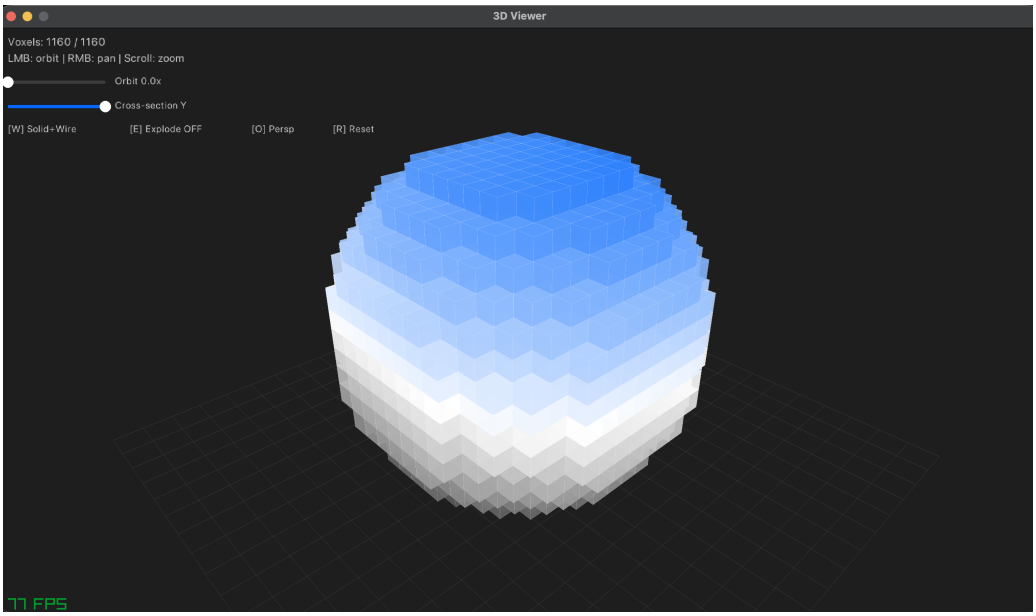
Tugas Kecil II — IF2211 Strategi Algoritma

3D Voxelization using Octree

March 28, 2026



Made Branenda Jordhy — 13524026

Valentino Daniel Kusumo — 13524104

School of Electrical Engineering and Informatics

Institut Teknologi Bandung

Semester II 2025/2026

Contents

1 Introduction **3**

2 Theoretical Background **4**
2.1 Voxelization 4
2.2 Octree 4
2.3 Divide and Conquer 4
2.4 Separating Axis Theorem (SAT) 5

3 Divide and Conquer Algorithm **6**
3.1 High-Level Steps 6
3.2 Pseudocode 7
3.3 Complexity Analysis 9

4 Implementation **11**
4.1 System Architecture 11
4.2 Program Flow 11
4.3 Implementation Details 12

5 Results and Analysis **13**
5.1 Test Case 1 13
5.2 Test Case 2 13
5.3 Test Case 3 14
5.4 Test Case 4 14
5.5 Test Case 5 15
5.6 Performance Summary 15

6 Bonus Features **17**
6.1 Concurrency 17
6.2 Interactive 3D Viewer 17

7 Conclusion **19**

Repository **21**

Lampiran **21**

1 Introduction

Dalam *computer graphic*, objek 3D itu umumnya disimpan sebagai *polygon mesh*. Format ini fleksibel dan ringan tetapi tidak selalu cocok untuk kebutuhan tertentu. Seperti yang kita tau, *grid based physics simulations*, *3D printing*, hingga game seperti Minecraft, itu semuanya membutuhkan representasi volumetrik yang lebih terstruktur. Di sinilah proses *voxelization* digunakan yaitu mengubah *polygon mesh* menjadi kubus kecil seragam yang disebut *voxel*.

Problem utamanya adalah kita diberikan sebuah file `.obj` berisi *polygon mesh* dan program harus menghasilkan `.obj` baru yang mana seluruhnya tersusun dari voxel yang seragam yang merpresentasikan objek aslinya. Namun, ada obstacle yang terletak pada scale, pada depth octree yang tinggi, jumlah candidate voxel cenderung tumbuh secara eksponensial (8^N pada depth N).

Lalu bagaimana caranya kita develop? Ya, dengan menggunakan paradigma *Divide and Conquer* melalui *Octree* (ya sesuai spesifikasi tucil...). Ruang 3D dibungkus oleh bounding box (AABB) berbentuk cubic lalu dipotong secara rekursif menjadi 8 sub-cubic pada setiap level. Di setiap step, kita melakukan uji perpotongan antara segitiga mesh dan kotak menggunakan *Separating Axis Theorem* (SAT). Kotak yang tidak bersinggungan dengan segitiga manapun langsung di-*pruned* sehingga hanya bagian yang relevan yang ditelusuri lebih dalam. Proses ini naturally membentuk paradigma *Divide and Conquer*, divide ruang, conquer setiap bagian independently, lalu terakhir tinggal combine.

Program yang kita develop di sini sepenuhnya dalam `C++`. Selain fungsionalitas utama (wajib), kami juga mengimplementasikan dua fitur bonus yaitu *concurrency* untuk paralelisasi rekursif pada octree menggunakan `std::thread` serta *interactive 3D viewer* dengan raylib yang memungkinkan user melakukan eksplorasi hasil voxelization.

2 Theoretical Background

2.1 Voxelization

Secara sederhana, *voxelization* adalah proses mengubah model 3D yang mulus (biasanya *polygon mesh*) menjadi kumpulan kotak - kotak kecil dalam grid 3D. Analogi sederhananya adalah jika dunia 2D memiliki *pixel*, maka dunia 3D memiliki *voxel* (*volumetric pixel*) sebagai blok bangunan utamanya. Setiap voxel merepresentasikan sebuah kubus kecil di ruang dan menyimpan informasi apakah area tersebut dianggap terisi atau kosong.

Dalam praktiknya, ada dua pendekatan utama. Pertama, *surface voxelization*, yang hanya menandai kotak - kotak pada bagian permukaan objeknya saja. Kedua, *solid voxelization*, yang benar - benar mengisi sampai ke bagian dalam objeknya. Pada tugas ini, pendekatan yang digunakan adalah *surface voxelization* karena target utama kita hanya struktur bentuk luarnya saja agar lebih efisien.

Mengapa metode ini penting? Karena banyak sistem lebih mudah bekerja pada data berbasis grid, misalnya untuk *3D printing*, visualisasi permainan, dan masih banyak contoh lainnya.

2.2 Octree

Octree adalah struktur data hierarkis untuk partisi ruang 3D. Satu node merepresentasikan sebuah *axis-aligned bounding box* (AABB) berbentuk kubus, lalu node tersebut dibagi menjadi 8 child kubus yang ukurannya setengah pada masing-masing sumbu. Proses pembagian ini bisa dilakukan berulang sampai kedalaman tertentu.

Kelebihan utama Octree adalah kemampuannya melakukan *adaptive refinement*. Area ruang yang akan digunakan (misalnya dekat permukaan objek) dapat dibagi lebih dalam, sedangkan area kosong dapat dipangkas lebih awal. Hal ini jauh lebih efisien dibanding membuat seluruh grid *uniform* dari awal, terutama saat model hanya menempati sebagian kecil ruang.

Tentu ada *trade-off* antara kedalaman dan resolusi. Semakin dalam tingkat Octree-nya, voxel-nya akan semakin kecil dan detail objek akan semakin tajam. Namun, jumlah sel dapat meningkat secara drastis. Oleh karena itu, strategi *pruning* (pemangkasan) wajib untuk diimplementasikan agar komputasi tetap efisien.

2.3 Divide and Conquer

Algoritma *Divide and Conquer* bekerja melalui tiga tahap: *divide*, *conquer*, dan *combine*. Pertama, masalah besar dipecah menjadi submasalah yang lebih kecil. Kedua, setiap submasalah diselesaikan secara independen (*umumnya rekursif*). Ketiga, hasil submasalah digabung untuk membentuk solusi akhir.

Pada kasus voxelization berbasis Octree, pola ini muncul secara alami. Tahap *divide* dilakukan saat satu kubus dibagi menjadi 8 sub-kubus. Tahap *conquer* dilakukan dengan mengecek masing-masing sub-kubus: apakah perlu dibagi lagi, dipangkas, atau dijadikan voxel akhir. Tahap *combine* terjadi ketika seluruh hasil voxel dari subtree dikumpulkan menjadi satu kesatuan model yang utuh.

Keuntungan pendekatan ini adalah struktur rekursifnya bersih, modular, dan sangat cocok untuk dibuat paralel. Karena banyak subtree saling independen, proses *conquer* dapat dijalankan bersamaan menggunakan *thread* terpisah jika diperlukan (*fitur bonus*).

2.4 Separating Axis Theorem (SAT)

Inti dari voxelization permukaan adalah untuk mengecek *apakah sebuah segitiga mesh berpotongan dengan sebuah AABB?* Untuk itu, kita menggunakan *Separating Axis Theorem* (SAT).

Prinsip SAT menyatakan bahwa dua objek konveks (*himpunan titik atau bentuk geometri yang tidak memiliki lekukan ke dalam*) tidak berpotongan jika dan hanya jika ada setidaknya satu sumbu proyeksi yang memisahkan keduanya (*separating axis*). Jadi, untuk membuktikan dua objek berpotongan, kita cukup memastikan bahwa tidak ada sumbu pemisah.

Pada pengujian segitiga vs AABB, himpunan sumbu uji yang umum dipakai berjumlah 13, yaitu 3 sumbu normal AABB (sumbu x, y, z), 1 sumbu normal bidang segitiga, dan 9 sumbu hasil *cross product* antara 3 edge segitiga dengan 3 sumbu AABB. Jika pada semua sumbu tersebut proyeksi kedua objek saling overlap, maka segitiga dan AABB dinyatakan berpotongan.

3 Divide and Conquer Algorithm

Sekarang kita masuk ke inti dari project ini. Bagaimana *Divide and Conquer* benar-benar bekerja di dalam proses voxelization. Untuk SAT test-nya sendiri, kita mengacu pada paper Akenine-Möller [5] yang memang jadi standar untuk triangle-AABB intersection. Sedangkan konsep octree-based voxelization-nya terinspirasi dari Schwarz dan Seidel [6], meskipun paper mereka target-nya GPU, kita adapt untuk CPU secara sequential (plus concurrency sebagai bonus).

3.1 High-Level Steps

Pertama tama, bungkus seluruh model dalam satu kotak besar, lalu potong-potong secara rekursif. Di setiap level, lakukan pengecekan "ada segitiga mesh yang menyentuh kotak ini tidak?" Kalau tidak ada, buang (pruned). Kalau ada dan sudah di level terdalam, itu voxel. Kalau ada tapi belum cukup dalam, potong lagi jadi 8 child.

Untuk lebih detail, prosesnya seperti berikut ini.

1. **Bungkus model dalam bounding box.** Pertama, kita hitung AABB terkecil yang membungkus semua vertex. Lalu kita normalisasi jadi kubus sempurna, pakai sisi terpanjang sebagai acuan. Kenapa harus kubus sempurna? Karena kalau tidak, voxel di leaf level ukurannya tidak seragam, dan hasilnya jadi aneh.

2. **Filter segitiga pakai SAT.** Untuk kotak saat ini, kita loop semua segitiga dan cek satu per satu: "segitiga ini berpotongan dengan kotak ini?" Cek-nya pakai SAT dengan 13 sumbu uji. Yang lolos kemudian dikumpulkan di list `intersecting`, sisanya tidak perlu diteruskan ke level berikutnya.

3. **Pruning. Kotak kosong langsung dibuang.** Jika `intersecting`-nya kosong, berarti tidak ada permukaan objek di area ini. Node langsung di-mark sebagai *pruned*, rekursi stop. Mayoritas ruang 3D itu kosong, jadi mayoritas node itu di-prune.

4. **Leaf node. Save sebagai voxel.** Jika depth sudah sama dengan `maxDepth` dan masih ada segitiga yang menyentuh, berarti kotak ini valid sebagai voxel. Langsung push ke `result.voxels`.

5. **Divide. Split jadi 8 child.** Jika belum `maxDepth` dan masih ada segitiga, kotak dipotong jadi 8 sub-kubus. Dengan cara split titik tengah di setiap sumbu (X, Y, Z). Di implementasi, kita pakai bit manipulation, index 0–7 menentukan kombinasi half-space di tiap sumbu.

6. **Conquer. Rekursi tiap child.** Setiap child di-handle independently, masing-masing cuma nerima segitiga yang sudah difilter (bukan seluruh mesh). Proses berulang dari step 2. Karena tiap subtree itu saling independen, ini juga yang bikin concurrency-nya natural.

7. **Combine. Kumpulkan semua voxel.** Setelah semua cabang rekursi selesai, semua leaf node dari setiap subtree terkumpul jadi satu list. List itulah yang akhirnya kita export jadi file `.obj`.

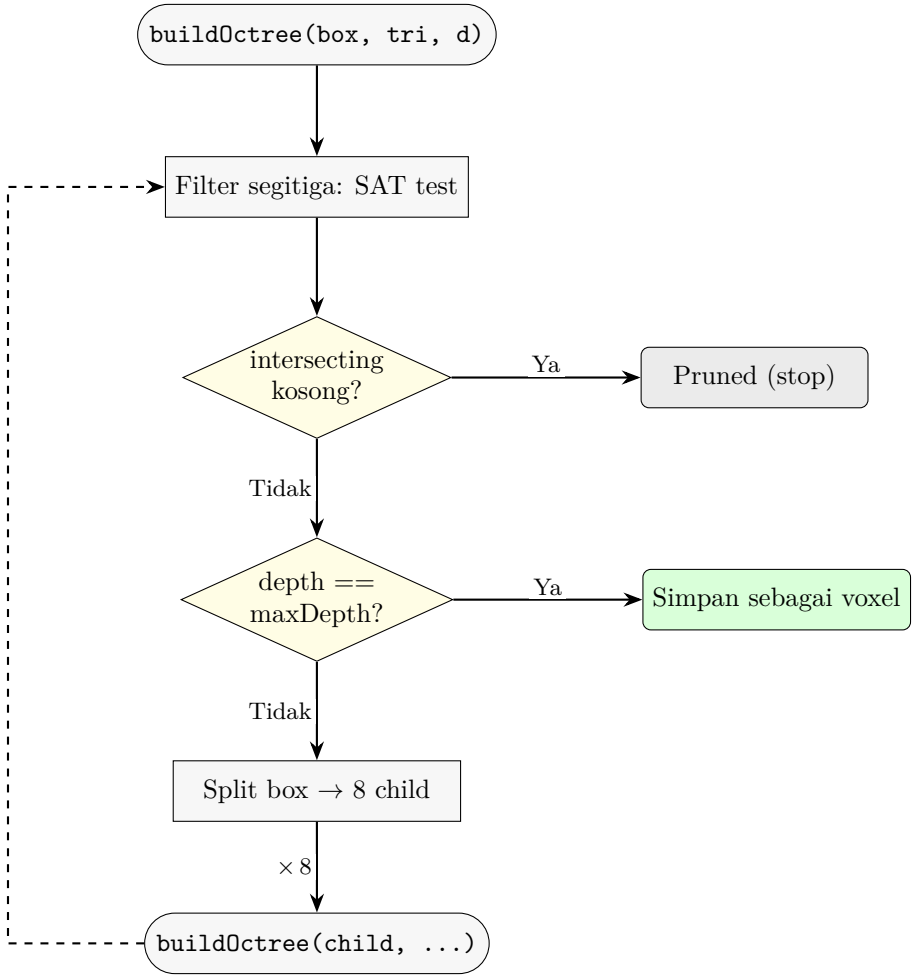
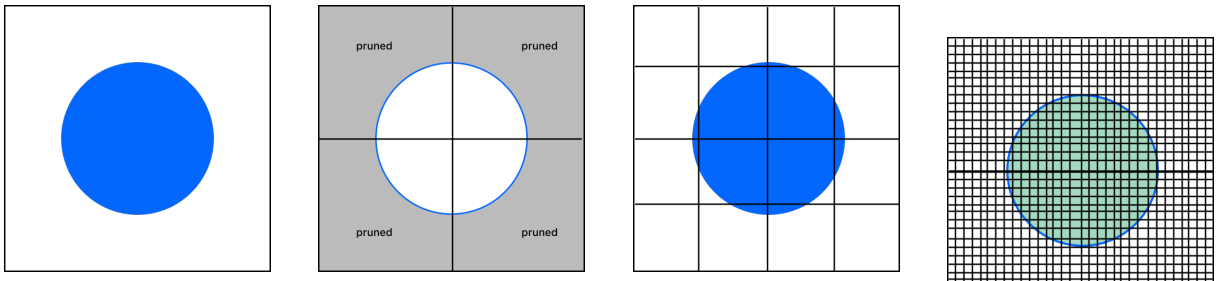


Figure 1: Flow chart keputusan pada setiap level rekursi `buildOctree`.



(a) Bounding box awal membungkus seluruh objek.

(b) Dibagi 4 (2D). Kotak kosong di-prune (abu-abu).

(c) Subdivisi lanjutan pada kotak yang tersisa.

(d) Level terdalam: kotak hijau = voxel akhir.

Figure 2: Ilustrasi 2D (quadtree) dari proses subdivisi bertahap. Pada 3D, setiap kotak dibagi menjadi 8 (bukan 4), tetapi prinsipnya identik.

3.2 Pseudocode

Pseudocode di bawah ini merepresentasikan fungsi `buildOctree` yang jadi core dari seluruh proses. Fungsinya cukup compact, terima kotak, daftar segitiga, depth saat ini, dan depth maksimum, lalu bangun octree secara top-down.

BuildOctree, Rekursi Divide and Conquer

```
function BuildOctree(box, triangles, depth, maxDepth):
    // Input:  box (AABB), triangles (daftar segitiga),
    //         depth (kedalaman saat ini), maxDepth (kedalaman maks)
    // Output: leaf nodes (voxel) ditambahkan ke result

    nodesPerDepth[depth] += 1

    // Filter: hanya simpan segitiga yang menyentuh kotak ini
    intersecting = []
    for each tri in triangles:
        if IntersectsTriangleBox(tri, box) then
            intersecting.append(tri)
        end if
    end for

    // Pruning: tidak ada segitiga -> kotak kosong
    if intersecting is empty then
        prunedPerDepth[depth] += 1
        return
    end if

    // Base case: kedalaman maksimum tercapai -> voxel
    if depth == maxDepth then
        result.voxels.append(box)
        return
    end if

    // Divide: potong kotak menjadi 8 sub-kubus
    children = SplitBox(box)

    // Conquer: rekursi ke setiap child secara independen
    for each child in children:
        BuildOctree(child, intersecting, depth + 1, maxDepth)
    end for
end function
```

`IntersectsTriangleBox` itu SAT test lengkap 13 sumbu yang sudah dibahas di Bagian 2. Dan `SplitBox` tinggal bagi kotak jadi 8 pakai midpoint tiap sumbu. Satu hal penting, di step *Conquer*, tiap child cuma nerima segitiga yang sudah difilter (`intersecting`), bukan seluruh mesh. Jadi semakin dalam level-nya, semakin sedikit segitiga yang perlu dicek.

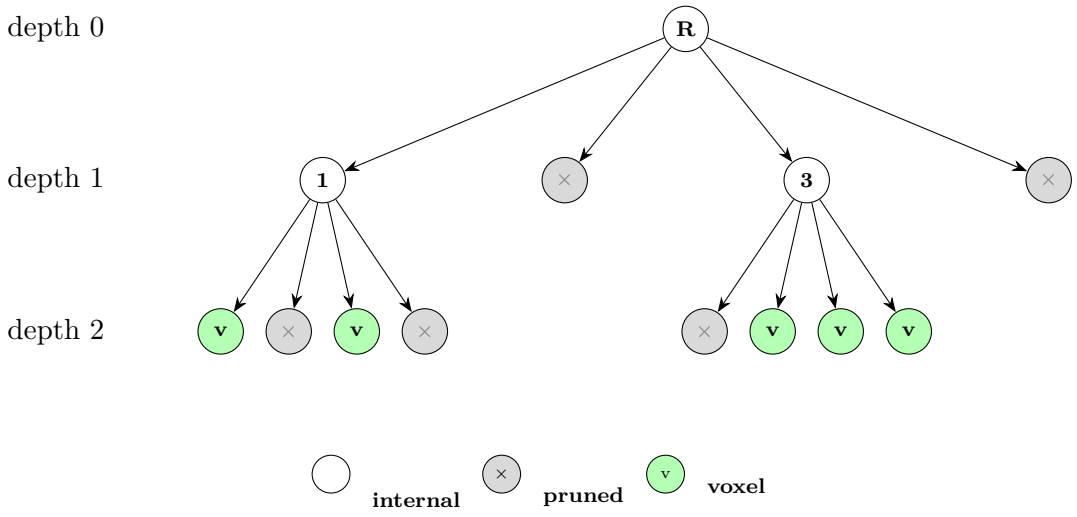


Figure 3: Recursion tree pada octree depth 2 (disederhchildan, 4 child per node). Node pruned (abu-abu) tidak ditelusuri lebih lanjut. Hanya leaf hijau yang jadi voxel akhir.

3.3 Complexity Analysis

Misalkan $N =$ `maxDepth` dan $T =$ jumlah segitiga di mesh input.

Worst case (tanpa pruning). Kalau tidak ada node yang di-prune sama sekali (kasus yang unlikely tapi tetap worth dianalisis), setiap node punya 8 child di setiap level. Total node:

$$\sum_{d=0}^{N} 8^d = \frac{8^{N+1} - 1}{7}$$

Di setiap node, kita jalankan SAT test ke semua segitiga yang diteruskan. SAT test sendiri $O(1)$ per segitiga (13 axis, operasi aritmatika konstan). Jadi worst case totalnya $O(8^N \cdot T)$. Untuk gambaran, $N = 7$ saja itu sudah 2 juta node, kalikan dengan T segitiga per node, angkanya jadi sangat besar.

Average case (dengan pruning). Di praktiknya, pruning memangkas mayoritas node. Model 3D tipikal itu cuma menempati sebagian kecil bounding box-nya, sisanya kosong. Jadi di setiap level, cuma node yang kena permukaan objek yang ditelusuri lebih lanjut. Ditambah lagi, filtering segitiga di tiap level juga bantu, semakin dalam, semakin sedikit segitiga yang relevan buat node tertentu.

Jumlah voxel yang dihasilkan itu proporsional terhadap luas permukaan objek, bukan volume bounding box. Kompleksitas aktualnya jauh di bawah worst case.

Space complexity. Kita tidak menyimpan tree secara eksplisit, yang disimpan cuma leaf nodes

(voxel) di `result.voxels`. Jadi space untuk output itu $O(V)$ di mana V = jumlah voxel. Call stack rekursi memakan $O(N)$ depth, dan di setiap level ada salinan list segitiga yang sudah difilter.

Table 1: Pertumbuhan node maksimum vs. aktual (tipikal) per depth.

Depth (N)	**Max nodes (8^N)**	**Catatan**
1	8	Selalu terjangkau
2	64	Masih sangat cepat
3	512	Umumnya $<$ 200 node aktual
5	32.768	Pruning memangkas $>$ 80%
7	2.097.152	Butuh pruning
10	1.073.741.824	Tidak praktis tanpa pruning

4 Implementation

4.1 System Architecture

Implementasi program dibagi menjadi beberapa modul sehingga tiap komponen punya tugas yang jelas:

- **Parser** (`parser/ObjParser.cpp`): membaca file `.obj`, mengekstrak vertex dan face, lalu mengubah face menjadi kumpulan segitiga melalui *fan triangulation*.
- **Model/Mesh** (`model/Mesh.cpp`): menyusun data mesh, menghitung bounding box, dan menormalisasi bounding box menjadi kubus (AABB cubic) sebagai root octree.
- **Intersection & Geometry** (`model/BoundingBox.cpp`): berisi operasi geometri dasar (`splitBox`, `centerBox`, dll.) serta uji `intersectsTriangleBox` berbasis SAT.
- **Octree** (`octree/Octree.cpp`): inti rekursi divide and conquer untuk membangun voxel leaf, sekaligus mengisi statistik node dan pruning per depth.
- **Exporter** (`exporter/ObjExporter.cpp`): menulis hasil voxel ke `output.obj`, dengan 8 vertex dan 12 face segitiga per voxel.
- **Statistics** (`stats/stats.cpp`): memberikan *output* ringkasan metrik (jumlah voxel, vertex, face, distribusi node, dan waktu eksekusi).
- **Viewer** (`viewer/Viewer.cpp`): visualisasi interaktif voxel menggunakan raylib (orbit, zoom, clip slider, render mode, dll).

Secara dependensi, alur modul dapat diringkas sebagai: `main -> parser/model -> octree -> exporter/stats -> viewer`.

4.2 Program Flow

Alur eksekusi program di `main.cpp` adalah sebagai berikut:

1. Program membaca input path file OBJ dan nilai `maxDepth` dari user.
2. Fungsi `loadObj()` dipanggil untuk mem-parse mesh, membentuk daftar segitiga, menghitung bounds, lalu membuat bounds kubik.
3. Timer dimulai, kemudian `buildOctree()` dieksekusi dari root box dengan daftar segitiga awal.
4. Setelah rekursi selesai, timer dihentikan dan waktu eksekusi dihitung.
5. Leaf voxel diekspor ke `../test/output.obj` melalui `exportObj()`.
6. Statistik dibangun memakai `buildStats()` dan dicetak dengan `printStats()`.
7. Data voxel ditampilkan pada viewer interaktif lewat `launchViewer()`.

Secara rekursif, `buildOctree()` melakukan proses berikut pada setiap node:

1. Tambah counter node pada depth saat ini.

2. Filter segitiga yang berpotongan dengan box saat ini (menggunakan `intersectsTriangleBox()`).
3. Jika tidak ada segitiga yang berpotongan, node di-prune.
4. Jika depth sudah mencapai `maxDepth`, box disimpan sebagai voxel leaf.
5. Jika belum, box dibagi jadi 8 child dan diproses rekursif.

4.3 Implementation Details

Beberapa keputusan implementasi penting yang memengaruhi hasil akhir:

- **Fan triangulation saat parsing face.** Face OBJ dengan lebih dari 3 vertex dipecah menjadi beberapa segitiga. Pendekatan ini membuat proses berikutnya cukup menangani primitive segitiga saja.
- **Support indeks negatif OBJ.** Pada parsing face, indeks negatif tetap didukung dan di-resolve relatif terhadap jumlah vertex yang sudah terbaca, sehingga kompatibilitas parser lebih baik.
- **Normalisasi root bounds menjadi kubus.** Setelah bounds mesh dihitung, sisi kubus diambil dari dimensi terbesar (`max3(size.x, size.y, size.z)`). Hal ini penting agar subdivisi octree benar-benar uniform pada ketiga sumbu.
- **Uji SAT.** Sebelum SAT penuh, implementasi mengecek: titik segitiga di dalam box, lalu overlap antara triangle AABB dan box AABB. Jika gagal di tahap awal, perhitungan axis SAT tidak dijalankan.
- **13 sumbu SAT untuk triangle vs box.** Implementasi memeriksa 9 sumbu hasil *cross product* edge segitiga dengan sumbu box, 3 sumbu utama box, dan 1 normal segitiga.
- **Thread pada hasil octree.** Struktur `OctreeResult` memakai `mutex` untuk update `nodesPerDepth`, `prunedPerDepth`, dan `voxels`. Kode sudah mendukung eksekusi paralel per subtree, namun saat ini `THREAD_LIMIT` diset 1 agar eksekusi efektifnya tetap stabil.
- **Representasi voxel langsung sebagai kubus OBJ.** Setiap voxel ditulis sebagai 8 vertex dan 12 face segitiga, sehingga output siap diproses tool 3D standar tanpa format tambahan.
- **Viewer.** Viewer menyediakan fitur clip, wireframe, orbit camera, dan membantu memvalidasi (*secara visual*) apakah voxelisasi permukaan sudah mengikuti bentuk mesh asli.

5 Results and Analysis

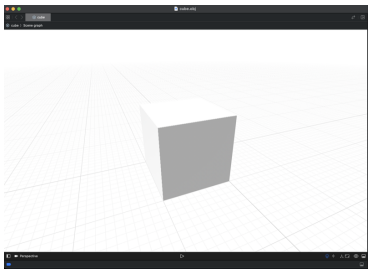
Pada bab ini, pengujian dilakukan dengan parameter depth octree tetap ($d = 4$) untuk perbandingan antar-model. Setiap pengujian menghasilkan metrik yang dicetak oleh program: jumlah voxel, jumlah vertex/face hasil ekspor, distribusi node octree, jumlah node yang ter-prune, serta waktu eksekusi.

Secara umum, metrik yang diamati adalah:

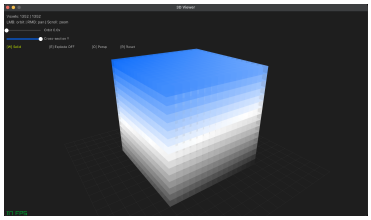
- Kompleksitas input (jumlah vertex dan triangle mesh awal),
- Kompleksitas struktur octree (node aktif dan node ter-prune),
- Ukuran output voxel (voxel, vertex, face),
- Waktu komputasi (ms).

5.1 Test Case 1

Model: `cube.obj`, **depth:** 4



(a) Mesh awal



(b) Hasil voxelization

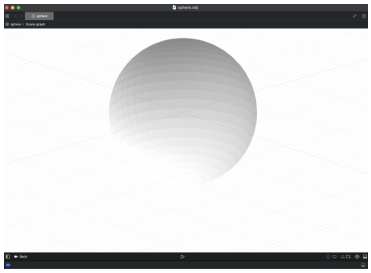
Metrik	Nilai
Vertex awal	8
Triangle awal	12
Voxel count	1352
Vertex output	10816
Face output	16224
Node total	2889
Pruned total	1176
Waktu	4.34 ms

Figure 4: Voxelization `cube.obj` (depth 4)

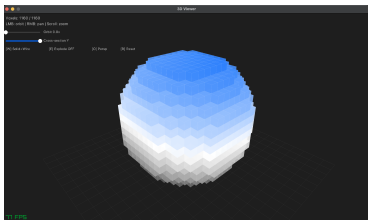
Model kubus menghasilkan voxel count cukup tinggi karena seluruh permukaan terdistribusi merata pada sisi-sisi AABB, sehingga banyak leaf level akhir tetap relevan.

5.2 Test Case 2

Model: `sphere.obj`, **depth:** 4



(a) Mesh awal



(b) Hasil voxelization

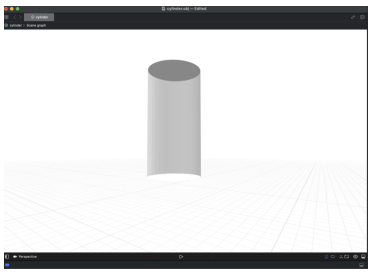
Metrik	Nilai
Vertex awal	1106
Triangle awal	2208
Voxel count	1160
Vertex output	9280
Face output	13920
Node total	2697
Pruned total	1200
Waktu	7.50 ms

Figure 5: Voxelization `sphere.obj` (depth 4)

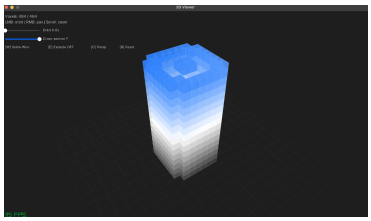
Meskipun voxel count sedikit lebih rendah dari kubus, waktu eksekusi lebih tinggi karena jumlah triangle input jauh lebih besar. Hal ini menunjukkan biaya SAT bergantung pada kompleksitas mesh sumber.

5.3 Test Case 3

Model: `cylinder.obj`, **depth:** 4



(a) Mesh awal



(b) Hasil voxelization

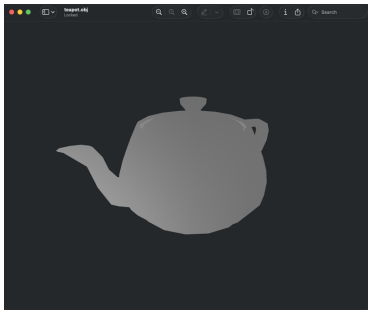
Metrik	Nilai
Vertex awal	66
Triangle awal	128
Voxel count	640
Vertex output	5120
Face output	7680
Node total	1801
Pruned total	936
Waktu	3.95 ms

Figure 6: Voxelization `cylinder.obj` (depth 4)

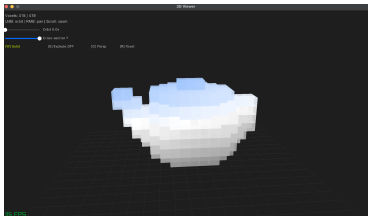
Silinder memiliki geometri lebih sederhana daripada sphere dan torus, sehingga menghasilkan jumlah node aktif dan waktu eksekusi yang lebih kecil.

5.4 Test Case 4

Model: `teapot.obj`, **depth:** 4



(a) Mesh awal



(b) Hasil voxelization

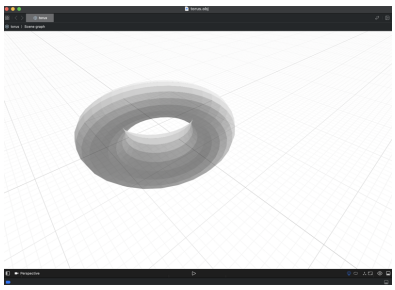
Metrik	Nilai
Vertex awal	530
Triangle awal	1024
Voxel count	418
Vertex output	3344
Face output	5016
Node total	1129
Pruned total	570
Waktu	3.87 ms

Figure 7: Voxelization `teapot.obj` (depth 4)

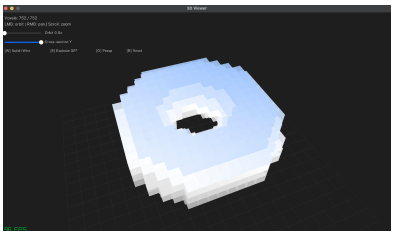
Teapot menghasilkan voxel count paling kecil pada depth yang sama, yang berarti distribusi permukaan objek relatif tidak memenuhi banyak sel leaf.

5.5 Test Case 5

Model: `torus.obj`, **depth:** 4



(a) Mesh awal



(b) Hasil voxelization

Metrik	Nilai
Vertex awal	512
Triangle awal	1024
Voxel count	752
Vertex output	6016
Face output	9024
Node total	1865
Pruned total	880
Waktu	4.84 ms

Figure 8: Voxelization `torus.obj` (depth 4)

Lubang pada torus menyebabkan pemetaan voxel yang khas, yaitu lebih banyak voxel dibanding teapot, namun tetap lebih rendah dari cube/sphere karena bagian volume tengah (void) tidak menambah permukaan terisi.

5.6 Performance Summary

Tabel berikut merangkum kelima pengujian utama.

Table 2: Ringkasan performa seluruh test case (depth 4)

Model	Triangles	Voxels	Node Total	Pruned	Waktu (ms)
cube.obj	12	1352	2889	1176	4.34414
sphere.obj	2208	1160	2697	1200	7.50067
cylinder.obj	128	640	1801	936	3.94558
teapot.obj	1024	418	1129	570	3.87233
torus.obj	1024	752	1865	880	4.83732

Selain perbandingan antar-model, pengaruh depth juga diuji pada `cube.obj`.

Table 3: Pengaruh depth terhadap jumlah voxel dan waktu (cube.obj)

Depth	Voxels	Node Total	Waktu (ms)
2	56	73	1.67802
3	296	521	2.01615
4	1352	2889	4.34414
5	5768	13705	14.74950

Dari tabel depth terlihat bahwa penambahan resolusi meningkatkan detail, tetapi mendorong kenaikan jumlah node dan voxel secara sangat cepat. Nilai ini konsisten dengan sifat subdivisi octree ($\approx 8^d$ pada skenario terburuk), meskipun pruning menekan pertumbuhan aktual agar tetap jauh di bawah batas teoritis.

Secara keseluruhan, implementasi mampu menjaga waktu eksekusi pada orde milidetik untuk depth menengah (hingga depth 4) pada model uji yang beragam.

6 Bonus Features

6.1 Concurrency

Fitur concurrency diimplementasikan pada proses rekursif pembentukan octree di `octree/Octree.cpp`. Ide utamanya adalah setiap child hasil pembelahan node (8 sub-box) dapat diproses independen, sehingga secara alami cocok untuk model *task parallelism*.

Strategi yang dipakai:

- Pada node yang masih memenuhi syarat paralelisasi, setiap child dieksekusi melalui `std::thread` terpisah.
- Setiap thread memanggil `buildOctree()` secara rekursif pada subtree masing-masing.
- Setelah semua thread dibuat, dilakukan `join()` untuk memastikan sinkronisasi sebelum fungsi parent selesai.

Karena seluruh thread menulis ke satu struktur hasil global (`OctreeResult`), implementasi menambahkan proteksi `mutex` pada *critical section* berikut:

- update `nodesPerDepth`,
- update `prunedPerDepth`,
- penambahan voxel leaf ke `result.voxels`.

Implementasi saat ini memakai konstanta:

```
const int THREAD_LIMIT = 1;
```

Konfigurasi ini mengartikan bahwa kerangka paralel sudah tersedia, tetapi secara default eksekusi dijaga stabil. Jika `THREAD_LIMIT` diperbesar, waktu eksekusi dapat meningkat signifikan karena beberapa hal, seperti *context switching*, perebutan sumber daya (*terlalu banyak thread membuatnya harus menunggu*), *thread creation overhead*, dan berbagai alasan lainnya.

Secara umum, manfaat utama fitur ini adalah:

- Kode rekursif tetap modular
- Mekanisme sinkronisasi sudah siap

6.2 Interactive 3D Viewer

Setelah voxelization selesai dan file `.obj` sudah ter-generate, ada fitur viewer. Hasil file `.obj` pada dasarnya bisa dibuka melalui Blender, tetapi cenderung overkill untuk sekadar cek apakah bentuknya masuk akal.

Viewer ini dibangun menggunakan **raylib** (C++, cross-platform), dan sifatnya fully interactive. Begitu proses voxelization dan export selesai, program langsung membuka window baru yang menampilkan seluruh voxel hasil konversi dalam 3D space.

Camera System

Kamera menggunakan model *spherical orbit* yang mengelilingi pusat scene. User bisa kontrol penuh lewat mouse. Selain itu, kamera juga bisa otomatis *auto-orbit* mengelilingi objek dengan kecepatan yang bisa diatur lewat slider (0 sampai 5× multiplier).

Render Modes

Ada tiga mode render yang bisa di-cycle dengan key `[W]`:

1. **Solid + Wireframe** (default). face solid dengan overlay wireframe semi-transparan.
2. **Solid**. face solid tanpa wireframe jadi lebih clean.
3. **Wireframe only**. hanya edge yang direnderr, interior transparan.

Cross-Section Clipping

Fitur tambahan. Ini fitur yang mungkin paling berguna untuk analisis. Ada slider *clip Y* yang bisa digeser dari -1.0 sampai 1.0, dan semua voxel di atas garis potong itu langsung disembunyikan.

Explode View

Dengan key `[E]`, semua voxel bisa di-"explode" secara radial dari pusat scene. Fitur ini sebenarnya tambahan juga dan berguna untuk melihat distribusi spasial voxel secara individual, terutama di bagian yang biasanya tertutup voxel lain.

7 Conclusion

Dari hasil pengujian pada lima model (cube, sphere, cylinder, teapot, torus) dengan depth 4, program berhasil menghasilkan output voxel yang secara visual merepresentasikan bentuk asli masing-masing objek. Waktu eksekusi berada di miliseconds untuk depth menengah, yang menunjukkan bahwa pruning bekerja efektif. Pengujian pengaruh depth pada `cube.obj` juga mengkonfirmasi trade-off antara resolusi dan biaya komputasi, semakin tinggi depth, semakin detail hasilnya, tapi jumlah node dan waktu eksekusi naik secara signifikan.

Dua fitur bonus juga berhasil diimplementasikan. Pertama, *concurrency* dengan `std::thread` dan proteksi `mutex` pada shared state, di mana kerangka paralelisasi sudah tersedia meskipun secara default dijalankan konservatif untuk menjaga stabilitas. Kedua, *interactive 3D viewer* dengan raylib yang menyediakan orbit camera, multiple render modes, cross-section clipping, dan explode view, sehingga user bisa langsung memvalidasi hasil voxelization tanpa tool eksternal.

References

[1] R. Munir, "Algoritma Divide and Conquer (2026) — Bagian 1," Teknik Informatika, Institut Teknologi Bandung, 2026. https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2025-2026/07-Algoritma-Divide-and-Conquer-(2026)-Bagian1.pdf

[2] R. Munir, "Algoritma Divide and Conquer (2026) — Bagian 2," Teknik Informatika, Institut Teknologi Bandung, 2026. https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2025-2026/08-Algoritma-Divide-and-Conquer-(2026)-Bagian2.pdf

[3] R. Munir, "Algoritma Divide and Conquer (2026) — Bagian 3," Teknik Informatika, Institut Teknologi Bandung, 2026. https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2025-2026/09-Algoritma-Divide-and-Conquer-(2026)-Bagian3.pdf

[4] R. Munir, "Algoritma Divide and Conquer (2026) — Bagian 4," Teknik Informatika, Institut Teknologi Bandung, 2026. https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2025-2026/10-Algoritma-Divide-and-Conquer-(2026)-Bagian4.pdf

[5] T. Akenine-Möller, "Fast 3D Triangle-Box Overlap Testing," *Journal of Graphics Tools*, vol. 6, no. 1, pp. 29–33, 2001. https://doi.org/10.1080/10867651.2001.10487535 (Accessed: 18 March 2026).

[6] M. Schwarz and H.-P. Seidel, "Fast Parallel Surface and Solid Voxelization on GPUs," *ACM Transactions on Graphics*, vol. 29, no. 6, article 179, 2010. https://doi.org/10.1145/1882261.1866201 (Accessed: 18 March 2026).

[7] J. Baert, "Out-Of-Core Construction of Sparse Voxel Octrees," 2012. https://www.forceflow.be/2012/07/24/out-of-core-construction-of-sparse-voxel-octrees/ (Accessed: 26 March 2026).

Repository

https://github.com/ethj0r/Tucil2_13524026_13524104

Lampiran

Table 4: Checklist kelengkapan program

No	Poin	Ya	Tidak
1	Program berhasil dikompilasi tanpa kesalahan	✓	
2	Program berhasil dijalankan	✓	
3	Hasil konversi `.obj` terdiri dari kubus-kubus kecil	✓	
4	Program menampilkan informasi-informasi atau report	✓	
5	Program dapat menerima masukan `.obj` dan mengembalikan output	✓	
6	[Bonus] Program menggunakan concurrency	✓	
7	[Bonus] Program dapat menampilkan visualisasi `.obj`	✓	

Tugas ini disusun sepenuhnya tanpa bantuan kecerdasan buatan (Generative AI), melainkan hasil pemikiran dan analisis mandiri.

Made Branenda Jordhy
13524026

Valentino Daniel Kusumo
13524104