BAB 3
DASAR TEORI

3.1. Simulasi Komputasi Fluida Dinamis

Simulasi adalah imitasi dari sistem atau proses yang terjadi dalam dunia nyata dalam serangkaian waktu (Banks, et al., 2004). Simulasi memiliki beberapa keuntungan seperti, mampu menjawab pertanyaan "bagaimana jika", memberikan hipotesa mengenai *bagaimana* dan *mengapa* suatu fenomena dapat terjadi, serta waktu fenomena yang sedang diamati dapat dipercepat maupun diperlambat. Simulasi dapat dilakukan dengan komputer. Simulasi komputer merupakan alat yang secara virtual mampu menginvestigasi perilaku sistem yang sedang dipelajari. Dengan mengubah beberapa variabel, simulasi ini dapat membuat prediksi.

Komputasi fluida dinamis (Computational Fluid Dynamics) merupakan sekumpulan metodologi yang memungkinkan komputer menyajikan simulasi numerik dari aliran fluida. Seluruh sistem, ditransformasikan ke dalam bentuk virtual, dan dapat divisualisakan melalui komputer (Hirsch, 2007). Komponen-komponen dalam komputasi fluida dinamis adalah sebagai berikut.

1. Pemilihan model matematis

   Pada tahap ini, ditentukan batasan dunia fisik yang akan disimulasikan, dan model matematika yang relevan. Model tersebut berbentuk persamaan diferensial parsial dan hukum-hukum tambahan sesuai dengan jenis fluida.

2. Diskritisasi

   Pada tahap ini, dilakukan diskritisasi spasial untuk menentukan ruang geometri (*mesh*), dan diskritisasi model persamaan untuk menentukan skema numerik.

3. Analisis Skema Numerik

   Skema numerik yang digunakan perlu dianalisis untuk memenuhi serangkaian kondisi dan aturan, dan menghasilkan akurasi dan stabilitas yang diinginkan.

4. Penyelesaian Numerik

   Solusi dari skema numerik harus diperoleh, dengan metode integrasi waktu tertentu.

5. Pemrosesan Grafis (*post-processing*)

   Pada tahap ini, data-data numerik hasil simulasi ditampilkan melalui visualisasi grafis agar dapat dimengerti dan diinterpretasikan.

Sementara, Versteeg dan Malalasekera (Versteeg & Malalasekera, 2007) membagi struktur pada komputasi fluida dinamis ke dalam tiga elemen berikut.

1. *Pre-Processor*

   Tahap ini berisi masukan (*input)* dari permasalahan aliran fluida, antara lain:

   - Pendefinisian domain komputasi, yaitu pendefinisian ruang geometri yang diinginkan.
   - Pembuatan/*generate grid*/*mesh*.
   - Pemilihan fenomena yang akan dimodelkan.
   - Pendefinisian atribut-atribut fluida.
   - Penentuan kondisi-kondisi batas yang diinginkan.

2. *Solver*

   Dengan metode *finite volume*, algoritma untuk penyelesaian numerik terdiri dari beberapa langkah berikut:

   - Integrasi persamaan aliran fluida yang digunakan, pada seluruh domain.
   - Diskretisasi, yaitu konversi dari persamaan integral ke dalam sistem persamaan aljabar.
   - Solusi persamaan aljabar dengan metode iteratif.

3. *Post-Processor*

   Tahap ini berupa visualisasi data-data hasil simulasi, mencakup:

   - Tampilan domain geometri dan *grid.*
   - Plot vektor.

- Plot garis dan bayangan.
- 2D dan 3D *surface plot.*
- *Particle Tracking.*
- *View manipulation* (*translation, rotation, scalling,* dan lain-lain).

Komputasi fluida dinamis sangat berguna di berbagai bidang baik industri maupun nonindustri. Beberapa contohnya adalah aerodinamik pesawat dan kendaraan, hidrodinamika kapal, pembangkit listrik, mesin turbo, rekayasa elektrik dan elektronik, rekayasa proses kimia, lingkungan eksternal dan internal bangunan, teknik kelautan, teknik lingkungan, hidrologi dan oseanografi, meteorologi, dan rekayasa biomedis.

3.2. Persamaan Diferensial Parsial

Persamaan diferensial parsial digunakan di seluruh bidang matematika terapan dan bisa dimanfaatkan untuk memodelkan beragam permasalahan praktis seperti peramalan cuaca, desain pesawat terbang, mobil berkecepatan tinggi, serta penilaian potensi investasi saham finansial (Griffiths, et al., 2015). Persamaan ini juga dapat digunakan untuk menjelaskan beragam sistem dalam dunia fisik, seperti mekanika fluida dan benda padat, evolusi populasi dan penyakit, serta fisika matematis (Shearer & Levy, 2015).

Diberikan sebuah fungsi u yang bergantung pada x dan y, turunan parsial dari u terhadap x di sembarang titik (x, y) didefinisikan dengan

$$\frac{\partial u}{\partial x} = \lim_{\Delta x \to 0} \frac{u(x + \Delta x, y) - u(x, y)}{\Delta x}$$

Serupa, turunan parsial u terhadap y di sembarang titik (x, y) didefinisikan sebagai

$$\frac{\partial u}{\partial y} = \lim_{\Delta y \to 0} \frac{u(x, y + \Delta y) - u(x, y)}{\Delta y}$$

Sebuah persamaan yang mengandung turunan parsial dari fungsi yang tidak diketahui, dengan dua atau lebih variabel bebas disebut dengan *persamaan*

diferensial parsial (Chapra & Canale, 2015). Contoh bentuk persamaan tersebut adalah sebagai berikut.

$$\frac{\partial^2 u}{\partial x^2} + 2xy\frac{\partial^2 u}{\partial y^2} + u = 1 \tag{3.1}$$

$$\frac{\partial^3 u}{\partial x^2 \partial y} + x\frac{\partial^2 u}{\partial y^2} + 8u = 5y \tag{3.2}$$

$$\left(\frac{\partial^2 u}{\partial x^2}\right)^3 + 6\frac{\partial^3 u}{\partial x^2 \partial y} = x \tag{3.3}$$

$$\frac{\partial^2 u}{\partial x^2} + xu\frac{\partial u}{\partial y} = x \tag{3.4}$$

Bentuk persamaan diferensial parsial dapat dikaji berdasarkan orde, linearitas, serta karakterisktiknya. *Orde* adalah tingkat tertinggi suku turunan. Sementara linearitas bergantung pada bentuk fungsi *u*, turunan *u*, dan koefisien persamaan tersebut. Suatu persamaan disebut fungsi linear jika fungsi tersebut linear pada *u* dan turunan *u*, serta koefisien persamaan tersebut hanya bergantung pada variabel bebas (*x* atau *y*) atau konstanta. Contoh klasifikasi orde dan linearitas persamaan diferensial parasial terdapat dalam Tabel 3.1 berikut.

Tabel 3.1. Klasifikasi Orde dan Linearitas Persamaan Diferensial Parsial

Persamaan	Orde	Linear
(3.1)	2	Ya
(3.2)	3	Ya
(3.3)	3	Tidak
(3.4)	2	Tidak

Persamaan diferensial parsial linear orde dua, dengan dua variabel bebas, dapat dikelompokkan menjadi eliptik, parabolik, dan hiperbolik. Beberapa persamaan tersebut dapat dinyatakan dalam bentuk umum berikut,

$$A\frac{\partial^2 u}{\partial x^2} + B\frac{\partial^2 u}{\partial x \partial y} + C\frac{\partial^2 u}{\partial x^2} - D = 0$$

dengan *A*, *B*, dan *C* adalah fungsi dari *x* dan *y*, dan *D* aalah sebauah fungsi dari *x*, *y*, u, ∂u, $\partial u/\partial x$, dan $\partial u/\partial y$.

Tabel 3.2. Klasifikasi Persamaan Diferensial Parsial Orde Dua - Linear

$B^2 - 4AC$	Klasifikasi
<0	Eliptik
=0	Parabolic
>0	Hiperbolik

Klasifikasi persamaan tersebut ditentukan berdasarkan nilai diskriminannya sesuai dengan Tabel 3.2 di atas. Persamaan *eliptik* biasa digunakan untuk sistem dengan karakteristik yang stabil (*steady-state*). Persamaan *parabolik*, menunjukkan bagaimana suatu fungsi bervariasi dalam ruang dan waktu. Beberapa kasus merujuk pada masalah penjalaran, yaitu bagaimana solusi menjalar atau berubah dalam waktu. Sementara untuk kategori *hiperbolik* juga merujuk penjalaran pada solusi, namun disertasi osilasi.

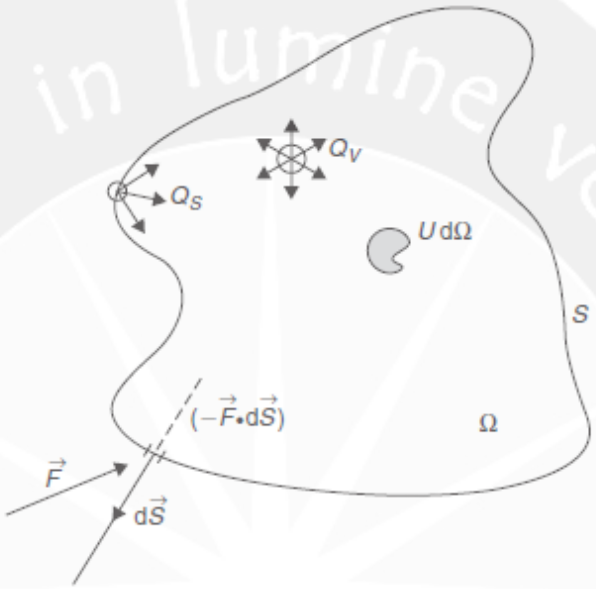
3.3.Bentuk Umum Hukum Konservasi

Hukum konservasi menjadi dasar dalam pemahaman mengenai dunia fisik, tentang proses yang dapat atau tidak dapat terjadi di alam. Menurut Hirch (Hirsch, 2007), hukum konservasi pada sebuah kuantitas *U* mengikuti aturan logis dan konsisten berikut

> *"The variation of the total amount of a quantity U inside a given domain is equal to the balance between the amount of that quantity entering and leaving the considered domain, plus the contributions from eventual sources generating that quantity."*

Perubahan total kuantitas *U* pada sebuah domain, sebanding dengan jumlah kuantitas yang masuk dan keluar pada domain tersebut, ditambah kontribusi dari beberapa sumber penghasil kuantitas tersebut. Jumlah kuantitas yang masuk dan keluar ini disebut dengan fluks.

Berdasarkan studi sifat fisik pada sistem aliran fluida, tidak semua aliran kuantitas mematuhi hukum konservasi. Seperti yang diketahui hingga kini, hukum-hukum yang menjelaskan tentang aliran fluida (dinamika fluida), didefinisikan oleh konservasi dari tiga kuantitas berikut, yaitu massa, momentum (produk dari densitas dan kecepatan), dan energi.



Gambar 3.1. Bentuk umum persamaan konservasi untuk kuantitas skalar.
Sumber: Hirch, 20076. Gambar telah diolah.

Suatu volume Ω, dibatasi oleh sebuah permukaan tertutup S. Simbol Ω disebut dengan *kontrol volume*, dan *S* disebut dengan *kontrol permukaan*. Jumlah total kuantitas *U* di dalam sebuah domain volume Ω, disimbolkan sebagai berikut.

$$\int_{\Omega} U\, d\Omega$$

Sementara perubahan (∂) per unit waktu (∂t) pada jumlah total kuantitas *U* di dalam Ω, disimbolkan sebagai berikut.

$$\frac{\partial}{\partial t}\int_{\Omega} U\, d\Omega$$

Total fluks merujuk pada hukum konservasi "jumlah kuantitas *U* yang masuk dan keluar pada domain". Fluks sendiri didefinisikan sebagai jumlah kuantitas *U* yang melintasi suatu unit permukaan per unit waktu. Fluks adalah vektor, yaitu besaran yang memiliki nilai dan arah. Jika vektor ini paralel dengan permukaan, maka tidak ada fluks yang akan memasuki domain. Oleh karena itu, hanya fluks yang searah dengan normal permukaan saja yang akan memasuki suatu domain, dan berkontribusi terhadap perubahan kuantitas *U*. Jadi, jumlah *U* yang melintasi permukaan suatu elemen $d\vec{S}$ per unit waktu, didefiniskan oleh produk skalar dari fluks dan elemen permukaan berikut.

$$F_n \mathrm{d}S = \vec{F} \cdot d\vec{S}$$

Dengan vektor elemen permukaan $d\vec{S}$ menunjuk sepanjang *normal arah keluar*. Total kontribusi dari fluks yang masuk adalah jumlah pada seluruh elemen permukaan $d\vec{S}$ dari permukaan tertutup S, dan disimbolkan sebagai berikut.

$$-\oint_S \vec{F} \cdot d\vec{S}$$

Tanda minus artinya, fluks berkontribusi positif ketika memasuki domain.

Selanjutnya sumber-sumber lain yang turut berkontribusi pada kuantitas *U*, dibagi menjadi *sumber volume* dan *sumber permukaan*, Q_v dan $\vec{Q}_s$ dan total kontribusinya berbentuk sebagai berikut.

$$\int_\Omega Q_v \, d\Omega + \oint_S \vec{Q}_s \cdot d\vec{S}$$

Berikut, bentuk umum hukum konservasi pada kuantitas *U*,

$$\frac{\partial}{\partial t}\int_\Omega U \, d\Omega = -\oint_S \vec{F} \cdot d\vec{S} + \int_\Omega Q_v \, d\Omega + \oint_S \vec{Q}_s \cdot d\vec{S}$$

yang biasanya ditulis sebagai berikut.

$$\frac{\partial}{\partial t}\int_{\Omega} U\, d\Omega + \oint_{S} \vec{F} \cdot d\vec{S} = \int_{\Omega} Q_v\, d\Omega + \oint_{S} \vec{Q}_s \cdot d\vec{S} \tag{3.5}$$

Teorema Gauss menyatakan bahwa integral permukaan dari fluks sama dengan integral volume dari divergen fluks tersebut,

$$\oint_{S} \vec{F} \cdot d\vec{S} = \int_{\Omega} \vec{\nabla}\vec{F}\, d\Omega$$

dengan catatan bahwa tiap volume Ω diselimuti oleh permukaan S, sehingga bentuk persamaan (3.5), dapat dinyatakan sebagai berikut.

$$\int_{\Omega} \frac{\partial U}{\partial t} d\Omega + \int_{\Omega} \vec{\nabla}\vec{F}\, d\Omega = \int_{\Omega} Q_v\, d\Omega + \int_{\Omega} \vec{\nabla}\vec{Q}_s\, d\Omega$$

Persamaan di atas diintegralkan pada domain yang sama, yaitu pada volume Ω, sehingga akan berlaku juga secara lokal di tiap titik pada domain tersebut. Dengan kata lain, persamaan di atas dapat dinyatakan dalam bentuk diferensial berikut.

$$\frac{\partial U}{\partial t} + \vec{\nabla}\vec{F} = Q_v + \vec{\nabla}\vec{Q}_s \tag{3.6}$$

Jika tidak ada sumber pada domain, maka $Q_v = Q_s = 0$, sehingga persamaan (3.6) berbentuk sebagai berikut.

$$\frac{\partial U}{\partial t} + \vec{\nabla}\vec{F} = 0$$

Fluks dihasilkan dari dua kontribusi, yaitu transpor konvektif dan difusi. Fluks konvektif $\vec{F}_C$, merepresentasikan jumlah kuantitas U yang diangkut oleh aliran dengan kecepatan $\vec{v}$,

$$\vec{F}_C = U\vec{v}$$

dengan $U = \rho u$, variabel u merupakan kuantitas per unit massa. Sementara Fluks difusi $\vec{F}_D$ adalah kontribusi yang dihasilkan fluida dalam kondisi tenang, berkenaan dengan efek makroskopik atau agitasi molekul,

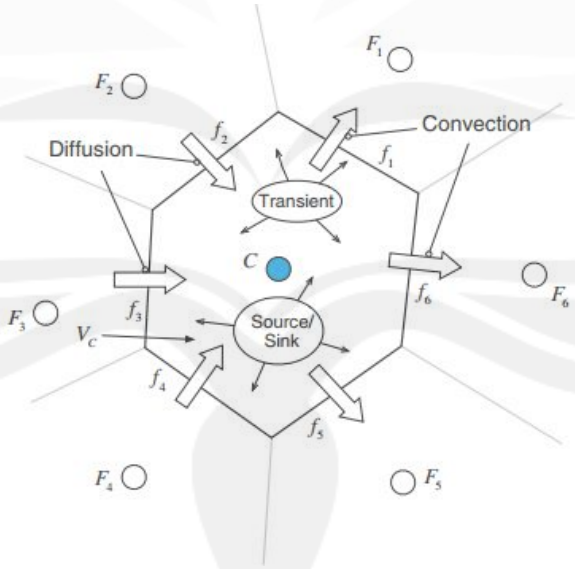
$$\vec{F}_D = -\kappa\rho\vec{\nabla}u$$

dengan κ adalah koefisien difusi., sehingga persamaan (3.6) dapat dinyatakan dalam bentuk berikut.

$$\frac{\partial U}{\partial t} + \vec{\nabla}.(\kappa\rho\vec{\nabla}u) = \vec{\nabla}.\left(\kappa\rho\vec{\nabla}u\right) + Q_v + \vec{\nabla}\vec{Q}_s$$

Persamaan di atas disebut juga persamaan transport dalam bentuk konservatif.

Moukalled dkk., (Moukalled, et al., 2016) mengilustrasikan bentuk persamaan transport konveksi difusi seperti gambar berikut, dengan F_i adalah elemen tetangga, f_i adalah sisi ke-*i* sel *C*, dan V_c adalah volume kontrol.



$$\frac{\partial U}{\partial t} + \vec{\nabla}.\left(\kappa\rho\vec{\nabla}u\right) = \vec{\nabla}.\left(\kappa\rho\vec{\nabla}u\right) + Q_v + \vec{\nabla}\vec{Q}_s$$

transient + convective = diffusive + source term

Gambar 3.2. Konservasi pada elemen diskret.
Sumber: Moukalled, 2016. Gambar telah diolah.

3.4.Persamaan Air Dangkal

Persamaan air dangkal atau sering disebut dengan SWE (Shallow Water Equation) digunakan untuk memodelkan aliran gelombang pada permukaan air yang dipengaruhi oleh gaya gravitasi, seperti aliran gelombang pada permukaan sungai, danau, lautan, ataupun dalam domain yang lebih kecil, misalnya pada permukaan air bak mandi. Persamaan air dangkal berlaku dengan syarat panjang gelombang jauh lebih besar dibandingkan dengan kedalamannya (Ahmad, et al., 2013). Dalam satu dimensi, *persamaan air dangkal* berbentuk sebagai berikut.

$$h_t + (uh)_x = 0$$

$$(hu)_t + (hu^2 + \frac{1}{2}gh^2)_x = 0$$

Sementara, dalam dua dimensi, persamaan ini memiliki bentuk sebagai berikut.

$$h_t + (hu)_x + (hv)_y = 0 \tag{3.7}$$

$$(hu)_t + (hu^2 + \frac{1}{2}gh^2)_x + (huv)_y = 0 \tag{3.8}$$

$$(hv)_t + (huv)_x + (hu^2 + \frac{1}{2}gh^2)_y = 0 \tag{3.9}$$

Konstata *g* merupakan gravitasi bumi, *h* adalah kedalaman dan (*u, v*) adalah vektor kecepatan alir, serta *hu* dan *hv* adalah momentum dalam dua arah (LeVeque, 2002). Persamaan di atas berlaku pada topografi dasar yang rata, dengan batasan tanpa memperhatikan temperatur, efek Coriolis, friksi, ataupun kekentalan fluida. Persamaan air dangkal adalah persamaan diferensial parsial nonlinear, orde dua, dan hiperbolik.

Persamaan (3.7) yang didasarkan pada hukum kekekalan massa, serta persamaan (3.8) dan (3.9) yang didasarkan pada hukum kekekalan momentum, dapat dinyatakan dalam bentuk vektor

$$q_t + f(q)_x + g(q)_y = 0 \tag{3.10}$$

dengan q merupakan vektor kuantitas, serta *f(q)* dan *g(q)* masing –masing adalah vektor fluks pada orientasi *x* dan *y*.

$$q = \begin{bmatrix} h \\ hu \\ hv \end{bmatrix}, \qquad f(q) = \begin{bmatrix} hu \\ hu^2 + \frac{1}{2}gh^2 \\ huv \end{bmatrix}, \qquad g(q) = \begin{bmatrix} hv \\ huv \\ hv^2 + \frac{1}{2}gh^2 \end{bmatrix}$$

Andai nilai-nilai elemen $f(q)$ dan $g(q)$ dinyatakan dalam bentuk $q^1 = h,\ q^2 = hu$, dan $q^3 = hv$, maka vektor fluks berbentuk sebagai berikut.

$$f(q) = \begin{bmatrix} q^2 \\ (q^2)^2/q^1 + \frac{1}{2}g(q^1)^2 \\ q^2q^3/q^1 \end{bmatrix}, \qquad g(q) = \begin{bmatrix} q^3 \\ q^2q^3/q^1 \\ (q^3)^2/q^1 + \frac{1}{2}g(q^1)^2 \end{bmatrix}$$

Persamaan (3.10) juga dapat dinyatakan dalam bentuk *quasilinear*,

$$q_t + f'(q)q_x + g'(q)q_y = 0$$

dengan $f'(q)$ dan $g'(q)$ merupakan matriks fluks Jacobian.

$$f'(q) = \begin{bmatrix} \partial f^1/\partial q^1 & \partial f^1/\partial q^2 & \partial f^1/\partial q^3 \\ \partial f^2/\partial q^1 & \partial f^2/\partial q^2 & \partial f^2/\partial q^3 \\ \partial f^3/\partial q^1 & \partial f^3/\partial q^2 & \partial f^3/\partial q^3 \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 \\ -u^2 + gh & 2u & 0 \\ -uv & v & u \end{bmatrix}$$

$$g'(q) = \begin{bmatrix} \partial g^1/\partial q^1 & \partial g^1/\partial q^2 & \partial g^1/\partial q^3 \\ \partial g^2/\partial q^1 & \partial g^2/\partial q^2 & \partial g^2/\partial q^3 \\ \partial g^3/\partial q^1 & \partial g^3/\partial q^2 & \partial g^3/\partial q^3 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 1 \\ -uv & v & u \\ -v^2 + gh & 0 & 2v \end{bmatrix}$$

Matriks Jacobian $f'(q)$ memiliki nilai eigen dan vektor eigen,

$$\lambda^{x1} = u - c, \qquad \lambda^{x2} = u, \qquad \lambda^{x3} = u + c$$

$$r^{x1} = \begin{bmatrix} 1 \\ u - c \\ v \end{bmatrix}, \qquad r^{x2} = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}, \qquad r^{x3} = \begin{bmatrix} 1 \\ u + c \\ 1 \end{bmatrix},$$

sementara matriks Jacobian g’(q) juga memiliki nilai eigen dan vektor eigen

$$\lambda^{y1} = v - c, \qquad \lambda^{y2} = v, \qquad \lambda^{y3} = v + c$$

$$r^{y1} = \begin{bmatrix} 1 \\ u \\ v - c \end{bmatrix}, \qquad r^{y2} = \begin{bmatrix} 0 \\ -1 \\ 0 \end{bmatrix}, \quad r^{y3} = \begin{bmatrix} 1 \\ u \\ v + c \end{bmatrix},$$

dengan $c = \sqrt{gh}$, yaitu kecepatan gelombang gravitasi pada air dangkal.

3.5. Diskritisasi

Komputer hanya dapat mengenali angka, sehingga model matematis dan geometris harus ditransformasi ke bentuk angka-angka perhitungan. Proses transformasi ini dinamakan diskritisasi. Terdapat dua komponen utama diskritisasi (Hirsch, 2007), yaitu:

1. Diskritisasi ruang/spasial.

   Pada bagian ini, ditentukan bentuk dan batasan ruang geometri yang akan digunakan dalam simulasi. Kemudian dilakukan pendistribusian titik-titik di seluruh permukaan/daerah dalam domain geometri tersebut. Himpunan titik-titik ini, yang menggantikan kontinuitas pada ruang nyata dengan sejumlah titik-titik terisolasi (*isolated point*), dinamakan *grid* atau *mesh*.
2. Diskritisasi model persamaan matematika.

   Pada diskritisasi ini, bentuk derivatif pada persamaan deferensial parsial akan ditransformasi menjadi beberapa operasi aritmatik. Hasilnya, akan diperoleh sekumpulan relasi aljabar antara nilai-nilai pada titik/sel *mesh* (*mesh point values*) yang saling bertetangga. Relasi-relasi ini dinamakan skema numerik. Skema numerik sendiri, dapat dikonstruksi menggunakan berbagai macam metode seperti *finite different*, *finite volume*, dan *finite element*.

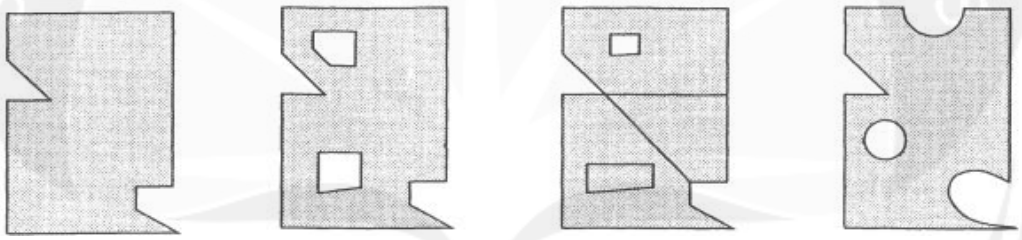
Dalam penyelesaian skema numerik, biasanya juga dilakukan diskritisasi terhadap waktu (*temporal discretization*). Beberapa contoh metode yang dapat digunakan antara lain *forward Euler*, *Leap-Frog*, *Adams-Bashforth* (Shirkhani, et al., 2015), dan *Runge-Kutta* (Brodtkorb, et al., 2012).

3.6. Mesh

Mesh merupakan diskritisasi ruang geometri dalam bentuk potongan-potongan sederhana seperti segitiga, kuadrilateral (dua dimensi), heksahedral atau tetrahedral (tiga dimensi). *Mesh* digunakan di berbagai bidang terapan, seperti geografi, kartografi, grafika komputer, dan utamanya sangat penting dalam penyelesaian numerik pada persamaan diferensial parsial.

Terdapat empat jenis domain geometri pada *mesh* dua dimensi (Bern & Plassman, 2000), yaitu:

1. *Simple polygon*
2. *Polygon with holes*
3. *Multiple domain*
4. *Curve domains*



a. b. c. d.

Gambar 3.3. Empat jenis domain pada *mesh* dua dimensi.

Sumber: Benn & Plassman, 2000.

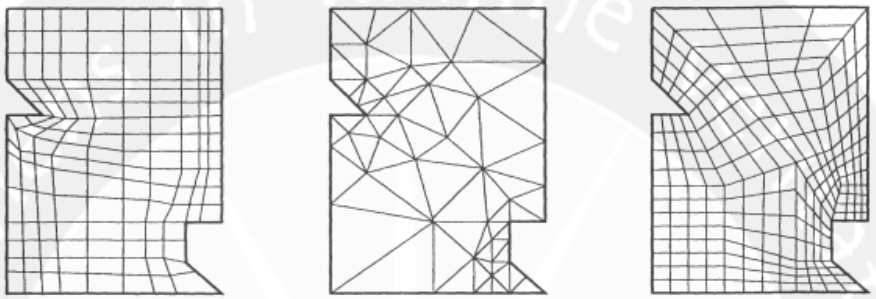
Menurut Bern dan Plassman, tiga domain pertama (*simple polygon*, *polygon with holes*, *multiple domain*) disebut dengan *polygonal domains*. Sebuah *simple polygon* mencakup batas tepi (*boundary*) dan interior (Gambar 3.3.a). Sementara *polygon with holes* adalah sebuah *simple polygon* dikurangi interior dari beberapa *simple polygon* lain (Gambar 3.3.b), dan batas tepinya memiliki lebih dari sebuah komponen terhubung. Pada *multiple domain*, terdapat batas-batas internal. Model objek pada domain ini, terbentuk oleh lebih dari satu material (Gambar 3.3.c).

Domain ke empat, yaitu *curved domains*, memiliki bagian sisi yang berbentuk kurva, contohnya seperti *spline*. Daerah pada domain ini mencakup batas tepi, dan

bagian interior dengan atau tanpa lubang (*holes*) ataupun batas-batas internal (Gambar 3.3.d).

Berdasarkan strukturnya, *mesh* memiliki jenis-jenis sebagai berikut.

1. Mesh tersturktur
2. Mesh tidak terstruktur
3. Mesh blok-terstruktur/*hybrid*



a. b. c.

Gambar 3.4. Tiga jenis struktur *mesh*.
Sumber: Benn & Plassman, 2000.

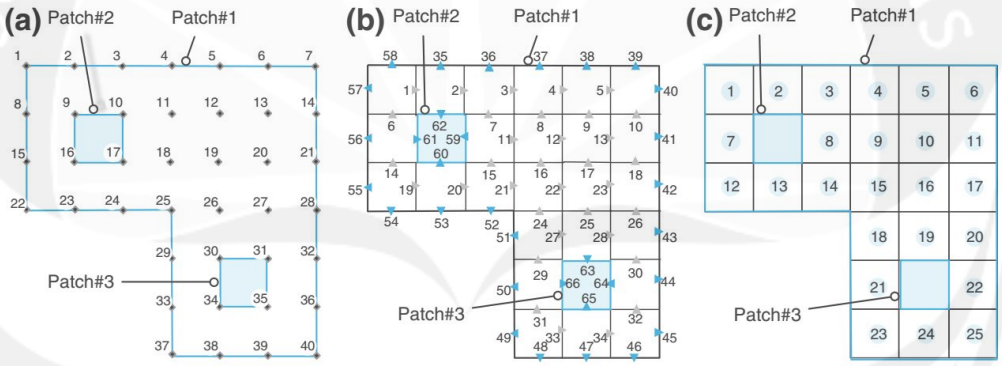
Simpul-simpul (*vertices*) interior pada *mesh* terstruktur memiliki topologi yang sama (Gambar 3.4.a). Sebaliknya, pada *mesh* tidak terstruktur, simpul-simpulnya memiliki variasi yang sembarang (Gambar 3.4.b). Sementara itu, pada blok-terstruktur/*hybrid*, *mesh* dibentuk oleh sejumlah kecil bagian terstruktur, yang secara keseluruhan. dikombinasikan dalam pola tidak terstruktur (Gambar 3.4.b).

Secara umum, mesh terstruktur lebih sederhana dan data-data yang dipetakan menggunakan mesh ini, lebih mudah untuk diakses. Sementara mesh tidak terstruktur lebih adaptif (*refinement*/*derefinement*) dan cocok digunakan pada domain dengan bentuk geometri yang kompleks. *Mesh* tidak terstruktur biasanya menggunakan elemen berbentuk segitiga, sementara *mesh* terstruktur biasanya menggunakan kuadrilateral. Ada banyak pendekatan yang digunakan untuk menghasilkan mesh tidak terstruktur. Beberapa diantarnya adalah *Dealunay triangulation, constrained Dealunay triangulation,* dan *quadtrees*.

3.7. Topologi Mesh

Informasi mengenai topologi mesh dibutuhkan dalam proses dikritisasi ruang/spasial. Informasi tersebut antar lain relasi elemen ke elemen, relasi sisi ke elemen, permukaan (*surface*), sentroid suatu elemen, sentroid suatu sisi, area, dan arah normal (Moukalled, et al., 2016).

Titik (*point*) atau simpul (*vertices*) merupakan tingkat paling dasar yang merepresentasikan lokasi dalam ruang geometri. Sebuah elemen dikelilingi dan dibatasi/disekat oleh beberapa sisi (*face*). Tiap-tiap sisi pada suatu elemen juga merupakan sekat bagi sebuah elemen lainnya kecuali pada bagian tepi (*boundary*). Bagi suatu elemen, elemen lain ini disebut dengan elemen tetangga. Berikut contoh gambar yang merepresentasikan komponen-komponen yang terdapat pada *mesh*, yaitu simpul, sisi, dan elemen.



Gambar 3.5. Komponen-komponen *mesh*.
Sumber: Moukalled dkk., 2016.

Mesh pada gambar di atas berbentuk *quadrilateral*. Patch#1, Patch#2, dan Patch#3 adalah tepi-tepi batas (*boundary*). Gambar 3.5.a menunjukkan pesebaran simpul dalam domain spasial yang dibatasi oleh garis-garis tepi berwarna biru. Simpul-simpul ini diberi indeks nomor mulai dari 1 hingga 40. Gambar 3.5.b menunjukkan komponen sisi, yang diberi indeks mulai dari 1 hingga 66. Sementara Gambar 3.5.c menunjukkan elemen-elemen *mesh*, yang diberi indeks mulai dari 1 hingga 25. Sejumlah bidang arsir berwarna biru menandakan keberadaan lubang (*holes*) dalam domain spasial.

Terdapat beberapa jenis relasi pada topologi mesh, yaitu relasi pada konektifitas elemen, konektifitas sisi, dan konektifitas simpul. Pada konektifitas elemen, terdapat tiga relasi, yaitu

1. Elemen ke elemen.

   Relasi ini menyatakan, sebuah elemen terhubung ke elemen-elemen lain yang merupakan tetangganya.
2. Elemen ke sisi.

   Relasi ini menyatakan, sebuah elemen dibatasi/disekat oleh beberapa sisi.
3. Elemen ke simpul.

   Relasi ini menyatakan bahwa sebuah elemen memiliki beberapa simpul.

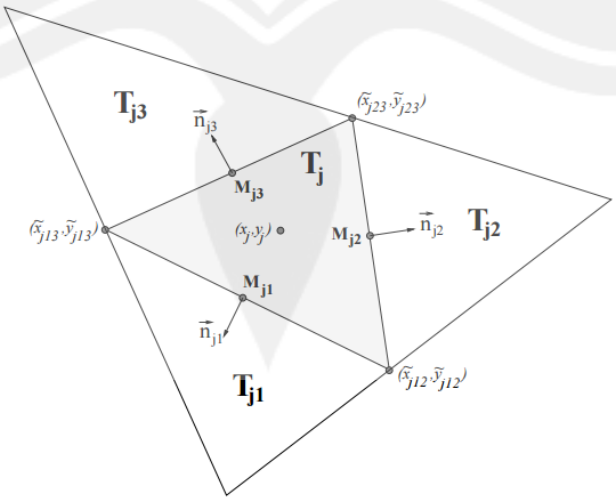
Konektifitas simpul, berguna untuk *post processing*, dan perhitungan gradien. Pada konektifitas simpul, terdapat beberapa relasi, yaitu

1. Simpul ke elemen.

   Relasi ini menyatakan bahwa sebuah simpul digunakan secara bersama-sama oleh beberapa elemen.
2. Simpul ke sisi.

   Relasi ini menyatakan bahwa sebuah simpul digunakan secara bersama-sama oleh beberapa sisi.

Pada mesh segitiga, komponen-komponen mesh dapat direpresentasikan melalui gambar berikut.



Gambar 3.6. Sebuah sel segitiga dengan 3 buah tetangga.

Sumber: Bryson dkk., 2011.

Dalam buku Handbook of Mathematics (Vialar, 2015), dinyatakan bahwa "*Triangulation is the division of a surface or plane polygon into a set of triangle, usually with the restriction that each triangle side entirely shared by two adjacent triangle*".

Kutipan di atas bermaksud mendefinisikan istilah 'triangulasi'. Triangulasi merupakan pembagian suatu permukaan atau bidang yang berbentuk poligon ke dalam sekumpulan segitiga. Pada triangulasi, biasanya tiap-tiap sisi yang terdapat pada suatu segitiga seluruhnya digunakan bersama-sama oleh dua segitiga yang saling berdekatan. Dengan kata lain, pada dua buah segitiga yang saling berdekatan, masing-masing segitiga memiliki sebuah sisi yang saling berhimpit dan menyatu, sehingga kedua sisi tersebut dapat dinyatakan sebagai sebuah sisi yang digunakan secara bersama-sama oleh dua buah segitga tersebut.

Berdasarkan Gambar 3.6 di atas (Bryson, et al., 2011), diasumsikan bahwa sebuah triangulasi $\mathrm{T} = \bigcup_j T_j$ pada domain komputasi, memiliki sel-sel yang berbentuk segitiga T_j, dengan luas area $|T_j|$. Unit-unit normal, yang mengarah ke luar, yang berasal dari tiap-tiap sisi pada segitiga T_j, dilambangkan dengan $\vec{n}_{jk} = \left(\cos\left(\theta_{jk}\right), \sin\left(\theta_{jk}\right)\right)$. Sementara, panjang (skalar) masing-masing sisi segitiga T_j tersebut dilambangkan dengan l_{jk}, *k=1,2,3*. Pusat massa T_j terdapat pada koordinat (x_j, y_j) , dan titik tengah (*midpoint*) tiap-tiap sisi k pada sebuah T_j, dilambangkan dengan $M_{jk} = (x_{jk}, y_{jk})$, *k=1,2,3*. Sementara itu, T_{j1}, T_{j2}, T_{j3} adalah segitiga-segitiga tetangga, masing-masing saling berbagi sebuah sisi dengan T_j.

3.8. Metode Finite Volume

Metode *finite volume,* didasarkan pada pengintegralan bentuk hukum konservasi (LeVeque, 2002). Metode ini, merupakan salah satu metode numerik yang digunakan untuk menyelesaikan persamaan diferensial parsial. Dibandingkan dengan metode klasik *finite different*, *finite volume* memiliki beberapa kelebihan, salah satunya *mesh* dapat dibentuk dengan pola yang tidak terstruktur sehingga diskritisasi spasial menjadi lebih fleksibel. Fleksibilitas ini, artinya kualitas sel dapat ditingkatkan (*refined*), serta sisi-sisi yang menjadi batasan domain pada *mesh*

dapat dirancang bebas sesuai dengan keinginan, (Couason, et al., 2011). Selain itu, skema numerik pada metode *finite volume* mampu mengatasi diskontinuitas seperti gelombang kejut.

3.8.1. Diskontinuitas

Pada kenyataannya, dalam penyelesaian persamaan differensial parsial dapat dihasilkan solusi yang tidak halus (*non-smooth*), mengandung diskontinuitas seperti gelombang kejut (*shock waves*). Khususnya pada bentuk yang nonlinear, diskontinuitas dapat terjadi secara spontan, meskipun data inisialisasi halus (*smooth*). Diskontinuitas membuat komputasi menjadi lebih sulit. Hal inilah yang menjadi kekurangan metode *finite different,* yaitu pada kondisi diskontinu gagal menghasilkan solusi sesuai dengan yang diharapkan (LeVeque, 2002).

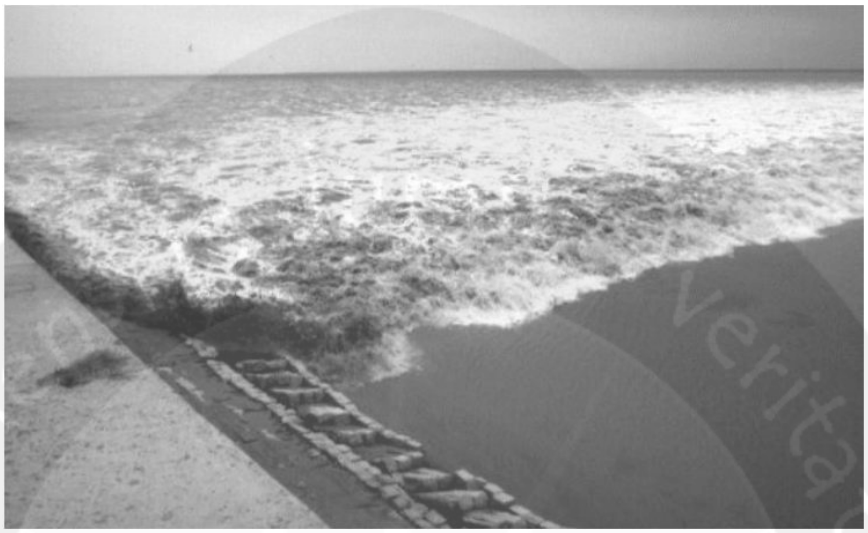
Metode *finite difference* didasarkan bentuk diferensial dari suatu persamaan. Ketika terdapat diskontinuitas, penyelesaian dengan metode ini sulit, karena pada persamaan diferensial, diasumsikan bahwa solusi yang akan dihasilkan adalah solusi yang halus (*smooth*). Sebaliknya, metode *finite volume*, didasarkan pada bentuk integral dari suatu persamaan. Tidak ada asumsi bahwa solusi harus halus (*smooth*), sehingga metode *finite volume* dapat digunakan dalam penyelesaian solusi yang halus (*smooth*) maupun tidak halus (*non-smooth*) (Hidayat, et al., 2014).

3.8.2. Gelombang Kejut

Nonlinearitas dapat menciptakan diskontinuitas, atau dekat dengan diskontinuitas. Ada dua klasifikasi utama fenomena yaitu lompatan hidrolik dan *shock fronts.* Dalam cairan yang tidak dapat dimampatkan (*incompressible liquid*) dengan pengaruh gravitasi, lompatan hidrolik ditandai dengan kenaikan ketinggian secara tiba-tiba di permukaan terbuka pada aliran yang cepat (Lautrup, 2011).

Lompatan hidrolik adalah salah satu bentuk gelombang kejut. Lompatan ini dapat bersifat stasioner ataupun dinamis. Lompatan stasioner contohnya tempat cuci piring. Kolom air yang keluar dari keran, akan jatuh dan melebar membentuk pola aliran melingkar, dan pada radius tertentu aliran yang tipis tiba-tiba menebal (Gambar 3.8). Sementara lompatan dinamis, artinya lompatan tersebut bergerak

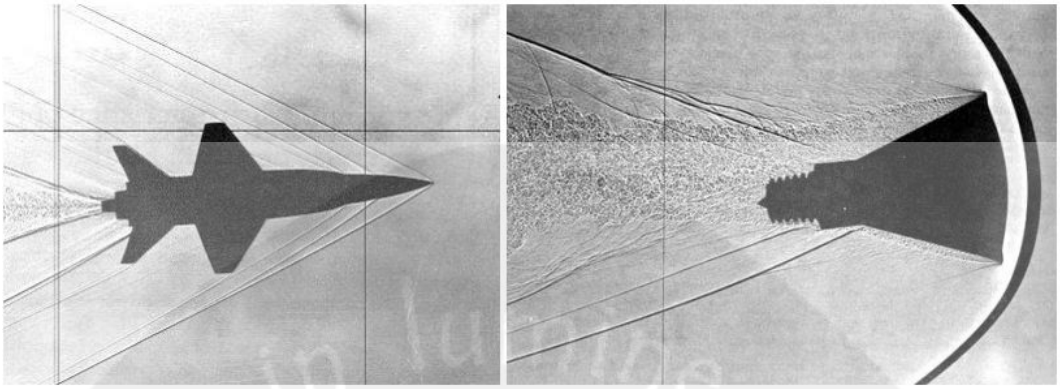
mengikuti arah perambatan gelombang, contohnya adalah gelombang tidal Sungai Qiantang (Gambar 3.7).



Gambar 3.7. Gelombang tidal Sungai Qiantang.
Sumber: Lautrup, 2011.

Gambar 3.8. Lompatan hidrolik pada tempat cuci piring (kiri), dan lompatan hidrolik dengan bentuk sirkular sempurna (kanan).
Sumber: Lautrup, 2011.

Lompatan secara tiba-tiba dalam suatu fluida pada supersonik (*supersonic front)* disebut dengan *shock* (Gambar 3.9). Pemahaman mengenai *shock* sangat penting untuk mendesain pesawat supersonic, jet, dan mesin roket.

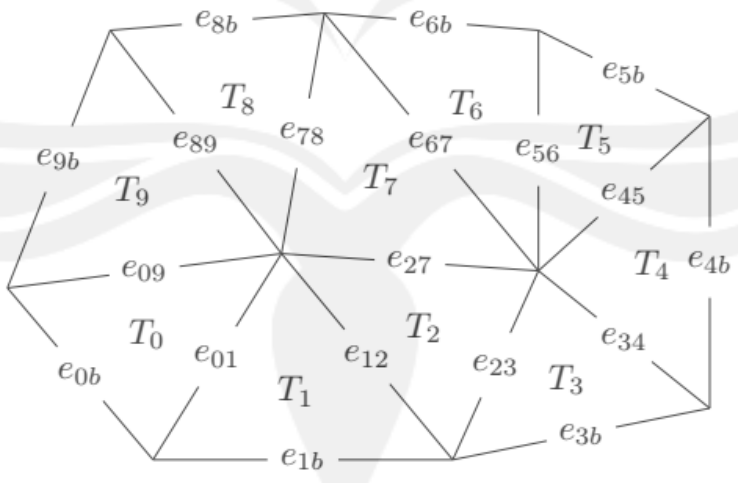


Gambar 3.9. Gelombang kejut dari pesawat supersonic (kiri). dan kendaraan luar angkasa Mercury (kanan).

Sumber: Lautrup, 2011.

3.8.3. Skema Numerik pada Mesh Segitiga Tidak Terstruktur

Skema *finite volume* dapat diterapkan pada *mesh* berbentuk segitiga tidak terstruktur. Gambar 3.10 berikut menunjukkan bentuk *mesh* dengan elemen-elemen sel berbentuk segitiga, dengan ukuran tiap segitiga dapat bervariasi (Roberts, et al., 2015).



Gambar 3.10. *Mesh* segitiga yang digunakan dalam metode *finite volume*.

Sumber: Roberts, 2015.

Skema *finite volume* didapatkan dengan cara mengintegralkan bentuk diferensial hukum konservasi pada persamaan (3.6), di tiap-tiap sel segitiga, yaitu

$$\frac{\partial U}{\partial t} + \vec{\nabla}\vec{F} = Q_v + \vec{\nabla}\vec{Q}_s$$

$$\int_\Omega \frac{\partial U}{\partial t}\, d\Omega + \int_\Omega \vec{\nabla}\vec{F}\, d\Omega = \int_\Omega Q_v\, d\Omega + \int_\Omega \vec{\nabla}\vec{Q}_s\, d\Omega$$

$$\int_{T_i} \frac{\partial \mathbf{U}}{\partial t}\, d\mathbf{x} + \int_{T_i} \left(\frac{\partial \mathbf{E}}{\partial x} + \frac{\partial \mathbf{G}}{\partial y}\right) d\mathbf{x} = \int_{T_i} \mathbf{S}\, d\mathbf{x}$$

dengan fluks arah *x* dan *y* dinyatakan dengan **E** dan **G**, domain volume Ω merupakan sel segitiga T_i, dan komponen *source* **S** melingkupi Q_v dan $\vec{\nabla}\vec{Q}_s$,

$$\vec{F} := (\mathbf{E}, \mathbf{G}), \qquad \mathbf{S} = Q_v + \vec{\nabla}\vec{Q}_s, \qquad \vec{\nabla} := \left(\frac{\partial}{\partial x}, \frac{\partial}{\partial y}\right),$$

$$\Omega = T_i, \qquad d\Omega = d\mathbf{x}$$

Dengan menerapkan teorema divergen,

$$\oint_C (\mathbf{E}, \mathbf{G}) \cdot \mathbf{n}\, ds = \oint_C \vec{F} \cdot d\vec{r} = \int_\Omega \vec{\nabla}\vec{F}\, d\Omega$$

C merupakan kurva yang mengelilingi segitiga (*counterclockwise*) dan merupakan gabungan dari tiga buah kurva sisi segitiga, $d\vec{r} := (dx, dy)$ adalah vektor yang menunjukkan tangensial sepanjang kurva C, dan $\sqrt{dx^2 + dy^2} = ds$, maka didapatkan sebuah persamaan yang berlaku pada tiap sel T_i, yang menyatakan perubahan rata-rata kuantitas konservasi tiap sel, dengan catatan fluks mengalir melewati sisi-sisi sel, dan terdapat sumber (*source term*) pada domain,

$$\int_{T_i} \frac{\partial \mathbf{U}}{\partial t}\, d\mathbf{x} + \oint_C (\mathbf{E}, \mathbf{G}) \cdot \mathbf{n}\, ds = \int_{T_i} \mathbf{S}\, d\mathbf{x}$$

$$\frac{d}{dt}\frac{1}{A_i}\int_{T_i} \mathbf{U}\, d\mathbf{x} + \frac{1}{A_i}\sum_{j \in N_i} \int_{e_{ij}} (\mathbf{E}, \mathbf{G}) \cdot \mathbf{n}\; ds = \frac{1}{A_i}\int_{T_i} \mathbf{S}\, d\mathbf{x}$$

dan **n** merupakan vektor normal sisi segitiga yang mengarah keluar. Jadi, bentuk semi diskret terkait nilai rata-rata pada tiap sel yaitu,

$$\frac{d\mathbf{U}_i}{dt} + \frac{1}{A_i}\sum_{j\in N_i} \mathbf{H}_{ij} l_{ij} = \mathbf{S}_i \tag{3.11}$$

dengan

- **U, E, G,** masing-masing merupakan vektor kuantitas, fluks arah *x*, dan fluks arah *y*.
- *i* merupakan indeks, menunjuk pada sel segitiga T_i ke – *i*,
- *j* merupakan indeks, menunjuk pada sel tetangga ke – *j*, dari segitiga T_i,
- N_i merupakan jumlah sisi pada sel segitiga ke – *i*,
- A_i merupakan area sel segitiga ke – *i*,
- e_{ij} merupakan sisi di antara sel ke – *i* dan ke – *j*,
- l_{ij} adalah panjang skalar sisi e_{ij},
- $\mathbf{U}_i$ adalah vektor rata-rata kuantitas konservasi pada sel ke – *i*, yaitu $\frac{1}{A_i}\int_{T_i} \mathbf{U}\, d\mathbf{x}$,
- $\mathbf{S}_i$ adalah rata-rata sumber (*source term)* terkait dengan sel ke – *i*, yaitu $\frac{1}{A_i}\int_{T_i} \mathbf{S}\, d\mathbf{x}$, dan
- $\mathbf{H}_{ij} l_{ij}$ adalah taksiran fluks normal arah keluar dari material sepanjang sisi ke – *ij*, yaitu $\int_{e_{ij}} (\mathbf{E}, \mathbf{G}) \cdot \mathbf{n}\ ds$.

Integrasi waktu dapat dilakukan dengan metode Euler eksplisit, yaitu

$$\frac{\mathbf{U}_i^{n+1} - \mathbf{U}_i^n}{\Delta t} + \frac{1}{A_i}\sum_{j\in N_i} \mathbf{H}_{ij} l_{ij} = \mathbf{S}_i$$

sehingga persamaan (3.11) berbentuk seperti berikut.

$$\mathbf{U}_i^{n+1} = \mathbf{U}_i^n - \frac{\Delta t}{A_i}\sum_{j\in N_i} \mathbf{H}_{ij} l_{ij} + \Delta t \mathbf{S}_i \tag{3.12}$$

3.8.4. Perhitungan Fluks dengan Central Upwind

Diskontinuitas menyulitkan proses komputasi. Permasalahan utamanya adalah menentukan fungsi perhitungan fluks yang baik, yang mampu menaksir nilai dengan akurat (LeVeque, 2002). Salah satu skema yang dapat digunakan untuk perhitungan fluks yaitu *central-upwind.* Skema ini memiliki beberapa keuntungan yaitu kesederhanan, universal, dan ketahanannya, serta dapat diterapkan pada kasus dengan geometri yang kompleks (Kurganov & Petrova, 2005).

Pada persamaan (3.11), fluks dalam arah vektor normal **n** := (n_x, n_y), dinyatakan dengan

$$\mathbf{H}(\mathbf{U}) = \mathbf{E}(\mathbf{U})n_x + \mathbf{G}(\mathbf{U})n_y$$

Skema numerik didapatkan dengan menaksir fluks **H** menggunakan *fungsi perhitungan fluks.* Dengan menerapkan skema *central-upwind* (Roberts, et al., 2015), fungsi fluks tersebut menjadi,

$$\mathbf{H}_{ij} = \frac{a_{ij}^{+}\mathbf{H}(\mathbf{U}_{ij}^{i}) - a_{ij}^{-}\mathbf{H}(\mathbf{U}_{ij}^{j})}{a_{ij}^{+} - a_{ij}^{-}} + \frac{a_{ij}^{+}a_{ij}^{-}}{a_{ij}^{+} - a_{ij}^{-}}\left[\mathbf{U}_{ij}^{j} - \mathbf{U}_{ij}^{i}\right] \tag{3.13}$$

dengan a_{ij}^{+} dan a_{ij}^{-} merupakan kecepatan lokal gelombang berarah. Nilai kecepatan tersebut didefinisikan dengan

$$\begin{aligned} a_{ij}^{+} &= \max\left\{\lambda_3\left[V_{ij}(\mathbf{U}_{ij}^{i})\right], \lambda_3\left[V_{ij}(\mathbf{U}_{ij}^{j})\right], 0\right\}, \\ a_{ij}^{-} &= \min\left\{\lambda_1\left[V_{ij}(\mathbf{U}_{ij}^{i})\right], \lambda_1\left[V_{ij}(\mathbf{U}_{ij}^{j})\right], 0\right\}, \end{aligned} \tag{3.14}$$

$\lambda_1[V_{ij}] \leq \lambda_2[V_{ij}] \leq \lambda_3[V_{ij}]$ adalah nilai-nilai eigen dari matrix (Bryson, et al., 2011) berikut.

$$V_{ij} = n_x\frac{\partial \mathbf{E}}{\partial \mathbf{U}} + n_y\frac{\partial \mathbf{G}}{\partial \mathbf{U}}$$

3.8.5. Finite Volume Central-Upwind pada Mesh Segitiga Tidak Terstruktur untuk Persamaan Air Dangkal

Persamaan air dangkal (3.7), (3.8), (3.9) didiskritisasi pada mesh segitiga tidak terstruktur dengan skema numerik metode *finite-volume* (3.12) dan fungsi perhitungan fluks menggunakan *central-upwind* (3.13). Adapun kecepatan lokal gelombang a_{ij}^{+} dan a_{ij}^{-} memiliki nilai sebagai berikut (Roberts, et al., 2015),

$$\begin{aligned} a_{ij}^{+} &= \max\left\{\mathbf{u}_{ij}^{i} + \sqrt{gh_{ij}^{i}}, \mathbf{u}_{ij}^{j} + \sqrt{gh_{ij}^{j}}, 0\right\}, \\ a_{ij}^{-} &= \min\left\{\mathbf{u}_{ij}^{i} - \sqrt{gh_{ij}^{i}}, \mathbf{u}_{ij}^{j} - \sqrt{gh_{ij}^{j}}, 0\right\}, \end{aligned} \tag{3.15}$$

g adalah konstanta gravitasi, h_{ij}^{i} adalah ketinggian permukaan air pada sel segitiga T_i, dan h_{ij}^{j} adalah ketinggian permukaan air pada sel tetangga ke – j, dari segitiga T_i. Sementara $\mathbf{u}_{ij}^{i}$ dan $\mathbf{u}_{ij}^{j}$ adalah kecepatan arah normal (Bryson, et al., 2011),

$$\begin{aligned} \mathbf{u}_{ij}^{i} &= n_x u_{ij}^{i} + n_y v_{ij}^{i} \\ \mathbf{u}_{ij}^{j} &= n_x u_{ij}^{j} + n_y v_{ij}^{j}, \end{aligned} \tag{3.16}$$

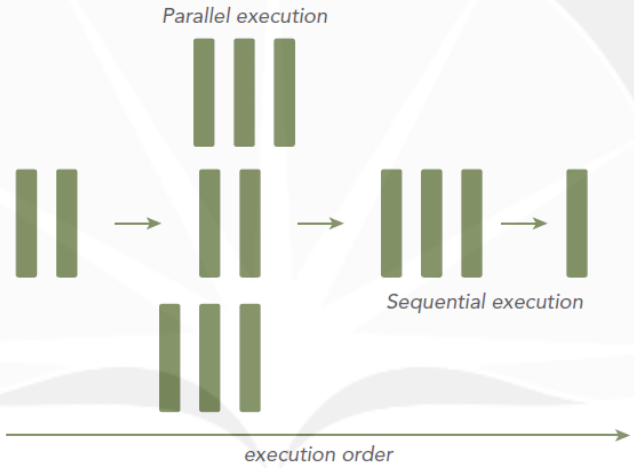
dengan u_{ij}^{i} dan v_{ij}^{i} adalah kecepatan arah *x* dan arah *y* dari aliran pada sel T_i, dengan sementara u_{ij}^{j} dan v_{ij}^{j} adalah kecepatan arah *x* dan arah *y* dari aliran pada tetangga ke – j sel T_i.

3.9. Komputasi Paralel GPU CUDA

Proses simulasi melibatkan komputasi yang memerlukan waktu. Komputasi yang dilakukan secara sekuensial, artinya program harus menunggu satu instuksi selesai sebelum mengeksekusi instruksi selanjutnya, menyebabkan waktu komputasi berlangsung lama. Jika data dan iterasi cukup banyak, maka waktu komputasi yang diperlukan pun juga besar. Tetapi, para peneliti menginginkan agar

hasil simulasi dapat diperoleh dengan cepat, sehingga proses komputasi harus dilakukan dengan pendekatan yang lain, yaitu secara paralel.

Gambar 3.11 Ilustrasi eksekusi instruksi pada program sekuensial.
Sumber: Cheng dkk., 2014.

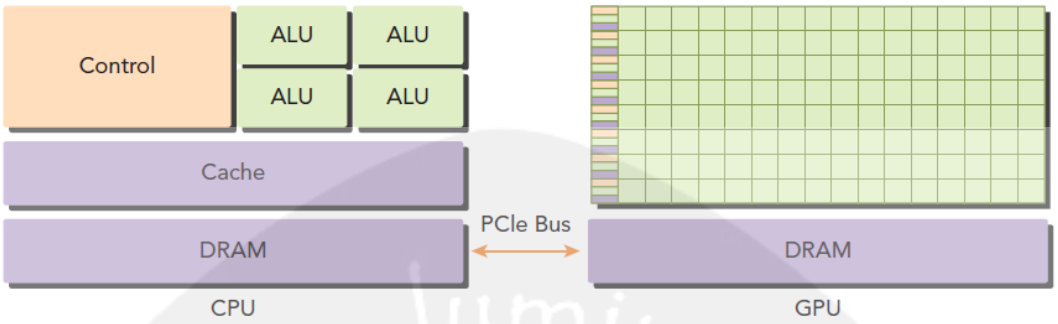


Gambar 3.12 Ilustrasi eksekusi instruksi pada program paralel.
Sumber: Cheng dkk., 2014.

Perbedaan program sekuensial dan paralel diilustrasikan pada Gambar 3.11 dan Gambar 3.12. Pada program sekuensial, instruksi dieksekusi satu demi satu berdasarkan urutannya. Sementara pada pendekatan paralel, beberapa instruksi dan/atau data didistribusikan secara paralel, sehingga instruksi-instruksi tersebut dapat dieksekusi secara bersamaan/sekaligus dalam sekali waktu. Dalam penerapannya, terdapat dua jenis pendekatan secara paralel, yaitu *task parallelism* dan *data parallelism*. Pada *task parallelism*, yang didistribusikan secara paralel adalah *task* atau fungsinya. Sementara pada *data parallelism* yang didistribusikan secara paralel adalah *item* data. Data ini dapat dioperasikan pada satu waktu yang sama (Cheng, et al., 2014).

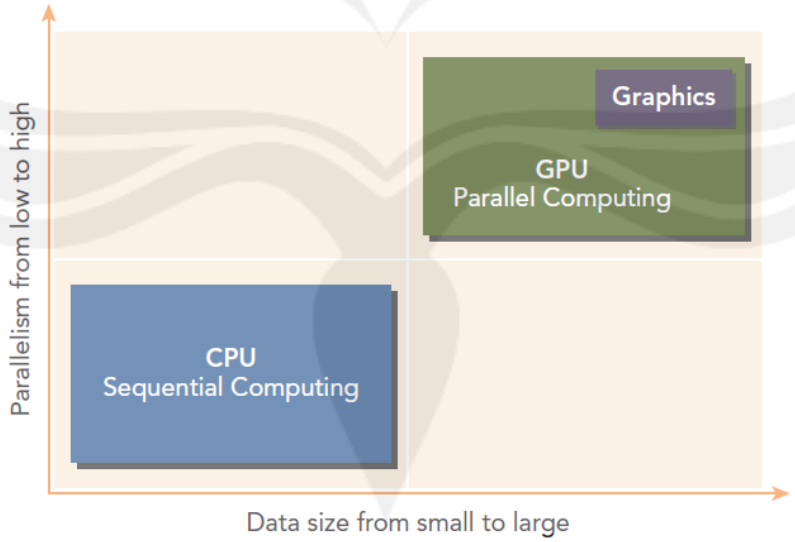
Peralatan komputasi saat ini yang dapat digunakan untuk melakukan simulasi adalah komputer. Komputer memiliki unit inti yang disebut CPU (Central Processing Unit), yang pada dasarnya terdiri beberapa kontrol dan memori untuk memproses intruksi secara serial. Pada perkembangannya, performa CPU ditingkatkan dengan cara menambah jumlah inti mikroposesor, meningkatkan kecepatan *clock*, menambah jumlah operasi yang dapat dilakukan tiap detik. Namun, sejak tahun 2000-an perkembangan CPU mendekati *stagnan*. Arsitektur CPU yang terdiri dari banyak kontrol tidak mampu dikembangkan lagi untuk mencapai akselerasi yang lebih baik. Maka, sebagai alternatif, digunakanlah GPU (Graphic Processing Unit), yang memiliki ribuan *core*. GPU yang paling sering digunakan untuk komputasi saat ini adalah GPU CUDA keluaran NVIDIA. GPU ini akan memproses intruksi secara paralel, dengan pendekatan *data parallelism* (Cheng, et al., 2014).

GPU tidak dapat berdiri sendiri. Untuk menggunakan GPU, tetap dibutuhkan CPU. Dengan kata lain, untuk melakukan komputasi secara paralel, digunakan arsitektur yang heterogen, dengan CPU sebagai *host*, dan GPU sebagai *device*. Sebagai *host*, CPU bertanggung jawab untuk menginisialisasi, mengatur lingkungan, kode, dan data sebelum dimuat ke *device*. Sebagai *device*, GPU bertanggung jawab untuk mendistribusikan data ke dalam memori GPU, dan melakukan komputasi dengan mengeksekusi kernel dan menciptakan serta menjalankan *thread-thread* secara paralel. Dilihat dari aliran data dan instruksi pada inti-inti prosesor, arsitektur ini disebut sebagai SIMD (Single Instruction Multiple Data), artinya satu instruksi, banyak data. Masing-masing *core* dapat mengeksekusi instruksi yang sama pada satu waktu, dengan *data stream* yang berbeda (Cheng, et al., 2014). Gambar 3.13 menunjukkan arsitektur heterogen antara CPU dan GPU, keduanya memiliki memori, dan dihubungkan dengan PCIe Bus.



Gambar 3.13. Host (CPU) dan device (GPU) saling berinteraksi.
Sumber: Cheng dkk., 2014.

Kemunculan GPU tidak bermaksud menggantikan komputasi dengan CPU. Masing-masing pendekatan memiliki keunggulan tersendiri. Komputasi dengan CPU baik untuk instruksi yang menggunakan banyak kontrol. Sementara komputasi dengan GPU baik untuk intruksi yang mengakses data secara paralel. Jika sebuah kasus memiliki data yang sedikit, membutuhkan *control logic*, sedikit bagian yang dapat diparalelkan, maka CPU adalah pilihan yang baik. Jika sebuah kasus memilki banyak data dan banyak bagian yang dapat diparalelkan, maka menggunakan GPU adalah pilihan yang tepat. Hal ini dapat dilustrasikan melalui Gambar 3.14 berikut.



Gambar 3.14. Ukuran data dan *paralelism* menentukan arsitektur pemrograman.
Sumber: Cheng dkk., 2014.

3.10. Model Pemrograman GPU CUDA

Sistem heterogen terdiri atas CPU (*host*) dan GPU (*device*), masing-masing memiliki memori sendiri-sendiri. Model pemrograman CUDA memungkinkan aplikasi dijalankan pada sistem yang heterogen tersebut. Komponen yang penting dalam model pemrograman ini adalah *kernel*, yaitu kode yang berjalan pada *device* GPU. Ketika kernel diluncurkan, kontrol dikembalikan ke *host* (CPU), dan sementara secara asinkron kode kernel tersebut dijalankan dalam GPU.

Proses utama dalam sebuah program CUDA adalah sebagai berikut.

1. Salin data dari memori CPU ke memori GPU.
2. Pemanggilan kernel untuk mengoperasikan data yang disimpan dalam memori GPU.
3. Salin data kembali dari memori GPU ke CPU.

3.10.1. Manajemen Memori

Cuda menyediakan fungsi untuk mengalokasikan memori *device*, membebaskan memori *device*, serta mentransfer data antara memori *host* dan memori *device.* Untuk mengalokasikan memori GPU digunakan `cudaMalloc`,

```
cudaError_t cudaMalloc (void** devPtr, size_t size)
```

Fungsi tersebut mengalokasikan memori *device* dengan spesifikasi ukuran dalam *bytes.* Hasil alokasi memori dikembalikan melalui pointer `devPtr`.

Fungsi untuk mentransfer data antara *host* dan *device* yaitu `cudaMemcpy`,

```
cudaError_t cudaMemcpy (void* dst, const void* src,
                        size_t count, cudaMemcpyKind kind)
```

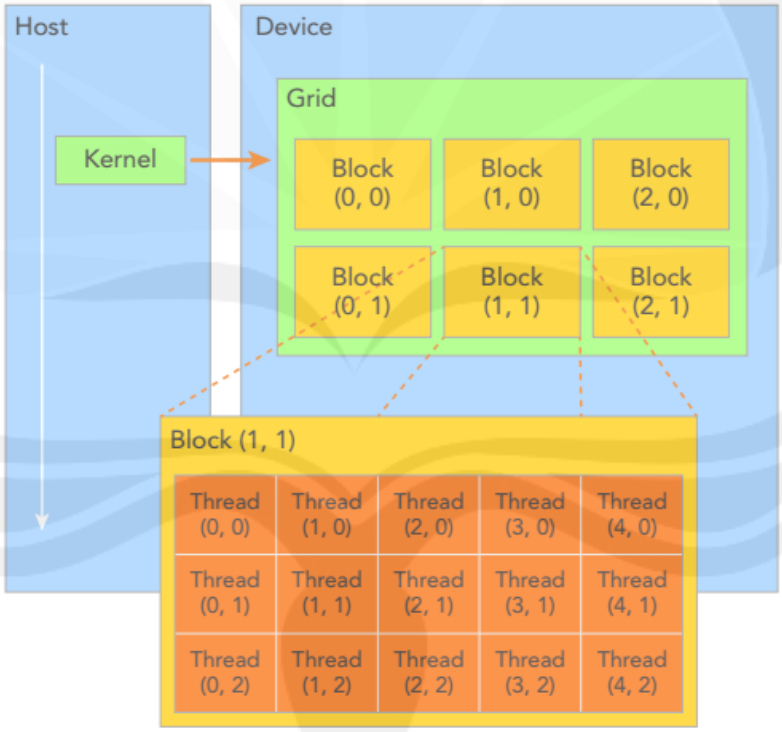
Fungsi ini akan menyalin spesifik *bytes* dari area memori sumber yang ditunjuk oleh pointer *src*, ke area memori tujuan yang ditunjuk oleh pointer *dst*, dengan spesifik arah ditentukan oleh `kind`, yang bernilai salah satu dari empat tipe berikut,

- `cudaMemcpyHostToHost`, yaitu menyalin data dari *host* ke *host*,
- `cudaMemcpyHostToDevice`, yaitu menyalin data dari *host* ke *device*,
- `cudaMemcpyDeviceToHost`, yaitu menyalin data dari *device* ke *host*,
- `cudaMemcpyDeviceToDevice`, yaitu menyalin data dari *device* ke *device.*

Sementara itu `cudaFree` digunakan untuk membebaskan memori pada GPU, dan sintaks `cudaError_t` adalah tipe yang menyimpan kode *error*. Adapun `CudaSetDevice`, digunakan untuk menginisialisasi *device* GPU, dan `CudaResetDevice`, digunakan untuk mer-*reset* device GPU.

3.10.2. Pengorganisasian Thread Ekseskusi Kernel CUDA

Ketika sebuah *kernel* di jalankan pada sisi *host*, eksekusi berpindah ke *device*. Sejumlah besar *thread* diciptakan, dan tiap *thread* mengekseskusi baris-baris kode yang telah dispesifikasi oleh fungsi kernel. CUDA mengabstraksikan pengorganisasian *thread-thread* tersebut ke dalam struktur hieararki *block of thread*, dan *grid of block* seperti gambar berikut (Cheng, et al., 2014).



Gambar 3.15. Hirarki pengorganisasian *thread-thread* pada GPU CUDA.
Sumber: Cheng dkk., 2014.

Seluruh *thread* dihasilkan oleh sebuah kernel yang diluncurkan, yang secara kolektif disebut *grid*. Sebuah *grid* terdiri dari beberapa blok, dan sebuah blok terdiri dari beberpa *thread*. CUDA mengelola *grid-grid* dan blok-blok dalam tiga dimensi.

Gambar 3.15 menunjukkan struktur hirarki sebuah *grid* berisi 2 dimensi blok - blok, dan sebuah blok berisi 2 dimensi *thread* - *thread*.

Thread dari blok yang berbeda tidak dapat bekerja sama. *Thread-thread* memiliki dua koordinat unik, yang membedakan *thread* tersebut dari *thread-thread* yang lain, yaitu

blockIdx, yaitu indeks suatu blok dalam sebuah *grid*,

threadIdx, yaitu indeks suatu *thread* dalam sebuah blok.

Koordinat tersebut bertipe `uint3`, masing-masing komponen menunjukkan posisi dalam arah *x*, *y*, dan *z*.

3.11. Arsitektur dan Model Eksekusi GPU-NVIDIA

Arsitektur GPU tersusun atas banyak Streaming Multiprocessor (SM). Tiap-tiap SM di GPU didesain agar mendukung eksekusi ratusan *thread* secara serentak, sehingga dalam satu GPU, dimungkinkan ada ribuan *thread* yang dieksekusi secara bersamaan. Sebuah blok *thread* dijadwalkan hanya pada sebuah SM. Semenjak sebuah blok *thread* dijadwalkan pada sebuah SM, blok *thread* tersebut akan tetap di sana hingga ekseskusi selesai. Sebuah SM dapat menangani lebih dari sebuah blok *thread* dalam satu waktu yang sama (Cheng, et al., 2014).

Ketika menjalankan sebuah kernel, terlihat bahwa *thread-thread* dalam kernel berjalan secara paralel. Secara logis hal ini dapat diterima, namun dari sisi *hardware,* tidak semua *thread* dapat secara fisik dieksekusi paralel dalam satu waktu yang sama. Tiap 32 *thread* dari *thread-thread* tersebut sebenarnya dikelompokkan ke dalam satuan unit eksekusi, disebut dengan *warp.* Ketika sebuah blok *thread* dijadwalkan ke sebuah SM, *thread-thread* pada blok *thread* akan dipartisi ke dalam *warp-warp.*

Merubah ukuran blok *thread* adalah salah satu cara meningkatkan utilisasi. Namun terdapat beberapa batasan dalam memanipulasi blok *thread* yaitu:

1. Blok *thread* kecil: Jumlah *thread* per blok yang terlalu kecil akan membatasi *hardware,* yaitu jumlah *warp* per SM yang dicapai.
2. Blok *thread* besar: Terlalu banyak *thread* per blok mengakibatkan terlalu sedikitnya sumber daya per-SM yang tersedia bagi tiap *thread*

Berikut merupakan berberapa langkah untuk menentukan ukuran blok dan *grid.*

1. Tetap jaga jumlah *thread* per blok kelipatan dari ukuran *warp* (32).
2. Hindari ukuran blok yang kecil, mulai dengan paling tidak 128 atau 256 *thread* per blok.
3. Naik/turunkan ukuran blok sehubungan dengan kebutuhan.
4. Tetap jaga ukuran blok lebih besar dari pada jumlah SM.
5. Lakukan eksperimen untuk menemukan konfigurasi eksekusi terbaik.

3.12. Visualisasi Grafis dengan OpenGL

Untuk melakukan visualisasi grafis, digunakan OpenGl. OpenGL adalah sebuah API (*application programming interface*) yang sebenarnya adalah pustaka *software* untuk mengakses grafis *hardware*. OpenGL dirancang sebagai pustaka bebas *hardware*, dapat diterapkan pada berbagai jenis *hardware* (Shreiner, Sellers, John, & Licea-Kane, 2013).

Untuk menggambar, OpenGL memiliki primitif seperti *points*, *line*, dan *triangle*. *Line* dan *triangle* dapat dikombinasikan untuk membentuk *strips*, *loops*, dan *fans*. *Points*, *lines*, dan *triangles* adalah primitif asli yang didukung oleh kebanyakan *hardware*. Pustaka OpenGL juga menyediakan beragam fungsi lainnya seperti pewarnaan, tekstur, cahaya, pembentukan bayangan, dan lain sebagainya. (Shreiner, Sellers, John, & Licea-Kane, 2013).

3.13. Vertex Buffer Object

Dua metode yang digunakan untuk menyimpan bentuk grafis dan geometri dalam *cache hardware* grafis adalah *display list* dan *vertex buffer object* (VBO). *Display list*, menyusun sebuah daftar bentuk dan representasi geometri, serta menyimpan daftar tersebut dalam memory grafis. Keuntungan menggunakan *display list*, yaitu mereka telah umum, dan baik dalam menyimpan bentuk dan representasi geometri yang statis. Namun, metode ini tidak dapat bekerja dengan baik untuk bentuk geometri yang dinamis/selalu berubah.

Metode lain, yaitu VBO, merupakan *buffer* yang menjadi ciri memori berperforma tinggi pada grafis hardware, yang juga menjimpan data *array* verteks. Metode ini mampu memetakan ke dalam memori grafis untuk akses dan *update* yang cepat. Dengan VBO, geometri dapat dimodifikasi secara dinamis, dengan kemungkinan kehilangan performa yang kecil, dibandingkan menggunakan *display list* (Shirley & Marschner, 2009).