Programmers Documentation

Written by: Zsikai Eszter

Table of Contents

Header File Overview

scene.h

Purpose: Defines the abstract base class Scene and its subclasses representing individual screens or states in the game (e.g., menu, character overview, and combat). Central to scene management and UI rendering.

Classes:

- Scene (base class)
- ListingScene
- CharOverview
- Fight
- AdventuresMenu
- PreFight

Key Concepts:

- Inheritance-driven architecture to isolate scene-specific logic.
- Virtual methods such as UpdateTweens(), LoadScene(), CheckButtonsOnClick(), and Init() help define the interface, and scene specific behaviour for all scenes.
- Contains function for smooth Scene transition, a one time tutorial overlay, and visual feedback logic.
- In same cases different Scene subclasses use structured containers (e.g., combatChConts, animProps) similar to Unity's empty GameObjects for modularity, consistency and maintainability, with static ui boundles (vectors).

Fight-Specific Features:

- Integrates logic and visuals deeply.
- Maintains a queue (turnOrderFIFO) to preload upcoming actions based on speed stats.
- Coordinates characters, enemies, buffs, skill logic, and animation timing.
- Acts as a coordinator: computes targets, delegates damage, manages turn state, and visual feedback.
- Contains a state machine (CombatState enum) processed in the main game loop.

uiAndDisplay.h

Purpose: Contains the UI system, based on a polymorphic hierarchy of drawable and interactive UI elements.

Classes:

- UIElement (base class)
- Button, Text, Sprite, Bar (derived classes)
- OutlinedText, SpriteButton, TextButton (composite classes)

Helper Structs/Enums:

- Struct Transform

Key Concepts:

- Similar to Unity's GameObject/Component model, with a similar Transform component, for easy scaling, rotating, repositioning
- Allows composable UI via containers and layering.
- Visual behavior (e.g., tinting, scaling) is handled through simple methods used in gameplay (e.g., combat feedback).

Special Features:

- Animation pointer (anim*) for managing active leap transitions.
- Conditional rendering and event handling via visibility and interactivity flags(for Button).

---

animation.h

Purpose: Encapsulates the animation logic using the Tweeny library. Provides smooth interpolation for visual transitions via a UI element's Trasform component.

Classes:

- Animation

Usage:

- Applies to any UIElement's Transform (position, scale, rotation, opacity, etc.).
- Animations are initiated via AddTweeny() and managed with the current scene's UpdateTweens(), that calls all currently visible UIElement's tweeny updates, for smooth animation via delta time.
- Supports transient or persistent end states, using optional after-effects.

Additional Features:

- Contains helper functions in animation.cpp for preset compound animations involving multiple elements (e.g. CombatUltAnim()).

charactersAndEnemies.h

Purpose: Defines the RPG-like character and enemy system, including base and current stats, leveling logic, and verious attack/skill definitions.

Classes:

- Mob (abstract base class)
- Character (derived class)
- Enemy (derived class)

Helper Structs/Enums:

- Struct Buff (with duration, type and amount tracking)
- Struct SkillStruct (with cooldown track and lambda)

Key Concepts:

- Dual-stat system (baseStats and currentStats) for flexibility, easy resets after fight.
- Supports buffs/debuffs, turn countdowns, cooldown management.
- ResetStats() and GetStat() offer runtime control for dynamic stat management during fight.
- Characters and Enemies are loaded from files and persist across sessions.
- Visual representations are deeply tied to gameplay and scene immersion.
- Skill system uses lambda functions for modular design.

---

gameEngine.h

Purpose: Global singleton-style coordinator that wraps player data, persistent state, and scene management.

Classes:

- Game

Key Concepts:

- Provides ChangeScene() and manages active scenes.
- Holds references to reusable UI bundles.

- Initializes player data from save files, including name, materials, character progress.
- Tracks tutorial state for one-time tutorial overlays.

Additional "Features":

- Some parts are designed as a global access point for simplicity, especially useful for cross-scene data like inventory or tutorial state, and preset ui boundles for scenes.

---

filesAndSaving.h

Purpose: Handles file I/O for saving and loading game data.

Key Functions:

- LoadCharactersFromFile()
- LoadEnemyFromFile()
- Save/load materials, progress, and tutorial state.

Key Concepts:

- Tight integration with gameEngine.h, characters, and scene tutorials.
- Text-based file format for ease of debugging and flexibility.

Classes Individual Overview

UIAndDisplay.h – Game Objects

The UIAndDisplay.h file defines the foundational UI components of the game. It establishes a hierarchy of classes to represent various UI elements, enabling modularity, reusability, and ease of maintenance. The design draws inspiration from frameworks like Unity, emphasizing a component-based architecture.

UIElement

Purpose: UIElement serves as the base class for all UI components, encapsulating common properties and behaviors such as positioning, visibility, rendering, and animation.

Key Members:

- std::string name;
- std::shared_ptr<SDL_Texture> texture;
- Transform transform;
- bool visible;
- Animation* anim;

Core Functions:

- **void Render();**
  Renders the UI element using its texture and transform properties.
- **void SetTexture(std::shared_ptr<SDL_Texture> tex);**
  Assigns a new texture to the UI element.
- **void SetVisible(bool isVisible);**
  Sets the visibility state of the UI element.
- **Transform& GetTransform(int x, int y);**
  Helps change the transform of the element.
- **void Tint(SDL_Color color);**
  Applies a color tint to the UI element.
- **void Untint();**
  Removes any applied color tint.
- **void addTweeny(Animation* animation);**
  Assigns an animation to the UI element.

Design Considerations:

- **Smart Pointers**: Utilizes std::shared_ptr for texture management to ensure proper memory handling and avoid leaks.

- **Transform Structure**: Inspired by Unity's Transform component, it encapsulates position, scale, and rotation, promoting a clean and intuitive interface for spatial manipulations.
- **Animation Integration**: The anim pointer allows for dynamic animations, enabling smooth transitions and effects.

Button : UIElement

Purpose: Button extends UIElement to represent interactive buttons with event handling capabilities.

Key Members:

- std::function<void()> onClick;
- std::function<void()> onRelease;
- std::function<void()> onHover;
- bool isActive;
- SDL_Rect clickField;

Core Functions:

- **bool HandleClick/ HandleHover(int mouseX, int mouseY);**
  Processes SDL events to determine if the button was clicked, released, or hovered over, invoking the corresponding callbacks. Returns true if cicked/hovered.

- **void AddOnClick/AddHoverBehaviour(std::function<void()> func);**
  Assigns a functions to be called upon a mouse event.

- **void Disable(bool)**
  Adjusts the activity of a button.

- **Static void *AddBasicScaleUpHoverAnim*(...)**
  Static function, to add a simple scale up hover animation to a Button. Used frequently, hence the preset.

Design Considerations:

- **Lambda Functions**: Embraces the use of lambdas for event callbacks, providing flexibility and encapsulation of behavior.
- **Active State Management**: The isActive flag allows for enabling or disabling button interactions dynamically.
- **Custom Click Field**: The clickField member defines the interactive area, which can differ from the visual representation, offering precise control over user interactions.

Sprite : UIElement

Purpose: Sprite extends UIElement to manage multiple textures, facilitating state changes and animations.

Key Members:

- std::vector<std::string> stateNames;
- std::vector<std::shared_ptr<SDL_Texture>> textures;
- int currentStateIndex;

Core Functions:

- **void AddState(cosnt char* name, std::shared_ptr<SDL_Texture> tex);**
  Adds a new state with the associated texture.
- **void ChangeState(cosnt char* name/int index);**
  Switches to the texture associated with the given state name or index.
- **std::shared_ptr<SDL_Texture> GetTextureAt(cosnt char* name/int index);**
  Returns the texture at a specific state.

Design Considerations:

- **State Management**: Allows for dynamic switching between different visual states, essential for representing animations, status changes or packing different states to a ui element.
- **Texture Organization**: By associating names with textures, it simplifies the process of identifying and switching between states.

Text : UIElement

Purpose: Text extends UIElement to render textual content with customizable properties.

Key Members:

- std::string content;
- TTF_Font* font;
- SDL_Color color;
- int fontSize;

Core Functions:

- **void ChangeText(std::string newText);**
  Updates the displayed text content.

- **void ChangeFont(TTF_Font* newFont);**
  Changes the font used for rendering text.
- **void ChangeColor(SDL_Color newColor);**
  Sets the color of the text.

Design Considerations:

- **Dynamic Text Rendering**: Facilitates real-time updates to text content, essential for displaying changing game information.
- **Customization**: Offers flexibility in appearance through adjustable fonts, sizes, and colors.

OutlinedText : Text

Purpose: OutlinedText inherits from Text to render text with an outline, enhancing readability against various backgrounds.

Key Members:

- SDL_Color outlineColor;
- int outlineSize;

Design Considerations:

- **Enhanced Visibility**: Outlining text improves visibility, especially in visually complex scenes.
- **Separate Class**: By creating a distinct class, it avoids cluttering the base Text class with outline-specific logic.

Bar : UIElement

Purpose: Bar extends UIElement to represent progress indicators like health or mana bars.

Key Members:

- double* valuePtr;
- double maxValue;
- std::shared_ptr<SDL_Texture> barTexture;
- bool dir;

Core Functions:

- **void update();**
  Recalculates the fill level texture based on the current value.

- **void SetCur(double* cur);**
  Assigns the pointer to the value being tracked.
- **void setMax (double max);**
  Sets the maximum value charge for the bar.

Design Considerations:

- **Pointer Usage**: By referencing a value directly, the bar reflects real-time changes without additional updates.
- **Flexible Orientation**: Supports multiple fill directions, accommodating various UI designs.

SpriteButton : Sprite, Button

Purpose: Composite class for interactive buttons with multiple texture states.

TextButton : Text, Button

Purpose: Composite class for interactive buttons with text on top, enabling mass producing similar option buttons (e.g. for tutorials).

scene.h — Scene Management and UI Hierarchy

This header defines the Scene base class and its derived subclasses that represent different UI/gameplay screens in the project.
The scene system is inspired by Unity's scene and LoadScene concept, providing a framework for managing UI bundles, rendering, input handling, and scene switching in a modular and extensible way.

Overarching Design Considerations:

- Scene manages a collection of UI elements (grouped in UIBundle) and controls rendering, input, and updates per frame.

- Scenes encapsulate distinct screens or modes such as menus, character overlays, fight preparation, actual fights, and adventure node selection.

- **A shared static SDL renderer** is used for all UI rendering, ensuring consistency and efficiency.

- **Scene switching** is handled via a global pointer to the active scene; on switching, the new scene's Init() method sets up required state.

- **Input handling**, especially button clicks, used for button onClick callbacks are extensively used for flexibility. However, some scene pointers or UI elements needed inside lambdas are not available at compile time. To workaround this, some buttons use empty lambdas initially. Scenes override CheckButtonsOnClick() to detect clicks manually and trigger the required actions. This approach maintains clean separation of concerns and avoids messy global state or premature captures.

- **Static UI bundles and initialization.** Most UI elements (textures, positions) are initialized once in main(). Static bundles hold frequently reused UI components shared across scenes. This avoids repeated loading and setup, improving performance and consistency. Passing UI elements explicitly between classes would be messier, so static bundles act as centralized shared resources.

Scene

Purpose: The abstract base class representing a generic scene/screen.

Key members:

- std::string name;
- SDL_Texture* bg;
- std::vector<UIElement*> uiBundle;
- Scene::static SDL_Renderer* renderer

Core functions:

- **Scene(const std::string& name, std::shared_ptr<SDL_Texture> background)**
  Constructor initializes scene name and background texture.
- **virtual void Init()**
  Virtual method called after scene switching to initialize scene-specific data or UI layout. Override in derived classes for custom setup.

- **virtual void LoadScene()**
  Called every update cycle to render all UI elements in the scene's bundle. Base implementation renders all elements; override for extra rendering steps.
- **virtual void RenderOne()**
  Used to render UI elements that must be on top (e.g., overlays).
- **virtual void UpdateTweens(float deltaTime)**
  Updates all animations/tweening of UI elements based on delta time. Ensures smooth animation updates every frame.
- **virtual void CheckButtonsOnClick/Hover(int mouseX, int mouseY)**
  Cycles thourgh the ui boundle, and calls each Button's HandleEvent functions. Overriden in some cases, for more complex button event handling.

Design Considerations:

- **Static Renderer:** The use of a shared static renderer ensures all scenes and UI elements render to the same SDL window context.
- **Virtual methods:** Allow each scene to customize rendering and input logic as needed.
- **UIBundles** are used to organize elements hierarchically, mirroring Unity's GameObject hierarchy.

ListingScene : Scene

Purpose: Used to display a list of selectable options, primarily designed for the tutorials menu.

Key Members:

- std::vector<Button*> triggers;
- std::vector<std::vector<UIElement*>> menus;

Core functions:

- **void Init() override**
  Custom initialization for ListingScene, resetting or preparing the menu display.
- **void ChangeMenu(int index/ const char* name)**
  Switches the displayed menu to the one at the index, or the trigger buttons name. Called by the triggers' lambdas when a button is clicked.

Design Considerations:

- Enables simple menu selection with a clear separation of trigger buttons and content.
- Designed for menus with relatively static option sets and content.

CharOverview : ListingScene

Purpose: Specialized LsitingScene to display character stats and skills in an overview format.

Key Members:

- Character* currentCharacter;
- std::vector<Text*> statTexts;

Core functions:

- **void Init() override**
  Prepares the overlay by resetting and updating displayed stats.

- **void UpdateCharacterDisplay(Character* newCharacter)**
  Updates text and UI elements to reflect the selected character's stats. Instead of creating separate bundles per character, dynamically updates existing UI elements for efficiency.

Design Considerations:

- **Separate class:** Inherits ListingScene's trigger/menu switching functionality but adds character-specific dynamic content.
- **Dynamic text:** Avoids excessive UI bundle creation by updating text elements in place.
- Keeps a reference to the currently selected character for consistent visual updates.

PreFight : Scene

Purpose: Scene for selecting your party members before entering battle.

Key Members:

- UIBundle charSlotCont — container holding UI elements representing all available characters.
- std::vector<Character*> selectedCharacters — tracks currently selected party members (max 3).

Core functions:

- **void Init() override**
  Sets up character slots and selection state.
- **void AddCharacter(Character* newCharacter)**
  Adds or removes a character from the party by index, enforcing max party size of 3.
- **void StartFight()**
  Instantiates a new Fight scene (Fight*) with the selected party (of 3) and preloaded enemies (from file) and starts the battle.

Design Considerations:

- Allows dynamic party composition with simple toggling mechanics.
- Enforces party size constraints programmatically, adding intuitive visuals.

AdventuresMenu : Scene

Purpose: Scene displaying a map of fight nodes representing different battle encounters.

Key Members:

- std::vector<FightNode*> fightNodes — the nodes representing battles, loaded from external data files on startup.
- **Fight nodes contain:**
  - Enemy data from file
  - Pointer to their corresponding PreFight scene, due to consistent enemies, battle setup in the same node.
  - Attempt count and star ratings for tracking player progress.
  - Located in adventure.h

Core functions:

- **void Init() override**
  Loads fight nodes and sets up the UI for node selection.
- **void UpdateLastNode(FightNode* node, int stars, bool attempted)**
  Updates node data and saves progress using FightNode's WriteNodeDataToFile().
- **bool CheckButtonsOnClick(int mouseX, int mouseY) override**
  Manages button clicks to select nodes and start fights, again with manual click handling due to runtime pointer capture limitations.

Design Considerations:

- Acts as a container and controller for progression through fights.

- Loads and saves persistent data externally to keep node state consistent across runs.
- Uses static UI bundles shared across scenes for efficiency.

class Fight- Turn based logic and user friendly visualization

Overview

The Fight class is a central component of the game's combat system, inheriting from the Scene class. It encapsulates the turn-based combat mechanics, managing both the visual representation and the underlying logic of battles. The class orchestrates the flow of combat, handling player and enemy actions, updating the UI accordingly, and ensuring a cohesive gaming experience.

Fight : Scene

Key Members:

- **UI Containers**

Purpose: These containers group related UI elements, facilitating organized rendering and updates, easier access to frequently used elements, lessening the need to cycle through each ui boundle element

  - **std::vector<std::vector<UIElement*>> combatChConts:** Container for character-related UI elements.
  - **std::vector<UIElement*> combatNATypeCont:** Container for normal attack type indicators.
  - **std::vector<UIElement*> animProps:** Container for animation properties and effects.
  - **std::vector<UIElement*> turnOrderCont:** Container for turn indicators.
  - **UIElement* combatTargetIndicator** – visual element for current enemy target

- **Turned based logic elements**

Purpose: These elements are used to make it easier to implement the turned based logic, giving feedback of the current battle state to the user.

- **std::vector<Mob*> turnOrderFIFO:** A queue representing the order of upcoming actions.
- **Character* currentlyActing**: Pointer to the character currently taking action.
- **Enemy* currentlyActingEnemy**: Pointer to the enemy currently taking action.
- **CombatState combatState**: Helper enum:

```
enum CombatState {
    • ENEMYACTION,
    • WAITINGFORACTION,
    • WAITINGFORTYPE,
    • WAITINGFORTARGET,
    • OVER,
    • WAITINGFORENEMYANIM
};
```

  Used in main.cpp's update loop, for a state machine-like input handling.

- **double AV:** A double value tracking the action value, used to calculate action order via speed.

Core methods:

- **Initialization and Setup**
  - **void Init()**

    Overrides the base Scene initialization to set up combat-specific elements.

  - **void CalculateNext()**

    Calculates the upcoming actions based on character and enemy speed stats, populating the turnOrderFIFO.

  - **void CountNextAction()**

    Processes the next action in the queue, updating the currentlyActing or currentlyActingEnemy pointers and setting the appropriate combatState.

- **Damage, Character and Enemy action deligating**
  - **void Normal/Skill/Ult()**

    Calculates the target of the currentlyActing character, and calls their respective attack pattern functions. Displays the got damage and it's properties to the screen.

  - **void EnemyAtk/EnemyAction()**

    Calculates the enemies target, and implements the AI logic of enemy actions, while displaying it's corresponding visuals.

  - **void AfterActionCleanup /CheckIfOver/End()**

    Processes the visuals, logic after an enemy or character action. Checks if the game is over, updating the combat state and visuals accordingly.

    Design Considerations

    - **UI Containers**: Inspired by Unity's empty GameObjects, these containers provide a structured way to manage related UI elements, simplifying updates and rendering.
    - **Turn Order Queue**: Maintaining a FIFO queue of upcoming actions allows for dynamic turn order calculations based on character stats, adding strategic depth to combat.
    - **Combat States**: Using an enumeration to represent combat states enables a clear and manageable flow of combat phases, facilitating input handling and state transitions.
    - **Visual Feedback**: A lot of helper functions dedicated to visual updates enhance player engagement by providing immediate and intuitive feedback on actions and state changes.
    - **Input Handling**: Centralizing input processing within the Fight class ensures that combat-specific inputs are handled appropriately, maintaining separation of concerns.

Final Notes

The Fight class serves as the backbone of the game's combat system, integrating UI management, turn-based logic, and player interaction into a cohesive module. Its design reflects a balance between functionality and maintainability, drawing

inspiration from established game development practices to deliver an engaging combat experience.

charactersAndEnemies.h – Entities with RPG style behavior control

The combat system's foundation relies on the Mob abstraction, which encapsulates shared behaviors of both player-controlled characters and AI enemies. This approach enforces clean inheritance and enables scalable combat logic.

Overarching Design Considerations:

- All combatants inherit from Mob, allowing them to maintain:
  - Core battle-related stats,
  - A mechanism for dynamic changes during battle (like buffs),
  - A visual representation for both gameplay and narrative menus,
  - A robust system for stat restoration and level progression.
- **Stat Separation**: Combat depends on easily changable current stats (currentStats) derived from persistent base stats (baseStats). This duality makes stat manipulation (buffs, debuffs, damage) clean and reversible.
- **Buff System**: A time-bound system for stat modifications that expire and revert properly, supporting deep turn-based mechanics.
- **Smart Visual Design**: Every Mob includes a sprite loaded via a name-based asset pathing convention, simplifying visual logic across both enemies and player characters.
- **Save-Compatibility**: Characters are loaded and saved using helper functions (see filesAndSaving.h) that persist levels, stats, and progress across sessions.

Mob (abstract)

Purpose: Serves as the base class for all entities that participate in battle: both Character and Enemy. It provides shared mechanisms for stat handling, damage, buffs, leveling, and visuals.

Helper Structs/Enums:

- **struct Buff** {
  std::string type;
  double amount;
  int turnsLeft;
  };
- **struct Dmg** {
  double amount ;
  bool isCrit;
  int starsGenerated;
  };

Key Members:

- std::string name
- Sprite* sprite
- various doubles baseStats;
- various doubles currentStats;
- std::vector<Buff> buffs
- bool isDead
- int level

Core Functions:

- **void SetHp(double amount)**
  Reduces HP; if HP drops to 0 or below, sets isDead = true.
- **void ResetStats()**
  Resets currentStats to baseStats; called at start of combat or post-buff cleanup.
- **double& GetStat(const std::string& statName)**
  Provides direct mutable reference to a specific stat in currentStats. Useful for buffs/debuffs.
- **void AddBuff(Buff buff)**
  Adds a new temporary buff to the Mob.
- **void EraseBuff()**
  Iterates over buffs and removes them if expired, restoring original stat values.
- **void LevelUp()**
  Increases level and uses a logarithmic formula to scale base stats accordingly.

Design Considerations

- The use of GetStat() returning a double& is key to making buff and damage logic clean, avoiding unnecessary duplication or boilerplate.
- Buffs being managed as a vector allows stacking and turn-based expiration.
- The sprite is embedded directly in Mob, which simplifies loading visuals using the name as a filepath component.

Enemy : Mob

Purpose: Represents AI-controlled enemies in combat. Unlike Characters, they are not player-interactive and have hardcoded attack types with minimal external influence.

Key Members

- Added def and critRate current and base stats.

Core Methods

- **void Atk1/2/3(Character* target)**
  These methods execute set attack patterns on the given target.

Design Considerations

- Despite being simpler than Character, the inclusion of different attack types adds strategic depth and variety.

Character : Mob

Purpose: Represents the player's controllable units in combat. Supports complex mechanics including skills, ultimates, visual variants, leveling, and persistence via file I/O.

Key Members:

- Various extra doubles added for character specific base nd current stats.
- SDL_Color color - Unique color for this character, used in visual indicators and UI elements.
- SkillStruct skill – skill with costumized lambdas and a cooldown machnic.

Helper Structs/Enums:

- **struct SkillSturct** {
  std::function<void(std::vector<Enemy*>
  targets,std::vector<Character*> allies, Character* herself)> effect;

```
        int cooldown;
        int activeCooldown;
    };
```

- std::function<float(std::vector<Enemy*> targets,std::vector<Character*> allies, Character* herself)> ult;

Core Methods:

- **void CountdownTurn()**
  Called each turn to reduce cooldown and check buffs.
- **Dmg Normal(...)**
  Executes one of three attack styles using CardType(Buster, Arts, Quick) enum. Each Normal type has slightly different abilities, defined by the TypeMultipler struct.
- **void Skill(...)**
  Executes the skill's effect if cooldown is ready.
- **Dmg Ult(...)**
  Executes the ultimate ability, if energy level is enough.

Design Considerations:

- Sprites makes it trivial to swap a character's visual depending on game context (menu, combat, overview, etc.).
- Skills being cooldown-based encourages planning and cooldown management.
- Skills and ultimates use std::function, allowing for flexible behavior without inheritance bloat.
- Color is an efficient visual cue to identify characters during combat and transitions.
- Characters are loaded from file via LoadCharactersFromFile() and saved via SaveCharactersToFile() (see filesAndSaving.h), preserving progress like levels and cooldown states.

animation.h – Dynamic and static leaping animations

The animation system handles smooth UI element transitions using Tweeny — a lightweight C++ tweening library. Animations interpolate Transform properties (position, scale, opacity, rotation, etc.) over time with custom easing functions. It integrates deeply with the UI system by directly modifying Transform&, enabling reusable, lightweight, and fluid animations.

Overarching Design Considerations:

- **Pointer-Based Updating**: Animations modify the actual UI state in-place through Transform&, eliminating sync complexity.
- **Completion-Tracking**: Each animation reports when it's finished (Update(dt) returns true), so UI elements and scenes can know when they're idle.
- **After-State Handling**: Supports temporary visual effects by allowing a post-animation "afterTransform" to apply once the main animation completes.
- **Scene-Aware Control**: Scenes (like Fight) query whether a UI element is still animating, by checking if its anim pointer is non-null.

Animation

Purpose: Handles smooth transition of a UI element's Transform from its current state to a target state over a set duration and easing.

Key Members:

- Transform* target;
- Transform afterState; - Optional transform applied immediately after animation completes (for transient effects).
- std::vector<tweeny::tween<float>> transformTweens;

Core Functions:

- **Animation(Transform* target, const Transform& result, double duration, easing, optional after)**
  Initializes a new animation on the given Transform*, setting the desired result state, duration and easing.
- **bool Update(float dt)**
  Advances the animation by dt seconds, interpolates the target Transform fields. Returns true if the animation has finished (including any post-processing via afterState).

Design Considerations:

- Animation modifies the UI element directly via pointer, making it stateless in usage and ideal for one-off or chained effects.
- Update() being a bool-returning function makes it trivial for UI systems to manage animation lifecycle.
- When Update() returns true, the owning UIElement clears its anim pointer — a simple and effective way to know if it's idle or in transition.

Integration in UIElement

Each UIElement includes:

- Animation anim
  When non-null, an animation is active and Update(dt) is called.
- void AddTweeny(...)
  Sets up a new animation. Replaces any existing animation in anim.

This makes animations easy to assign and manage per-element. Since anim is checked against nullptr, scenes can easily pause input or actions (e.g., in Fight) until all animations are finished.

Animation Preset Functions (in animation.cpp)

Purpose: There are three standalone functions that orchestrate animations for multiple UI elements simultaneously — e.g., sliding in/out panels, skill an ult effects, etc.Provide reusable complex UI animations using multiple Tweeny instances internally.

Final Notes

This animation system balances simplicity and flexibility. It integrates well with the existing scene/UI architecture, tracks animation state cleanly, and supports layered visual polish like after-effects. With just Transform, Animation, and some smart update logic, it brings UI to life without overwhelming the codebase.

gameEngine.h – Global datas and player profile handling

Overarching Design Considerations

- Game is the central orchestrator of your entire system. It's a singleton-like wrapper that:
  - Initializes key game data (like player info and progression),
  - Controls scene transitions (ChangeScene),
  - Manages shared state across all systems (like materials, tutorial flags, etc.),

- Serves as a static access point for common UI containers used in multiple scenes.
- It's intentionally global in scope — not in the C++ static sense, but in design — because it holds game-wide truth: what the player owns, where they are, and what they've already seen. While this may seem “weird” or “everything-at-once,” it's actually quite fitting for a single-player, scene-driven game with persistent state and UI. A built global game context, and that’s exactly what this is.

Game

Purpose: Acts as a central runtime context for the game. Responsible for:

- Holding persistent player state (name, materials),
- Handling scene transitions (ChangeScene()),
- Tracking one-time tutorial progress,
- Storing static UI containers used across multiple scenes.

Helper Structs/Enums:

```
struct GameMats {
        int tier1 = 0;
        int tier2 = 0;
        int tier3 = 0;
};
```

Key Members:

- static Scene* currentScene;
- static preset ui boundles for various scenes.
- bool MM/CP/AD/PF Tutorials - Four booleans representing whether the player has completed one-time tutorials for specific scenes. Loaded from and saved to file.
- GameMats mats;

Core Functions:

- **void Set/GetGameMats ()**
  Enables acesseing the players materials for e.g. leveling up characters.
- **void Set/GetTutorialNeed()**
  Acessing one-time tutorial need, and adjusting it when done.

One-Time Tutorial System

Purpose: A very thoughtful and player-friendly feature.

Overview:

- Every scene (Scene, Fight, CharacterOverview, etc.) has:
    - A **tutorial overlay** (Sprite*),
    - A **bool flag** indicating whether the tutorial should be shown.
- On first entry, the scene checks with Game if the tutorial for that scene has already been completed:
    - If **not**, it activates the overlay and disables other input.
    - Once the player interacts with the tutorial (e.g., dismisses it), the scene sets a local tutorialsActive= true, then saves it to local files.
- This is remembered using the save file — only ever played once, unless the player manually resets progress.

Design Considerations:

- Scene behavior is adjusted while tutorials are active — they "lock" other buttons/interactions to prevent confusion.
- A player-first, UX-minded approach. Tutorial-only input mode is smart and avoids overwhelming players.

Overarching Design Considerations:

- Scene-Driven, Global State Management: Game is the glue — enabling clean per-scene design while still offering persistent state access across everything.
- Minimal and Focused: Despite its central role, it avoids bloating by only holding essentials: scene control, player identity, game materials, and tutorial state.
- Expandable: Easy to add more static containers or flags in the future (e.g., daily rewards, settings).
- Safe Static Usage: Since everything is static, it’s globally accessible — but controlled through encapsulated interfaces like InitGame() and ChangeScene().

Final Notes

While unconventional at first glance, Game is actually a clean solution for what this game needs: Persistent state across completely separate scenes. A unified way to handle transition logic. Global knowledge of the player's progress (tutorials, materials, etc.). It’s clear, well-integrated, and ready for expansion.