A Simple Game Engine

110504517 資電四 李睿穎

Abstract

System architecture is crucial in all software. This project, based on WinForms CLR in C++, provides the foundational engine support needed for 2D games, including event handling, collision detection, and encapsulation of WinForms calls.

I. Introduction

This project focuses on a universal framework suitable for various applications. Users can quickly implement diverse functionalities by adding objects without worrying too much about resource management. We have divided the features into the following subsystems below. Next, we will introduce how to use the engine.

a.	Manager	- Control the app's operation and invoke various functionalities.
b.	Input	- Capture user inputs, such as keyboard and mouse actions.
c.	GameObject	- Functional modules available for user extension.

II. Run Project

You can directly clone or download the entire project to run it without any additional steps. You should be able to compile, execute, and open a window(*Fig.1*).

Fig.1 Default app window

Now you can add objects by including executable code. Add your .cpp file anywhere in the project and create an executable object using the sample code (*Block.1*). The function will be triggered at different execution phases, as shown in *Table 1*.

```
#include "App.h"
auto run_setup =
Start::Create([]() {
    cout << "init\n";

});
```

Block.1 Create executable object

Event Name	Invoke Time	Usage
Start	When app start	Init object
PreUpdate	Before Update	
Update	Every Frame	Main logic
LateUpdate	After Update	
Render	When rendering screen	Not recommend to use, you can do it in GameObject.
GameReset	When manager called reset function	Reset values

Table.1 Global event invoke and usage

Although this method allows you to execute the program, a better approach is to create executable objects (GameObject). There are two ways to create them: the first is by inheriting from the GameObject class and overriding the Update and Render functions; the second is by creating a FuncGameObject to create function-based objects (suitable for lambda functions).

Here, we create a red square as an example(*Fig.2*)

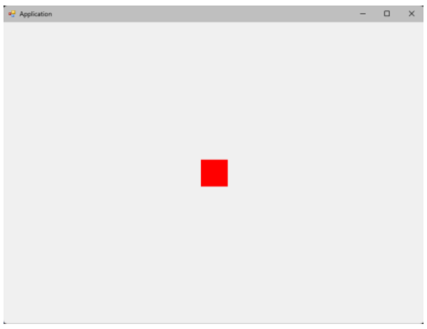


Fig.2 Red square

```
class RedSquare : public GameObject {
public:
    RedSquare(Vector2 size)
        : size(size) {}

    Vector2 size;

    virtual void Update() override {

    }
    virtual void Render() override {
        Vector2 halfSize = size / 2.0f;
        Drawer::AddFillRect(Color(1, 0, 0), Rect(-halfSize, size));
    }
};

auto run_setup2 =
Start::Create([]() {
    auto obj = new RedSquare(Vector2(50, 50));

});
```

Block.2a Create a red square object by inheriting GameObject

```
auto run_setup2 =
Start::Create([]() {
    auto obj = new FuncGameObject([]() {
        //Update
        }, []() {
        //Render
            Vector2 size = { 50,50 };
            Vector2 halfSize = size / 2.0f;
            Drawer::AddFillRect(Color(1, 0, 0), Rect(-halfSize, size));
        });
});
```

Block.2b Create a red square object by FuncGameObject

III. Utilities

We provide various features to optimize the development experience. Below are the more commonly used features. If you need detailed information, refer to the appendix documentation.

a. Global vals

    You can access various standard information such as deltaTime, screenSize, key inputs, etc., through the Global class. For C# control classes, use RefGlobal.

b. Transform

    All objects contain transform information, including position, rotation, and scale. You can manipulate the object's position directly through it, or use the rigidbody for operations (GameObject.rigidbody).

c. Collider

    The engine comes with built-in collision detection. For objects that require it, you can add collision volumes through the collider (GameObject.collider).

d. Camera

    The Camera affects the rendering perspective and can be accessed through Global::MainCamera. It is also one of the GameObject objects.

e. Tags

    We use an int32 flag as the Tag system to optimize calculations, such as collider filtering, and special functions like preventing deletion during reset.

f. Drawer

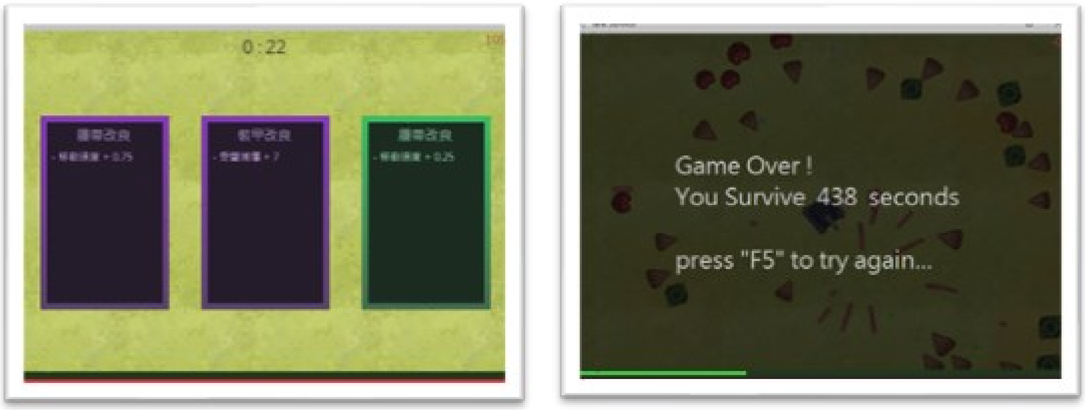
    Use Drawer for object rendering. The rendering of each GameObject is based on their local coordinates.

g. UI

    UI is a GameObject with the 'UI' tag. Unlike regular objects, the rendering of UI elements is not affected by the camera's position, and they also have a higher default rendering priority.

IV. Example Game – CppTank

CppTank is a simple vampire-survivor-like game, with all its features implemented within the architecture of this project. In the game, waves of monsters continuously attack the player, and the player must find ways to survive and upgrade to deal with the endless monsters.



Game Over !
You Survive 438 seconds

press "F5" to try again...

Fig.3 CppTank game screen-shot

Here is a brief introduction to the three types of monsters in the game.

- Continuously approach the player.
- Maintain a certain distance from the player and shoot bullets to attack.
- Follow other enemies and heal them.

The player can level up by gaining experience from defeating monsters, including but not limited to attack power, defense, and shooting frequency.

This example fully utilizes the project's architecture and extends it with a combat system. It also allows for easy addition of different monsters and upgrades through inheritance, making it a sufficiently complete game framework.

V. Conclusion

This project extends the WinForms CLR foundation using C++ to build a complete 2D game engine architecture, including but not limited to collision detection, image rendering, and other functions. It also provides a simple game, CppTank, implemented using this architecture. For those interested in the development process, this project serves as an example.

VI. Appendix-API

Tag

- **Purpose**: Represents flags used to categorize or tag game objects.
- **Attributes**:
  - flag: int - The current flag value.
- **Methods**:
  - Add(int value): Adds a flag to the object.
  - Remove(int value): Removes a flag from the object.
  - Contains(int value): Checks if the object contains a specific flag.
  - Any(int value): Checks if any of the specified flags are set.

Layer

- **Purpose**: Defines different layers used in rendering or sorting game objects.
- **Constants**:
  - UI: int - UI layer.

GameObject

- **Purpose**: Represents a game object in the game world.
- **Attributes**:
  - tag: Tag - The tag of the game object.
  - render_layer: int - The rendering layer of the object.
  - enable: bool - Indicates whether the object is enabled.
  - position: Vector2 - The position of the game object.
  - rotation: float - The rotation of the game object.
  - collider: Collider - The collider for collision detection.
  - rigidbody: Rigidbody - The rigidbody for physical behavior.
- **Static Methods**:
  - GetInstances(): Returns all instances of the game object.
  - GetInstances(int tagMask): Returns game objects filtered by tag mask.
  - GetInstancesByType(int tagMask): Returns game objects filtered by tag and type.
- **Methods**:
  - Update(): Updates the game object state.
  - Render(): Renders the game object.
  - OnCollide(GameObject* other, CollideInfo collideInfo): Handles the collision event with another game object.
  - is_Destroy(): Returns whether the object is marked for destruction.
  - Destroy(): Destroys the game object.

Collider

- **Purpose**: Class for collision detection.
- **Attributes**:
    - gameObject: GameObject* - Pointer to the related game object.
    - hitboxes: vector<Polygon2D> - A list of hitboxes in local space.
    - hitboxes_world: vector<Polygon2D> - A list of hitboxes in world space.
- **Static Methods**:
    - GetIgnoreCollideList(): Returns the list of objects to ignore for collisions.
    - FindObject(const Circle& range, function<bool(GameObject*)> filter): Finds game objects within a specified range using a filter.
    - FindObject(const Circle& range): Finds game objects within a specified range.
    - AddIgnore(const int lhs, const int rhs): Adds tags to the ignore collision list.
    - IsIgnore(GameObject* lhs, GameObject* rhs): Checks whether to ignore collision between two objects.
- **Methods**:
    - AddRect(const Rect& rect): Adds a rectangular hitbox for collision.
    - AddCircle(const Circle& circle): Adds a circular hitbox for collision.
    - CollideWith(Collider& other): Detects collision with another collider.
    - Update(): Updates the state of the collider.

Rigidbody

- **Purpose**: Class that handles the physical movement of a game object.
- **Attributes**:
    - gameObject: GameObject* - Pointer to the related game object.
    - decelerate: float - Deceleration factor (default: 0.9).
    - maxSpeed: float - Maximum speed (default: 5.0).
    - movement: Vector2 - The movement vector of the object.
    - relate_rotation: bool - Whether the movement is affected by rotation.
    - enable: bool - Whether the rigidbody is enabled.
- **Methods**:
    - AddForce(Vector2 force): Adds a force to the rigidbody.
    - AddForce(Vector2 direction, float force): Adds a force in a specified direction.
    - AddForce(float rotation, float force): Adds a force in a specified rotation.

UI

- **Purpose**: A class representing a basic UI element.
- **Attributes**:
  - tag: Tag - Specifies the tag for the UI element, including UI and DontDestroyOnReset by default.
  - render_layer: int - Defines the render layer of the UI element.

UI_Text

- **Purpose**: A UI element that displays text.
- **Attributes**:
  - anchor: Anchor - The anchor point for the text (e.g., UpperLeft).
  - text: string - The text content to be displayed.
  - color: Color - The color of the text (default is a dark gray).
  - size: int - The font size of the text.

UI_Clickable

- **Purpose**: A UI element that can be clicked.
- **Attributes**:
  - IsHovering: bool - Indicates if the cursor is hovering over the UI element.
  - OnClick: vector<function<void()>> - A list of functions to be called when the element is clicked.
- **Constructor**:
  - Sets the tag to Clickable.
- **Methods**:
  - Can register and invoke OnClick functions when clicked.

UI_ProgressBar

- **Purpose**: A UI element that displays a progress bar.
- **Attributes**:
  - IsHorizontal: bool - Specifies whether the progress bar is horizontal or vertical (default is true).
  - IsReverse: bool - Specifies whether the progress bar is filled in reverse (default is false).
  - MinValue: float - The minimum value of the progress bar.
  - MaxValue: float - The maximum value of the progress bar.
  - Value: float - The current value of the progress bar.
  - EmptyColor: Color - The color of the empty part of the progress bar.
  - FillColor: Color - The color of the filled part of the progress bar.
  - Bound: Rect - The bounding box of the progress bar.

UI_Button

- **Purpose: A clickable UI button element.**
- **Attributes:**
  - **label: string - The label of the button.**
  - **color_h: float - The hue value for button color (default is 90).**
  - **color_s: float - The saturation value for button color (default is 0.1).**
- **Methods:**
  - **SetBound(const Rect& bound): Sets the bounding box of the button.**
  - **GetBound(): Returns the bounding box of the button.**

Drawer

- **Purpose**: A static utility class for drawing various shapes, text, and images on the screen.
- **Constants**:
  - DefaultFontSize: int - The default font size used for text (default is 14).
- **Methods**:
  - SetRenderTarget(GameObject* obj, Camera* camera): Sets the rendering target to a specific game object and camera.
  - SetPen(Color color, float thickness): Sets the pen color and thickness for outlines.
  - SetBrush(…): Sets the brush color for filling shapes.

Drawing Methods:

  - AddCircle(…): Adds a circle to the drawing context.
  - AddFillCircle(…): Adds a filled circle.
  - AddRect(…): Adds a rectangle to the drawing context.
  - AddFillRect(…): Adds a filled rectangle.
  - AddPoly(…): Adds a polygon to the drawing context.
  - AddFillPoly(…): Adds a filled polygon.
  - AddText(…): Adds text at a specific position with an anchor.
  - AddImage(Image^ image, Rect position): Adds an image at a specified rectangle position.

RefGlobal

- **Purpose**: A reference class that stores static instances of global objects.
- **Attributes**:
  - MainTimer: Timer^ - A reference to the main timer object for the application.
  - MainWindow: Window^ - A reference to the main window object for the application.
  - CurrentGraphics: Graphics^ - A reference to the current graphics context used for rendering.

Global

- **Purpose**: A utility class for accessing global application state and input handling functions.
- **Methods**:
  - **Keyboard Input**:
    - GetKey(Keys keyCode): Returns true if the specified key is currently being pressed.
    - GetKeyDown(Keys keyCode): Returns true if the specified key was pressed during the current frame.
    - GetKeyUp(Keys keyCode): Returns true if the specified key was released during the current frame.
- **Attributes**:
  - Time: float - The total elapsed time since the application started.
  - RealTime: float - The real-world elapsed time since the application started.
  - DeltaTime: float - The time that has passed since the last update (game's time step).
  - RealDeltaTime: float - The real-world time elapsed between updates.
  - TimeScale: float - A factor to scale the game's time (affects speed of time-related events).
  - ScreenSize: Vector2 - The current screen resolution (width and height).
  - ScreenDiagonal: float - The diagonal size of the screen.
  - MousePosition: Vector2 - The current position of the mouse on the screen.
  - MouseScroller: float - The amount the mouse scroll wheel has moved.
  - UpdatePerSecond: int - The number of updates per second.
  - UpdateCount: long long int - A counter for tracking how many updates have occurred.
  - MainCamera: Camera* - A reference to the main camera object used for rendering and world view.