BAB II

LANDASAN TEORI

2.1. Software Framework

Menurut Johnson (1992) framework adalah desain *reusable* dari sebuah program atau bagian dari sebuah program yang diekspresikan sebagai sekumpulan kelas. Dengan demikian, maka software framework dapat diartikan sebagai sekumpulan kode atau *library* yang menyediakan fungsionalitas umum yang digunakan untuk menghasilkan aplikasi spesifik sesuai kebutuhan *user*. Biasanya satu library normal akan menyediakan satu fungsi tertentu, tetapi berbeda dengan framework yang menawarkan jangkauan yang lebih luas yang seringkali digunakan untuk pengembangan salah satu jenis aplikasi. Daripada membuat ulang logika yang sudah umum, *programmer* dapat memanfaatkan framework dimana biasanya menyediakan fungsi-fungsi yang sering digunakan, mengurangi waktu yang dibutuhkan untuk membangun sebuah aplikasi dan serta mengurangi kemungkinan penambahan *error* atau *bug* baru.

Software framework dapat terdiri dari progam pendukung, kompiler, kode library, *application programming interface* (API), dan sekumpulan alat (*tool*) yang menggabungkan semua komponen yang berbeda untuk menjalankan pengembangan suatu proyek aplikasi. Sebagai contoh framework apilkasi web mungkin menyediakan manajemen user *session*, penyimpanan data, dan sistem *template*. Sedangkan framework aplikasi *dekstop* mungkin menyediakan fungsi user interface dan *widget* (elemen *GUI* umum). Hampir semua framework mengontrol setidaknya beberapa aspek dari alur eksekusi aplikasi.

Framework memiliki fitur kunci pembeda yang memisahkan mereka dari library normal yaitu:

1. Inversi dari kontrol: Tidak seperti library atau aplikasi user yang biasanya, alur kontrol dari program secara keseluruhan tidak diatur oleh pemanggil (*caller*), melainkan oleh framework (Riehle 2000).
2. *Default behavior*: Framework memiliki default behavior yaitu suatu perilaku atau aturan operasi yang selalu sama dalam setiap penggunaannya sehingga akan memudahkan user dalam mengembangkan aplikasi. Default behavior harus merupakan suatu operasi yang bermanfaat bukan serangkaian operasi yang tidak melakukan apa-apa.
3. Ekstensibilitas: Framework dapat diperluas oleh user biasanya dengan cara melakukan *overriding* secara selektif atau spesialisasi kode user yang menghasilkan fungsionalitas spesifik.
4. Kode framework *Non-Modifiable*: Secara umum kode framework tidak diijinkan untuk dimodifikasi. User dapat menambah framework tetapi bukan memodifikasi kodenya.

Ada berbagai jenis dari framework bahkan di dalam satu klasifikasi dari aplikasi. Beberapa menawarkan sedikit lebih banyak dari fungsi utilitas dasar dari suatu aplikasi. Dan yang lainnya menawarkan hampir menjadi sebuah aplikasi lengkap dan memerlukan organisasi kode yang baku dan beberapa aturan-aturan lainnya. Memilih framework yang terbaik untuk suatu proyek seringkali membutuhkan programmer untuk menyeimbangkan seberapa banyak fungsi-fungsi yang mereka dapatkan dari framework terhadap fleksibilitas yang mereka harapkan.

2.2. Komputer Grafis 3D

Menurut Santosa (1994:2), “Grafika komputer (komputer grafis) pada dasarnya adalah suatu bidang ilmu komputer yang mempelajari tentang cara-cara untuk meningkatkan dan memudahkan komunikasi antara manusia dengan mesin (komputer) dengan jalan membangkitkan, menyimpan dan memanipulasi gambar model suatu obyek menggunakan komputer”. Berdasarkan penjelasan diatas aplikasi yang berbasis komputer grafis 3D menggunakan representasi tiga dimensi dari data geometris (biasanya menggunakan sistem koordinat Kartesius) yang tersimpan pada komputer yang ditujukan untuk melakukan kalkulasi dan rendering gambar 2D.

Dalam pembuatan komputer grafis 3D dibagi menjadi 3 tahapan yaitu sebagai berikut:

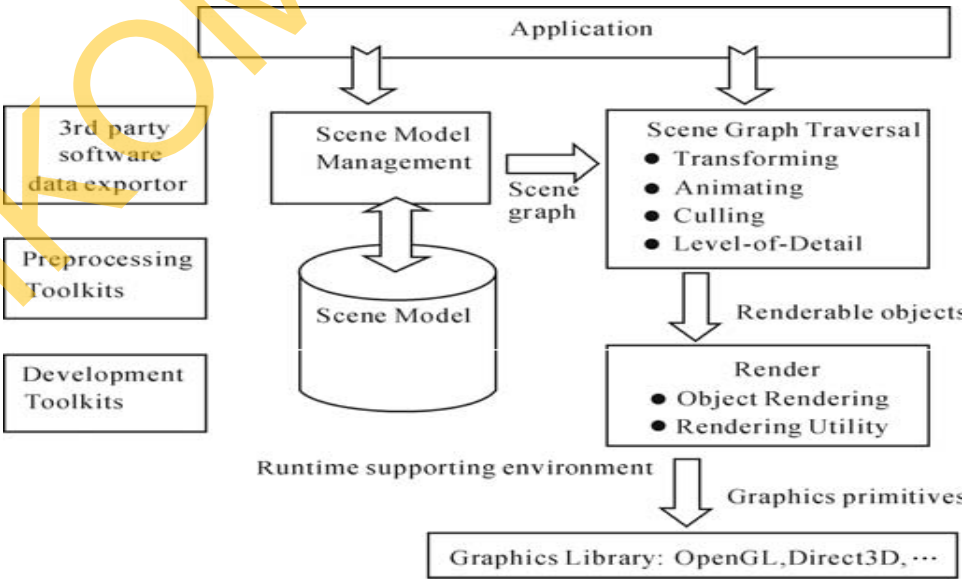
1. Pemodelan 3D: Proses pembentukan model komputer dari bentuk obyek.
2. *Layout* dan animasi: Pergerakan dan penempatan dari obyek dalam scene.
3. Rendering 3D: Perhitungan komputer yang berdasarkan pada penempatan cahaya, jenis permukaan, dan kualitas lainnya dalam pembuatan gambar.

2.3. Rendering Engine

Dalam pemrograman komputer grafis, proses rendering engine hampir sama dengan cara kerja mesin pada umumnya yaitu merupakan bagian dari proyek dimana menjalankan fungsionalitas tertentu dari program. Dimulai dari memanggil fungsi menyalakan engine untuk melakukan persiapan. Kemudian engine akan mencari adapter grafis yang siap dijalankan. Ketika engine dikendalikan seperti memasukkan model 3D kedalam adapter grafis, maka rendering engine dengan segera menampilkannya kedalam layar. Rendering

engine melakukan semua pekerjaaan tingkat rendah seperti melakukan komunikasi dengan adapter grafis, mengatur *render states*, transformasi model, dan berurusan dengan matematika yang rumit seperti matriks rotasi. Pekerjaan-pekerjaan kotor ini sangat diperlukan akan tetapi seperti pengemudi kendaraan dalam hal ini sebagai developer tidak perlu harus memahami cara kerja mesin didalamnya dan yang terpenting adalah agar dapat digunakan untuk mencapai tujuannya.

Kebanyakan rendering engine dibangun dari application programming interface (API) grafis seperti Direct3D, Java3D atau OpenGL dimana menyediakan abstraksi software dari *graphics processing unit* (GPU). Menurut Bao dan Hua (2011), Seringkali rendering engine menyediakan setidaknya satu modul yaitu *scene graph manager* (rendering engine yang lain mungkin memiliki nama yang berbeda) untuk membantu memanipulasi grafis di dalam scene seperti membuat, memodifikasi, mereferensi, menduplikasi, mencari, menghapus node dari scene graph.



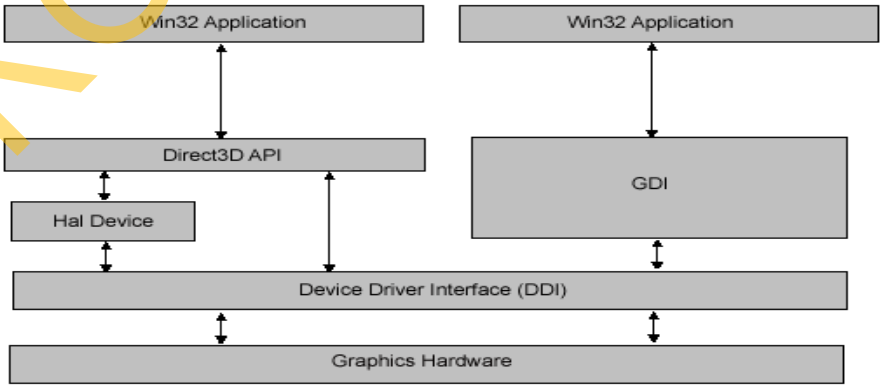
Gambar 2.1. Contoh struktur dari Rendering Engine
[Sumber: Bao dan Hua, 2011]

Zerbst dan Düval (2004) menyimpulkan bahwa rendering engine harus melakukan kegiatan-kegiatan sebagai berikut:

1. Memanajemen semua data dan wilayah tanggung jawabnya.
2. Melakukan komputasi semua data berdasarkan tugas wilayahnya.
3. Menyampaikan semua data menuju instansi berikutnya, jika diperlukan.
4. Menerima semua data untuk dikelola dan untuk dikomputasi dari instansi sebelumnya.

2.4. Direct3D API

Direct3D merupakan *subset* atau bagian dari Microsoft DirectX application programming interface (API). Menurut Luna (2003) Direct3D adalah API grafis tingkat rendah yang memungkinkan kita untuk merender dunia 3D menggunakan akselerasi hardware. Direct3D dapat digambarkan sebagai mediator antara aplikasi dan *graphics device* (hardware 3D). Sebagai contoh, untuk menginstruksikan graphics device untuk membersihkan layar, aplikasi akan memanggil fungsi Direct3D IDirect3DDevice9::Clear. Gambar dibawah ini menunjukkan hubungan antara aplikasi, Direct3D, dan hardware grafis.



Gambar 2.2. Hubungan antara Aplikasi, Direct3D dan Hardware
[Sumber: http://www.codeproject.com/KB/graphics/DirectX_Lessons_2_/device.jpg Time Download: 31/07/2013 11:38:34]

Bagian Direct3D pada gambar diatas mendefinisikan sekumpulan interface dan fungsi yang dipaparkan ke aplikasi atau programmer. Interface dan fungsi tersebut mewakili seluruh fitur dimana mendukung versi Direct3D saat ini. Seperti yang ditunjukkan gambar tersebut, disana juga ada langkah penengah antara Direct3D dan graphics device yaitu HAL (*Hardware Abstraction Layer*). Direct3D tidak dapat berinteraksi secara langsung dengan graphics device karena karena terdapat bergam kartu grafis yang beredar di pasaran, dan masing-masing kartu tersebut memiliki beragam kemampuan dan cara dalam implementasinya. Sebagai contoh, dua kartu grafis yang berbeda mungkin mengimplementasikan operasi membersihkan layar secara berbeda. Dengan demikian Direct3D membutuhkan manufaktur device untuk mengimplementasikan HAL. HAL adalah sekumpulan kode spesifik device yang menginstruksikan device untuk melakukan operasi. Dengan cara ini Direct3D dapat menghindari untuk mengetahui secara detil spesifik dari device, dan spesifikasinya dapat dibuat secara independen dari hardware device.

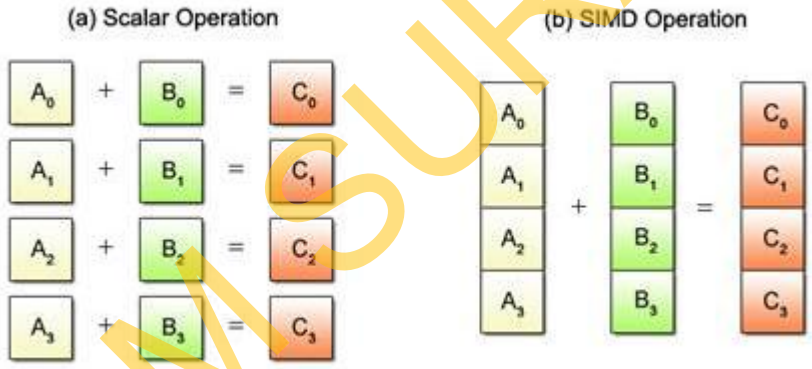
Manufaktur device mengimplementasikan semua fitur yang mereka dukung kedalam HAL. Fitur yang dipaparkan oleh Direct3D tetapi tidak didukung oleh device tidak akan diimplementasikan kedalam HAL. Pemanggilan fungsi Direct3D yang tidak diimplementasikan oleh HAL akan menghasilkan kegagalan (*failure*), kecuali operasi pemrosesan verteks, dimana fungsionalitas yang diharapkan dapat diemulasi di dalam software, jika menggunakan *software vertex processing* oleh Direct3D *runtime*.

Dengan dukungan dari kartu grafis yang mengimplementasikan HAL tersebut Direct3D dapat memaparkan kemampuan grafis tingkat lanjut (*advanced*)

dari hardware grafis 3D, antara lain *z-buffering*, *anti-aliasing*, *alpha blending*, *mipmapping*, *atmospheric effect*, dan *perspective-correct texture mapping*.

2.5. Teknologi SIMD

SIMD merupakan singkatan dari Single Instruction Multiple Data. Umumnya unit prosesing (CPU) mengolah data dan melakukan instruksi pada data tersebut dalam sekuensial linear yang disebut sebagai operasi skalar, akan tetapi dengan menggunakan SIMD dapat melakukan sebuah instruksi pada beberapa data sekaligus secara paralel (Cockshott dan Renfrew 2004). Seperti gambar dibawah ini:



Gambar 2.3 Perbedaan (a) operasi skalar, dan (b) operasi SIMD
[Sumber: https://www.kernel.org/pub/linux/kernel/people/geoff/cell/ps3-linux-docs/CellProgrammingTutorial/CellProgrammingTutorial.files/image008.jpg
Time Download: 19/07/2013 10:02:34]

Seandainya ada sebuah vektor yang memiliki tiga buah data *floating* point masing-masing sebesar 32 bit dengan total 96 bit. Jika menggunakan teknologi SIMD seperti SSE dengan register 128-bit, maka vektor tersebut dapat melakukan instruksi misalnya penambahan hanya sekali saja. Sedangkan menggunakan register normal seperti 64-bit atau 32-bit membutuhkan instruksi

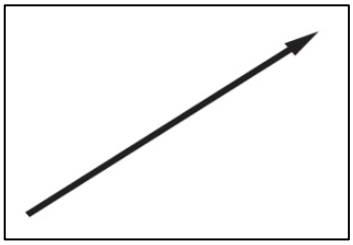
sebanyak 2 atau 3 kali. Dengan demikian SIMD mampu meningkatkan performa dari operasi terhadap vektor tersebut secara signifikan.

2.6. Matematika 3D

Matematika 3D biasanya berhubungan dengan komputasi geometri, dimana berurusan dengan pemecahan permasalahan geometri secara algoritma (Dunn dan Parberry 2002). Matematika 3D dan komputasi geometri memiliki aplikasi dalam bidang yang sangat beragam yang menggunakan komputer untuk memodelkan atau hal-hal yang mendasari dunia dalam 3D, seperti grafis, game, simulasi, robotik, *virtual reality*, dan sinematografi.

2.6.1. Vektor

Vektor merupakan entitas matematika formal yang biasanya digunakan untuk melakukan perhitungan matematika 2D dan 3D. Kata vektor memiliki arti yang berbeda tapi saling berhubungan. Definisi secara matematika vektor adalah daftar bilangan (kumpulan bilangan) atau dalam komputer sebagai *array* dari bilangan. Vektor memiliki dimensi mulai 1D (atau disebut skalar), 2D, 3D dan seterusnya contoh penulisan vektor: [1, 2, 3]. Sedangkan secara geometri vektor adalah segmen garis berarah yang memiliki besaran (*magnitude*) dan arah (Gambar 2.4.), akan tetapi vektor juga dapat mewakili suatu titik (*point*) di dalam ruang koordinat walaupun secara konseptual memiliki arti yang berbeda.



Gambar 2.4. Vektor 2D memiliki magnitude dan arah

Vektor memiliki beragam operasi-operasi yang sangat bermanfaat untuk matematika tiga dimensi. Secara geometri operasi negasi pada vektor akan menghasilkan magnitude yang sama tetapi memiliki arah yang berlawanan, contoh penulisan: - [3, 5, -4] = [-3, -5, 4]. Vektor memiliki magnitude (besaran) atau juga disebut panjang vektor (*norm*), dan persamaan yang digunakan untuk mencari magnitude tersebut adalah sebagai berikut (Lengyel 2004):

$$\| \mathbf{v} \| = \sqrt{\sum_{i=1}^{n} \mathbf{v}_i{}^2}$$

Untuk mempermudah perhitungan matematika 3 dimensi terkadang kuantitas vektor berkosentrasi pada arahnya bukan magnitude, dengan demikian vektor tersebut dijadikan vektor unit. Vektor unit adalah vektor yang memiliki nilai magnitude 1, yang juga dikenal sebagai normalisasi vektor (*normal*). Untuk membuat normalisasi vektor persamaannya adalah sebagai berikut:

$$\mathbf{v}_{norm} = \frac{\mathbf{v}}{\| \mathbf{v} \|}, \mathbf{v} \neq \mathbf{0}$$

Operasi produk (perkalian) pada 2 vektor bersamaan ada dua jenis yaitu *dot product* dan *cross product*. Dot product diambil dari simbol titik pada notasinya: **a . b** yang menghasilkan nilai skalar. Persamaannya adalah sebagai berikut:

$$\mathbf{a} \cdot \mathbf{b} = \sum_{i=1}^{n} \mathbf{a}_i \mathbf{b}_i$$

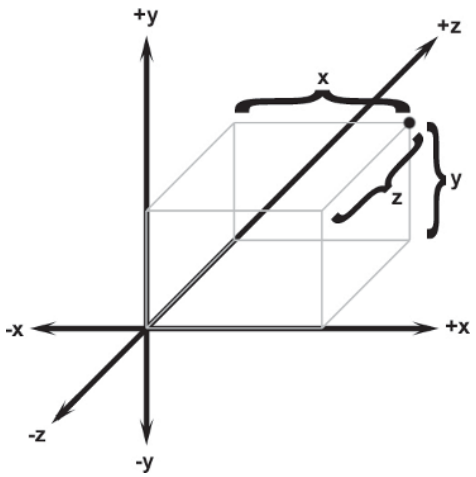
Sedangkan cross product hanya dapat digunakan pada vektor 3 dimensi. Hasil produknya akan menghasilkan vektor 3 dimensi juga, dengan notasi: **a** × **b** persamaannya adalah sebagai berikut:

$$\boldsymbol{P} \times \boldsymbol{Q} = \langle PyQz - PzQy, PzQx - PxQz, PxQy - PyQx \rangle$$

2.6.2. Sistem Koordinat Kartesius

Menurut Dunn dan Parberry (2002), Matematika Kartesian ditemukan oleh dan diberi nama sesuai dengan matematikawan dan filsuf terkenal Perancis yang bernama René Descartes yang hidup antara tahun 1596 sampai dengan tahun 1650. Dia juga terkenal dengan merevolusi matematika yang menyediakan hubungan secara sistematik pertama kali antara geometri dan aljabar.

Sistem koordinat Kartesius digunakan untuk menentukan tiap titik dalam bidang dengan menggunakan dua bilangan yang biasa disebut koordinat x dan koordinat y dari titik tersebut. Sistem koordinat Kartesius dapat pula digunakan pada dimensi-dimensi yang lebih tinggi, seperti 3 dimensi, dengan menggunakan tiga sumbu (sumbu x, y, dan z).



Gambar 2.5. Menentukan lokasi pada 3D
[Sumber: Dunn dan Parberry, 2002]

Dalam menentukan lokasi suatu titik dalam ruang tiga dimensi ditentukan dengan tiga angka x, y, dan z yang masing-masing memiliki jarak terhadap bidang yz, xz, dan xy (Gambar 2.5.). Akan tetapi tidak seperti pada ruang 2 dimensi yang memiliki hasil setara bila dibalik, sedangkan pada ruang tiga dimensi ada permasalahan dalam menentukan suatu posisi, jika kordinat z ditentukan maka ada dua kemungkinan yang terjadi dan mengarah kearah yang berbeda. Ini menyimpulkan bahwa semua ruang koordinat tidak setara, dan ada dua tipe berbeda dari koordinat ruang tiga dimensi yaitu ruang koordinat tangan kiri (*left-handed*) dan ruang koordinat tangan kanan (*right-handed*). Direct3D menggunakan sistem ruang koordinat tangan kiri sebagai acuan dalam penggunaan dan penentuan lokasi dalam ruang tiga dimensinya.

Masing-masing obyek dalam dunia virtual 3D memiliki ruang koordinat beserta titik asal dan dan sumbunya masing-masing yang disebut dengan ruang obyek (*object space*). Dengan koordinat ruang obyek yang independen maka dapat mudah melakukan perubahan pada obyek tersebut seperti memperbesar atau memperkecil obyek. Jika semua obyek sudah siap selanjutnya obyek itu akan digabung kedalam satu ruang koordinat yaitu *world space*. Di dalam world space masing-masing ruang koordinat obyek tersebut akan ditransformasikan terhadap koordinat obyek lain seperti ukuran, orientasi, bahkan jarak antar masing-masing obyek. Dan terakhir world space akan ditransformasikan lagi ke ruang koordinat kamera (*camera space*). Pada tahapan ini akan menentukan obyek-obyek mana yang masuk kedalam luang lingkup yang terlihat oleh kamera. Dalam ruang kamera sumbu positif x akan mengarah kekanan, sumbu positif y keatas dan

sumbu positif z mengarah kedalam untuk sistem koordinat tangan kiri (atau sistem koordinat tangan kanan sumbu negatif z mengarah kedalam).

2.6.3. Matriks dan Transformasi

Transformasi adalah sebuah operasi yang membutuhkan entitas seperti titik, vektor, atau warna dan mengkonversi mereka dalam beberapa cara (Möller, Haines dan Hoffman 2008). Dengan tranformasi dapat memposisikan, mengubah bentuk, dan melakukan animasi pada obyek, cahaya (*light*), dan kamera. Dan juga dapat memastikan bahwa semua perhitungan dilakukan dalam sistem koordinat yang sama, dan melakukan proyeksi pada objek kedalam suatu bidang dengan cara yang berbeda.

Secara umum matriks bujur sangkar (*square matrix*) dapat mendefinisikan setiap transformasi linear. Secara geometri dapat dikatakan bahwa transformasi linear mempertahankan garis lurus dan paralelnya, dan tidak melakukan translasi, yang berarti bahwa titik asal tidak berubah. Ketika tansformasi linear mempertahankan garis lurusnya, sifat-sifat yang lain dari geometri seperti panjang, sudut, luas, dan volumenya kemungkinan dapat berubah oleh transformasi, contohnya antara lain seperti rotasi, *scaling*, refleksi, proyeksi *orthogonal*, dan *shearing*.

Transformasi rotasi adalah tranformasi yang memutar vektor (baik posisi atau arah) berdasarkan sudut yang diberikan disekitar sumbu melalui titik asal. Persamaan dari transformasi rotasi bedasarkan sudut θ untuk masing-masing sumbu x, y dan z adalah sebagai berikut:

$$\mathbf{R}_x(\theta) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & cos\theta & -sin\theta & 0 \\ 0 & sin\theta & cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$\mathbf{R}_y(\theta) = \begin{pmatrix} cos\theta & 0 & sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -sin\theta & 0 & cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$\mathbf{R}_z(\theta) = \begin{pmatrix} cos\theta & -sin\theta & 0 & 0 \\ sin\theta & cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Selain transformasi linear juga ada bermacam-macam transformasi lainnya seperti transformasi affine untuk melakukan translasi (memindahkan posisi obyek dari titik asal) dengan menggunakan matriks homogen 4x4. Dengan memakai matriks homogen 4x4, translasi dapat dikombinasikan dengan transformasi linear lainnya. Persamaan dari matriks homogen 4x4 untuk melakukan translasi dalam tiga dimensi berdasarkan vektor **t** = (t_x, t_y, t_z) adalah sebagai berikut:

$$\mathbf{T}(\mathbf{t}) = \mathbf{T}(t_x, t_y, t_z) = \begin{pmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

2.6.4. Orientasi Euler Angle dan Quaternion

Seperti yang dibahas sebelumnya transformasi rotasi menggunakan matriks dapat mewakili orientasi dari obyek, akan tetapi ketika melakukan rotasi menggunakan matriks ada beberapa kelemahan yaitu membutuhkan lebih banyak memori, sangat sulit untuk digunakan secara intuitif, dan nilainya bisa cacat (akibat gabungan antar transformasi dan data eksternal). Maka ada teknik lain

selain matriks dalam mewakili orientasi yaitu menggunakan sudut Euler (*Euler angle*), dan Quaternion.

Sudut Euler merupakan teknik yang ditemukan oleh matematikawan terkenal yang bernama Leonhard Euler (1707-1783) yang membuktikan bahwa perpindahan sudut berurutan setara dengan perpindahan sudut tunggal. Ide dasarnya adalah untuk mendefinisikan perpindahan sudut berurutan dari tiga rotasi terhadap tiga sumbu yang saling tegak lurus. Tiga rotasi itu disebut *heading*, *pitch*, *bank* masing-masing terhadap sumbu positif x kekanan, positif y keatas, dan positif z kedepan (menggunakan sistem koordinat tangan kiri). Dengan cara tersebut akan lebih intuitif bagi manusia dalam melakukan rotasi terhadap suatu obyek, contohnya untuk mengukur rotasi navigasi pesawat terbang. Tapi teknik ini juga memiliki kelemahan yaitu terjadinya *Gimbal Lock*. Gimbal lock adalah hilangnya satu derajat kebebasan dalam ruang tiga dimensi yang terjadi ketika dua sumbu didorong kedalam konfigurasi paralel, sehingga sistem akan terkunci kedalam rotasi yang menurun ke dua dimesi.

Möller, Haines dan Hoffman (2008) menjelaskan bahwa *quaternion* adalah alat yang sangat ampuh untuk membangun transformasi dengan fitur yang menarik, dan dalam beberapa hal lebih unggul dari sudut Euler dan matriks, terutama mengenai rotasi dan orientasi (termasuk mengatasi gimbal lock). Pertama kali ditemukan oleh Sir William Hamilton pada tahun 1843 sebagai perluasan dari bilangan kompleks, dengan persamaan matematika sebagai berikut:

$$\hat{\mathbf{q}} = (\mathbf{q}_v, q_w) = iq_x + jq_y + kq_z + q_w = \mathbf{q}_v + q_w,$$

$$\mathbf{q}_v = iq_x + jq_y + kq_z = (q_x, q_y, q_z),$$

$$i^2 = j^2 = k^2 = -1, jk = -kj = i, ki = -ik = j, ij = -ji = k$$

Variabel q_w merupakan bagian real dari quaternion $\hat{\mathbf{q}}$. Sedangkan bagian imajiner adalah $\mathbf{q}_v$, dan *i*, *j*, dan *k* dinamakan unit imajiner.

Dengan persamaan diatas maka proses perhitungan matematika menjadi lebih luas sehingga terbuka untuk terbentuknya rumus-rumus perhitungan matematika yang baru, tidak terkecuali pada perhitungan ruang tiga dimensi. Quaternion memiliki mempunyai beragam operasi antara lain seperti mencari negasi, magnitude, identitas, perkalian, konjugasi, invers dan sebagainya. Untuk menentukan rotasi vektor $\mathbf{p} = (p_x\, p_y\, p_z\, p_w)^T$ terhadap sumbu vektor $\mathbf{u}_q$ berdasarkan sudut θ, dan diasumsikan bahwa unit quaternion $\hat{\mathbf{q}} = (\sin\theta \mathbf{u}_q, \cos\theta)$, maka persamaannya sebagai berikut:

$$\hat{\mathbf{q}}\mathbf{p}\hat{\mathbf{q}}^{-1}$$

Rotasi dengan menggunakan matriks, sudut Euler dan quaternion masing-masing memiliki kelebihan dan kelemahan tersendiri. Akan tetapi teknik-teknik tersebut dapat dikonversikan antara satu dengan yang lain sehingga dapat digunakan secara optimal sesuai kondisi dan kebutuhan.

2.6.5. Plane dan Ray

Plane (bidang) dalam 3D adalah sekumpulan dari titik-titik yang berjarak sama (*equidistant*) dari dua titik (Dunn dan Parberry 2002). Secara geometri plane adalah permukaan 2 dimensi yang datar, tidak memiliki ketebalan, dan melebar tak terbatas. Persamaan dari plane umumnya ditulis sebagai berikut:

$$Ax + By + Cz + D = 0$$

dimana komponen x, y, z merupakan bagian dari titik **p** = (*x*, *y*, *z*), yang berada di dalam plane, serta komponen vektor normal **n** = [*A*, *B*, *C*] yang tegak lurus

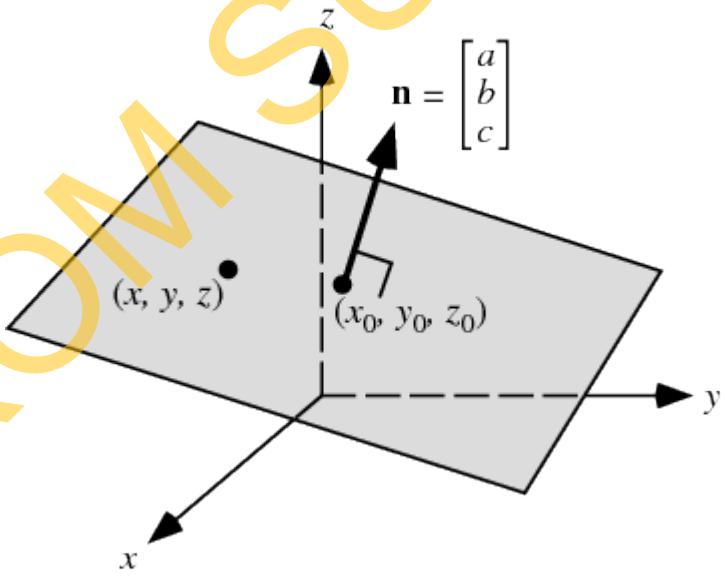
dengan plane. Berdasarkan persamaan tersebut maka untuk menentukan jarak ke titik asal *D*, adalah dengan persamaan di bawah ini:

$$D = -(\boldsymbol{n}.\boldsymbol{p})$$

Cara lain untuk mendefinisikan plane adalah dengan memberikan 3 titik *noncollinear* yang tidak pada garis lurus yang sama. Untuk mencari vektor normal **n** dari ketiga titik noncollinear **p1**, **p2**, dan **p3** adalah dengan persamaan sebagai berikut:

$$\boldsymbol{n} = (\boldsymbol{p2} - \boldsymbol{p1}) \times (\boldsymbol{p3} - \boldsymbol{p1})$$

Sedangkan untuk mencari jarak ke titik asal *D* yaitu menggunakan rumus yang sama seperti sebelumnya akan tetapi titik **p** digantikan oleh titik **p1**.



Gambar 2.6. Plane berdasarkan titik dan vektor normal
[Sumber: http://mathworld.wolfram.com/images/eps-gif/Plane_1001.gif
Time Download: 19/07/2013 15:54:12]

Jika ingin menghitung jarak *a* antara suatu titik **q** yang tidak berada di dalam plane dengan plane, berdasarkan titik **p** yang saling tegak lurus dengan titik **q,** dan vektor normal **n** adalah dengan rumus sebagai berikut:

$$a = \boldsymbol{q}.\boldsymbol{n} - D$$

apabila *a* hasil perhitungan rumus diatas adalah positif maka titik **q** berada di depan plane (*front face*), dan sebaliknya bila negatif maka berada di belakang plane (*back face*).

Ray adalah garis yang memiliki titik asal (*origin*) dan memanjang tak terbatas dalam satu arah (Dunn dan Parberry 2002). Ray didefinisikan secara matematika dengan persamaan sebagai berikut:

$$\boldsymbol{P}(t) = \boldsymbol{S} + t\boldsymbol{V}$$

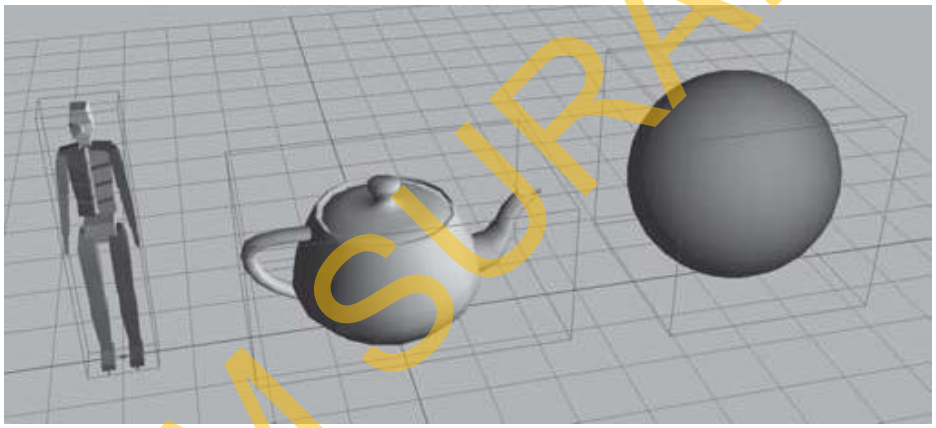
dimana **S** mewakili posisi awal dari ray dan **V** mewakili arah dimana ray menuju. Sedangkan parameter *t* pada kasus tertentu memiliki nilai dari 0..*l*, dan *l* sebagai panjang dari ray.

2.6.6. Bounding Volume

Bounding volume dibangun agar dapat menutupi semua verteks milik dari sebuah *triangle mesh* (sekumpulan segitiga yang membentuk obyek tiga dimensi), maka dengan demikian dapat memastikan bahwa setiap segitiga di dalam mesh tersebut masuk ke dalam bounding volume (Lengyel 2004). Bounding volume digunakan untuk meningkatkan efisiensi dari operasi geometri dengan menggunakan volume yang sangat sederhana untuk melingkupi obyek tiga dimensi yang lebih kompleks. Biasanya, volume yang lebih sederhana memiliki

cara yang juga lebih sederhana untuk menguji obyek yang saling tumpang tindih (*overlap*).

Salah satu bounding volume yang paling umum digunakan adalah *Axially Aligned Bounding Box* (AABB). Dalam 3D AABB adalah kotak bersegi enam sederhana, dimana masing-masing sisinya paralel ke salah satu *cardinal* plane yaitu suatu plane (bidang) dimana titik pusatnya melewati seperti ketika individu berdiri di posisi anatomis. Gambar di bawah ini menunjukkan gambar obyek 3D sederhana berserta dengan AABBnya.



Gambar 2.7. Obyek 3D dan AABBnya.
[Sumber: Dunn dan Parberry, 2002]

Apabila suatu titik berada di dalam AABB, maka koordinatnya berada di dalam persamaan di bawah ini:

$$x_{min} \leq x \leq x_{max}$$
$$y_{min} \leq y \leq y_{max}$$
$$z_{min} \leq z \leq z_{max}$$

Berdasarkan dari persamaan diatas maka dapat menentukan 2 titik ekstrim yang menandakan batas dari AABB dengan persamaan sebagai berikut:

$$\boldsymbol{p}_{min} = [x_{min}\ y_{min}\ z_{min}]$$
$$\boldsymbol{p}_{max} = [x_{max}\ y_{max}\ z_{max}]$$

Dengan demikian titik pusat **c** dari AABB tersebut dapat dihitung dengan persamaan di bawah ini:

$$\boldsymbol{c} = (\boldsymbol{p}_{min} + \boldsymbol{p}_{max}) / 2$$

Berbeda dengan teknik OBB (*Oriented* Bounding Box), bahwa OBB harus menyimpan vektor lokalnya (sebagai sumbu) agar dapat mengetahui orientasinya (Zerbst dan Düval 2004). Vektor tersebut harus menjadi vektor arah yang dinormalisasi. Kemudian menyimpan separuh dari perpanjangan bounding box pada masing-masing sumbunya. Dan informasi terakhir yang dibutuhkan dari OBB adalah titik pusatnya.

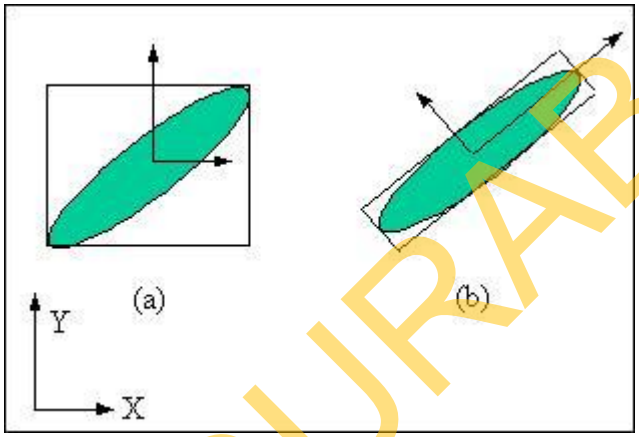
Apabila sebuah OBB didefinisikan berdasarkan titik pusat **c**, dengan sumbu *right-handed orthonomal* $\mathbf{A}_0$, $\mathbf{A}_1$, $\mathbf{A}_2$, dan perpanjangan $a_0 > 0$, $a_1 > 0$, $a_2 > 0$, maka dapat digambarkan ke delapan verteks dari OBB tersebut dengan persamaan di bawah ini:

$$\boldsymbol{C} + \sum_{i=0}^{2} \sigma_i a_i \boldsymbol{A}_i$$

diimana $|\sigma_i| = 1$ untuk semua *i*

Walaupun AABB sangat mudah untuk dibuat dan mampu meningkatkan kemampuan untuk menguji obyek yang saling tumpang tindih, akan tetapi AABB

bukan merupakan solusi bounding volume yang optimal. Hal ini dibuktikan oleh OBB, walaupun memerlukan perhitungan yang lebih banyak sehingga dapat mempengaruhi performanya, akan tetapi OBB menghasilkan bounding volume yang lebih erat dengan obyek 3D yang dilingkupinya sehingga meminimalisir tingkat kesalahannya. Contoh perbedaan AABB dan OBB dapat dilihat pada gambar dibawah ini:



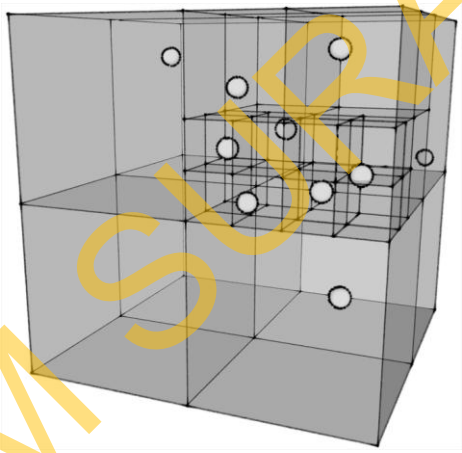
Gambar 2.8. Perbedaan bounding volume (a) AABB, dan (b) OBB
[Sumber: http://portal.ku.edu.tr/~cbasdogan/Tutorials/imageU21.JPG
Time Download: 19/07/2013 15:20:09]

2.7. Octree

Foley, van Dam, Feiner, dan Hughes (1996), menyimpulkan bahwa octree merupakan varian secara hirarki dari enumerasi penempatan spasial, pendekatan tersebut dirancang untuk ditujukan mengatasi kebutuhan persyaratan penyimpanan. Octree merupakan turunan dari konsep *quadtree*, yang merupakan representasi dua dimensi dari format yang digunakan untuk *encoding* gambar.

Sebuah kotak akan dibagi secara simultan disepanjang semua ketiga sumbunya, dan titik pemotongan harus berada pusat dari kotak. Ini akan membuat delapan kotak baru, sehingga dinamakan octree (diagram pohon dengan delapan simpul anak).

Ketika membagi kotak (dalam hal ini adalah ruang tiga dimensi), obyek-obyek berdasarkan kotak tersebut dimasukkan kedalam simpul (*node*) pada diagram pohon (*tree*). Untuk menempatkan obyek kedalam octree, dimulai dari simpul akar (*root*). Jika obyek sepenuhnya di dalam satu simpul anak (child), maka turunkan kedalam simpul anak tersebut. Lanjutkan penelusuran kebawah dari diagram pohon selama obyek sepenuhnya berada di dalam simpul anak atau sudah mencapai simpul daun (*leaf*). Jika sebuah obyek tercakup diantara dua bidang yang terpisah, maka penurunan kebawah dihentikan dan mendaftarkan obyek tersebut kedalam simpul berdasarkan pada tingkatannya (*level*).



Gambar 2.9. Mengelompokkan Obyek dengan Octree

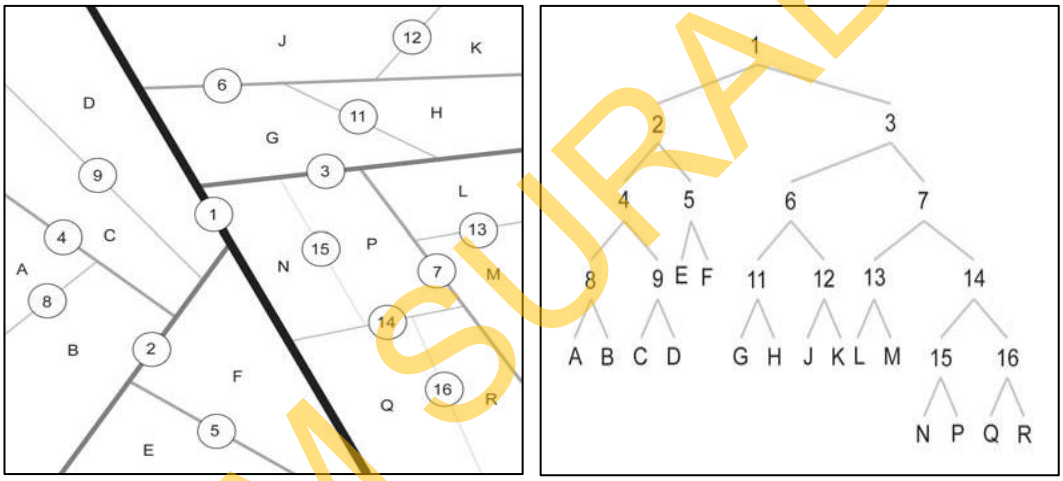
Ketika obyek dimasukkan atau didaftarkan kedalam simpul, maka octree menjadi alat yang sangat ampuh dalam mencari lokasi obyek dalam radius tertentu dari titik yang tentukan, atau untuk melakukan *culling* (penyisihan) pada obyek untuk rendering dan *collision detection* (deteksi benturan).

2.8. BSP Tree

BSP merupakan kependekan dari *binary space partitioning*. Seperti yang dapat dibayangkan berdasarkan namanya, BSP adalah struktur diagram pohon

dimana setiap simpulnya memiliki dua anak. BSP tree secara rekursif membagi ruang kedalam sepasang sub-ruang, dan masing-masing dipisahkan oleh bidang (*plane*) dengan orientasi dan posisi yang sewenang-wenang (*arbitrary*) (Foley, van Dam, Feiner, dan Hughes 1996). Tidak seperti octree bidang pemisah BSP tree tidak harus sejajar sumbu (*axially aligned*).

Sebagai contoh dari BSP tree yang ditunjukkan pada gambar 2.10, ketebalan pada garis mewakili bidang pemisah dengan tingkatan yang lebih lebih tinggi dalam diagram pohon.



Gambar 2.10. Hirarki dari BSP Tree
[Sumber: Dunn dan Parberry, 2002]

Dalam gambar tersebut simpul diberi label beserta struktur diagram pohon yang sebenarnya. Simpul interior memakai angka dan simpul daun memakai huruf, akan tetapi sangat penting untuk dipahami bahwa masing-masing simpul mewakili sebuah area dari ruang, bahkan termasuk simpul interior. Dalam praktek untuk melacak simpul anak depan dan belakang dari bidang pemisah, ditentukan dengan menggunakan *normal* dari bidang.

Seperti octree obyek disimpan kedalam simpul-simpul BSP tree yang menurun sejauh mungkin. Untuk memproses obyek dalam diagram pohon, dimulai dari simpul akar dan memproses semua obyek di dalam simpul tersebut. Kemudian menentukan apakah area yang diinginkan (untuk rendering, collision detection dll) berada sepenuhnya pada satu sisi dari bidang pemisah atau pada sisi yang lainnya. Jika hanya menginginkan isi volume dari ruang pada satu sisi bidang pemisah, maka isi seluruh cabang pada sisi lainnya dapat dihiraukan. Dan jika area yang diinginkan terbentang diantara bidang pemisah, maka kedua simpul anak tersebut harus diproses.

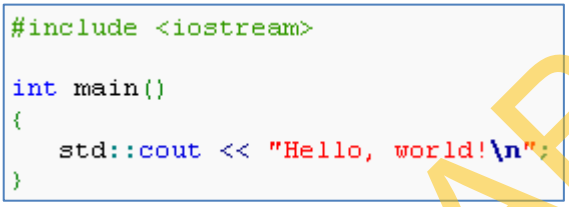
Menggunakan BSP tree ketika telah dibangun sangatlah mudah. Kuncinya adalah menentukan penempatan bidang pemisah. Dan dalam menentukan bidang pemisah tersebut jauh lebih fleksibel daripada menggunakan octree.

2.9. Pemrograman C++

C++ dimulai sebagai versi perluasan dari bahasa C. C++ pertama kali ditemukan oleh Bjarne Stroustrup pada tahun 1979 di Laboratorium Bell Murray Hill, New Jersey (Schildt 2003). C++ seringkali dinamakan sebagai bahasa komputer tingkat menengah, karena mengkombinasikan elemen-elemen terbaik dari bahasa tingkat tinggi dengan kontrol dan fleksibilitas dari bahasa *assembly*. C++ mampu melakukan manipulasi secara langsung dari *bit*, *byte* dan alamat memori (*pointer*) dari elemen dasar beserta fungsi komputer, sehingga sangat cocok untuk pemrograman level sistem.

Pemrograman C++ juga merupakan bahasa pemrograman yang memiliki sifat pemrograman yang berorientasi obyek. Untuk menyelesaikan masalah, C++

melakukan langkah pertama dengan menjelaskan class-class yang merupakan anak class yang dibuat sebelumnya sebagai abstraksi obyek-obyek fisik. Class tersebut berisi keadaan obyek, anggota-anggotanya dan kemampuan dari obyeknya. Setelah beberapa class dibuat kemudian masalah akan dipecahkan dengan class. Gambar dibawah ini merupakan contoh sintaks program yang menggunakan bahasa C++.



```
#include <iostream>

int main()
{
    std::cout << "Hello, world!\n";
}
```

Gambar 2.11. Contoh sintaks bahasa C++

2.10. UML

"*Unified Modeling Language* (UML) adalah keluarga notasi grafis yang didukung oleh meta-model tunggal, yang membantu pendeskripsian dan desain sistem perangkat lunak, khususnya sistem yang dibangun menggunakan pemrograman berorientasi obyek (OO)." (Fowler 2005:1). UML mulai diperkenalkan *Object Management Group*, sebuah organisasi yang telah mengembangkan model, teknologi, dan standar OOP sejak tahun 1980-an. Sekarang UML sudah mulai banyak digunakan oleh para praktisi OOP.

UML terdiri dari sejumlah elemen grafis yang dikombinasikan untuk membentuk diagram. Karena merupakan bahasa, UML memiliki aturan untuk mengkombinasikan elemen tersebut. Macam-macam dari diagram UML tersebut antara lain:

1. *Class Diagram*

   Class diagram mendeskripsikan jenis-jenis obyek dalam sistem dan berbagai macam hubungan statis yang terdapat diantara mereka. Class diagram juga menunjukkan properti dan operasi sebuah class dan batasan-batasan yang terdapat dalam hubungan-hubungan obyek tersebut.

2. *Component Diagram*

   Component diagram mendeskripsikan bagaimana sebuah sistem perangkat lunak dibagi menjadi komponen-komponen dan menjelaskan hubungan antar komponen tersebut.

2.11. Pengujian Perangkat Lunak

Pengujian perangkat lunak merupakan suatu investigasi yang dilakukan untuk mendapatkan informasi mengenai kualitas dari produk atau layanan yang sedang diuji. Pengujian perangkat lunak juga memberikan pandangan mengenai perangkat lunak secara obyektif dan independen, yang bermanfaat dalam operasional bisnis untuk memahami tingkat resiko pada implementasinya.

Teknik-teknik pengujian mencakup, namun tidak terbatas pada, proses mengeksekusi suatu bagian program atau keseluruhan aplikasi dengan tujuan untuk menemukan *bug* perangkat lunak. Teknik-teknik pengujian perangkat lunak yang paling sering dipakai yaitu:

1. *White-box Testing*

   White-box testing merupakan metode pengujian perangkat lunak yang menguji struktur internal dari sebuah aplikasi, bukan pada fungsionalitasnya. White-box testing meramalkan cara kerja perangkat lunak secara rinci, karenanya jalur logika perangkat lunak akan diuji

dengan menyediakan *test case* yang akan mengerjakan kumpulan kondisi dan atau pengulangan secara spesifik.

2. *Black-box* Testing

   Black-box testing adalah metode pengujian perangkat lunak yang menguji fungsionalitas dari sebuah aplikasi. Pengujian ini bertujuan untuk menguji fungsionalitas dari operasi dari sebuah sistem, apakah pemasukan data keluaran telah berjalan sebagaimana yang diharapkan dan apakah informasi yang disimpan secara eksternal selalu dijaga kemutakhirannya.