PENGEMBANGAN *2D GAME ENGINE* MENGGUNAKAN ARSITEKTUR *ENTITY COMPONENT SYSTEM*

SKRIPSI

Oleh :
CHANDRA GUNAWAN
NIM. 18650071

PROGRAM STUDI TEKNIK INFORMATIKA
FAKULTAS SAINS DAN TEKNOLOGI
UNIVERSITAS ISLAM NEGERI MAULANA MALIK IBRAHIM
MALANG
2025

PENGEMBANGAN *2D GAME ENGINE* MENGGUNAKAN ARSITEKTUR *ENTITY COMPONENT SYSTEM*

SKRIPSI

Diajukan kepada:
Universitas Islam Negeri Maulana Malik Ibrahim Malang
Untuk memenuhi Salah Satu Persyaratan dalam
Memperoleh Gelar Sarjana Komputer (S.Kom)

Oleh :
CHANDRA GUNAWAN
NIM. 18650071

PROGRAM STUDI TEKNIK INFORMATIKA
FAKULTAS SAINS DAN TEKNOLOGI
UNIVERSITAS ISLAM NEGERI MAULANA MALIK IBRAHIM
MALANG
2025

HALAMAN PERSETUJUAN

PENGEMBANGAN 2D *GAME ENGINE* MENGGUNAKAN ARSITEKTUR *ENTITY COMPONENT SYSTEM*

SKRIPSI

Oleh :
CHANDRA GUNAWAN
NIM. 18650071

Telah Diperiksa dan Disetujui untuk Diuji:
Tanggal: 2 Desember 2024

Pembimbing I,	Pembimbing II,
Dr. Fachrul Kurniawan, M.MT, IPU	Dr. Muhammad Faisal, M.T
NIP. 19771020 200912 1 001	NIP. 19740510 200501 1 007

Mengetahui,
Ketua Program Studi Teknik Informatika
Fakultas Sains dan Teknologi
Universitas Islam Negeri Maulana Malik Ibrahim Malang

Dr. Fachrul Kurniawan, M.MT, IPU
NIP. 19771020 200912 1 001

iii

HALAMAN PENGESAHAN

PENGEMBANGAN *2D GAME ENGINE* MENGGUNAKAN ARSITEKTUR *ENTITY COMPONENT SYSTEM*

SKRIPSI

Oleh :
CHANDRA GUNAWAN
NIM. 18650071

Telah Dipertahankan di Depan Dewan Penguji Skripsi
dan Dinyatakan Diterima Sebagai Salah Satu Persyaratan
Untuk Memperoleh Gelar Sarjana Komputer (S.Kom)
Tanggal: 20 Maret 2025

Susunan Dewan Penguji

Ketua Penguji	: Dr. Fresy Nugroho, M.T NIP. 19710722 201101 1 001	( )
Anggota Penguji I	: Nur Fitriyah Ayu Tunjung Sari, M.Cs NIP. 19911226 202012 2 001	( )
Anggota Penguji II	: Dr. Ir. Fachrul Kurniawan ST, M.MT, IPU NIP. 19771020 200912 1 001	( )
Anggota Penguji III	: Dr. Muhammad Faisal, M.T NIP. 19740510 200501 1 007	( )

Mengetahui dan Mengesahkan,
Ketua Program Studi Teknik Informatika
Fakultas Sains dan Teknologi
Universitas Islam Negeri Maulana Malik Ibrahim Malang

Dr. Ir. Fachrul Kurniawan ST, M.MT, IPU
NIP. 19771020 200912 1 001

iv

PERNYATAAN KEASLIAN TULISAN

Saya yang bertanda tangan di bawah ini:

Nama	: **Chandra Gunawan**
NIM	: 18650071
Fakultas / Program Studi	: Sains dan Teknologi / Teknik Informatika
Judul Skripsi	: Pengembangan 2D *Game Engine* Menggunakan Arsitektur *Entity Component System*

Menyatakan dengan sebenarnya bahwa Skripsi yang saya tulis ini benar-benar merupakan hasil karya saya sendiri, bukan merupakan pengambil alihan data, tulisan, atau pikiran orang lain yang saya akui sebagai hasil tulisan atau pikiran saya sendiri, kecuali dengan mencantumkan sumber cuplikan pada daftar pustaka.

Apabila dikemudian hari terbukti atau dapat dibuktikan skripsi ini merupakan hasil jiplakan, maka saya bersedia menerima sanksi atas perbuatan tersebut.

Malang, 20 Maret 2025
Yang membuat pernyataan,

Chandra Gunawan
NIM.18650071

MOTTO

"...Life is not just about being happy..."

HALAMAN PERSEMBAHAN

Karya ini saya persembahkan kepada keluarga saya, terutama ibu dan kakak laki-laki saya, Maili Fitma dan Suhardian Kurnia. Merekalah yang memberikan segala hal-hal fundamental bagi saya untuk bisa melakukan semuanya, terutama selama masa perkuliahan ini dan menjadi pribadi seperti sekarang ini. Dengan karya ini juga saya membuktikan kalau pesimis dan penuh keraguan pun dapat menghasilkan dan menguasai hal-hal yang dikira jauh dari genggaman dan imajinasi.

Awalnya projek skripsi ini adalah aktivitas *escapism* yang saya lakukan. Tanpa terasa setahun berlalu dan saya berhasil membuat projek ini, karya ini. Projek yang saya mulai dengan niat yang "kurang bagus" menjadi projek yang paling membantu saya untuk tumbuh sebagai pribadi yang beredukasi.

Sebagai penutup, saya ingin memberikan sebuah pengingat kepada diri saya di masa depan : "Jika semua terasa monoton, jika dunia seakan tidak mendukung, jika percaya diri yang dipunya pun tidak cukup dan menyerah seakan satu-satunya opsi yang tersisa, hadapi dan lakukan saja. Tentu berat dan pahit, tapi percayalah, semuanya akan sepadan pada akhirnya"

KATA PENGANTAR

Assalamu'alaikum Wr. Wb.

Syukur alhamdulillah penulis haturkan kehadirat Allah SWT yang telah melimpahkan Rahmat dan Hidayah-Nya, sehingga penulis dapat menyelesaikan studi di Fakultas Sains dan Teknologi Universitas Islam Negeri Maulana Malik Ibrahim Malang sekaligus menyelesaikan Skripsi ini dengan baik. Selanjutnya penulis haturkan ucapan terima kasih seiring do'a dan harapan jazakumullah ahsanal jaza' kepada semua pihak yang telah membantu terselesaikannya Skripsi ini. Ucapan terima kasih ini penulis sampaikan kepada :

1. Prof. Dr. M. Zainuddin, M.A., Selaku rektor Universitas Islam Negeri Maulana Malik Ibrahim Malang.
2. Prof. Dr. Hj. Sri Harini, M.Si., selaku dekan Fakultas Sains dan Teknologi Universitas Islam Negeri Maulana Malik Ibrahim Malang.
3. Dr. Ir. Fachrul Kurniawan, S.T, M.MT, IPU selaku Ketua Jurusan Teknik Informatika Universitas Islam Negeri Maulana Malik Ibrahim Malang dan dosen pembimbing I yang senantiasa memberikan dorongan dan telah memberikan arahan serta membimbing sehingga penulis dapat menyelesaikan pengerjaan skripsi.
4. Dr. Muhammad Faisal, M.T, selaku dosen pembimbing II yang telah membimbing dan memberi arahan serta masukkan kepada penulis sehingga penulis dapat menyelesaikan skripsi.
5. Keluarga penulis yang selalu membantu dan memberikan motivasi dan dukungan agar penulis dapat menyelesaikan skripsi.

6. Seta Murdha Pamungkas selaku teman penulis yang telah membantu penulis dalam pengumpulan data dan peminjaman fasilitas sehingga penulis dapat menyelesaikan skripsi.
7. Penulis sendiri yang telah berusaha sebaik mungkin agar skripsi ini dapat selesai dengan sebaik-baiknya.

Skripsi yang telah ditulis ini masih jauh dari kata sempurna, oleh karena itu penulis sangat menghargai dan senang jika terdapat kritik dan saran. Semoga skripsi ini memberikan manfaat.

Wassalamu'alaikum, Wr. Wb

Malang, 20 Maret 2025

Penulis

DAFTAR ISI

DAFTAR GAMBAR

DAFTAR TABEL

DAFTAR SIMBOL

Lambang Romawi

Lambang	*Kuantitas*	*Satuan*
dx	Selisih pusat dua lingkaran pada sumbu x	-
dy	Selisih pusat dua lingkaran pada sumbu y	-
x	Posisi pusat lingkaran pada sumbu x	-
y	Posisi pusat lingkaran pada sumbu x	-
M	Nilai *Cylomatic Complexity*	-
E	Jumlah tepi (*edge*)	-
N	Jumlah simpul (*node*)	-
P	Jumlah komponen (*segment*)	-

Singkatan

ECS	*Entity Component System*
CFG	*Control Flow Diagram*
OOP	*Object Oriented Programming*
FPS	*frame per second*
AABB	*Axis-Aligned Bounding Box*
VR	*Virtual Reality*
PC	*Personal Computer*
SFML	*Simple Fast Multimedia Library*

ABSTRAK

Gunawan, Chandra. 2025. **Pengembangan 2D *Game* Engine Menggunakan Arsitektur Entity Component System.** Skripsi. Jurusan Teknik Informatika Fakultas Sains dan Teknologi Universitas Islam Negeri Maulana Malik Ibrahim Malang. Pembimbing: (I) Dr. Ir. Fachrul Kurniawan, S.T, M.MT, IPU (II) Dr. Muhammad Faisal, M.T.

`

Kata kunci: arsitektur, *game engine*, *entity component system.*

Dalam pembuatan *video game* atau *game* komputer dibutuhkan sebuah perangkat lunak *game engine* yang menyediakan semua keperluan dalam membuat *game.* Pengembangan *game engine* pada dasarnya dimulai dengan merepresentasikan objek *game*. Melalui arsitektur *Entity Component System* (ECS), representasi *game* objek menjadi lebih mudah dengan memisahkan data dan implementasi sehingga dapat dengan mudah dimanipulasi dan dipelihara. ECS digunakan sebagai paradigma dan pendekatan berpikir dalam menulis basis kode untuk membangun sistem-sistem *game engine*. Manfaat ECS terlihat melalui pengujian menggunakan metode *Cyclomatic Complexity*. Hasil pengujian menunjukkan rata-rata tiap bagian kode pada basis kode memiliki nilai *Cyclomatic Complexity* yang rendah. Hal ini menunjukkan bahwa, basis kode mudah dibaca dan mudah dipelihara.

ABSTRACT

Gunawan, Chandra. 2025. **2D *Game* Engine Development Using Entity Component System Architecture.** Thesis. Department of Informatics Engineering, Faculty of Science and Technology, Maulana Malik Ibrahim State Islamic University Malang. Adviser: (I) Dr. Ir. Fachrul Kurniawan, S.T, M.MT, IPU (II) Dr. Muhammad Faisal, M.T.

Game engine software is needed to make video games or computer games. Game engine development begins with representing game objects. Through the Entity Component System (ECS) architecture, game object representation becomes easier by separating data and implementation so that it can be easily manipulated and maintained. ECS is used as a paradigm and thinking approach in writing code bases to build game engine systems. The benefits of ECS are seen through testing using the Cyclomatic Complexity method. The test results show that each section of code in the code base has a low Cyclomatic Complexity value. This indicates that the code base is easy to read and easy to maintain.

Key words: architecture, game engine, entity component system.

مستخلص البحث

غوناوان، جاندرا. 2025. تطوير محرك اللعبة ثنائي الأبعاد باستخدام بنية نظام مكونات الكيان. البحث الجامعي. قسم الهندسة المعلوماتية، كلية العلوم والتكنولوجيا، جامعة مولانا مالك إبراهيم الإسلامية الحكومية مالانج. المشرف: (١) الدكتور فخرول كورنيوان الماجستير(2) الدكتور محمد فيصل الماجستير.

الكلمات الأساسية: العمارة، محرك اللعبة، نظام مكون الكيان.

في صناعة ألعاب الفيديو أو ألعاب الكمبيوتر، تحتاج إلى برنامج محرك ألعاب يوفر كافة متطلبات صناعة الألعاب. يبدأ تطوير محرك اللعبة بشكل أساسي بتمثيل كائنات اللعبة. من خلال بنية نظام مكونات الكيان (ECS)، أصبح تمثيل كائنات اللعبة أسهل من خلال فصل البيانات والتنفيذ بحيث يمكن معالجتها وصيانتها بسهولة. يتم استخدام ECS كنموذج ونهج تفكير في كتابة قواعد التعليمات البرمجية لبناء أنظمة محركات الألعاب. يمكن رؤية فوائد ECS من خلال الاختبار باستخدام طريقة التعقيد الدوري. تظهر نتائج الاختبار أن كل جزء من الكود في قاعدة الكود له في المتوسط قيمة تعقيد سيكلوماتيكية منخفضة. يوضح هذا أن قاعدة التعليمات البرمجية سهلة القراءة وسهلة الصيانة.

BAB I

PENDAHULUAN

1.1 Latar Belakang

Teknologi adalah aplikasi ilmu pengetahuan yang bertujuan mempermudah kehidupan manusia. Teknologi merupakan hasil dari pembelajaran manusia yang mana dihasilkan dari belajar. Dengan mengamati dunia, manusia memahami sekelilingnya dan menggunakan pemahaman tersebut untuk memajukan tidak hanya diri sendiri tetapi juga sesama. Sebagaimana yang tercantum pada Surah Yunus ayat ke-101.

قُلِ انْظُرُوْا مَاذَا فِى السَّمٰوٰتِ وَالْاَرْضِۗ وَمَا تُغْنِى الْاٰيٰتُ وَالنُّذُرُ عَنْ قَوْمٍ لَّا يُؤْمِنُوْنَ ﴿١٠١﴾

"Katakanlah (Nabi Muhammad), "Perhatikanlah apa saja yang ada di langit dan di bumi!" Tidaklah berguna tanda-tanda (kebesaran Allah) dan peringatan-peringatan itu (untuk menghindarkan azab Allah) dari kaum yang tidak beriman." (Q.S. Yunus 10:101).

Teknologi mengacu pada berbagai macam jenis dan memiliki kegunaan yang spesifik, ada yang digunakan untuk mempermudah manusia berpindah tempat, yaitu kendaraan bermotor, atau teknologi yang digunakan untuk memberikan informasi dan menghibur seperti buku, yaitu *video game*.

Video game atau *game* komputer adalah permainan elektronik yang melibatkan interaksi antara manusia dan mesin untuk menghasilkan timbal balik visual dan/atau audio dari perangkat seperti monitor komputer atau televisi. *Video* iitu sendiri merupakan perpaduan antara matematika, seni gambar, seni musik dengan medium komputer. Karena pada dasarnya, *video game* adalah simulasi dari realita, sehingga diperlukan perhitungan matematis agar *video game* bisa

berperilaku seperti nyata dan seni untuk menampilkan hasil perilaku tersebut. Pada masa modern seperti sekarang ini, manusia dapat memainkan *video game* dengan berbagai macam cara, seperti menggunakan *joystick*, *keyboard*, *screen-touch* dan yang paling terkini adalah dengan gerakan. Sehingga *video game* semakin mendekati dengan keadaan realita yang dapat diinderakan secara bebas.

Dalam pembuatan *video game*, dibutuhkan sebuah perangkat lunak yang bernama *game engine*. *Game engine* ini menyediakan semua keperluan dalam membuat *game*, seperti *graphics rendering*, audio, *window*, *network* dan banyak hal lainnya. Penggunaan *game engine* sangat mempermudah dalam pembuatan *game*, sehingga pengembang tidak perlu membuat dari awal unsur-unsur mendasar sebuah *game* karena semua hal dasar dalam membuat *game* tersebut telah disediakan oleh *game engine*.

Masalah terbesar yang ditemui dalam pengembangan *game engine* adalah tentang bagaimana merepresentasikan objek *game*. Objek *game* bisa berupa sebuah gambar dua dimensi sederhana tanpa kontrol atau interaksi hingga sebuah objek rumit tiga dimensi dengan kontrol, suara, animasi dan *artificial intelligence* (Hall et al, 2014). Secara konsep, mudah untuk dipahami jika sebuah objek *game* mempresentasikan entitas di dalam sebuah *game* dengan beberapa properti atau fitur. Tetapi akan timbul isu ketika mencoba mengorganisir sebuah arsitektur yang dapat mengurus kombinasi dari fitur atau properti dari sebuah objek *game*.

Pendekatan umum untuk menyelesaikan masalah ini adalah dengan menggunakan arsitektur *Object Oriented Programming* (OOP). Namun dikarenakan sifat *inheritance* dan hirarki dari OOP, kesulitan muncul ketika

mencoba merepresentasikan sebuah objek *game*. Dengan sifat *inheritance*, dapat menimbulkan hirarki yang menyebabkan program sulit untuk dikelola dan diorganisir. Lalu, karena hirarki yang ditimbulkan oleh *inheritance* tersebut, memanipulasi atau menambahkan fungsi baru yang spesifik pada program menjadi sulit. Isu terbesar dari hirarki ini adalah memperluas program. Objek turunan sangat bergantung pada objek induknya, sehingga menambah sebuah fitur atau merubah sebuah fitur yang telah ada menjadi mustahil tanpa rekonstruksi program panjang lebar (Hall et al, 2014).

Pembuatan *game* memerlukan proses iterasi yang tinggi, seperti menambah mekanisme *game* baru atau interaksi fitur yang membutuhkan perubahan pada model desain awal fitur lainnya. Perubahan ini memungkin tumbuhnya konsekuensi yang signifikan dalam pengembangan dan membutuhkan program untuk direfaktor yang mana merugikan dalam hal waktu pengembangan dan bisa menjadi sumber bug (Gay dan Ralston, 2021).

Entity Component System adalah sebuah arsitektur perangkat lunak yang biasa digunakan dalam pengembangan *video game* (Capdevila, 2013). Arsitektur ini menggunakan pendekatan berorientasi data dan terbangun dari tiga konsep. *Entity* (entitas) merepresentasikan objek tetapi tidak mengandung data ataupun fungsi. Sebuah entitas adalah referensi sederhana dari kumpulan *component* (komponen) yang mengandung data. Komponen mendeskripsikan properti dari sebuah entitas, seperti warna, ukuran, kecepatan dan lain sebagainya. Sebuah komponen dapat ditambahkan atau dihapus secara dari sebuah entitas. *System* (sistem) yang mendefinisikan logika *game*. Sistem mengakses komponen dari

entitas-entitas secara berurutan dan memperbarui entitas-entitas tersebut. Sistem memodifikasi data *game* dan kemudian memperbarui simulasi. ECS digunakan untuk menjawab dua masalah, yaitu memperbaiki sifat modularitas program dan memperbaiki performa *game engine* (Gay dan Ralston, 2021) .

Berdasarkan uraian diatas, maka penelitian ini akan mengajukan penerapan arsitektur *Entity Component System* (ECS) dalam pengembangan perangkat lunak *game engine* dua dimensi. Arsitektur ECS dipilih karena dapat mempermudah dalam mengorganisir struktur penulisan kode dan mempermudah jalan dalam memperluas skala *game engine* serta mengamankan performa dari *game engine* itu sendiri. Oleh karena itu, penulis mengangkat judul **"Pengembangan 2D *Game Engine* Menggunakan Arsitektur *Entity Components System*"**.

1.2 Rumusan Masalah

Bagaimana penerapan arsitektur *Entity Component System* dalam pengembangan 2D *Game Engine* ?`

1.3 Batasan Masalah

Penelitian dibatasi oleh hal-hal berikut:

1. Penelitian lebih berfokus pada implementasi arsitektur *Entity Component System* (ECS).
2. *Game engine* menggunakan *Simple and Fast Multimedia Library* (SFML) sebagai penyedia *interface* pada perangkat keras mesin komputer yang digunakan untuk pengembangan.
3. *Game engine* ini berfokus pada pembuatan *game* dua dimensi.

4. Fitur-fitur pada *game engine* ini diimplementasi menggunakan bahasa pemrograman C++.
5. *Game engine* ini memiliki fitur-fitur yang menangani hal-hal dasar dalam pembuatan *game* seperti *scene system*, manajemen *resource*, *input*, *game loop*, *entity manager*, *sprite system*, dan *game physics*.

1.4 Tujuan Penelitian

Penelitian ini bertujuan untuk membangun *game engine* dua dimensi yang menerapkan arsitektur *Entity Component System*.

1.5 Manfaat Penelitian

Manfaat dari penelitian antara lain :

1. Memberikan gambaran tentang seberapa bagus pendekatan arsitektur *Entity Component System* (ECS) dalam pengembangan perangkat lunak, terutama dalam bidang pengembangan *game engine*.
2. Pengembangan *game engine* dua dimensi ini dapat dijadikan referensi mengenai pengembangan teknologi *game engine*.
3. Memberikan pendekatan atau alternatif baru dalam pengembangan perangkat lunak selain pendekatan desain berorientasi objek (OOP)

BAB II

STUDI PUSTAKA

2.1 Penelitian Terkait

Pengembangan *game engine* dalam penelitian yang diteliti oleh (Daniel Hall et al., 2014) mengajukan menggunakan pendekatan *Entity Component System* (ECS)). Penelitian ini bertujuan untuk menjelajahi desain *game engine* dan mengembangkan sebuah desain *game engine* yang modular dan dapat diperluas. Penelitian mengulas desain ECS dengan cara mengembangkan purwarupa *game* sederhana menggunakan dua sistem ECS berbeda, yaitu dengan *frame*work *Cupcake* dan *Artemis*. Kemudian kedua sistem ECS tersebut dites dan dianalisis untuk mencari tahu keunggulan dan kekurangan dari keduanya. Hasil penelitian ini mengajukan arsitektur sistem ECS yang menggabungkan kedua *framework* tersebut karena keduanya dapat saling melengkapi untuk menjaga sifat modular.

Penelitian dengan judul "*A Game Engine Designed to Simplify 2D Video Game Development*" mengajukan pengembangan *game engine* dua dimensi baru yang mengurangi kompleksitas dari proses pembuatan *game*. Penelitian ini mengajukan penyederhanaan spesifikasi *game*, pengurangan kompleksitas arsitektur *engine* dan memperkenalkan *game environment editor* yang mudah digunakan untuk pembuatan *game*. Penelitian ini menghasilkan *game engine* sederhana yang dapat digunakan oleh orang-orang yang minim pengalaman tentang pemrograman (Chover et al, 2020).

Muratet dan Garbarini juga menggunakan pendekatan ECS sebagai arsitektur perangkat lunak untuk mengintegrasi aksesibilitas fitur-fitur dalam sebuah *game* serius yang telah ada. Mereka melakukan penelitian ini dengan cara menyajikan fitur aksesibilitas pada *game* yang dikembangankan dengan ECS, yaitu *E-LearningScape*. Hasilnya, keuntungan dari penerapan arsitektur ECS, yaitu ECS memberikan keunggulan dalam membawa transformasi-transformasi yang berdampak pada sejumlah besar objek *game* dan menghindari manipulasi berulang yang merupakan sumber error (Muratet & Garbarini, 2020).

Hansen dan Öhrström melakukan penelitian perbandingan performa *runtime* berbagai *library open-source* ECS populer ketika mengambil data dari entitas lain selama iteras. Terdapat tiga kasus percobaan yang diterapkan: iterasi linier, iterasi dengan akses, dan iterasi dengan akses yang mengecek jika sasaran masih sah. Mereka juga melakukan tes secara *Object Oriented Design* pada semua *library* ini untuk dibanding dan membuktikan jika kekurangan dari orientasi objek (*object oriented*) masih bisa terlihat didalam tiga kasus tes tersebut. Hasil penelitian ini menunjukan bahwa secara rata-rata, arsitektur ECS mampu memberikan *runtime* performa yang cepat terhadap tiga kasus percobaan tersebut (Hansen & Öhrström, 2020)

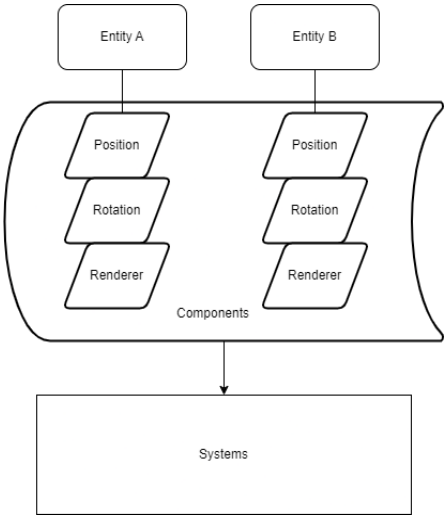
2.2 Desain Berbasis Data (*Data-Oriented Design*)

Data-Oriented Design (desain berbasis data) telah ada selama beberapa dekade dalam berbagai bentuk namun baru secara resmi diberi nama oleh Noel Llopis di artikelnya di tahun 2009. Banyak yang percaya jika hal ini bisa digunakan bersamaan dengan paradigma pemrograman yang lain seperti pemrograman

berorientasi objek, prosedur, atau fungsional. Walaupun desain berbasis data digunakan bersamaan dengan paradigma pemrograman lainnya, namun hal tersebut tidak menghalangi desain berbasis data untuk menjadi salah satu cara pendekatan dalam pemrograman secara utuh (Fabian, 2018).

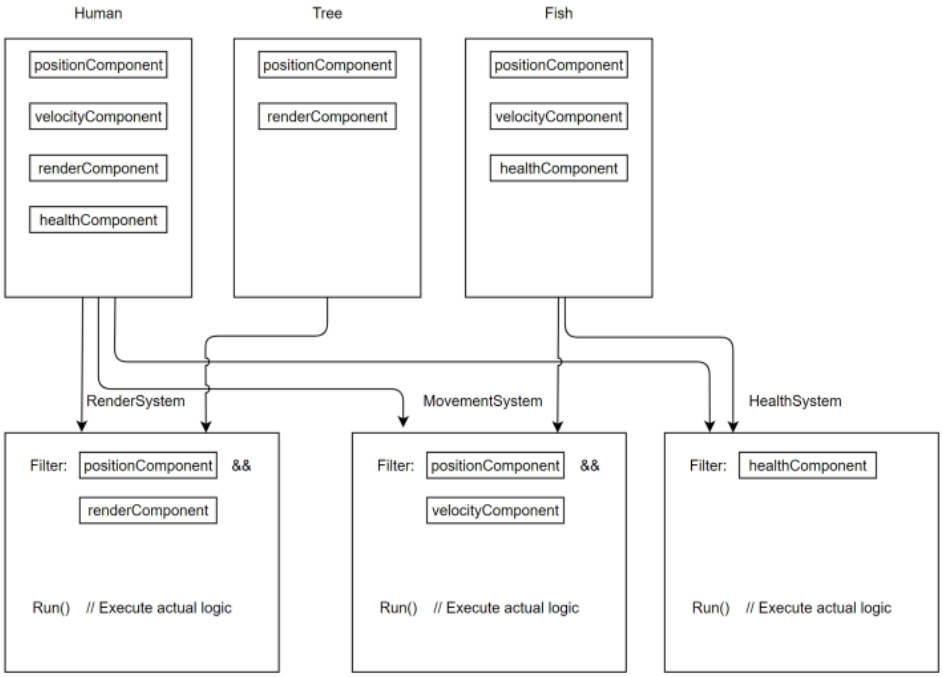
Dalam buku berjudul "*Data-Oriented Design*" oleh Richard Fabian menjelaskan bahwa entitas berbasis komponen merupakan salah satu pendekatan *level* tinggi desain berbasis data. Hal ini merupakan jalan yang menyediakan paket fungsionalitas berbasis data pada entitas-entitas dan memungkinkan desainer untuk mengontrol sesuatu yang biasanya hanya bisa dilakukan oleh programmer. Oleh karena itu, beberapa *game engine* seperti *Unity* menggunakan pendekatan ECS sebagai arsitektur dalam pembangunan perangkat lunak.

2.3 Arsitektur *Entity Component System* (ECS)



Gambar 2.1 Struktur sederhana dari *Entity*, *Component*, dan *System*

Pola arsitektur ECS menggunakan desain berbasis komponen seperti pada gambar 2.1 yang mana menggunakan komposisi dibanding turunan (*inheritance*) untuk memberikan entitas (*entity*) *game* perilaku berbeda. Seperti namanya, entitas sama dengan objek-objek yang di *game world*. Tetapi entitas tidak diimplementasi sebagai objek atau struktur, melainkan mereka hadir sebagai wadah yang menampung berbagai perilaku yang dibutuhkan di dalam sebuah *game world*.



Gambar 2.2 Contoh desain game dengan arsitektur *Entity Component System*

Sebagai contoh pada gambar 2.2, seorang *Human* dengan sebuah komponen *Position*, sebuah komponen *Velocity*, beberapa data untuk *rendering* dan beberapa angka untuk *Health* (Fabian, 2018). Untuk menyediakan entitas dengan fungsionalitas tersebut, komponen (*component*), dan sistem (*system*) digunakan sebagai penghubung. Komponen hanya menyimpan data sedangkan Sistem menggunakan data tersebut untuk melakukan pemrosesan.

Dalam konteks *game* dua dimensi seperti yang terlihat pada gambar 2.2, komponen *Velocity* digunakan untuk menyimpan nilai kecepatan yang berupa nilai X dan Y yang memiliki *tipe* data *float*. Kemudian komponen *Position* merupakan data mengenai posisi X dan Y entitas di dalam *game world* yang keduanya juga memiliki *tipe* data *float*. Lalu, komponen *Health* digunakan untuk merepresentasikan nilai kesehatan entitas yang nilainya berada diantara rentang 0 dan 100. Nilai kesehatan bisa memakai *tipe* data *integer* atau *float*.

Kemudian ada sistem, yang mana berisi kode yang dibutuhkan untuk membuat logika *game*. Tiap sistem beroperasi hanya pada entitas yang memiliki semua komponen yang dibutuhkan untuk sistem bisa bekerja. Melanjutkan contoh sebelumnya, sebuah Sistem *Movement* melakukan iterasi ke seluruh entitas dengan komponen *Position* dan komponen *Velocity*. Setelah itu Sistem *Movement* melakukan komputasi yang mengubah semua posisi entitas di *game world*. Lalu, sebuah sistem *Health* beroperasi pada semua komponen *Health* dan mungkin mengurangi atau menambah nilai kesehatan pada komponen tersebut. Kedua sistem melakukan iterasi secara linier terhadap sekumpulan data agar logika dapat dilakukan.

2.4 *Game Engine*

Penggunaan istilah "*game engine*" pertama kali muncul pada pertengahan tahun 1990-an yang mana mereferensikan *game first-person shooter* (FPS) yang sangat populer berjudul *Doom* oleh *id Software* (Gregory, 2019). *Doom* memiliki arsitektur yang bagus karena memisahkan komponen inti perangkat lunak (seperti grafis *rendering* tiga dimensi, *collision detection system*, atau *audio system*) dan

aset yang digunakan, *game worlds*, dan peraturan-peraturan yang membentuk pengalaman bermain pengguna (Noor Ahmad, 2013).

2D *Game Engine* adalah perangkat lunak yang dirancang khusus untuk membuat dan mengembangkan permainan dua dimensi. Perbedaan utama antara 2D dan 3D *Game Engine* adalah fokusnya pada representasi grafika dalam dua dimensi, yang lebih sederhana daripada dunia tiga dimensi yang lebih kompleks. 2D *Game Engine* dirancang untuk merender dan menangani grafis dua dimensi. Ini mencakup elemen-elemen seperti *sprite*, *tilemap*, teks, dan efek khusus yang digunakan untuk membuat tampilan visual permainan.

Pengembangan *game engine* merupakan proses pembuatan perangkat lunak yang kompleks dan membutuhkan waktu serta tenaga yang banyak. *Game engine* memerlukan waktu yang banyak dikarenakan *game engine* terdiri dari fitur-fitur seperti grafis, suara, *input*, *event handler*, and *window* (Bishop et al, 2016). Untuk mengimplementasi fitur-fitur tersebut diperlukan interaksi langsung dengan *hardware* dari mesin komputer itu sendiri, sehingga akan memerlukan banyak waktu dan tenaga untuk mengimplementasi fitur-fitur rumit tersebut secara mandiri. Oleh karena itu, penggunaan sebuah *library* yang menangani fitur-fitur dasar tersebut akan menjadi sangat membantu dalam proses pengembangan *game engine*. Sehingga pengembang bisa fokus pada implementasi fitur-fitur *game engine* itu sendiri.

2.5 *Game Object*

Dalam pengembangan *game* dan *game engine*, istilah "*game object*" merujuk pada elemen dasar yang membentuk struktur inti dari suatu permainan.

Game object merepresentasikan entitas dalam dunia virtual yang dapat memiliki berbagai atribut, perilaku, dan properti yang dapat dimanipulasi selama permainan.

Game object mewakili entitas dalam permainan, seperti karakter, objek, senjata, atau bahkan objek logika tertentu. Mereka berfungsi sebagai wadah untuk menyimpan informasi terkait entitas tersebut. *Game object* memiliki atribut dan properti yang dapat diatur untuk menggambarkan karakteristik uniknya. Misalnya, karakter mungkin memiliki atribut seperti kecepatan, kesehatan, atau kekuatan, sedangkan objek mungkin memiliki properti seperti warna, bentuk, dan ukuran. *Game object* dapat memiliki perilaku yang terkait dengan mereka. Ini dapat berupa logika atau kode yang mendefinisikan bagaimana *game object* berinteraksi dengan lingkungannya, merespons *input* pemain, atau melakukan tindakan tertentu selama permainan.

Dalam *game engine*, *game object* dapat disusun dalam struktur hierarki. Ini memungkinkan *game developer* untuk mengatur dan mengelola hubungan antara *game object*, seperti membuat *game object* anak atau memaketkan beberapa *game object* bersama untuk memudahkan manipulasi dan pengelolaan. *Game object* menyimpan informasi visual yang diperlukan untuk merender entitas tersebut di layar. Ini termasuk model 3D, tekstur, animasi, dan elemen visual lainnya yang membentuk penampilan *game object*. *Game object* dapat berinteraksi satu sama lain dalam permainan. Mereka dapat bertukar informasi, memicu peristiwa tertentu, atau merespons aksi yang dilakukan oleh *game object* lain atau pemain. *Game object* memiliki siklus hidup, yang mencakup pembuatan, pembaruan, dan penghancuran. *Game engine* akan mengelola siklus hidup ini untuk setiap *game*

object, memastikan mereka dibuat, diperbarui, dan dihancurkan sesuai kebutuhan permainan.

Pemahaman konsep *game object* sangat penting dalam pengembangan *game*, karena hal ini merupakan dasar dari hampir semua elemen dalam permainan dan memungkinkan *game developer* untuk mengatur, mengelola, dan mengimplementasikan logika permainan dengan lebih efisien.

2.6 *Entity Manager*

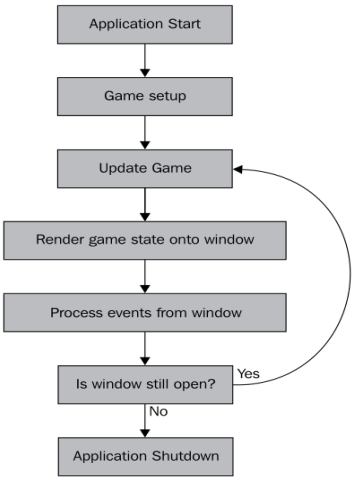
Entity Manager dalam *game engine* adalah komponen yang bertanggung jawab atas pembuatan, pengaturan, dan pengelolaan entitas (*entities*) dalam permainan. Entitas dalam *game engine* adalah objek atau elemen utama yang berpartisipasi dalam permainan, seperti karakter, objek, atau elemen lainnya. *Entity Manager* menyediakan infrastruktur untuk membuat dan mengorganisir entitas, serta memberikan cara untuk mengakses, memanipulasi, dan menyinkronkan perilaku mereka dalam permainan.

Entity Manager dapat menyediakan mekanisme untuk menyimpan dan memuat entitas dari penyimpanan persisten, seperti *file* atau *database*. Ini memungkinkan pemain untuk menyimpan kemajuan permainan atau status entitas untuk digunakan di sesi permainan selanjutnya. Dengan bantuan *Entity Manager*, pengembang dapat mengatur, mengelola, dan menyinkronkan entitas-entitas dalam permainan dengan lebih efisien, yang membantu menciptakan pengalaman permainan yang kohesif dan terstruktur.

2.7 *Game Loop*

Game loop atau bisa juga disebut *main loop* karena fungsi inilah yang mengatur masa hidup dari sebuah aplikasi. Seperti namanya, selama fungsi ini melakukan iterasi, aplikasi akan tetap berjalan. Pada *game*, biasanya aplikasi akan berhenti jika *window* telah ditutup (Moreira, 2013). Tidak seperti program lainnya, *game* tetap berjalan meskipun tidak ada *input* dari pengguna.

Tiap proses dari kejadian dari *window*, kemudian memperbarui *game* dan akhirnya hasilnya ditampilkan di layar disebut sebuah *frame* atau sebuah *tick*. Proses yang terjadi pada sebuah *frame* atau *tick* terlihat seperti pada gambar 2.3. Istilah umum yang digunakan adalah *frame per second* (FPS). Hal ini adalah pengukuran mengenai seberapa banyak iterasi dari *game loop* dari sebuah *game* dalam satu detik.



Gambar 2.3 Contoh *flowchart* dari *game loop*

Penggunaan pengukuran *frame per second* memungkinkan kecepatan komputasi yang konsisten terhadap berbagai perangkat komputer yang memiliki spesifikasi perangkat keras yang berbeda-beda. Oleh karena itu, iterasi *game loop*

dibatasi dengan enam puluh *frame per second*, yang berarti *game loop* hanya akan melakukan iterasi sebanyak enam puluh kali dalam satu detik. Implementasi hal ini dicapai dengan cara memakai waktu yang telah terlewati sejak pemanggilan *frame* terakhir. Hal ini dinamai istilah *delta time*. Jadi, *delta time* menghitung waktu yang diperlukan perangkat keras untuk melakukan iterasi sebuah *frame*.

2.8 *Real-time Rendering*

Rendering adalah proses terakhir dalam pembuatan sebuah gambar atau animasi. Ketika rendering sebuah objek atau karakter dalam sebuah *game* dilakukan pemrosesan bentuk geometris dari sebuah model, tekstur, *shading* dan *lighting*. Hal tersebut biasanya diproses oleh *game engine* ke sebuah gambar digital atau grafis raster.

Real-time rendering merupakan proses menghasilkan sebuah gambar dari sebuah adegan dari sebuah deskripsi yang berbeda dari adegan tersebut. *Real-time* sistem didefinisikan sebagai sistem yang mengandung operasi-operasi yang memiliki keterbatasan waktu. Dengan kata lain, kebenaran dari perilaku sistem tidak bergantung sepenuhnya pada hasil logis dari definisi operasi di dalam sistem, namun pada waktu operasi-operasi ini dilakukan (Chiu, 2008).

Tidak seperti perangkat lunak lain yang hanya melakukan *rendering* jika dibutuhkan. Pengembang *game* memakai konsep *real-time rendering* yang mengabaikan permintaan *frame* yang sebelumnya telah diketahui dan menggambar ke layar tanpa tahu apapun secara cepat mungkin. *Game* biasanya menggunakan menggunakan enam puluh atau tiga puluh *frame per second* (FPS) karena mata manusia tidak dapat membedakan lebih banyak *frame* dari jumlah tersebut.

Real-time rendering pada *game* dicapai dengan menggunakan konsep *double buffering*. *Double buffering* adalah sebuah teknik yang dibuat untuk meniadakan gangguan grafis yang disebabkan oleh tidak sinkronnya rendering. *Double buffering* mendefinisikan dua layar virtual untuk menggambarkan grafis. *Buffer* depan dan *buffer* belakang adalah nama yang dipilih sebagai alamat target *rendering* (Barbier, 2015). *Buffer* depan adalah *frame* yang sedang digambarkan dan ditampilkan pada layar sedangkan *buffer* belakang adalah persiapan *frame* yang akan ditampilkan pada layar. Setelah *frame* pada *buffer* belakang siap, kemudian *frame* akan ditampilkan pada layar. Jadi, *buffer* belakang menjadi *buffer* depan lalu *buffer* depan menjadi *buffer* belakang. Cara ini memastikan *frame* sebelumnya aman tersimpan dan *buffer* yang sedang berjalan bisa diubah kapanpun secara aman tanpa mengganggu apa yang ada di layar.

2.9 *Audio*

Audio dalam *game engine* merujuk pada elemen suara yang digunakan untuk meningkatkan pengalaman permainan dengan menambahkan efek suara, musik, dan dialog. Suara dapat menciptakan atmosfer yang mendalam, memberikan umpan balik kepada pemain, dan meningkatkan kedalaman bermain.

Audio memiliki *tipe file* yang berbagai macam, seperti MP3, WAV, OGG dan AIFF. *Tipe-tipe file* ini memiliki keunggulan dan kekurangannya masing-masing. Terdapat kasus tertentu yang dianjurkan untuk menggunakan *tipe file* tertentu. Misalnya seperti *file* WAV yang memiliki ukuran data yang besar, sehingga akan membutuhkan lebih banyak tempat penyimpanan. Jadi, lebih tidak dianjurkan digunakan sebagai musik latar belakang dibanding dengan *tipe file* MP3.

Tetapi dengan sifat *game engine* yang general, maka memberikan infrastruktur untuk penggunaan berbagai macam *tipe file* audio merupakan suatu keharusan. Oleh karena itu, *game engine* memiliki sistem audio yang menjawab kebutuhan akan hal ini.

Game engine memiliki sistem audio yang bertanggung jawab atas pengelolaan dan pemutaran suara dalam permainan. Fungsi utamanya adalah menyediakan fitur untuk mengintegrasikan efek suara, musik dan dialog ke dalam *game* yang dibuat. Sistem audio juga memberikan cara mudah untuk memuat dan mengorganisir audio data ke dalam *engine* serta fitur untuk memutar audio data tersebut (Gregory, 2019)

`

2.10 *Game Physics*

Physics dalam *game engine* merujuk pada simulasi fenomena fisika di dunia nyata dalam lingkungan permainan komputer. Ini mencakup simulasi gerakan benda, tabrakan, gravitasi, dinamika fluida, dan elemen fisika lainnya untuk menciptakan pengalaman permainan yang lebih realistis.

Dikutip dari Russel dalam tesisnya yang berjudul "*Game Physics: An Analysis of Physics Engine for First-Time Physics Developer*" menjelaskan bahwa objek dalam *game world* biasanya berbentuk persegi panjang terlepas dari representasi grafis dari objek tersebut. Persegi panjang ini adalah *bounding box* (kotak pembatas) yang selaras dengan X dan Y pada *game world* atau dikenal dengan nama *Axis Aligned Bounding Boxes* (AABB). Pemilihan bentuk persegi panjang ini menyederhanakan perhitungan karena *collision detection* (deteksi tabrakan) bisa dieksekusi dengan wilayah X dan Y (Britton, 2018).

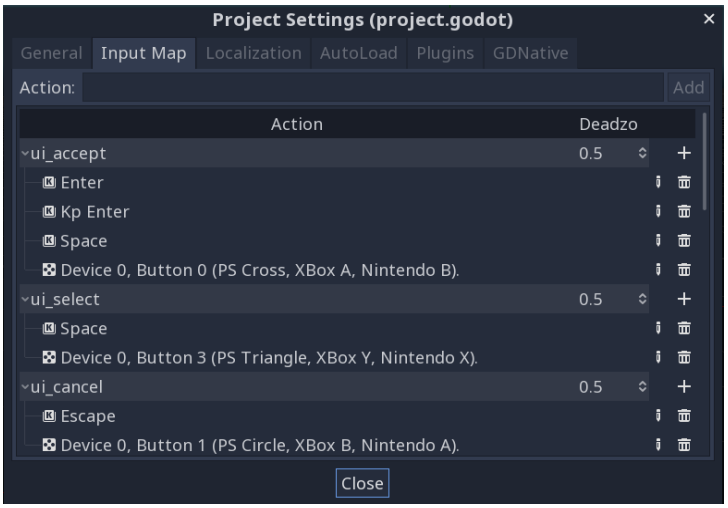
Gambar 2.4 Contoh *Bounding Box* Pada *Game Super Mario Bros*

Collision detection diperlukan agar objek dapat berinteraksi satu sama lain. Pada *game Super Mario Bros*, banyak hal yang dapat dilakukan oleh karakter Mario bisa terjadi dikarenakan adanya *collision detection* (terlihat pada gambar 2.4). Hal-hal tersebut seperti dapat berdiri diatas tanah, tidak menembus tembok, mengambil objek Bintang, dan mengalahkan Goomba (nama musuh pada *game Super Mario Bros*) dengan menginjaknya yang mana semua ini terjadi ketika objek *game* Mario mengalami tabrakan dengan *game* objek lain di dalam *game world*.

2.11 *Input Mapping* (Pemetaan *Input*)

Input dalam *game engine* merujuk pada segala bentuk masukan atau aksi yang diberikan oleh pemain atau perangkat lainnya untuk memanipulasi atau mengontrol permainan. Input dapat berasal dari berbagai sumber, seperti *keyboard*, *mouse*, *joystick*, layar sentuh, atau perangkat lainnya. *Game engine* menyediakan mekanisme untuk menangani *input* ini dan memungkinkan pengembang mengaitkannya dengan tindakan atau respons tertentu dalam permainan.

Game pada *personal computer* (PC) biasanya menggunakan *input keyboard* dan *mouse* untuk dimainkan. Saat ini, banyak jenis perangkat lain yang mempunyai cara peng*input*an seperti *touch screen* pada *smartphone* atau gerakan (*motion*) pada perangkat *Virtual Reality* (VR). Oleh karena hal ini, *game engine* perlu memberikan jalan untuk mengakomodasi berbagai jenis *input* tersebut agar *game* dapat dimainkan pada perangkat-perangkat tersebut. Maka dari itu *game engine* mempunyai sistem *input mapping* untuk menjawab hal ini. Contoh sistem *input mapping* terlihat pada gambar 2.5.

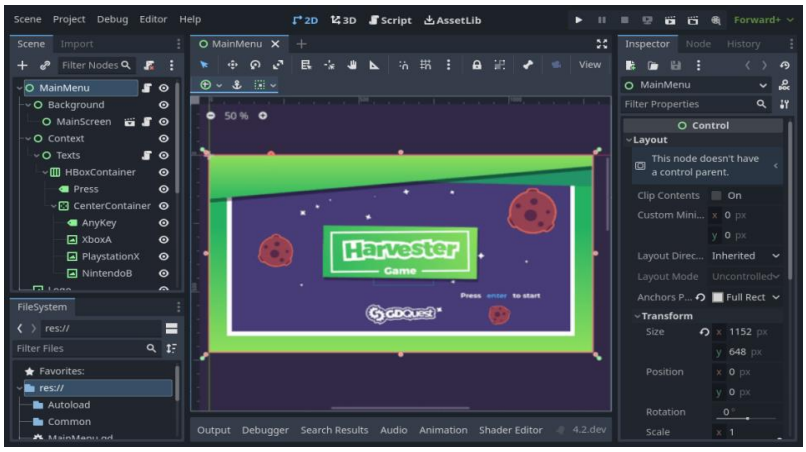


Gambar 2.5 Contoh *input mapping* pada *game engine*

Input mapping adalah proses menetapkan tindakan atau fungsi tertentu ke tombol atau aksi pada perangkat *input*. Seperti pada penelitian yang dilakukan oleh Eun-Seok dan Byeong-Seok, mereka mengajukan sistem *input mapping flexible* yang mempermudah penambahan perangkat *input* baru dengan perangkat *input* yang telah ada, sehingga memungkinkan penggunaan *code* yang telah dengan perangkat baru tanpa perlu mengganggu proses pengembangan (Lee dan Shin, 2021).

2.12 *Scene System*

Dalam *game engine*, *scene* adalah konsep ruang atau lingkungan dalam permainan dimana objek, karakter, dan elemen lainnya berinteraksi. *Scene* adalah wadah untuk menyusun, mengatur, dan mengelola elemen-elemen yang membentuk suatu *level* atau bagian tertentu dari permainan. Dengan *scene*, pengembang bisa fokus pada konten scene seperti objek beserta pengaturannya di dalam *scene* dan berpikir tentang cara terbaik untuk menyajikannya dan bisa melupakan kompleksitas *rendering pipeline* (Cheahand and Ng, 2005).



Gambar 2.6 Contoh scene system pada *game engine*

Game engine menyediakan sistem *scene* sebagai infrastruktur untuk pengembang *game* membuat, mengelola, dan berpindah antar *level* dalam *game* Salah satu contohnya terlihat pada gambar 2.6 yang mana Godot *game engine* menyediakan antar muka visual untuk menyunting *scene*. Sistem *scene* memungkinkan pengembang *game* untuk mengatur objek-objek *game* dalam *scene* menjadi struktur hirarki. Sehingga pengelompokkan dan manipulasi pada objek-objek ini menjadi lebih mudah dan efisien.

2.13 *Resource Management*

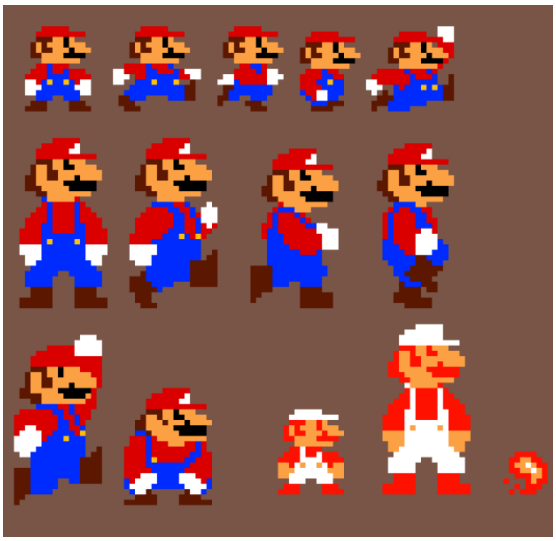
Dalam pengembangan *game*, istilah *resource* (sumber daya) merujuk pada komponen eksternal yang dimuat oleh aplikasi selama *runtime*. Istilah lain yang sering digunakan adalah aset. *Resource* berhubungan dengan berkas-berkas multimedia seperti gambar, musik, atau huruf. *Resource* adalah bagian penting dari pengembangan *game*, dan *game engine* menyediakan sistem untuk mengelola, memuat, dan menggunakan *resource* tersebut. Dikarenakan *game* membutuhkan berbagai *tipe resource*, sistem manajemen *resource* pada *game engine* diperlukan untuk mengelola semua ini secara terpisah.

Ide dibalik sistem manajemen *resource* adalah bagian-bagian kode yang membutuhkan dan kemudian melepas sebuah *resource* dengan *tipe* tertentu. Saat sebuah *resource* diperlukan, *resource* tersebut akan dimuat ke dalam *memory* dan disimpan di sana. Setelah *resource* ini tidak diperlukan lagi, maka akan dilepaskan dan dihapus dari memori (Pupius, 2017). Hal ini memberikan jalan agar *resource* dapat dipakai di berbagai tempat dalam program sehingga bisa berjalan lebih efisien dan memungkinkan manajemen memori untuk *resource* tersebut menjadi lebih mudah.

2.14 *Sprite System*

Sprite system dalam *game engine* adalah infrastruktur yang dirancang untuk membuat, mengelola, dan memanipulasi elemen grafis dua dimensi yang disebut "*sprite*". *Sprite* adalah gambar dua dimensi yang digunakan untuk merepresentasikan karakter, objek, atau elemen visual lainnya dalam permainan. Sistem *sprite* menyediakan alat dan fitur untuk mengatur tampilan dan perilaku

sprite dalam ruang dua dimensi. Pada *game* dua dimensi, biasanya menggunakan kumpulan *sprite* yang dikelompokkan dalam satu berkas atau yang biasa disebut dengan *sprite sheet*. Contoh *sprite sheet* seperti pada gambar 2.7.



Gambar 2.7 Contoh *sprite sheet* karakter *Mario* pada *game Super Mario Bros*

Untuk membuat ilusi gerakan atau animasi pada *game* dapat dicapai dengan menggunakan teknik *sprite masking*. *Sprite masking* adalah teknik yang digunakan dalam *game engine* untuk membatasi render *sprite* ke area tertentu. *Sprite sheet* dibagi menjadi beberapa bagian-bagian persegi panjang sesuai dengan jumlah *sprite* di dalam *sprite sheet*. Bagian-bagian inilah yang disebut jumlah *frame* pada suatu animasi. Jadi, tiap di dalam *game*, tiap bagian tersebut dirender bergantian secara berurutan untuk menciptakan animasi.

2.15 *Control Flow Graph* (CFG)

Control Flow Graph (CFG) memberikan tampilan normal dari semua kemungkinan alur eksekusi program. CFG adalah grafik berarah berakar di mana simpul-simpul mewakili blok dasar dan busur mewakili kemungkinan transfer

kontrol langsung dari satu blok dasar ke blok dasar lainnya (Kontogiannis, 2004). CFG memberikan representasi visual mengenai aliran eksekusi dari sebuah program. Representasi tersebut menggambarkan perpindahan kontrol dari satu bagian kode ke bagian kode lainnya, yang diwakili sebagai simpul (*node*) dan tepi (*edge*). CFG berguna untuk mempelajari struktur program dan mengidentifikasi jalur eksekusi.

2.16 *Cyclomatic Complexity*

Cyclomatic Complexity adalah sebuah metrik pengembangan perangkat lunak yang digunakan untuk mengukur kompleksitas aliran kontrol dari sebuah program. Metrik ini menyediakan perhitungan kuantitatif jumlah jalur linier independen melalui sumber kode program. Konsep ini pertama kali diperkenalkan oleh Thomas J. McCabe Sr. pada tahun 1976. Metode ini digunakan untuk dua maksud, yaitu untuk memberikan jumlah tes dan untuk menjaga perangkat lunak tetap *reliable*, *testable*, dan *manageable* (McCabe & Watson, 1996).

Pada papernya, A dan B mendapatkan temuan implikasi yang signifikan mengenaik penggunaan metode cyclomatic complexity sebagai alat untuk memonitor secara terus menerus tingkat kompleksitas kode mereka. Metode ini juga secara efektif mencegah timbulnya "sphagetti code" yang mana mempersulit basis kode untuk dimengerti dan dipelihara (Odeh *et al*, 2024).

Teknik *cyclomatic complexity* biasanya digunakan untuk menentukan kompleksitas bagian-bagian kode atau fungsionalitas. Teknik ini membantu mengidentifikasi tiga persoalan dari program atau fitur, yaitu tentang tingkat uji,

keterbacaan, dan keandalan (Kalagara, 2020). *Cyclomatic complexity* dicari dengan formula sebagai berikut.

$$\boldsymbol{M = E - N + 2P} \tag{2.1}$$

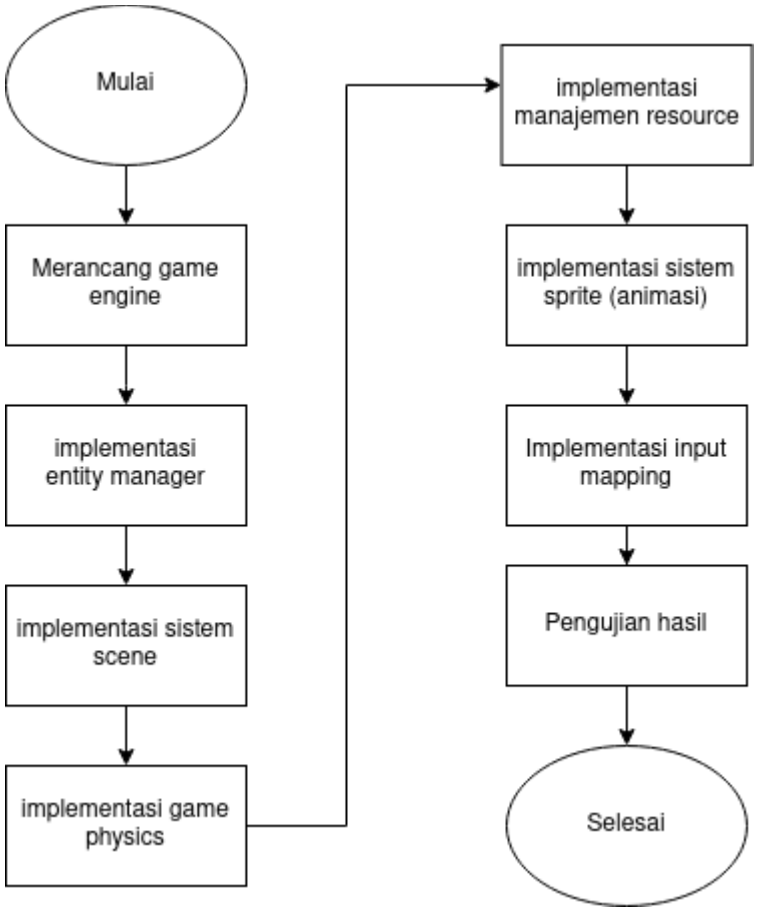
Keterangan :
M adalah nilai Cyclomatic Complexity.
E adalah jumlah tepi (edge) pada grafik aliran kontrol
N adalah jumlah simpul (node) pada grafik aliran kontrol
P adalah jumlah komponen (segment) pada grafik aliran kontrol

Perhitungan dicari dengan merepresentasikan aliran kontrol dengan grafik, lalu dihitung jumlah simpul (*node*), jumlah tepi (*edge*), dan jumlah komponen (*segment*) yang terhubung. Hasil perhitungan diinterpretasikan berdasarkan angka yang dihasilkan. Semakin rendah nilai hasil perhitungan, maka semakin rendah tingkat kompleksitas bagian kode tersebut dan sebaliknya, jika semakin tinggi nilai hasil perhitungan maka semakin tinggi tingkat kompleksitas bagian kode tersebut.

BAB III

DESAIN DAN PERANCANGAN SISTEM

3.1 Alur Penelitian



Gambar 3.1 Alur penelitian

Penelitian ini akan memiliki alur seperti pada gambar 3.1. Untuk penjelasan lengkapnya sebagai berikut :

1. Penelitian dimulai dengan perancangan *game engine* dengan arsitektur *Entity Component System* (ECS). *Game engine* digambarkan sebagai diagram *entity-*

relationship dan *game engine flowchart*. Perancangan ini bermaksud untuk mengkonsep tentang bagaimana *game engine* akan dibangun dan bagaimana *game engine* ini bekerja.

2. Setelah perancangan *game engine* selesai, penelitian dilanjutkan dengan pengimplementasian *game loop* sebagai pondasi awal dari sebuah *game engine*. *Game loop* diimplementasi berdasarkan API yang disediakan oleh *multimedia library* yang dipakai. Pada *game loop* juga menunjukan bagaimana *real-time rendering* bekerja.
3. Pada tahap implementasi *entity manager* ini sekaligus menunjukkan bagaimana *Entity* dan *Component* berinteraksi. *Entity manager* bertugas melakukan pembuatan dan penghapusan sebuah entitas serta komponennya. Pada tahap ini *game engine* sederhana sudah mulai terlihat.
4. Setelah itu, penelitian dilanjutkan dengan pengimplementasian sistem *scene*. *Sistem scene* dilakukan dengan memanfaatkan fitur dari fitur pemrograman berbasis objek *inheritance* yang disediakan oleh bahasa pemrograman yang digunakan. *Inheritance* digunakan sebab, tiap *scene* memiliki basis properti yang sama, sehingga dengan pemanfaatan *inheritance*, jumlah kode yang ditulis menjadi lebih sedikit dan efisien,
5. Kemudian implementasi *game physics* untuk mendapatkan interaksi di dalam *game world* yang pada umumnya ada di suatu *game*. Logika *game physics* yang diimplementasi secara spesifik adalah perhitungan deteksi tabrakan (*collision detection*). Diantaranya adalah deteksi tabrakan antara dua objek lingkaran dan deteksi tabrakan antara objek segi empat atau yang dikenal dengan nama *Axis*

Aligned Bounding Box (AABB) dan lingkaran. Pada tahap ini, *game engine* telah mampu menciptakan *game* sederhana dengan menggunakan bentuk-bentuk geometri sederhana seperti persegi atau lingkaran.

6. Setelah itu penelitian dilanjutkan dengan pengimplementasian sistem manajemen *resource* untuk membaca, memuat dan menyimpan aset grafis berupa *sprite*, *sound*, dan *font*. Pada tahap ini, *game engine* telah mampu menggunakan membuat *game* yang lebih imersif.
7. Pada tahap ini, sistem *sprite* atau animasi diimplementasi. Pada sistem ini, arsitektur ECS tidak diterapkan. Logika tentang bagaimana animasi bekerja terdapat pada satu kelas yang sama. Hal ini dilakukan karena animasi tidak berhubungan dengan sistem lain, sehingga sistem dapat diorganisir dengan lebih baik.
8. Pada tahap ini, *input* dirancang menjadi sebuah sistem sendiri. *Input* dapat diubah secara dinamis dan mudah tanpa perlu menyentuh sumber ode secara langsung.
9. Pada tahap terakhir, hasil implementasi sistem-sistem pada *game engine* dilakukan uji dengan menghitung nilai *cyclomatic complexity* dari basis kode dan membandingkan kasus penggunaan yang telah diimplementasi dengan salah satu *game engine*, yaitu Godot Engine.

3.2 Fokus Penelitian

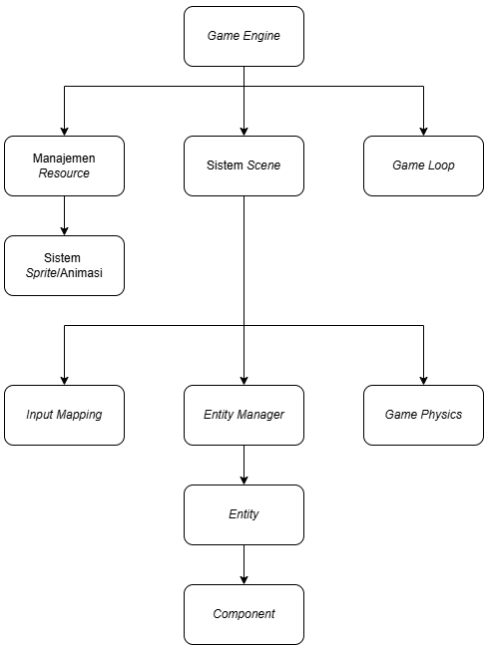
Penelitian ini berfokus pada pembuatan *game engine* dua dimensi dengan menerapkan arsitektur *Entity Component System* (ECS) sebagai pendekatan dalam membangun dan mengorganisir *game engine*. Seperti namanya, ECS memisahkan

game engine ke dalam tiga elemen, yaitu *Entity*, *Component*, dan *System*. *Entity* merepresentasikan *game object* yang memiliki kumpulan komponen. Komponen berfungsi menyimpan data-data yang digunakan oleh *Entity*. Kemudian, *System* bertanggung jawab untuk memanipulasi *Component* yang ada pada *Entity*, atau dengan kata lain *System* mengandung logika yang dibutuhkan dalam suatu *game*. Ketiga elemen tersebut, diorganisir secara terpisah dan independen. Implementasi arsitektur ini memanfaatkan fitur-fitur orientasi objek yang diadopsi oleh bahasa pemrograman.

Penelitian ini menggunakan bahasa pemrograman yang mengadopsi paradigma pemrograman berbasis objek, C++ dan *Simple Fast Multimedia Library* (SFML). SFML adalah *library multimedia* yang menyediakan sebuah lapisan antara pengembang dan perangkat keras (Moreira, 2013). *Library* ini menangani hal-hal fundamental yang diperlukan *game engine*, sehingga peneliti tidak perlu secara langsung berinteraksi dengan perangkat keras dan dapat langsung mengembangkan *game engine*.

3.3 Perancangan *Game Engine*

Game engine dua dimensi yang dibangun pada penelitian ini memiliki sifat general atau dapat digunakan untuk membuat berbagai jenis macam *game* dua dimensi dan sifat modular, yaitu *game engine* dapat dengan mudah diperluas dan dimanipulasi. Pendekatan arsitektur ECS menyebabkan rancangan *game engine* menjadi lebih linier. Berikut rancangan *game engine* dua dimensi yang akan dibangun.



Gambar 3.2 Desain *game engine*

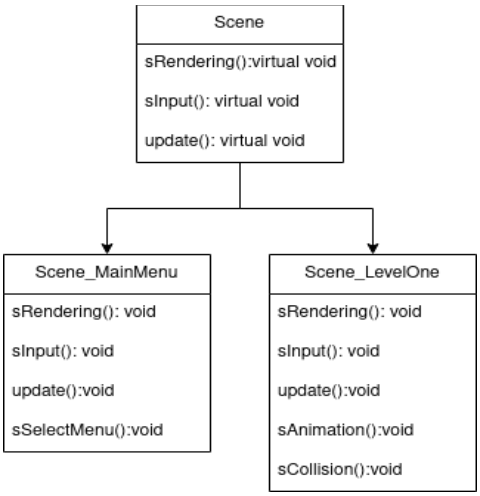
Game engine dirancang dengan *game loop* seperti pada gambar 3.2. Sistem *scene* dan sistem manajemen *resource* berada pada lapisan yang sama. Sistem manajemen *resource* ditempatkan pada *layer* yang sama dengan *game loop* dan sistem *scene* karena *resource* adalah data yang akan digunakan di berbagai bagian kode. Kemudian pada sistem *scene*, terdapat tiga sistem, yaitu *input mapping*, *entity manager*, *game physics*. Ketiga sistem ini adalah sistem-sistem yang selalu dibutuhkan dalam pembuatan *scene*. Pada sistem *entity manager*, digunakan untuk membentuk dan menghapus entitas serta komponennya. Setelah itu sistem *sprite* atau animasi menjadi salah satu bagian dari *resource* yang akan disimpan oleh sistem *resource*.

3.4 Perancangan Sistem

Pada penelitian *game engine* dua dimensi ini terdapat beberapa sistem yang dibutuhkan. Sistem yang dibutuhkan antara lain sistem *scene*, sistem manajemen *resource*, *input mapping*, sistem *sprite* atau animasi, dan *entity manager*.

3.4.1 Perancangan Sistem *Scene*

Dalam tiap *scene*, nantinya terdapat metode-metode yang mengatur *rendering*, logika *game*, dan jenis *input* tersendiri. Namun walaupun begitu, secara general, *scene* membutuhkan metode-metode tersebut untuk bisa bekerja, walaupun diimplementasi secara berbeda. serta, sebuah *game* tentunya memerlukan lebih dari satu *scene*, seperti menu utama dan tingkatan *level* yang berbeda-beda. Oleh karena itu, *scene* perlu *abstract class*.

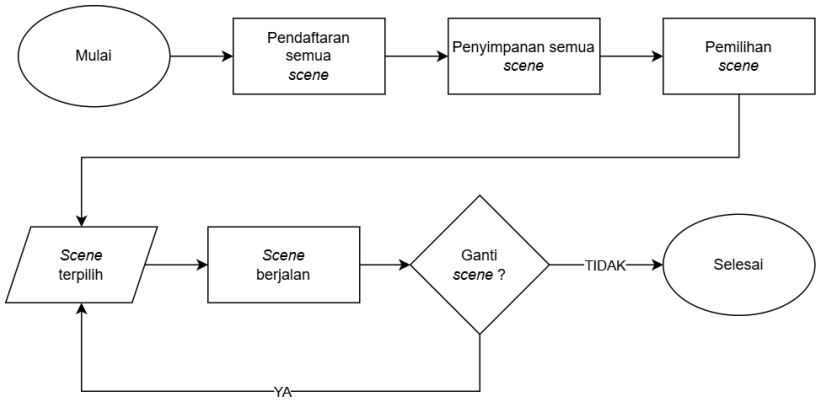


Gambar 3.3 Desain sistem *scene*

Scene merupakan sebuah *abstract class* yang berisi metode-metode mendasar yang dibutuhkan sebuah *Scene*, yaitu metode *rendering*, *input*, dan *update*. Pada gambar 3.3 desain sistem diatas, *class Scene LevelOne* dan *class Scene*

MainMenu merupakan turunan dari *abstract class* tersebut, sehingga keduanya memiliki metode-metode dari *abstract class*. Lalu setiap *scene* memiliki implementasi yang tersendiri, seperti metode *sSelectMenu* pada *class Scene MainMenu* agar dapat bekerja sebagai menu utama pada *game*, kemudian metode *sAnimation* dan *sCollision* pada *class LevelOne* sebagai sistem animasi dan sistem deteksi tabrakan pada *game*.

Untuk pergantian antar *scene*, akan ditangani pada *class Game Engine*. Pada *class* ini, tiap-tiap *scene* yang akan digunakan didaftarkan dan disimpan. Kemudian, dengan *class* ini juga yang menangani pergantian *scene*. Berikut *flowchart* tentang pergantian *scene*.

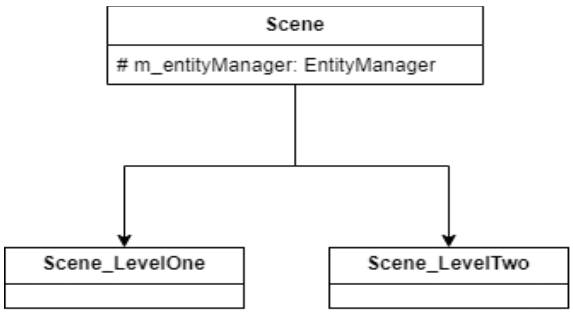


Gambar 3.4 *Flowchart* sistem *scene*

Pada gambar 3.4, sistem *scene* diawali dengan pendaftaran semua *scene* yang akan digunakan di dalam *game*, kemudian semua *scene* tersebut disimpan. Setelah itu, ketika *game* telah berjalan, pergantian *scene* dapat dilakukan. Jika akan terjadi perubahan *scene*, maka akan dilakukan pemilihan *scene* yang kemudian akan dijalankan.

3.4.2 Perancangan *Entity Manager*

Entity manager dalam penelitian ini dirancang dengan peran seperti suatu pabrik, yang akan memproduksi dan menghapus entitas yang digunakan dalam *game*. Sistem ini juga bertugas menyimpan tiap entitas yang sedang digunakan di dalam *game*. *Entity manager* juga merupakan hal yang wajib dimiliki oleh suatu *scene*. Berikut adalah diagram dari *entity manager*.



Gambar 3.5 Diagram *entity manager*

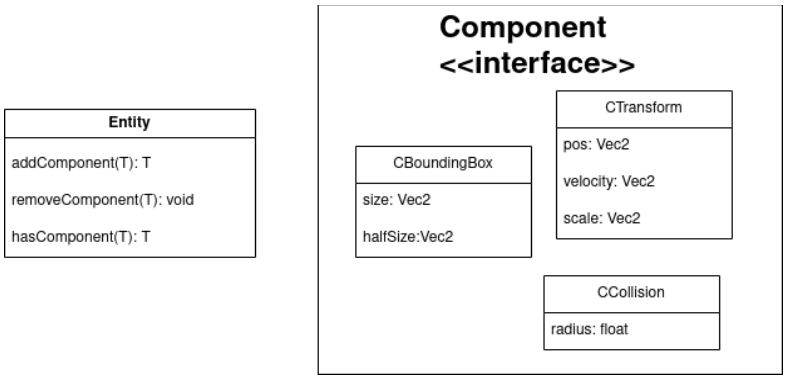
Pada gambar 3.5 diatas, menunjukkan bahwa *Entity Manager* merupakan properti dari *abstract class Scene*, sehingga *Entity Manager* merupakan hal yang wajib dimiliki sebuah *scene* dan dapat digunakan pada semua turunan *scene* tanpa pengulangan deklarasi.

3.4.3 Perancangan *Entity* dan *Component*

Pada penelitian ini, *Entity* dan *Component* diimplementasi menggunakan fitur yang didukung oleh bahasa pemrograman C++, yaitu *template* dan *interface*. *Template* adalah fitur dalam bahasa pemrograman C++ yang memungkinkan sebuah metode untuk menerima *input* parameter tanpa perlu menentukan *tipe* data yang diterima, sehingga metode dapat menerima *input* parameter dengan *tipe* data

apapun. *Template* dipakai karena nantinya *Component* akan memiliki *tipe* data yang berbeda.

Kemudian untuk implementasi *Component*, fitur *interface*. *Interface* hanya mendeskripsikan sifat atau kapabilitas dari suatu kelas. Dikarenakan *Component* pada arsitektur *Entity Component System* hanya mengandung data atau informasi tanpa implementasi logika apapun menjadikan penggunaan fitur *interface* menjadi hal yang tepat. Berikut rancangan *Entity* dan *Component* yang diimplementasi pada penelitian ini.

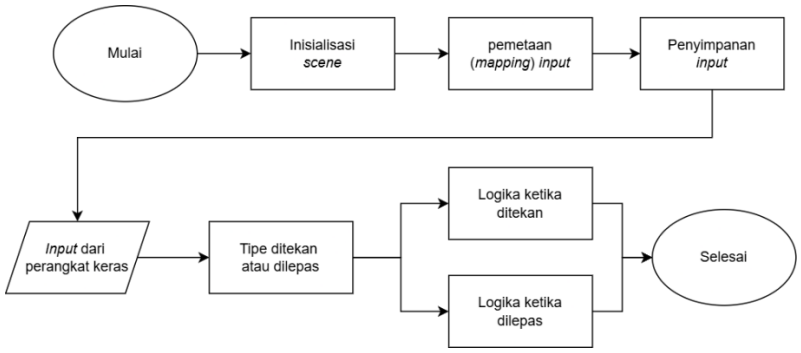


Gambar 3.6 Desain *Entity* dan *Component*

Pada gambar 3.6, *Entity* digunakan untuk menampung *component* sehingga *entity* memiliki metode untuk menambahkan komponen, menghapus komponen, dan memastikan komponen yang dimiliki. Lalu pada bagian *component* sendiri merupakan interface sebagai penampung data, seperti *component transform* yang menampung data mengenai posisi, kecepatan, dan skala dari sebuah *entity* atau *component bounding box* yang berisi data tentang ukuran dan setengah ukuran dari *bounding box* dari sebuah *entity* yang digunakan pada sistem *collision detection* nantinya.

3.4.4 Perancangan *Input Mapping*

Peng*input*an dirancang dengan memetakkan *input* dari perangkat keras seperti *keyboard*, *mouse*, *gamepad*, atau lainnya. Untuk *input* yang berasal dari *keyboard* dan *mouse*, telah disediakan *Application Programming Interface* (API) oleh *library* SFML yang digunakan. Sehingga, sistem dirancang dengan menyimpan nama dan *tipe* dari *input* yang telah dikonfigurasikan. *Tipe* disini dimaksud dengan kondisi *input* ditekan atau dilepaskan. *Input* awalnya didaftarkan pada fase inisialisasi sebuah *scene*, lalu implementasi logika dari *input* yang telah didaftarkan tersebut ditangani oleh sistem tersendiri. Berikut *flowchart* dari sistem *input*.

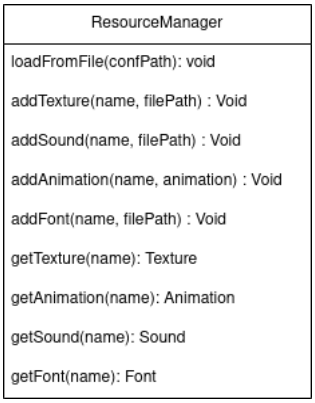


Gambar 3.7 *Flowchart input mapping*

Pada gambar 3.7, sistem *input* bermulai ketika inisialisasi dari sebuah *scene*, karena tiap *scene* mungkin memiliki pemetaan *input* yang berbeda. Lalu pemetaan *input* tersebut didaftarkan dan disimpan. Kemudian *input* yang dipetakan tersebut telah dapat digunakan pada *scene* yang berjalan. Lalu, *Input* dari perangkat keras akan dicek terlebih dahulu kondisinya, apakah ditekan atau dilepaskan, lalu akhirnya dijalankan logika berdasarkan kondisi tersebut.

3.4.4 Perancangan Sistem Manajemen *Resource*

Pada penelitian ini, sistem manajemen *resource* dirancang dengan tujuan mempermudah proses memuat dan menyimpan *resource* dari penyimpanan pada perangkat. Caranya adalah dengan menggunakan *file* teks konfigurasi yang berisi *tipe resource*, nama, dan alamat *resource* pada perangkat. Teks konfigurasi nantinya akan dibaca pada sistem, lalu disimpan pada memori agar dapat digunakan pada saat *game* berjalan. *Tipe-tipe resource* yang bisa disimpan adalah *sound*, tekstur atau gambar, *font*, dan animasi. Berikut adalah rancangan dari sistem manajemen *resource*.

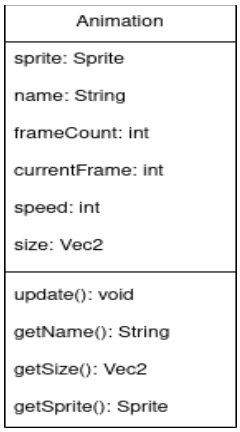


Gambar 3.8 Desain manajemen *resource*

Pada gambar 3.8, *Resource Manager* didesain sebagai penampung dari semua *resource* yang akan digunakan pada *game*, sehingga memiliki metode untuk membaca, memuat, dan menyimpan data dari perangkat keras penyimpanan, dan metode untuk menambahkan dan menyiapkan *resource* seperti *texture*, *sound*, *animation*, dan *font*. Kemudian *resource manager* juga memiliki metode untuk mendapatkan semua *resource* tersebut agar bisa digunakan di dalam *game*.

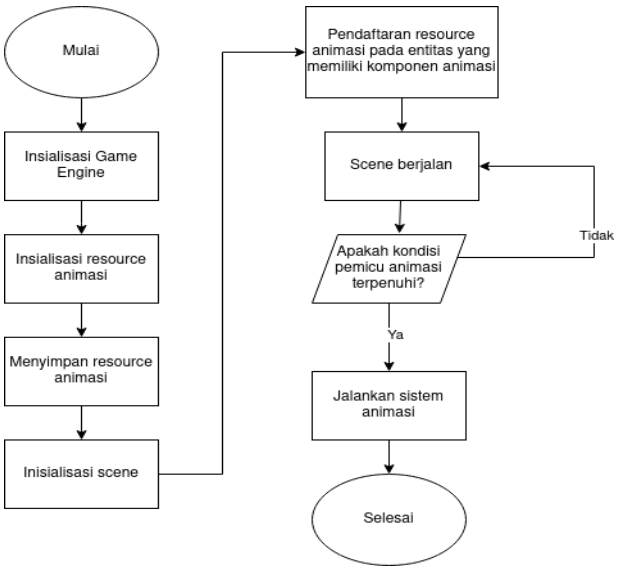
3.4.5 Perancangan Sistem *Sprite* (Animasi)

Sistem *sprite* atau animasi pada penelitian dirancang tanpa penerapan arsitektur *Entity Component System*, yaitu pemisahan antara implementasi logika dan data. Hal ini dilakukan karena sistem *sprite* memerlukan simulasi perhitungan *frame* sendiri dan implementasi dari sistem tidak berinteraksi dengan sistem atau data yang lain serta di setiap *scene* menerapkan logika yang sama sehingga lebih mudah jika tanpa menerapkan arsitektur ECS. Berikut rancangan sistem sprite.



Gambar 3.9 Desain sistem *sprite*

Rancangan pada gambar 3.9, Sistem *sprite* atau animasi ini diimplementasi dengan cara menetapkan posisi dari *sprite sheet* yang akan di-*render*. *Resource* tekstur berupa *sprite sheet* yang telah disimpan di memori serta konfigurasi tentang jumlah *frame* dan kecepatan animasi dari *file* teks konfigurasi digunakan untuk menjalankan sistem, lalu sistem ini dijalankan pada *scene*. Sistem ini juga disimpan sebagai sebuah komponen tersendiri, sehingga animasi dapat dipanggil oleh entitas-entitas yang memiliki komponen animasi tersebut. Berikut adalah *flowchart* dari sistem *sprite*.

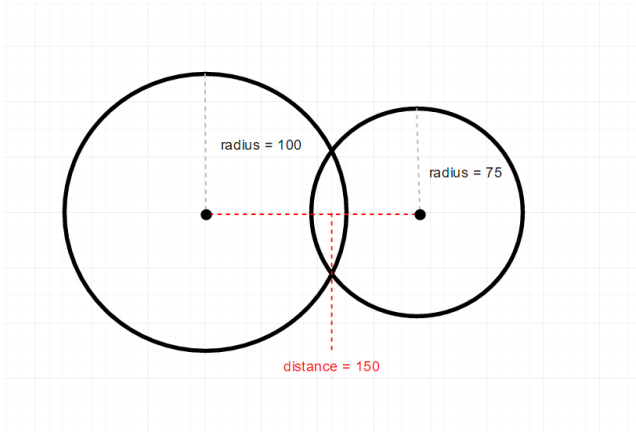


Gambar 3.10 *Flowchart* sistem *sprite* atau animasi

Pada gambar 3.10, Alur dari *sistem sprite* dimulai dari *game engine* dijalankan. Setelah itu *resource sprite sheet* dimuat, disiapkan, dan disimpan. Setelah itu animasi didaftarkan pada entitas yang memiliki komponen animasi pada fase inisialisasi *scene*. Animasi yang telah didaftarkan siap digunakan dalam *scene* yang berjalan dan sistem ini akan berjalan jika kondisi pemicu terpenuhi.

3.5 *Game Physics* (Deteksi Tabrakan)

Game engine pada penelitian ini, merepresentasikan *game object* dengan bentuk persegi panjang atau lingkaran. Representasi bentuk ini adalah dasar dari pengimplementasian *game physics*. Penelitian ini melakukan perhitungan untuk mendeteksi tabrakan (*collision detection*) antar *game object*. Entitas-entitas yang bisa bertabrakan memiliki komponen *Bounding Box* (untuk bentuk persegi panjang) atau komponen *Collision Radius* (untuk bentuk lingkaran). Komponen inilah yang digunakan untuk melakukan perhitungan deteksi tabrakan.



Gambar 3.11 Penggambaran deteksi tabrakan lingkaran

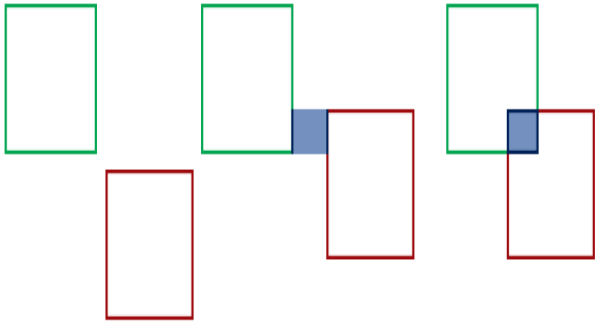
Pada gambar 3.11, tabrakan terjadi ketika kedua lingkaran bersinggungan. Caranya dengan menghitung nilai garis lurus (*distance*) antara kedua pusat lingkaran. Jika panjang garis tersebut lebih dari atau sama dengan jumlah radius atau jari-jari kedua lingkaran, maka kedua lingkaran tersebut bertabrakan.

Untuk *game object* dengan representasi bentuk lingkaran, deteksi tabrakan dihitung dengan mencari titik singgung antara dua lingkaran. Jika jarak antara dua pusat kedua lingkaran dalam titik koordinat lebih dari sama dengan jarak antara pusat kedua lingkaran, maka kedua lingkaran tersebut bertabrakan. Berikut adalah rumus jarak antara dua pusat kedua lingkaran dalam titik dan rumus jarak antara pusat kedua lingkaran.

$$\boldsymbol{d} = \sqrt{(\boldsymbol{x_2} - \boldsymbol{x_1})^2 + (\boldsymbol{y_2} - \boldsymbol{y_1})^2} \tag{3.1}$$

$$\boldsymbol{d} = (\boldsymbol{r_1} + \boldsymbol{r_2})^2 \tag{3.2}$$

Keterangan :
d adalah garis singgung persekutuan dua lingkaran
x dan y adalah koordinat pusat lingkaran pada sumbu x dan sumbu y
r adalah radius atau jari-jari lingkaran



Gambar 3.12 Penggambaran deteksi tabrakan Axis *Aligned Bounding Box*

Seperti pada gambar 3.12, untuk *game object* dengan representasi bentuk persegi panjang, deteksi tabrakan dihitung dengan mendeteksi posisi dari antara dua kotak pembatas (*Bounding Box*). Hal ini dikenal dengan nama *Axis Aligned Bounding Box* (AABB). Jika kedua kotak pembatas berpotongan secara vertikal dan horizontal, maka kedua kotak pembatas tersebut bertabrakan. Berikut rumus yang digunakan untuk mendeteksi tabrakan antara dua kotak pembatas.

$$\boldsymbol{dx = x_2 - x_1} \quad (3.3)$$

$$\boldsymbol{dy = y_2 - y_1} \quad (3.4)$$

Keterangan :
dx dan dy adalah selisih dari pusat dua persegi panjang pada sumbu x dan sumbu y.
x dan y adalah koordinat dari pusat persegi panjang pada sumbu x dan sumbu y.

3.5.1 Contoh Implementasi Perhitungan Deteksi Tabrakan Lingkaran

Contoh implementasi perhitungan dari deteksi tabrakan pada lingkaran direpresentasikan dalam bentuk *pseudocode* (gambar 3.13) berikut.

```
FUNCTION CircleCollision(circle1, circle2):
   deltaX = circle2.center.x - circle1.center.x
   deltaY = circle2.center.y - circle1.center.y
   distanceSquared = deltaX^2 + deltaY^2
   radiusSumSquared = (circle1.radius + circle2.radius)^2
   IF distanceSquared <= radiusSumSquared:
      RETURN TRUE
   ELSE:
       RETURN FALSE
```

Gambar 3.13 *pseudocode* perhitungan tabrakan lingkaran

Perhitungan dimulai dengan perhitungan jarak titik pusat antara dua lingkaran dengan menghitung selisih keduanya pada sumbu X dan sumbu Y. Kemudian hasil perhitungan jarak tersebut dikuadratkan masing-masing lalu dijumlahkan. Setelah itu dihitung jumlah dari radius atau jari-jari kedua lingkaran dan kemudian dikuadratkan. Lalu kemudian dicek, apakah nilai kuadrat jarak yang dijumlahkan kurang dari atau sama dengan jumlah radius yang dikuadratkan. Bila demikian, maka kedua lingkaran bertabrakan. Sebaliknya, jika tidak demikian, maka kedua lingkaran tidak bertabrakan.

3.5.2 Contoh Implementasi Perhitungan *Axis-Aligned Bounding Box*

Contoh implementasi perhitungan dari deteksi tabrakan pada persegi panjang (*Axis Aligned Bounding Box*) direpresentasikan dalam bentuk *pseudocode* (gambar 3.14) berikut.

```
FUNCTION CheckAABBCollision(AABB1, AABB2):
   IF AABB1.maxX < AABB2.minX OR AABB1.minX > AABB2.maxX:
      RETURN FALSE
   IF AABB1.maxY < AABB2.minY OR AABB1.minY > AABB2.maxY:
       RETURN FALSE
   RETURN TRUE
```

Gambar 3.14 *pseudocode* perhitungan tabrakan *axis-aligned bounding box*

Perhitungan *axis aligned aounding box* dilakukan dengan cara mengecek titik kiri, titik kanan, titik atas, dan titik bawah dari kedua kotak pembatas (bounding box). Untuk titik kiri adalah nilai terendah pada sumbu X, titik kanan adalah nilai tertinggi dari sumbu X, nilai titik atas adalah nilai terendah dari sumbu Y, dan nilai titik bawah adalah nilai tertinggi dari sumbu Y. Kondisi pertama pada pseudocode diatas adalah untuk mengecek, apakah kedua kotak pembatas tidak berpotongan pada sumbu X atau berpotongan secara horizontal. Kemudian kondisi kedua untuk mengecek, apakah kedua kotak pembatas tidak berpotongan pada sumbu Y atau berpotongan secara vertikal. Jika salah satu dari kondisi tersebut terpenuhi, maka kedua kotak pembatas tidak bertabrakan. Sebaliknya, jika kedua kondisi tersebut tidak terpenuhi, maka kedua kotak pembatas tersebut bertabrakan.

3.6 Pengujian Hasil dengan *Cyclomatic Complexity*

Basis kode dari *game engine* dilakukan perhitungan *cyclomatic complexity* untuk mencari tingkat kompleksitas aliran kontrol tiap bagian kode pada basis kode. Setelah didapat nilai *cyclomatic complexity* pada tiap bagian kode, analisis dilakukan.

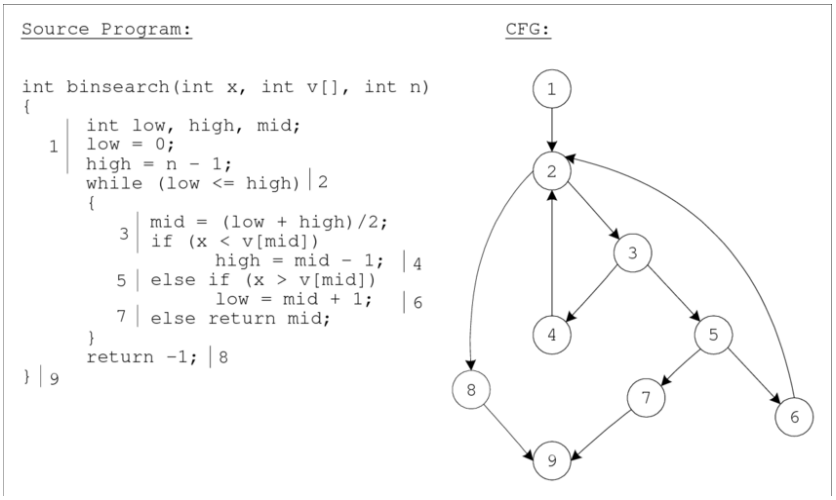
Tabel 3.1 Matrik penilaian tingkat kompleksitas dari Kalagara

Tingkat Kompleksitas	Arti
1 – 10	Terstruktur dan ditulis dengan baik, biaya pengetesan dan upaya rendah.
11 – 20	Kode kompleks, biaya pengetesan dan upaya sedang.
21 – 40	Kode sangat kompleks, biaya pengetesan dan upaya tinggi.
> 40	Tidak semua bagian kode dapat dites, biaya dan upaya sangat tinggi.

Penggunaan perhitungan *cyclomatic complexity* berguna untuk menghitung kualitas penstrukturan kode, penulisan kode, tingkat pengetesan dan tingkat pemeliharaan. Seperti metrik perhitungan yang diajukan oleh Kalagara (2020) pada tabel 3.1, tingkat semakin tinggi nilai *cyclomatic complexity*, maka semakin tinggi tingkat kompleksitas struktur dan penulisan kode, serta semakin susah untuk dilakukan pengetesan dan pemeliharaan.

3.6.1 Contoh Perhitungan *Control Flow Graph* (CFG)

Perhitungan *cyclomatic complexity* dilakukan dengan membuat grafik aliran kontrol dari bagian kode yang akan dihitung nilai *cyclomatic complexity*-nya. Grafik aliran kontrol berguna dalam menghitung jumlah tepi, jumlah simpul, dan jumlah segmen yang ada. Contoh dari grafik aliran kontrol dari sebuah bagian kode adalah sebagai berikut.



Gambar 3.15 Contoh grafik aliran kontrol dalam bahasa pemrograman C

Dari gambar 3.15 (bagian kiri), *flow* dari bagian kode dimulai dari awal *scope* yang ditandai dengan kurung kurawal buka. Simpul pertama adalah tiga baris pertama mengenai deklarasi variabel-variabel setelah kurung kurawal buka. Kemudian simpul kedua adalah *while loop*, simpul ketiga adalah deklarasi variabel di dalam *scope while loop* dan ekspresi kondisi *if*; simpul keempat adalah ekspresi komputasi di dalam *scope if*; simpul kelima adalah cabang kondisi *else if*; simpul keenam adalah ekspresi komputasi di dalam *scope else if*; simpul ketujuh adalah cabang kondisi *else*; simpul kedelapan adalah ekspresi *return*; dan simpul kesembilan adalah akhir dari scope baris kode. Hasil dari penentuan simpul-simpul tersebut kemudian disambungkan dengan tepi sesuai dengan aliran kontrol program dan menghasilkan CFG seperti pada gambar (bagian kanan).

3.6.2 Contoh Perhitungan *Cyclomatic Complexity*

Pada gambar 3.15 (bagian kanan) terlihat bahwa terdapat sembilan simpul (*node*), sebelas panah atau disebut tepi (*edge*), dan hanya satu kesatuan simpul yang

terhubung atau disebut komponen (*segment*). Dengan begitu perhitungan dengan formula.

$$\boldsymbol{M = E - N + 2P} \quad (3.5)$$

Keterangan :
M adalah nilai Cyclomatic Complexity.
E adalah jumlah tepi (edge) pada grafik aliran kontrol
N adalah jumlah simpul (node) pada grafik aliran kontrol
P adalah jumlah komponen (segment) pada grafik aliran control

Hasil perhitungan

$M = 11 - 9 + 2\ (1)$

$M = 5$

Dari hasil perhitungan menghasilkan nilai *cyclomatic complexity* dari contoh pada gambar 3.15 adalah lima. Dengan nilai tersebut dapat disimpulkan bahwa bagian kode pada contoh tersebut memiliki tingkat *cyclomatic complexity* yang rendah.

BAB IV

HASIL DAN PEMBAHASAN

Pada bab ini berisi penjelasan teknis mengenai implementasi dari *game engine* berdasarkan rancangan pada bab sebelumnya.

4.1 Implementasi *Entity Manager*

Sebelum implementasi *Entity Manager,* dibutuhkan dua elemen dasar, yaitu *Entity* dan *Component*. Berikut implementasi dari *Entity* dan *Component*.

4.1.1 Hasil Implementasi *Entity Class*

```
template <typename T>
T & getComponent()
{
    return std::get<T>(m_components);
}

template <typename T>
const T & getComponent() const
{
    return std::get<T>(m_components);
}

template <typename T>
void removeComponent() const
{
    return getComponent<T>() = T();
}
```

```
template <typename T>
bool hasComponent() const
{
    return getComponent<T>().has;
}

template <typename T, typename... TArgs>
T & addComponent(TArgs&&... mArgs)
{
    auto & component = getComponent<T>();
    component = T(std::forward<TArgs>(mArgs)...);
    component.has = true;
    return component;
}
```

Gambar 4.1 Implementasi dari *Entity Class* dengan fitur *template*

Entity diimplementasikan dengan menggunakan fitur *template* yang disediakan oleh bahasa pemrograman C++ seperti pada gambar 4.1. Fitur *template* ini memungkinkan metode untuk menerima *tipe* data apapun sebagai parameter, sehingga dapat sumber kode yang sama dapat digunakan untuk mengolah berbagai jenis *tipe* data. *Template* ini memungkinkan entitas untuk menerima komponen-

komponen yang memiliki *tipe* data berbeda-beda. Setiap fungsi berguna seperti namanya masing-masing dan semuanya berhubungan dengan *component*, yaitu untuk mendapatkan mendapatkan, menghapus, mengecek, dan menambahkan. Kemudian *keyword template* diatas setiap fungsi merupakan sintaksis untuk mendeklarasikan fungsi yang dapat menerima berbagai jenis *tipe* data.

```
typedef std::tuple<
            CTransform,
            CLifespan,
            CShape,
            CRectangle,
            CCollision,
            CInput,
            CBoundingBox,
            CAnimation,
            CState,
            CGravity,
            CRaycast
        > ComponentTuple;
```

Gambar 4.2 Struktur data *tuple* dari komponen

Entity memiliki properti yang bernama *Components* yang berbentuk struktur data *tuple*. Pada gambar 4.2, *tuple* dideklarasikan dan elemen-elemennya merupakan setiap *tipe* data *component* yang telah dibuat. Kemudian tuple diatas diberi nama *ComponentTuple* dengan menggunakan *keyword typedef*. Struktur data ini digunakan karena komponen merupakan data yang disiapkan/didefinisikan pada *compile time* sehingga cocok jika menggunakan *tuple*. *Entity* juga memiliki properti *tag* dan *id* sebagai pembeda antara entitas dan properti *active* yang digunakan untuk menghapus entitas nantinya. *Entity* juga memiliki beberapa metode yang berkaitan dengan mengakses mengakses komponen yang dimiliki oleh entitas tersebut.

4.1.2 Hasil Implementasi *Component Class*

Component diimplementasi dengan cara mendeklarasikan tiap komponen sebagai *class* tanpa *body*. Jadi, tiap komponen nantinya merupakan sebuah *class* yang hanya berisi properti serta *constructor*. *Component* nantinya akan diinisialisasi pada saat pembuatan entitas, sehingga entitas hanya menginisialisasi komponen-komponen yang digunakan. Pada penelitian ini terdapat dua belas komponen yang berfungsi menyimpan data yang berkaitan dengan tiap sistem.

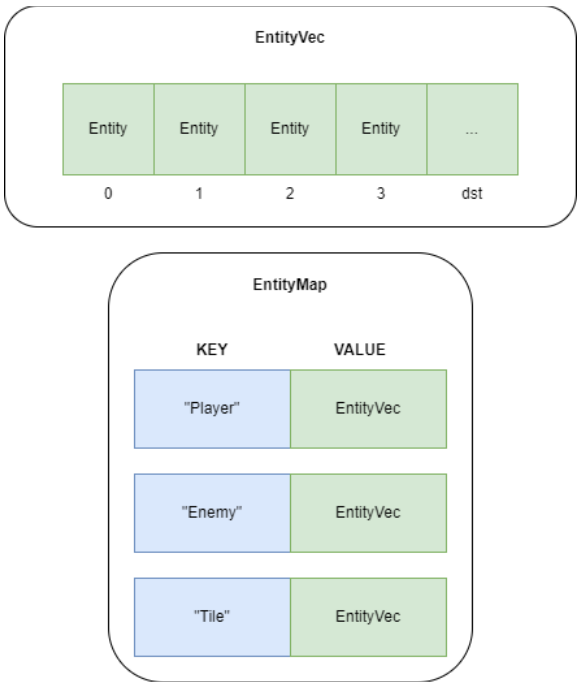
```
class CTransform : public Component
{
public:
    Vec2 pos = {0.0, 0.0};
    Vec2 prevPos = {0.0, 0.0};
    Vec2 scale = {1.0, 1.0};
    Vec2 velocity = {0.0, 0.0};
    float angle = 0;

    CTransform() {}
    CTransform(const Vec2 & p)
        : pos(p)
    {}
    CTransform(const Vec2 & p, const Vec2 & vel, const Vec2 & sc, float a)
        :pos(p), prevPos(p), velocity(vel), scale(sc), angle(a)
    {}
};
```

Gambar 4.3 Implementasi dari *component transform*

Pada gambar 4.3 diatas merupakan salah satu komponen yang diimplementasi pada penelitian ini. *Component* tersebut adalah komponen *Transform*. Transform dideklarasikan sebagai sebuah kelas tanpa implementasi, yang mana berarti hanya mengandung properti-properti dan constructor untuk menginisialisasi properti-properti tersebut. Komponen ini menampung data mengenai posisi (*pos*), posisi sebelumnya (*prevPos*), skala (*scale*), kecepatan (*velocity*), dan sudut (*angle*). Komponen ini salah satunya digunakan pada sistem untuk menggerakkan entitas yang memiliki komponen ini.

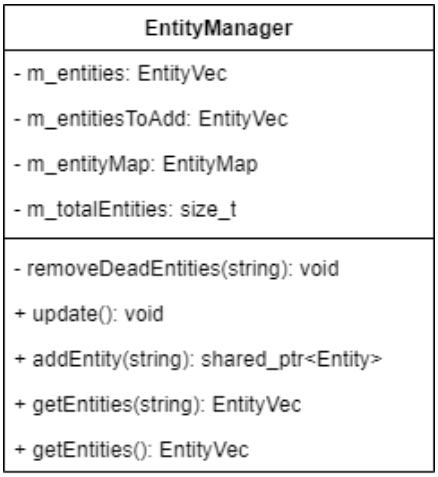
4.1.3 Hasil Implementasi *Entity Manager*

Entity Manager merupakan sistem yang bertugas untuk memproduksi, menyimpan, dan menghapus setiap entitas yang ada pada *game engine*. Pada *Entity Manager* digunakan data struktur *map* yang mana menggunakan *key* (kunci) untuk mengakses *value* (nilai). Struktur data *map* tersebut diberi alias *EntityMap* dan untuk vector/array diberi alias *EntityVec* seperti pada gambar 4.4.



Gambar 4.4 Hasil implementasi dari *Entity Class*

Dengan begini entitas-entitas dapat dibedakan dan diakses sesuai dengan nama atau *tipe* entitas. Data struktur *map* ini memiliki *key* dengan *tipe* data string dan *vector/array* sebagai *value*. *Vector/array* digunakan karena tiap *tipe*/nama entitas dapat memiliki jumlah lebih dari satu.

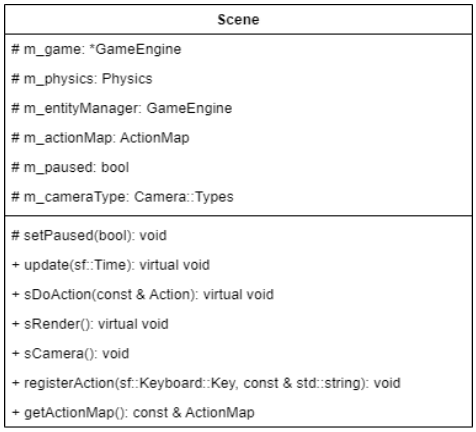


Gambar 4.5 Diagram kelas dari *Entity Manager*

Pada gambar 4.5, salah satu metode *Entity Manager* adalah *addEntity*. Metode tersebut digunakan untuk membuat entitas baru dengan cara *input* nama entitas yang kemudian disimpan ke dalam *m_entitiesToAdd*, lalu pada metode *update*, *entity* yang berada pada *m_entitiesToAdd*, disimpan ke dalam *m_entities* dan *m_entityMap*. Dengan begitu *m_entities* berisi semua entitas sedangkan *m_entitiesMap* berisi entitas-entitas yang dikelompokkan berdasarkan *tag* (nama) entitas tersebut.

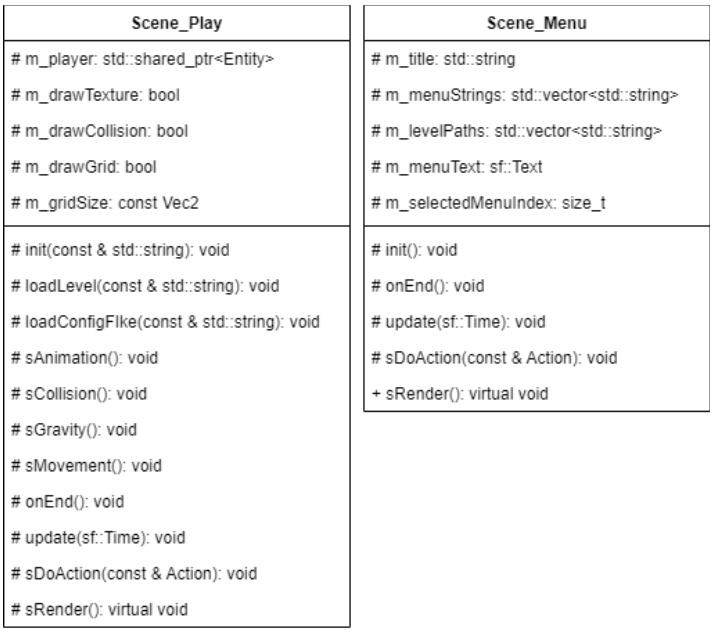
4.2 Hasil Implementasi Sistem *Scene*

Sistem *scene* diimplementasi dengan membuat *abstract class*. *Abstract class* ini nantinya digunakan sebagai kerangka untuk membuat *scene* dengan kegunaan yang lebih spesifik, seperti pembuatan *scene* menu utama dan *scene gameplay*.



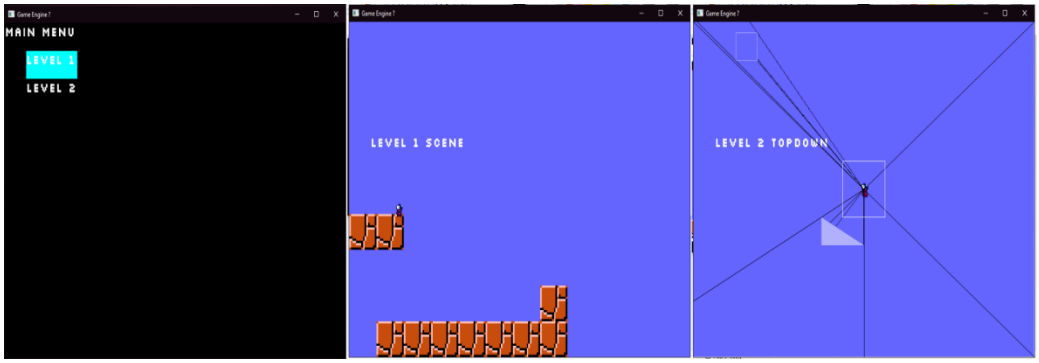
Gambar 4.6 Diagram kelas dari sistem *scene*

Seperti pada gambar 4.6, *scene* di abstraksi berdasarkan metode dasar yang selalu ada pada sebuah *scene*. Metode-metode tersebut adalah *render*, *action/input*, *camera*, dan *update/scene loop*. Selain metode-metode, terdapat properti yang selalu dibutuhkan pada sebuah *scene*. Properti *m_game* adalah sebuah *class* dengan nama *GameEngine* yang mana merupakan tempat semua *scene* akan disimpan, *assets*, *game loop*, sistem *window* dan sistem *rendering*. Untuk detail implementasi *class GameEngine* akan dijelaskan pada sub bab selanjutnya.



Gambar 4.7 Diagram kelas dari *Scene Play* dan *Scene Menu*

Pada gambar 4.7, merupakan *class* diagram dari *scene menu* dan *scene play* yang merupakan turunan dari *abstract class scene*. Metode-metode dari *abstract class*, yaitu *update*, *sDoAction*, dan *sRender* di-*override*. Dengan begitu implementasi dari metode-metode tersebut dapat berbeda, sesuai dengan rancangan dan kegunaan dari kedua *scene* turunan tersebut.



Gambar 4.8 Hasil implementasi *ScenePlay*, *SceneMenu*, dan *SceneTopDown*

Gambar 4.8 merupakan hasil implementasi dari *scene*. Ketiga *scene* tersebut merupakan turunan dari *abstract class* dari implementasi *scene*. *Scene* pertama (sebelah kiri) merupakan *scene menu* untuk memilih *level* atau *scene* lainnya (tengah dan kanan). Ketiganya memiliki metode sistem dan entitas yang berbeda sehingga ketiganya memiliki keunikan tersendiri.

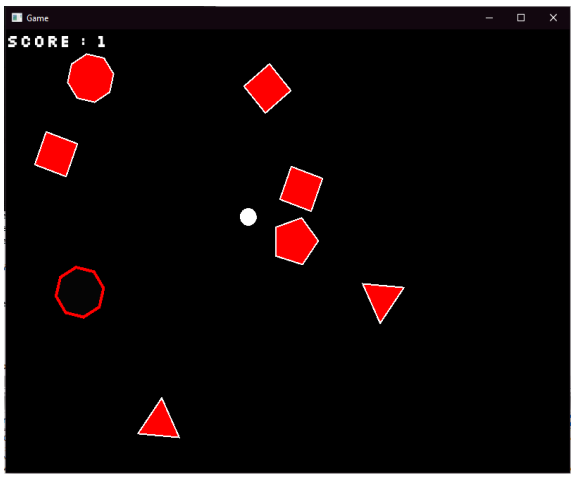
4.3 Hasil Implementasi *Physics Class* (Deteksi Tabrakan)

Pada *physics class* berisi implementasi deteksi tabrakan antara dua lingkaran dan deteksi tabrakan antara dua persegi panjang (*Axis Aligned Bounding Box*). Berikut adalah hasil implementasi keduanya berdasarkan *pseudocode* pada bab sebelumnya.

```
bool Physics::isCircleIntersect(float x1, float y1, float r1, float x2, float y2, float r2)
{
    float sqrDeltaPos = std::pow((x1 - x2), 2) + std::pow((y1 - y2),2);
    float sqrRad = std::pow((r1 + r2), 2);
    return sqrDeltaPos <= sqrRad;
}
```

Gambar 4.9 Implementasi dari algoritma deteksi tabrakan dua lingkaran

Pada gambar 4.9, metode deteksi tabrakan antara dua lingkaran membutuhkan posisi dan radius dari kedua lingkaran yang mana terlihat pada parameter yang dibutuhkan pad metode tersebut. Kemudian dihitung jumlah dari selisih posisi dari kedua lingkaran lalu di akar kuadrat. Nilai ini disebut *sqrDeltaPos*. Setelah itu dicari nilai kuadrat dari jumlah radius kedua lingkaran. Nilai ini disebut *sqrRad*. Bila *sqeDeltaPos* kurang dari sama dengan *sqrRad*, maka tabrakan terjadi.



Gambar 4.10 Hasil dari implementasi deteksi tabrakan dua lingkaran

Pada gambar 4.10 diatas merupakan hasil dari deteksi tabrakan lingkaran. pada *game Geometry Shooter* diatas memiliki tujuan untuk menembaki entitas *Enemy* untuk mendapatkan skor dengan menggunakan entitas *Bullet* dan bila *Player* menyentuh *Enemy*, maka akan dimulai kembali dari awal. Pemicu kedua kejadian tersebut terjadi melalui pengecekan tabrakan antara dua lingkaran.

```
bool Physics::isOverlap(Vec2 overlap)
{
    return (overlap.x > 0.0f && overlap.y > 0.0f)? true : false;
}

Vec2 Physics::getOverlap(std::shared_ptr<Entity> a, std::shared_ptr<Entity> b) const
{

    if (a->hasComponent<CBoundingBox>() && b->hasComponent<CBoundingBox>())
    {
        Vec2 overlap = {0, 0};
        Vec2 & aHalfSize= a->getComponent<CBoundingBox>().halfSize;
        Vec2 & bHalfSize= b->getComponent<CBoundingBox>().halfSize;

        Vec2 & aPos = a->getComponent<CTransform>().pos;
        Vec2 & bPos = b->getComponent<CTransform>().pos;

        float dx = std::abs(aPos.x - bPos.x);
        float dy = std::abs(aPos.y - bPos.y);

        float ox = aHalfSize.x + bHalfSize.x - dx;
        float oy = aHalfSize.y + bHalfSize.y - dy;

        return overlap = Vec2(ox, oy);

    }
```

Gambar 4.11 Hasil implementasi *Axis-Aligned Bounding Box*

Pada gambar 4.11, perhitungan tabrakan antara dua persegi panjang atau *Axis Aligned Bounding Box* (AABB) dihitung menggunakan *component BoundingBox*. Komponen ini berisi data mengenai ukuran (*size*) dan setengah dari ukuran (*half size*). Kemudian dihitung nilai absolut dari selisih posisi x dan posisi y. Lalu, kedua nilai tersebut dikurangi dengan jumlah *half size* dari kedua persegi panjang. Nilai yang dihasilkan dari perhitungan menghasilkan nilai tentang seberapa banyak perpotongan terjadi (*overlap*). Untuk mengecek apakah berpotongan terjadi melalui pengecekan apakah nilai x atau nilai y pada *overlap* lebih dari nol atau tidak.

Gambar 4.12 Penggunaan *Axis-Aligned Bounding Box*

Seperti pada gambar 4.12 diatas kegunaan dari *overlap* pada *game* 2D *side scroller* digunakan sebagai cara agar karakter *player* bisa berdiri diatas tanah dan tidak menembus tembok. Caranya adalah menambah posisi *player* dengan nilai *overlap* sebelum *rendering* terjadi, dengan begitu *player* terlihat berada di atas tanah dan tidak menembus tembok. Hal ini disebut dengan *collision resolution*.

4.4 Hasil Implementasi Manajemen *Resources*/*Assets Class*

Resource atau aset yang digunakan pada *game* akan disimpan pada data struktur *map*. Pada kelas ini, aset dibagi menjadi empat bagian, yaitu *Texture*, *Animation*, *Sound*, dan *Font*. Tiap bagian aset tersebut disimpan pada struktur data tersendiri. Pengaturan mengenai aset apa saja yang akan dimuat pada *game*, diatur melalui sebuah *file* konfigurasi. *File* konfigurasi tersebut berisi *tipe* aset, nama aset, dan alamat *file (filepath)*. Jadi, konfigurasi aset bisa dengan mudah diedit.

Gambar 4.13 Kelas diagram dari *Assets Class*

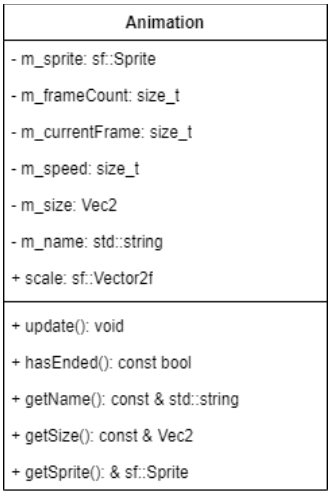
Seperti di UML class pada gambar 4.13, *asset class* memiliki metode *loadFromFile*, yang berguna untuk memuat aset dari *hardware* penyimpanan data ke memori komputer. Untuk pemrosesan yang menangani hal tersebut telah menggunakan metode yang disediakan oleh *library* SFML. Kemudian metode-metode lain berfungsi sesuai dengan *prefix* namanya yaitu *add* (menambah) aset dan *get* (mendapatkan) aset.

4.5 Hasil Implementasi Sistem *Sprite/Animation Class*

Library SFML menyediakan sebuah *namespace* yang bernama *Sprite*. *Sprite* ini memungkinkan sebuah *file* gambar bisa di-*render* hanya pada area tertentu. Dengan begitu rangkaian gambar (*sprite sheet*) mengenai animasi bisa dianimasikan. Hal ini disebut dengan *sprite masking*.

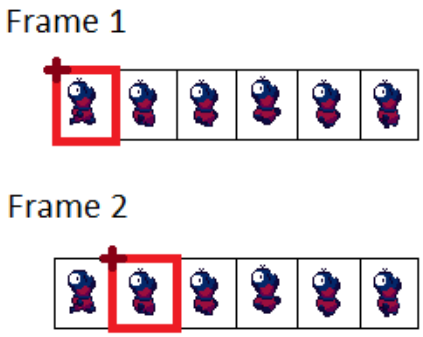
Gambar 4.14 Salah satu *spritesheet* yang digunakan

Pemuatan *sprite sheet* dilakukan melalui *file* konfigurasi pada *assets class* dengan tambahan jumlah *frame* dan kecepatan animasinya.



Gambar 4.15 Kelas diagram dari *Animation class*

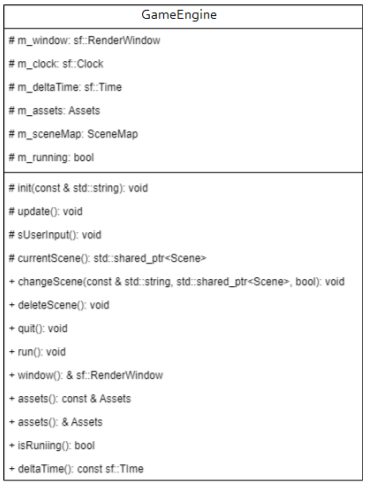
Pada *diagram class* pada gambar 4.15, *animation class* diimplementasi dengan rancangan berorientasi objek, dikarenakan pada *class* ini berisi bukan hanya data, tetapi juga pemrosesan (*System* pada ECS) animasi itu sendiri (pada metode *update*). Hal ini dilakukan karena dibutuhkan perhitungan *frame* tersendiri agar penganimasian dapat dilakukan.



Gambar 4.16 Penggambaran cara kerja sistem animasi

Gambar 4.16 adalah cara kerja dari pemrosesan animasi. Ukuran lebar *sprite sheet* dibagi dengan banyaknya jumlah *frame* (karena rangkaian disusun secara horizontal), dengan begitu posisi x titik pusat (pojok kiri atas) dari kotak *render* dapat ditentukan. Hal ini dilakukan secara terus tiap *frame* sejak animasi dipanggil. Dengan begitu keenam *frame* tersebut akan di-*render* satu persatu tiap dan menghasilkan animasi.

4.6 Hasil Implementasi *Game Engine Class*



Gambar 4.17 Kelas diagram dari *GameEngine* class

Seperti pada gambar kelas diagram 4.17, *GameEngine class* merupakan sebuah *class* yang digunakan untuk menampung metode-metode fundamental dari *game engine*, yaitu *rendering window*, *game loop*, semua *resource*/aset, dan data semua *scene*. Hal ini diperlukan agar tiap *scene* dapat dengan mudah mengakses semua metode-metode tersebut dengan mudah.

```
void GameEngine::update()
{
    m_deltaTime = m_clock.restart();

    sUserInput();

    //loop all the systems here
    currentScene()->update(m_deltaTime);
    currentScene()->sCamera();


    window().display();

    deleteScene();
}
```

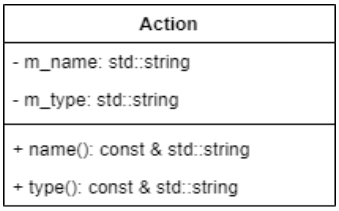
Gambar 4.18 Hasil implementasi *game loop*

Sumber kode pada gambar 4.18 merupakan implementasi dari *game loop*. *DeltaTime* diimplementasi dengan cara menghitung seberapa lama waktu yang

dibutuhkan untuk memproses satu *frame*. Dengan begitu kecepatan proses tiap *frame*-nya akan sama walaupun dijalankan pada perangkat yang memiliki spesifikasi *hardware* CPU yang berbeda. Pada *game loop* tersebut juga dilakukan pemrosesan *input*, kamera dan *update* metode atau *loop* pada *scene* yang sedang berjalan. Lalu untuk *rendering* terjadi pada *window().display()*. Aset visual yang telah didaftarkan di metode *update* pada *scene* ditampilkan. Kemudian pada *deleteScene()*, dilakukan pengecekan, apakah sebuah scene pada *SceneMap* telah berakhir.

4.7 Hasil Implementasi *Input Map*/*Action Class*

Input map diimplementasi dengan cara membungkus *input* dari *hardware* seperti *keyboard* menjadi sebuah *class* (bernama *action*) dan tiap *input* nantinya akan disimpan di dalam sebuah data struktur *map*.



Gambar 4.19 Kelas diagram dari *action class*

Seperti pada gambar 4.19, *action class* memiliki dua properti, yaitu nama dan *tipe*. Nama digunakan untuk melabeli *input* dan *tipe* digunakan untuk menyimpan keadaan dari *input* tersebut, apakah sedang terjadi *input* atau tidak. Dengan implementasi seperti ini, penambahan perangkat *input* baru seperti *joystick* dapat dengan mudah dilakukan karena pembungkusan seperti ini memiliki tingkat konfigurasi yang tinggi, serta penambahan tersebut juga tidak mengganggu *gameplay code*.

4.8 Penggunaan *Game Engine*

Setelah semua sistem dibangun menjadi sebuah *Game Engine*, sebuah *game* dibangun untuk melihat hasil konkrit dari semua sistem tersebut. Dalam konteks ECS, semua sistem pada *game engine* ini merupakan *Entity* dan/atau *Component*. Dengan kata lain merupakan kumpulan data dan cara mempersiapkan atau menginisialisasi data. Belum ada *System* yang memproses data-data tersebut yang digunakan untuk membangun sebuah *game*. Pengimplementasian *System* dilakukan dengan membangun sebuah *game*.

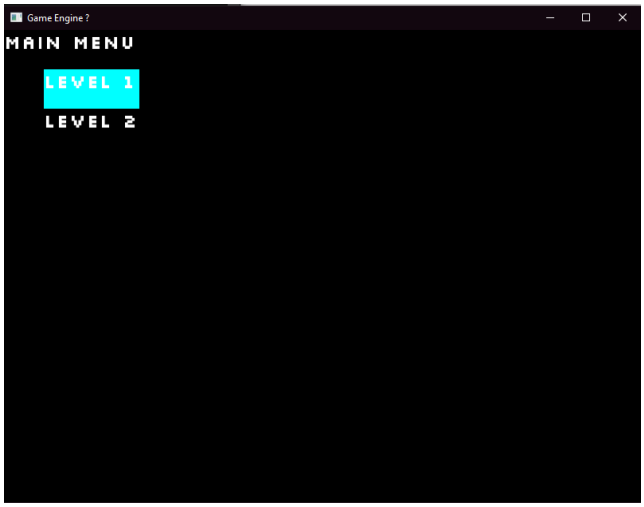
Game yang dibangun dari *game engine* ini terdiri dari tiga *scene* yaitu *SceneMenu* dan *ScenePlay*. Seperti *game* pada umumnya, saat pertama kali *game* dijalankan akan muncul tampilan menu utama (*SceneMenu*), kemudian *player* dapat memilih *level* atau mulai memainkan *game* (*ScenePlay*).

Pada *SceneMenu* terdapat beberapa *System* yang diproses dalam metode tersendiri. Berikut beberapa metode pada *class SceneMenu* beserta penjelasannya.

Tabel 4.1 Metode-metode (*systems*) dan penjelasannya pada *SceneMenu*

Nama Metode/*System*	**Penjelasan**
Scene_Menu::sRender	*Rendering* teks, *background*, dan visual kursor untuk memilih *level* (*ScenePlay* atau *SceneTopDown*)
Scene_Menu::sDoAction	Logika pemprosesan *input* (pindah ke *ScenePlay* atau *SceneTopDown*)

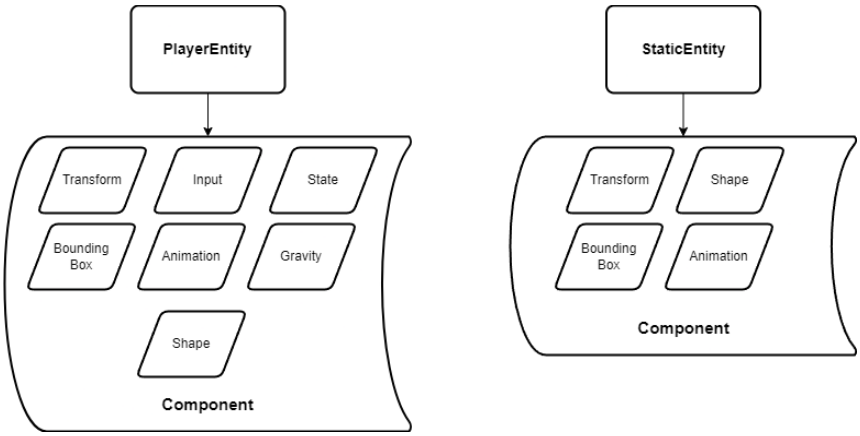
Seperti namanya, *SceneMenu* berguna hanya sebagai *user interface* untuk berpindah-pindah *scene* ketika *game* dimainkan. Berikut hasil dari *SceneMenu*.



Gambar 4.20 Hasil implementasi dari *SceneMenu*

Pada *ScenePlay*, dirancang untuk mendemonstrasikan penggunaan *Animation*/*Sprite*, *Physics* (*Axis-Aligned Bounding Box*), serta penggunaan *Entity* dan *Component*.

Pada *scene* ini, terdapat dua *Entity*, yaitu *PlayerEntity* dan *StaticEntity*. *PlayerEntity* merupakan entitas yang digerakkan dan dapat berinteraksi dengan sekitarnya, sedangkan *StaticEntity* merupakan entitas di sekitarnya seperti tanah dan tembok yang mana disebut sebagai *tiles*.



Gambar 4.21 Komponen-komponen pada entitas *Player* dan *Static*

Pada gambar 4.21, merupakan komponen-komponen yang dimiliki oleh *PlayerEntity* dan *StaticEntity*. Komponen-komponen yang ada pada entitas menjelaskan secara implisit bagaimana perilaku dan kapabilitas dari entitas tersebut. Komponen-komponen ini nantinya akan akan dilakukan pemrosesan pada *System* yang membutuhkan untuk menghasilkan perilaku atau kemampuan tertentu.

Tabel 4.2 Metode-metode (*systems*) dan penjelasannya pada *ScenePlay*

Nama Metode/System	**Penjelasan**	**Komponen Yang Dipakai**	**Sistem *Game Engine* Yang Dipakai**
Scene_Play::sRender	*Rendering* teks, *background*, dan *sprite texture* dari setiap entitas	*Animation*	*Animation/Sprite System*
Scene_Play::sDoAction	Logika pemrosesan *input* dari *PlayerEntity*	*Input*	*Action/Input Map*
Scene_Play::sCollision	Pemrosesan deteksi tabrakan *Axis-Aligned Bounding Box* dan *collision resolution.*	*Transform, Bounding Box, Gravity*	*Physics/Collision*
Scene_Play::sMovement	**Pemrosesan *movement* dari PlayerEntity.**	***Transform, Input, State***	***Action/Input Map***
Scene_Play::sAnimation	Pemilihan animasi yang dijalankan	*State, Animation*	*Animation/Sprite System*

Pada tabel 4.2 merupakan semua *System* yang ada pada *ScenePlay* beserta komponen dan sistem *game engine* apa saja yang digunakan. Pada ScenePlay, terdapat berbagai System yang merupakan logika *game*play yang mana menggunakan Entity dan Component untuk diproses. Contohnya pada metode *Scene_Play::sMovement* (baris yang ditebalkan pada tabel 4.2), terdapat logika untuk agar *game* objek *PlayerEntity* dapat bergerak yang mana memproses data

posisi dan data velocity dari komponen *Transform* dan data *input map* dari komponen *Input* serta data *state* dari komponen *State* dari entitas yang bernama *PlayerEntity*. Berikut hasil dari *ScenePlay*.

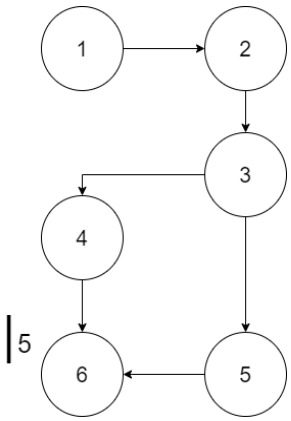
Gambar 4.22 Hasil implementasi dari *Scene Play*

Pada gambar 4.22 terlihat bahwa sistem *rendering* dan penggunaan *texture* berhasil dilakukan. Lalu pada Tabel 4.2 menunjukkan perilaku dan kapabilitas yang dihasilkan, yaitu menggerakkan *PlayerEntity* setelah *input* terjadi dan menghasilkan animasi yang berbeda tergantung gerakkan yang dilakukan dan dapat berdiri diatas tanah dan tidak menembus tembok. Perilaku dan kapabilitas tersebut merupakan hasil dari pemrosesan data yang telah disiapkan dengan menggunakan sistem-sistem pada *game engine* yang telah diimplementasi pada menjadi beberapa *class* tersendiri.

4.9 Hasil Pengujian dengan Perhitungan *Cyclomatic Complexity*

Nilai *cyclomatic complexity* didapat dengan menghitung jumlah panah (*edge*), jumlah simpul (*node*), dan jumlah satu kesatuan simpul (*segment*) pada *control flow diagram* (CFG) pada tiap bagian kode.

```
void Assets::addTexture(const std::string & name, const std::string & path)
{
  //auto texture = std::shared_ptr<sf::Texture>(new sf::Texture());
  sf::Texture texture; | 1

  if(!texture.loadFromFile(path))  | 2
  {
    std::cout << "Load texture failed !" << "\n";    | 3
  }
  else  | 4
  {
    m_texturesMap.insert(std::pair<std::string, sf::Texture>(name, texture));
    std::cout << "Texture loaded" << "\n";
  }

}  | 6
```



Gambar 4.23 *Control flow graph* (CFG) pada metode *Assets::addTexture*

Pada gambar 4.23 merupakan *control flow graph* (CFG) dari metode *addTexture* pada *class Assets*. Terlihat pada CFG tersebut bahwa terdapat enam jumlah *edge*, enam jumlah *node*, dan satu jumlah *segment*. Dengan menggunakan rumus *cyclomatic complexity* (nomor rumus 3.5), didapatkan nilai *cyclomatic complexity* sebesar dua.

Perhitungan *cyclomatic complexity* yang dilakukan diatas dapat diotomatisasi dengan menggunakan *package* penganalisis berbasis bahasa pemrograman *Python* yang bernama *Lizard*. Penganalisis ini menghitung nilai *cyclomatic complexity*, *line of code* (LOC) pada bagian kode tersebut, jumlah token, dan jumlah parameter. Penganalisis ini juga mampu mendeteksi kode duplikasi dan mengabaikan *file header* sehingga kompleksitas penulisan kode dapat dinilai dengan efektif. Terlebih lagi penganalisis ini telah digunakan untuk mendeteksi masalah potensial pada kode yang dapat mempengaruhi kualitas kode dan bug pada projek aplikasi bernama ATLAS yang tediri dari 3.8 juta baris kode C++ dan 1.4 juta baris kode kode Python (S Martin-Haugh et al, 2017).

Berdasarkan Kaner dan Bond (2004), LOC merupakan metrik tidak langsung yang digunakan dalam pengukuran ukuran sistem perangkat lunak, kompleksitas, produktivitas pengembang, serta upaya dan biaya pengembangan. Maka dari itu dapat disimpulkan bahwa semakin tinggi nilai jumlah baris kode (LOC) beserta jumlah token dan jumlah parameter yang merupakan bagian dari LOC maka tinggi kompleksitas serta upaya dan biaya pengembangan. Oleh karena itu, ketiga indikator tersebut perlu dipertimbangkan dalam menilai tingkat keakuratan nilai *cyclomatic complexity*.

4.9.1 Pembahasan Pengujian

Berikut adalah hasil otomasi perhitungan nilai *cylcomatic complexity* pada basis kode *class Assets*.

Tabel 4.3 Hasil otomasi analisis kelas *Assets*

No.	Nama Metode/*Function*	Jumlah Baris Kode	Jumlah Token	Jumlah Parameter	Nilai *Cyclomatic Complexity*
1.	Assets::Assets	1	7	0	1
2.	Assets::loadFrom*File*	34	241	1	6
3.	Assets::addTexture	13	83	2	2
4.	Assets::addAnimation	4	39	2	1
5.	Assets::addSound	16	118	2	2
6.	Assets::addFont	13	83	2	2
7.	Assets::getTexture	4	19	1	1
8.	Assets::getAnimation	4	19	1	1
9.	Assets::getSound	4	19	1	1
10.	Assets::getFont	4	19	1	1

Pada tabel 4.3, merupakan hasil analisis pada *class Assets/Resource*. Pada tabel tersebut terlihat nilai *cyclomatic complexity* setiap metode memiliki nilai dari rentang 1 - 6. Jika dikaji pada metrik yang diajukan oleh Kalagara, maka semua

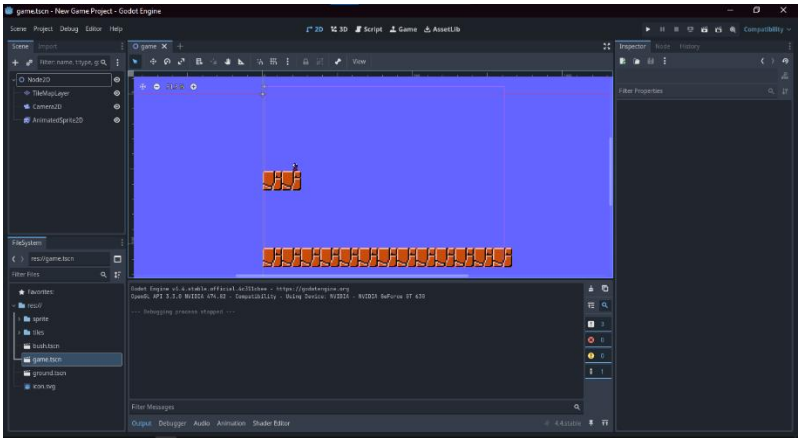
kode metode pada class ini terstruktur, ditulis dengan baik, mudah dites, dan rendah *effort*.

Terlihat pada metode *Assets::loadFromFile* (baris yang ditebalkan pada tabel 4.3), memiliki baris kode dan jumlah token tertinggi. Baris kode dan jumlah token menunjukkan seberapa banyak kode yang ditulis pada bagian metode tersebut. Hal ini merujuk pada tingkat *readability*. Jika semakin banyak kedua faktor tersebut, maka semakin lama dan susah kode untuk dipahami. Namun dengan rendahnya nilai *cyclomatic complexity* pada metode tersebut, maka dapat dikatakan bahwa, walaupun memiliki banyak kode yang tertulis, metode tersebut dapat dengan mudah dipahami dan tidak membutuhkan waktu yang lama.

Dengan begitu tiap metode dilabeli dengan tingkat kompleksitas berdasarkan metrik yang diajukan oleh Kalagara (Tabel 3.1). Nilai dari rentang 1 - 10 dilabeli rendah, nilai dari rentang 11 - 20 dilabeli sedang, nilai dari rentang 21 - 40 dilabeli tinggi, dan nilai diatas 40 dilabeli sangat tinggi.

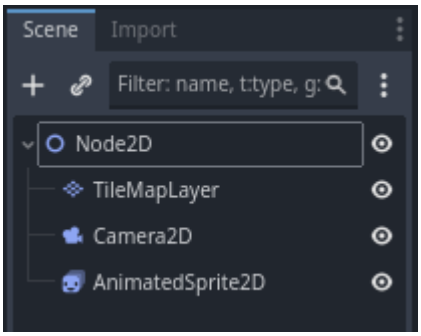
4.10 Perbandingan dengan Godot Engine

Hasil implementasi dicoba dibandingkan dengan *game engine* yang dapat diakses secara publik dan bebas dipakai untuk tujuan apapun, yaitu Godot Engine. Tujuannya adalah untuk mengetahui melihat perbedaan antar keduanya. Dikarenakan *game engine* adalah perangkat lunak yang kompleks, maka yang dikaji hanyalah satu kasus penggunaan yang dapat merepresentasikan perbedaan antara hasil implementasi *game engine* pada penelitian ini dengan Godot Engine. Kasus penggunaan yang dipakai adalah menyiapkan aset *game* yang ditampilkan melalui *scene* lalu ditampilkan di *window screen*.



Gambar 4.24 *Game Editor* pada *Godot Engine*

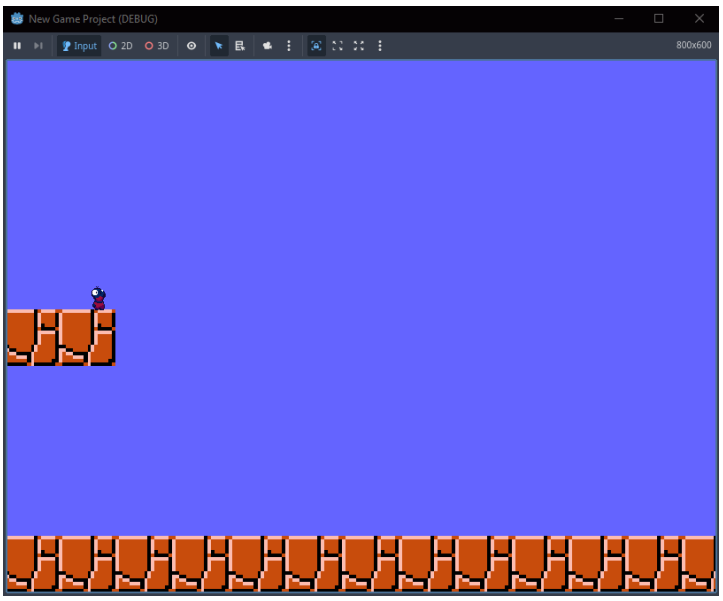
Pada Godot Engine, untuk menyiapkan aset *game* dan ditampilkan pada *scene* dilakukan melalui *game editor* yang terlihat seperti pada gambar 4.24. Oleh karena itu untuk kasus ini, tidak perlu dilakukan penulisan kode sama sekali. Semuanya dapat dilakukan melalui *game editor*.



Gambar 4.25 Susunan *Node* pada *scene* di *game editor Godot Engine*

Untuk membuat *scene* yang serupa seperti pada gambar 4.22, diperlukan cara untuk memuat aset gambar dari perangkat penyimpanan, memposisikannya sesuai dengan keinginan, dan menampilkannya pada *window screen*. Hal ini dilakukan melalui *building blocks* yang disediakan Godot Engine yang disebut *Node*. Susunan *Node* yang dibutuhkan terlihat pada gambar 4.25. Kegunaan masing-masingnya bisa dilihat dari namanya. *Node TileMapLayer* digunakan untuk

menyusun tile ground, *Node Camera2D* sebagai kamera untuk merender gambarnya nanti, dan *Node AnimatedSprite2D* untuk menganimasikan aset *spritesheet*.



Gambar 4.26 Hasil kasus penggunaan pada *Godot Engine*

Lalu jika *game* dijalankan, susunan *Node* akan menampilkan seperti pada gambar 4.26. Terlihat bahwa hasilnya sangat mirip dengan hasil implementasi pada gambar 4.22. Dari sini terlihat kalau sistem memuat gambar dari perangkat penyimpanan, memposisikan *game* objek, dan menampilkannya pada *window screen* telah diabstraksi oleh Godot Engine itu sendiri melalui *editor* atau grafis antarmuka, sehingga pengguna tidak perlu lagi membuat mengimplementasikan sistem-sistem tersebut sendiri.

```
Texture idlePlayer sprite/idle.png
Texture walkPlayer sprite/walk.png
Texture airPlayer sprite/air.png
Texture ground tiles/ground.png
Texture bush tiles/bush.png
Font fontName font/Silkscreen-Regular.ttf
Animation idlePlayer idlePlayer 5 10
Animation walkPlayer walkPlayer 6 10
Animation airPlayer airPlayer 1 6
```

Gambar 4.27 File konfigurasi asset

Berbeda dengan *game engine* pada penelitian ini. Semua tahap dilakukan melalui penulisan kode. Aset disiapkan melalui sebuah *file* konfigurasi mengenai data-data dari aset yang akan dimuat, seperti *tipe* aset, nama aset, alamat aset (*file path*), dan jumlah *frame* serta kecepatan *frame* (khusus aset animasi) seperti pada gambar 4.27. Lalu, perlu menyiapkan *Entity* dan *Component* serta logika untuk memproses data-data tersebut (*System*) untuk dapat menghasilkan gambar seperti pada gambar 4.22.

Berdasarkan perbandingan ini dapat disimpulkan bahwa *game engine* pada penelitian ini sudah mampu melakukan hal yang sama seperti yang dilakukan oleh *game engine* publik yaitu Godot Engine. Meskipun terlihat alur kerja dari Godot Engine jauh lebih mudah dikarenakan adanya *game editor* (grafis) yang sangat memudahkan pengguna dalam mengoperasikan aplikasi dibandingkan *game engine* pada penelitian ini yang cara pengoperasiannya masih melalui penulisan kode (tekstual).

4.11 Integrasi Islam

Islam sangat menekankan pentingnya penggunaan akal dan ilmu. Seperti dalam Al-Quran pada surat Al-'Alaq yang mana juga merupakan wahyu pertama yang diberikan Allah kepada Nabi Muhammad SAW yang berisi mengenai pentingnya ilmu pengetahuan dan pembelajaran. Terutama pada surat Al-'Alaq:1-5 yang berbunyi :

اِقْرَأْ بِاسْمِ رَبِّكَ الَّذِيْ خَلَقَۚ (١) خَلَقَ الْاِنْسَانَ مِنْ عَلَقٍۚ (٢) اِقْرَأْ وَرَبُّكَ الْاَكْرَمُۙ (٣) الَّذِيْ عَلَّمَ بِالْقَلَمِۙ (٤) عَلَّمَ الْاِنْسَانَ مَا لَمْ يَعْلَمْۗ (٥)

Artinya: "*Bacalah dengan (menyebut) nama Tuhanmu Yang menciptakan, Dia telah menciptakan manusia dari segumpal darah. Bacalah, dan Tuhanmulah Yang Maha Pemurah, Yang mengajar (manusia) dengan perantaraan qalam. Dia mengajarkan kepada manusia apa yang tidak diketahuinya*"

Dikutip dari Mukmin, Surat Al-'Alaq ayat 1-5 menerangkan bahwa Allah SWT menciptakan manusia dari benda yang hina dan memuliakannya dengan mengajar membaca, menulis, dan memberinya pengetahuan (Mukmin, 2016). Dengan begitu, teknologi yang merupakan hasil dari pembelajaran dan pengetahuan adalah upaya manusia untuk terlihat mulia dihadapan-Nya.

Ditambah lagi pada Al-Quran surat Al-Mulk ayat ke-15 yang berbunyi :

هُوَ الَّذِيْ جَعَلَ لَكُمُ الْاَرْضَ ذَلُوْلًا فَامْشُوْا فِيْ مَنَاكِبِهَا وَكُلُوْا مِنْ رِّزْقِهٖۗ وَاِلَيْهِ النُّشُوْرُ ﴿١٥﴾

Artinya: "*Dialah yang menjadikan bumi untuk kamu dalam keadaan mudah dimanfaatkan. Maka, jelajahilah segala penjurunya dan makanlah sebagian dari rezeki-Nya. Hanya kepada-Nya kamu (kembali setelah) dibangkitkan*" (QS. Al-Mulk:15).

Menurut Quraish Shihab, ayat ini menegaskan kekuasaan Allah sekaligus kelemahlembutan-Nya dalam pengaturan makhluk termasuk manusia, agar mereka mensyukuri nikmat-Nya. Allah-lah yang menjadikan bumi ini nyaman untuk hidup dan dihuni, sehingga mudah sekali untuk melakukan aktivitas, baik berjalan, bertani, berniaga, dan silahkan kapan saja kamu mau, berjalanlah di penjuru-penjurunya bahkan pegunungan-pegunungannya dan makanlah sebagian dari rezeki-Nya melimpah melebihi kebutuhan kamu dan mengabdilah kepada-Nya sebagai tanda syukur atas limpahan rezekiNya, dan hanya kepada-Nya kamu masing-masing dibangkitkan untuk mempertanggung jawabkan amalan-amalanmu (Shihab, 2001). Tafsir surat Al-Mulk ayat 15 tersebut dapat disimpulkan bahwa Allah telah memberikan kemudahan bagi manusia untuk beraktifitas dalam rangka memenuhi segala kebutuhan hidup. Allah telah melapangkan bumi ini dan

menyediakan berbagai macam fasilitas yang bisa dimanfaatkan manusia untuk berkreasi dan bermanfaat bagi sesamanya. Untuk itu *game engine* yang merupakan teknologi hasil dari ilmu pengetahuan dibuat untuk memudahkan dalam membuat *game* yang tak hanya berguna untuk peniliti sendiri tetapi juga untuk orang lain. Hal ini bisa dicapai karena Allah telah memberikan kemudahan bagi manusia untuk belajar dengan memanfaatkan semua fasilitas sebaik mungkin.

BAB V

KESIMPULAN DAN SARAN

5.1 Kesimpulan

Berdasarkan hasil dari implementasi arsitektur *Entity Component System* (ECS) dalam pengembangan *game engine* dua dimensi didapatkan beberapa poin utama, yaitu:

a. *Game engine* yang dibangun dengan arsitektur ECS merupakan kumpulan dari berbagai macam sistem yang memudahkan dalam pembuatan *game*. Sistem-sistem tersebut adalah *entity manager*, sistem *scene*, *physics*, sistem manajemen *resource*/aset, sistem *sprite*/animasi, dan pemetaan *input*.
b. Arsitektur ECS digunakan dalam mengorganisir sumber kode serta menjadi paradigma dan pendekatan berpikir dalam menyelesaikan masalah.
c. Pengujian dilakukan dengan menggunakan metrik pengukuran *Cyclomatic Complexity* untuk menilai seberapa bagus tingkat penulisan kode. Berdasarkan hasil pengujian dengan *Cyclomatic Complexity*, didapatkan bahwa hampir semua bagian kode memiliki nilai *Cyclomatic Complexity* yang rendah. Dengan begitu penggunaan arsitektur *Entity Component System* memiliki tingkat keterbacaan dan pemeliharaan yang rendah serta dapat dengan mudah dimanipulasi dan dipahami oleh pengembang.

Dari beberapa poin tersebut, dapat disimpulkan bahwa penggunaan arsitektur *Entity Component System* (ECS) dalam pengembangan 2D *game engine* memberikan kemudahan dalam mengorganisir dan mengimplementasi sistem-

sistem pada *game engine*. Hal ini tervalidasi dari pengujian basis kode menggunakan metode *Cyclomatic Complexity* yang mana menunjukkan rendahnya tingkat keterbacaan dan pemeliharaan tiap bagian kode. Hal tersebut juga secara langsung menunjukkan mudahnya bagian kode untuk dimanipulasi dan dipahami oleh pengembang.

5.2 Saran

Dalam penelitian ini, Penulis memahami beberapa kekurangan dalam pengembangan *game engine* dua dimensi dengan menggunakan arsitektur *entity component system* (ECS) yang perlu untuk dipertimbangkan untuk mencapai hasil yang lebih baik dan maksimal. Kekurangan-kekurangan tersebut antara lain:

a. Dari segi performa, *game engine* ini dapat ditingkatkan bila lebih menerapkan perancangan berorientasi data (*data oriented design*) karena adanya masalah *cache miss* yang terjadi bila menggunakan heap memory dibanding stack memory (contohnya mengakses data menggunakan pointer).
b. Pada penelitian ini, *game engine masih* belum memiliki *graphical user interface* (GUI) yang dapat memudahkan dalam penggunaan *game engine* itu sendiri. Hal ini dapat dikembangkan pada peneliti selanjutnya untuk menambahkan GUI pada *game engine*, sehingga dapat dengan mudah dipakai oleh orang lain.
c. *Game engine* yang diimplementasi belum mendukung pengembangan tiga dimensi *game* yang mana sangat dapat diperluas dengan implementasi *game engine* saat ini.

d. Penggunaan bahasa pemrograman juga dapat ditingkatkan dengan menggunakan bahasa pemrograman modern yang memang berkonsep berorientasi data seperti *Odin*, *Zig*, dan *Jai*. Kelemahan C++ yang mana digunakan pada penelitian ini adalah sintaksis-nya yang susah dibaca, tingkat ergonomis yang rendah, pesan kesalahan (*error message*) yang diberikan oleh *compiler* sering tidak membantu, dan waktu kompilasi yang lama. Bahasa pemrograman modern telah menyelesaikan semua masalah ini, sehingga pengembangan dapat dilakukan dengan lebih cepat dan lebih mudah untuk dipelihara nantinya dibanding menggunakan bahasa C++.

DAFTAR PUSTAKA

Ahmad, F. N. (2013). An Overview Study of *Game* Engines. D Int. Journal of Engineering Research and Applications, Vol. 3(5), 1673–1693.

Al-Ekram, R., & Kontogiannis, K. (2004). Source Code Modularization Using Lattice of Concept Slices. *Proceedings of the European Conference on Software Maintenance and Reengineering, CSMR*, *8*, 195–203.

Bahnassi, W. (2013). *Game* Engine Design. GPU Pro, (June), 265.

Bishop, L., Eberly, D., Whitted, T., Finch, M., & Shantz, M. (1998). Designing a PC *game engine*. IEEE Computer Graphics and Applications, 18(1), 46–53.

Capdevila, B. (2013). Serious Game Architecture and Design: Modular Component-based Data-driven Entity System Framework to Support Systemic Modeling and Design in Agile Serious Game Developments. Université Pierre et Marie Curie, Paris, France.

Cheah, T. C. S., & Ng, K. (2005). A Practical Implementation of a 3-D *Game* Engine Faculty of Information Technology , Multimedia University. Proceedings of the Computer Graphics, Imaging and Vision: New Trends (CGIV'05), 8.

ChimiChanga. (2024, 14 Juli). *Andrew Kelley Practical Data Oriented Design (DoD).* Youtube.

CppCon. (2014, 30 September). *CppCon 2014: Mike Acton "Data-Oriented Design and C++".* Youtube.

Dave Churchill. (2023, 17 November). *COMP4300 - Game Programming - Lecture 19 - Caching + ECS Memory Poo*. Youtube.

Fabian, Richard. (2018). Data-oriented Design: Software Engineering for Limited Resources and Short Schedules. Richard Fabian.

Furness, P. N., & Lam, E. W. H. (1987). Serious *Game* Architecture and Design. Journal of Clinical Pathology. Packt.

Gregory, J. (2000). *Game* Engine Architecture 3rd Edition.CRC Press.

Guana, V., Stroulia, E., & Nguyen, V. (2015). Building a *Game* Engine: A Tale of Modern Model-Driven Engineering. Proceedings - 4th International Workshop on *Game*s and Software Engineering, GAS 2015, 15–21.

Haller, J., Hansson, H. V., Moreira, A. (2013). SFML *Game* Development. Packt.

Härkonen, T. (2019). Advantages and implementation of Entity-Component-Systems, (April), 1–25.

Kalagara, S. (2020). Cyclomatic Complexity in Software Development. International Journal of Engineering Research & Technology. Volume 8, Issue 16.

Lee, E. S., & Shin, B. S. (2021). A Flexible Input Mapping System for Next-generation Virtual Reality Controllers. Electronics (Switzerland), 10(17), 1–11.

Maggiorini, D., Ripamonti, L. A., Zanon, E., Bujari, A., & Palazzi, C. E. (2016). SMASH: A Distributed Game Engine Architecture. Proceedings - IEEE Symposium on Computers and Communications, 2016-August, 196–201.

Martin-Haugh, S., Kluth, S., Seuster, R., Snyder, S., Obreshkov, E., Roe, S., Sherwood, P., & Stewart, G. A. (2017). C++ software quality in the ATLAS experiment: Tools and experience. *Journal of Physics: Conference Series*, *898*(7). https://doi.org/10.1088/1742-6596/898/7/072011

Mukmin, Taufik. 2016. *Urgensi Belajar Dalam Perspektif Al-Qur'an Surat Al-Alaq Ayat 1-5 Menurut Tafsir Ibni Katsir.* Sumatera Selatan: El-Ghiroh, Sekolah Tinggi Agama Islam Bumi Silampari

Müller, M. (2014). Physics in *Game*s. Graphics Interface Keynote, (November 2018), 0–13.

Odeh, A. H., Odeh, M., Odeh, H., & Odeh, N. (2024). Measuring Cyclomatic Complexity of Source Code Using Machine Learning. *Revue d'Intelligence Artificielle*, *38*(1), 183–191. https://doi.org/10.18280/ria.380118

Park, H., & Baek, N. (2020). Developing an open-source lightweight *game engine* with DNN support. Electronics (Switzerland), 9(9), 1–15.

Pupius, R. (2017). Mastering SFML *Game* development: create complex and visually stunning *game*s using all the advanced features available in SFML development. Packt.

Shihab, Q. M. (2001). Tafsir al-Misbah. Lentera Hati. Vol.15, 423.

LAMPIRAN-LAMPIRAN

Lampiran 1

Lampiran hasil pengujian *Cyclomatic Complexity* :

Nama Metode	**Jumlah Token**	**Jumlah Parameter**	**Jumlah baris kode**	**Nilai *Cyclomatic Complexity***	**Kompleksitas**
Action::Action	7	0	1	1	Rendah
Action::Action	30	2	4	1	Rendah
Action::name	11	0	4	1	Rendah
Action::type	11	0	4	1	Rendah
Action::toString	8	0	3	1	Rendah
Animation::Animation	7	0	1	1	Rendah
Animation::Animation	31	2	4	1	Rendah
Animation::Animation	132	4	11	1	Rendah
Animation::update	51	0	7	1	Rendah
Animation::hasEnded	11	0	4	1	Rendah
Animation::getName	10	0	4	1	Rendah
Animation::getSize	10	0	4	1	Rendah
Animation::getSprite	10	0	4	1	Rendah
Assets::Assets	7	0	1	1	Rendah
Assets::loadFrom*File*	241	1	34	6	Rendah
Assets::addTexture	83	2	13	2	Rendah
Assets::addAnimation	39	2	4	1	Rendah
Assets::addSound	118	2	16	2	Rendah
Assets::addFont	83	2	13	2	Rendah
Assets::getTexture	19	1	4	1	Rendah
Assets::getAnimation	19	1	4	1	Rendah
Assets::getSound	19	1	4	1	Rendah
Assets::getFont	19	1	4	1	Rendah

Nama Metode	**Jumlah Token**	**Jumlah Parameter**	**Jumlah baris kode**	**Nilai *Cyclomatic Complexity***	**Kompleksitas**
CTransform::CTransform	5	0	1	1	Rendah
CTransform::CTransform	14	1	3	1	Rendah
CTransform::CTransform	47	4	3	1	Rendah
CRaycast::CRaycast	5	0	1	1	Rendah
CRaycast::CRaycast	24	2	3	1	Rendah
CShape::CShape	5	0	1	1	Rendah
CShape::CShape	64	5	8	1	Rendah
CRectangle::CRectangle	5	0	1	1	Rendah
CRectangle::CRectangle	80	1	8	1	Rendah
CCollision::CCollision	5	0	1	1	Rendah
CCollision::CCollision	12	1	3	1	Rendah
CScore::CScore	5	0	1	1	Rendah
CScore::CScore	12	1	3	1	Rendah
CBoundingBox::CBoundingBox	5	0	1	1	Rendah
CBoundingBox::CBoundingBox	29	1	3	1	Rendah
CGravity::CGravity	5	0	1	1	Rendah
CGravity::CGravity	12	1	1	1	Rendah
CLifespan::CLifespan	5	0	1	1	Rendah
CLifespan::CLifespan	17	1	3	1	Rendah
CInput::CInput	5	0	1	1	Rendah
CAnimation::CAnimation	5	0	1	1	Rendah
CAnimation::CAnimation	22	2	1	1	Rendah
CState::CState	5	0	1	1	Rendah
CState::CState	16	1	1	1	Rendah

Nama Metode	**Jumlah Token**	**Jumlah Parameter**	**Jumlah baris kode**	**Nilai *Cyclomatic Complexity***	**Kompleksitas**
Entity::Entity	28	2	4	1	Rendah
Entity::isActive	11	0	4	1	Rendah
Entity::id	11	0	4	1	Rendah
Entity::tag	11	0	4	1	Rendah
Entity::destroy	11	0	4	1	Rendah
Entity::hasComponent	16	0	4	1	Rendah
Entity::addComponent	45	1	7	1	Rendah
Entity::getComponent	16	0	4	1	Rendah
Entity::getComponent	17	0	4	1	Rendah
Entity::removeComponent	18	0	4	1	Rendah
EntityManager::EntityManager	85	0	11	1	Rendah
EntityManager::update	69	0	14	3	Rendah
EntityManager::removeDeadEntities	72	1	13	2	Rendah
EntityManager::addEntity	43	1	6	1	Rendah
EntityManager::getEntities	10	0	4	1	Rendah
EntityManager::getEntities	19	1	4	1	Rendah
loadConfig*File*	319	1	38	7	Rendah
Game::*Game*	11	0	4	1	Rendah
Game::registerAction	43	2	4	1	Rendah
Game::getActionMap	11	0	4	1	Rendah
Game::doAction	235	1	41	11	Rendah
Game::init	205	0	21	1	Rendah
Game::setPaused	13	1	4	1	Rendah
Game::run	57	0	18	3	Rendah

Nama Metode	**Jumlah Token**	**Jumlah Parameter**	**Jumlah baris kode**	**Nilai *Cyclomatic Complexity***	**Kompleksitas**
Game::sRender	200	0	20	3	Rendah
Game::sUserInput	208	0	26	9	Rendah
Game::sMovement	557	0	41	12	Rendah
Game::spawnPlayer	174	0	15	1	Rendah
Game::spawnBullet	150	2	13	1	Rendah
Game::spawnEnemy	181	0	15	1	Rendah
Game::spawnSmallEnemies	183	3	12	1	Rendah
Game::sLifespan	127	0	25	7	Rendah
Game::sEnemySpawner	44	0	10	3	Rendah
Game::sCollision	31	0	7	2	Rendah
Game::sAnimation	104	0	17	5	Rendah
*Game*Engine::*Game*Engine	18	1	4	1	Rendah
*Game*Engine::init	58	1	7	1	Rendah
*Game*Engine::currentScene	13	0	4	1	Rendah
*Game*Engine::isRunning	16	0	4	2	Rendah
*Game*Engine::window	10	0	4	1	Rendah
*Game*Engine::assets	11	0	4	1	Rendah
*Game*Engine::assets	10	0	4	1	Rendah
*Game*Engine::deltaTime	10	0	4	1	Rendah
*Game*Engine::run	19	0	7	2	Rendah
*Game*Engine::quit	13	0	4	1	Rendah
*Game*Engine::sUserInput	148	0	17	7	Rendah
*Game*Engine::changeScene	75	3	11	3	Rendah
*Game*Engine::deleteScene	38	0	10	3	Rendah
*Game*Engine::update	48	0	9	1	Rendah

Nama Metode	**Jumlah Token**	**Jumlah Parameter**	**Jumlah baris kode**	**Nilai *Cyclomatic Complexity***	**Kompleksitas**
main	32	0	6	1	Rendah
Physics::Physics	7	0	1	1	Rendah
Physics::isCircleIntersect	74	6	6	1	Rendah
Physics::isPointInCircle	57	3	5	4	Rendah
Physics::isOverlap	32	1	4	3	Rendah
Physics::getOverlap	182	2	16	3	Rendah
Physics::getPreviousOverlap	181	2	16	3	Rendah
Physics::getOverlapDirection	191	2	27	7	Rendah
Physics::lineIntersection	175	4	24	5	Rendah
Physics::approach	52	3	13	3	Rendah
Scene::Scene	7	0	1	1	Rendah
Scene::Scene	15	1	4	1	Rendah
Scene::registerAction	45	2	4	1	Rendah
Scene::sCamera	538	0	53	10	Rendah
Scene::windowWidth	20	0	4	1	Rendah
Scene::windowHeight	20	0	4	1	Rendah
Scene::getActionMap	11	0	4	1	Rendah
Scene::hasEnded	11	0	4	1	Rendah
Scene::drawLine	78	2	9	1	Rendah
Scene::createVertex	63	2	9	2	Rendah
Scene::createRaycast	90	3	9	1	Rendah
Scene_Menu::Scene_Menu	19	1	5	1	Rendah
Scene_Menu::init	141	0	18	1	Rendah
Scene_Menu::update	18	1	5	1	Rendah
Scene_Menu::sRender	225	0	23	2	Rendah

Nama Metode	**Jumlah Token**	**Jumlah Parameter**	**Jumlah baris kode**	**Nilai *Cyclomatic Complexity***	**Kompleksitas**
Scene_Menu::sDoAction	177	1	30	9	Rendah
Scene_Menu::onEnd	13	0	4	1	Rendah
Scene_Play::Scene_Play	32	2	6	1	Rendah
Scene_Play::init	168	1	18	1	Rendah
Scene_Play::loadLevel	34	1	7	1	Rendah
Scene_Play::loadConfig*File*	194	1	31	5	Rendah
Scene_Play::spawnPlayer	205	0	18	1	Rendah
Scene_Play::update	48	1	13	2	Rendah
Scene_Play::sRender	263	0	34	5	Rendah
Scene_Play::sDoAction	373	1	69	18	Sedang
Scene_Play::sCollision	429	0	43	11	Rendah
Scene_Play::sGravity	84	0	15	4	Rendah
Scene_Play::sMovement	390	1	42	7	Rendah
Scene_Play::sAnimation	342	0	44	12	Rendah
Scene_Play::onEnd	28	0	5	1	Rendah
Scene_Play::settingUpStaticEntity	129	4	14	2	Rendah
Scene_Play::gridToMidPixel	108	3	12	2	Rendah
Scene_Play::drawCollision	95	2	10	2	Rendah
Scene_Play::drawGrid	266	1	22	4	Rendah
Scene_TopDown::Scene_TopDown	32	2	6	1	Rendah
Scene_TopDown::init	168	1	18	1	Rendah
Scene_TopDown::loadLevel	42	1	9	1	Rendah
Scene_TopDown::loadConfig*File*	194	1	31	5	Rendah

Nama Metode	**Jumlah Token**	**Jumlah Parameter**	**Jumlah baris kode**	**Nilai *Cyclomatic Complexity***	**Kompleksitas**
Scene_TopDown::spawnPlayer	225	0	18	1	Rendah
Scene_TopDown::update	44	1	12	2	Rendah
Scene_TopDown::sRender	271	0	36	5	Rendah
Scene_TopDown::renderSightPolygon	11	0	4	1	Rendah
Scene_TopDown::raycastDetection	759	0	88	14	Rendah
Scene_TopDown::sortSightPoint	120	0	16	3	Rendah
Scene_TopDown::calculateRaycastAngle	129	1	9	1	Rendah
Scene_TopDown::renderWalls	43	0	7	2	Rendah
Scene_TopDown::setupWalls	595	0	56	3	Rendah
Scene_TopDown::sDoAction	373	1	69	18	Sedang
Scene_TopDown::sCollision	154	0	21	4	Rendah
Scene_TopDown::sMovement	230	1	25	6	Rendah
Scene_TopDown::sAnimation	305	0	41	12	Rendah
Scene_TopDown::onEnd	28	0	5	1	Rendah
Scene_TopDown::settingUpStaticEntity	129	4	14	2	Rendah
Scene_TopDown::gridToMidPixel	108	3	12	2	Rendah
Scene_TopDown::drawCollision	95	2	10	2	Rendah
Scene_TopDown::drawGrid	266	1	22	4	Rendah
Scene_TopDown::AStarPathfinding	148	0	22	4	Rendah
Vec2::Vec2	7	0	1	1	Rendah

Nama Metode	**Jumlah Token**	**Jumlah Parameter**	**Jumlah baris kode**	**Nilai *Cyclomatic Complexity***	**Kompleksitas**
Vec2::Vec2	22	2	4	1	Rendah
Vec2::operator ==	28	1	4	2	Rendah
Vec2::operator !=	28	1	4	2	Rendah
Vec2::operator <	28	1	4	2	Rendah
Vec2::operator >	28	1	4	2	Rendah
Vec2::operator +	29	1	4	1	Rendah
Vec2::operator -	29	1	4	1	Rendah
Vec2::operator /	24	1	4	1	Rendah
Vec2::operator *	24	1	4	1	Rendah
Vec2::operator *	29	1	4	1	Rendah
Vec2::operator +=	24	1	5	1	Rendah
Vec2::operator -=	24	1	5	1	Rendah
Vec2::operator / =	22	1	5	1	Rendah
Vec2::operator * =	22	1	5	1	Rendah
Vec2::length	32	0	4	1	Rendah
Vec2::normalized	21	0	5	1	Rendah
Vec2::cross	24	1	4	1	Rendah
Vec2::approach	91	3	17	5	Rendah