”ALEXANDRU-IOAN CUZA” UNIVERSITY

COMPUTER SCIENCE

THESIS PAPER

OpenGL Framework

author

Dragoş-Andrei Bobu

session: July, 2024

scientific coordinator

Dr. Cosmin Varlan, Lecturer

"ALEXANDRU-IOAN CUZA" UNIVERSITY

COMPUTER SCIENCE

OpenGL Framework

Dragoş-Andrei Bobu

Session July, 2024

Scientific Coordinator

Dr. Cosmin Varlan, Lecturer

Computer software operates as an organized array of specialized tools designed for distinct tasks. Similarly, a game engine constitutes a sophisticated software framework comprising diverse computer graphics components.

In contrast, a game editor empowers users to craft new tools seamlessly integrated with existing robust engines, thereby amplifying functionality and customization opportunities.

This article delves into essential components crucial for a game engine, exploring their underlying mechanisms: graphics rendering, physics simulations, and input handling. By unraveling these interactions within the framework, we uncover the intricate processes that underpin the creation of immersive and dynamic gaming experiences.

This paper explores foundational literature informing the development of a game engine architecture, focusing on three critical domains: mathematical frameworks, rendering engines, and C++ software infrastructure.

"Mathematics For Game Developers" by Christopher Tremblay introduces vector mathematics and geometric algorithms pivotal for physics and spatial computations.

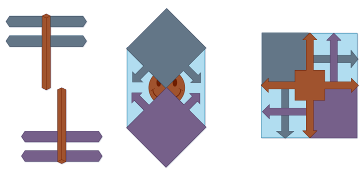
Donald Hearn's "Computer Graphics in C/C++" delves into rendering techniques like rasterization and shading.

"OpenGL SuperBible" by Jr. Richard S. offers modern OpenGL principles crucial for shader development and GPU-based rendering.

Additionally, "C++ Primer" by Bjarne Stroustrup guides C++ language features and best practices, shaping the engine's software backbone.

This exploration showcases how these resources collectively contribute to a robust game engine architecture, facilitating core systems like graphics rendering, physics simulations, and input handling, essential for immersive gaming experiences.

The thesis project comprises three primary components: an editor, a render engine, and a math engine, collectively forming a framework for game development and computational simulations.



Contents

Part I

Introduction

Motivation

The immersiveness of games could be significantly enhanced by integrating machine learning (ML) with the existing dialogue tree frameworks. Such integration has the potential to create more dynamic and responsive NPC interactions.

Introduction

Non-Player Characters (NPCs) serve as guides for players and act as extensions of the game designers within the game world. Consequently, it is crucial for NPCs to have fluid dialogue that maintains the illusion of choice without easily breaking it.

Currently, dialogue trees are the predominant solution for managing NPC interactions. While effective, dialogue trees can still feel somewhat rigid and may disrupt the immersion when players are limited to selecting from a set of predefined responses.

This implementation required extensive study across multiple fields
of computer science, including:

- Computer Graphics
- Game Development
- Machine Learning

Additionally, the following courses have provided essential tools and skills that have significantly contributed to the success of this project:

- Object-Oriented Programming, Data Structures, and Software Engineering
- Python Programming, Programming Language Principles, and Logics

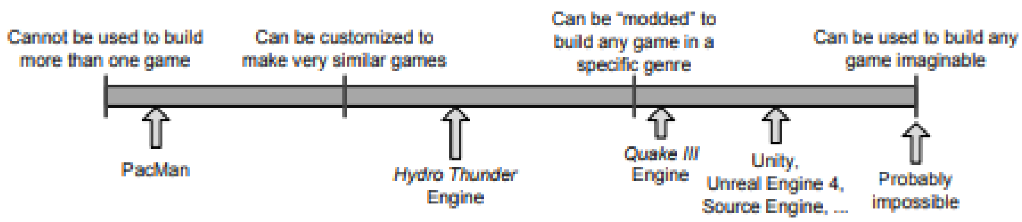


Figure 1.1. Game engine reusability gamut.

Game Engines are essential not only for games but also for other diverse applications.

Each Game Engine defines its functionality through its implemented components.

Problem Statement

Definition Setup:

Let S denote a software entity. The function isGameEngine(S) outputs true if S qualifies as a game engine based on the following criteria:

$$\text{isGameEngine}(S) \equiv \text{isFramework}(S) \wedge (\{S.\text{components}\} \subseteq \text{GE::E})$$

Where:

- isFramework(S) indicates that S is classified as a software framework.
- GE::E represents the set of components common to various game engines, defined as:

$$\text{GE::E} = \bigcup_{G_i \in \text{GameEngines}} \{(G_i, C_j) \mid C_j \in \{G_i.\text{components}\}\}$$

Scenario:

Assume a scenario where a company is developing a new game engine, GG, which includes the following components: MathEngine (M), RendererEngine (R), GUI Editor (E), and File Manager (FM).

1. Define GG as:

$$GG = \{M, R, E, FM\}$$

Tasks:

a) **Verification of Game Engine Status:**

Verify whether GG qualifies as a game engine based on the formal definition.

Solutions

Solution to Task (a):

To verify if GG is a game engine:

$$\text{isGameEngine}(GG) \equiv \text{isFramework}(GG) \wedge (\{GG.\text{components}\} \subseteq \text{GE::E})$$

Given $GG = \{M, R, E, FM\}$:

$$\text{isGameEngine}(GG) \equiv \text{isFramework}(GG) \wedge (\{M, R, E, FM\} \subseteq \text{GE::E})$$

Since GG includes components that are standard across various game engines (MathEngine, RendererEngine, GUI Editor, File Manager), and assuming isFramework(GG) is true (given the context), GG satisfies the criteria and is classified as a game engine.

Problem 2: Enhancing Game Engine with Machine Learning Capabilities

Formal Definition

Let GE denote a game engine with components $\{\text{Probabilities Engine}, \text{ID3 Generation}, \text{Modular Architecture}\}$.

Natural Language Understanding

Integrating machine learning capabilities into GE enhances its ability to predict outcomes, analyze data patterns, and support decision-making processes within games.

Machine Learning Integration

To enhance GE with machine learning capabilities:

- **Probabilities Engine**: Implement algorithms to calculate probabilities for in-game events and decisions.
- **ID3 Generation**: Develop decision tree algorithms for automated decision-making based on game state and player interactions.
- **Modularity**: Leverage GE's modular architecture to integrate popular Python solutions for machine learning, such as scikit-learn, TensorFlow, or PyTorch.
- **Adaptation**: Customize and adapt existing Python libraries to work seamlessly within GE's framework, ensuring compatibility and performance optimization.

Formal Proof

Define ML(GE) as the machine learning enhanced version of GE:

$$\text{ML}(GE) = \text{GE} + \text{ML\_Components}$$

Where ML_Components includes modules and functionalities for machine learning integration tailored to GE's requirements.

Scope

The project aimed to develop a foundational game engine and enhance it with adaptable machine learning functionalities. Key components included the rendering engine, math engine, editor, and file manager, pivotal for shaping the engine's capabilities.

Engine Components

Rendering Engine

Central to visualizing game worlds, our rendering engine efficiently handles complex graphics tasks with modern techniques, ensuring immersive and smooth gameplay experiences.

Math Engine

As the computational powerhouse, the math engine supports physics simulations, AI behaviors, and real-time interactions, enhancing realism and interactivity through robust algorithms.

Editor

The versatile editor streamlines game content creation and modification with an intuitive interface and extensive customization, fostering creativity and productivity among developers.

File Manager

Critical for efficient data management, the file manager organizes game assets and supports version control, ensuring seamless deployment across platforms.

Integration of Machine Learning

Integrating machine learning capabilities extends the engine's functionalities, enabling seamless adaptation of ML models and algorithms. This enhances gameplay dynamics, AI behaviors, and player interaction, driving innovation in interactive entertainment.

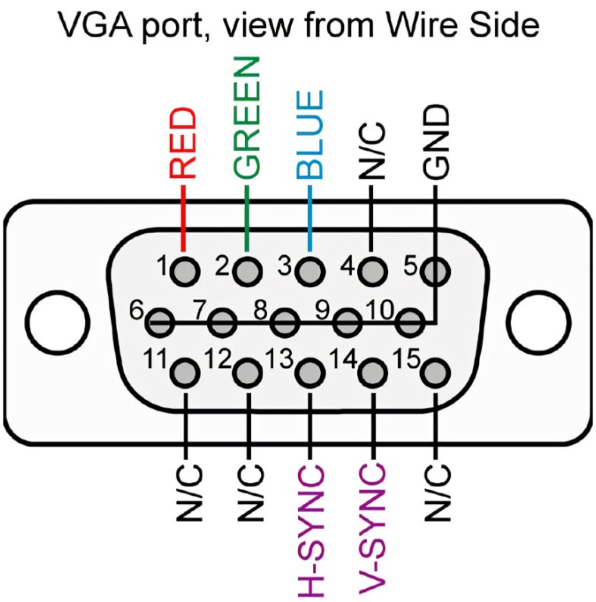
From RGB LEDs to VGA

- **RGB LED:** Basic color representation with R, G, B channels.
- **VGA Protocol:** Standard for analog video output from computers.

Communicating with VGA Protocol

The VGA protocol is implemented using various pins on a connector. It supports several analog signals for color and synchronization. Here's a simplified overview:

Pinout Diagram:



Signal Overview:

- **Red, Green, Blue (RGB):** Analog signals for color intensity.
- **Horizontal Sync (HSYNC):** Synchronizes horizontal lines.
- **Vertical Sync (VSYNC):** Synchronizes vertical frames.
- **Analog Grounds:** Reference points for signals.

Algorithm for Displaying an Image using VGA

To display an image using the VGA protocol, the following steps are typically involved:

1. **Initialize VGA Controller:** Set up registers for resolution, color depth, and synchronization timings.
2. **Generate Horizontal Sync Signal (HSYNC):** Send pulses to synchronize the start of each line.
3. **Generate Vertical Sync Signal (VSYNC):** Send pulses to synchronize the start of each frame.
4. **Output RGB Signals:** For each pixel in the image:
   - Calculate appropriate RGB voltages based on the color information of the pixel.
   - Output analog signals through the corresponding RGB pins.
5. **Repeat for Each Frame:** Continuously update the screen by repeating the above steps for each frame.

OpenGL: Revolutionizing Graphics Rendering

The rendering engine is built on top of OpenGL, which adheres to the PHIGS standard. In the following, we will explore what PHIGS represents.

PHIGS

Structure Definition

$$\text{Structure } S = \{e_1, e_2, \ldots, e_n\}$$

$$\text{where } e_i \in \{\text{Primitive, Attribute, Reference to another structure}\}$$

Structure Store Definition

$$\text{Structure Store } SS = \{(ID_1, S_1), (ID_2, S_2), \ldots, (ID_m, S_m)\}$$

$$\text{where } ID_i \text{ is the identifier of structure } S_i$$

Workstation Definition

$$\text{Workstation } W : SS \to \text{Rendered Image}$$

Output Primitives Definitions

$$\text{Point } P = (x, y)$$

$$\text{Line } L = \{P_1, P_2\} = \{(x_1, y_1), (x_2, y_2)\}$$

$$\text{Polygon } G = \{P_1, P_2, \ldots, P_k\} = \{(x_1, y_1), (x_2, y_2), \ldots, (x_k, y_k)\}$$

$$\text{Text } T = (P, \text{string}) = ((x, y), \text{string})$$

Interaction Example

```
// Define a structure with primitives
structure_id = 1;
open_structure(structure_id);
add_primitive(POINT, {x: 10, y: 20});
add_primitive(LINE, {start: {x: 0, y: 0}, end: {x: 100, y: 100}});
close_structure(structure_id);

// Update structure store
structure_store.add({id: structure_id, structure: structure});

// Render on workstation
workstation.render(structure_store);
```

Game Engines

Game engines are comprehensive software frameworks designed for the development and creation of video games. They provide essential tools and libraries for rendering graphics, processing physics, managing assets, and scripting game logic, allowing developers to focus on creating engaging and immersive experiences. A robust game engine is integral to the efficiency and success of game development projects.

Key Features of Game Engines

Rendering Engine

The rendering engine is responsible for drawing graphics on the screen, handling everything from 2D sprites to complex 3D environments. Advanced rendering engines support features such as lighting, shading, reflections, and particle effects to create visually stunning scenes.

Physics Engine

The physics engine simulates real-world physics to provide realistic movement and interactions between objects in the game world. This includes collision detection, rigid body dynamics, fluid dynamics, and soft body physics.

Scripting and AI

Scripting languages and AI systems are crucial for defining game behavior, character actions, and non-player character (NPC) intelligence. They allow developers to create complex interactions and behaviors without deep programming knowledge.

Audio Engine

The audio engine manages sound effects, music, and voice acting, ensuring they are synchronized with the gameplay and enhance the overall immersive experience.

Asset Management

Efficient asset management tools within the game engine help organize, store, and retrieve game assets such as textures, models, animations, and sounds. This is essential for maintaining a smooth workflow and ensuring all assets are readily accessible.

Licensing and Intellectual Property

Developing a game engine involves critical licensing and intellectual property (IP) considerations. Ensuring compliance with licenses for third-party libraries, assets, and tools, as well as protecting proprietary components, is essential to avoid infringement and unauthorized use.

Open Software vs. Closed Software

Open Software

Open-source software, such as P5.js, provides publicly available source code that can be freely used, modified, and distributed. This fosters a collaborative and innovative community, encouraging learning and experimentation across various fields.

Closed Software

Proprietary software, like Rockstar's RAGE engine used in the Grand Theft Auto series, restricts access to its source code and limits its use, modification, and distribution. This ensures that Rockstar retains exclusive rights and maintains a competitive edge.

Figure 1: Vectorial Math in RAGE Engine

Hybrid Models

Unity represents a hybrid model, offering a free version with basic features and an open API, while advanced features require paid licenses. This approach balances accessibility with commercial viability, supporting widespread use and continuous innovation.

Open Software in this Project

This project embraces open-source principles, making the game engine's source code publicly available to foster innovation and customization. This openness enhances the engine's versatility and encourages a community of contributors to drive its evolution.

Machine Learning Integration

It is common for companies to develop their own in-house game engines and subsequently build their applications on these platforms. However, the integration of pre-installed machine learning components is less prevalent.

Typically, integrating machine learning into games involves installing complex extensions over pre-existing game engines.

inWorld AI

InWorld AI is an example of such a service. It provides installable extensions for popular game engines and offers an interface for communicating with pre-trained OpenAI agents. This solution is excellent for facilitating human-to-AI communication, making it ideal for dialogues and NPC integration.

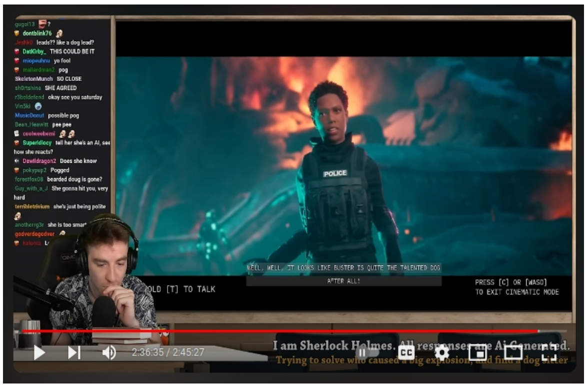


Figure 2: inWorld Ai Demo

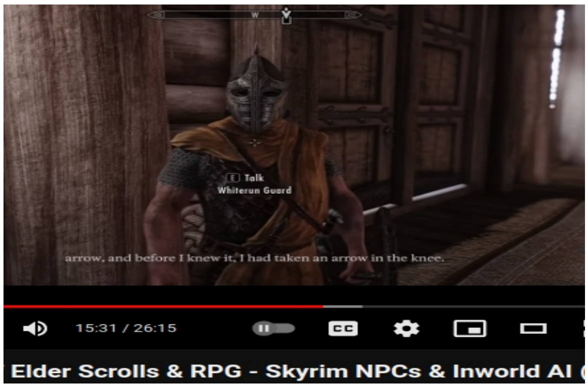


Figure 3: inWorld Ai Demo

Interactive Simulacra of Human Behavior

Another notable example is this research paper that successfully trained multiple ML agents to interact within a pre-created world. These agents are capable of remembering conversations and interactions and can even organize activities amongst themselves.

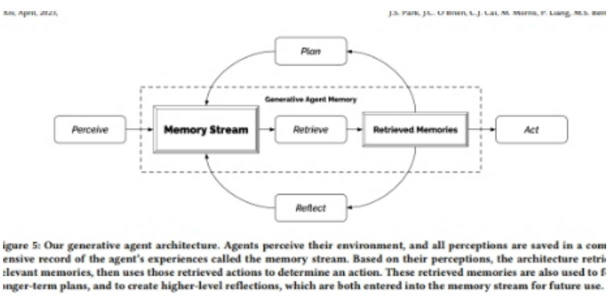


Figure 4: Data structure used for memory management

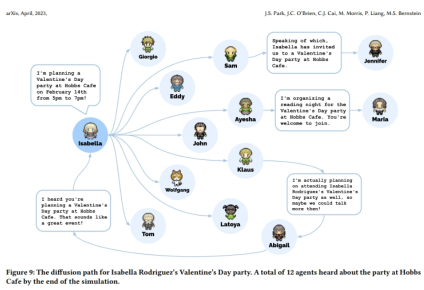


Figure 5: One Agent organised a birthday party

By integrating these advanced machine learning solutions, game developers can significantly enhance the immersion and interactivity of their NPCs, creating more engaging and dynamic gameplay experiences.

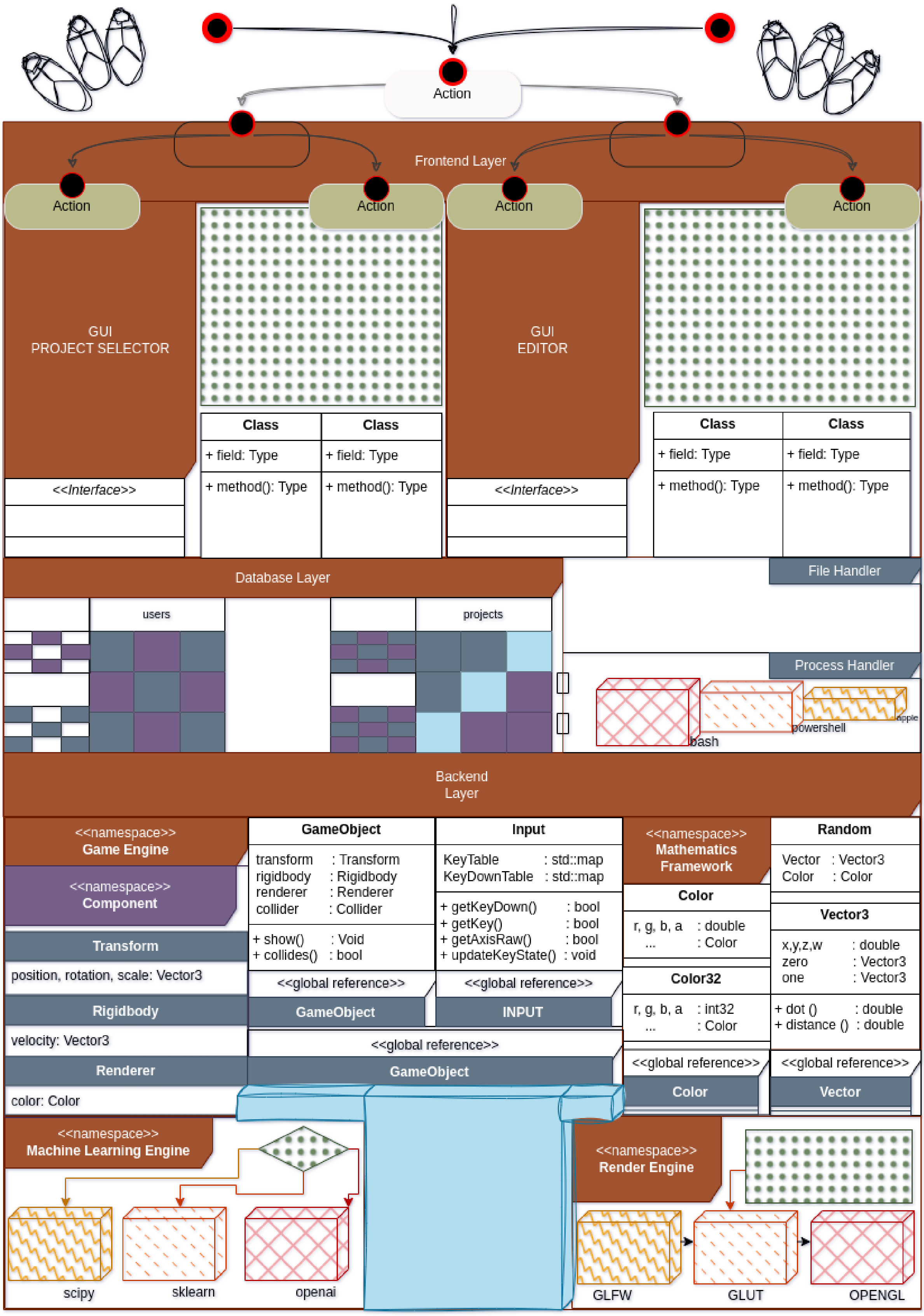
To address the absence of game engines with inherent machine learning capabilities, I propose the development of a new game engine designed from the ground up with ML integration as a core feature. This engine will enable developers to create more immersive and interactive experiences by seamlessly incorporating machine learning into their game development process.

Key Features of the Proposed Game Engine

- **Built-In Machine Learning Frameworks:**
  - The engine will come pre-integrated with popular machine learning libraries such as TensorFlow, PyTorch, and OpenAI's GPT. This eliminates the need for complex extensions and allows developers to leverage ML capabilities directly within the engine.
- **Easy Integration with Existing ML Models:**
  - Developers will be able to import pre-trained models easily and use them to enhance NPC behaviors, generate dynamic content, and more. This feature streamlines the process of integrating sophisticated AI into games.
- **Real-Time Learning and Adaptation:**
  - The engine will support real-time learning, enabling game entities to adapt based on player interactions. This creates a more dynamic and responsive game environment.
- **Voice Interaction Capabilities:**
  - With built-in support for microphone input, developers can create NPCs that engage in live dialogue with players, enhancing the realism and immersion of the game.
- **Compatibility with Popular Game Development Practices:**
  - Drawing inspiration from industry-leading platforms like Unity and p5.js, the engine will offer a user-friendly interface and robust documentation. This ensures that developers can transition smoothly to using the new engine without a steep learning curve.

Benefits of the Proposed Solution

- **Enhanced Immersion:**
  - By integrating machine learning, games can offer more lifelike and unpredictable NPC behaviors, creating a richer player experience.
- **Dynamic Content Generation:**
  - The engine will enable the creation of content that evolves based on player actions, providing a unique and personalized gaming experience.



Action
Frontend Layer
Action
Action
Action
Action
GUI
PROJECT SELECTOR
GUI
EDITOR
Class
+ field: Type
+ method(): Type
<<Interface>>
Database Layer
users
projects
File Handler
Process Handler
bash
powershell
apple
Backend
Layer
<<namespace>>
Game Engine
<<namespace>>
Component
Transform
position, rotation, scale: Vector3
Rigidbody
velocity: Vector3
Renderer
color: Color
GameObject
transform : Transform
rigidbody : Rigidbody
renderer : Renderer
collider : Collider
+ show() : Void
+ collides() : bool
Input
KeyTable : std::map
KeyDownTable : std::map
+ getKeyDown() : bool
+ getKey() : bool
+ getAxisRaw() : bool
+ updateKeyState() : void
<<global reference>>
GameObject
INPUT
<<namespace>>
Mathematics
Framework
Color
r, g, b, a : double
... : Color
Color32
r, g, b, a : int32
... : Color
Random
Vector : Vector3
Color : Color
Vector3
x,y,z,w : double
zero : Vector3
one : Vector3
+ dot () : double
+ distance () : double
Color
Vector
<<namespace>>
Machine Learning Engine
scipy
sklearn
openai
<<namespace>>
Render Engine
GLFW
GLUT
OPENGL

Part II

Implementation

Results

```cpp
#include "../../engine/engine.h"
int main()
{
  RenderEngine::setStart(start); RenderEngine::setUpdate(update); RenderEngine::Enabled(true);
}
GameObject dvd; GameObject vWalls[2]; GameObject hWalls[2];
void start()
{
  dvd.rigidbody.velocity = Vector3(0.01, 0.001);

  vWalls[0].transform.position = Vector3(-1, 0); vWalls[0].transform.scale = Vector3(0.01, 2);
  ...
  hWalls[1].transform.position = Vector3(0, 1); hWalls[1].transform.scale = Vector3(2, 0.01);
}
void update()
{
  RenderEngine::background(0);
  dvd.transform.position += dvd.rigidbody.velocity;
  dvd.show();
  for (GameObject& wall : vWalls)
    if(dvd.collides(wall))
      dvd.rigidbody.velocity.x = -dvd.rigidbody.velocity.x;

  for (GameObject& wall : hWalls)
    if(dvd.collides(wall))
      dvd.rigidbody.velocity.y = -dvd.rigidbody.velocity.y;

  if(GameEngine::Input::getKeyDown(GameEngine::KEY_R))
    dvd.transform.position = Vector3::zero;
}
```

Listing 1: DVD_Logo_Bouncer Example Code

Analysis

start Function Description:

- **DVD Velocity:** $(0.01, 0.001)$
- **Vertical Walls:**
  - Left: $(-1, 0)$
  - Right: $(1, 0)$
- **Horizontal Walls:**
  - Bottom: $(0, -1)$
  - Top: $(0, 1)$

update Function Description:

- **Collision Detection:**
  - Let A and B be two axis-aligned squares with:
    - Center of A: (x_A, y_A), Side length: s_A
    - Center of B: (x_B, y_B), Side length: s_B
  - Collision occurs if:

$$(x_A - x_B)^2 + (y_A - y_B)^2 \leq \left(\frac{s_A + s_B}{2}\right)^2$$

Each Game Engine has its own style based by its functionality

Each Game Engine defines its functionality through its implemented components.

Sources of Inspiration

The development of the game engine draws inspiration from several established platforms:

Unity: The math engine draws inspiration from Unity's advanced mathematical computations and transformations, leading to the development of a robust internal framework.

```
void start();
void update();
int main() { Awake(); }
void Awake() {
  RenderEngine::setStart(start);
  RenderEngine::setUpdate(update);
  RenderEngine::setFixedUpdate(fixedUpdate);
  //
  RenderEngine::START(true);
}
void start() {
  Gameobject go;
  go.transform.position = Random::Vector3().normalised *  Random::Value(-5, 5);
  Debug::Log(go.transform.position)
}

```

p5.js: The rendering engine draws inspiration from p5.js, renowned for its simplicity and accessibility in the creative coding community. The straightforward approach to rendering and graphical output in p5.js has influenced our rendering pipeline's design, making it powerful and user-friendly.

```

point(x, y, c); // Changes the color of the pixel at location <x, y> to c

line(<<x>, <y>>,
     <<x>, <y>>); // Draws a line between 2 points

background(color);

square(point1, point2);
fill(color);
noStroke();
circle(point , radius);
```

Each component in a software framework must work in harmony

Each Game Engine defines its functionality through its implemented components.

The Frontend

The frontend component is organized into several directories and files. It includes functionalities for the editor, hub, and splash interfaces.

$$\text{Directory } D := \{d_1, d_2, \ldots, d_n\}, \quad \text{where TypeOf}(d_i) \in \{\text{File}, \text{Directory}\}$$

$$\forall\, module_i \in \text{frontend} \Rightarrow module_i \supseteq \{\text{init.py}, \text{main.py}, \text{cli.py}\}$$

Specific to this implementation, we have the following modules and sub-modules:

$$(\exists\ \text{Editor} \in \text{Modules}).\ \text{Editor.submodules}\ \rightarrow_{\text{depend on}}\ \text{PyQt}$$

$$(\exists\ \text{Editor} \in \text{Modules}).\ \text{Editor.submodules}\ \supseteq \{\text{Hierarchy}, \text{Scene View}, \text{Inspector}, \text{Assets}, \text{Terminal}\}$$

$$(\exists\ \text{Assets} \in \text{Editor.submodules}).\ \text{Assets} \rightarrow_{\text{modifies}} \{\text{Project Tree}\}$$

$$(\exists\ \text{Hierarchy} \in \text{Editor.submodules}).\ \text{Hierarchy} \rightarrow_{\text{modifies}}\ \text{JSON\_Scene\_File}$$

$$(\exists\ \text{Inspector} \in \text{Editor.submodules}).\ \text{Inspector} \rightarrow_{\text{modifies}}\ \text{JSON\_Scene\_File}$$

The interactions between different modules can be represented using functions and mappings. Let $f : E \to H$ represent the function mapping elements from the Editor to the Hub. Similarly, $g : H \to S_p$ represents the mapping from Hub to Splash.

$$\text{user} \xrightarrow{\text{launch}} \text{splash} \xrightarrow{\text{launch}} \text{Hub} \xrightarrow{\text{launch}} \text{Editor}$$

$$\text{Editor} \xrightarrow{\text{launch}} \{\text{Hierarchy}, \text{SceneView}, \text{Inspector}, \text{Assets}\}$$

$$\text{SceneView} \xrightarrow{\text{launch}} (\text{c++}\ \wedge \text{python})\ \text{Runner} \xrightarrow{\text{launch}} \{\text{opengl}, \text{scipy}\}$$

$$\text{Hierarchy} \xrightarrow{\text{reads}} \{\text{active\_scene.json}\}$$

$$\text{Inspector} \xrightarrow{\text{reads}} \{\text{active\_game\_object.json}\}$$

$$\text{Assets} \xrightarrow{\text{reads}} \{\text{FileTreeOf(projectPath)}\}$$

$$(\text{user} \wedge \text{Hub}) \vee (\text{user} \wedge \text{Editor}) \vee (\text{user} \wedge \text{CLInterface}) \xrightarrow{\text{Request}} (\text{FileManager} \vee \text{SceneManager}) \xrightarrow{\text{response}}$$

The Backend

The backend component is organized into several directories and files. It includes functionalities for the opengl renderer, math engine, machine learning interface and input handling.

Data Structure

```
class Color32
{
  public:
    unsigned int r, g, b, a;

    Color32(double grayscale)
           : Color(grayscale, grayscale, grayscale, 1.0f) { }
    Color32(double _r, double _g, double _b, double _a = 1.0f) {  }
```

Listing 2: Color Declaration

```
  ...
  static std::map<unsigned char, bool> KeyUpTable;
  static std::map<unsigned char, bool> KeyTable;
  static std::map<unsigned char, bool> KeyDownTable;

  static void updateKeyState(unsigned char key, bool state = true);
  static void resetKeyStates();

class Input
{
  public:
    static bool getKeyDown(GameEngine::KeyCode key);
    static bool getKeyDown(GameEngine::KeyCode key);
    static bool getKeyUp(GameEngine::KeyCode key);

    static bool getAxisRaw(std::string axisName);
};

enum Keycode
{
  KEY_A = 'a',
  ...
}
```

Listing 3: Input Handler

Rendering Pipeline

```
class opengl {
  private:
    static void DisplayFunc;  static void ReshapeFunc;
    static void KeyboardFunc; static void KeyboardUpFunc; static void MouseFunc;
  public:
    void setAwake (void (*func)()); void setStart(void (*func)());
    void setUpdate(void (*func)()); void setFixedUpdate(void (*func)());

    void background(Color c);           void fill();      void noFill();
    void stroke(Color c);               void strokeWeight(double weight);
    void point(Vector3 pos);            void line(Vector3 start, Vector3 end);
    void rect(Vector3 bL, Vector3 tR);  void circle(Vector3 c, double r, double seg=1000);
};
```

Listing 4: Renderer Declaration

```
...
line: (R^2, R^2) -> Render,
                              glBegin(GL_LINES)
                                glVertex2f(x1, y1)
                                glVertex2f(x2, y2)
                              glEnd()
rect: (R^2, R^2) -> Render
                              glBegin(GL_QUADS)
                                glVertex2f(x1, y1)
                                glVertex2f(x2, y1)
                                glVertex2f(x2, y2)
                                glVertex2f(x1, y2)
                              glEnd()
circle: (R^2, R) -> Render
                              glBegin(GL_TRIANGLE_FAN)
                                glVertex2f(xc + r * cos(i), yc + r * sin(i))
                                for i in [0, 2pi, 2pi/segments]:
                                  glVertex2f(xc + r * cos(i), yc + r * sin(i))
                              glEnd()
```

Listing 5: Renderer Definition

Data Structure

For storing a 3D point, a conventional approach is to use a vector of length 3, often represented as (x, y, z). This data structure is generally sufficient for typical use cases.

However, there are situations where a more sophisticated solution becomes necessary. This advanced approach involves employing a $(k+1)$-dimensional vector space to represent entities of dimension k and it not only handles edge cases but also simplifies other computational tasks.

```
class Vector3 {
  public:
    double x, y, z, w;
    Vector3(double _x, double _y = 0, double _z = 0, double _w = 1):x(_x),y(_y),z(_z),w(_w){}
```

Listing 6: Vector3 Declaration

Properties

$$\mathbf{v_i} = \begin{pmatrix} x & y & z & w \end{pmatrix}.$$

$$v_i.\text{w} = \begin{cases} 1 & \text{if } v_i \text{ is used for describing a point} \\ 0 & \text{if } v_i \text{ is used for describing an arrow} \end{cases}$$

```
Vector3 v = Vector3::one * 7;
Debug::Log(v); // <7,7,7,1>

```

Listing 7: Vector3 Usage

Coordinate System Conversion:

$$(x, y, z) \quad = \quad \text{Cartesian Coordinates} \qquad\qquad (r, \theta, \phi) \quad = \quad \text{Polar Coordinates}$$

toPolar(Vector3 point):

$$\begin{cases} \theta & = \arctan\left(\frac{p.y}{p.x}\right) \\ \phi & = \arccos\left(\frac{p.z}{r}\right) \\ r & = \sqrt{p.x^2 + p.y^2 + p.z^2} \end{cases}$$

toCartesian(Vector3 point):

$$\begin{cases} x & = r \sin\phi \cos\theta \\ y & = r \sin\phi \sin\theta \\ z & = r \cos\phi \end{cases}$$

Operators

```
Vector3& operator=(const Vector3& other);
Vector3& operator+=(const Vector3& other);
Vector3& operator*=(double scalar);
Vector3 operator*(double scalar) const;
Vector3 operator-(const Vector3& other) const;
Vector3 operator+(const Vector3& other) const;
double dot(const Vector3& other) const;
Vector3 operator/(double scalar) const;
double distance(const Vector3& other) const;

```

Listing 8: Vector3 Operators

- **Dot Product**: Calculates the dot product between the current vector and another.

$$\mathbf{v} \cdot \mathbf{w} = \sum_{i=1}^{n} v_i \cdot w_i$$

- **Distance**: Calculates the Euclidean distance between the current vector and another.

$$\text{distance}(\mathbf{v}, \mathbf{w}) = \sqrt{\sum_{i=1}^{n} (v_i - w_i)^2}$$

Operations

Translation:

$$T(x, y, z) = \begin{pmatrix} 1 & 0 & 0 & x \\ 0 & 1 & 0 & y \\ 0 & 0 & 1 & z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Scaling:

$$S(s_x, s_y, s_z) = \begin{pmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

These equasions must be really easy to use. For this i have chosen the option that highly resembles Unity's ecosystem.

```
...
int main() { Awake(); }
void Awake()
{
  RenderEngine::setStart(start);  RenderEngine::setUpdate(update);
  RenderEngine::START(true);
}
void start()
{
  Gameobject go;                              int speed = 0.01;
  Debug::Log(go.transform.name);              Debug::Log(go.transform.position)

  go.transform.translate(Vector3::Right * speed);  Debug::Log(go.transform.position)
  go.transform.position = Vector3::one * Math::sqrt(Math::pi);
}
```

Listing 9: source-code

Here lies the slight inconvenience mentioned earlier in this chapter, which justifies the use of Homogeneous Coordinates.

Rotation with Euler Angles (XYZ order):

$$R_{XYZ}(\alpha, \beta, \gamma) = R_X(\alpha) \cdot R_Y(\beta) \cdot R_Z(\gamma) \tag{1}$$

where

$$R_X(\alpha) : \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\alpha & -\sin\alpha & 0 \\ 0 & \sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} R_Y(\beta) : \begin{pmatrix} \cos\beta & 0 & \sin\beta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\beta & 0 & \cos\beta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} R_Z(\gamma) : \begin{pmatrix} \cos\gamma & -\sin\gamma & 0 & 0 \\ \sin\gamma & \cos\gamma & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Gimbal Lock Issue: Euler angles suffer from gimbal lock, where two of the three rotational axes align, leading to a loss of one degree of freedom.

Solution: Quaternions

$$q = \cos\left(\frac{\theta}{2}\right) + \sin\left(\frac{\theta}{2}\right)(u_x i + u_y j + u_z k) \tag{2}$$

where θ is the rotation angle and (u_x, u_y, u_z) is the unit vector representing the axis of rotation.

Integration Capabilities

Python's extensive ecosystem empowers this project to incorporate a wide array of tools and libraries. For instance, the project's modular design facilitates integration with advanced statistical packages like NumPy and SciPy for robust mathematical computations. Furthermore, visualization tools such as Matplotlib or interactive frameworks like Plotly can enhance data representation and exploration.

Data Structures

```cpp
class Event {
public:
    std::string name;
    double probability;

    Event(std::string _name, double _probability) : name(_name), probability(_probability) {}
};
```

Listing 10: Event Class Declaration

```cpp
class Outcome {
public:
    std::string description;
    double value;

    Outcome(std::string _description, double _value) : description(_description), value(_value
    ) {}
};
```

Listing 11: Outcome Class Declaration

```cpp
std::vector<Outcome> omega = {
    {"Outcome 1", 0.25},
    {"Outcome 2", 0.35},
    {"Outcome 3", 0.4}
};
```

Listing 12: Omega (Set of All Possible Outcomes)

```cpp
class ProbabilitiesEngine {
public:
    double calculateProbability(double event, double totalEvents);
    double calculateConditionalProbability(double eventA, double eventB);
    double calculateJointProbability(double eventA, double eventB);
    double calculateBayesTheorem(double eventA, double eventB);
};
```

Listing 13: Probabilities Engine Declaration

Bayes' Theorem:

$$P(A|B) = \frac{P(B|A) \cdot P(A)}{P(B)}$$

The ID3 (Iterative Dichotomiser 3) algorithm is a decision tree learning algorithm used for classification. Here's a concise overview of its implementation:

```
class ID3Engine {
public:
    void createDecisionTree();
    void calculateEntropy();
    void calculateInformationGain();
    void pruneDecisionTree();
};
```

Listing 14: ID3 Engine Declaration

Mathematical Formulas:

- **Entropy Calculation:**

$$H(S) = -\sum_{i=1}^{c} P_i \cdot \log_2(P_i)$$

- **Information Gain:**

$$IG(S, A) = H(S) - \sum_{v \in Values(A)} \frac{|S_v|}{|S|} \cdot H(S_v)$$

Project Architecture Flexibility

The project's architecture allows seamless integration with various Python tools beyond the Probability Engine and ID3 Decision Tree. Python's versatility is showcased, enabling diverse applications.

Customizability and Extensibility

Developers extend the project with additional machine learning algorithms from libraries like Scikit-learn or apply NLP techniques using NLTK or spaCy. This ensures adaptability to evolving requirements.

Practical Applications

The architecture supports applications from data analysis to natural language processing and computer vision. Python's ecosystem allows tailored solutions for scalability and performance.

Conclusion

While the Probability Engine and ID3 Decision Tree highlight capabilities, the architecture fosters integration with a vast array of Python tools, promoting exploration and customization.

Deployment and Distribution

Packaged Builds

The project aims to streamline deployment with packaged builds for easier distribution. Plans include packaging for AUR (Arch User Repository) and as a PyPi (Python Package Index) library.

Module Packaging

Each module within the application should be independently packaged to facilitate modular use and distribution through PyPi.

Current Progress

Early builds are available for testing via the following command:

```
pip install -i https://test.pypi.org/simple/ game-genie
```

Support for pip installation is temporarily paused pending further stabilization of the application.

Performance Enhancement

Evaluation and Optimization

Efforts are ongoing to evaluate and optimize the application's performance, focusing on runtime efficiency and responsiveness.

Speed Analysis

Initial benchmarks indicate satisfactory performance within current scope. Future optimizations will target critical execution time areas.

Optimization Strategies

Proposed strategies include algorithmic improvements, caching mechanisms, and leveraging parallel processing capabilities.

Conclusion

These planned enhancements are designed to elevate the application's functionality, performance, and usability. By focusing on deployment, optimization, and integration, the project aims to deliver a more robust and efficient toolset for its users.

Integration with Web Scraper

Enhancing Machine Learning Capabilities

Integrating a web scraper component enhances the application's data acquisition capabilities for machine learning tasks.

Web Crawler: Inner Workings

The web crawler script utilizes Selenium for web browsing and Colorama for output coloring. Below are key functions and their mathematical underpinnings:

```python
def getProductDetails(driver):
    try:
        productDetails = driver.find_element(By.CLASS_NAME, "product-details")
    except Exception as e:
        print(Back.RED + f"Could not find product details -> {e} ")
        return None
    return productDetails
```

Listing 15: Function to Retrieve Product Details

```python
def getLowestPrice(driver):
    productDetails = getProductDetails(driver)
    if productDetails is None:
        print(Back.RED + f"Could not get product details")
        return -1
    try:
        lowestPriceText = productDetails.find_element(By.XPATH, "//*[contains(@itemprop, '
lowPrice')]")
        lowestPrice = getFloat(lowestPriceText.text)
    except Exception as e:
        print(Back.RED + f"Lowest Price could not be found -> {e} ")
        return -1
    return lowestPrice
```

Listing 16: Function to Extract Lowest Price

Conclusion

By integrating Selenium for web automation and leveraging mathematical parsing techniques, the web crawler efficiently gathers product data from URLs provided as command-line arguments.

Part III

Bibliography

Bibliography

Literature

In this section, I present the literature that informed this research, organized by the specific knowledge gained from each resource.

Mathematical Framework

1. *Mathematics For Game Developers* by Christopher Tremblay
   Provided foundational knowledge on vector mathematics and geometric algorithms crucial for implementing physics and spatial computations in the game engine.

Rendering Engine

1. *Computer Graphics in C/C++* by Donald Hearn
   Contributed insights into fundamental rendering techniques such as rasterization, shading, and texture mapping, which were essential for building the rendering engine.
2. *OpenGL SuperBible* by Bjarne Stroustrup
   Detailed explanations and examples of modern OpenGL programming, including shader programming and GPU-based rendering techniques, which directly influenced the rendering pipeline development.

C++ Software Infrastructure

1. C++ *Primer* by Bjarne Stroustrup
   Provided comprehensive coverage of C++ language features, syntax, and best practices, which formed the backbone of the software infrastructure of the game engine.