Available online at www.sciencedirect.com

ScienceDirect

Procedia Computer Science 235 (2024) 2996–3006

Procedia Computer Science

www.elsevier.com/locate/procedia

International Conference on Machine Learning and Data Engineering (ICMLDE 2023)

Development of Two Dimension (2D) Game Engine with Finite State Machine (FSM) Based Artificial Intelligence (AI) Subsystem

Abidemi Emmanuel ADENIYI[a], Biswajit BRAHMA[b], Marion Olubunmi ADEBIYI[c], Joseph Bamidele AWOTUNDE[d], Rasheed Gbenga JIMOH[d], Enoch OLASINDE[c], Anjan BANDYOPADHYAY[e,*]

[a]*Department of Computer Science, Bowen University, Iwo, Osun State, Nigeria*
[b]*McKesson Corporation, USA 32559 Lake Bridgeport St, Fremont, CA 94555 USA*
[c]*Department of Computer Science, Landmark University, Omu-Aran, Nigeria*
[d]*Department of Computer Science, Faculty of Information and Communication Sciences, University of Ilorin, Nigeria*
[e]*School of Computer Engineering, Kalinga Institute of Industrial Technology, Bhubaneswar, India*

Abstract

With Al becoming more and more relevant in today's world, this project aims to develop a 2D game engine with an Al subsystem for state-driven agents, which is rarely implemented by a lot of 2D engines out there. In this study, a 2D game engine was designed with an FSM (Finite State Machine)--based AL subsystem using state-driven game agents. The engine was implemented using the Javascript programming language and the WebGL 2.0 graphics library/API. It is targeted at web-based games/simulations. Components and subsystems include physics, audio, math, rendering, and AI (based on finite-state machines). The FSM-based AI subsystem is a solution aimed at reducing the ambiguity and performance hits associated with creating 2D game AI in the naive approach. The AI subsystem creates an interface for 2D games to be created with a common paradigm, and simulated with a great level of realism. The state machine used in this study is used to represent a variety of behaviours, such as wandering, attacking, and fleeing. The following conclusions were drawn as regards the impact of the AI approach used on rendering performance; the naive approach to implementing game AI is also compared with the FSM approach in terms of rendering (frames per second/ FPS, or frame rate). With the naive approach (vector math) being used to implement AI, there was a drop in the rendering frame rate to 50 FPS. The FSM approach didn't affect the frame rate, which is usually at 60FPS. With a well-developed FSM (Finite State Machine), game agents can transition between states easily as a response to user input or stimulus from the game environment. The proposed method was tested by creating a prototype game with the 2D game engine. The prototype game was a straightforward side-scrolling platformer featuring a cast of non-player characters (NPCs). This approach to implementing 2D game AI goes a long way toward improving performance and mitigating ambiguity in the code.

Engineering
Keywords: Artificial intelligence, 2D game engine, Finite state machine, software.

1. INTRODUCTION

A game engine is a graphical software that provides a complete solution to the development of games of any genre via providing tools and utilities which vary across engines, to power the development of these games [1,2]. They also aid the creation of all sorts of simulations and almost any other graphical content one comes across like 3D architectural designs, VR (Virtual Reality), and AR (Augmented Reality) [3]. It is a large system with components that form the building block for its operations. How well these components are developed goes a long way into affecting a lot of things in the engine's development phase; like maintainability, ease of use, performance and a few others [4].

To make the most of the visual hardware, a game for the Atari 2600, for instance, had to be created from scratch in the 1990s; currently, game developers for older systems refer to this fundamental display code as the kernel [5]. Before game engines, games were frequently developed as standalone entities. Other platforms provided more freedom, but even when the display was unimportant, memory constraints frequently prohibited the data-heavy designs required by engines. Even on simpler systems, there wasn't much that could be used between games. A significant portion of the code had to be abandoned later due to the swift development of arcade hardware, which controlled the industry at the time, while subsequent generations of games employed whole new game ideas that consumed more resources. As a result, the vast majority of 1980s game designs relied on a hard-coded rulebook with few levels and illustrative data. It has been customary for video game firms to create internal engine technologies for use with third-party applications ever since the age of arcade video games.

There were several 2D game production systems created in the 1980s enabling independent video game development, while third-party game engines were uncommon until the introduction of 3D computer graphics in the 1990s. Games were made with this strategy in mind, with the engine and content being created independently, such as id Software's Quake II! Arena and Epic Games' 1998 Unreal. This approach to game development has made it easier to make changes to an engine without disrupting a game's functionality if not needed. It has also improved maintenance, and re-usability, as engines can be built differently from games, and monetized.

Artificial intelligence (AI) is used in video games to provide receptive, adaptive, or autonomous behaviours in non-player characters (NPCs) [6]. Since video games were first developed in the 1950s, they have been significantly influenced by artificial intelligence. Video game scientific AI is a unique subfield in AI [7]. In contrast to machine learning or decision-making, it enhances the gaming experience. The idea of Al opponents was widely adopted during the heyday of arcade video games in the form of graded complexity settings, diverse movement patterns, and in-game events that relied on human engagement. Pathfinding and decision trees are two technologies commonly employed in modern games to control the behaviour of NPCs. AI is typically used in procedures that are not immediately evident to the user, such as data mining and procedural content production [8]. The foundation of an AI subsystem would require a finite state machine (FSM) [9].

Historically, mathematicians have employed tightly codified tools called finite-state machines to address difficulties, machines are used. Most people are probably familiar with Alan Turing's hypothetical machine, the Turing machine, which he discussed in his 1936 essay "On Computable Logic." This was a device that read, wrote, and erased characters on an endless loop of tape, foreshadowing current programmable processors. Thankfully, as AI developers, we can substitute an intuitive explanation for the formal mathematical explanation of a finite state machine: A device or model of a device with a limited amount of states that it can be in simultaneously and that can respond to input by changing its state or by inducing a result or action is known as a finite state machine. A finite-state computer can only stay in one state at a time. As a result, the goal of a finite state machine is to deconstruct the actions of an object into smaller, easier-to-manage "chunks" or states [10].

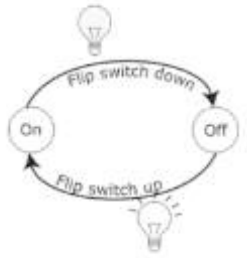


Figure 1. A light switch is an example of a finite-state machine. (Note that in Europe and many other areas of the world, the switches are inverted)

FSM provide a mechanism for state management for game agents and holds a great deal of relevance to an AI subsystem for game engines. It is called finite because they are a finite number of states that a game character can have. An FSM in design, holds the current state, previous state and global state of a game agent/character. A state is a representation of a game character's being at a given time. It can change due to inputs from a player or other stimuli from the game environment. Each character or agent would have an instance of an FSM as an attribute that manages how the state transition occurs based on a given input of energy trading considered how the isolated MG system could be designed for the annual cost minimization of meeting energy demand. In [9], an islanded MG is presented to implement centralized EMS. The optimization is performed for overall cost minimization and to optimize the operation of power generation so that reliable, clean and secure electricity is achieved. In [10], mixed-integer non-linear programming is proposed to solve EMS in an MG connected in islanded mode. The idea is to maximize the utilization of DERs to obtain the cheapest price. Helal, et al. in [11] present hybrid AC/DC MG for remote communities. The new energy management is used for the scheduling of MG generation technologies, which is formulated using (MINLP) and a microgrid central controller by minimizing the cost of distributed generation units. In [12], EMS is developed for islanded building micro-grid networks (BMGs). It uses adjustable power, which minimizes the operation cost of the network. Abdul Mohsen, et al. [13], designed the operation of an islanded microgrid using a two-stage strategy. The hierarchical approach is used to design the strategy to achieve optimal operation in real time. The above methods selfishly minimize MG cost but do not consider mutual cost minimization.

There are two genres of game engines, namely, 2D engines and 3D engines. Most 3D engines are capable of handling 2D simulations but the focus of this study is on 2D engines and how intelligent behaviour is implemented in NPCs (Non-Playable Characters). In this study, a 2D game engine is developed with an FSM (finite state machine) based AI subsystem. The FSM-based AI subsystem provides flexibility for state management in-game AI, reduces ambiguity in AI code and significantly aids performance boost as opposed to other approaches [11-12]. The study's novelty is that it enables more complicated and dynamic AI behaviour than previous AI methodologies. FSMs are a straightforward but effective method for modelling state-based systems. They may be used to depict the behaviour of any system that has a finite number of states. There are several current game engines, however, most of them are developed for 3D games. A 2D game engine would address a market void by providing developers with a tool particularly suited for generating 2D games.

The rest of this study is further divided into five sections. Section 2 gives related works. Section 3 gives the methodology used to accomplish the implementation of the study. Section 4 discusses various results obtained while section 5 concludes the study.

2. RELATED WORKS

A gaming engine's concept [13] is often rather simple to understand. It provides a framework for routine input, physics-related computing, and rendering, allowing developers (including artists, designers, scripters, and other programmers) to focus on the intricacies that give their game its distinct personality. Engines, in actuality, are a collection of reusable components that may be altered to bring a game to life. There are numerous differences between a game and a gaming engine. Rendering, loading, movement, object collision detection, physics, inputs, GUI, and AI are all included in some of an engine's primary components. On the other hand, a game's actuality is determined by its content, distinctive characters and settings, the reason why things collide, how real-world items behave, etc. Thanks to game engines, which function as middleware, the same game may be played on a variety of platforms, including gaming consoles and personal computers, with just small source code modifications.

FSM provide a mechanism for state management for game agents and holds a great deal of relevance to an AI subsystem for game engines. It is called finite because there are a finite number of states that a game character can have. An FSM in design, holds the current state, previous state and global state of a game agent/character. A state is a representation of a game character's being at a given time. It can change due to inputs from a player or other stimuli from the game environment. Each character or agent would have an instance of an FSM as an attribute that manages how the state transition occurs based on a given input. Sung et al., [13], in his work, developed a game engine targeted at the web. Several simulations and a game were made to illustrate its functionalities. They took an inheritance approach to the development and management of the game entities. Artificial intelligence techniques can be utilized to assist in manufacturing system design. For instance, Stocker, Schmid, and Reinhart [14] automate the creation of vibratory bowl feeders (VBFs) using reinforcement learning. The existing manual trial-and-error approach is mirrored by reinforcement learning employed to discover the most efficient trap layout. Rikkonen [15] implemented a flexible artificial intelligence system for a video game in his research. The research created and merged existing AI technologies for the Unity gaming engine. The study's goal was to utilize a visual editor to generate and alter AI behaviours without having to write any actual programming code [16]. The system's foundation was selected to be the behaviour designer. The outcome demonstrates a functional system that has reduced the barrier to modifying and creating AI behaviours for members of the development team who have little to no programming knowledge. Bourg & Bywalec [17], proposed a model for a physics engine design. The physics engine was designed to use C++. The engine focuses on rigid-body simulations using the Jacobian approach to process objects in contact. Forces are used to resolve contact between objects rather than Impulses. De Vries [18], proposed the use of batch rendering to optimize rendering and reduce the number of draw calls. Claussmann et al., [19] proposed a more modern approach to FSM creation; using a finite state machine for each game agent or character.

Millington [20], the physics engine is designed to use C++ and the major aspect of physics considered is mechanics, as it is the most relevant field of physics to graphical simulations. The engine focuses on rigid-body simulations using the iterative approach to process objects in contact. Impulses are used to resolve contact between objects rather than forces. In short, the physics engine developed is a rigid-body, iterative and impulse-based engine. Buckland, [21] proposed several ways for creating an FSM. The naive way, using if-else statements, the switch-case use-case, having each game agent or character manage its current state - and finally, a finite state machine for each game agent or character. Nystrom, [22] proposed the more modern approach to FSM creation; using a finite state machine for each game agent or character. He also proposed the use of a stack-based finite state machine other than the conventional approach. Sutherland, [23] in his work, developed a game engine targeted at mobile devices (Android). Several simulations and a game were made to illustrate its functionalities. He used the ECS (Entity Component System) approach to the development and management of the game entities.

2.1 Summary of Related Works

According to the research, FSM was a simple and effective approach to generating dynamic and difficult gameplay. It is also a flexible approach that may be used to generate a wide range of behaviours. FSM is a viable approach for designing 2D games. More study, however, is required to fully explore the possibilities of FSM in this area.

3. MATERIALS AND METHODS

In this section, When designing a 2D game engine or any graphics engine in general, one has to take into consideration what graphics library/API to use to power the engine and how to effectively abstract the graphics API. This project uses webGL 2.0 (web graphics library) and JavaScript, which is a programming language known for the web. The webGL serves as an interface between the engine and the GPU as do all graphics libraries/APIs. In deeper terms, the webGL API facilitates the engine's communication with the GPU and all one has to do is access methods provided or supported by a GPU via some library(in this case, the browser API). Then, use these abstracted functionalities to render graphics to the screen. This abstraction has made it easier for people to program GPUs seamlessly and not have to worry about coding in lower-level languages to achieve this. Using the webGL library, the engine is developed and then an interface to communicate with the engine's functionality is provided. This interface is what gameplay programmers use to build games with the engine without knowledge of how the engine communicates with the GPU. This makes game development easier and drastically reduces development time. The AI subsystem in this study is also developed on FSM. AI states can easily be created, managed, and transitioned between with this development approach (figurative references can be made to Figure 2.

The workflow of the engine developed in this study is highlighted in Figure 3. For starters, every game to be developed with the engine will have a scene or scenes as its constituents. Every game also has a starter scene that commences the game. In the case of one scene, the single scene is the starter scene. In the case of multiple scenes, the first scene will be the starter scene and then link to other scenes as needed. Before a scene gets initialized, the engine's core is initialized, making necessary resources available for the scene. After a successful initialization of the engine's core, the scene is initialized and the resources needed to run the scene are loaded into memory, after which the game loop commences, and runs in a continuous loop, updating, and rendering the active scene per iteration. For the game loop to stop execution, a scene would have to stop running (this is the same as stopping a game in execution). Whatever button needs to be pressed to stop the game's execution is left to the gameplay programmer to define when implementing the game logic. Once a scene stops execution, the game loop stops and seeks to switch to another scene just in case the game has more than one scene/level. In the absence of another scene, the current scene triggers its unload method which removes all game resources in memory associated with it. Finally, the engine's cleanup method is activated and causes all allocated resources to be freed.

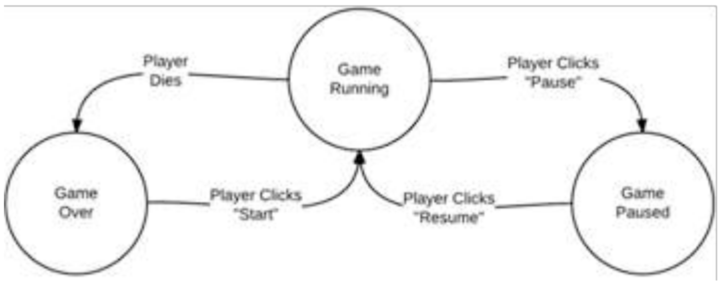


Figure 2 Finite State Machine

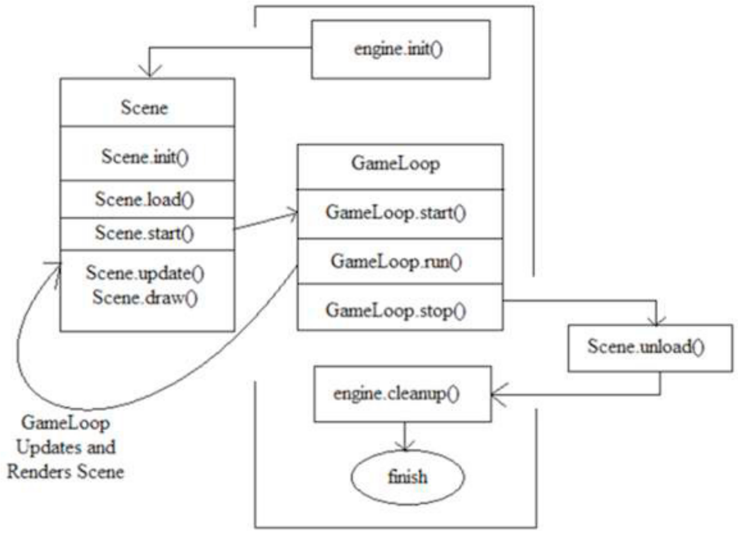


Figure 3. A basic workflow of the developed engine

3.1 The Math Subsystem

Math in computer graphics and fields built upon it like gaming, modelling and animations, graphics design, movies (where VFX is involved) and others are generally based on linear algebra. This engine uses an external library called gl-matrix to handle mathematical operations in the math subsystem. The library is built to facilitate the development of graphical applications in the browser. Not all aspects of linear algebra are relevant to computer graphics but the relevant ones form the basis for mathematical computations in computer graphics.

3.2 The Rendering Subsystem

The rendering pipeline is the stages involved in rendering a single pixel to screen in a graphical application. Figure 4 shows the stages of the rendering pipeline in WebGL 2.0.

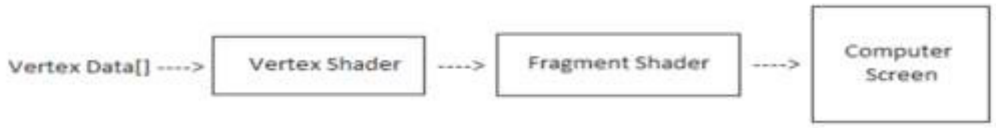


Figure 4. Stages of rendering pipeline in webGL2.0

Shaders are mini-programs that run in the cores of our GPU (Graphics Processing Unit). The rendering pipeline with shaders in webGL 2.0 comprises two stages; the vertex shader and the fragment shader, and the engine built in this project takes advantage of the two stages to achieve rendering. To render anything to the screen, vertices that represent the structure (e.g. a triangle), have to be defined. Then, the vertices are sent to the vertex shader after which a primitive assembly is performed (the construction of the given structure via graphics primitives such as points, lines, triangles, and quadrilaterals). After the primitive assembly, rasterization (the mapping of the graphics primitives to the screen via coordinates) occurs. Finally, the fragment shader executes for each pixel to be rendered to the screen. So if the engine were to render to a 1280 by 720 display, the fragment shader would run 1280*720 = 921,600 times for each pixel. A quick note, There are other aspects of the rendering pipeline in webGL that are not of relevance to this study. The structure of the renderable class is displayed in Figure 5.

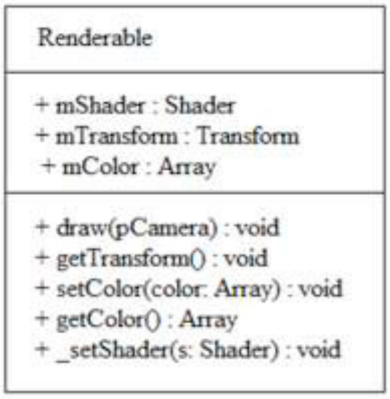


Figure 5. The interface/structure of the Renderable class

The rendering subsystem in the engine is responsible for sending data needed to render a particular scene for a game to the GPU. The rendering technique/approach used in this project is the forward rendering technique. The forward rendering technique involves passing all the data down the graphics pipeline at once as opposed to the deferred rendering, where the rendering is delayed until certain calculations (e.g. lighting calculations) are completed at a certain stage, and then, the scene is rendered.

This process occurs each loop or each time a frame has to be drawn to the screen via the game loop. Figure 6 shows a briefing of the post operations that occur after invoking the draw method of a renderable object. Renderable objects can also have their visibility altered (made invisible). Examples of such invisible objects are triggers which are used to trigger certain actions in a game scene. For example, the cutscene for a particular game level. As stated earlier, the rendering subsystem is built on renderable objects and these renderable objects simply abstract the process of passing data down the graphics pipeline.

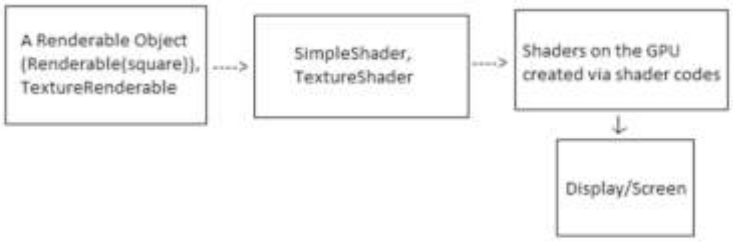

A Renderable Object (Renderable(square)), TextureRenderable
SimpleShader, TextureShader
Shaders on the GPU created via shader codes
Display/Screen

Figure 6. The rendering workflow of a renderable object.

3.3 FSM implementation in game AI

Game AI is any game character that is created to simulate intelligence. It can vary from bosses (enemies in a game) to regular characters that contribute to a game story. Every Game AI can have one or more states and these states are defined by classes that inherit from the State class. For example, a game AI usually has a patrol state and possibly a chase (if it is a boss in a game level), or evade state. The patrol state and chase state will have to be created from the State class. The defined state classes will also have to override the execute method they inherit from the State class. The execute method is where the game logic needed to process the movement of the game AI is implemented. The enter and exit methods can also be overridden if certain actions need to be performed by a game AI on a call to each respectively. Then, instances of these defined state classes (patrol and chase) will be used for a given game AI. Each game AI instance will have to implement an attribute that is an instance of a subclass of the State class (like the patrol and chase states mentioned earlier). The game object will also implement a method called changeState which is used to switch between states.

4. RESULTS AND DISCUSSION

To put the model to the test, a full-fledged action role-playing game is developed. This section presents the sequence of processes in the development of a game with a designed model which includes screenshots of the codes written in Visual Studio code.

4.1 Development of a 2D role-playing game

A game titled The Adventures of Dye is developed to test the capabilities of the engine. The game will have three different scenes, one for the welcome page/home screen and the latter two, for the actual game levels. The game would feature a single player with several AIs(bosses/enemies). The bosses would attack the player when in a certain range and leave him be if he can successfully evade them.

4.2 Creation of game characters

All discussed game characters in this section are entities derived from the GameObject class and then modified to illustrate characters. The main/hero character (protagonist) for the game is named Shaw. She can run and jump basically. Her abilities are limited since intelligent behaviour is our main focus for this project.

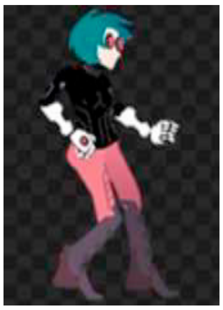


Figure 7. The Hero Character

There are quite several enemies in this game and its' own uniqueness. They are discussed in the order of their intelligence. Figure 8 shows all the enemies featured in this game.

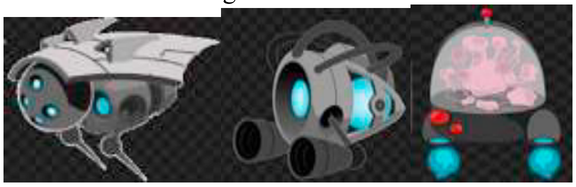


Figure 8. The Enemies/Bosses

The leftmost enemy is the simplest of all and exhibits no form of intelligence. The middle enemy is a notorious one and features an ability (shooting balls) that is harmful to the player. The rightmost enemy is the smartest and takes advantage of the developed model in this study.

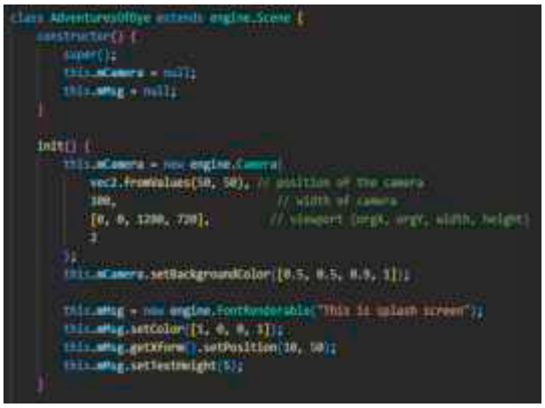
4.3 Creation of levels

The organization of the code files for developing the game is explained in the prevision section. To begin with, a scene file is created and named the adventures_of_dye.js. This scene file holds a Scene class called AdventuresOfDye. This scene will display the welcome message to a player and ask for the player to press the "spacebar" to start playing.

Figure 9. The Welcome Screen for the AdventureOfDye Scene

The newly created scene file is then imported into our root HTML file; this process was discussed earlier in the previous section.

The init method initializes the 2D camera for the scene and creates a FontRenderable object which is used to display text in the game's UI. Some properties of the created FontRenderable object are also altered to adjust its position and size. Any visual reference needed can be made to Figure 10. The updated method is used to define the input logic required to start the game. Once the space-bar button is pressed, the AdventureOfDye scene is stopped and the next game level is loaded and rendered.



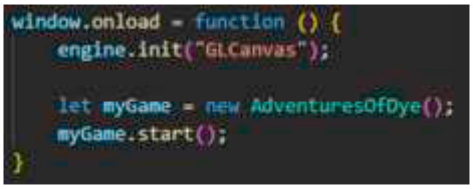
```
class AdventuresOfDye extends engine.Scene {
    constructor() {
        super();
        this.mCamera = null;
        this.mMsg = null;
    }

    init() {
        this.mCamera = new engine.Camera(
            vec2.fromValues(50, 50), // position of the camera
            100,                     // width of camera
            [0, 0, 1280, 720],       // viewport (orgX, orgY, width, height)
            2
        );
        this.mCamera.setBackgroundColor([0.5, 0.5, 0.9, 1]);

        this.mMsg = new engine.FontRenderable("This is splash screen");
        this.mMsg.setColor([1, 0, 0, 1]);
        this.mMsg.getXform().setPosition(10, 50);
        this.mMsg.setTextHeight(5);
    }
```

Figure 10. AdventuresOfDye Scene Class

The draw method is used to trigger the rendering of the FontRenderable object(this. msg) to the screen via the created 2D camera(this.mCamera). The next method of the class is also implemented to switch to the first level of the game.



```
window.onload = function () {
    engine.init("GLCanvas");

    let myGame = new AdventuresOfDye();
    myGame.start();
}
```

Figure 11. The commencement of the game

Figure 12 shows how the game commences after a successful window load. The engine is first initialized, and then, an instance of the level is created and started. Once the user presses “spacebar” as instructed, the game commences by switching to the next level.

The next scene commences when the user presses the spacebar and the figure below shows what the level looks like. This scene is called Level_01 which signifies it’s the first level in the game

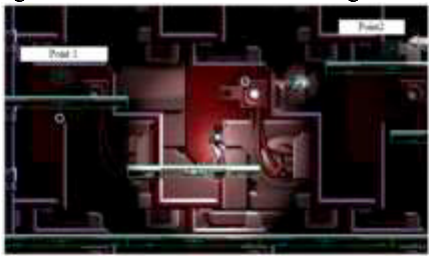


Figure 12. Level_01 for the game

The objectives of the level are simple; the player is tasked with reaching some points which will open a path for him to the next level. Some of the points are highlighted in Figure 13; once all the points are reached, a doorway is opened for the player to move to the next level as seen in Figure 13. The enemies in this level(Level_01) don’t implement any smart behaviour and won’t give the player any attention except when she comes in contact with them or their emitting bullets.

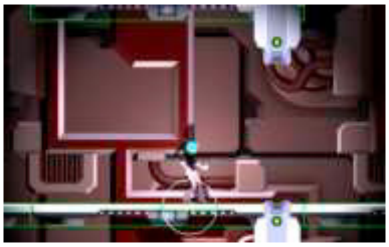


Figure 13. The doorway to the next level

When the player works through the doorway the next and final scene for the game commences. The enemies in this level are made intelligent via the developed model in this project and attack the player when he is within a certain range relative to them

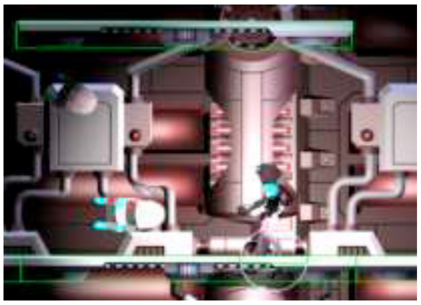


Figure 14. Level_02 for the game; also shows the AIs that implement the FSM approach

4.4 Creation of AI states

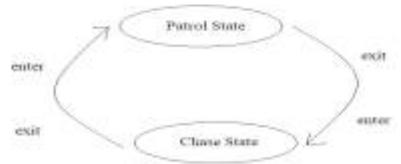


Figure 15. The state execution model for an AI

Figure 15 shows the state execution model for an AI. Every AI for this game have two states; one for patrol (the default state), and the other for chasing. The patrol state is the idle state for an AI. The chase state is triggered once a player gets within a certain range of the AI. To implement the discussed states, two classes are defined to represent these states with each deriving from the State interface/class.

4.5 Performance Evaluation of the Model

With the use of the AI subsystem, similar AI behaviours (states) that game AI shares can be implemented with the same state classes. Via this mechanism, behavioural logic for game AI wouldn't have to be rewritten for each game AI that exists, rather, state classes are defined to handle this behavioural logic and are shared among every game AI that needs to implement such behaviours. Another crucial benefit of this model is that AI code can be written entirely outside of the update method of AI game objects and the currently active scene; this gives an advantage of reducing the code base needed to implement intelligent behavior for game AI, thereby, mitigating ambiguity and improving code readability.

The naive approach to implementing game AI is also compared with the FSM approach in terms of rendering (frames per second/FPS or frame rate). With the naive approach (vector math) being used to implement AI, there was a drop in the rendering frame rate to 50FPS. The FSM approach didn't affect the frame rate which is usually at 60FPS. From the derived data, the following conclusions were drawn as regards the impact of the AI approach used on rendering performance.

naiveFrameCount = number of frames drawn per sec in the naive approach

fsmFrameCount = number of frames drawn per sec in the FSM approach

performance boost(%) = (fsmFrameCount - naiveFrameCount) / 60(expected frame rate) * 100

derivedPerformanceBoost = (60 - 50) / 60 * 100 = 16.6667 %

The derivedPerformanceBoost is calculated using the number of frames drawn per sec using the naive approach, and comparing it with the number of frames drawn per sec using the FSM approach to simulate intelligence. From the calculations, it is seen that there is a significant performance boost of 16.667% when using the FSM approach as opposed to the naive approach.

5. CONCLUSION

The development of a 2D game engine with an FSM base AI subsystem using methods similar to conventional game development is a significant contribution to the field of game development. This unique approach to AI enables more intricate and dynamic behaviour than previous AI approaches. The engine provides facilities for developing 2D games and simulations with a standard level of appealing visuals. The developed FSM-based AI subsystem also helps to speed up the implementation of game AI, easy state management and replication of intelligent behaviour in an efficient way. Game AI (bosses/enemies) created for game levels can all implement similar states when the need arises without having to rewrite the logic for each. This provides a simple and efficient way to manage their behaviors as any required changes can simply be made to the state class used to manage their operations. Therefore, a 2D game engine model with an FSM-based AI subsystem was designed and employed to replicate intelligent behaviour in-game AI. This approach to game AI had no significant impact on rendering performance in the engine and also gave room to designing game AI in a more defined and less ambiguous manner with a derived performance boost of 16.667%. Other approaches to creating FSMs should be examined. For example, a stack-based FSM (a finite state machine that handles states with a stack data structure) gives room to more sophisticated state management routines. This study is noteworthy because it has the potential to have a huge influence on the future of game development.

References

1. Gregory, J. (2018). Game engine architecture. AK Peters/CRC Press.
2. Foxman, M. (2019). United We Stand: Platforms, tools and innovation with the Unity game engine. Social Media+ Society, 5(4), 2056305119880177.
3. Adeniyi, A. E., Olagunju, M., Awotunde, J. B., Abiodun, M. K., Awokola, J., & Lawrence, M. O. (2022, July). Augmented Intelligence Multilingual Conversational Service for Smart Enterprise Management Software. In International Conference on Computational Science and Its Applications (pp. 476-488). Cham: Springer International Publishing.
4. Cowan, B., & Kapralos, B. (2017). An overview of serious game engines and frameworks. Recent Advances in Technologies for Inclusive Well-Being, 15-38.
5. Montfort, N., & Bogost, I. (2020). Racing the Beam: The Atari video computer system. MIT Press.
6. Ranjitha, M., Nathan, K., & Joseph, L. (2020). Artificial intelligence algorithms and techniques in the computation of player-adaptive games. In Journal of Physics: Conference Series (Vol. 1427, No. 1, p. 012006). IOP Publishing.
7. Schijven, M. P., & Kikkawa, T. (2022). Is there any (artificial) intelligence in gaming? Simulation & Gaming, 10468781221101685.
8. Oladipupo, M. A., Obuzor, P. C., Bamgbade, B. J., Adeniyi, A. E., Olagunju, K. M., & Ajagbe, S. A. (2023). An Automated Python Script for Data Cleaning and Labeling using Machine Learning Technique. Informatica, 47(6).
9. Linda Tucci. (2021). What is Artificial Intelligence (AI)? - AI Definition and How it Works. https://www.techtarget.com/searchenterpriseai/definition/AI-Artificial-Intelligence
10. Koutsomichalis, M. (2020). Weierstrass function, finite state automata, and l-system.
11. Oladipo, I. D., AbdulRaheem, M., Awotunde, J. B., Bhoi, A. K., Adeniyi, E. A., & Abiodun, M. K. (2021). Machine learning and deep learning algorithms for smart cities: a start-of-the-art review. IoT and IoE driven smart cities, 143-162.
12. Abiodun, K. M., Awotunde, J. B., Aremu, D. R., & Adeniyi, E. A. (2022). Explainable AI for fighting COVID-19 pandemic: opportunities, challenges, and prospects. Computational Intelligence for COVID-19 and Future Pandemics: Emerging Applications and Strategies, 315-332.
13. Sung, K., Pavleas, J., Munson, M., & Pace, J. Build Your 2D Game Engine and Create Great Web Games
14. Stocker, C., Schmid, M., & Reinhart, G. (2019). Reinforcement learning–based design of orienting devices for vibratory bowl feeders. The International Journal of Advanced Manufacturing Technology, 105(9), 3631-3642.
15. Rikkonen, J. (2017). Implementing a flexible artificial intelligence system for a video game: Case Northbound.
16. Awotunde, J. B., Folorunso, S. O., Jimoh, R. G., Adeniyi, E. A., Abiodun, K. M., & Ajamu, G. J. (2021). Application of artificial intelligence for COVID-19 epidemic: an exploratory study, opportunities, challenges, and prospects. Artificial Intelligence for COVID-19, 47-61.
17. Bourg, D. M., & Bywalec, B. (2013). Physics for Game Developers: Science, math, and code for realistic effects. "O'Reilly Media, Inc.".
18. DeVries, T. (2021). Towards High Fidelity Generation of Synthetic 3D Worlds (Doctoral dissertation, University of Guelph).
19. Claussmann, L., Revilloud, M., Gruyer, D., & Glaser, S. (2019). A review of motion planning for highway autonomous driving. IEEE Transactions on Intelligent Transportation Systems, 21(5), 1826-1848.
20. Millington, I. (2010). Game physics engine development: how to build a robust commercial-grade physics engine for your game. CRC Press.
21. Buckland, M. (2005). Programming game AI by example. Jones & Bartlett Learning.
22. Nystrom, R., 2014. Game Programming Patterns. 1st ed. s.l.:Genever Benning.
23. Sutherland, M. (2013). Four interviews with young Scottish students and graduates from a 1990s Scottish university who set up new-start computer games studios. The Computer Games Journal, 2(3), 77-102.