PENERAPAN KONSEP STATE PATTERN PADA GAME ENGINE (STUDI KASUS GAME WIPE IT OFF)

SKRIPSI

Untuk memenuhi sebagian persyaratan
memperoleh gelar Sarjana Komputer

Disusun oleh:
Firadi Surya Pramana
NIM: 145150201111111

PROGRAM STUDI TEKNIK INFORMATIKA
JURUSAN TEKNIK INFORMATIKA
FAKULTAS ILMU KOMPUTER
UNIVERSITAS BRAWIJAYA
MALANG
2018

PENGESAHAN

PENERAPAN KONSEP STATE PATTERN PADA GAME ENGINE
(STUDI KASUS WIPE IT OFF)
SKRIPSI

Diajukan untuk memenuhi sebagian persyaratan
memperoleh gelar Sarjana Komputer

Disusun Oleh :
Firadi Surya Pramana
NIM: 145150201111111

Skripsi ini telah diuji dan dinyatakan lulus pada
15 Januari 2018
Telah diperiksa dan disetujui oleh:

Dosen Pembimbing I	Dosen Pembimbing II
Eriq Muh. Adams Jonemaro, S.T, M.Kom	Muhammad Aminul Akbar, S.Kom., M.T
NIP: 19850410 201212 1 001	NIK. 20160789 1013 001

Mengetahui
Ketua Jurusan Teknik Informatika

Tri Astoto Kurniawan, S.T, M.T, Ph.D
NIP: 19710518 200312 1 001

PERNYATAAN ORISINALITAS

Saya menyatakan dengan sebenar-benarnya bahwa sepanjang pengetahuan saya, di dalam naskah skripsi ini tidak terdapat karya ilmiah yang pernah diajukan oleh orang lain untuk memperoleh gelar akademik di suatu perguruan tinggi, dan tidak terdapat karya atau pendapat yang pernah ditulis atau diterbitkan oleh orang lain, kecuali yang secara tertulis disitasi dalam naskah ini dan disebutkan dalam daftar pustaka.

Apabila ternyata didalam naskah skripsi ini dapat dibuktikan terdapat unsur-unsur plagiasi, saya bersedia skripsi ini digugurkan dan gelar akademik yang telah saya peroleh (sarjana) dibatalkan, serta diproses sesuai dengan peraturan perundang-undangan yang berlaku (UU No. 20 Tahun 2003, Pasal 25 ayat 2 dan Pasal 70).

Malang, 3 Januari 2018

Firadi Surya Pramana

NIM: 145150201111111

KATA PENGANTAR

Puji syukur kehadirat Allah SWT, yang telah memberikan rahmat dan hidayahNya, serta kepada Nabi Muhammad SAW, sehingga penulis dapat menyelesaikan Laporan Skripsi yang berjudul "Penerapan Konsep *State Pattern* pada *Game Engine* (Studi Kasus *Game Wipe It Off*)". Laporan ini disusun untuk memenuhi syarat kelulusan pada Fakultas Ilmu Komputer Universitas Brawijaya. Pada kesempatan kali ini penulis ingin mengucapkan terima kasih kepada:

1. Allah SWT yang dengan Hidayah dan kuasa-Nya, penulis tidak akan mampu menyelesaikan Laporan Skripsi,
2. Bapak Wayan Firdaus Mahmudy, S.Si, M.T, Ph.D selaku Dekan Fakultas Ilmu Komputer Universitas Brawijaya,
3. Bapak Tri Astoto Kurniawan, S.T, M.T, Ph.D selaku Ketua Jurusan Teknik Informatika Fakultas Ilmu Komputer Universitas Brawijaya,
4. Bapak Agus Wahyu Widodo, S.T, M.Cs selaku Ketua Program Studi Teknik Informatika Fakultas Ilmu Komputer Universitas Brawijaya,
5. Bapak Eriq Muh. Adams Jonemaro, S.T, M.Kom selaku Dosen Pembimbing I Skripsi Keminatan Teknologi Media, Game dan Piranti Bergerak Fakultas Ilmu Komputer Universitas Brawijaya,
6. Bapak Muhammad Aminul Akbar , S.Kom., M.T selaku Dosen Pembimbing II Skripsi Keminatan Teknologi Media, Game dan Piranti Bergerak Fakultas Ilmu Komputer Universitas Brawijaya,
7. Orang Tua penulis yang telah memberikan motivasi, sarana dan prasarana kepada penulis dalam pengerjaan laporan Skripsi,
8. Serta semua pihak yang telah membantu secara langsung maupun tidak langsung kepada penulis sehingga penulis dapat menyelesaikan laporan Skripsi.

Penulis sangat menyadari bahwa laporan ini masih jauh dari sempurna. Oleh karena itu, kami mengharapkan kritik dan saran serta penilaian dari pembaca sebagai sarana untuk membangun laporan dengan lebih baik lagi.

Malang, 3 Januari 2018

Penulis

fspramana@gmail.com

ABSTRAK

Game yang menggunakan *state machine* sebagai dasar pengembangan perilaku agen bukan merupakan hal baru lagi. Namun, masih belum banyak *game engine* yang dapat menangani kebutuhan tersebut. *State pattern* merupakan salah satu *design pattern* yang dapat menangani kebutuhan *state machine* pada *game*. Perancangan *state pattern* dilakukan untuk diterapkan dalam *game engine*. Komponen yang terdapat dalam *state pattern* adalah *Initial State*, *Check State*, dan *Handle State*. Perancangan *finite state machine* yang digunakan dalam *game* juga dilakukan. Komponen yang terdapat dalam *finite state machine* adalah *Idle State*, *Moving State*, dan *Cleaning State*. Implementasi *state pattern* pada *game engine* dilakukan dengan membuat kelas *interface* dan menjadi parent dari seluruh *state* dalam *finite state machine*. Implementasi *finite state machine* pada *game* dilakukan dengan memisahkan seluruh state menjadi kelas yang berbeda dan melakukan pewarisan sifat dari kelas *interface*. Pengujian *state pattern* pada *game engine* dilakukan dengan *white-box testing*. Didapatkan bahwa hasil pengujian dari seluruh komponen state pattern adalah valid. Pengujian *finite state machine* pada *game* dilakukan dengan *black-box testing*. Didapatkan bahwa hasil pengujian perpindahan *state* dari *finite state machine* adalah valid. Dengan keberadaan *game engine* ini diharapkan bahwa pengembang *game* tidak lagi kesulitan dalam mengembangkan *game* yang memiliki *state machine* sebagai dasar kebutuhannya.

Kata kunci: *Game engine, Game, State, State pattern.*

ABSTRACT

Game that uses state machine as their agent behaviour decision making is common nowaday. However, there still lots of game engine that can't handle that requirement. State pattern is one of design pattern that could handle that requirement. Components of state pattern on game engine are defined and given the name of Initial State, Check State, and Handle State. Components of finite state machine on game are defined and given the name of Idle State, Moving State, and Cleaning State. State pattern on game engine implemented as a interface class that become thhe parernt of each state on finite state machine. Finite state machine on game implemented as an individual class that inherit the interface class from state pattern. State pattern tested with white-box testing and all the compoent are given the value of valid. Finite state machine tested with black-box testing and all the transitions are succeded and given the value of valid. With this game engine, therefore there is no game developer that will experience difficulty in developing game that uses state machine as their basic need.

Keywords: Game engine, Game, State, State pattern.

DAFTAR ISI

DAFTAR TABEL

DAFTAR GAMBAR

BAB 1 PENDAHULUAN

1.1 Latar Belakang

Dalam proses pembuatan *game*, dibutuhkan *game engine* yang berperan sebagai dasar kontrol dari *game*. G*ame engine* dibuat dan diatur hanya untuk menjalankan game di *platform* tertentu (Gregory, 2015:12). Saat ini, banyak *game engine* yang dikembangkan untuk memenuhi kebutuhan pengembang *game*. Namun, terdapat *game engine* yang sulit untuk dipelajari sehingga dapat memakan waktu dan tidak efisien dalam proses pengembangan *game*. Pengembangan *game engine* seharusnya tidak hanya memperhatikan performa yang lebih baik serta fungsionalitas yang kompleks saja, tetapi juga harus mempertimbangkan struktur internal yang baik (Maggiorini dkk., 2016). Oleh karena itu, diperlukan adanya *design pattern* dalam pengembangan *game engine*.

Design pattern digunakan untuk pengembangan *game engine* berdasarkan pola yang disediakan sehingga *game engine* dapat dikembangkan dengan sistematis. Tujuan dari *design pattern* adalah agar kode yang dibuat mudah dipahami serta mempercepat pengerjaan dari fitur yang ingin diterapkan. Penelitian Guana dkk. (2015), Maggiorini dkk. (2016), dan Munro dkk. (2009) tentang arsitektur *game engine* diketahui bahwa *design pattern game engine* sangat berperan dalam pembangunan *game engine* yang baik dan mudah dipahami oleh pengembang *game*.

Salah satu dari *design pattern* untuk pengembangan *game engine* adalah *state pattern*. *State pattern* adalah pola dimana *game* memanfaatkan *state machine* untuk memodelkan kebutuhan *game* (Nystom, 2014). Pola ini memungkinkan *game* untuk menangani kebutuhan yang tidak disediakan oleh pengecekan kondisi secara konvensional. Dengan pola ini, tidak dibutuhkan penambahan kondisi baru apabila terdapat *state* baru yang ingin diimplementasikan.

Banyak *game* yang menerapkan model *state machine* sebagai dasar perhitungan dalam melakukan pemilihan aksi serta perilaku dari *game object*. Pada penelitian Dianty dkk. (2015), *state machine* digunakan sebagai dasar skenario pengobatan yang dapat dilakukan pemain. Pemain dapat dengan mudah memahami pengetahuan dasar tentang pengobatan pertama yang ditawarkan oleh *game*. Pada penelitian Saini dkk. (2011), *state machine* digunakan sebagai dasar strategi yang dapat dilakukan oleh *Artificial Intelligence(AI)* petarung. *AI* tersebut dapat meniru perilaku dari petarung manusia(pemain) melalui *state machine* yang telah dimodelkan. Dari penelitian di atas, disimpulkan bahwa model *state machine* berpengaruh penting terhadap kemudahan pemahaman dari sistem di dalam *game*.

Berdasarkan tinjauan di atas, penulis tertarik untuk melakukan penelitian mengenai penerapan konsep *state pattern* sebagai dasar pengembangan *game engine*. Diharapkan bahwa *game engine* yang dikembangkan dengan *state*

pattern dapat memudahkan pengembang *game* dalam melakukan pembuatan *game*.

1.2 Rumusan Masalah

Rumusan masalah yang dibahas dalam penelitian ini adalah:

1. Bagaimana penerapan konsep *state pattern* pada *game engine*?
2. Bagaimana penerapan *finite state machine* pada *game* menggunakan *game engine* yang telah dikembangkan?
3. Bagaimana hasil pengujian dari konsep *state pattern* pada *game engine* dan *finite state machine* pada *game*?

1.3 Tujuan

Tujuan yang ingin dicapai dari penelitian ini yaitu,

1. Menerapkan *game engine* dengan konsep *state pattern*.
2. Menerapkan *finite state machine* pada *game* dengan menggunakan *game engine* yang telah dikembangkan.
3. Mendapatkan hasil pengujian dari konsep *state pattern* pada *game engine* dan *finite state machine* pada *game* yang sesuai.

1.4 Manfaat

Manfaat dari penelitian ini yaitu untuk mempermudah pengembang *game* yang ingin mengembangkan *game* yang memiliki model *state machine* agar lebih efisien waktu serta penulisan kode yang lebih efektif.

1.5 Batasan Masalah

Ruang lingkup permasalahan:

1. Penelitian ini hanya membahas proses penerapan state pattern pada *game engine*.
2. Penelitian ini hanya mengangkat *Finite-State Machine* sebagai *state machine* yang diterapkan.

Agar permasalahan yang dibahas dalam penelitian ini mencapai sasaran yang diharapkan, maka batasan-batasan yang perlu diberikan dalam penelitian ini adalah sebagai berikut:

1. Penelitian ini dilakukan pada *platform Windows*.
2. Penelitian ini menggunakan bahasa pemrograman *C++*.
3. Penelitian ini menggunakan *OpenGL 3 Graphic API*.

1.6 Sistematika Pembahasan

Sistematika laporan yang digunakan dalam penyusunan laporan penelitian ini adalah sebagai berikut:

BAB I : PENDAHULUAN

Menguraikan mengenai latar belakang, rumusan masalah, batasan masalah, tujuan, manfaat penelitian dan sistematika laporan.

BAB II : LANDASAN KEPUSTAKAAN

Menguraikan mengenai dasar teori dan referensi yang digunakan sebagai dasar penelitian.

BAB III : METODOLOGI

Menguraikan tentang metode dan langkah kerja yang terdiri dari , studi literatur, perancangan, implementasi, dan pengujian dari *state pattern* dan *finite state machine*.

BAB IV : PERANCANGAN

Menguraikan tentang perancangan *state pattern* pada *game engine* dan perancangan *finite state machine* pada *game*.

BAB V : IMPLEMENTASI

Menguraikan tentang implementasi *state pattern* pada *game engine* dan implementasi *finite state machine* pada *game*.

BAB VI : PENGUJIAN

Menguraikan tentang pengujian *state pattern* pada *game engine* dan pengujian *finite state machine* pada *game*.

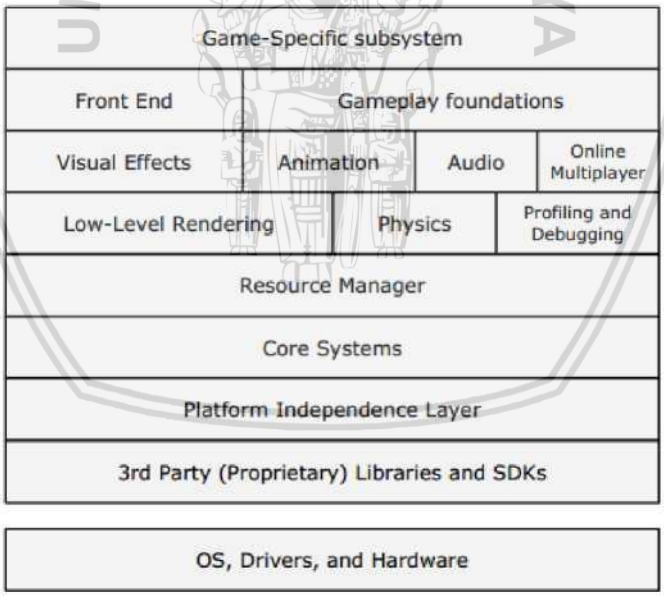
BAB VII : PENUTUP

Menguraikan kesimpulan yang diperoleh serta saran-saran untuk penelitian lebih lanjut.

BAB 2 LANDASAN KEPUSTAKAAN

2.1 Game Engine

Game engine adalah sebuah kumpulan modul perangkat lunak yang digunakan sebagai dasar perancangan dari *game* serta aplikasi yang menggunakan dunia *virtual* dan visualisasi (Ali & Usmar, 2016:1199). *Game engine* memadatkan kompleksitas dari tingkat perancangan *low-level* dan menyediakan *application program interface*(API) dan *software development kit*(SDK) untuk membuat *game* baru (GameCareerGuide: What is a Game Engine?, 2008). Terdapat *game engine* yang berfungsi secara umum dan dapat menangani seluruh permintaan dari *game* yang akan dibuat. Namun, semakin umum *game engine* dibuat, maka semakin tidak optimal *game engine* tersebut untuk menjalankan *game* di *platform* tertentu. Oleh karena itu, *game engine* dibuat dan diatur hanya untuk menjalankan *game* di *platform* tertentu (Gregory, 2015:12). Secara umum, *game engine* yang telah dapat digunakan terbentuk dari beberapa modul penting yaitu *graphic engine, audio engine, physics engine, event module, network module, tools module, dan engine script* dimana seluruh modul-modul tersebut bekerja sesuai dengan fungsinya masing-masing dan bersifat *reusable* yang dapat digunakan secara terus menerus dan berulang-ulang (Yan-Hui, 2011). Komponen dasar dari *game engine* terdapat pada Gambar 2.1.



Gambar 2.1 Komponen dasar *game engine*

(Diambil dari Maggiorini dkk. 2016)

Pada penelitian Guana dkk. (2015), peneliti mengembangkan *game engine* menggunakan pendekatan arsitektur model modern. Dalam penelitian ini diketahui bahwa pendekatan arsitektur model dilakukan dengan memodelkan bahwa komponen dari *game engine* merupakan penurunan dari model komponen yang sama. Peneliti menggunakan *PhsDSL-2* serta *ChainTracker* untuk

menentukan tingkat kompleksitas serta evolusi dari *game engine*. Dengan penelitian ini, ditemukan bahwa pendekatan arsitektur model modern memiliki tingkat efektivitas yang tinggi dalam proses pengembangan *game engine*.

Pada penelitian Munro dkk. (2009), peneliti melakukan analisis dari *game engine Quake* yaitu, *Quake 1* pada *game Quake I*, *QuakeWorld* pada *game QuakeWorld*, *id Tech 2* pada *game Quake II*, *id Tech 3* pada *game Quake III*, *id Tech 3+* pada *game ioquake3*. Dari *game engine* yang telah ditentukan lalu dianalisa untuk mendapatkan arsitektur, kelebihan, serta kekurangan dari *game engine* tersebut. Dari penelitian tersebut didapatkan hasil analisa bahwa modul fungsionalitas, modul logika *game*, modul *client-server*, dan modul *output* multimedia yang sama dari setiap *game engine*. Hasil tersebut dapat digunakan untuk melakukan perancangan *game engine* berdasarkan arsitektur *Quake* dengan memperhatikan kelemahan serta kelebihan dari setiap *game engine*.

Pada penelitian yang dilakukan oleh Maggiorini dkk. (2016), peneliti melakukan perancangan arsitektur *game engine* baru yaitu *Stackless Microkernel Architecture for SHared environment (SMASH)*. *SMASH* diimplementasikan dengan memperhatikan 3 komponen utamanya, yaitu *soft real-time scheduler*, *dynamic game modules manager*, dan *messaging system between module*. Dengan penelitian ini, ditemukan bahwa *SMASH* yang diterapkan kepada *game rubik's cube* dapat menangani banyak *render module* dan dapat dirubah pada saat *game* berjalan.

2.2 State Pattern

State pattern menggunakan pendekatan berorientasi objek dimana setiap cabang dari suatu kondisi dapat ditangani dengan dinamis, yaitu menggunakan *virtual* method (Nystrom, 2014). Pattern ini memperbolehkan *game object* untuk menangani perubahan *state* yang terjadi dari objek itu sendiri. Kelas *interface* digunakan untuk melakukan pendefinisian seluruh aksi yang berkaitan dengan penanganan *state*. Setiap *state* yang telah didefinisikan, dirancang menjadi kelas dengan melakukan *inheritance* dari kelas *interface* yang telah dibuat. Kelas *state* dapat dirancang dengan melakukan *overloading* dari fungsi yang didefinisikan di kelas *interface* berdasarkan perilaku dari *state* serta kondisi perpindahannya.

Tabel 2.1 *HeroineState*

No	HerioneState
1	`class HeroineState`
2	`{`
3	`public:`
4	`  virtual ~HeroineState() {}`
5	`  virtual void handleInput(Heroine& heroine, Input input) {}`
6	`  virtual void update(Heroine& heroine) {}`
7	
8	`  static StandingState standing;`
9	`  static DuckingState ducking;`
10	`  static JumpingState jumping;`
11	`  static DivingState diving;`
12	

```
13    // Other code...
14
15  };
```

Pada Tabel 2.1, kelas *HeroineState* merupakan contoh penerapan *state pattern* pada *game engine*. Baris 5 dan 6 adalah fungsi dari state pattern dalam bentuk virtual yang harus dilakukan *override* sebelum digunakan oleh state machine. Baris 8-11 adalah seluruh *state* yang terdapat dalam *state machine*. Seluruh *state* tersebut didefinisikan sehingga dapat digunakan dalam pengembangan *game*.

Tabel 2.2 *DuckingState*

No	DuckingState
1	`HeroineState* DuckingState::handleInput(Heroine& heroine,`
2	`                                       Input input)`
3	`{`
4	`  if (input == RELEASE_DOWN)`
5	`  {`
6	`    heroine.setGraphics(IMAGE_STAND);`
7	`    return new StandingState();`
8	`  }`
9	
10	`  // Other code...`
11	`}`

Pada Tabel 2.2, kelas *DuckingState* merupakan contoh penerapan *state* pada *game*. Baris 1-11 adalah fungsi *handleInput* yang di-*override* dari fungsi *handleInput* pada kelas *HeroineState* pada Tabel 2.1.

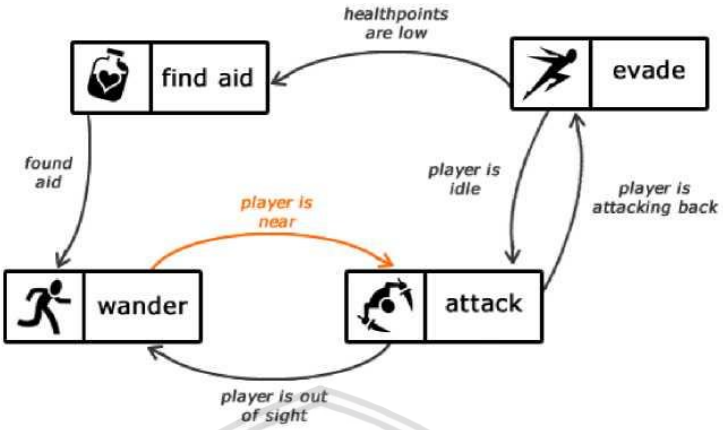
2.3 Finite State Machine

Finite State Machine(FSM) adalah sebuah model komputasi yang dibuat dari satu atau banyak *state* (Wright, 2005). *State* adalah sebuah aksi atau perilaku yang sedang dilakukan. *FSM* menampung seluruh *state* yang dapat dilakukan. *State* pada *FSM* dapat berubah dari satu *state* ke *state* lainnya melalui masukan tertentu sehingga terjadi transisi antar *state*. Perubahan *state* mengakibatkan perubahan perilaku serta aksi yang dilakukan oleh objek tersebut. Namun, hanya terdapat satu *state* saja yang dapat aktif pada satu waktu. *FSM* pada Gambar 2.2 merupakan contoh sederhana dalam penggunaan *state machine*.

Pada penelitian Dianty dkk. (2015), *FSM* digunakan untuk memodelkan skenario pengobatan yang dapat dilakukan pemain. Skenario yang dimodelkan dalam *FSM* terdapat 10 skenario. Penelitian dimulai dengan pendefinisian *state machine* yaitu *player action at map, bonus game interface,* dan *player action on bonus game*. Peneliti mendapati bahwa pemain dapat dengan mudah memahami pengetahuan dasar tentang pengobatan pertama yang ditawarkan oleh *game*.

Pada penelitian Saini dkk. (2011), *FSM* digunakan untuk memodelkan strategi yang dapat dilakukan oleh *AI* petarung. *AI* tersebut dapat meniru perilaku dari petarung manusia(pemain) melalui *FSM* yang telah dimodelkan. Dari penelitian

di atas, dapat diketahui bahwa *FSM* berpengaruh penting terhadap kemudahan pemahaman dari sistem di dalam *game*.



Gambar 2.2 *Finite State Machine*

(Diambil dari gamedevelopment.tutplus.com)

2.4 OpenGL

Open Graphic Library(OpenGL) adalah *API cross-platform* dan *cross-language* untuk menjalankan *render* grafis 2D dan 3D (OpenGL, 1992). *API* ini digunakan untuk melakukan interaksi dengan *graphics processing unit(GPU)* untuk mencapai *hardware-accelerated rendering*. *API* ini dapat dimanfaatkan dalam bidang desain, *virtual reality*, visualisasi sains, visualisasi informasi, simulasi penerbangan, dan *video game*.

Terdapat banyak jenis dari *OpenGL*. *GL* yang menangani pengerjaan 2D dan 3D yaitu *OpenGL*, *OpenGL ES*, *OpenGL SC*, *OpenVG* dan *Vulkan*. *GL* yang menangani pengerjaan pada *web* yaitu *WebGL*, *glTF* dan *COLLADA*. *GL* yang menangani pengerjaan *VR*, *vision*, *neural network* yaitu *OpenXR*, *OpenVX* dan *NNEF*. *GL* yang menangani pengerjaan komputasi paralel yaitu *SPIR*, *SYCL* dan *OpenCL*.

2.5 C/C++ Programming Language

C/C++ adalah bahasa pemrograman yang dapat menyediakan akses terhadap manipulasi memori *low-level* (C++ Language, 1985). *C++* memberikan kontrol penuh dari program kepada *developer*. Bahasa ini memiliki dua komponen utama, *mapping hardware* secara langsung serta *mapping* yang memiliki abstraksi *zero-overhead*. *C++* adalah bahasa pemrograman yang berdasar pada *C* dengan penambahan beberapa fitur. Fitur tersebut adalah *object storage, objects, encapsulation, inheritance, operators* dan *operator overloading*, *polymorphism, lambda expressions* serta *exception handling*. Kode pada Tabel 2.3 adalah contoh sederhana dari *source code C++* yang akan menampilkan "Hello World!" pada *console*.

Tabel 2.3 *HelloWorld*

No	HelloWorld
1	`#include <iostream>`
2	
3	`int main()`
4	`{`
5	`  std::cout << "Hello World!";`
6	`}`

2.6 Simple DirectMedia Layer

Simple DirectMedia Layer(SDL) adalah sebuah perangkat *library* pengembangan yang memungkinkan kita untuk mengakses *keyboard*, *mouse*, *audio*, *joystick*, dan *graphic driver* dengan menggunakan *interface* pemrograman aplikasi *OpenGL* atau *Direct3D* (Simple DirectMedia Layer, 1998). *SDL* dalam penelitian ini akan mendasari pembuatan *game engine* dalam hal pengelolaan *window*. *SDL* berdiri diatas bahasa pemrograman *C*, namun secara *native* dapat bekerja dengan bahasa pemrograman *C++*, dan mendukung untuk bahasa pemrograman *C#* dan juga *Phyton*. *SDL* juga merupakan *library* pengembangan yang *multiplatform* yang dapat bekerja pada *platform windows, linux,* maupun *Mac OS X*, dan juga *platform mobile* yaitu *IOS* dan *Android*.

2.7 IrrKlang

IrrKlang adalah salah satu *Sound Engine* dan *Audio Library* yang menyediakan fitur untuk memainkan file *audio* berformat *WAV, MP3, OGG, FLAC*, dan masih banyak lagi (irrKlang, 2006). *Library* ini juga dapat digunakan di beberapa *platform* seperti *Windows, Linux, Mac OS X. irrKlang* bersifat *free* untuk projek yang tidak dikomersialkan, namun untuk yang dikomersialkan *irrKlang* memiliki produk lain yang mempunyai dukungan lebih. *Library* ini bekerja pada beberapa bahasa pemrograman seperti *C++,* dan semua turunan *.NET*.

2.8 The OpenGL Extension Wrangler Library

The OpenGL Extension Wrangler Library(GLEW) adalah *library loader* yang bersifat *cross-platform* dan *open-source* menggunakan bahasa *C/C++* (The OpenGL Extension Wrangler Library, 2002). *GLEW* menyediakan fitur untuk menentukan versi *OpenGL* yang cocok untuk suatu target *platform* sehingga penggunaan *OpenGL* akan lebih maksimal dan tidak mengalami kegagalan.

2.9 OpenGL Mathematics

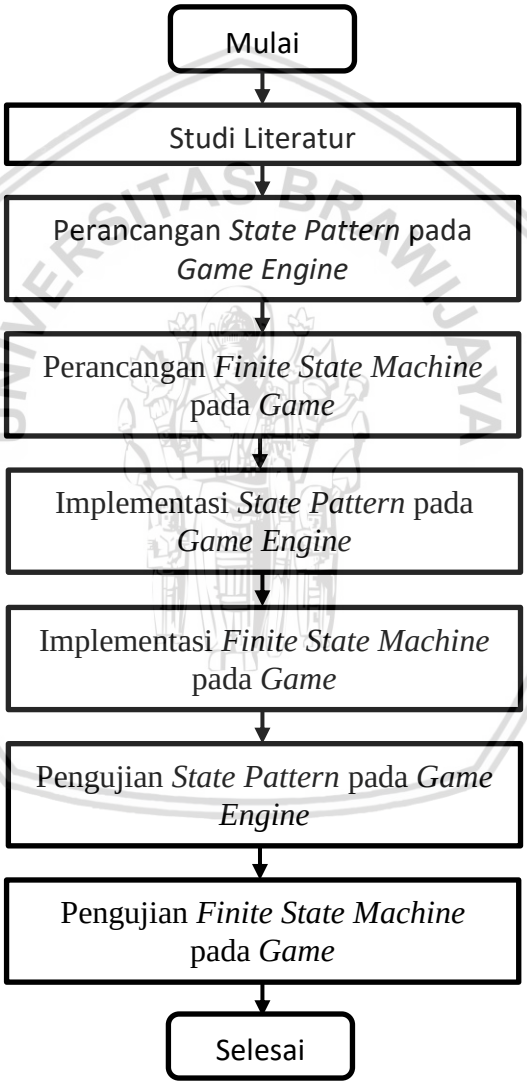
OpenGL Mathematics(GLM) adalah sebuah *library* yang berisi *library* matematis yang diperuntukkan pemrograman grafis menggunakan *OpenGL Shading Language* (OpenGL Mathematics, 2005). Namun tidak hanya itu *GLM* juga menyediakan beberapa fitur matematis yang lainnya. Seperti halnya *library* yang memiliki dasar *OpenGL, GLM* juga bersifat *cross-platfom* dan *open-source*.

BAB 3 METODOLOGI

Metode penelitian yang digunakan adalah dengan melakukan penerapan dimana peneliti mengumpulkan data untuk dilakukan pengujian dan analisa hasil uji. Adapun metode dalam penelitian ini meliputi:

3.1 Diagram Alir Penelitian

Diagram alir penelitian penerapan konsep *state pattern* pada *game engine* (studi kasus *game Wipe It Off*) dapat ditunjukkan pada Gambar 3.1.



Gambar 3.1 Diagram alir penelitian

3.2 Studi Literatur

Studi literatur dilakukan dengan mempelajari buku-buku literatur yang berkaitan dengan *game engine*. Studi literatur dapat berupa media cetak dan media elektronik.

3.3 Perancangan *State Pattern* pada *Game Engine*

Pada tahap ini, dilakukan penggambaran alur kerja *state pattern*. Pada awal pertama kali game berjalan, proses pertama yang dilakukan adalah Initial State. Setelah itu proses *Update* akan dijalankan secara iteratif. Di dalam proses tersebut terdapat proses Check State dan Handle State yang dijalankan secara sekuensial.

3.4 Perancangan *Finite State Machine* pada *Game*

Pada tahap ini, dilakukan penggambaran *finite state machine* yang digunakan dalam *game*. Terdapat tiga komponen yang dirancang yaitu *Idle State*, *Moving State*, dan *Cleaning State*. *Idle State* adalah *state* pada saat pemain tidak melakukan aksi apapun. *Moving State* adalah *state* pada saat pemain melakukan akse bergerak. *Cleaning State* adalah state pada saat pemain melakukan aksi membersihkan.

3.5 Implementasi *State Pattern* pada *Game Engine*

Pada tahap ini, dilakukan implementasi sesuai dengan alur kerja *state pattern* sehingga setiap komponen dapat digunakan pada *game engine*. State pattern diimplementasikan menjadi sebuah kelas *interface* yang menjadi *parent* dari setiap *state* dalam *finite state machine*. Kode program juga akan ditampilkan dan dijelaskan untuk setiap barisnya.

3.6 Implementasi *Finite State Machine* pada *Game*

Pada tahap ini, dilakukan implementasi sesuai penggambaran *finite state machine* yang digunakan dalam *game* sehingga setiap komponen dapat digunakan. Finite state machine diimplementasikan menjadi kelas yang berbeda dari setiap state. Setiap kelas state memiliki perilaku yang berbeda. Kode program juga akan ditampilkan dan dijelaskan untuk setiap barisnya.

3.7 Pengujian *State Pattern* pada *Game Engine*

Pada tahap ini, dilakukan pengujian alur kerja *state pattern* pada *game engine*. Dilakukan pembuatan pesudocode dari alur kerja state pattern. Pengujian dilakukan dengan menggunakan *white-box testing* sehingga struktur *state pattern* dapat diuji.

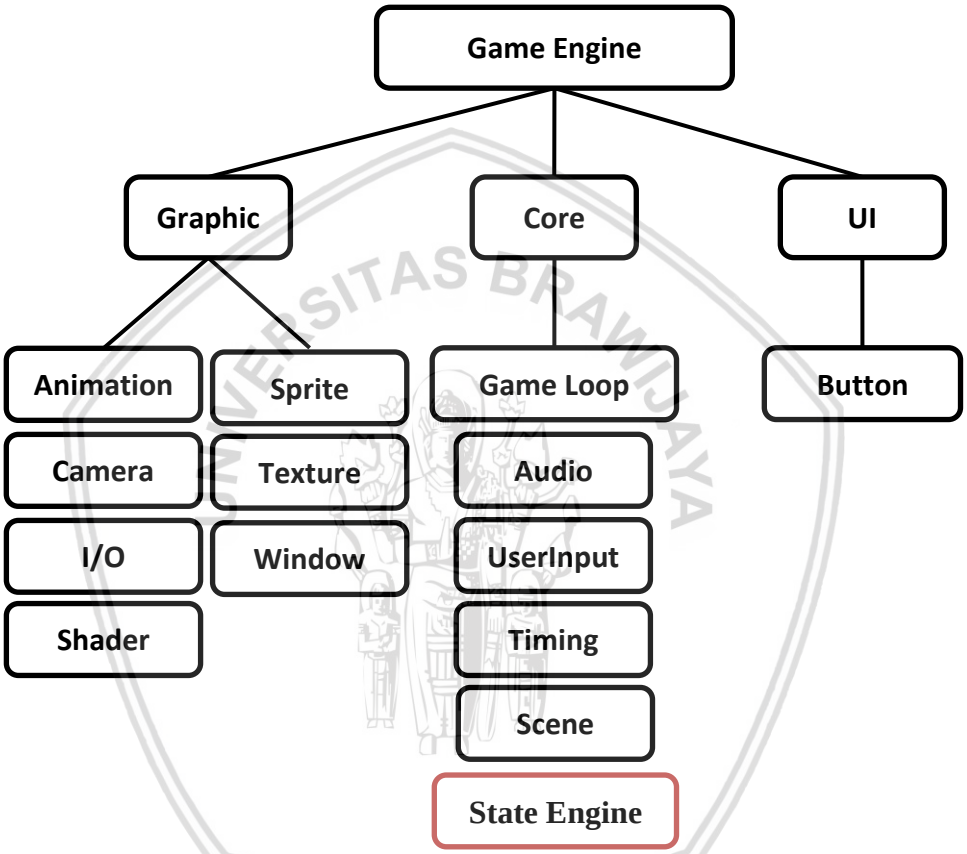
3.8 Pengujian *Finite State Machine* pada *Game*

Pada tahap ini, dilakukan pengujian setiap state dalam *state machine* yang digunakan dalam *game*. Seluruh transisi dari state diuji. Pengujian dilakukan dengan menggunakan *black-box testing* sehingga perpindahan state dapat diketahui keberhasilannya.

BAB 4 PERANCANGAN

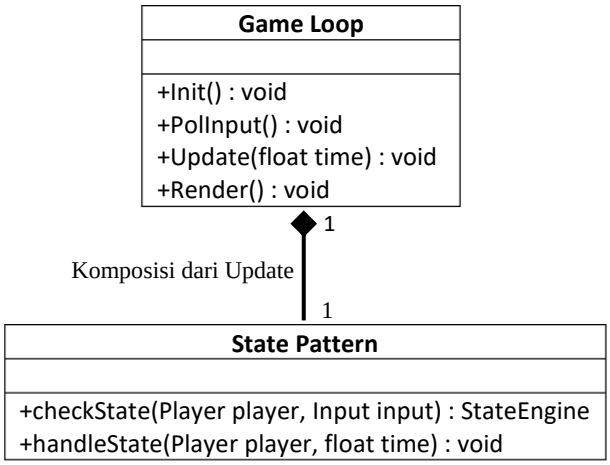
4.1 Perancangan *State Pattern* pada *Game Engine*

State pattern digunakan untuk mengenkapsulasi setiap *state* yang berada pada *state machine*. *State pattern* berada di dalam segmen *Core* pada *game engine* karena merupakan komponen utama yang mengatur jalan dari *game*. Penempatan komponen *state pattern* terdapat pada Gambar 4.1.



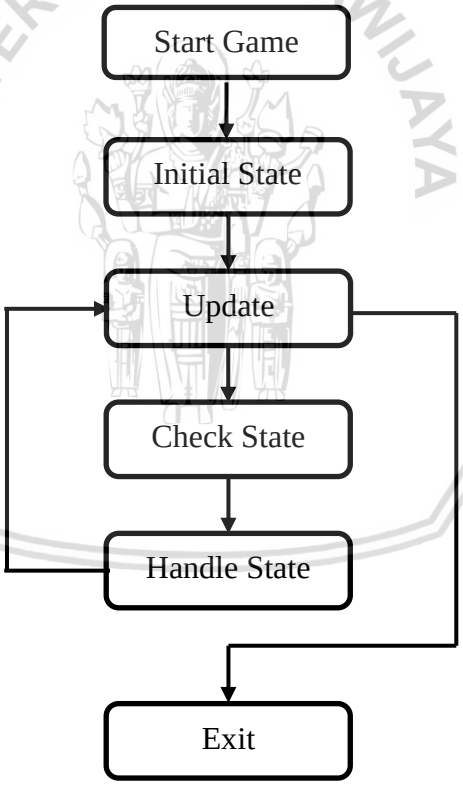
Gambar 4.1 Komponen *game engine*

State pattern dirancang dalam bentuk kelas *interface*. Kelas tersebut menjadi *parent* dari seluruh *state* dalam *state machine* pada *game*. Karena seluruh *state* memiliki perilaku yang berbeda, pembuatan fungsi di dalam kelas tersebut juga harus dapat diubah sesuai dengan perilaku *state* yang diwarisi. *State pattern* merupakan bagian dari *game loop* yang berjalan pada saat *Update game engine* dilakukan. *State pattern* tidak akan berjalan apabila *game loop* tidak berjalan. Oleh karena itu, *state pattern* memiliki hubungan komposisi dari *game loop*. *Class diagram* dari *state pattern* terdapat pada Gambar 4.2.



Gambar 4.2 *Class diagram state pattern*

State pattern memiliki alur kerja sehingga dapat mencapai proses yang diharapkan. Alur kerja *state pattern* yang digunakan pada *game engine* terdapat pada Gambar 4.3.



Gambar 4.3 Alur *state pattern*

Terdapat tiga komponen penting yang harus dirancang. Komponen tersebut adalah *Initial State*, *Check State*, dan *Handle State*.

Alur dari *state pattern* dimulai dari *intial state* pertama pada *state machine*. Pada proses tersebut dibuat sebuah variabel yang menyimpan kondisi awal dari *state*.

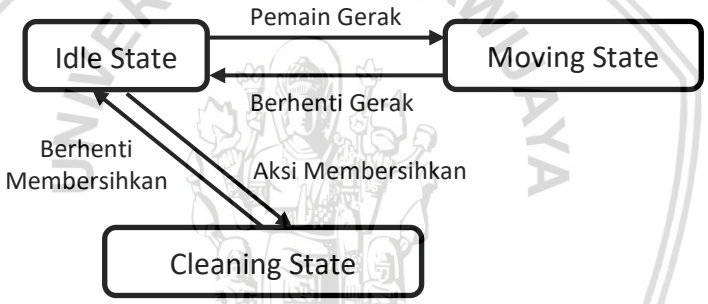
Proses *update* adalah proses dimana game sedang berlangsung. Proses ini dilakukan dengan cara iterasi. Pada proses ini, dilakukan dua proses yaitu *check state* dan *handle state*.

Proses *check state* adalah proses pengecekan apabila terdapat masukan dari pemain yang dapat mengakibatkan transisi *state* sehingga state dalam kondisi yang tepat.

Proses *handle state* dilakukan untuk menjalankan seluruh *behaviour* atau perilaku dari pemain ataupun agen. Seluruh proses akan berhenti pada saat proses update berhenti dan keluar dari program.

4.2 Perancangan *Finite State Machine* pada *Game*

Finite state machine digunakan pada *game Wipe It Off* untuk mengenkapsulasi perilaku dari pemain serta bagaimana cara untuk berpindah menuju *state* lain. Finite state machine yang digunakan terdapat pada Gambar 4.4.



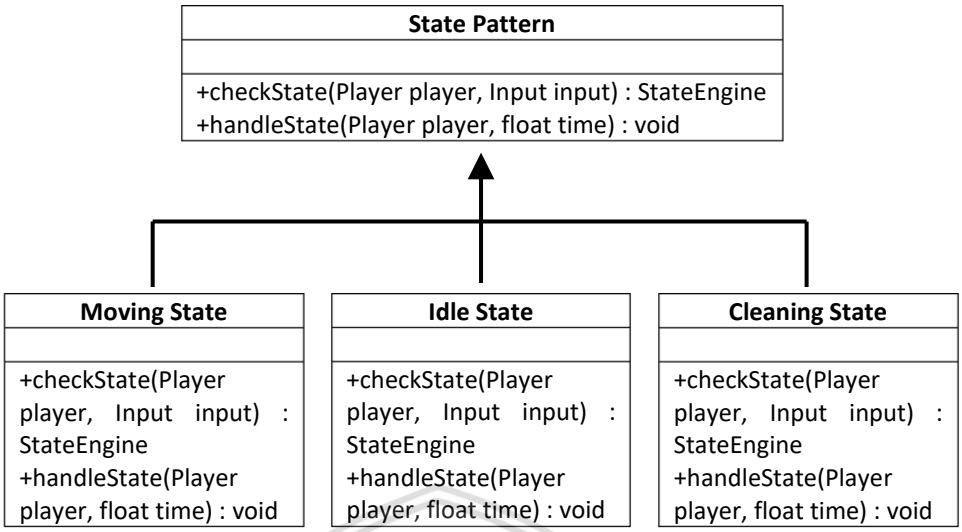
Gambar 4.4 *Finite state machine game wipe it off*

State pada *game* ini yaitu *Idle state*, *Moving state*, dan *Cleaning state*. Pada saat game pertama kali dimulai, pemain akan memasuki *Idle state* dimana pemain hanya diam tidak melakukan aktivitas apapun.

Pada saat terdapat aksi pemain bergerak, maka terjadi transisi dari *Idle state* menuju *Moving state*. Pada *Moving state*, pemain dapat bergerak menuju arah yang diinginkan oleh pemain namun pemain tidak dapat melakukan aksi membersihkan. Saat pemain berhenti bergerak, maka terjadi transisi dari *Moving state* menuju *Idle state*.

Pada saat terdapat aksi pemain membersihkan, maka terjadi transisi dari *Idle state* menuju *Cleaning state*. Pada *Cleaning state*, pemain tidak dapat bergerak namun pemain dapat melakukan aksi membersihkan kaca. Saat pemain berhenti membersihkan, maka terjadi transisi dari *Cleaning state* menuju *Idle state*.

Setiap kelas *state* dalam *finite state machine* merupakan anak dari kelas *interface* pada *state pattern*. *Class diagram* dari *finite state machine* terdapat pada Gambar 4.5.



Gambar 4.5 *Class digram finite state machine game wipe it off*

Kelas *Idle State*, *Moving State*, dan *Cleaning State* merupakan pendefinisian kelas dalam *finite state machine*. Kelas tersebut mendapatkan pewarisan sifat dari kelas *State Pattern*. Setiap fungsi yang terdapat pada state pattern di-*override* oleh masing-masing kelas. Proses tersebut dilakukan karena perilaku Check State dan Handle State dari masing-masing kelas berbeda.

BAB 5 IMPLEMENTASI

5.1 Implementasi *State Pattern* pada *Game Engine*

Implementasi *state pattern* dilakukan sesuai dengan alur yang didefinisikan pada Gambar 4.2. Proses *Initial State* dilakukan pada agen atau pemain yang menerapkan *finite state machine*. Inisialisasi dilakukan dengan pendeklarasian variabel bertipe data *StateEngine* dan diberi nilai *state* awal.

Kelas *StateEngine* digunakan sebagai kelas *interface* yang menjadi *parent* dari kelas *state* pada *FSM*. Kode dari *state pattern* terdapat pada Tabel 5.1.

Tabel 5.1 *State engine*

No	StateEngine
1	`class StateEngine {`
2	`public:`
3	`virtual ~StateEngine() {}`
4	`virtual StateEngine* checkState(Player& player,`
5	`InputManager& input) = 0;`
6	`virtual void handleState(Player& player, float`
7	`deltaTime) = 0;`
8	`};`

Baris 4-5 adalah proses alur *Check State* yang didefinisikan ke dalam fungsi *checkState*. Fungsi ini digunakan untuk melakukan pengecekan masukan dari pemain serta sehingga transisi *state* dapat dilakukan dengan cara melakukan *return* instansiasi dari objek.

Baris 6-7 adalah proses alur *Handle State* yang didefinisikan ke dalam fungsi *handleState* yang digunakan untuk menjalankan perilaku dari *state* yang sedang aktif.

5.2 Implementasi *Finite State Machine* pada *Game*

Implementasi *finite state machine* dilakukan sesuai dengan perancangan yang didefinisikan pada Gambar 4.3. Terdapat tiga state yang diimplementasikan yaitu *Idle State*, *Moving State*, dan *Cleaning State*.

5.2.1 Idle State

Idle State adalah *state* dalam *state machine* dimana pemain tidak melakukan aksi bergerak maupun membersihkan. Kode dari *Idle State* terdapat pada Tabel 5.2.

Tabel 5.2 *Idle state*

No	IdleState
1	`StateEngine* IdleState::checkState(Player& player,`
2	`InputManager& input)`
3	`{`
4	`if (input.isKeyDown(SDLK_d)) {`
5	`return new MovingState();`
6	`}`

```
7       else if (input.isKeyDown(SDLK_a)) {
8             return new MovingState();
9       }
10      else if (input.isKeyDown(SDLK_w)) {
11            return new MovingState();
12      }
13      else if (input.isKeyDown(SDLK_s)) {
14            return new MovingState();
15      }
16      else if (input.isKeyDown(SDLK_SPACE)) {
17            return new CleaningState();
18      }
19
20      return NULL;
21  }
22
23  void IdleState::handleState(Player& player, float
24  deltaTime)
25  {
26      player.set_action(false);
27      player.set_ulang(false);
28      player.set_gondolaMove(false);
29  }
```

Baris 1-21 adalah fungsi *checkState* yang digunakan untuk melakukan pengecekan dari masukkan dari pemain sehingga *state* dapat berganti. Pada saat tombol W, A, S, atau D ditekan, maka fungsi akan melakukan *return* sebuah instansiasi kelas objek *MovingState*. Namun pada saat tombol SPASI ditekan, maka fungsi akan melakukan *return* sebuah instansiasi kelas objek *CleaningState*.

Baris 23-29 adalah fungsi *handleState* yang berisi semua perilaku yang dapat dilakukan oleh pemain ketika *state Idle* dalam keadaan aktif. Pada *state Idle*, pemain memiliki perilaku dimana pemain tidak sedang membersihkan dan bergerak. Maka nilai dari aksi membersihkan(action), mengulang aksi membersihkan(ulang), dan aksi bergerak(gondolaMove) harus dirubah menjadi *false*. Perilaku tersebut hanya terjadi pada *Idle State* saja.

5.2.2 Moving State

Moving State adalah *state* dalam finite *state machine* dimana pemain melakukan aksi gerak. Kode dari *Moving State* terdapat pada Tabel 5.3.

Tabel 5.3 *Moving state*

No	MovingState
1	`StateEngine* MovingState::checkState(Player& player,`
2	`InputManager& input)`
3	`{`
4	`      if (!input.isKeyDown(SDLK_d) &&`
5	`!input.isKeyDown(SDLK_a) &&`
6	`            !input.isKeyDown(SDLK_w) &&`
7	`!input.isKeyDown(SDLK_s)) {`
8	`            return new IdleState();`
9	`      }`

```
10
11          return NULL;
12  }
13
14  void MovingState::handleState(Player& player, float
15  deltaTime)
16  {
17          if (player.get_input()->isKeyDown(SDLK_d)) {
18                  player.set_tempX(player.get_tempX() +
19  player.get_speed() * deltaTime *
20  player.get_drunkFactor());
21                  player.set_gondolaMove(true);
22                  player.set_action(false);
23          }
24          else if (player.get_input()->isKeyDown(SDLK_a))
25  {
26                  player.set_tempX(player.get_tempX() -
27  player.get_speed() * deltaTime *
28  player.get_drunkFactor());
29                  player.set_gondolaMove(true);
30                  player.set_action(false);
31          }
32          else if (player.get_input()->isKeyDown(SDLK_w))
33  {
34                  player.set_tempY(player.get_tempY() +
35  player.get_speed() * deltaTime *
36  player.get_drunkFactor());
37                  player.set_gondolaMove(true);
38                  player.set_action(false);
39          }
40          else if (player.get_input()->isKeyDown(SDLK_s))
41  {
42                  player.set_tempY(player.get_tempY() -
43  player.get_speed() * deltaTime *
44  player.get_drunkFactor());
45                  player.set_gondolaMove(true);
46                  player.set_action(false);
47          }
48  }
```

Baris 1-12 adalah fungsi *checkState* yang digunakan untuk melakukan pengecekan dari masukkan dari pemain sehingga *state* dapat berganti pada saat tombol W, A, S, dan D dilepas. Jika kondisi terpenuhi, maka fungsi akan melakukan *return* sebuah instansiasi kelas objek *IdleState*.

Baris 14-48 adalah fungsi *handleState* yang berisi semua perilaku yang dapat dilakukan oleh pemain ketika *state Moving* dalam keadaan aktif. Pada *state Moving*, pemain memiliki perilaku dimana pemain sedang melakukan aksi bergerak. Fungsi harus melakukan pengecekan terlebih dahulu terhadap tombol manakah yang sedang ditekan oleh pemain sebelum melakukan eksekusi dari perilaku yang seharusnya dilakukan. Setelah pengecekan dilakukan, maka pemain dapat menggerakkan karakternya dengan mengubah nilai dari posisi X(tempX) sesuai dengan perpindahan dan kecepatan bergerak, nilai dari posisi Y(tempY) sesuai dengan perpindahan dan kecepatan bergerak, aksi

bergerak(gondolaMove) menjadi *true*, dan aksi membersihkan(action) menjadi *false*. Perilaku tersebut hanya terjadi pada *Moving State* saja.

5.2.3 Cleaning State

Cleaning State adalah *state* dalam *state machine* dimana pemain melakukan aksi membersihkan. Kode dari *Cleaning State* terdapat pada Tabel 5.4.

Tabel 5.4 *Cleaning state*

No	CleaningState
1	`StateEngine * CleaningState::checkState(Player &`
2	`player, InputManager & input)`
3	`{`
4	`  if (!input.isKeyDown(SDLK_SPACE)) {`
5	`  return new IdleState();`
6	`  }`
7	`  return NULL;`
8	`}`
9	
10	`void CleaningState::handleState(Player & player,`
11	`float deltaTime)`
12	`{`
13	`  if (player.get_input()->isKeyDown(SDLK_SPACE))`
14	`{`
15	`  player.set_action(true);`
16	`  player.set_ulang(true);`
17	`  player.set_gondolaMove(false);`
18	`  player.set_counterrepeat(0);`
19	`  }`
20	`}`

Baris 1-8 adalah fungsi *checkState* yang digunakan untuk melakukan pengecekan dari masukkan dari pemain sehingga *state* dapat berganti pada saat tombol SPASI dilepas. Jika kondisi terpenuhi, maka fungsi akan melakukan *return* sebuah instansiasi kelas objek *IdleState*.

Baris 10-20 adalah fungsi *handleState* yang berisi semua perilaku yang dapat dilakukan oleh pemain ketika *state Cleaning* dalam keadaan aktif. Pada *state Cleaning*, pemain memiliki perilaku dimana pemain sedang melakukan aksi membersihkan. Fungsi harus melakukan pengecekan terlebih dahulu terhadap tombol manakah yang sedang ditekan oleh pemain sebelum melakukan eksekusi dari perilaku yang seharusnya dilakukan. Setelah pengecekan dilakukan, maka pemain dapat membersihkan menggunakan karakternya dengan mengubah nilai dari aksi membersihkan(action) menjadi *true*, mengulang aksi membersihkan(ulang) menjadi *true*, dan aksi bergerak(gondolaMove) menjadi *false*. Perilaku tersebut hanya terjadi pada *Cleaning State* saja.

BAB 6 PENGUJIAN

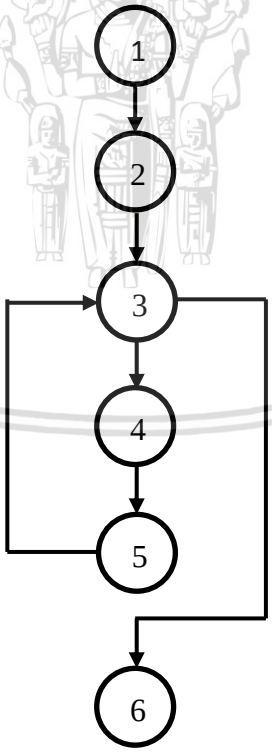
6.1 Pengujian *State Pattern* pada *Game Engine*

State pattern pada *game engine* diuji menggunakan metode *white-box testing*. Pengujan ini bertujuan untuk menguji struktur dari alur state pattern pada Gambar 4.2. Pseudocode dari state pattern terdapat pada Tabel 6.1.

Tabel 6.1 *Pseudocode state pattern*

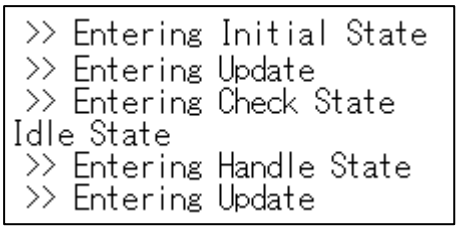
No	Pseudocode State Pattern
1	`Define variable state engine (1)`
2	
3	`Initialize state of state engine variable (2)`
4	
5	`while Update is true (3)`
6	
7	`      Check active state (4)`
8	
9	`      Handle behavior of active state (5)`
10	
11	`End Update (6)`

Basic path state pattern dari Tabel 6.1 terdapat pada Gambar 6.1. Basic path digunakan untuk menguji berbagai macam path yang dapat dijalankan.



Gambar 6.1 Basic path *state pattern*

Bukti pengujian *state pattern* terdapat pada Gambar 6.2. Hasil tersebut didapat dari pengujian path dari Gambar 6.1. Path pertama adalah 1-2-3-4-5-3-6. Path kedua adalah 1-2-3-6.



Gambar 6.2 Hasil uji *state pattern* pada *game engine*

Hasil pengujian dari setiap *path* terdapat pada Tabel 6.4.

Tabel 6.2 Pengujian *state pattern* pada *game engine*

No	Test Case	Expected Result	Result	Valid
1	Update true	Check State dan Handle State berjalan	Check State dan Handle State berjalan	Ya
2	Update false	Keluar dari program	Keluar dari program	Ya

Pengujian state pattern yang dilakukan mendapatkan hasil bahwa seluruh path yang diuji telah mendapatkan nilai valid. Nilai valid menjelaskan bahwa pengujian sukses dan struktur dari state pattern sudah benar.

6.2 Pengujian *Finite State Machine* pada *Game*

Finite state machine pada *game* diuji menggunakan metode *black-box testing*. Pengujian ini bertujuan untuk menguji validitas dari transisi state. Hasil pengujian dari finite state machine terdapat pada Tabel 6.3.

Tabel 6.3 Pengujian *finite state machine* pada *game*

No	Komponen	Test Case	Expected Result	Result	Valid
1	Idle State	Tidak ada tombol ditekan	Tetap di Idle State	Tetap di Idle State	Ya
2	Idle State	Tombol W ditekan	State berganti menjadi Moving State	State berganti menjadi Moving State	Ya
3	Idle State	Tombol SPASI ditekan	State berganti menjadi Cleaning State	State berganti menjadi Cleaning State	Ya
4	Moving State	Tombol W ditekan	Tetap di Moving State	Tetap di Moving State	Ya

5	Moving State	Tidak ada tombol ditekan	State berganti menjadi Idle State	State berganti menjadi Idle State	Ya
6	Cleaning State	Tombol SPASI ditekan	Tetap di Cleaning State	Tetap di Cleaning State	Ya
7	Cleaning State	Tidak ada tombol ditekan	State berganti menjadi Idle State	State berganti menjadi Idle State	Ya

Pengujian *finite state machine* yang dilakukan mendapatkan hasil bahwa seluruh perpindahan seluruh state yang diuji telah mendapatkan nilai valid. Nilai valid menjelaskan bahwa pengujian sukses dan seluruh transisi dalam finite state machine telah benar.

BAB 7 PENUTUP

7.1 Kesimpulan

Dalam penelitian ini dapat diambil kesimpulan bahwa:

- Penerapan konsep *state pattern* pada *game engine* dapat dilakukan dengan membuat kelas *interface* yang berisi fungsi *Check State* dan *Handle State*. Kelas tersebut menjadi *parent* dari seluruh *state* dalam *finite state machine* yang akan dikembangkan dalam *game*.
- Penerapan *finite state machine* pada *game* dilakukan dengan mengonversi setiap *state* menjadi kelas yang mewarisi kelas *interface* yang telah dibuat pada *game engine*. Seluruh fungsi yang diwariskan dilakukan *overriding* sehingga setiap *state* dapat bekerja sesuai dengan kebutuhan dari *state machine*. Fungsi *Check State* diisi dengan syarat transisi dari setiap *state*. Sedangkan fungsi *Handle State* diisi dengan *behavior* dari setiap *state*.
- Hasil pengujian *state pattern* pada *game engine* dan *finite state machine* pada *game* adalah menunjukkan nilai valid. Nilai tersebut menudeskripsikan bahwa komponen telah sukses diuji dan sesuai.

7.2 Saran

Game engine dapat dikembangkan lebih lanjut lagi dengan cara optimasi komponen dengan melakukan pengujian performa dari setiap komponen sehingga struktur dari komponen menjadi lebih baik, lebih rapi, dan lebih sederhana.

DAFTAR PUSTAKA

Ali, Z., Usman, M., 2016. A Framework for Game Engine Selection for Gamification and Serious Games. Future Technologies Conference(FTC).

C++ Language, 1985. Tersedia di:< http://www.cplusplus.com> [Diakses 16 Agustus 2017]

Dianty, E.D., Azhari, A.M., Hakim, M.F.A., Kuswardayan, I., Yuniarti, A., Herumurti, D., 2015. First Aid Simulation Game with Finite State Machine Model. International Conference on Information, Communication Technology and System(ICTS).

Gregory, Jason, 2015. Game Engine Architecture. 2nd ed. Boca Raton: CRC Press.

Guana, V., Stroulia, E., Nguyen, V., 2015. Building a Game Engine: A Tale of Modern Model-Driven Engineering. IEEE/ACM 4th International Workshop on Games and Software Engineering.

irrKlang, 2006. Tersedia di:< https://www.ambiera.com/irrklang> [Diakses 16 Agustus 2017]

Maggiorini, D., Ripamonti, L.A., Zanon, E., Bujari, A., Palazzi, C.E., 2016. SMASH: a Distributed Game Engine Architecture. IEEE Symposium on Computers and Communication(ISCC).

Munro, J., Boldyreff, C., Capiluppi, A., 2009. Architectural Studies of Game Engines - the Quake Series. International IEEE Consumer Electronics Society's Games Innovations Conference.

Nystrom, Bob, 2014. Game Programming Patterns. Tersedia di:<http://gameprogrammingpatterns.com> [Diakses 16 Agustus 2017]

OpenGL, 1992. Tersedia di:<https://www.opengl.org> [Diakses 16 Agustus 2017]

OpenGL Mathematics, 2005. Tersedia di:<https://glm.g-truc.net> [Diakses 16 Agustus 2017]

Saini, S., Chung, P.W.H., Dawson, C.W., 2011. Mimicking Human Strategies in Fighting Games using a Data Driven Finite State Machine. 6th IEEE Joint International Information Technology and Artificial Intelligence Conference.

Simple DirectMedia Layer, 1998. Tersedia di:<https://www.libsdl.org> [Diakses 16 Agustus 2017]

The OpenGL Extension Wrangler Library, 2002. Tersedia di:< http://glew.sourceforge.net> [Diakses 16 Agustus 2017]

What is a Game Engine?, GameCareerGuide, 2008. Tersedia di:<https://www.gamecareerguide.com/features/529/what_is_a_game_.php> [Diakses 16 Agustus 2017]

Wright, David R., 2005. Finite State Machine. *CSC215 Class Notes*. David R. Wright website, N. Carolina State Univ.

Yan-hui, W., Xia-xia, Y., He-Jin, 2011. Design and Implementation of the Game Engine Based on Android Platform. International Conference on Internet Technology and Applications(iTAP).