BAB II

LANDASAN TEORI

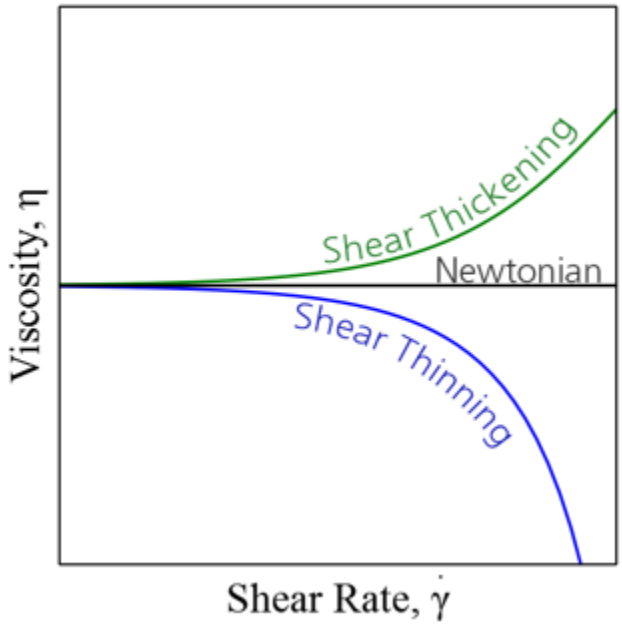
Telaah literatur yang berhubungan dalam penelitian ini disebutkan dan dijelaskan di bawah ini.

2.1 Newtonian dan Non-Newtonian Fluids

Fluid yang sering ditemukan, seperti air dan minyak, merupakan sebuah *Newtonian Fluid*. Satu-satunya faktor yang mempengaruhi tingkat *viscosity* mereka adalah suhu, sehingga tingkat *viscosity* tetap konstan. Namun beberapa tipe *fluid* memiliki *viscosity* yang dapat dipengaruhi oleh faktor lain selain suhu seperti contohnya kecap yang tingkat *viscosity*-nya berkurang jika diguncang. *Fluid* ini tidak mengikuti hukum *viscosity* Newton sehingga disebut dengan *Non-Newtonian fluid*, perubahan tingkat *viscosity Non-Newtonian fluid* dipengaruhi oleh tingkat pergolakan atau tekanan, secara teknis diketahui dengan istilah *shear stress*. *Newtonian fluid* tidak akan dipengaruhi oleh *shear stress* (Rohrig, 2017).

Gambar 2.1 memperlihatkan bahwa tingkat geser *fluid* akan selalu konstan pada *Newtonian fluid*, dan dapat berubah pada *Non-Newtonian fluid* jika tingkat gesernya bertambah. *Non-Newtonian fluid* juga terbagi menjadi dua kelompok, yaitu *Shear Thickening* dan *Shear Thinning*. *Shear Thinning* adalah fenomena dimana jika *Non-Newtonian fluid* menerima benda dengan gaya geser atau *shear force* yang tinggi, maka tingkat *viscosity* dari *fluid* tersebut akan berkurang, sehingga air tersebut menjadi lebih cair, sebaliknya, jika *Non-Newtonian fluid* menerima benda dengan gaya geser yang rendah, maka tingkat *viscosity* atau

tingkat kekentalan *fluid* akan bertambah, contohnya kecap. Sebaliknya untuk *Shear Thickening*, jika *fluid* menerima benda dengan gaya geser yang tinggi maka tingkat *viscosity* dari *fluid* tersebut akan meningkat sehingga *fluid* tersebut menjadi lebih kental, contohnya adalah Oobleck.



Gambar 2.1 Perbandingan Nilai *Viscosity* pada *Newtonian* dan *Non-Newtonian Fluid* (Akhremitchev *et al*., 2015)

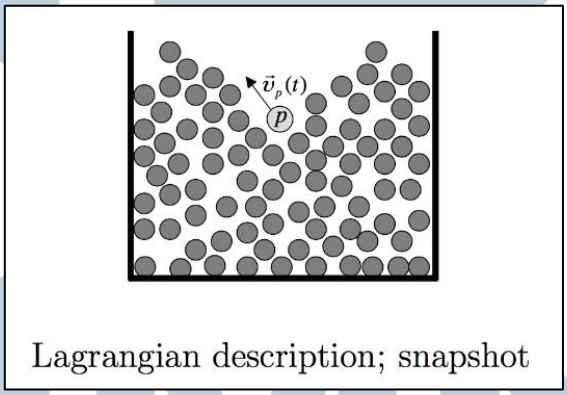
2.2 Computational Fluid Dynamics

Fluid Dynamics, menurut kamus bahasa inggris *The American Heritage Dictionary of the English Language* (American Heritage Publishing Company, 1969) adalah cabang dari ilmu pengetahuan terapan yang berhubungan dengan pergerakan air dan gas. *Fluid Dynamics* adalah salah satu dari dua cabang *Fluid Mechanics*, yang mempelajari *fluid* dan bagaimana tenaga mempengaruhinya. Cabang lainnya adalah *Fluid Statics* yang berhubungan dengan *fluid* pada keadaan statis atau terdiam (Lucas, 2019).

Untuk mencapai simulasi Computational Fluid Dynamics terdapat dua metode yang paling mendasar yaitu metode Eulerian dan Lagrangian.

2.2.1 Metode Lagrangian

Metode Lagrangian menggambarkan *fluid* sebagai partikel-partikel yang dimana setiap partikel tersebut memiliki informasi posisi awal, dan posisi yang baru. Karena Lagrangian bergerak berdasarkan partikel-partikel, maka metode ini dapat disebut juga dengan *particle-based fluid dynamics* Partikel akan bergerak berdasarkan informasi tersebut, hal ini menyebabkan metode Lagrangian memiliki contoh spasial yang kurang terkontrol, karena partikel akan pergi kemana pun arus pergi. Contoh algoritma berdasarkan metode Lagrangian adalah *Smoothed Particle Hydrodynamics* (Price, 2006).

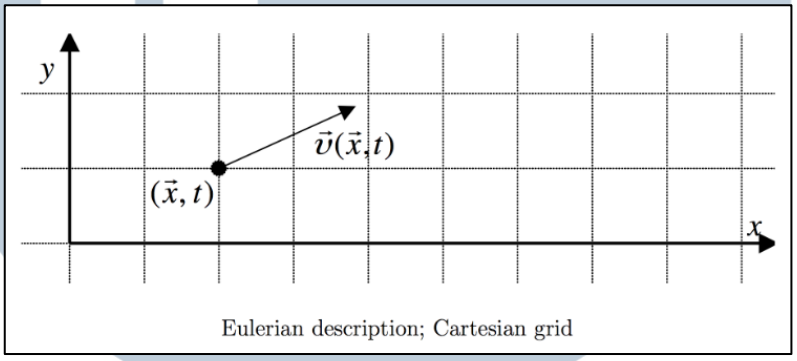


Gambar 2.2 Visualisasi Metode Lagrangian (Newman 2018, Ertekin 2015)

Karena metode Lagrangian merupakan metode yang menyimpan informasi pada setiap partikelnya seperti yang dijelaskan pada Gambar 2.2, maka properti *velocity*, kepadatan massa, tekanan, dan lain lain, untuk partikel ke j dapat direpresentasikan secara matematis seperti: $\vec{\mathbf{v}}_j(t), \rho_j(t), \mathrm{p}_j(t)$. Konservasi massa, dan hukum Newton pada metode Lagrangian berlaku pada setiap partikelnya (Newman 2018, Ertekin 2015).

2.2.2 Metode Eulerian

Metode Eulerian, atau dapat disebut juga dengan metode *grid-based fluid dynamics* mengukur arus dengan cara membangun struktur *grid* yang bersifat tetap, lalu setiap *cell* dari *grid* nya memiliki informasi. Metode ini cocok digunakan jika tujuannya adalah mengobservasi rata-rata waktu arus bergerak pada sebuah *channel* atau *cell*. Contoh algoritma berdasarkan metode Eulerian adalah *Staggered Marker and Cell Grid* (Price, 2006).



Gambar 2.3 Visualisasi Metode Eulerian (Newman 2018, Ertekin 2015)

Metode Eulerian menyimpan informasi partikel pada *grid* seperti yang dijelaskan pada Gambar 2.3, artinya pergerakan partikel memiliki ketergantungan terhadap lokasi. Properti *velocity*, kepadatan massa, tekanan, dan lain lain, dapat direpresentasikan secara matematis sebagai: $\vec{\mathrm{v}}(\vec{\mathrm{x}},t), \vec{\rho}(\vec{\mathrm{x}},t), \vec{\mathrm{p}}(\vec{\mathrm{x}},t)$ (Newman 2018, Ertekin 2015). Lokasi yang disebutkan diatas dideskripsikan pada sistem koordinat, Gambar 2.3 menggambarkan posisi partikel pada *cartesian grid*.

2.3 Smoothed Particle Hydrodynamics

Smoothed Particle Hydrodynamics (SPH) adalah metode komputasi yang digunakan untuk simulasi mekanik media kontinuum, seperti mekanik padat dan aliran *fluid*. Di kembangkan oleh Gingold dan Monaghan (Gingold & Monaghan,

1977), dan Lucy (Lucy, 1977) pada tahun 1977, yang pada awalnya dibuat untuk mengatasi masalah astrofisika, balistik, vulkanologi, dan oseanografi. Merupakan metode Lagrangian yang berarti menggunakan sistem partikel untuk simulasi *fluid* dan informasi pergerakan *fluid* disimpan pada setiap partikel. *Smoothed Particle Hydrodynamics* lalu mendistribusi kuantitas lokal ke partikel tetangga dari setiap partikel menggunakan radial *symmetrical smoothing kernels* (Muller *et al*., 2003).

2.3.1 Integral Interpolant

SPH merupakan metode *interpolation* yang memungkinkan fungsi apapun dinyatakan perihal nilainya pada suatu kumpulan titik-titik yang acak, atau partikel (Monaghan, 1992).

Rumus 2.1 (Monaghan, 1992) Rumus Integral Interpolant.

$$A_{I(\mathbf{r})} = \int_{\Omega} A(\mathbf{r}')W(\mathbf{r}-\mathbf{r}',h)\mathbf{dr}' \quad \ldots(2.1)$$

Seperti yang telah dijelaskan sebelumnya, SPH adalah sebuah metode *interpolation* yang menggunakan *kernel*. *Kernel* ini memodelkan fungsi delta, sesuai dengan posisi partikel. Integral interpolant dari fungsi A(x) didefinisikan pada Rumus 2.1. Dimana Ω adalah ruang terjadinya kalkulasi *interpolation*, ***r*** adalah posisi partikel, *W* adalah fungsi *smoothing kernel* dengan *h* sebagai radiusnya (Strantzi, 2016).

Rumus 2.2 (Monaghan, 1992) Smoothing Kernel.

$$A_S(r) = \sum_j m_j \frac{A_j}{\rho_j} W(r-r_j,h) \quad \ldots(2.2)$$

Rumus 2.3 (Monaghan, 1992) Smoothing Kernel Gradient.

$$\nabla A_S(r) = \sum_j m_j \frac{A_j}{\rho_j} \nabla W(r-r_j,h) \quad \ldots(2.3)$$

Rumus 2.4 (Monaghan, 1992) Smoothing Kernel Laplacian.

$$\nabla^2 A_S(r) = \sum_j m_j \frac{A_j}{\rho_j} \nabla^2 W(r - r_j, h) \quad \ldots(2.4)$$

Ketiga rumus merupakan Kernel yang akan digunakan. Pada Rumus 2.2 simbol *A* merupakan atribut partikel, dan jumlah *j* melambangkan indeks partikel dari seluruh partikel yang ada pada simulasi. Partikel *j* memiliki massa m_j, posisi $\mathbf{r}_j$, kepadatan massa ρ_j, dan *velocity* $\mathbf{v}_j$. Dari Rumus 2.2, fungsi *interpolant* dapat diturunkan dari nilainya pada partikel atau *interpolation points,* menggunakan kernel yang dapat diturunkan (Monaghan, 1992). Ketika menerapkan operator Gradient dan Laplacian, maka diperoleh rumus yang dijabarkan pada Rumus 2.3 dan 2.4. Oleh karena itu, operator Gradient dan Laplacian hanya diterapkan pada fungsi kernel karena sesuai dengan aturan diferensiasi, $m_j \frac{A_j}{\rho_j}$ sama dengan 0 ketika diturunkan (Strantzi, 2016).

2.3.2 Smoothing Kernel

Smoothing Kernel adalah fungsi pembobotan yang menyesuaikan kuantitas partikel sesuai dengan jarak masing-masing. Terdapat beberapa Smoothing Kernel yang berbeda yang dapat digunakan untuk penaksiran SPH, tergantung pada kuantitas yang butuh dikalkulasikan (Strantzi, 2016). Muller *et al*. (2003) juga menyatakan bahwa pilihan Smoothing Kernel sangat mempengaruhi stabilitas, akurasi, dan kecepatan. Smoothing Kernel yang digunakan pada penelitian ini adalah 6th Polynomial, Spiky Gradient, dan Viscosity Laplacian dimana perbedaan didefinisikan secara berurutan pada Rumus 2.5, 2.6, dan 2.7. Karena pemanggilan fungsi $W(r, h)$ dilakukan dengan cara $W(r - r_j, h)$ maka *r* pada Smoothing Kernel disini merupakan jarak antara posisi *r* yang dicari dengan tetangga r_j.

Rumus 2.5 (Muller *et al*., 2003) 6th Polynomial atau Poly6 Kernel.

$$W_{poly6}(r,h) = \frac{315}{64\pi h^9}\begin{cases}(h^2 - r^2)^3, & 0 \le r \le h \\ 0, & otherwise\end{cases} \quad \ldots(2.5)$$

Rumus 2.6 (Muller *et al*., 2003) Spiky Gradient.

$$W_{spiky}(r,h) = \frac{15}{\pi h^6}\begin{cases}(h - r)^3, & 0 \le r \le h \\ 0, & otherwise\end{cases} \quad \ldots(2.6)$$

Rumus 2.7 (Muller *et al*., 2003) Viscosity Laplacian.

$$W_{viscosity}(r,h) = \frac{15}{2\pi h^3}\begin{cases}-\frac{r^3}{2h^3} + \frac{r^2}{h^3} + \frac{h}{2r} - 1, & 0 \le r \le h \\ 0, & otherwise\end{cases} \quad \ldots(2.7)$$

Kernel 6th Polynomial, yang dijabarkan pada Rumus 2.5 memiliki fitur penting yaitu *r* yang hanya dipangkatkan yang artinya dapat dievaluasikan tanpa menghitung akar pada perhitungan jarak. Namun, jika Kernel ini digunakan untuk komputasi gaya tekanan, partikel cenderung membuat kelompok pada tekanan yang tinggi. Saat posisi partikel mendekat dengan satu sama lain, gaya tolakan hilang karena Gradient dari Kernel mendekati nol pada titik tengah.

Desbrun (1996) mengatasi masalah ini menggunakan Spiky Gradient Kernel dengan Gradient yang tidak menghilang pada titik tengahnya. Untuk komputasi tekanan, penelitian ini menggunakan Spiky Gradient Kernel milik Desbrun yang dijabarkan pada Rumus 2.6. Kernel ini menghasilkan gaya tolakan yang dibutuhkan.

Viscosity adalah fenomena yang disebabkan oleh gaya geser dan mengurangi energi kinetik fluida dengan mengubahnya menjadi panas. Oleh karena itu, viscosity seharusnya hanya memiliki efek *smoothing* pada velocity. Namun, jika Kernel standar digunakan untuk velocity, hasil gaya viscosity tidak selalu memiliki properti ini. Untuk dua partikel yang mendekat satu sama lainnya, Laplacian dari velocity yang sudah *smoothed* dapat memiliki nilai negatif sehingga

menghasilkan gaya yang meningkatkan velocity relatifnya. Oleh karena itu, untuk komputasi viscosity digunakan Kernel ketiga yaitu Kernel Viscosity Laplacian yang dijabarkan pada Rumus 2.7 (Muller *et al*., 2003).

2.3.3 Algoritma SPH

Algoritma yang digunakan memiliki dasar dari penelitian Muller *et al*. (2003) yang dijelaskan pada Gambar 2.4.

```
for i in particles
   for j in particles
      distance = j.position - i.position
      r2 = squareMagnitudeOf(distance)
      if r2 < SQUARED_SMOOTHING_RADIUS
         i.density += MASS * POLY6_KERNEL
   i.pressure = GAS_CONSTANT * (i.density - REST_DENSITY)

// Hitung gaya
for i in particles
   for j in particles
      if i == j
         continue
      distance = j.position - i.position
      r = magnitudeOf(distance)
      n = normalize(distance)

      forcePressure, forceViscosity, forceGravity = 0

      if r2 < SQUARED_SMOOTHING_RADIUS
         forcePressure += -n * MASS * (i.pressure + j.pressure) / (2/j.density) * SPIKY_GRADIENT_KERNEL
         forceViscosity += VISCOSITY * MASS * (j.velocity - i.velocity) / j.density * VISCOSITY_LAPLACIAN_KERNEL
      forceGravity = GRAVITY * i.density * GRAVITY_MULTIPLIER
      i.force = forcePressure + forceViscosity + forceGravity

// integrasi
for i in particles
   i.velocity += DELTA_TIME * i.force / i.density
   i.position += DELTA_TIME * i.velocity
```

Gambar 2.4 Algoritma Dasar SPH

Algoritma yang dijelaskan pada Gambar 2.4 adalah algoritma dasar SPH yang belum diterapkan optimisasi apapun sehingga pada fungsi komputasi kepadatan massa dan tekanan, serta gaya masih menggunakan *nested-for-loop* yang memiliki kompleksitas $O(n^2)$. fungsi tersebut berguna untuk melakukan komputasi partikel berdasarkan partikel sebelahnya, sehingga memungkinkan optimisasi yang menggunakan algoritma pencarian tetangga atau Neighbor Search Algorithm.

2.4 Nearest Neighbor Search Algorithm

Bagian penting pada simulasi Computational Fluid Dynamics menggunakan metode Lagrangian adalah Nearest Neighbor Search. Semua perhitungan partikel dilakukan dengan mempertimbangkan atribut partikel yang memiliki lokasi dekat dengan partikel yang sedang dikalkulasikan. Tanpa menggunakan algoritma Nearest Neighbor Search atau melakukan pencarian dengan Linear Search, jika jumlah partikel sudah mencapai ribuan atau lebih, pencarian partikel tetangga dari seluruh partikel (*n*) atau pencarian partikel secara linear akan sangat berat.

```
for i in particles
    i.density = 0
    for j in particles
        distance = j.position - i.position
        squaredMagnitude = rij.sqrMagnitude

        if squaredMagnitude < SQUARED_SMOOTHING_RADIUS
            i.density += MASS * POLY6_KERNEL * Mathf.Pow(SQUARED_SMOOTHING_RADIUS - squaredMagnitude, 3)
```

Gambar 2.5 Perhitungan Kepadatan Massa Tanpa Menggunakan Algoritma Nearest Neighbor

Gambar 2.5 menjelaskan perhitungan kepadatan massa yang didefinisikan sebagai variabel *density*. Perhitungan ini memperoleh jumlah massa dikali kernel dari setiap partikel *j* yang berada didalam *smoothing radius* dari partikel *i*. Namun, pada pencarian linear ini, pencarian partikel *j* akan dilakukan pada seluruh partikel (*n*) dan diulang setiap iterasi partikel *i*. Kompleksitas yang didapat adalah $O(n^2)$, sehingga akan menggunakan sumber daya komputasi yang sangat tinggi. Oleh karena itu metode optimisasi dibutuhkan. Pada penelitian ini, digunakan metode Spatial Hashing (Strantzi, 2016).

```
// Fowler Noll Vo
int Hash(Vector3 grid){
   int hash = grid.x;
   hash = (hash * FNV_Prime) ^ grid.y;
   hash = (hash * FNV_Prime) ^ grid.z;
   return hash;
}
```

Gambar 2.6 Fungsi Hash Fowler Noll Vo (FNV) (Fowler *et al*., 1991)

Spatial Hashing adalah algoritma Nearest Neighbor Search yang memiliki kompleksitas komputasional O(1). Cara algoritma ini bekerja adalah setiap posisi partikel di *hash* dengan menggunakan fungsi *hash* Fowler Noll Vo (FNV) seperti yang dijelaskan pada Gambar 2.6 untuk mendapatkan sebuah *key*, dimana *key* tersebut berperan sebagai indeks pada sebuah *hashtable*. Gambar 2.4 merupakan implementasi fungsi *hash* menggunakan bahasa pemrograman C# dengan menerima parameter dengan tipe data vektor tiga dimensi. Fungsi ini akan mengalikan sumbu x dari vektor yang diterima dengan bilangan prima, lalu melakukan operasi XOR pada sumbu y. Kemudian, hasil dari perkalian dan XOR pada sumbu x dan y akan diberlakukan perkalian pada bilangan prima dan terakhir dilakukan operasi XOR pada nilai sumbu z. Fungsi *hash* tersebut akan mengembalikan nilai yang sudah di-*hash* lalu digunakan untuk memanggil partikel yang terdapat di *cell* yang sama sesuai dengan indeks *hash*-nya (Strantzi, 2016).

```
for i in particles
    i.density = 0.0f
    neighbors = hashtable[hash(i.position)]
    for j in neighbors
        distance = j.position - i.position
        squaredMagnitude = rij.sqrMagnitude
        if squaredMagnitude < SQUARED_SMOOTHING_RADIUS
            i.density += MASS * POLY6_KERNEL * Mathf.Pow(SQUARED_SMOOTHING_RADIUS - squaredMagnitude, 3)
```

Gambar 2.7 Perhitungan Kepadatan Massa Menggunakan Spatial Hashing

Gambar 2.7 menjelaskan penerapan algoritma Spatial Hashing pada pencarian kepadatan massa. Perbedaan dengan pencarian secara linear adalah sebelum

mengiterasikan partikel *j* dari seluruh partikel (*n*), algoritma ini akan mencari Nearest Neighbor atau tetangga terdekat yang sudah disimpan pada *hashtable*. Sehingga, partikel *j* hanya akan beriterasi sebanyak *m* kali. Oleh karena itu, kompleksitas yang didapat dari algoritma ini adalah O(n*m) dimana *m* lebih kecil dari *n*.

2.5 Entity Component System (ECS)

Entity Component System (ECS) adalah sebuah paradigma pemrograman yang sesuai dengan namanya terdiri dari tiga bagian yaitu Entity, Component, dan System (atau subsystem).

2.5.1 Entity

Entity adalah fondasi dasar konseptual pada ECS, setiap Entity merepresentasikan objek berbeda-beda di dalam game yang bersifat konkrit. Untuk setiap "benda" yang dapat dilihat pada dunia game yang dibuat, terdapat satu Entity. Dalam satuan jumlah instansi pada saat *runtime*, Entity dapat dicontohkan sebagai Object dari OOP, contohnya jika terdapat 100 buah mobil yang sama, maka terdapat 100 buah Entity, bukan 1.

Namun, dalam hal tingkah laku, Entity bekerja seperti Class pada OOP, secara tidak langsung, Entity mendefinisikan tingkah laku semua objek di dalam game. Kesimpulannya, Entity tidak berguna jika berdiri sendiri, mereka hanya berfungsi sedikit lebih banyak daripada sebuah label atau *tag* setiap objek di dalam game sebagai benda yang terpisah. Pada bagian ini lah dimana Component berfungsi.

2.5.2 Component

Setiap objek di dalam game memiliki aspek yang menjelaskan apa dan bagaimana objek tersebut berinteraksi dengan dunia game. Contohnya, sebuah mobil adalah:

1) Terbuat dari logam
2) Dapat digunakan oleh manusia
3) Sebuah alat transportasi
4) Benda yang dapat diperjualbelikan

Pada skenario seperti ini terdapat sebuah Entity bernama mobil yang memiliki empat Component seperti yang disebutkan diatas. Kesimpulannya, Component memiliki fungsi dasar yaitu memberikan Entity sebuah label yang menandakan Entity tersebut dapat melakukan hal yang sesuai dengan Component tersebut.

2.5.3 System (atau Subsystem)

Pada paradigma OOP, semua "benda" akan menjadi sebuah Object. Namun pada ECS, terdapat dua hal berbeda yaitu "Entity dan Component", serta "Systems". Dimana "Entity dan Component" merupakan objek yang memiliki "label", dan "Systems" adalah fungsi yang menggerakan "Entity dan Component" yang memiliki "label" yang sesuai.

OOP memiliki performa yang sangat baik untuk mengimplementasikan bagian program apapun yang memiliki banyak data dan metode yang dibutuhkan untuk dijalankan pada bagian atau subset data yang kecil pada suatu program. Namun OOP sangat buruk jika digunakan untuk mengimplementasikan objek yang sama dalam jumlah yang sangat besar. ECS memecahkan masalah ini dengan secara

eksplisit dengan cara menjalankan hal-hal yang bersifat global menggunakan Systems, yang terdapat di luar dunia Entity atau Component. Setiap System berjalan secara kontinu dan melakukan aksi global terhadap seluruh Entity, namun metode atau fungsi sebuah System hanya akan berjalan ketika Entity yang bersangkutan memiliki Component yang sesuai.

Contoh System yang umum pada sebuah game adalah: Rendering System, Input System, Animation System, dll.

1) Pada sistem ini, Rendering System akan berjalan setiap 16 *milliseconds*, lalu mencari seluruh Entity yang memiliki Component yang dapat direnderer.
2) Animation System secara konstan mencari apapun yang dapat men-*trigger* sebuah animasi baru, atau menyebabkan animasi yang sedang berjalan berubah atau berhenti, dan memperbaharui data dari setiap Entity yang memiliki Component yang dapat dianimasikan.
3) Kemudian Input System akan menunggu input dari *keyboard*, *mouse*, *joystick*, dll. dan merubah state dari Entity apapun yang memiliki Component yang dapat menerima input.

Pada OOP tradisional, jika sebuah game terdapat 100 unit, dan setiap unit direpresentasikan menggunakan Object, maka secara teori terdapat 100 salinan dari setiap metode yang dapat dipanggil pada setiap unit. Namun pada ECS, jika dalam game yang sama terdapat 100 unit, yang setiap unit nya direpresentasikan oleh sebuah Entity, maka tidak akan ada 100 salinan metode yang dapat dipanggil setiap unit, karena Entity dan Component tidak menampung metode. Sebagai gantinya terdapat sistem secara eksternal untuk setiap aspek, dan sistem tersebut memiliki

semua metode yang dibutuhkan Entity yang memiliki Component yang menandakan Entity tersebut dapat menjalankan metode tersebut melalui System (Martin, 2007).

2.6 Data Oriented Tech Stack (DOTS)

Data Oriented Tech Stack (DOTS) adalah *tech stack* yang dibuat oleh *game engine* Unity yang bertujuan untuk meningkatkan performa dengan memanfaatkan Unity C# Job System, Entity Component System (ECS), dan Burst Compiler. DOTS ini baru dirilis ke publik sebagai versi Preview pada Unity versi 2019.1.

2.6.1 Unity C# Job System

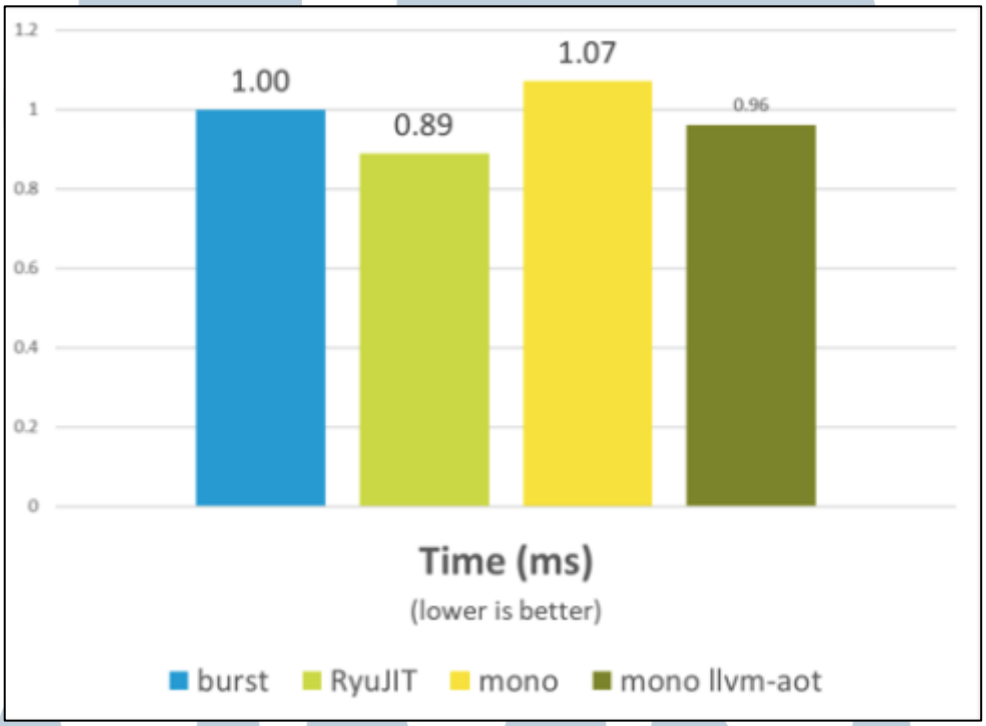
Unity C# Job System adalah sebuah fitur yang disediakan oleh *game engine* Unity yang berfungsi untuk memudahkan penulisan *multithreaded code*. Penulisan *multithreaded code* dapat memberikan performa yang tinggi termasuk peningkatan *frame rate*. Aspek yang penting dari C# Job System adalah integrasi dengan Unity *native job system*, sebuah fitur *multithreading* yang terdapat pada inti *game engine* Unity namun tidak dapat digunakan oleh pengguna. *Source code* yang ditulis oleh pengguna menggunakan C# Job System berbagi *worker thread*. Kerjasama ini mencegah pembuatan thread yang lebih banyak daripada CPU core yang dimiliki perangkat keras pengguna. (Unity, 2018)

2.6.2 Burst Compiler

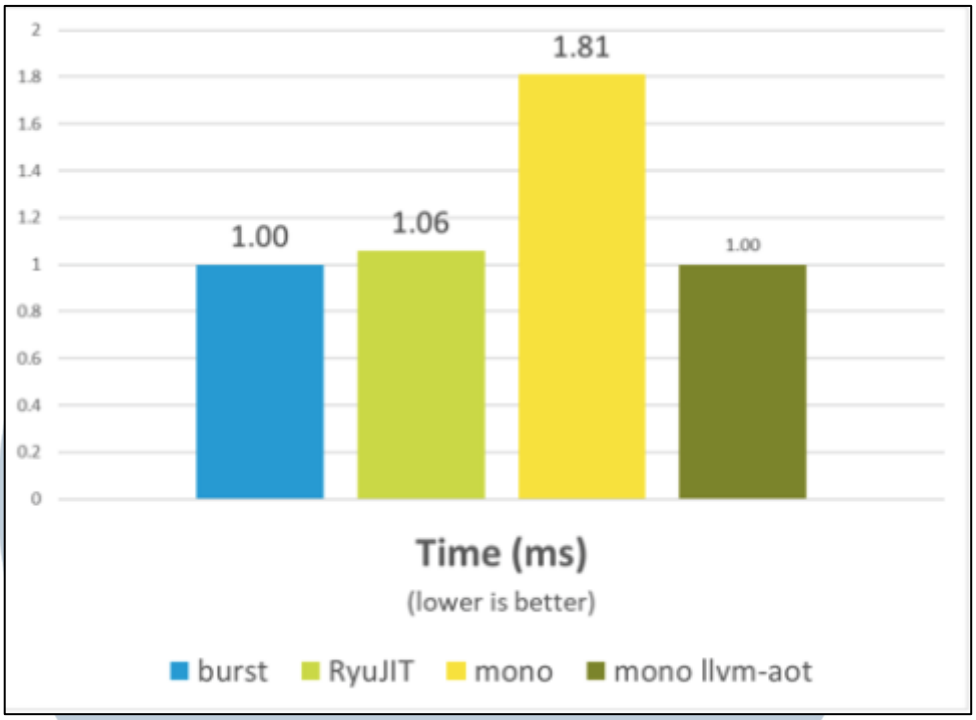
Burst Compiler adalah sebuah compiler yang didukung oleh Unity versi 2018.3 keatas, berbasis LLVM, yang men-compile C# Jobs menjadi *machine code* yang sudah dioptimisasikan. Bagian-bagian tersebut menurut Alexandra (Mutel, 2019) adalah sebagai berikut.

A. LLVM Codegen dibandingkan dengan Mono/.NET JIT

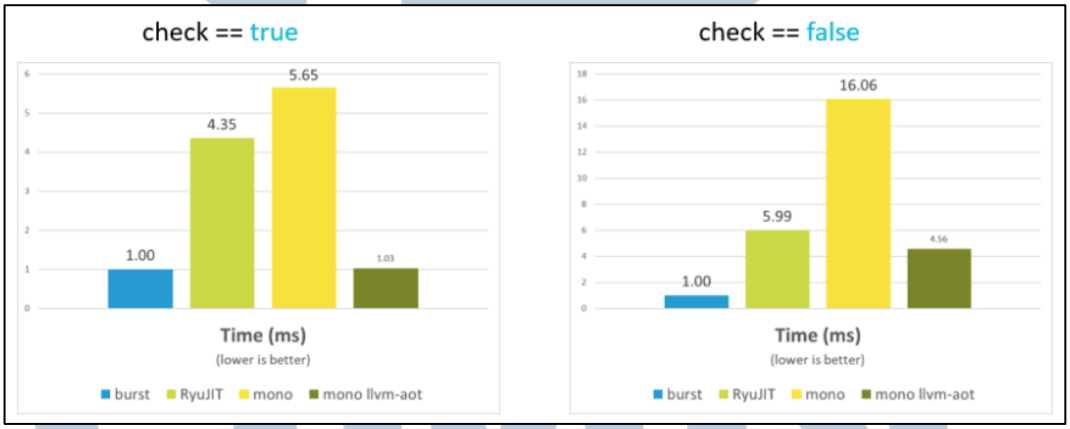
Berdasarkan penelitian Alexandra (Mutel, 2019) perbandingan kecepatan eksekusi dari beberapa algoritma yaitu algoritma Binary Search pada Gambar 2.8, algoritma Mandelbrot pada Gambar 2.9, algoritma Loop Unswitch pada Gambar 2.10, dan algoritma Count Bits pada Gambar 2.11.



Gambar 2.8 Hasil Perbandingan Compiler pada Algoritma Binary Search (Mutel, 2019)

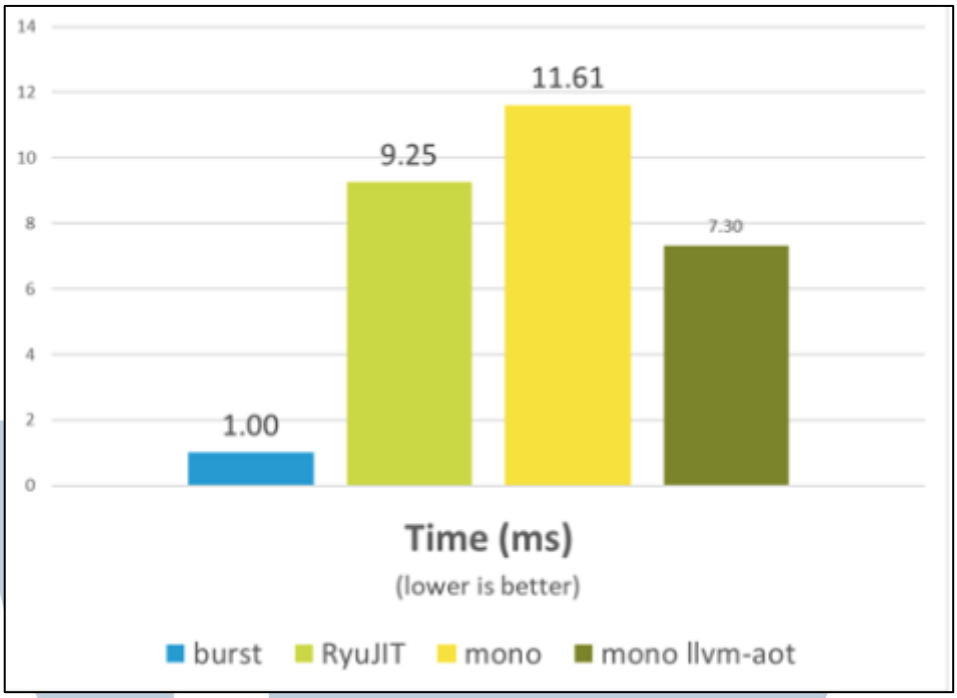


Gambar 2.9 Hasil Perbandingan Compiler pada Algoritma Mandelbrot (Mutel, 2019)



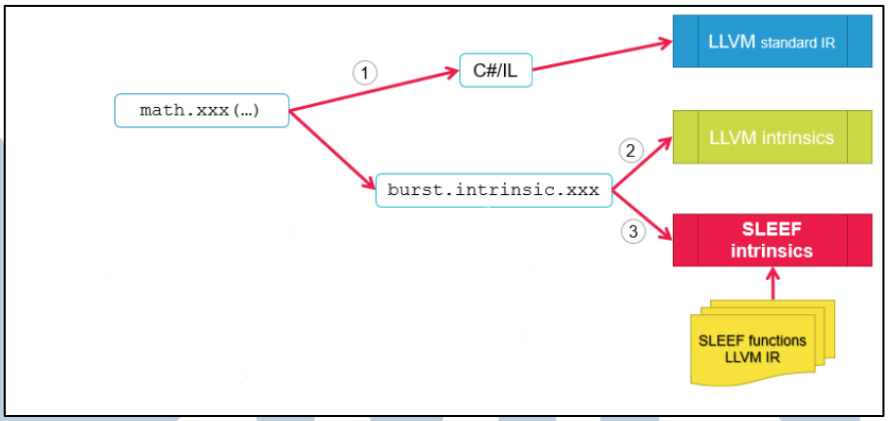
Gambar 2.10 Hasil Perbandingan Compiler pada Algoritma Loop Unswitch (Mutel, 2019)

Gambar 2.8, 2.9, 2.10, dan 2.11 (lebih kecil lebih baik) menggambarkan bahwa Burst Compiler yang digunakan pada beberapa algoritma terbukti lebih cepat dibandingkan dengan *compiler* yang tersedia. Ini membuktikan bahwa Burst Compiler dapat meningkatkan performa secara signifikan.



Gambar 2.11 Hasil Perbandingan Compiler pada Algoritma Count Bits (Mutel, 2019)

B. Optimisasi dari Unity.Mathematics pada Burst Compiler



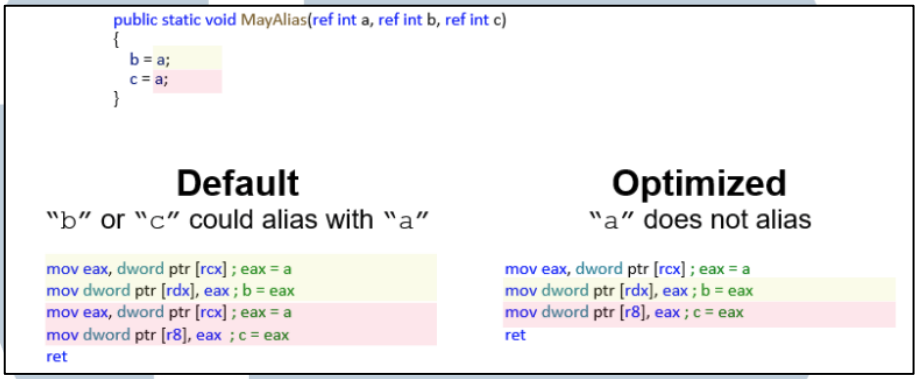
Gambar 2.12 Ilustrasi Optimisasi Fungsi pada Unity.Mathematics (Mutel, 2019)

Fungsi matematika dari Unity.Mathematics dioptimisasikan dengan tiga cara yang berbeda seperti pada Gambar 2.12.

1) Isi dari fungsi matematika tidak memiliki perubahan.
2) Isi dari fungsi matematika diterjemahkan kedalam LLVM intrinsic

3) Isi dari fungsi matematika diterjemahkan kedalam fungsi yang disediakan oleh SLEEF Library, sebuah library SIMD matematika yang bersifat *open source*.

C. Non-aliasing memory access dan Data Oriented Design (DOD)



Gambar 2.13 Perbandingan *Aliasing Memory Access* dengan *Non-Aliasing Memory Access* (Mutel, 2019)

Memory Aliasing terjadi jika lokasi memori dapat diakses melalui variabel yang berbeda-beda pada metode atau program yang sama. Secara standar, sebuah *compiler* akan mengasumsi *Aliasing*. Namun, jika variabel tersebut dapat diindikasikan tidak akan melakukan alias maka compiler dapat mengoptimisasi *code* seperti yang digambarkan pada Gambar 2.13, untuk menghindari *duplicate load, reorder instructions, vectorized loop,* dll. Prinsip fundamental ini sangat bermanfaat untuk DOD (Mutel, 2019).

D. Native Code dan Garbage Collector (GC)

Managed Code dan GC bekerja secara sinkron dimana setiap *thread* harus melakukan sinkronisasi secara periodik sehingga *thread* yang sedang berjalan harus mengalami perhentian sejenak. Pada *Burst Compiler*, *Native code* yang berjalan pada sebuah thread tidak akan diberhentikan oleh GC.

2.7 GPU Instancing

Untuk menggambar sebuah objek pada layar, Unity harus memanggil sebuah *draw call* terhadap *graphics* API seperti OpenGL atau Direct3D. *Draw calls* seringnya memakan sumber daya komputasi yang tinggi dimana *graphics* API melakukan komputasi yang signifikan untuk setiap *draw call*. Hal ini seringnya disebabkan oleh perubahan *state* yang dilakukan diantara setiap *draw call* seperti mengganti ke material yang berbeda. Untuk mengatasi masalah ini, Unity menyediakan fitur GPU Instancing (Unity, 2017).

GPU Instancing adalah teknik penggambaran beberapa salinan dari suatu *mesh* yang sama sekaligus, menggunakan jumlah *draw call* yang kecil. Teknik ini berguna untuk menggambar objek seperti bangunan, pohon-pohonan, rumput, dan objek lainnya yang terlihat diulang berkali-kali. GPU Instancing hanya menggabungkan *draw call* dari *mesh* yang memiliki material yang sama. Penggabungan *draw call* ini, atau disebut juga dengan *batching* secara signifikan meningkatkan performa *rendering* (Unity, 2017).