihat daun pertama yang perpotongan dengan interval yang telah diubah adalah daun selanjutnya yang perlu dikunjungi. Dengan mengulang proses ini, kita akan dapat mengunjungi semua daun yang perpotongan dengan berkas cahaya dalam urutan yang sama dengan *traversal* k-d *tree* standar. Namun, untuk setiap daun yang dikunjungi dengan *traversal* yang dimulai dari akar pohon, sumber daya komputasi yang terpakai untuk mengunjungi *m* buah daun adalah $O(m \cdot h)$ dengan h merupakan tinggi dari pohon tersebut. Pada pohon yang seimbang hal ini menyebabkan operasi dengan kasus terburuk $O(\log n)$. Tetapi jika kita berasumsi bahwa jumlah rata-rata simpul daun yang dikunjungi dibatasi oleh suatu konstanta yang kecil, maka dapat diharapkan operasi ini hanya membutuhkan $O(\log n)$. [9]



Gambar 10: setelah gagal menemukan perpotongan dengan obyek pada suatu simpul daun, kd-*restart* memindahkan interval (*tmin,tmax*) ke depan. Perhatikan interval baru mulai pada daun selanjutnya yang perlu dikunjungi. [9]

C. *Algoritma kd-Backtrack*

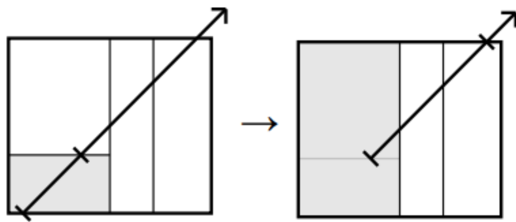
Algoritma kd-*backtrack* memodifikasi kd-*restart* untuk menjaga batas kemungkinan terburuk agar tetap linear dengan penambahan ruang penyimpanan tiap simpul. Pertama kita amati tiap simpul yang dimasukkan ke dalam tumpukan pada algoritma standar sebelumnya selalu anak lainnya dengan leluhur yang sama pada pohon biner kd-*tree*. Sehingga seharusnya mungkin untuk mencapai orang tua dari simpul yang berada di atas tumpukan dengan mengikuti serangkaian rantai simpul orang tua (informasi yang bisa kita simpan di dalam simpul-simpul pada sebuah pohon) dari simpul yang sedang dikunjungi.

Jika kita menggunakan kembali taktik memajukan *tmin* ke ujung dari daun terakhir yang dikunjungi, maka kita dapat mengenali orang tua yang tepat sebagai leluhur terdekat yang perpotongan dengan sisa interval (*tmin,tmax*). Gambar 11 mendemonstrasikan hal ini. Kita dapat melakukan pengecekan perpotongan ini menggunakan kotak pembatas (*bounding box*) yang disimpan oleh setiap simpul dalam. Kita dapat mengimplementasi pendekatan ini dengan fungsi *continue-search* yang telah dimodifikasi:

```
continue-search( leaf, ray, tmin, tmax )
  if( tmax == global-tmax )
    fail()
  else
    tmin = tmax
    tmax = global-tmax
    backtrack( leaf.parent, ray, tmin, tmax )

backtrack( split, ray, tmin, tmax )
  (t0,t1) = intersect( split.bounds, ray, tmin, 
tmax )
  if( no-intersection )
    backtrack( split.parent, ray, tmin, tmax )
  else
    search( split, ray, t0, t1 )
```

Algoritma kd-*backtrack* menambah ruang penyimpanan yang dibutuhkan untuk tiap simpul. Algoritma ini membutuhkan kotak-kotak pembatas disimpan di dalam simpul dalam dan juga rantai simpul orang tua di setiap simpul. Namun, tidak seperti kd-*restart*, pendekatan ini mempertahankan kompleksitas waktu kemungkinan terburuk asimtotik dari algoritma *traversal* standar. Untuk melihat ini, coba perhatikan bahwa setiap langkah mundur (*bactracking step*) tidak akan mengunjungi simpul yang tidak pernah dikunjungi saat pencarian menurun pada pohon. Selain itu, setiap simpul akan dikunjungi maksimal dua kali oleh proses langkah mundur, sekali dari anak sebelah kiri dan sekali lagi dari anak sebelah kanan. Sehingga kd-*backtarack* paling banyak 3 kali lebih lambat dari algoritma standar sebelumnya. [9]



Gambar 11: setelah mencari perpotongan pada sebuah simpul daun, kd-*backtrack* melanjutkan pencarian pada leluhur pertama yang berpotongan dengan interval (*tmin,tmax*) yang diberikan. [9]

IV. KESIMPULAN

Ray tracing merupakan proses pembangun gambar yang membutuhkan sumber daya komputasi yang sangat besar sehingga jarang digunakan untuk menampilkan gambar secara *real-time*. Menggunakan struktur data kd-*tree* dan algoritma *traversal*-nya (standar, kd-*restart*, maupun kd-*backtrack*) kita dapat meminimalkan sumber daya komputasi yang dibutuhkan oleh proses *ray tracing* ini. Penggunaan kd-*tree* ini sendiri juga cukup serba guna, karena implementasinya dapat dilakukan menggunakan CPU maupun GPU, terutama mengingat mulai berkembangnya perangkat keras yang mendukung akselerasi *ray tracing*.

V. UCAPAN TERIMA KASIH

Penulis bersyukur kepada Tuhan YME atas penyertaan-Nya dan berkat-Nya sehingga penulis dapat menyelesaikan makalah untuk mata kuliah Matematika Diskrit 2018 ini. Penulis ingin berterima kasih kepada Dra. Harlili M.Sc. sebagai dosen Matematika Diskrit 2018 kelas K2. Selain itu terima kasih juga untuk keluarga penulis yang telah memberi dukungan dan arahan demi kelancaran kuliah penulis di ITB.

REFERENSI

[1] R. Munir, *Matematika Diskrit,* edisi ketiga, revisi kelima. Bandung: Penerbit Informatika, 2014, bab 8-9.
[2] K.H. Rosen, *Discrete Mathematics and its Applications*, 7th ed. New York: McGraw-Hill, 2012, ch 10-11. .
[3] J. L. Bentley, "Multidimensional binary search trees used for associative searching," *Communications of the ACM*, Vol.18 – N0.9, September 1975.
[4] A. Appel, "Some Techniques for Shading Machine Rendering of Solids," AFIPS Conference proceedings, Vol 32, pp 37-45.
[5] V. Havran, "Heuristic Ray Shooting Algorithms," 2000.
[6] P. Dutre, K. Bala, P. Bekaert, P. Shirley, "Advanced Global Illumination," AK Peters Ltd, 2006.
[7] T. Hachisuka, "Ray Tracing on Graphics Hardware," 2011.
[8] I. Wald, T. J. Purcell, J. Schmittler, C. Benthin, P. Slusallek, "Realtime Ray Tracing and its use for Interactive Global Illumination," 2003.
[9] T. Foley, J, Sugerman, "KD-Tree Acceleration Structures for a GPU Raytracer," *Graphics Hardware, 2005,* pp 14-22.

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, 9 Desember 2018

Stefanus Ardi Mulia 13517119