Blank Engine

"Where developers fill the gap!"

Created by:

Entuna, John Alson H.

Aguilar, Paul Anton Rae D.

Introduction

Game engines nowadays contains highly complex set of APIs that were designed to serve a wide variety of games. These engines focus on different innovations and improvements in terms of graphics, sound and music management, artificial intelligence, and other parts of game development. In an attempt to create an engine somewhat similar to these, the Blank Engine shall be developed.

The Blank Engine is an RPG engine that provides developers to create games with *terrain navigation, collision, party creation,* and *basic turn-based battle systems*. Further support for *sound management, 2D animations,* and *user interface* will also be provided to be utilized and used to make games more visually appealing. With these simple capabilities of the engine, the implementation of RPGs for developers can be a breeze. The engine's versatility can also enable developers to create RPGs which aren't only turn-based. In fact, other genres can also be created with this engine.

As there are several engines that have features that are similar to what Blank Engine has, the team's objectives with the production of this engine is to:

- Create the engine through the application of *Object-Oriented Programming*.
- Create an engine that follows the *Component Based Design*.
- Create an engine that is flexible and expandable.
- Create an engine that is capable, but not only limited to, producing a basic turn-based RPG.
- Create an engine that is versatile enough to create games from other genres in 2D.

With this attempt to create an engine using C++ as the base language, the team may be able to provide new and aspirant developers with an array of knowledge regarding *Object-Oriented Programming* and *Component Based* Design which in turn, allows them to use and expand the current contents of the engine. The engine should also be able to welcome designers and artists alike as this would make it easy enough for them to create games.

Features of the Game Engine

In order to give the game engine a powerful way on providing graphics, the *Simple and Fast Multimedia Library* or *SFML* shall be used. Some of the classes will be inherited from a class in SFML to utilize its given features. Though, due to the component based architecture of the engine, some of its classes can stand alone.

A. GameObject

Allows the creation of an object and storage of information about it. The engine allows game objects to add and remove components. *Vector2f* and *Vector3f* from SFML were also used to control the positions and movement of the game object.

Methods:

- void SetPosition(sf::Vector2f position);
- void SetScale(sf::Vector2f scale);
- void SetRotation(sf::Vector3f rotation);
- void SetTag(int tag);
- void SetName(string name);
- void Translate(sf::Vector2f direction);
- void Rotate(sf::Vector3f rotation);
- void Scale(sf::Vector2f scale);
- void AddComponent(Component* component);
- void RemoveComponent(int componentID);
- void ViewObjectComponents();

B. Component

An abstract class that allows the creation of components that can be attached to the GameObject. This class shall be inherited by other classes like *Animator, SpriteRenderer, and PlayerMovement* so that it would be possible to attach it to specific game objects.

Methods:

- Int GetComponentID();
- string& GetComponentName();

C. *SpriteRenderer*

Allows the creation of Sprites by using SFML's sf::Texture and sf::Sprite. The texture of a sprite can be immediately be set to the specific sprite by using a specific method. The *Animator* component was used to allow animations if ever desired to be used.

Methods:

- void SetSprite(string filename);

D. *Animation*

Allows the collection of different Textures to be animated. A vector of sf::Texture was used to store all the required texture for a specific animation.

Methods:

- void AddTexture(sf::Texture texture);
- void GetFrame(int index);
- int GetFramesCount();

E. *Animator*

Allows the playing of a set of animation. This component requires Animations which will be played on specific events.

Methods:

- void SetAnimation(const Animation& animation);
- void SetFrameTime(sf::Time time);
- void Play();
- void SetLooped(bool looped);
- const Animation* GetAnimation();
- bool IsPlaying();

F. *Collision*

Allows the checking of collision between two Sprites. *DISCLAIMER: The original author of this class is Nick Koirala (original version) and ahnonay (SFML2 compatibility).*

Methods:

- bool PixelPerfectTest(const sf::Sprite& Object1 ,const sf::Sprite& Object2, sf::Uint8 AlphaLimit = 0);

- bool CreateTextureAndBitmask(sf::Texture &LoadInto, const std::string& Filename);
- bool CircleTest(const sf::Sprite& Object1, const sf::Sprite& Object2);
- bool BoundingBoxTest(const sf::Sprite& Object1, const sf::Sprite& Object2);

G. Layer

Allows the separation of Sprites per layers. This shall fix issues regarding overlapping images. Each layer contains a vector of Sprites which will be rendered according to its index number.

Methods:

- void Render(sf::RenderWindow *window);
- void AddSprite(sf::Sprite sprite);

H. SoundManager

Allows the collection of different sounds effects and background music. These shall be contained inside a vector. A method to play the specified track was also included.

Methods:

- void AddSound(Sound music);
- void PlayBGMusic(string name);

I. Sound

Allows the creation of sounds that can be used using the SoundManager.

Methods:

- void InitializeSound(string name, SoundType sound, const string& filename);

J. GUIObject

Allows the creation of GUIs that can be used as buttons. This is an abstract class which is inherited by *GUISelector, GUIImage, and GUILabel.*

Methods:

- void setTag(unsigned int tag);

- void getTag();
- void addChild(GUIObject* obj);
- void removeChild(GUIObject* obj);
- void draw(sf::RenderTarget& target, sf::RenderStates states);
- virtual onDraw (sf::RenderTarget& target, const sf::RenderStates states);

K. GUIDelegate

An abstract class that allows the setting of different effects whenever a GUI/button is selected or pressed.

Methods:

- virtual void unSelect(GUIObject* p_object) = 0;
- virtual void onSelect(GUIObject* p_object) = 0;
- virtual void activate(GUIObject* p_object) = 0;

L. Scenes/Screens

Allows control over which are supposed to be enabled. Scenes can be used to separate the features of a game, ie. Battle System, Terrain Navigation, and Party Creation.

Methods:

- virtual void Start() = 0;
- virtual void Load() = 0;
- virtual void Update(float time) = 0;
- virtual void Exit() = 0;
- virtual void DrawScene(sf::Window& window) = 0;
- void SetSceneName(const string& name);
- const string& GetSceneName();

M. Unit

Allows the creation of an RPG unit. This contains all basic RPG character elements.

Methods:

- void InitializeUnit(const string& name, float health, float mana, float strength, float attack, float magic, float agility, float dexterity, float luck, int level);
- virtual bool BasicAttack(Unit* target) = 0;
- virtual bool Flee() = 0;

- virtual void Print() = 0;
- bool IsAlive();
- string getName();
- float getStat(STAT stat);
- int getLevel();

N. PlayerMovement

Allows the control of a GameObject for TerrainNavigation. An object must be tagged as Player.

Methods:

- void SetPlayer(GameObject* object);
- void Update();

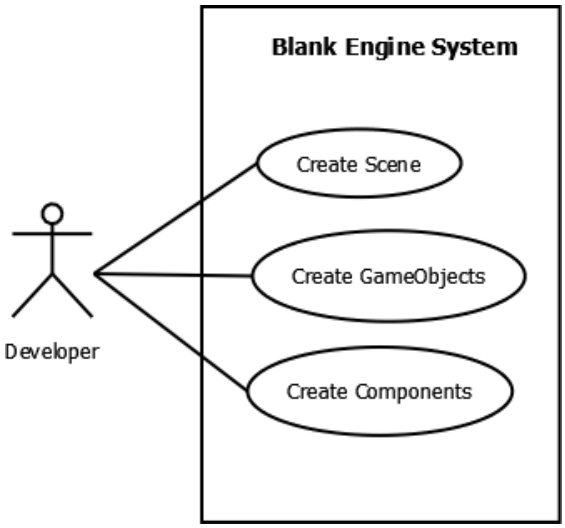
O. Enemy

Allows the use of Artificial Intelligence to control over the possible enemies. Enemies inherited from the Unit class.

Methods:

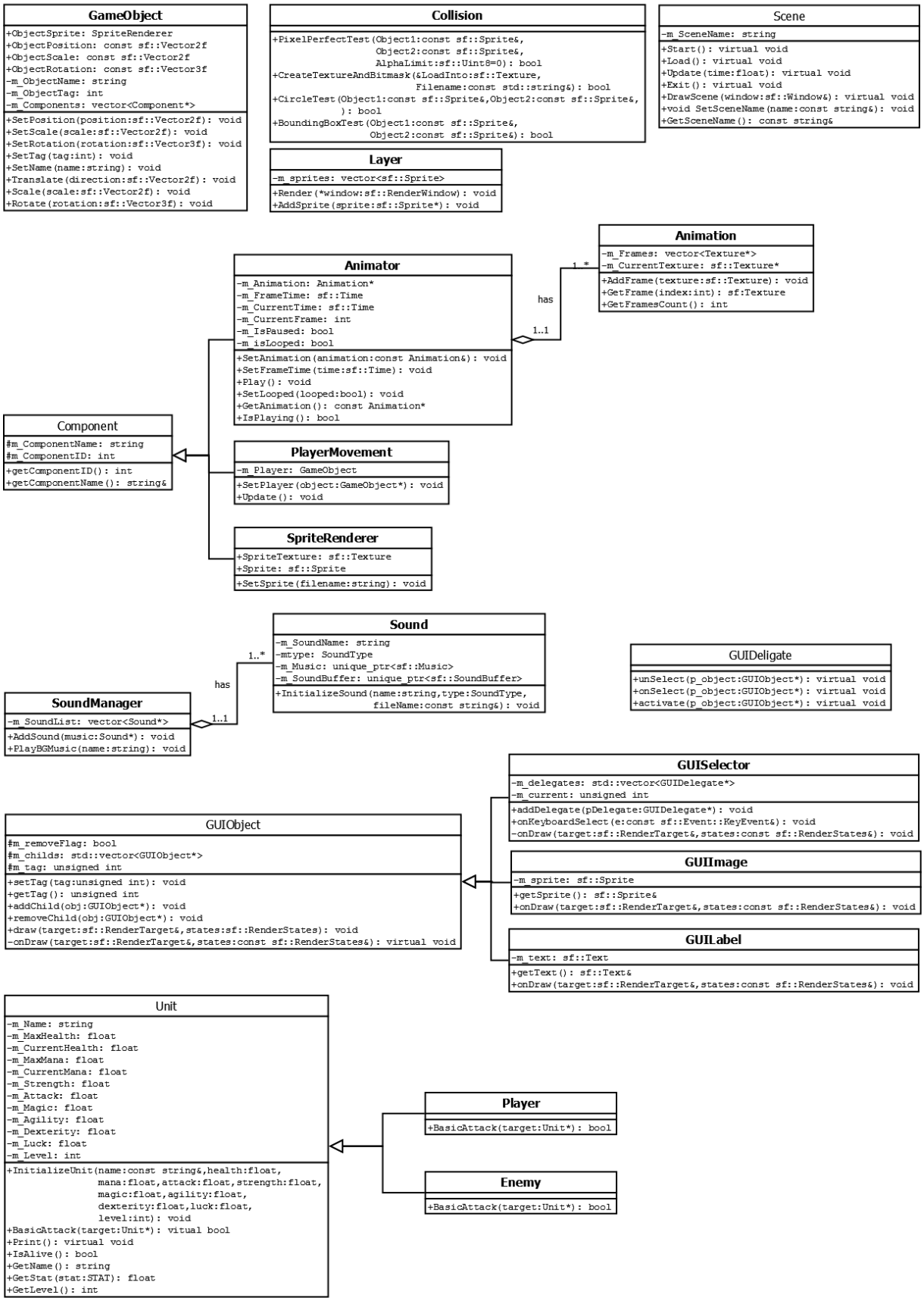
- Enemy(const string& name, float health, float mana, float strength, float attack, float magic, float agility, float dexterity, float luck, int level);
- bool BasicAttack(Unit* target) override;

UML Diagrams



Simple Activity Diagram for Blank Engine

Class Diagram



Design Considerations of the Game Engine

The Blank Engine follows different programming principles and architectures to achieve its desired state. With the team's current knowledge in the used programming language (C++), application of concepts taught and learned were considered to produce this RPG engine.

Object-Oriented Programming (OOP)

Object-Oriented Programming is a programming paradigm that is highly based on *objects*. These objects contain attributes and methods which handles the data and procedures within it. OOP has features and techniques to make the game engine efficient. These features are *Abstraction, Encapsulation, Inheritance,* and *Polymorphism*.

The use of Object-Oriented Programming allowed the inheritance of classes from SFML for the game engine. The team also managed to create an abstract class for units in the game as it branches out from a player controlled unit to an AI controlled unit. Through the careful consideration of OOP, the team achieved a proper way of implementing the features of the engine.

Component Based Architecture (CBA)

Component Based Design is a pattern that uses various components. These components only contains specific attributes and methods related to its sole use. This architecture allows any program to be flexible as it each components can dictate the behavior.

The use of Component Based Architecture allowed the code to be more flexible for future updates and expansions. This also allowed the team to avoid problems such as deep inheritance.

SOLID Principle

The SOLID Principle is an acronym for the five basic principles of object-oriented programming. It was said that it takes time to master such principles and each principle is somewhat highly connected to each other. In a way, SOLID is a guideline for programmers to avoid the refractor of codes.

The team followed some of the principles of SOLID. The key principle that was followed was the O of SOLID which is, Open/Closed Principle. As stated in the objectives, the engine should be flexible and expandable. This goal is somewhat risky as it may cause different errors that may break the code and bring the need to refractor. Through following SOLID's Open/Closed Principle, the engine will become open for extensions but closed for modifications.

Coding Style

- *Camel Case*
  - Used for declaring private variables and functions.
  - Example: void camelCase();

- *Pascal Case*
  - Used for declaring public variables and functions.
  - Example: void PascalCase();

- *Variable Naming Conventions*
  - *m_variable* naming convention shall be used when declaring member variables of a class.

Limitations of the Game Engine

The attempt to produce an engine that's simple yet powerful was decided upon by the team even if there were a limited pool of knowledge. With the use of SFML, the engine can contain multiple versatile features that can surely benefit the developers. The engine was even furthered through the research of different libraries and APIs that can support the demand of the engine. Though, even if some of the features are versatile enough, there are many considered limitations for the engine.

These are some of the identified limitations:

- 3D Support
- Mouse Control
- Text Inputs
- Networking
- Performance Fixes for insufficient Hardware Requirements (The performance of the engine depends on how updated the PC is)

One of the team's goal in creating this engine was to make it flexible and expandable. Through following different architectures and programming styles, the engine, in some way, can be expanded to lift the limitations that were identified. Also, it was stated that this engine's target games were RPGs, but it was not included in the limitations as the engine is also flexible enough to produce other games in 2D.

If time permits, the team would like to further the capabilities of the engine and make it capable of producing more games to help all types of developers.