Tarea 02 - Videojuegos - Henry Campos

Generated by Doxygen 1.9.8

1 Class Documentation

1.1 AnimationComponent Struct Reference

Component for handling sprite animations in the ECS.

```
#include <AnimationComponent.hpp>
```

Public Member Functions

- AnimationComponent (int numFrames=1, int frameSpeedRate=1, bool isLoop=true)

  Constructs an AnimationComponent with specified parameters.

Public Attributes

- int **numFrames**

  Total number of frames in the animation.
- int **currentFrame**

  Current frame of the animation.
- int **frameSpeedRate**

  Speed at which frames are updated (in milliseconds).
- bool **isLoop**

  Indicates whether the animation should loop.
- int **startTime**

  Time when the animation started, in milliseconds.

1.1.1 Detailed Description

Component for handling sprite animations in the ECS.

1.1.2 Constructor & Destructor Documentation

AnimationComponent()

```
AnimationComponent::AnimationComponent (
              int numFrames = 1,
              int frameSpeedRate = 1,
              bool isLoop = true )  [inline]
```

Constructs an AnimationComponent with specified parameters.

Parameters

numFrames	Total number of frames in the animation (default: 1).
frameSpeedRate	Speed of frame updates in milliseconds (default: 1).
isLoop	Whether the animation should loop (default: true).

The documentation for this struct was generated from the following file:

- Components/AnimationComponent.hpp

1.2 AnimationSystem Class Reference

System that handles sprite animation.

```
#include <AnimationSystem.hpp>
```

Public Member Functions

- AnimationSystem ()

  Construct a new Animation System object.
- void Update ()

  Update all animated entities.

Public Member Functions inherited from System

- **System** ()=default

  Default constructor for the system.
- ∼**System** ()=default

  Default destructor for the system.
- void AddEntityToSystem (Entity entity)

  Adds an entity to the system.
- void RemoveEntityFromSystem (Entity entity)

  Removes an entity from the system.
- std::vector< Entity > GetSystemEntities () const

  Gets the list of entities managed by this system.
- const Signature & GetComponentSignature () const

  Gets the component signature of the system.
- template<typename TComponent >
  void RequireComponent ()

  Specifies a required component for the system.

1.2.1 Detailed Description

System that handles sprite animation.

Updates animation frames for entities with both AnimationComponent and SpriteComponent based on elapsed time and animation parameters.

1.2.2 Constructor & Destructor Documentation

AnimationSystem()

```
AnimationSystem::AnimationSystem ( )  [inline]
```

Construct a new Animation System object.

Requires entities to have both AnimationComponent and SpriteComponent

1.2.3 Member Function Documentation

Update()

```
void AnimationSystem::Update ( )  [inline]
```

Update all animated entities.

Calculates current animation frame based on:

- Time elapsed since animation started
- Frame speed rate
- Total number of frames Updates sprite source rectangle to show current frame

The documentation for this class was generated from the following file:

- Systems/AnimationSystem.hpp

1.3 AssetManager Class Reference

Manages game assets such as textures, fonts, and sounds.

```
#include <AssetManager.hpp>
```

Public Member Functions

- **AssetManager** ()

  Constructs an AssetManager instance.
- ∼**AssetManager** ()

  Destroys the AssetManager and frees all loaded assets.
- void **ClearAssets** ()

  Clears all loaded assets from memory.
- void AddTexture (SDL_Renderer ∗renderer, const std::string &textureId, const std::string &filePath)

  Loads and adds a texture to the manager.
- SDL_Texture ∗ GetTexture (const std::string &textureId)

  Retrieves a texture by its ID.
- void AddFont (const std::string &fontId, const std::string &filePath, int fontSize)

  Loads and adds a font to the manager.
- TTF_Font ∗ GetFont (const std::string &fontId)

  Retrieves a font by its ID.
- void AddSound (const std::string &soundId, const std::string &filePath)

  Loads and adds a sound effect to the manager.
- Mix_Chunk ∗ GetSound (const std::string &soundId)

  Retrieves a sound effect by its ID.

1.3.1 Detailed Description

Manages game assets such as textures, fonts, and sounds.

1.3.2 Member Function Documentation

AddFont()

```
void AssetManager::AddFont (
            const std::string & fontId,
            const std::string & filePath,
            int fontSize )
```

Loads and adds a font to the manager.

Parameters

fontId	The unique identifier for the font.
filePath	The file path to the font file.
fontSize	The size of the font.

AddSound()

```
void AssetManager::AddSound (
            const std::string & soundId,
            const std::string & filePath )
```

Loads and adds a sound effect to the manager.

Parameters

sound↩ Id	The unique identifier for the sound.
filePath	The file path to the sound file.

AddTexture()

```
void AssetManager::AddTexture (
            SDL_Renderer * renderer,
            const std::string & textureId,
            const std::string & filePath )
```

Loads and adds a texture to the manager.

Parameters

renderer	The SDL renderer used to create the texture.
texture↩ Id	The unique identifier for the texture.
filePath	The file path to the texture image.

GetFont()

```
TTF_Font * AssetManager::GetFont (
             const std::string & fontId )
```

Retrieves a font by its ID.

Parameters

font↩ Id	The unique identifier of the font.

Returns

Pointer to the TTF_Font, or nullptr if not found.

GetSound()

```
Mix_Chunk * AssetManager::GetSound (
             const std::string & soundId )
```

Retrieves a sound effect by its ID.

Parameters

sound↩ Id	The unique identifier of the sound.

Returns

Pointer to the Mix_Chunk, or nullptr if not found.

GetTexture()

```
SDL_Texture * AssetManager::GetTexture (
             const std::string & textureId )
```

Retrieves a texture by its ID.

Parameters

texture↩ Id	The unique identifier of the texture.

Returns

Pointer to the SDL_Texture, or nullptr if not found.

The documentation for this class was generated from the following files:

- AssetManager/AssetManager.hpp
- AssetManager/AssetManager.cpp

1.4 ChargeManageSystem Class Reference

Manages charge systems for various game mechanics.

```
#include <ChargeManageSystem.hpp>
```

Public Member Functions

- ChargeManageSystem ()

  Construct a new Charge Manage System object.
- void Update ()

  Update all charge components.
- bool HasSufficientCharge (int colorIndex)

  Check if sufficient charge exists for a specific type.
- bool ConsumeChargeForDrawing (int colorIndex)

  Attempt to consume charge for an operation.

Public Member Functions inherited from System

- **System** ()=default

  Default constructor for the system.
- ∼**System** ()=default

  Default destructor for the system.
- void AddEntityToSystem (Entity entity)

  Adds an entity to the system.
- void RemoveEntityFromSystem (Entity entity)

  Removes an entity from the system.
- std::vector< Entity > GetSystemEntities () const

  Gets the list of entities managed by this system.
- const Signature & GetComponentSignature () const

  Gets the component signature of the system.
- template<typename TComponent >
  void RequireComponent ()

  Specifies a required component for the system.

1.4.1 Detailed Description

Manages charge systems for various game mechanics.

Handles natural recharge and consumption of different charge types:

- Damage charge
- Sprint charge
- Slow charge Uses time-based updates for natural recharge.

1.4.2 Constructor & Destructor Documentation

ChargeManageSystem()

```
ChargeManageSystem::ChargeManageSystem ( )  [inline]
```

Construct a new Charge Manage System object.

Initializes the last recharge time to current time

1.4.3 Member Function Documentation

ConsumeChargeForDrawing()

```
bool ChargeManageSystem::ConsumeChargeForDrawing (
             int colorIndex )  [inline]
```

Attempt to consume charge for an operation.

Parameters

colorIndex	The charge type to consume (0=Damage, 1=Sprint, 2=Slow)

Returns

true If charge was successfully consumed

false If insufficient charge was available

HasSufficientCharge()

```
bool ChargeManageSystem::HasSufficientCharge (
             int colorIndex )  [inline]
```

Check if sufficient charge exists for a specific type.

Parameters

colorIndex	The charge type to check (0=Damage, 1=Sprint, 2=Slow)

Returns

true If charge meets minimum requirement

false If charge is insufficient

Update()

```
void ChargeManageSystem::Update ( )  [inline]
```

Update all charge components.

Performs natural recharge of all charge types at fixed intervals when they're not fully charged.

The documentation for this class was generated from the following file:

- Systems/ChargeManageSystem.hpp

1.5 CircleColliderComponent Struct Reference

Component for defining a circular collider in the ECS.

```
#include <CircleColliderComponent.hpp>
```

Public Member Functions

- CircleColliderComponent (int radius=0, int width=0, int height=0)

  Constructs a CircleColliderComponent with specified parameters.

Public Attributes

- int **radius**

  Radius of the circular collider.
- int **width**

  Width of the collider's bounding box.
- int **height**

  Height of the collider's bounding box.

1.5.1 Detailed Description

Component for defining a circular collider in the ECS.

1.5.2 Constructor & Destructor Documentation

CircleColliderComponent()

```
CircleColliderComponent::CircleColliderComponent (
                int radius = 0,
                int width = 0,
                int height = 0 )  [inline]
```

Constructs a CircleColliderComponent with specified parameters.

Parameters

radius	The radius of the circular collider (default: 0).
width	The width of the collider's bounding box (default: 0).
height	The height of the collider's bounding box (default: 0).

The documentation for this struct was generated from the following file:

- Components/CircleColliderComponent.hpp

1.6 ClickableComponent Struct Reference

Component for handling click interactions in the ECS.

```
#include <ClickableComponent.hpp>
```

Public Member Functions

- **ClickableComponent** ()

  Constructs a ClickableComponent with default values.

Public Attributes

- bool **isClicked**

  Indicates whether the entity has been clicked.

1.6.1 Detailed Description

Component for handling click interactions in the ECS.

The documentation for this struct was generated from the following file:

- Components/ClickableComponent.hpp

1.7 ClickEvent Class Reference

Event representing a mouse click action.

```
#include <ClickEvent.hpp>
```

Public Member Functions

- ClickEvent (int buttonCode=0, int x=0, int y=0)

  Construct a new Click Event object.

Public Member Functions inherited from Event

- **Event** ()=default

  Construct a new Event object (default)
- virtual ∼**Event** ()=default

  Destroy the Event object (virtual for proper inheritance)

Public Attributes

- int **buttonCode**

  SDL button code of the mouse button clicked.
- int **x**

  X-coordinate of the click position.
- int **y**

  Y-coordinate of the click position.

1.7.1 Detailed Description

Event representing a mouse click action.

Contains information about the mouse button clicked and the screen coordinates where the click occurred. Inherits from the base Event class for use with EventManager.

1.7.2 Constructor & Destructor Documentation

ClickEvent()

```
ClickEvent::ClickEvent (
              int buttonCode = 0,
              int x = 0,
              int y = 0 )  [inline]
```

Construct a new Click Event object.

Parameters

buttonCode	SDL button code (default: 0)
x	X-coordinate of click (default: 0)
y	Y-coordinate of click (default: 0)

The documentation for this class was generated from the following file:

- Events/ClickEvent.hpp

1.8 CollisionEvent Class Reference

Event representing a collision between two entities.

```
#include <CollisionEvent.hpp>
```

Public Member Functions

- CollisionEvent (Entity entityA, Entity entityB)

  Construct a new Collision Event object.
- ∼**CollisionEvent** ()

  Destroy the Collision Event object.

Public Member Functions inherited from Event

- **Event** ()=default

    Construct a new Event object (default)
- virtual ∼**Event** ()=default

    Destroy the Event object (virtual for proper inheritance)

Public Attributes

- Entity **entityA**

    First entity involved in the collision.
- Entity **entityB**

    Second entity involved in the collision.

1.8.1 Detailed Description

Event representing a collision between two entities.

Contains references to both entities involved in the collision. Inherits from the base Event class for use with the EventManager system.

1.8.2 Constructor & Destructor Documentation

CollisionEvent()

```
CollisionEvent::CollisionEvent (
            Entity entityA,
            Entity entityB )  [inline]
```

Construct a new Collision Event object.

Parameters

entityA	First entity in the collision
entityB	Second entity in the collision

The documentation for this class was generated from the following file:

- Events/CollisionEvent.hpp

1.9 CollisionSystem Class Reference

Handles circular collision detection between entities.

```
#include <CollisionSystem.hpp>
```

Public Member Functions

- CollisionSystem ()

  Construct a new Collision System object.
- void Update (std::unique_ptr< EventManager > &eventManager)

  Update collision detection for all entities.

Public Member Functions inherited from System

- **System** ()=default

  Default constructor for the system.
- ∼**System** ()=default

  Default destructor for the system.
- void AddEntityToSystem (Entity entity)

  Adds an entity to the system.
- void RemoveEntityFromSystem (Entity entity)

  Removes an entity from the system.
- std::vector< Entity > GetSystemEntities () const

  Gets the list of entities managed by this system.
- const Signature & GetComponentSignature () const

  Gets the component signature of the system.
- template<typename TComponent >
  void RequireComponent ()

  Specifies a required component for the system.

1.9.1 Detailed Description

Handles circular collision detection between entities.

Detects collisions between entities with CircleColliderComponent and TransformComponent, emitting Collision↩ Events when collisions occur.

1.9.2 Constructor & Destructor Documentation

CollisionSystem()

```
CollisionSystem::CollisionSystem ( )  [inline]
```

Construct a new Collision System object.

Requires entities to have both CircleColliderComponent and TransformComponent

1.9.3 Member Function Documentation

Update()

```
void CollisionSystem::Update (
            std::unique_ptr< EventManager > & eventManager )  [inline]
```

Update collision detection for all entities.

Parameters

eventManager	Reference to the event manager for emitting collision events

The documentation for this class was generated from the following file:

- Systems/CollisionSystem.hpp

1.10 Component< TComponent > Class Template Reference

Templated base class for components, providing unique ID generation.

```
#include <ECS.hpp>
```

Static Public Member Functions

- static int GetId ()

  Gets the unique ID for the component type.

Additional Inherited Members

Static Protected Attributes inherited from IComponent

- static int **nextId** = 0

  Static counter for assigning unique component IDs.

1.10.1 Detailed Description

template<typename TComponent>
class Component< TComponent >

Templated base class for components, providing unique ID generation.

Template Parameters

TComponent	The specific component type.

1.10.2 Member Function Documentation

GetId()

```
template<typename TComponent >
static int Component< TComponent >::GetId ( )  [inline], [static]
```

Gets the unique ID for the component type.

Returns

The component's unique ID.

The documentation for this class was generated from the following file:

- ECS/ECS.hpp

1.11 ControllerManager Class Reference

Manages keyboard and mouse input states.

```
#include <ControllerManager.hpp>
```

Public Member Functions

- **ControllerManager** ()
  Construct a new Controller Manager object.
- ∼**ControllerManager** ()
  Destroy the Controller Manager object.
- void **Clear** ()
  Clears all input states and mappings.
- void AddActionKey (const std::string &action, int keyCode)
  Add a keyboard action mapping.
- void KeyDown (int keyCode)
  Set key state to pressed.
- void KeyUp (int keyCode)
  Set key state to released.
- bool IsActionActivated (const std::string &action)
  Check if an action is currently active (key pressed)
- void AddMouseButton (const std::string &action, int buttonCode)
  Add a mouse button action mapping.
- void MouseButtonDown (int buttonCode)
  Set mouse button state to pressed.
- void MouseButtonUp (int buttonCode)
  Set mouse button state to released.
- bool IsMouseButtonDown (const std::string &name)
  Check if a mouse button action is pressed.
- void SetMousePosition (int x, int y)
  Update mouse cursor position.
- std::tuple< int, int > GetMousePosition ()
  Get current mouse cursor position.

1.11.1 Detailed Description

Manages keyboard and mouse input states.

Tracks keyboard actions, mouse buttons, and cursor position with action-name mapping functionality.

1.11.2 Member Function Documentation

AddActionKey()

```
void ControllerManager::AddActionKey (
             const std::string & action,
             int keyCode )
```

Add a keyboard action mapping.

Parameters

action	Name of the action to map
keyCode	SDL key code for the action

AddMouseButton()

```
void ControllerManager::AddMouseButton (
            const std::string & action,
            int buttonCode )
```

Add a mouse button action mapping.

Parameters

action	Name of the action to map
buttonCode	SDL mouse button code

GetMousePosition()

```
std::tuple< int, int > ControllerManager::GetMousePosition ( )
```

Get current mouse cursor position.

Returns

std::tuple<int, int> Current (x, y) coordinates

IsActionActivated()

```
bool ControllerManager::IsActionActivated (
            const std::string & action )
```

Check if an action is currently active (key pressed)

Parameters

action	Name of the action to check

Returns

true if the action's key is pressed

false otherwise

IsMouseButtonDown()

```
bool ControllerManager::IsMouseButtonDown (
             const std::string & name )
```

Check if a mouse button action is pressed.

Parameters

name	Name of the mouse action to check

Returns

true if the button is pressed

false otherwise

KeyDown()

```
void ControllerManager::KeyDown (
             int keyCode )
```

Set key state to pressed.

Parameters

keyCode	SDL key code of the pressed key

KeyUp()

```
void ControllerManager::KeyUp (
             int keyCode )
```

Set key state to released.

Parameters

keyCode	SDL key code of the released key

MouseButtonDown()

```
void ControllerManager::MouseButtonDown (
             int buttonCode )
```

Set mouse button state to pressed.

Parameters

buttonCode	SDL mouse button code

MouseButtonUp()

```
void ControllerManager::MouseButtonUp (
            int buttonCode )
```

Set mouse button state to released.

Parameters

buttonCode	SDL mouse button code

SetMousePosition()

```
void ControllerManager::SetMousePosition (
            int x,
            int y )
```

Update mouse cursor position.

Parameters

x	New X coordinate
y	New Y coordinate

The documentation for this class was generated from the following files:

- ControllerManager/ControllerManager.hpp
- ControllerManager/ControllerManager.cpp

1.12 DamageChargeComponent Struct Reference

Component for managing a charge-based damage system in the ECS.

```
#include <DamageChargeComponent.hpp>
```

Public Member Functions

- DamageChargeComponent (int total=100, int initialCharge=100)

  Constructs a DamageChargeComponent with specified total and initial charge.
- void **updateChargeDisplay** ()

  Updates the charge display string to reflect current and total charge.
- void **Recharge** ()

Fully recharges the component to its total capacity.
- void Charge (int amount)

  Adds a specified amount to the current charge, capped at total capacity.
- void Discharge (int amount)

  Removes a specified amount from the current charge, preventing negative values.
- float GetPercentage () const

  Calculates the current charge as a percentage of the total capacity.
- bool IsFullyCharged () const

  Checks if the charge is at full capacity.
- bool IsEmpty () const

  Checks if the charge is depleted.

Public Attributes

- int **totalCharge**

  Total charge capacity.
- int **currentCharge**

  Current charge level.
- std::string **chargeDisplay**

  String representation of the charge level in the format "currentCharge/totalCharge".

1.12.1 Detailed Description

Component for managing a charge-based damage system in the ECS.

1.12.2 Constructor & Destructor Documentation

DamageChargeComponent()

```
DamageChargeComponent::DamageChargeComponent (
            int total = 100,
            int initialCharge = 100 )  [inline]
```

Constructs a DamageChargeComponent with specified total and initial charge.

Parameters

total	The total charge capacity (default: 100).
initialCharge	The initial charge level (default: 100).

1.12.3 Member Function Documentation

Charge()

```
void DamageChargeComponent::Charge (
            int amount )  [inline]
```

Adds a specified amount to the current charge, capped at total capacity.

Parameters

amount	The amount of charge to add.

Discharge()

```
void DamageChargeComponent::Discharge (
             int amount )  [inline]
```

Removes a specified amount from the current charge, preventing negative values.

Parameters

amount	The amount of charge to remove.

GetPercentage()

```
float DamageChargeComponent::GetPercentage ( ) const  [inline]
```

Calculates the current charge as a percentage of the total capacity.

Returns

> The percentage of charge remaining (0.0f if totalCharge is 0).

IsEmpty()

```
bool DamageChargeComponent::IsEmpty ( ) const  [inline]
```

Checks if the charge is depleted.

Returns

> True if current charge is 0, false otherwise.

IsFullyCharged()

```
bool DamageChargeComponent::IsFullyCharged ( ) const  [inline]
```

Checks if the charge is at full capacity.

Returns

> True if current charge equals total charge, false otherwise.

The documentation for this struct was generated from the following file:

- Components/DamageChargeComponent.hpp

1.13 DamageSystem Class Reference

System that handles damage application when collisions occur.

```
#include <DamageSystem.hpp>
```

Public Member Functions

- DamageSystem ()

  Construct a new Damage System object.
- void SubscribeToCollisionEvent (std::unique_ptr< EventManager > &eventManager)

  Subscribe to collision events.
- void OnCollision (CollisionEvent &event)

  Handle collision events and apply damage.

Public Member Functions inherited from System

- **System** ()=default

  Default constructor for the system.
- ∼**System** ()=default

  Default destructor for the system.
- void AddEntityToSystem (Entity entity)

  Adds an entity to the system.
- void RemoveEntityFromSystem (Entity entity)

  Removes an entity from the system.
- std::vector< Entity > GetSystemEntities () const

  Gets the list of entities managed by this system.
- const Signature & GetComponentSignature () const

  Gets the component signature of the system.
- template<typename TComponent >
  void RequireComponent ()

  Specifies a required component for the system.

1.13.1 Detailed Description

System that handles damage application when collisions occur.

Listens for collision events and applies damage when:

- A projectile hits an entity with health
- The damaged entity is a player Manages player death state when health reaches zero.

1.13.2 Constructor & Destructor Documentation

DamageSystem()

```
DamageSystem::DamageSystem ( )  [inline]
```

Construct a new Damage System object.

Requires entities to have CircleColliderComponent

1.13.3 Member Function Documentation

OnCollision()

```
void DamageSystem::OnCollision (
            CollisionEvent & event )  [inline]
```

Handle collision events and apply damage.

Parameters

event	The collision event containing involved entities

SubscribeToCollisionEvent()

```
void DamageSystem::SubscribeToCollisionEvent (
            std::unique_ptr< EventManager > & eventManager )  [inline]
```

Subscribe to collision events.

Parameters

eventManager	Reference to the event manager

The documentation for this class was generated from the following file:

- Systems/DamageSystem.hpp

1.14 DrawableComponent Struct Reference

Component for managing drawable elements with colored points and timestamps in the ECS.

```
#include <DrawableComponent.hpp>
```

Public Member Functions

- **DrawableComponent** ()

  Default constructor initializing with white color and three empty point lists.
- DrawableComponent (SDL_Color col)

  Constructor initializing with a specified color and three empty point lists.

Public Attributes

- SDL_Color **color**

  Color used for rendering the drawable element.
- std::vector< std::vector< std::pair< glm::vec2, std::chrono::steady_clock::time_point > > > **colorPoints**

  Vector of point lists for different colors (red, blue, green), each with position and timestamp.

1.14.1 Detailed Description

Component for managing drawable elements with colored points and timestamps in the ECS.

1.14.2 Constructor & Destructor Documentation

DrawableComponent()

```
DrawableComponent::DrawableComponent (
            SDL_Color col )  [inline]
```

Constructor initializing with a specified color and three empty point lists.

Parameters

col	The SDL_Color to set for the component.

The documentation for this struct was generated from the following file:

- Components/DrawableComponent.hpp

1.15 DrawingEffectSystem Class Reference

System that handles drawing-based effects on game entities.

`#include <DrawingEffectSystem.hpp>`

Public Member Functions

- DrawingEffectSystem ()

  Construct a new Drawing Effect System object.
- void Update ()

  Update drawing effects for all drawing entities.

Public Member Functions inherited from System

- **System** ()=default

  Default constructor for the system.
- ∼**System** ()=default

  Default destructor for the system.
- void AddEntityToSystem (Entity entity)

  Adds an entity to the system.
- void RemoveEntityFromSystem (Entity entity)

  Removes an entity from the system.
- std::vector< Entity > GetSystemEntities () const

  Gets the list of entities managed by this system.
- const Signature & GetComponentSignature () const

  Gets the component signature of the system.
- template<typename TComponent >
  void RequireComponent ()

  Specifies a required component for the system.

1.15.1 Detailed Description

System that handles drawing-based effects on game entities.

Processes drawing traces to apply various effects:

- Red traces damage enemies
- Blue traces boost player speed
- Green traces slow enemies Uses collision detection between drawing points and entities.

1.15.2 Constructor & Destructor Documentation

DrawingEffectSystem()

```
DrawingEffectSystem::DrawingEffectSystem ( )  [inline]
```

Construct a new Drawing Effect System object.

Requires entities to have DrawableComponent

1.15.3 Member Function Documentation

Update()

```
void DrawingEffectSystem::Update ( )  [inline]
```

Update drawing effects for all drawing entities.

Processes each color channel separately to apply different effects

The documentation for this class was generated from the following file:

- Systems/DrawingEffectSystem.hpp

1.16 DrawSystem Class Reference

System responsible for rendering drawable components.

```
#include <DrawSystem.hpp>
```

Public Member Functions

- DrawSystem ()

  Construct a new Draw System object.
- void Update (SDL_Renderer ∗renderer)

  Update and render all drawable components.

Public Member Functions inherited from System

- **System** ()=default

  Default constructor for the system.
- ∼**System** ()=default

  Default destructor for the system.
- void AddEntityToSystem (Entity entity)

  Adds an entity to the system.
- void RemoveEntityFromSystem (Entity entity)

  Removes an entity from the system.
- std::vector< Entity > GetSystemEntities () const

  Gets the list of entities managed by this system.
- const Signature & GetComponentSignature () const

  Gets the component signature of the system.
- template<typename TComponent >
  void RequireComponent ()

  Specifies a required component for the system.

1.16.1 Detailed Description

System responsible for rendering drawable components.

Handles the drawing of colored points and manages their lifetime.

1.16.2 Constructor & Destructor Documentation

DrawSystem()

```
DrawSystem::DrawSystem ( )  [inline]
```

Construct a new Draw System object.

Requires entities to have a DrawableComponent

1.16.3 Member Function Documentation

Update()

```
void DrawSystem::Update (
             SDL_Renderer * renderer )  [inline]
```

Update and render all drawable components.

Parameters

renderer	SDL renderer used for drawing

The documentation for this class was generated from the following file:

- Systems/DrawSystem.hpp

1.17 EffectReceiverComponent Struct Reference

Component for managing status effects on entities in the ECS.

```
#include <EffectReceiverComponent.hpp>
```

Public Member Functions

- **EffectReceiverComponent** ()=default

  Default constructor for EffectReceiverComponent.

Public Attributes

- bool **takingDamage** = false

  Indicates if the entity is currently taking damage (used for enemies, red effect, index 0).
- bool **slowed** = false

  Indicates if the entity is slowed (used for enemies, green effect, index 2).
- bool **speedBoosted** = false

  Indicates if the entity has a speed boost (used for players, blue effect, index 1).

1.17.1 Detailed Description

Component for managing status effects on entities in the ECS.

The documentation for this struct was generated from the following file:

- Components/EffectReceiverComponent.hpp

1.18 EnemyComponent Struct Reference

Component for managing enemy-related data in the ECS.

```
#include <EnemyComponent.hpp>
```

Public Member Functions

- EnemyComponent (int amountToSpawn=0, int spawnerId=0, int totalAmount=0, int points=0)

  Constructs an EnemyComponent with specified parameters.

Public Attributes

- int **amountToSpawn**

  Number of enemies to spawn.
- int **spawnerId**

  Identifier of the spawner associated with the enemy.
- int **totalAmount**

  Total number of enemies for this component.
- int **points**

  Points awarded for defeating the enemy.

1.18.1 Detailed Description

Component for managing enemy-related data in the ECS.

1.18.2 Constructor & Destructor Documentation

EnemyComponent()

```
EnemyComponent::EnemyComponent (
             int amountToSpawn = 0,
             int spawnerId = 0,
             int totalAmount = 0,
             int points = 0 )  [inline]
```

Constructs an EnemyComponent with specified parameters.

Parameters

amountToSpawn	Number of enemies to spawn (default: 0).
spawnerId	Identifier of the spawner (default: 0).
totalAmount	Total number of enemies (default: 0).
points	Points awarded for defeating the enemy (default: 0).

The documentation for this struct was generated from the following file:

- Components/EnemyComponent.hpp

1.19 EnemySystem Class Reference

System that manages enemy spawning and behavior.

```
#include <EnemySystem.hpp>
```

Public Member Functions

- EnemySystem ()

  Construct a new Enemy System object.
- void Update (std::unique_ptr< Registry > &registry)

  Update enemy spawners and spawn new enemies.
- void CreateEnemyProjectile (std::unique_ptr< Registry > &registry, glm::vec2 velocity, glm::vec2 position, double rotation, int damage)

  Create an enemy projectile.

Public Member Functions inherited from System

- **System** ()=default

  Default constructor for the system.
- ∼**System** ()=default

  Default destructor for the system.
- void AddEntityToSystem (Entity entity)

  Adds an entity to the system.
- void RemoveEntityFromSystem (Entity entity)

  Removes an entity from the system.
- std::vector< Entity > GetSystemEntities () const

  Gets the list of entities managed by this system.
- const Signature & GetComponentSignature () const

  Gets the component signature of the system.
- template<typename TComponent >
  void RequireComponent ()

  Specifies a required component for the system.

1.19.1 Detailed Description

System that manages enemy spawning and behavior.

Handles enemy spawning from spawner entities, projectile creation, and cloning of enemy templates with random positions.

1.19.2 Constructor & Destructor Documentation

EnemySystem()

```
EnemySystem::EnemySystem ( )  [inline]
```

Construct a new Enemy System object.

Requires entities to have EnemyComponent

1.19.3 Member Function Documentation

CreateEnemyProjectile()

```
void EnemySystem::CreateEnemyProjectile (
             std::unique_ptr< Registry > & registry,
             glm::vec2 velocity,
             glm::vec2 position,
             double rotation,
             int damage )  [inline]
```

Create an enemy projectile.

Parameters

registry	Reference to the ECS registry
velocity	Projectile velocity vector
position	Starting position
rotation	Initial rotation
damage	Damage value of the projectile

Update()

```
void EnemySystem::Update (
            std::unique_ptr< Registry > & registry )  [inline]
```

Update enemy spawners and spawn new enemies.

Parameters

registry	Reference to the ECS registry

The documentation for this class was generated from the following file:

- Systems/EnemySystem.hpp

1.20 Entity Class Reference

Represents an entity in the ECS, identified by a unique ID.

```
#include <ECS.hpp>
```

Public Member Functions

- Entity (int id)

  Constructs an entity with a given ID.
- int GetId () const

  Gets the entity's ID.
- void **Kill** ()

  Marks the entity for deletion.
- bool operator== (const Entity &other) const

  Equality operator for comparing entities.
- bool operator!= (const Entity &other) const

  Inequality operator for comparing entities.
- bool operator> (const Entity &other) const

  Greater-than operator for comparing entities.
- bool operator< (const Entity &other) const

  Less-than operator for comparing entities.
- template<typename TComponent , typename... TArgs>
  void AddComponent (TArgs &&... args)

Adds a component to the entity.

- template<typename TComponent >
  void RemoveComponent ()

  Removes a component from the entity.
- template<typename TComponent >
  bool HasComponent () const

  Checks if the entity has a specific component.
- template<typename TComponent >
  TComponent & GetComponent () const

  Gets a reference to a specific component of the entity.

Public Attributes

- class Registry ∗ **registry**

  Pointer to the ECS registry managing this entity.

1.20.1 Detailed Description

Represents an entity in the ECS, identified by a unique ID.

1.20.2 Constructor & Destructor Documentation

Entity()

```
Entity::Entity (
             int id )   [inline]
```

Constructs an entity with a given ID.

Parameters

id	The unique identifier for the entity.

1.20.3 Member Function Documentation

AddComponent()

```
template<typename TComponent , typename...  TArgs>
void Entity::AddComponent (
             TArgs &&...  args )
```

Adds a component to the entity.

Adds a component to an entity through the entity interface.

Template Parameters

TComponent	The type of component to add.
TArgs	Parameter pack for component constructor arguments.

Parameters

args	Arguments to forward to the component's constructor.

GetComponent()

```
template<typename TComponent >
TComponent & Entity::GetComponent ( ) const
```

Gets a reference to a specific component of the entity.

Gets a reference to a specific component of an entity through the entity interface.

Template Parameters

TComponent	The type of component to retrieve.

Returns

Reference to the component.

GetId()

```
int Entity::GetId ( ) const
```

Gets the entity's ID.

Returns

The entity's unique ID.

HasComponent()

```
template<typename TComponent >
bool Entity::HasComponent ( ) const
```

Checks if the entity has a specific component.

Checks if an entity has a specific component through the entity interface.

Template Parameters

TComponent	The type of component to check.

Returns

True if the entity has the component, false otherwise.

operator"!=()

```
bool Entity::operator!= (
            const Entity & other ) const  [inline]
```

Inequality operator for comparing entities.

Parameters

other	The other entity to compare with.

Returns

True if the entities have different IDs, false otherwise.

operator<()

```
bool Entity::operator< (
            const Entity & other ) const  [inline]
```

Less-than operator for comparing entities.

Parameters

other	The other entity to compare with.

Returns

True if this entity's ID is less than the other's, false otherwise.

operator==()

```
bool Entity::operator== (
            const Entity & other ) const  [inline]
```

Equality operator for comparing entities.

Parameters

other	The other entity to compare with.

Returns

True if the entities have the same ID, false otherwise.

operator>()

```
bool Entity::operator> (
             const Entity & other ) const  [inline]
```

Greater-than operator for comparing entities.

Parameters

other	The other entity to compare with.

Returns

True if this entity's ID is greater than the other's, false otherwise.

RemoveComponent()

```
template<typename TComponent >
void Entity::RemoveComponent ( )
```

Removes a component from the entity.

Removes a component from an entity through the entity interface.

Template Parameters

TComponent	The type of component to remove.

The documentation for this class was generated from the following files:

- ECS/ECS.hpp
- ECS/ECS.cpp

1.21 Event Class Reference

Base class for event objects.

```
#include <Event.hpp>
```

Public Member Functions

- **Event** ()=default

  Construct a new Event object (default)
- virtual ∼**Event** ()=default

  Destroy the Event object (virtual for proper inheritance)

1.21.1 Detailed Description

Base class for event objects.

Provides the foundation for event handling with default constructor and virtual destructor for proper polymorphic behavior. Can be extended to implement specific event types.

The documentation for this class was generated from the following file:

- EventManager/Event.hpp

1.22 EventCallback< TOwner, TEvent > Class Template Reference

Templated event callback implementation.

```
#include <EventManager.hpp>
```

Public Member Functions

- EventCallback (TOwner ∗owner, CallbackFunction callback)

  Construct a new Event Callback object.

Public Member Functions inherited from IEventCallback

- void Excute (Event &event)

  Executes the callback with the given event.

1.22.1 Detailed Description

template<typename TOwner, typename TEvent>
class EventCallback< TOwner, TEvent >

Templated event callback implementation.

Template Parameters

TOwner	Type of the owner class
TEvent	Type of the event to handle

1.22.2 Constructor & Destructor Documentation

EventCallback()

```
template<typename TOwner , typename TEvent >
EventCallback< TOwner, TEvent >::EventCallback (
             TOwner * owner,
             CallbackFunction callback )  [inline]
```

Construct a new Event Callback object.

Parameters

owner	Pointer to the owner instance
callback	Pointer to member function to call

The documentation for this class was generated from the following file:

- EventManager/EventManager.hpp

1.23 EventManager Class Reference

Manages event subscriptions and dispatching.

```
#include <EventManager.hpp>
```

Public Member Functions

- **EventManager** ()

  Construct a new Event Manager object.
- ∼**EventManager** ()

  Destroy the Event Manager object.
- void **Restart** ()

  Clears all event subscriptions.
- template<typename TOwner , typename TEvent >
  void SubscribeToEvent (TOwner ∗owner, void(TOwner::∗callback)(TEvent &))

  Subscribe to an event type.
- template<typename TEvent , typename... TArgs>
  void EmitEvent (TArgs &&... args)

  Emit an event to all subscribers.

1.23.1 Detailed Description

Manages event subscriptions and dispatching.

Implements a publish-subscribe pattern with type-safe event handling.

1.23.2 Member Function Documentation

EmitEvent()

```
template<typename TEvent , typename...  TArgs>
void EventManager::EmitEvent (
             TArgs &&...  args )  [inline]
```

Emit an event to all subscribers.

Template Parameters

TEvent	Type of the event to emit
TArgs	Argument types for event construction

Parameters

args	Arguments to forward to event constructor

SubscribeToEvent()

```
template<typename TOwner , typename TEvent >
void EventManager::SubscribeToEvent (
            TOwner * owner,
            void(TOwner::*)(TEvent &) callback )  [inline]
```

Subscribe to an event type.

Template Parameters

TOwner	Type of the owner class
TEvent	Type of the event to subscribe to

Parameters

owner	Pointer to the owner instance
callback	Member function to call when event occurs

The documentation for this class was generated from the following file:

- EventManager/EventManager.hpp

1.24 Game Class Reference

Main game class that manages the game loop, rendering, and resources.

```
#include <Game.hpp>
```

Public Member Functions

- void **init** ()

  Initializes the game, setting up SDL and other resources.
- void **run** ()

  Runs the main game loop.
- void **destroy** ()

  Cleans up and destroys game resources.

Static Public Member Functions

- static Game & GetInstance ()

  Gets the singleton instance of the Game class.

Public Attributes

- SDL_Renderer ∗ **renderer** = nullptr

  SDL renderer for drawing graphics.
- size_t **windowWidth** = 0

  Width of the game window.
- size_t **windowHeight** = 0

  Height of the game window.
- std::unique_ptr< ControllerManager > **controllerManager**

  Unique pointer to the controller manager for handling input.
- std::unique_ptr< AssetManager > **assetManager**

  Unique pointer to the asset manager for handling game assets.
- std::unique_ptr< EventManager > **eventManager**

  Unique pointer to the event manager for handling game events.
- std::unique_ptr< Registry > **registry**

  Unique pointer to the ECS registry for managing entities and components.
- std::unique_ptr< SceneManager > **sceneManager**

  Unique pointer to the scene manager for handling game scenes.
- sol::state **lua**

  Lua state for scripting support.
- int **enemiesLeftToSpawn** = 0

  Number of enemies left to spawn in the current level.
- int **enemiesLeft** = 0

  Number of enemies currently active in the game.
- int **totalPoints** = 0

  Total points accumulated by the player.
- int **totalPointsPrev** = 0

  Previous total points for tracking changes.
- bool **finDelNivel** = false

  Indicates whether the current level has ended.
- bool **win** = false

  Indicates whether the player has won the game.
- bool **isPaused** = false

  Indicates whether the game is paused.
- int **drawIndex** = -1

  Index used for drawing order.
- int **currentLevel** = 0

  Current level of the game.

1.24.1 Detailed Description

Main game class that manages the game loop, rendering, and resources.

1.24.2 Member Function Documentation

GetInstance()

```
Game & Game::GetInstance ( )  [static]
```

Gets the singleton instance of the Game class.

Returns

Reference to the Game instance.

The documentation for this class was generated from the following files:

- Game/Game.hpp
- Game/Game.cpp

1.25 HealthComponent Struct Reference

Component for managing health-related data for entities in the ECS.

```
#include <HealthComponent.hpp>
```

Public Member Functions

- HealthComponent (int health=100, int maxHealth=100, bool isPlayer=false, int damage=0, float attackTimeout=0.0f)

  Constructs a HealthComponent with specified parameters.

Public Attributes

- int **health**

  Current health of the entity.
- int **maxHealth**

  Maximum health capacity of the entity.
- bool **isPlayer**

  Indicates whether the entity is the player.
- int **damage**

  Damage dealt by the entity's attacks.
- float **attackTimeout**

  Time interval (in seconds) between consecutive attacks.
- std::chrono::steady_clock::time_point **attackTimeoutDuration**

  Timestamp of the last attack performed by the entity.
- std::chrono::steady_clock::time_point **lastDamageReceived**

  Timestamp of the last time the entity received damage.

1.25.1 Detailed Description

Component for managing health-related data for entities in the ECS.

1.25.2 Constructor & Destructor Documentation

HealthComponent()

```
HealthComponent::HealthComponent (
            int health = 100,
            int maxHealth = 100,
            bool isPlayer = false,
            int damage = 0,
            float attackTimeout = 0.0f )  [inline]
```

Constructs a HealthComponent with specified parameters.

Parameters

health	Current health of the entity (default: 100).
maxHealth	Maximum health capacity (default: 100).
isPlayer	Whether the entity is the player (default: false).
damage	Damage dealt by the entity's attacks (default: 0).
attackTimeout	Time interval between attacks in seconds (default: 0.0f).

The documentation for this struct was generated from the following file:

- Components/HealthComponent.hpp

1.26 HealthSystem Class Reference

System that manages health and damage effects for entities.

```
#include <HealthSystem.hpp>
```

Public Member Functions

- HealthSystem ()

  Construct a new Health System object.
- void Update ()

  Update health status and process effects.
- void ReduceHP (Entity entity, int damage, Entity attacker)

  Apply direct damage from an attacker.
- void SetHealth (Entity entity, int value)

  Set entity's health to specific value.
- void Heal (Entity entity, int amount)

  Heal entity by specified amount.

Public Member Functions inherited from System

- **System** ()=default

  Default constructor for the system.
- ∼**System** ()=default

  Default destructor for the system.
- void AddEntityToSystem (Entity entity)

  Adds an entity to the system.
- void RemoveEntityFromSystem (Entity entity)

  Removes an entity from the system.
- std::vector< Entity > GetSystemEntities () const

  Gets the list of entities managed by this system.
- const Signature & GetComponentSignature () const

  Gets the component signature of the system.
- template<typename TComponent >
  void RequireComponent ()

  Specifies a required component for the system.

1.26.1 Detailed Description

System that manages health and damage effects for entities.

Handles health management, damage application, healing, and status effects including damage zones, speed boosts, and slowdown effects.

1.26.2 Constructor & Destructor Documentation

HealthSystem()

```
HealthSystem::HealthSystem ( )  [inline]
```

Construct a new Health System object.

Requires entities to have HealthComponent

1.26.3 Member Function Documentation

Heal()

```
void HealthSystem::Heal (
            Entity entity,
            int amount )  [inline]
```

Heal entity by specified amount.

Parameters

entity	Target entity
amount	Healing amount

ReduceHP()

```
void HealthSystem::ReduceHP (
            Entity entity,
            int damage,
            Entity attacker )  [inline]
```

Apply direct damage from an attacker.

Parameters

entity	Target entity
damage	Damage amount
attacker	Attacking entity

SetHealth()

```
void HealthSystem::SetHealth (
            Entity entity,
            int value )  [inline]
```

Set entity's health to specific value.

Parameters

entity	Target entity
value	New health value

Update()

```
void HealthSystem::Update ( )  [inline]
```

Update health status and process effects.

Processes zone damage, speed effects, and other status effects for all entities with health components.

The documentation for this class was generated from the following file:

- Systems/HealthSystem.hpp

1.27 IComponent Class Reference

Base interface for components in the ECS.

```
#include <ECS.hpp>
```

Static Protected Attributes

- static int **nextId** = 0

  Static counter for assigning unique component IDs.

1.27.1 Detailed Description

Base interface for components in the ECS.

The documentation for this class was generated from the following files:

- ECS/ECS.hpp
- ECS/ECS.cpp

1.28 IdentifierComponent Struct Reference

Component for assigning an identifier and name to an entity in the ECS.

```
#include <IdentifierComponent.hpp>
```

Public Member Functions

- **IdentifierComponent** ()=default

  Default constructor for IdentifierComponent.
- IdentifierComponent (int id, const std::string &name)

  Constructs an IdentifierComponent with specified ID and name.

Public Attributes

- int **id**

  Unique identifier for the entity.
- std::string **name**

  Name associated with the entity.

1.28.1 Detailed Description

Component for assigning an identifier and name to an entity in the ECS.

1.28.2 Constructor & Destructor Documentation

IdentifierComponent()

```
IdentifierComponent::IdentifierComponent (
            int id,
            const std::string & name )  [inline]
```

Constructs an IdentifierComponent with specified ID and name.

Parameters

id	The unique identifier for the entity.
name	The name associated with the entity.

The documentation for this struct was generated from the following file:

- Components/IdentifierComponent.hpp

1.29 IEventCallback Class Reference

Interface for event callback functionality.

```
#include <EventManager.hpp>
```

Public Member Functions

- void Excute (Event &event)

  Executes the callback with the given event.

1.29.1 Detailed Description

Interface for event callback functionality.

Base class for implementing event callbacks with type erasure.

1.29.2 Member Function Documentation

Excute()

```
void IEventCallback::Excute (
            Event & event )  [inline]
```

Executes the callback with the given event.

Parameters

event	The event to process

The documentation for this class was generated from the following file:

- EventManager/EventManager.hpp

1.30 IPool Class Reference

Interface for component pools in the ECS framework.

```
#include <Pool.hpp>
```

1.30.1 Detailed Description

Interface for component pools in the ECS framework.

Provides a base interface for all component pool implementations.

The documentation for this class was generated from the following file:

- Utils/Pool.hpp

1.31 MovementSystem Class Reference

System that handles entity movement and screen boundary constraints.

```
#include <MovementSystem.hpp>
```

Public Member Functions

- MovementSystem ()

  Construct a new Movement System object.
- void Update (double dt)

  Update entity positions and handle boundary collisions.

Public Member Functions inherited from System

- **System** ()=default

  Default constructor for the system.
- ∼**System** ()=default

  Default destructor for the system.
- void AddEntityToSystem (Entity entity)

  Adds an entity to the system.
- void RemoveEntityFromSystem (Entity entity)

  Removes an entity from the system.
- std::vector< Entity > GetSystemEntities () const

  Gets the list of entities managed by this system.
- const Signature & GetComponentSignature () const

  Gets the component signature of the system.
- template<typename TComponent >
  void RequireComponent ()

  Specifies a required component for the system.

1.31.1 Detailed Description

System that handles entity movement and screen boundary constraints.

Updates entity positions based on velocity and ensures they stay within game boundaries. Also manages sprite flipping based on movement direction.

1.31.2 Constructor & Destructor Documentation

MovementSystem()

```
MovementSystem::MovementSystem ( )  [inline]
```

Construct a new Movement System object.

Requires entities to have:

- RigidBodyComponent
- TransformComponent
- SpriteComponent

1.31.3 Member Function Documentation

Update()

```
void MovementSystem::Update (
            double dt )  [inline]
```

Update entity positions and handle boundary collisions.

Parameters

dt	Delta time since last frame (in seconds)

The documentation for this class was generated from the following file:

- Systems/MovementSystem.hpp

1.32 Pool< TComponent > Class Template Reference

Templated component pool implementation.

```
#include <Pool.hpp>
```

Public Member Functions

- Pool (int size=1000)

  Constructs a Pool with initial size.
- bool IsEmpty () const

  Checks if the pool is empty.
- int GetSize () const

  Gets the current size of the pool.
- void Resize (int n)

Resizes the pool.

- void **Clear** ()

  Clears all elements from the pool.

- void Add (TComponent object)

  Adds a component to the end of the pool.

- void Set (unsigned int index, TComponent object)

  Sets a component at a specific index.

- TComponent & Get (unsigned int index)

  Gets a reference to the component at specified index.

- TComponent & operator[ ] (unsigned int index)

  Array access operator.

1.32.1 Detailed Description

template<typename TComponent>
class Pool< TComponent >

Templated component pool implementation.

Template Parameters

TComponent	The type of component to be stored in the pool.

Implements a resizable container for storing components of a specific type. Provides basic operations for component management in an ECS architecture.

1.32.2 Constructor & Destructor Documentation

Pool()

```
template<typename TComponent >
Pool< TComponent >::Pool (
             int size = 1000 )  [inline]
```

Constructs a Pool with initial size.

Parameters

size	Initial capacity of the pool (default: 1000).

1.32.3 Member Function Documentation

Add()

```
template<typename TComponent >
void Pool< TComponent >::Add (
             TComponent object )  [inline]
```

Adds a component to the end of the pool.

Parameters

object	Component to be added.

Get()

```
template<typename TComponent >
TComponent & Pool< TComponent >::Get (
              unsigned int index )  [inline]
```

Gets a reference to the component at specified index.

Parameters

index	Position of the component to retrieve.

Returns

Reference to the requested component.

GetSize()

```
template<typename TComponent >
int Pool< TComponent >::GetSize ( ) const  [inline]
```

Gets the current size of the pool.

Returns

Number of elements in the pool.

IsEmpty()

```
template<typename TComponent >
bool Pool< TComponent >::IsEmpty ( ) const  [inline]
```

Checks if the pool is empty.

Returns

True if the pool contains no elements, false otherwise.

operator[]()

```
template<typename TComponent >
TComponent & Pool< TComponent >::operator[] (
              unsigned int index )  [inline]
```

Array access operator.

Parameters

index	Position of the component to access.

Returns

Reference to the component at specified index.

Resize()

```
template<typename TComponent >
void Pool< TComponent >::Resize (
             int n )  [inline]
```

Resizes the pool.

Parameters

n	New size of the pool.

Set()

```
template<typename TComponent >
void Pool< TComponent >::Set (
             unsigned int index,
             TComponent object )  [inline]
```

Sets a component at a specific index.

Parameters

index	Position to set the component.
object	Component to be stored.

The documentation for this class was generated from the following file:

- Utils/Pool.hpp

1.33 ProjectileComponent Struct Reference

Component for managing projectile properties in the ECS.

```
#include <ProjectileComponent.hpp>
```

Public Member Functions

- ProjectileComponent (glm::vec2 position=glm::vec2(0.0, 0.0), glm::vec2 scale=glm::vec2(1.0, 1.0), double rotation=0.0, const glm::vec2 &velocity=glm::vec2(0.0f, 0.0f))

  Constructs a ProjectileComponent with specified parameters.

Public Attributes

- glm::vec2 **velocity**

  Velocity vector of the projectile.
- glm::vec2 **position**

  Position vector of the projectile.
- glm::vec2 **scale**

  Scale vector of the projectile.
- double **rotation**

  Rotation angle of the projectile in radians.
- bool **hasHit** = false

  Indicates whether the projectile has hit a target.
- int **damage**

  Damage dealt by the projectile upon impact.

1.33.1 Detailed Description

Component for managing projectile properties in the ECS.

1.33.2 Constructor & Destructor Documentation

ProjectileComponent()

```
ProjectileComponent::ProjectileComponent (
            glm::vec2 position = glm::vec2(0.0, 0.0),
            glm::vec2 scale = glm::vec2(1.0, 1.0),
            double rotation = 0.0,
            const glm::vec2 & velocity = glm::vec2(0.0f, 0.0f) )  [inline]
```

Constructs a ProjectileComponent with specified parameters.

Parameters

position	Initial position of the projectile (default: (0.0, 0.0)).
scale	Scale of the projectile (default: (1.0, 1.0)).
rotation	Initial rotation in radians (default: 0.0).
velocity	Initial velocity vector (default: (0.0, 0.0)).

The documentation for this struct was generated from the following file:

- Components/ProjectileComponent.hpp

1.34 Registry Class Reference

Manages entities, components, and systems in the ECS.

```
#include <ECS.hpp>
```

Public Member Functions

- **Registry** ()

  Constructs a new registry.
- ∼**Registry** ()

  Destroys the registry and its resources.
- void **Update** ()

  Updates the registry, processing pending entity additions and deletions.
- Entity CreateEntity ()

  Creates a new entity in the registry.
- void KillEntity (Entity entity)

  Marks an entity for deletion.
- template<typename TComponent , typename... TArgs>
  void AddComponent (Entity entity, TArgs &&... args)

  Adds a component to an entity.
- template<typename TComponent >
  void RemoveComponent (Entity entity)

  Removes a component from an entity.
- template<typename TComponent >
  bool HasComponent (Entity entity) const

  Checks if an entity has a specific component.
- template<typename TComponent >
  TComponent & GetComponent (Entity entity) const

  Gets a reference to a specific component of an entity.
- template<typename TSystem , typename... TArgs>
  void AddSystem (TArgs &&... args)

  Adds a system to the registry.
- template<typename TSystem >
  void RemoveSystem ()

  Removes a system from the registry.
- template<typename TSystem >
  bool HasSystem () const

  Checks if a system exists in the registry.
- template<typename TSystem >
  TSystem & GetSystem () const

  Gets a reference to a specific system.
- void AddEntityToSystems (Entity entity)

  Adds an entity to the relevant systems based on its components.
- void RemoveEntityFromSystems (Entity entity)

  Removes an entity from all systems.
- template<typename T >
  std::vector< Entity > GetEntitiesFromSystem ()

  Gets all entities associated with a specific system.
- void **ClearAllEntities** ()

  Clears all entities from the registry.

1.34.1 Detailed Description

Manages entities, components, and systems in the ECS.

1.34.2 Member Function Documentation

AddComponent()

```
template<typename TComponent , typename...  TArgs>
void Registry::AddComponent (
             Entity entity,
             TArgs &&...  args )
```

Adds a component to an entity.

Adds a component to an entity in the registry.

Template Parameters

TComponent	The type of component to add.
TArgs	Parameter pack for component constructor arguments.

Parameters

entity	The entity to add the component to.
args	Arguments to forward to the component's constructor.

AddEntityToSystems()

```
void Registry::AddEntityToSystems (
             Entity entity )
```

Adds an entity to the relevant systems based on its components.

Parameters

entity	The entity to add.

AddSystem()

```
template<typename TSystem , typename...  TArgs>
void Registry::AddSystem (
             TArgs &&...  args )
```

Adds a system to the registry.

Template Parameters

TSystem	The type of system to add.
TArgs	Parameter pack for system constructor arguments.

Parameters

args	Arguments to forward to the system's constructor.

CreateEntity()

```
Entity Registry::CreateEntity ( )
```

Creates a new entity in the registry.

Returns

The created entity.

GetComponent()

```
template<typename TComponent >
TComponent & Registry::GetComponent (
             Entity entity ) const
```

Gets a reference to a specific component of an entity.

Gets a reference to a specific component of an entity in the registry.

Template Parameters

TComponent	The type of component to retrieve.

Parameters

entity	The entity to retrieve the component from.

Returns

Reference to the component.

GetEntitiesFromSystem()

```
template<typename T >
std::vector< Entity > Registry::GetEntitiesFromSystem ( )
```

Gets all entities associated with a specific system.

Template Parameters

T	The type of system to query.

Returns

A vector of entities in the system.

GetSystem()

```
template<typename TSystem >
TSystem & Registry::GetSystem ( ) const
```

Gets a reference to a specific system.

Gets a reference to a specific system in the registry.

Template Parameters

TSystem	The type of system to retrieve.

Returns

Reference to the system.

HasComponent()

```
template<typename TComponent >
bool Registry::HasComponent (
             Entity entity ) const
```

Checks if an entity has a specific component.

Checks if an entity has a specific component in the registry.

Template Parameters

TComponent	The type of component to check.

Parameters

entity	The entity to check.

Returns

True if the entity has the component, false otherwise.

HasSystem()

```
template<typename TSystem >
bool Registry::HasSystem ( ) const
```

Checks if a system exists in the registry.

Template Parameters

TSystem	The type of system to check.

Returns

True if the system exists, false otherwise.

KillEntity()

```
void Registry::KillEntity (
             Entity entity )
```

Marks an entity for deletion.

Parameters

entity	The entity to delete.

RemoveComponent()

```
template<typename TComponent >
void Registry::RemoveComponent (
             Entity entity )
```

Removes a component from an entity.

Removes a component from an entity in the registry.

Template Parameters

TComponent	The type of component to remove.

Parameters

entity	The entity to remove the component from.

RemoveEntityFromSystems()

```
void Registry::RemoveEntityFromSystems (
            Entity entity )
```

Removes an entity from all systems.

Parameters

entity	The entity to remove.

RemoveSystem()

```
template<typename TSystem >
void Registry::RemoveSystem ( )
```

Removes a system from the registry.

Template Parameters

TSystem	The type of system to remove.

The documentation for this class was generated from the following files:

- ECS/ECS.hpp
- ECS/ECS.cpp

1.35 RenderSystem Class Reference

System responsible for rendering entities to the screen.

`#include <RenderSystem.hpp>`

Public Member Functions

- RenderSystem ()

  Construct a new Render System object.
- void Update (SDL_Renderer ∗renderer, std::unique_ptr< AssetManager > &AssetManager)

  Render all visible entities (excluding background if it's first)
- void UpdateBackground (SDL_Renderer ∗renderer, std::unique_ptr< AssetManager > &AssetManager)

  Render only background entities.

Public Member Functions inherited from System

- **System** ()=default

  Default constructor for the system.
- ∼**System** ()=default

  Default destructor for the system.
- void AddEntityToSystem (Entity entity)

  Adds an entity to the system.

- void RemoveEntityFromSystem (Entity entity)

  Removes an entity from the system.
- std::vector< Entity > GetSystemEntities () const

  Gets the list of entities managed by this system.
- const Signature & GetComponentSignature () const

  Gets the component signature of the system.
- template<typename TComponent >
  void RequireComponent ()

  Specifies a required component for the system.

1.35.1 Detailed Description

System responsible for rendering entities to the screen.

Handles the drawing of all visible entities with sprite components, including special handling for background rendering.

1.35.2 Constructor & Destructor Documentation

RenderSystem()

```
RenderSystem::RenderSystem ( )  [inline]
```

Construct a new Render System object.

Requires entities to have both TransformComponent and SpriteComponent

1.35.3 Member Function Documentation

Update()

```
void RenderSystem::Update (
            SDL_Renderer * renderer,
            std::unique_ptr< AssetManager > & AssetManager )  [inline]
```

Render all visible entities (excluding background if it's first)

Parameters

renderer	SDL renderer to draw to
AssetManager	Asset manager for texture access

UpdateBackground()

```
void RenderSystem::UpdateBackground (
            SDL_Renderer * renderer,
            std::unique_ptr< AssetManager > & AssetManager )  [inline]
```

Render only background entities.

Parameters

renderer	SDL renderer to draw to
AssetManager	Asset manager for texture access

The documentation for this class was generated from the following file:

- Systems/RenderSystem.hpp

1.36 RenderTextSystem Class Reference

System responsible for rendering text components.

```
#include <RenderTextSystem.hpp>
```

Public Member Functions

- RenderTextSystem ()

  Construct a new Render Text System object.
- void Update (SDL_Renderer ∗renderer, std::unique_ptr< AssetManager > &assetManager)

  Render all text components.

Public Member Functions inherited from System

- **System** ()=default

  Default constructor for the system.
- ∼**System** ()=default

  Default destructor for the system.
- void AddEntityToSystem (Entity entity)

  Adds an entity to the system.
- void RemoveEntityFromSystem (Entity entity)

  Removes an entity from the system.
- std::vector< Entity > GetSystemEntities () const

  Gets the list of entities managed by this system.
- const Signature & GetComponentSignature () const

  Gets the component signature of the system.
- template<typename TComponent >
  void RequireComponent ()

  Specifies a required component for the system.

1.36.1 Detailed Description

System responsible for rendering text components.

Handles text rendering for various game elements including:

- Health displays
- Charge indicators
- Score display
- General text rendering

1.36.2 Constructor & Destructor Documentation

RenderTextSystem()

```
RenderTextSystem::RenderTextSystem ( ) [inline]
```

Construct a new Render Text System object.

Requires entities to have both TextComponent and TransformComponent

1.36.3 Member Function Documentation

Update()

```
void RenderTextSystem::Update (
            SDL_Renderer * renderer,
            std::unique_ptr< AssetManager > & assetManager ) [inline]
```

Render all text components.

Parameters

renderer	SDL renderer to draw to
assetManager	Asset manager for font access

The documentation for this class was generated from the following file:

- Systems/RenderTextSystem.hpp

1.37 RigidBodyComponent Struct Reference

Component for managing physics-related properties of an entity in the ECS.

```
#include <RigidBodyComponent.hpp>
```

Public Member Functions

- RigidBodyComponent (const glm::vec2 &velocity=glm::vec2(0.0f, 0.0f))

  Constructs a RigidBodyComponent with a specified velocity.

Public Attributes

- glm::vec2 **velocity**

  Velocity vector of the entity.

1.37.1 Detailed Description

Component for managing physics-related properties of an entity in the ECS.

1.37.2 Constructor & Destructor Documentation

RigidBodyComponent()

```
RigidBodyComponent::RigidBodyComponent (
            const glm::vec2 & velocity = glm::vec2(0.0f, 0.0f) )  [inline]
```

Constructs a RigidBodyComponent with a specified velocity.

Parameters

velocity	Initial velocity vector of the entity (default: (0.0, 0.0)).

The documentation for this struct was generated from the following file:

- Components/RigidBodyComponent.hpp

1.38 SceneLoader Class Reference

Handles loading and parsing of game scenes from Lua configuration files.

```
#include <SceneLoader.hpp>
```

Public Member Functions

- **SceneLoader** ()

  Construct a new Scene Loader object.
- ∼**SceneLoader** ()

  Destroy the Scene Loader object.
- void LoadScene (const std::string &scenePath, sol::state &lua, std::unique_ptr< AssetManager > &assetManager, std::unique_ptr< ControllerManager > &controllerManager, std::unique_ptr< Registry > &registry, SDL_Renderer ∗renderer)

  Loads a complete scene from Lua configuration file.

1.38.1 Detailed Description

Handles loading and parsing of game scenes from Lua configuration files.

Manages loading of all scene components including assets, entities, input mappings, and game objects.

1.38.2 Member Function Documentation

LoadScene()

```
void SceneLoader::LoadScene (
            const std::string & scenePath,
            sol::state & lua,
            std::unique_ptr< AssetManager > & assetManager,
            std::unique_ptr< ControllerManager > & controllerManager,
            std::unique_ptr< Registry > & registry,
            SDL_Renderer * renderer )
```

Loads a complete scene from Lua configuration file.

Parameters

scenePath	Path to the scene configuration file
lua	Lua state reference
assetManager	Asset manager for storing loaded assets
controllerManager	Controller manager for input mappings
registry	ECS registry for entity management
renderer	SDL renderer for texture creation

The documentation for this class was generated from the following files:

- SceneManager/SceneLoader.hpp
- SceneManager/SceneLoader.cpp

1.39 SceneManager Class Reference

Manages game scenes and scene transitions.

```
#include <SceneManager.hpp>
```

Public Member Functions

- **SceneManager** ()

  Construct a new Scene Manager object.
- ∼**SceneManager** ()

  Destroy the Scene Manager object.
- void LoadSceneFromScript (const std::string &scenePath, sol::state &lua)

  Load scene configuration from a Lua script.
- void **LoadScene** ()

  Load the currently set next scene.
- std::string GetNextScene () const

  Get the name of the next scene to be loaded.
- void SetNextScene (const std::string &nextScene)

  Set the next scene to be loaded.
- bool IsSceneRunning () const

  Check if a scene is currently running.
- void **StartScene** ()

  Mark the scene as started/running.
- void **StopScene** ()

  Mark the scene as stopped.

1.39.1 Detailed Description

Manages game scenes and scene transitions.

Handles loading, running, and switching between different game scenes. Maintains a collection of available scenes and manages scene lifecycle.

1.39.2 Member Function Documentation

GetNextScene()

```
std::string SceneManager::GetNextScene ( ) const
```

Get the name of the next scene to be loaded.

Returns

std::string Name of the next scene

IsSceneRunning()

```
bool SceneManager::IsSceneRunning ( ) const
```

Check if a scene is currently running.

Returns

true if a scene is active

false if no scene is active

LoadSceneFromScript()

```
void SceneManager::LoadSceneFromScript (
             const std::string & scenePath,
             sol::state & lua )
```

Load scene configuration from a Lua script.

Parameters

scenePath	Path to the scene configuration file
lua	Reference to the Lua state

SetNextScene()

```
void SceneManager::SetNextScene (
             const std::string & nextScene )
```

Set the next scene to be loaded.

Parameters

nextScene	Name of the next scene

The documentation for this class was generated from the following files:

- SceneManager/SceneManager.hpp
- SceneManager/SceneManager.cpp

1.40 ScriptComponent Struct Reference

Component for attaching Lua script functions to an entity in the ECS.

```
#include <ScriptComponent.hpp>
```

Public Member Functions

- ScriptComponent (sol::function update=sol::lua_nil, sol::function onClick=sol::lua_nil)

    Constructs a ScriptComponent with specified Lua functions.

Public Attributes

- sol::function **update**

    Lua function to be called during the update phase.
- sol::function **onClick**

    Lua function to be called when the entity is clicked.

1.40.1 Detailed Description

Component for attaching Lua script functions to an entity in the ECS.

1.40.2 Constructor & Destructor Documentation

ScriptComponent()

```
ScriptComponent::ScriptComponent (
            sol::function update = sol::lua_nil,
            sol::function onClick = sol::lua_nil )  [inline]
```

Constructs a ScriptComponent with specified Lua functions.

Parameters

update	Lua function for the update phase (default: nil).
onClick	Lua function for click events (default: nil).

The documentation for this struct was generated from the following file:

- Components/ScriptComponent.hpp

1.41 ScriptSystem Class Reference

System for handling script components and Lua bindings.

```
#include <ScriptSystem.hpp>
```

Public Member Functions

- **ScriptSystem** ()

  Constructs a ScriptSystem and requires ScriptComponent for entities.
- void CreateLuaBinding (sol::state &lua)

  Creates Lua bindings for game functions and types.
- void Update (sol::state &lua)

  Updates all script components in the system.

Public Member Functions inherited from System

- **System** ()=default

  Default constructor for the system.
- ∼**System** ()=default

  Default destructor for the system.
- void AddEntityToSystem (Entity entity)

  Adds an entity to the system.
- void RemoveEntityFromSystem (Entity entity)

  Removes an entity from the system.
- std::vector< Entity > GetSystemEntities () const

  Gets the list of entities managed by this system.
- const Signature & GetComponentSignature () const

  Gets the component signature of the system.
- template<typename TComponent >
  void RequireComponent ()

  Specifies a required component for the system.

1.41.1 Detailed Description

System for handling script components and Lua bindings.

This system manages entities with ScriptComponents and provides Lua bindings for game functionality.

1.41.2 Member Function Documentation

CreateLuaBinding()

```
void ScriptSystem::CreateLuaBinding (
            sol::state & lua ) [inline]
```

Creates Lua bindings for game functions and types.

Parameters

lua	Reference to the Lua state to bind functions to.

Sets up various game-related functions in the Lua environment including:

- Input handling
- Entity manipulation
- Drawing functions
- Game state management

Update()

```
void ScriptSystem::Update (
             sol::state & lua )  [inline]
```

Updates all script components in the system.

Parameters

lua	Reference to the Lua state for script execution.

Iterates through all entities with ScriptComponents and calls their update function if it exists.

The documentation for this class was generated from the following file:

- Systems/ScriptSystem.hpp

1.42 SlowChargeComponent Struct Reference

Component for managing a charge-based system for slowing effects in the ECS.

```
#include <SlowChargeComponent.hpp>
```

Public Member Functions

- SlowChargeComponent (int total=100, int initialCharge=100)

    Constructs a SlowChargeComponent with specified total and initial charge.
- void **updateChargeDisplay** ()

    Updates the charge display string to reflect current and total charge.
- void **Recharge** ()

    Fully recharges the component to its total capacity.
- void Charge (int amount)

    Adds a specified amount to the current charge, capped at total capacity.
- void Discharge (int amount)

Removes a specified amount from the current charge, preventing negative values.

- float GetPercentage () const

  Calculates the current charge as a percentage of the total capacity.
- bool IsFullyCharged () const

  Checks if the charge is at full capacity.
- bool IsEmpty () const

  Checks if the charge is depleted.

Public Attributes

- int **totalCharge**

  Total charge capacity.
- int **currentCharge**

  Current charge level.
- std::string **chargeDisplay**

  String representation of the charge level in the format "currentCharge/totalCharge".

1.42.1 Detailed Description

Component for managing a charge-based system for slowing effects in the ECS.

1.42.2 Constructor & Destructor Documentation

SlowChargeComponent()

```
SlowChargeComponent::SlowChargeComponent (
             int total = 100,
             int initialCharge = 100 )   [inline]
```

Constructs a SlowChargeComponent with specified total and initial charge.

Parameters

total	The total charge capacity (default: 100).
initialCharge	The initial charge level (default: 100).

1.42.3 Member Function Documentation

Charge()

```
void SlowChargeComponent::Charge (
             int amount )   [inline]
```

Adds a specified amount to the current charge, capped at total capacity.

Parameters

amount	The amount of charge to add.

Discharge()

```
void SlowChargeComponent::Discharge (
             int amount )  [inline]
```

Removes a specified amount from the current charge, preventing negative values.

Parameters

amount	The amount of charge to remove.

GetPercentage()

```
float SlowChargeComponent::GetPercentage ( ) const  [inline]
```

Calculates the current charge as a percentage of the total capacity.

Returns

The percentage of charge remaining (0.0f if totalCharge is 0).

IsEmpty()

```
bool SlowChargeComponent::IsEmpty ( ) const  [inline]
```

Checks if the charge is depleted.

Returns

True if current charge is 0, false otherwise.

IsFullyCharged()

```
bool SlowChargeComponent::IsFullyCharged ( ) const  [inline]
```

Checks if the charge is at full capacity.

Returns

True if current charge equals total charge, false otherwise.

The documentation for this struct was generated from the following file:

- Components/SlowChargeComponent.hpp

1.43 SoundComponent Struct Reference

Component for managing sound properties for an entity in the ECS.

```
#include <SoundComponent.hpp>
```

Public Member Functions

- SoundComponent (const std::string &soundId="none", int volume=128, int loops=-1, bool autoPlay=true, bool active=true)

  Constructs a SoundComponent with specified parameters.

Public Attributes

- std::string **soundId**

  Identifier for the sound asset.
- int **volume**

  Volume level of the sound (0 to 128).
- int **loops**

  Number of times to loop the sound (-1 for infinite).
- bool **isPlaying**

  Indicates whether the sound is currently playing.
- bool **active**

  Indicates whether the sound component is active.
- bool **autoPlay**

  Indicates whether the sound should play automatically.

1.43.1 Detailed Description

Component for managing sound properties for an entity in the ECS.

1.43.2 Constructor & Destructor Documentation

SoundComponent()

```
SoundComponent::SoundComponent (
             const std::string & soundId = "none",
             int volume = 128,
             int loops = -1,
             bool autoPlay = true,
             bool active = true )  [inline]
```

Constructs a SoundComponent with specified parameters.

Parameters

soundId	Identifier for the sound asset (default: "none").
volume	Volume level of the sound (default: 128).
loops	Number of loops for the sound (-1 for infinite, default: -1).
autoPlay	Whether the sound should play automatically (default: true).
active	Whether the sound component is active (default: true).

The documentation for this struct was generated from the following file:

- Components/SoundComponent.hpp

1.44 SoundSystem Class Reference

System for managing sound playback in the ECS framework.

```
#include <SoundSystem.hpp>
```

Public Member Functions

- **SoundSystem** ()

  Constructs a SoundSystem and requires SoundComponent for entities.
- void Update (std::unique_ptr< AssetManager > &assetManager)

  Updates the sound system state and handles auto-play sounds.
- void PlaySound (std::unique_ptr< AssetManager > &assetManager, SoundComponent &sound)

  Plays a sound from the SoundComponent.
- void StopSound (SoundComponent &sound)

  Stops a sound playback.
- void PauseSound (SoundComponent &sound)

  Pauses sound playback.
- void ResumeSound (SoundComponent &sound)

  Resumes paused sound playback.

Public Member Functions inherited from System

- **System** ()=default

  Default constructor for the system.
- ∼**System** ()=default

  Default destructor for the system.
- void AddEntityToSystem (Entity entity)

  Adds an entity to the system.
- void RemoveEntityFromSystem (Entity entity)

  Removes an entity from the system.
- std::vector< Entity > GetSystemEntities () const

  Gets the list of entities managed by this system.
- const Signature & GetComponentSignature () const

  Gets the component signature of the system.
- template<typename TComponent >
  void RequireComponent ()

  Specifies a required component for the system.

1.44.1 Detailed Description

System for managing sound playback in the ECS framework.

Handles playing, stopping, pausing, and resuming sounds for entities with SoundComponents.

1.44.2 Member Function Documentation

PauseSound()

```
void SoundSystem::PauseSound (
            SoundComponent & sound )  [inline]
```

Pauses sound playback.

Parameters

sound	Reference to the SoundComponent to pause.

PlaySound()

```
void SoundSystem::PlaySound (
            std::unique_ptr< AssetManager > & assetManager,
            SoundComponent & sound )  [inline]
```

Plays a sound from the SoundComponent.

Parameters

assetManager	Reference to the asset manager for sound loading.
sound	Reference to the SoundComponent containing sound properties.

Loads and plays the specified sound with configured volume and loop settings. Updates the isPlaying flag based on playback success.

ResumeSound()

```
void SoundSystem::ResumeSound (
            SoundComponent & sound )  [inline]
```

Resumes paused sound playback.

Parameters

sound	Reference to the SoundComponent to resume.

StopSound()

```
void SoundSystem::StopSound (
            SoundComponent & sound )  [inline]
```

Stops a sound playback.

Parameters

sound	Reference to the SoundComponent to stop.

Update()

```
void SoundSystem::Update (
            std::unique_ptr< AssetManager > & assetManager )  [inline]
```

Updates the sound system state and handles auto-play sounds.

Parameters

assetManager	Reference to the asset manager for sound loading.

Iterates through all entities with SoundComponents and automatically plays sounds marked for autoPlay that aren't currently playing.

The documentation for this class was generated from the following file:

- Systems/SoundSystem.hpp

1.45 SprintChargeComponent Struct Reference

Component that manages sprint charge functionality.

```
#include <SprintChargeComponent.hpp>
```

Public Member Functions

- SprintChargeComponent (int total=100, int initialCharge=100)

  Construct a new Sprint Charge Component object.
- void **updateChargeDisplay** ()

  Updates the display string to reflect current charge state.
- void **Recharge** ()

  Fully recharge to maximum capacity.
- void Charge (int amount)

  Add a specific amount of charge.
- void Discharge (int amount)

  Remove a specific amount of charge.
- float GetPercentage () const

  Get current charge percentage.
- bool IsFullyCharged () const

  Check if fully charged.
- bool IsEmpty () const

  Check if completely discharged.

Public Attributes

- int **totalCharge**

  Maximum possible charge value.
- int **currentCharge**

  Current amount of charge available.
- std::string **chargeDisplay**

  String to display "currentCharge/totalCharge".

1.45.1 Detailed Description

Component that manages sprint charge functionality.

Tracks current and total charge amounts and provides utility methods for charge management and display.

1.45.2 Constructor & Destructor Documentation

SprintChargeComponent()

```
SprintChargeComponent::SprintChargeComponent (
             int total = 100,
             int initialCharge = 100 )  [inline]
```

Construct a new Sprint Charge Component object.

Parameters

total	Maximum charge capacity (default: 100)
initialCharge	Starting charge amount (default: 100)

1.45.3 Member Function Documentation

Charge()

```
void SprintChargeComponent::Charge (
             int amount )  [inline]
```

Add a specific amount of charge.

Parameters

amount	Quantity to charge (will not exceed totalCharge)

Discharge()

```
void SprintChargeComponent::Discharge (
             int amount )  [inline]
```

Remove a specific amount of charge.

Parameters

amount	Quantity to discharge (will not go below 0)

GetPercentage()

```
float SprintChargeComponent::GetPercentage ( ) const  [inline]
```

Get current charge percentage.

Returns

float Percentage of current charge relative to total (0.0-100.0)

IsEmpty()

```
bool SprintChargeComponent::IsEmpty ( ) const  [inline]
```

Check if completely discharged.

Returns

true When currentCharge equals 0

false Otherwise

IsFullyCharged()

```
bool SprintChargeComponent::IsFullyCharged ( ) const  [inline]
```

Check if fully charged.

Returns

true When currentCharge equals totalCharge

false Otherwise

The documentation for this struct was generated from the following file:

- Components/SprintChargeComponent.hpp

1.46 SpriteComponent Struct Reference

Component that handles sprite rendering properties.

```
#include <SpriteComponent.hpp>
```

Public Member Functions

- SpriteComponent (const std::string &textureId="none", int width=0, int height=0, int srcRectX=0, int srcRectY=0, bool active=true)

  Construct a new Sprite Component object.

Public Attributes

- SDL_Rect **srcRect**

  Source rectangle for sprite sheet cropping.
- std::string **textureId**

  ID of the texture resource to use.
- int **width**

  Width of the sprite in pixels.
- int **height**

  Height of the sprite in pixels.
- SDL_RendererFlip **flip**

  Current flip state (none, horizontal, vertical)
- bool **active**

  Whether the sprite should be rendered.

1.46.1 Detailed Description

Component that handles sprite rendering properties.

Stores texture information, dimensions, and rendering state for an entity.

1.46.2 Constructor & Destructor Documentation

SpriteComponent()

```
SpriteComponent::SpriteComponent (
             const std::string & textureId = "none",
             int width = 0,
             int height = 0,
             int srcRectX = 0,
             int srcRectY = 0,
             bool active = true )  [inline]
```

Construct a new Sprite Component object.

Parameters

textureId	ID of the texture to use (default: "none")
width	Sprite width in pixels (default: 0)
height	Sprite height in pixels (default: 0)
srcRectX	X position in source texture (default: 0)
srcRectY	Y position in source texture (default: 0)
active	Whether sprite is initially active (default: true)

The documentation for this struct was generated from the following file:

- Components/SpriteComponent.hpp

1.47 System Class Reference

Represents a system in the ECS that operates on entities with specific components.

```
#include <ECS.hpp>
```

Public Member Functions

- **System** ()=default

  Default constructor for the system.
- ∼**System** ()=default

  Default destructor for the system.
- void AddEntityToSystem (Entity entity)

  Adds an entity to the system.
- void RemoveEntityFromSystem (Entity entity)

  Removes an entity from the system.
- std::vector< Entity > GetSystemEntities () const

  Gets the list of entities managed by this system.
- const Signature & GetComponentSignature () const

  Gets the component signature of the system.
- template<typename TComponent >
  void RequireComponent ()

  Specifies a required component for the system.

1.47.1 Detailed Description

Represents a system in the ECS that operates on entities with specific components.

1.47.2 Member Function Documentation

AddEntityToSystem()

```
void System::AddEntityToSystem (
            Entity entity )
```

Adds an entity to the system.

Parameters

entity	The entity to add.

GetComponentSignature()

```
const Signature & System::GetComponentSignature ( ) const
```

Gets the component signature of the system.

Returns

The system's component signature.

GetSystemEntities()

```
std::vector< Entity > System::GetSystemEntities ( ) const
```

Gets the list of entities managed by this system.

Returns

A vector of entities.

RemoveEntityFromSystem()

```
void System::RemoveEntityFromSystem (
            Entity entity )
```

Removes an entity from the system.

Parameters

entity	The entity to remove.

RequireComponent()

```
template<typename TComponent >
void System::RequireComponent ( )
```

Specifies a required component for the system.

Specifies a required component for a system.

Template Parameters

TComponent	The type of component required.

The documentation for this class was generated from the following files:

- ECS/ECS.hpp
- ECS/ECS.cpp

1.48 TextComponent Struct Reference

Component that handles text rendering properties.

```
#include <TextComponent.hpp>
```

Public Member Functions

- TextComponent (const std::string &text="", const std::string &fontId="", u_char r=0, u_char g=0, u_char b=0, u_char a=0)

    Construct a new Text Component object.

Public Attributes

- std::string **text**

    The text string to be displayed.
- std::string **fontId**

    ID of the font resource to use.
- SDL_Color **color**

    Color of the text (RGBA)
- int **width**

    Width of the rendered text in pixels.
- int **height**

    Height of the rendered text in pixels.

1.48.1 Detailed Description

Component that handles text rendering properties.

Stores text content, font information, and rendering properties for text display.

1.48.2 Constructor & Destructor Documentation

TextComponent()

```
TextComponent::TextComponent (
            const std::string & text = "",
            const std::string & fontId = "",
            u_char r = 0,
            u_char g = 0,
            u_char b = 0,
            u_char a = 0 )  [inline]
```

Construct a new Text Component object.

Parameters

text	The text content (default: "")
fontId	ID of the font to use (default: "")
r	Red component (0-255) (default: 0)
g	Green component (0-255) (default: 0)
b	Blue component (0-255) (default: 0)
a	Alpha (transparency) component (0-255) (default: 0)

The documentation for this struct was generated from the following file:

- Components/TextComponent.hpp

1.49 TransformComponent Struct Reference

Component that handles entity transformation properties.

```
#include <TransformComponent.hpp>
```

Public Member Functions

- TransformComponent (glm::vec2 position=glm::vec2(0.0, 0.0), glm::vec2 scale=glm::vec2(1.0, 1.0), double rotation=0.0)

    Construct a new Transform Component object.

Public Attributes

- glm::vec2 **position**

    2D position vector (x,y coordinates)
- glm::vec2 **scale**

    2D scale vector (width,height multipliers)
- double **rotation**

    Rotation angle in degrees.

1.49.1 Detailed Description

Component that handles entity transformation properties.

Stores position, scale, and rotation information for entity transformation.

1.49.2 Constructor & Destructor Documentation

TransformComponent()

```
TransformComponent::TransformComponent (
            glm::vec2 position = glm::vec2(0.0,0.0),
            glm::vec2 scale = glm::vec2(1.0,1.0),
            double rotation = 0.0 )  [inline]
```

Construct a new Transform Component object.

Parameters

position	Initial position (default: (0.0, 0.0))
scale	Initial scale (default: (1.0, 1.0))
rotation	Initial rotation in degrees (default: 0.0)

The documentation for this struct was generated from the following file:

- Components/TransformComponent.hpp

1.50 UISystem Class Reference

System for handling UI interactions in the ECS framework.

```
#include <UISystem.hpp>
```

Public Member Functions

- UISystem ()

  Constructs a UISystem and requires necessary components.
- void SubscribeToClickEvent (std::unique_ptr< EventManager > &eventManager)

  Subscribes the system to click events.
- void OnClickEvent (ClickEvent &e)

  Handles click events on UI elements.

Public Member Functions inherited from System

- **System** ()=default

  Default constructor for the system.
- ∼**System** ()=default

  Default destructor for the system.
- void AddEntityToSystem (Entity entity)

  Adds an entity to the system.
- void RemoveEntityFromSystem (Entity entity)

  Removes an entity from the system.
- std::vector< Entity > GetSystemEntities () const

  Gets the list of entities managed by this system.
- const Signature & GetComponentSignature () const

  Gets the component signature of the system.
- template<typename TComponent >
  void RequireComponent ()

  Specifies a required component for the system.

1.50.1 Detailed Description

System for handling UI interactions in the ECS framework.

Manages clickable UI elements and handles click events by checking collision between click positions and UI elements.

1.50.2 Constructor & Destructor Documentation

UISystem()

```
UISystem::UISystem ( )  [inline]
```

Constructs a UISystem and requires necessary components.

Requires entities to have ClickableComponent, TransformComponent, and TextComponent to be processed by this system.

1.50.3 Member Function Documentation

OnClickEvent()

```
void UISystem::OnClickEvent (
             ClickEvent & e )  [inline]
```

Handles click events on UI elements.

Parameters

e	Reference to the ClickEvent containing click coordinates.

Checks if click coordinates are within any UI element's bounds and triggers the associated onClick Lua callback if it exists.

SubscribeToClickEvent()

```
void UISystem::SubscribeToClickEvent (
             std::unique_ptr< EventManager > & eventManager )  [inline]
```

Subscribes the system to click events.

Parameters

eventManager	Reference to the EventManager for event subscription.

Registers the OnClickEvent method as a callback for ClickEvents.

The documentation for this class was generated from the following file:

- Systems/UISystem.hpp

2 File Documentation

2.1 AssetManager.hpp

```
00001 #ifndef ASSETMANAGER_HPP
```

```
#define ASSETMANAGER_HPP

#include <SDL2/SDL.h>
#include <SDL2/SDL_image.h>
#include <SDL2/SDL_ttf.h>
#include <map>
#include <string>
#include <SDL2/SDL_mixer.h>
#include <unordered_map>

class AssetManager {
private:
    std::map<std::string, SDL_Texture*> textures;

    std::map<std::string, TTF_Font*> fonts;

    std::unordered_map<std::string, Mix_Chunk*> sounds;

public:
    AssetManager();

    ~AssetManager();

    void ClearAssets();

    void AddTexture(SDL_Renderer* renderer, const std::string& textureId, const std::string& filePath);

    SDL_Texture* GetTexture(const std::string& textureId);

    void AddFont(const std::string& fontId, const std::string& filePath, int fontSize);

    TTF_Font* GetFont(const std::string& fontId);

    void AddSound(const std::string& soundId, const std::string& filePath);

    Mix_Chunk* GetSound(const std::string& soundId);
};

#endif
```

2.2 LuaBinding.hpp

```
#ifndef LUABINDING_HPP
#define LUABINDING_HPP

#include <string>
#include "../Game/Game.hpp"
#include "../ECS/ECS.hpp"
#include "../Components/RigidBodyComponent.hpp"
#include "../Components/DrawableComponent.hpp"
#include "../Components/HealthComponent.hpp"
#include "../Components/TransformComponent.hpp"
#include "../Components/ProjectileComponent.hpp"
#include "../Components/SpriteComponent.hpp"
#include "../Systems/EnemySystem.hpp"
#include "../Systems/HealthSystem.hpp"
#include "../Systems/EnemySystem.hpp"
#include "../Systems/ChargeManageSystem.hpp"
#include "../Systems/DrawingEffectSystem.hpp"
#include "../Systems/RenderTextSystem.hpp"
#include <chrono>

bool IsActionActivated(const std::string& action) {
    return Game::GetInstance().controllerManager->IsActionActivated(action);
}

bool IsMouseButtonDown(const std::string& button_name) {
    return Game::GetInstance().controllerManager->IsMouseButtonDown(button_name);
}
std::tuple<int, int> GetMousePosition() {
    return Game::GetInstance().controllerManager->GetMousePosition();
}
std::tuple<int, int> GetPlayerPosition() {
    auto& registry = Game::GetInstance().registry;

    // Obtener todas las entidades gestionadas por HealthSystem
    auto entities = registry->GetEntitiesFromSystem<HealthSystem>();

    for (auto& entity : entities) {
        if (entity.HasComponent<HealthComponent>()) {
            auto& health = entity.GetComponent<HealthComponent>();
            if (health.isPlayer && entity.HasComponent<TransformComponent>()) {
                auto& transform = entity.GetComponent<TransformComponent>();
                return std::make_tuple(
                    static_cast<int>(transform.position.x),
                    static_cast<int>(transform.position.y)
                );
            }
        }
    }

    // Si no se encuentra el jugador, retornar 0,0
    return std::make_tuple(0, 0);
}

std::tuple<int, int> GetEnemyPosition(Entity self) {
    if (self.HasComponent<TransformComponent>()) {
        auto& transform = self.GetComponent<TransformComponent>();
        return std::make_tuple(
            static_cast<int>(transform.position.x),
            static_cast<int>(transform.position.y)
        );
    }

    // Si no tiene componente de posición, devolver por defecto
    return std::make_tuple(0, 0);
}
std::tuple<int, int> GetEnemyPositionById(int id) {
    auto& registry = Game::GetInstance().registry;
    auto entities = registry->GetEntitiesFromSystem<EnemySystem>();

    for (const auto& entity : entities) {
        if (entity.GetId() == id) {
            if (entity.HasComponent<TransformComponent>()) {
                auto& transform = entity.GetComponent<TransformComponent>();
                return std::make_tuple(
                    static_cast<int>(transform.position.x),
                    static_cast<int>(transform.position.y)
                );
            }
            break; // Encontramos la entidad, pero no tiene TransformComponent
        }
    }
    // No se encontró la entidad o no tiene TransformComponent
    return std::make_tuple(0, 0);
}
int GetAllEnemies(lua_State* L) {
    auto& registry = Game::GetInstance().registry;
    auto entities = registry->GetEntitiesFromSystem<EnemySystem>();

    lua_newtable(L);
    int index = 1;
    for (const auto& entity : entities) {
        lua_pushinteger(L, static_cast<lua_Integer>(entity.GetId()));
        lua_rawseti(L, -2, index);
        index++;
    }

    return 1;
}
void AttackMelee(Entity attacker) {
    auto& registry = Game::GetInstance().registry;

    if (!attacker.HasComponent<HealthComponent>()) return;

    Entity playerEntity(-1);
    bool foundPlayer = false;

    // Obtener entidades manejadas por HealthSystem
    auto entities = registry->GetEntitiesFromSystem<HealthSystem>();

    for (auto& entity : entities) {
        if (entity.HasComponent<HealthComponent>()) {
            auto& health = entity.GetComponent<HealthComponent>();
            if (health.isPlayer) {
                playerEntity = entity;
                foundPlayer = true;
                break;
            }
        }
    }

    if (foundPlayer && playerEntity.HasComponent<HealthComponent>()) {
        auto& attackerHealth = attacker.GetComponent<HealthComponent>();
        auto& healthSystem = registry->GetSystem<HealthSystem>();
        healthSystem.ReduceHP(playerEntity, attackerHealth.damage, attacker);
    }
}
void AttackRanger(Entity attacker) {
    auto& registry = Game::GetInstance().registry;
    if (!attacker.HasComponent<HealthComponent>()) return;

    // Obtener entidades manejadas por HealthSystem
    auto entities = registry->GetEntitiesFromSystem<EnemySystem>();

    for (auto& entity : entities) { // buggeado
        if (entity.HasComponent<ProjectileComponent>() &&
entity.GetComponent<SpriteComponent>().active == false) {

            auto& health = attacker.GetComponent<HealthComponent>();
            float damageInterval = health.attackTimeout;
            // Obtener tiempo actual
            auto now = std::chrono::steady_clock::now();

            // Calcular tiempo transcurrido desde el último daño recibido por esta entidad específica
            auto elapsed = std::chrono::duration_cast<std::chrono::milliseconds>(now -
attacker.GetComponent<HealthComponent>().attackTimeoutDuration).count();
            int intervalMs = static_cast<int>(damageInterval * 1000); //damageInterval es tiempos de
disparos

            // Si no ha pasado suficiente tiempo, no aplicar daño
            if (elapsed < intervalMs) {
                return;
            }
            attacker.GetComponent<HealthComponent>().attackTimeoutDuration = now;
            auto& ene = registry->GetSystem<EnemySystem>();
            std::tuple<int, int> playerPos = GetPlayerPosition();
            glm::vec2 enemyPosition = attacker.GetComponent<TransformComponent>().position;

            // Convertir posición del jugador a vec2
            glm::vec2 playerPosition = glm::vec2(std::get<0>(playerPos), std::get<1>(playerPos));

            // Calcular el vector dirección del enemigo hacia el jugador
            glm::vec2 direction = playerPosition - enemyPosition;

            // Normalizar la dirección para obtener el vector unitario
            glm::vec2 normalizedDirection = glm::normalize(direction);

            // Definir la velocidad del proyectil (ajusta este valor según tu juego)
            float projectileSpeed = 100.0f; // pixels per second, por ejemplo

            // Calcular la velocidad final
            glm::vec2 velocity = normalizedDirection * projectileSpeed;

            // Calcular la rotación en radianes usando atan2
            double arrowRotation = atan2(direction.y, direction.x);

            // Si necesitas la rotación en grados en lugar de radianes:
            double arrowRotationDegrees = glm::degrees(arrowRotation);
            ene.CreateEnemyProjectile(registry, velocity, enemyPosition, arrowRotationDegrees,
attacker.GetComponent<HealthComponent>().damage);
        }
    }
}
void SetLevel(int level) {
    Game::GetInstance().currentLevel = level;
}
void CurrentDrawIndex(Entity entity, int index) {

    if (Game::GetInstance().drawIndex == -1) {
        Game::GetInstance().drawIndex = index;
        auto& registry = Game::GetInstance().registry;
        for (auto entity : registry->GetSystem<RenderTextSystem>().GetSystemEntities()) {
            if (entity.HasComponent<DamageChargeComponent>() ||
entity.HasComponent<SprintChargeComponent>() || entity.HasComponent<SlowChargeComponent>() ) {
                if (entity.HasComponent<DamageChargeComponent>() && index == 0) {
                    entity.GetComponent<SpriteComponent>().active = true;
                } else if (entity.HasComponent<SprintChargeComponent>() && index == 1) {
                    entity.GetComponent<SpriteComponent>().active = true;
                } else if (entity.HasComponent<SlowChargeComponent>() && index == 2) {
                    entity.GetComponent<SpriteComponent>().active = true;
                }

            }
        }
    } else {
        int prevIndex = Game::GetInstance().drawIndex;
        Game::GetInstance().drawIndex = index;
        auto& registry = Game::GetInstance().registry;

        for (auto entity : registry->GetSystem<RenderTextSystem>().GetSystemEntities()) {
            if (entity.HasComponent<DamageChargeComponent>() ||
entity.HasComponent<SprintChargeComponent>() || entity.HasComponent<SlowChargeComponent>() ) {
                if (entity.HasComponent<DamageChargeComponent>() && index == 0) {

                    entity.GetComponent<SpriteComponent>().active = true;
                } else if (entity.HasComponent<SprintChargeComponent>() && index == 1) {
                    entity.GetComponent<SpriteComponent>().active = true;
                } else if (entity.HasComponent<SlowChargeComponent>() && index == 2) {
                    entity.GetComponent<SpriteComponent>().active = true;
                } else if (entity.HasComponent<DamageChargeComponent>() && prevIndex == 0) {

                    entity.GetComponent<SpriteComponent>().active = false;
                } else if (entity.HasComponent<SprintChargeComponent>() && prevIndex == 1) {
                    entity.GetComponent<SpriteComponent>().active = false;
                } else if (entity.HasComponent<SlowChargeComponent>() && prevIndex == 2) {
                    entity.GetComponent<SpriteComponent>().active = false;
                }

            }
        }
    }


}



void SetVelocity(Entity entity, float x, float y) {
    auto& rigidBody = entity.GetComponent<RigidBodyComponent>();
    rigidBody.velocity.x = x;
    rigidBody.velocity.y = y;
}
void GoToScene(const std::string& sceneName) {
    Game::GetInstance().sceneManager->SetNextScene(sceneName);
    Game::GetInstance().sceneManager->StopScene();
}
void PushDrawPoint(Entity entity, int index, int x, int y) {
    auto& draw = entity.GetComponent<DrawableComponent>();
    if (index >= 0 && index < (int)draw.colorPoints.size() && Game::GetInstance().registry->GetSystem<ChargeManageSystem>().HasSufficientCharge(index) == true) {
        // TODO: aunque no se dibujen igual se cuentan (hacer chequeo de la posicion (menor a 70/75 en Y))
        draw.colorPoints[index].emplace_back(glm::vec2(x, y), std::chrono::steady_clock::now());
        Game::GetInstance().registry->GetSystem<ChargeManageSystem>().ConsumeChargeForDrawing(index);
    }
}

#endif // LUABINDING_HPP
```

2.3 AnimationComponent.hpp

```
#ifndef ANIMATIONCOMPONENT_HPP
#define ANIMATIONCOMPONENT_HPP

#include <glm/glm.hpp>
#include <SDL2/SDL.h>

struct AnimationComponent {
    int numFrames;

    int currentFrame;

    int frameSpeedRate;

    bool isLoop;

    int startTime;

    AnimationComponent(int numFrames = 1, int frameSpeedRate = 1, bool isLoop = true) {
        this->numFrames = numFrames;
        this->currentFrame = 1;
        this->frameSpeedRate = frameSpeedRate;
        this->isLoop = isLoop;
        this->startTime = SDL_GetTicks();
    }
};

#endif
```

2.4 CircleColliderComponent.hpp

```
#ifndef CIRCLECOLLIDERCOMPONENT_HPP
#define CIRCLECOLLIDERCOMPONENT_HPP

struct CircleColliderComponent {
    int radius;

    int width;

    int height;

    CircleColliderComponent(int radius = 0, int width = 0, int height = 0) {
        this->width = width;
        this->height = height;
        this->radius = radius;
    }
};

#endif
```

2.5 ClickableComponent.hpp

```
#ifndef CLICKABLECOMPONENT_HPP
#define CLICKABLECOMPONENT_HPP

struct ClickableComponent {
    bool isClicked;

    ClickableComponent() {
        isClicked = false;
    }
};

#endif // CLICKABLECOMPONENT_HPP
```

2.6 DamageChargeComponent.hpp

```
#ifndef DAMAGECHARGECOMPONENT_HPP
#define DAMAGECHARGECOMPONENT_HPP

#include <string>

struct DamageChargeComponent {
    int totalCharge;

    int currentCharge;

    std::string chargeDisplay;

    DamageChargeComponent(int total = 100, int initialCharge = 100) {
        totalCharge = total;
        currentCharge = initialCharge;
        updateChargeDisplay();
    }

    void updateChargeDisplay() {
        chargeDisplay = std::to_string(currentCharge) + "/" + std::to_string(totalCharge);
    }

    void Recharge() {
        currentCharge = totalCharge;
        updateChargeDisplay();
    }

    void Charge(int amount) {
        currentCharge += amount;
        if (currentCharge > totalCharge) {
            currentCharge = totalCharge;
        }
        updateChargeDisplay();
    }

    void Discharge(int amount) {
        currentCharge -= amount;
        if (currentCharge < 0) {
            currentCharge = 0;
        }
        updateChargeDisplay();
    }

    float GetPercentage() const {
        if (totalCharge == 0) return 0.0f;
        return (static_cast<float>(currentCharge) / static_cast<float>(totalCharge)) * 100.0f;
    }

    bool IsFullyCharged() const {
        return currentCharge == totalCharge;
    }

    bool IsEmpty() const {
        return currentCharge == 0;
    }
};

#endif // DAMAGECHARGECOMPONENT_HPP
```

2.7 DrawableComponent.hpp

```
#ifndef DRAWABLE_COMPONENT_HPP
#define DRAWABLE_COMPONENT_HPP

#include <SDL2/SDL.h>
#include <vector>
#include <glm/vec2.hpp>
#include <chrono>

struct DrawableComponent {
    SDL_Color color;

    std::vector<std::vector<std::pair<glm::vec2, std::chrono::steady_clock::time_point»> colorPoints;

    DrawableComponent() {
        color = {255, 255, 255, 255}; // White
        colorPoints.resize(3); // Red, blue, green
    }

    DrawableComponent(SDL_Color col) {
        color = col;
        colorPoints.resize(3); // Red, blue, green
    }
};

#endif // DRAWABLE_COMPONENT_HPP
```

2.8 EffectReceiverComponent.hpp

```
#ifndef EFFECTRECEIVERCOMPONENT_HPP
#define EFFECTRECEIVERCOMPONENT_HPP

struct EffectReceiverComponent {
    bool takingDamage = false;

    bool slowed = false;

    bool speedBoosted = false;

    EffectReceiverComponent() = default;
};

#endif // EFFECTRECEIVERCOMPONENT_HPP
```

2.9 EnemyComponent.hpp

```
#ifndef ENEMYCOMPONENT_HPP
#define ENEMYCOMPONENT_HPP

struct EnemyComponent {
    int amountToSpawn;

    int spawnerId;

    int totalAmount;

    int points;

    EnemyComponent(int amountToSpawn = 0, int spawnerId = 0, int totalAmount = 0, int points = 0)
        : amountToSpawn(amountToSpawn), spawnerId(spawnerId), totalAmount(totalAmount), points(points)
{}
};

#endif // ENEMYCOMPONENT_HPP
```

2.10 HealthComponent.hpp

```cpp
#ifndef HEALTHCOMPONENT_HPP
#define HEALTHCOMPONENT_HPP

#include <chrono>

struct HealthComponent {
    int health;

    int maxHealth;

    bool isPlayer;

    int damage;

    float attackTimeout;

    std::chrono::steady_clock::time_point attackTimeoutDuration;

    std::chrono::steady_clock::time_point lastDamageReceived;

    HealthComponent(int health = 100, int maxHealth = 100, bool isPlayer = false, int damage = 0, float attackTimeout = 0.0f)
    {
        this->health = health;
        this->maxHealth = maxHealth;
        this->isPlayer = isPlayer;
        this->damage = damage;
        this->attackTimeout = attackTimeout;
        // Initialize timestamps to allow immediate damage or attack
        this->attackTimeoutDuration = std::chrono::steady_clock::now() - std::chrono::seconds(1);
        this->lastDamageReceived = std::chrono::steady_clock::now() - std::chrono::seconds(-1);
    }
};

#endif
```

2.11 IdentifierComponent.hpp

```cpp
#ifndef IDENTIFIERCOMPONENT_HPP
#define IDENTIFIERCOMPONENT_HPP

#include <string>

struct IdentifierComponent {
    int id;

    std::string name;

    IdentifierComponent() = default;

    IdentifierComponent(int id, const std::string& name)
        : id(id), name(name) {}
};

#endif
```

2.12 ProjectileComponent.hpp

```cpp
#ifndef PROJECTILECOMPONENT_HPP
#define PROJECTILECOMPONENT_HPP

#include <../libs/glm/glm.hpp>

struct ProjectileComponent {
    glm::vec2 velocity;

    glm::vec2 position;

    glm::vec2 scale;

    double rotation;

    bool hasHit = false;

    int damage;

    ProjectileComponent(glm::vec2 position = glm::vec2(0.0, 0.0), glm::vec2 scale = glm::vec2(1.0, 1.0), double rotation = 0.0, const glm::vec2& velocity = glm::vec2(0.0f, 0.0f)) {
        this->velocity = velocity;
        this->position = position;
        this->scale = scale;
        this->rotation = rotation;
    }
};

#endif
```

2.13 RigidBodyComponent.hpp

```
#ifndef RIGIDBODYCOMPONENT_HPP
#define RIGIDBODYCOMPONENT_HPP

#include <glm/glm.hpp>

struct RigidBodyComponent {
    glm::vec2 velocity;

    RigidBodyComponent(const glm::vec2& velocity = glm::vec2(0.0f, 0.0f)) {
        this->velocity = velocity;
    }
};

#endif
```

2.14 ScriptComponent.hpp

```
#ifndef SCRIPT_COMPONENT_HPP
#define SCRIPT_COMPONENT_HPP

#include <sol/sol.hpp>

struct ScriptComponent {
    sol::function update;

    sol::function onClick;

    ScriptComponent(sol::function update = sol::lua_nil,
                    sol::function onClick = sol::lua_nil) {
        this->update = update;
        this->onClick = onClick;
    }
};

#endif // SCRIPT_COMPONENT_HPP
```

2.15 SlowChargeComponent.hpp

```
#ifndef SLOWCHARGECOMPONENT_HPP
#define SLOWCHARGECOMPONENT_HPP

#include <string>

struct SlowChargeComponent {
    int totalCharge;

    int currentCharge;

    std::string chargeDisplay;

    SlowChargeComponent(int total = 100, int initialCharge = 100) {
        totalCharge = total;
        currentCharge = initialCharge;
        updateChargeDisplay();
    }

    void updateChargeDisplay() {
        chargeDisplay = std::to_string(currentCharge) + "/" + std::to_string(totalCharge);
    }

    void Recharge() {
        currentCharge = totalCharge;
        updateChargeDisplay();
    }

    void Charge(int amount) {
        currentCharge += amount;
        if (currentCharge > totalCharge) {
            currentCharge = totalCharge;
        }
        updateChargeDisplay();
    }

    void Discharge(int amount) {
        currentCharge -= amount;
        if (currentCharge < 0) {
            currentCharge = 0;
        }
        updateChargeDisplay();
    }

    float GetPercentage() const {
        if (totalCharge == 0) return 0.0f;
        return (static_cast<float>(currentCharge) / static_cast<float>(totalCharge)) * 100.0f;
    }

    bool IsFullyCharged() const {
        return currentCharge == totalCharge;
    }

    bool IsEmpty() const {
        return currentCharge == 0;
    }
};

#endif // SLOWCHARGECOMPONENT_HPP
```

2.16 SoundComponent.hpp

```
#ifndef SOUNDCOMPONENT_HPP
#define SOUNDCOMPONENT_HPP

#include <string>

struct SoundComponent {
    std::string soundId;

    int volume;

    int loops;

    bool isPlaying;

    bool active;

    bool autoPlay;

    SoundComponent(const std::string& soundId = "none", int volume = 128, int loops = -1, bool autoPlay = true, bool active = true) {
        this->soundId = soundId;
        this->volume = volume;
        this->loops = loops;
        this->isPlaying = false;
        this->active = active;
        this->autoPlay = autoPlay;
    }
};

#endif
```

2.17 SprintChargeComponent.hpp

```
#ifndef SPRINTCHARGECOMPONENT_HPP
#define SPRINTCHARGECOMPONENT_HPP
#include <string>

struct SprintChargeComponent {
    int totalCharge;
    int currentCharge;
    std::string chargeDisplay;

    SprintChargeComponent(int total = 100, int initialCharge = 100) {
        totalCharge = total;
        currentCharge = initialCharge;
        updateChargeDisplay();
    }

    void updateChargeDisplay() {
        chargeDisplay = std::to_string(currentCharge) + "/" + std::to_string(totalCharge);
    }

    void Recharge() {
        currentCharge = totalCharge;
        updateChargeDisplay();
    }

    void Charge(int amount) {
        currentCharge += amount;
        if (currentCharge > totalCharge) {
            currentCharge = totalCharge;
        }
        updateChargeDisplay();
    }

    void Discharge(int amount) {
        currentCharge -= amount;
        if (currentCharge < 0) {
            currentCharge = 0;
        }
        updateChargeDisplay();
    }

    float GetPercentage() const {
        if (totalCharge == 0) return 0.0f;
        return (static_cast<float>(currentCharge) / static_cast<float>(totalCharge)) * 100.0f;
    }

    bool IsFullyCharged() const {
        return currentCharge == totalCharge;
    }

    bool IsEmpty() const {
        return currentCharge == 0;
    }
};
#endif // SPRINTCHARGECOMPONENT_HPP
```

2.18 SpriteComponent.hpp

```
#ifndef SPRITECOMPONENT_HPP
#define SPRITECOMPONENT_HPP
#include <SDL2/SDL.h>
#include <string>

struct SpriteComponent {
    SDL_Rect srcRect;
    std::string textureId;
    int width;
    int height;
    SDL_RendererFlip flip;
    bool active;

    SpriteComponent(const std::string& textureId = "none",
                    int width = 0,
                    int height = 0,
                    int srcRectX = 0,
                    int srcRectY = 0,
                    bool active = true)
    {
        this->textureId = textureId;
        this->width = width;
        this->height = height;
        this->srcRect = { srcRectX, srcRectY, width, height };
        this->flip = SDL_FLIP_NONE;
        this->active = active;
    }
};
#endif // SPRITECOMPONENT_HPP
```

2.19 TextComponent.hpp

```
#ifndef TEXT_COMPONENT_HPP
#define TEXT_COMPONENT_HPP
#include <SDL2/SDL.h>
#include <SDL2/SDL_ttf.h>
#include <string>

struct TextComponent {
    std::string text;
    std::string fontId;
    SDL_Color color;
    int width;
    int height;

    TextComponent(const std::string& text = "",
                 const std::string& fontId = "",
                 u_char r = 0,
                 u_char g = 0,
                 u_char b = 0,
                 u_char a = 0) {
        this->text = text;
        this->fontId = fontId;
        this->color.r = r;
        this->color.g = g;
        this->color.b = b;
        this->color.a = a;
        this->width = 0;
        this->height = 0;
    }
};

#endif // TEXT_COMPONENT_HPP
```

2.20 TransformComponent.hpp

```
#ifndef TRANSFORM_COMPONENT_HPP
#define TRANSFORM_COMPONENT_HPP
#include <../libs/glm/glm.hpp>

struct TransformComponent {
    glm::vec2 position;
    glm::vec2 scale;
    double rotation;

    TransformComponent(glm::vec2 position = glm::vec2(0.0,0.0),
                      glm::vec2 scale = glm::vec2(1.0,1.0),
                      double rotation = 0.0) {
        this->position = position;
        this->scale = scale;
        this->rotation = rotation;
    }
};

#endif // TRANSFORM_COMPONENT_HPP
```

2.21 ControllerManager.hpp

```
#ifndef CONTROLLER_MANAGER_HPP
#define CONTROLLER_MANAGER_HPP

#include <SDL2/SDL.h>
#include <map>
#include <string>
#include <tuple>

class ControllerManager {
private:
    std::map<std::string, int> actionKeyName;
    std::map<int, bool> keyDown;

    std::map<std::string, int> mouseButtonName;
    std::map<int, bool> mouseButtonDown;

    int mousePosX;
    int mousePosY;

public:
    ControllerManager();

    ~ControllerManager();

    void Clear();

    // Keyboard related methods

    void AddActionKey(const std::string& action, int keyCode);

    void KeyDown(int keyCode);

    void KeyUp(int keyCode);

    bool IsActionActivated(const std::string& action);

    // Mouse related methods

    void AddMouseButton(const std::string& action, int buttonCode);

    void MouseButtonDown(int buttonCode);

    void MouseButtonUp(int buttonCode);

    bool IsMouseButtonDown(const std::string& name);

    void SetMousePosition(int x, int y);

    std::tuple<int, int> GetMousePosition();
};

#endif // CONTROLLER_MANAGER_HPP
```

2.22 ECS.hpp

```
#ifndef ECS_HPP
#define ECS_HPP

#include <cstddef>
#include <bitset>
#include <memory>
#include <vector>
#include <set>
#include <deque>
#include <typeindex>
#include <unordered_map>
#include <iostream>
#include "../Utils/Pool.hpp"

const unsigned int MAX_COMPONENTS = 64;

typedef std::bitset<MAX_COMPONENTS> Signature;

struct IComponent {
protected:
    static int nextId;
};

template <typename TComponent>
class Component : public IComponent {
public:
    static int GetId() {
        static int id = nextId++;
        return id;
    }
};

class Entity {
private:
    int id;

public:
    Entity(int id) : id(id) {}

    int GetId() const;

    void Kill();

    bool operator==(const Entity& other) const { return id == other.id; }

    bool operator!=(const Entity& other) const { return id != other.id; }

    bool operator>(const Entity& other) const { return id > other.id; }

    bool operator<(const Entity& other) const { return id < other.id; }

    template <typename TComponent, typename... TArgs>
    void AddComponent(TArgs&&... args);

    template <typename TComponent>
    void RemoveComponent();

    template <typename TComponent>
    bool HasComponent() const;

    template <typename TComponent>
    TComponent& GetComponent() const;

    class Registry* registry;
};

class System {
private:
    Signature componentSignature;

    std::vector<Entity> entities;

public:
    System() = default;

    ~System() = default;

    void AddEntityToSystem(Entity entity);

    void RemoveEntityFromSystem(Entity entity);

    std::vector<Entity> GetSystemEntities() const;

    const Signature& GetComponentSignature() const;

    template <typename TComponent>
    void RequireComponent();
};

class Registry {
private:
    int numEntity = 0;

    std::vector<std::shared_ptr<IPool» componentsPools;

    std::vector<Signature> entityComponentSignatures;

    std::unordered_map<std::type_index, std::shared_ptr<System» systems;

    std::set<Entity> entitiesToBeAdded;

    std::set<Entity> entitiesToBeKilled;

    std::deque<int> freeIds;

public:
    Registry();

    ~Registry();

    void Update();

    Entity CreateEntity();

    void KillEntity(Entity entity);

    template <typename TComponent, typename... TArgs>
    void AddComponent(Entity entity, TArgs&&... args);

    template <typename TComponent>
    void RemoveComponent(Entity entity);

    template <typename TComponent>
    bool HasComponent(Entity entity) const;

    template <typename TComponent>
    TComponent& GetComponent(Entity entity) const;

    template <typename TSystem, typename... TArgs>
    void AddSystem(TArgs&&... args);

    template <typename TSystem>
    void RemoveSystem();

    template <typename TSystem>
    bool HasSystem() const;

    template <typename TSystem>
    TSystem& GetSystem() const;

    void AddEntityToSystems(Entity entity);

    void RemoveEntityFromSystems(Entity entity);

    template<typename T>
    std::vector<Entity> GetEntitiesFromSystem();

    void ClearAllEntities();
};

```

```
template <typename TComponent>
void System::RequireComponent() {
    const int componentId = Component<TComponent>::GetId();
    componentSignature.set(componentId);
}

template<typename T>
std::vector<Entity> Registry::GetEntitiesFromSystem() {
    if (!HasSystem<T>()) return {};

    auto& systemDerived = GetSystem<T>();
    return systemDerived.GetSystemEntities();
}

template <typename TComponent, typename... TArgs>
void Registry::AddComponent(Entity entity, TArgs&&... args) {
    const int componentId = Component<TComponent>::GetId();
    const int entityId = entity.GetId();

    if (static_cast<long unsigned int>(componentId) >= componentsPools.size()) {
        componentsPools.resize(componentId + 10, nullptr);
    }

    if (!componentsPools[componentId]) {
        std::shared_ptr<Pool<TComponent>> newComponentPool = std::make_shared<Pool<TComponent>>();
        componentsPools[componentId] = newComponentPool;
    }

    std::shared_ptr<Pool<TComponent>> componentPool
        = std::static_pointer_cast<Pool<TComponent>>(componentsPools[componentId]);

    if (entityId >= componentPool->GetSize()) {
        componentPool->Resize(numEntity + 100);
    }

    TComponent newComponent(std::forward<TArgs>(args)...);
    componentPool->Set(entityId, newComponent);
    entityComponentSignatures[entityId].set(componentId);
}

template <typename TComponent>
void Registry::RemoveComponent(Entity entity) {
    const int componentId = Component<TComponent>::GetId();
    const int entityId = entity.GetId();

    entityComponentSignatures[entityId].set(componentId, false);
}

template <typename TComponent>
bool Registry::HasComponent(Entity entity) const {
    const int componentId = Component<TComponent>::GetId();
    const int entityId = entity.GetId();

    return entityComponentSignatures[entityId].test(componentId);
}

template <typename TComponent>
TComponent& Registry::GetComponent(Entity entity) const {
    const int componentId = Component<TComponent>::GetId();
    const int entityId = entity.GetId();

    auto componentPool =
        std::static_pointer_cast<Pool<TComponent>>(componentsPools[componentId]);
    return componentPool->Get(entityId);
}

template <typename TSystem, typename... TArgs>
void Registry::AddSystem(TArgs&&... args) {
    std::shared_ptr<TSystem> newSystem = std::make_shared<TSystem>(std::forward<TArgs>(args)...);
    systems.insert(std::make_pair(std::type_index(typeid(TSystem)), newSystem));
}

template <typename TSystem>
void Registry::RemoveSystem() {
    auto system = systems.find(std::type_index(typeid(TSystem)));
    systems.erase(system);
}

template <typename TSystem>
bool Registry::HasSystem() const {
    return systems.find(std::type_index(typeid(TSystem))) != systems.end();
}

template <typename TSystem>
TSystem& Registry::GetSystem() const {
    auto system = systems.find(std::type_index(typeid(TSystem)));
    return *(std::static_pointer_cast<TSystem>(system->second));
}

template <typename TComponent, typename... TArgs>
void Entity::AddComponent(TArgs&&... args) {
    registry->AddComponent<TComponent>(*this, std::forward<TArgs>(args)...);
}

template <typename TComponent>
void Entity::RemoveComponent() {
    registry->RemoveComponent<TComponent>(*this);
}

template <typename TComponent>
bool Entity::HasComponent() const {
    return registry->HasComponent<TComponent>(*this);
}

template <typename TComponent>
TComponent& Entity::GetComponent() const {
    return registry->GetComponent<TComponent>(*this);
}

#endif
```

2.23 Event.hpp

```
#ifndef EVENT_HPP
#define EVENT_HPP

class Event {
public:
    Event() = default;

    virtual ~Event() = default;

    //virtual void execute() = 0;

    //virtual void undo() = 0;

    //virtual void redo() = 0;

};

#endif // EVENT_HPP
```

2.24 EventManager.hpp

```
#ifndef EVENTMANAGER_HPP
#define EVENTMANAGER_HPP

#include "Event.hpp"
#include <list>
#include <memory>
#include <functional>
#include <iostream>
#include <map>
#include <typeindex>

class IEventCallback {
public:
    virtual ~IEventCallback() = default;

    void Excute(Event& event) {
        Call(event);
    }

private:
    virtual void Call(Event& event) = 0;
};

template<typename TOwner, typename TEvent>
class EventCallback : public IEventCallback {
private:
    typedef void (TOwner::*CallbackFunction)(TEvent&);
    TOwner* ownerInstance;
    CallbackFunction callbackFunction;

    virtual void Call(Event& event) override {
        std::invoke(callbackFunction, ownerInstance, static_cast<TEvent&>(event));
    }

public:
    EventCallback(TOwner* owner, CallbackFunction callback) {
        this->ownerInstance = owner;
        this->callbackFunction = callback;
    }
};

typedef std::list<std::unique_ptr<IEventCallback» HandlerList;

class EventManager {
private:
    std::map<std::type_index, std::unique_ptr<HandlerList» subscribers;

public:
    EventManager() {
        std::cout « "[EventManager] Se ejecuta constructor" « std::endl;
    };

    ~EventManager() {
        std::cout « "[EventManager] Se ejecuta destructor" « std::endl;
    };

    void Restart() {
        subscribers.clear();
    };

    template<typename TOwner, typename TEvent>
    void SubscribeToEvent(TOwner* owner, void (TOwner::*callback)(TEvent&)) {
        if (!subscribers[typeid(TEvent)].get()) {
            subscribers[typeid(TEvent)] = std::make_unique<HandlerList>();
        }
        auto subscriber = std::make_unique<EventCallback<TOwner, TEvent»(owner, callback);
        subscribers[typeid(TEvent)]->push_back(std::move(subscriber));
    };

    template<typename TEvent, typename... TArgs>
    void EmitEvent(TArgs&&... args) {
        auto handlers = subscribers[typeid(TEvent)].get();
        if (handlers) {
            for (auto it = handlers->begin(); it != handlers->end(); ++it) {
                auto handler = it->get();
                TEvent event(std::forward<TArgs>(args)...);
                handler->Excute(event);
            }
        }
    };
};

#endif // EVENTMANAGER_HPP
```

2.25 ClickEvent.hpp

```
#ifndef CLICKEVENT_HPP
#define CLICKEVENT_HPP

#include "../ECS/ECS.hpp"
#include "../EventManager/Event.hpp"

class ClickEvent : public Event {
public:
    int buttonCode;
    int x;
    int y;

    ClickEvent(int buttonCode = 0, int x = 0, int y = 0) {
        this->buttonCode = buttonCode;
        this->x = x;
        this->y = y;
    }
};

#endif // CLICKEVENT_HPP
```

2.26 CollisionEvent.hpp

```
#ifndef COLLISIONEVENT_HPP
#define COLLISIONEVENT_HPP

#include "../ECS/ECS.hpp"
#include "../EventManager/Event.hpp"

```

```
class CollisionEvent : public Event {
public:
    Entity entityA;
    Entity entityB;

    CollisionEvent(Entity entityA, Entity entityB)
        : entityA(entityA), entityB(entityB) {
    };

    ~CollisionEvent() {
    };
};

#endif // COLLISIONEVENT_HPP
```

2.27 Game.hpp

```
#ifndef GAME_HPP
#define GAME_HPP

#include <SDL2/SDL.h>
#include <SDL2/SDL_image.h>
#include <SDL2/SDL_ttf.h>
#include <SDL2/SDL_mixer.h>
#include <glm/glm.hpp>
#include <iostream>
#include <sol/sol.hpp>  // Sol2 al final
#include <string>
#include <fstream>
#include <vector>
#include <memory>
#include "../AssetManager/AssetManager.hpp"
#include "../ControllerManager/ControllerManager.hpp"
#include "../EventManager/EventManager.hpp"
#include "../ECS/ECS.hpp"
#include "../SceneManager/SceneManager.hpp"

const int FPS = 30;

const int MILLISECS_PER_FRAME = 1000 / FPS;

class Game {
private:
    SDL_Window* window = nullptr;

    int millisecsPreviousFrame = 0;

    bool isRunning = false;

    int mPreviousFrame = 0;

public:
    SDL_Renderer* renderer = nullptr;

    size_t windowWidth = 0;

    size_t windowHeight = 0;

    std::unique_ptr<ControllerManager> controllerManager;

    std::unique_ptr<AssetManager> assetManager;

    std::unique_ptr<EventManager> eventManager;

    std::unique_ptr<Registry> registry;

    std::unique_ptr<SceneManager> sceneManager;

    sol::state lua;

    int enemiesLeftToSpawn = 0;

    int enemiesLeft = 0;

    int totalPoints = 0;

    int totalPointsPrev = 0;

    bool finDelNivel = false;

    bool win = false;

    bool isPaused = false;

    int drawIndex = -1;

    int currentLevel = 0;

private:
    void Setup();

    void RunScene();

    void processInput();

    void update();

    void render();

    void readConfig();

    Game();

    ~Game();

public:
    static Game& GetInstance();

    void init();

    void run();

    void destroy();
};

#endif
```

2.28 SceneLoader.hpp

```
#ifndef SCENELOADER_HPP
#define SCENELOADER_HPP

#include <SDL2/SDL.h>
#include <memory>
#include <sol/sol.hpp>
#include <string>
#include "../AssetManager/AssetManager.hpp"
#include "../ControllerManager/ControllerManager.hpp"
#include "../ECS/ECS.hpp"

class SceneLoader {
private:
    void LoadBackground(SDL_Renderer* renderer, const sol::table& background, std::unique_ptr<AssetManager>& assetManager);

    void LoadSprites(SDL_Renderer* renderer, const sol::table& sprites, std::unique_ptr<AssetManager>& assetManager);

    void LoadFonts(const sol::table& fonts, std::unique_ptr<AssetManager>& assetManager);

    void LoadSounds(const sol::table& sounds, std::unique_ptr<AssetManager>& assetManager);

    void LoadKey(const sol::table& keys, std::unique_ptr<ControllerManager>& controllerManager);

    void LoadButtons(const sol::table& buttons, std::unique_ptr<ControllerManager>& controllerManager);

    void LoadEntities(sol::state& lua, const sol::table& entities, std::unique_ptr<Registry>& registry);

public:
    SceneLoader();

    ~SceneLoader();

    void LoadScene(const std::string& scenePath, sol::state& lua,
        std::unique_ptr<AssetManager>& assetManager, std::unique_ptr<ControllerManager>& controllerManager,
        std::unique_ptr<Registry>& registry, SDL_Renderer* renderer);
};

#endif // SCENELOADER_HPP
```

2.29 SceneManager.hpp

```
#ifndef SCENEMANAGER_HPP
```

```
#define SCENEMANAGER_HPP

#include <map>
#include <memory>
#include <sol/sol.hpp>
#include <string>
#include "../SceneManager/SceneLoader.hpp"

class SceneManager {
private:
    std::map<std::string, std::string> scenes;
    std::string nextScene;
    bool isSceneRunning = false;
    std::unique_ptr<SceneLoader> sceneLoader;

public:
    SceneManager();

    ~SceneManager();

    void LoadSceneFromScript(const std::string& scenePath, sol::state& lua);

    void LoadScene();

    std::string GetNextScene() const;

    void SetNextScene(const std::string& nextScene);

    bool IsSceneRunning() const;

    void StartScene();

    void StopScene();
};

#endif // SCENEMANAGER_HPP
```

2.30 AnimationSystem.hpp

```
#ifndef ANIMATIONSYSTEM_HPP
#define ANIMATIONSYSTEM_HPP

#include "../ECS/ECS.hpp"
#include "../Components/AnimationComponent.hpp"
#include "../Components/SpriteComponent.hpp"
#include <SDL2/SDL.h>

class AnimationSystem : public System {
public:
    AnimationSystem() {
        RequireComponent<AnimationComponent>();
        RequireComponent<SpriteComponent>();
    }

    void Update() {
        for (auto entity : GetSystemEntities()) {
            auto& animation = entity.GetComponent<AnimationComponent>();
            auto& sprite = entity.GetComponent<SpriteComponent>();

            // Calculate current frame based on elapsed time and animation speed
            animation.currentFrame = ((SDL_GetTicks() - animation.startTime) *
                                    animation.frameSpeedRate / 1000) %
                                    animation.numFrames;

            // Update sprite source rectangle to show current frame
            sprite.srcRect.x = animation.currentFrame * sprite.width;
        }
    }
};

#endif // ANIMATIONSYSTEM_HPP
```

2.31 ChargeManageSystem.hpp

```
#ifndef CHARGEMANAGESYSTEM_HPP
#define CHARGEMANAGESYSTEM_HPP

#include "../ECS/ECS.hpp"
#include "../Components/DamageChargeComponent.hpp"
#include "../Components/SprintChargeComponent.hpp"
#include "../Components/SlowChargeComponent.hpp"
```

```
#include "../Game/Game.hpp"
#include <chrono>

const int MINIMUM_CHARGE = 1;

class ChargeManageSystem : public System {
private:
    const int NATURAL_RECHARGE_RATE = 5;
    const float RECHARGE_INTERVAL = 1.0f;

    std::chrono::steady_clock::time_point lastRechargeTime;

public:
    ChargeManageSystem() {
        lastRechargeTime = std::chrono::steady_clock::now();
    }

    void Update() {
        auto now = std::chrono::steady_clock::now();
        auto timeSinceLastRecharge = std::chrono::duration_cast<std::chrono::milliseconds>(now -
lastRechargeTime).count() / 1000.0f;

        // Only recharge if enough time has passed
        if (timeSinceLastRecharge >= RECHARGE_INTERVAL) {
            auto allEntities = GetSystemEntities();

            for (auto entity : allEntities) {
                // Recharge DamageCharge
                if (entity.HasComponent<DamageChargeComponent>()) {
                    auto& charge = entity.GetComponent<DamageChargeComponent>();
                    if (!charge.IsFullyCharged()) {
                        charge.Charge(NATURAL_RECHARGE_RATE);
                    }
                }

                // Recharge SprintCharge
                if (entity.HasComponent<SprintChargeComponent>()) {
                    auto& charge = entity.GetComponent<SprintChargeComponent>();
                    if (!charge.IsFullyCharged()) {
                        charge.Charge(NATURAL_RECHARGE_RATE);
                    }
                }

                // Recharge SlowCharge
                if (entity.HasComponent<SlowChargeComponent>()) {
                    auto& charge = entity.GetComponent<SlowChargeComponent>();
                    if (!charge.IsFullyCharged()) {
                        charge.Charge(NATURAL_RECHARGE_RATE);
                    }
                }
            }

            lastRechargeTime = now;
        }
    }

    bool HasSufficientCharge(int colorIndex) {
        for (auto& entity : GetSystemEntities()) {
            if (entity.HasComponent<DamageChargeComponent>() ||
                entity.HasComponent<SprintChargeComponent>() ||
                entity.HasComponent<SlowChargeComponent>()) {

                // Check specific charge type
                if (entity.HasComponent<DamageChargeComponent>() && colorIndex == 0) {
                    return entity.GetComponent<DamageChargeComponent>().currentCharge >=
MINIMUM_CHARGE;
                } else if (entity.HasComponent<SprintChargeComponent>() && colorIndex == 1) {
                    return entity.GetComponent<SprintChargeComponent>().currentCharge >=
MINIMUM_CHARGE;
                } else if (entity.HasComponent<SlowChargeComponent>() && colorIndex == 2) {
                    return entity.GetComponent<SlowChargeComponent>().currentCharge >= MINIMUM_CHARGE;
                }
            }
        }
        return false;
    }

    bool ConsumeChargeForDrawing(int colorIndex) {
        for (auto& entity : GetSystemEntities()) {
            if (entity.HasComponent<DamageChargeComponent>() ||
                entity.HasComponent<SprintChargeComponent>() ||
                entity.HasComponent<SlowChargeComponent>()) {

                // Check and consume specific charge type
                if (entity.HasComponent<DamageChargeComponent>() && colorIndex == 0) {
                    auto& charge = entity.GetComponent<DamageChargeComponent>();
                    if (charge.currentCharge >= MINIMUM_CHARGE) {
                        charge.Discharge(MINIMUM_CHARGE);
                        return true;
                    }
                } else if (entity.HasComponent<SprintChargeComponent>() && colorIndex == 1) {
                    auto& charge = entity.GetComponent<SprintChargeComponent>();
                    if (charge.currentCharge >= MINIMUM_CHARGE) {
                        charge.Discharge(MINIMUM_CHARGE);
                        return true;
                    }
                } else if (entity.HasComponent<SlowChargeComponent>() && colorIndex == 2) {
                    auto& charge = entity.GetComponent<SlowChargeComponent>();
                    if (charge.currentCharge >= MINIMUM_CHARGE) {
                        charge.Discharge(MINIMUM_CHARGE);
                        return true;
                    }
                }
            }
        }
        return false; // Failed to consume charge
    }
};

#endif // CHARGEMANAGESYSTEM_HPP
```

2.32 CollisionSystem.hpp

```
#ifndef COLLISIONSYSTEM_HPP
#define COLLISIONSYSTEM_HPP

#include "../ECS/ECS.hpp"
#include "../Components/CircleColliderComponent.hpp"
#include "../Components/TransformComponent.hpp"
#include "../Components/HealthComponent.hpp"
#include "../Components/ProjectileComponent.hpp"
#include "../EventManager/EventManager.hpp"
#include "../Events/CollisionEvent.hpp"
#include <iostream>
#include <memory>

class CollisionSystem : public System {
public:
    CollisionSystem() {
        RequireComponent<CircleColliderComponent>();
        RequireComponent<TransformComponent>();
    }

    void Update(std::unique_ptr<EventManager>& eventManager) {
        auto entities = GetSystemEntities();

        // Check all entity pairs for collisions (n^2/2 checks)
        for (auto i = entities.begin(); i != entities.end(); ++i) {
            auto entityA = *i;
            auto transformA = entityA.GetComponent<TransformComponent>();
            auto colliderA = entityA.GetComponent<CircleColliderComponent>();

            for (auto j = std::next(i); j != entities.end(); ++j) {
                auto entityB = *j;

                auto transformB = entityB.GetComponent<TransformComponent>();
                auto colliderB = entityB.GetComponent<CircleColliderComponent>();

                // Calculate world-space centers including scale and offset
                glm::vec2 centerA = glm::vec2(
                    transformA.position.x + (colliderA.width / 2.0f) * transformA.scale.x,
                    transformA.position.y + (colliderA.height / 2.0f) * transformA.scale.y
                );

                glm::vec2 centerB = glm::vec2(
                    transformB.position.x + (colliderB.width / 2.0f) * transformB.scale.x,
                    transformB.position.y + (colliderB.height / 2.0f) * transformB.scale.y
                );

                // Calculate scaled radii
                float aRadius = colliderA.radius * transformA.scale.x;
                float bRadius = colliderB.radius * transformB.scale.x;

                // Check for collision
                bool collision = CheckCircularCollision(aRadius, bRadius, centerA, centerB);

                if (collision) {
                    eventManager->EmitEvent<CollisionEvent>(entityA, entityB);
                }
            }
        }
    }

private:
    bool CheckCircularCollision(float aRadius, float bRadius, const glm::vec2& aPos, const glm::vec2& bPos) {
        glm::vec2 diff = aPos - bPos;
        float distanceSquared = diff.x * diff.x + diff.y * diff.y;
        float radiusSum = aRadius + bRadius;

        // Compare squared distances to avoid sqrt operation
        return (radiusSum * radiusSum) >= distanceSquared;
    }
};

#endif // COLLISIONSYSTEM_HPP
```

2.33 DamageSystem.hpp

```
#ifndef DAMAGESYSTEM_HPP
#define DAMAGESYSTEM_HPP

#include <memory>
#include "../Components/CircleColliderComponent.hpp"
#include "../Components/HealthComponent.hpp"
#include "../Components/ProjectileComponent.hpp"
#include "../ECS/ECS.hpp"
#include "../EventManager/EventManager.hpp"
#include "../Events/CollisionEvent.hpp"

class DamageSystem : public System {
    public:
    DamageSystem() {
        RequireComponent<CircleColliderComponent>();
    }
    void SubscribeToCollisionEvent(std::unique_ptr<EventManager>& eventManager) {
        eventManager->SubscribeToEvent<DamageSystem, CollisionEvent>(this, &DamageSystem::OnCollision);
    }

    void OnCollision(CollisionEvent& event) {
        if (event.entityA.HasComponent<HealthComponent>() && event.entityB.HasComponent<ProjectileComponent>()) {
            auto& health = event.entityA.GetComponent<HealthComponent>();
            auto& arrow = event.entityB.GetComponent<HealthComponent>();



            if (health.isPlayer) {
                health.health -= arrow.damage;
                if (health.health <= 0) {
                    health.health = 0;
                    Game::GetInstance().finDelNivel = true;
                    Game::GetInstance().win = false;
                }
                event.entityB.Kill();
            }
        } else if (event.entityB.HasComponent<HealthComponent>() && event.entityA.HasComponent<ProjectileComponent>()) {
            auto& health = event.entityB.GetComponent<HealthComponent>();
            auto& arrow = event.entityA.GetComponent<HealthComponent>();

            if (health.isPlayer) {
                health.health -= arrow.damage;
                if (health.health <= 0) {
                    health.health = 0;
                    Game::GetInstance().finDelNivel = true;
                    Game::GetInstance().win = false;
                }
                event.entityA.Kill();
            }
        }
    }



};

#endif
```

2.34 DrawingEffectSystem.hpp

```
#ifndef DRAWINGEFFECTSYSTEM_HPP
```

```
#define DRAWINGEFFECTSYSTEM_HPP

#include "../ECS/ECS.hpp"
#include "../Components/DrawableComponent.hpp"
#include "../Components/EffectReceiverComponent.hpp"
#include "../Components/TransformComponent.hpp"
#include "../Components/CircleColliderComponent.hpp"
#include "../Components/EnemyComponent.hpp"
#include "../Systems/CollisionSystem.hpp"
#include <glm/vec2.hpp>
#include "../Game/Game.hpp"

class DrawingEffectSystem : public System {
private:
    const int EFFECT_RADIUS = 15; // Radio de detección aumentado para mejor cobertura
    const float DAMAGE_INTERVAL = 1.0f; // Intervalo de daño en segundos

public:
    DrawingEffectSystem() {
        RequireComponent<DrawableComponent>();
    }
    void Update() {
        for (auto drawingEntity : GetSystemEntities()) {
            auto& drawable = drawingEntity.GetComponent<DrawableComponent>();

            if (!drawable.colorPoints.empty() && !drawable.colorPoints[0].empty()) {
                ProcessDamageEffect(drawable.colorPoints[0]);
            }

            for (size_t colorIndex = 1; colorIndex < drawable.colorPoints.size(); ++colorIndex) {
                ProcessOtherEffects(drawable.colorPoints[colorIndex], colorIndex);
            }
        }
    }

private:
    void ProcessDamageEffect(const std::vector<std::pair<glm::vec2,
std::chrono::steady_clock::time_point»& points) {
        auto now = std::chrono::steady_clock::now();
        auto& registry = Game::GetInstance().registry;
        auto entitiesWithCollider = registry->GetEntitiesFromSystem<CollisionSystem>();

        int validPointsCount = 0;
        for (const auto& point : points) {
            auto duration = std::chrono::duration_cast<std::chrono::seconds>(now - point.second);
            if (duration.count() <= 4 && point.first.y > 75) {
                validPointsCount++;
            }
        }

        for (auto entity : entitiesWithCollider) {
            if (!entity.HasComponent<EffectReceiverComponent>() ||
                !entity.HasComponent<TransformComponent>() ||
                !entity.HasComponent<EnemyComponent>()) { // Solo enemigos reciben daño
                continue;
            }

            auto& transform = entity.GetComponent<TransformComponent>();
            auto& collider = entity.GetComponent<CircleColliderComponent>();
            auto& effectReceiver = entity.GetComponent<EffectReceiverComponent>();

            glm::vec2 entityCenter = glm::vec2(
                transform.position.x + (collider.width * transform.scale.x / 2),
                transform.position.y + (collider.height * transform.scale.y / 2)
            );

            int entityRadius = collider.radius * std::max(transform.scale.x, transform.scale.y) / 2;

            // Verificar si la entidad está actualmente sobre algún trazo rojo válido
            bool isOnDamageTrace = false;
            int collisionCount = 0;

            for (const auto& point : points) {
                auto duration = std::chrono::duration_cast<std::chrono::seconds>(now - point.second);
                if (duration.count() > 4) continue;

                if (point.first.y <= 75) continue;

                if (CheckPointToCircleCollision(point.first, entityCenter, entityRadius +
EFFECT_RADIUS)) {
                    isOnDamageTrace = true;
                    collisionCount++;
                }
            }

            effectReceiver.takingDamage = isOnDamageTrace;

        }
    }
    void ProcessOtherEffects(const std::vector<std::pair<glm::vec2, std::chrono::steady_clock::time_point»& points, int colorIndex) {
        auto now = std::chrono::steady_clock::now();
        auto& registry = Game::GetInstance().registry;
        auto entitiesWithCollider = registry->GetEntitiesFromSystem<CollisionSystem>();

        for (auto entity : entitiesWithCollider) {
            if (!entity.HasComponent<EffectReceiverComponent>() || !entity.HasComponent<TransformComponent>()) {
                continue;
            }

            auto& transform = entity.GetComponent<TransformComponent>();
            auto& collider = entity.GetComponent<CircleColliderComponent>();
            auto& effectReceiver = entity.GetComponent<EffectReceiverComponent>();

            glm::vec2 entityCenter = glm::vec2(
                transform.position.x + (collider.width * transform.scale.x / 2),
                transform.position.y + (collider.height * transform.scale.y / 2)
            );

            int entityRadius = collider.radius * transform.scale.x / 2;
            bool isOnTrace = false;

            for (const auto& point : points) {
                auto duration = std::chrono::duration_cast<std::chrono::seconds>(now - point.second);
                if (duration.count() > 4) continue;
                if (point.first.y <= 75) continue;

                if (CheckPointToCircleCollision(point.first, entityCenter, entityRadius + EFFECT_RADIUS)) {
                    isOnTrace = true;
                    break;
                }
            }

            // Aplicar efectos según el color
            bool isEnemy = entity.HasComponent<EnemyComponent>();

            switch (colorIndex) {
                case 1: // Azul - Speed (solo jugador)
                    if (!isEnemy) {
                        effectReceiver.speedBoosted = isOnTrace;
                    } else {
                        effectReceiver.speedBoosted = false;
                    }
                    break;

                case 2: // Verde - Slow (solo enemigos)
                    if (isEnemy) {
                        effectReceiver.slowed = isOnTrace;
                    } else {
                        effectReceiver.slowed = false;
                    }
                    break;
            }
        }
    }
    bool CheckPointToCircleCollision(const glm::vec2& point, const glm::vec2& circleCenter, int radius) {
        glm::vec2 diff = point - circleCenter;
        double distance = glm::sqrt((diff.x * diff.x) + (diff.y * diff.y));
        return distance <= radius;
    }
};

#endif // DRAWINGEFFECTSYSTEM_HPP
```

2.35 DrawSystem.hpp

```
#ifndef DRAWSYSTEM_HPP
#define DRAWSYSTEM_HPP

#include <SDL2/SDL.h>
#include <vector>
#include "../ECS/ECS.hpp"
#include "../Components/DrawableComponent.hpp"
#include <chrono>
#include "../Game/Game.hpp"
#include "../Systems/ChargeManageSystem.hpp"

class DrawSystem : public System {
```

```
public:
    DrawSystem() {
        RequireComponent<DrawableComponent>();
    }

    void Update(SDL_Renderer* renderer) {
        // Process each entity with drawable component
        for (auto entity : GetSystemEntities()) {
            auto& drawable = entity.GetComponent<DrawableComponent>();

            // Process each color channel
            for (size_t i = 0; i < drawable.colorPoints.size(); ++i) {
                // Set color based on channel index
                SDL_Color color;
                switch (i) {
                    case 0: color = {255, 0, 0, 255}; break;   // Red
                    case 1: color = {0, 0, 255, 255}; break;   // Blue
                    case 2: color = {0, 255, 0, 255}; break;   // Green
                    default: color = {255, 255, 255, 255}; break; // White
                }

                SDL_SetRenderDrawColor(renderer, color.r, color.g, color.b, color.a);

                // Remove points older than 4 seconds when game is not paused
                if (!Game::GetInstance().isPaused) {
                    auto now = std::chrono::steady_clock::now();
                    for (auto it = drawable.colorPoints[i].begin(); it != drawable.colorPoints[i].end(); ) {
                        auto duration = std::chrono::duration_cast<std::chrono::seconds>(now - it->second);
                        if (duration.count() > 4) {
                            it = drawable.colorPoints[i].erase(it); // Remove expired point
                        } else {
                            ++it;
                        }
                    }
                }

                // Draw remaining points in gameplay area (below y=175)
                for (const auto& point : drawable.colorPoints[i]) {
                    if (point.first.y > 175) {
                        int size = 10; // Stroke size
                        SDL_Rect drawRect = {
                            static_cast<int>(point.first.x) - size / 2,
                            static_cast<int>(point.first.y) - size / 2,
                            size,
                            size
                        };
                        SDL_RenderFillRect(renderer, &drawRect);
                    }
                }
            }
        }
    }
};

#endif // DRAWSYSTEM_HPP
```

2.36 EnemySystem.hpp

```
#ifndef ENEMYSYSTEM_HPP
#define ENEMYSYSTEM_HPP

#include "../ECS/ECS.hpp"
#include "../Components/EnemyComponent.hpp"
#include "../Components/AnimationComponent.hpp"
#include "../Components/CircleColliderComponent.hpp"
#include "../Components/HealthComponent.hpp"
#include "../Components/RigidBodyComponent.hpp"
#include "../Components/SpriteComponent.hpp"
#include "../Components/TransformComponent.hpp"
#include "../Components/ScriptComponent.hpp"
#include "../Components/DrawableComponent.hpp"
#include "../Components/EnemyComponent.hpp"
#include "../Components/EffectReceiverComponent.hpp"
#include "../Components/TextComponent.hpp"
#include "../Components/ProjectileComponent.hpp"
#include <memory>
#include <cstdlib>

class EnemySystem : public System {
public:
    EnemySystem() {
        RequireComponent<EnemyComponent>();
    }

    void Update(std::unique_ptr<Registry>& registry) {
        Game& game = Game::GetInstance();
        for (auto spawner : GetSystemEntities()) {
            auto& enemySpawner = spawner.GetComponent<EnemyComponent>();

            int currentAlive = CountClonesFrom(spawner.GetId());
            if (currentAlive < enemySpawner.amountToSpawn && enemySpawner.totalAmount > 0) {
                Entity newEnemy = registry->CreateEntity();
                CloneEntityFromTemplate(spawner, newEnemy);
                enemySpawner.totalAmount--;
                game.enemiesLeftToSpawn--;
            }
        }
    }

    void CreateEnemyProjectile(std::unique_ptr<Registry>& registry, glm::vec2 velocity, glm::vec2 position, double rotation, int damage) {
        for (auto spawner : GetSystemEntities()) {
            if (spawner.HasComponent<ProjectileComponent>()) {
                Entity newEnemy = registry->CreateEntity();
                CloneEntityFromTemplate(spawner, newEnemy);
                newEnemy.GetComponent<TransformComponent>().position = position;
                newEnemy.GetComponent<TransformComponent>().rotation = rotation;
                newEnemy.GetComponent<RigidBodyComponent>().velocity = velocity;
                newEnemy.GetComponent<HealthComponent>().damage = damage;
                break;
            }
        }
    }

private:
    int CountClonesFrom(int spawnerId) {
        int count = 0;
        for (auto entity : GetSystemEntities()) {
            auto& enemy = entity.GetComponent<EnemyComponent>();
            if (entity.GetId() != spawnerId && enemy.spawnerId == spawnerId) {
                count++;
            }
        }
        return count;
    }

    void CloneEntityFromTemplate(Entity source, Entity target) {
        // Animation
        if (source.HasComponent<AnimationComponent>()) {
            target.AddComponent<AnimationComponent>(source.GetComponent<AnimationComponent>());
        }

        // Collider
        if (source.HasComponent<CircleColliderComponent>()) {
            target.AddComponent<CircleColliderComponent>(source.GetComponent<CircleColliderComponent>());
        }

        // Health
        if (source.HasComponent<HealthComponent>()) {
            target.AddComponent<HealthComponent>(source.GetComponent<HealthComponent>());
        }

        // Rigidbody
        if (source.HasComponent<RigidBodyComponent>()) {
            target.AddComponent<RigidBodyComponent>(source.GetComponent<RigidBodyComponent>());
        }

        // Script
        if (source.HasComponent<ScriptComponent>()) {
            auto script = source.GetComponent<ScriptComponent>();
            target.AddComponent<ScriptComponent>(script);
        }

        // Sprite
        if (source.HasComponent<SpriteComponent>()) {
            auto sprite = source.GetComponent<SpriteComponent>();
            sprite.active = true;  // Activate visibility for clone
            target.AddComponent<SpriteComponent>(sprite);
        }

        // Transform with random position
        if (source.HasComponent<TransformComponent>()) {
            auto transform = source.GetComponent<TransformComponent>();
            transform.position = GetRandomSpawnPosition();
            target.AddComponent<TransformComponent>(transform);
        }

        if (source.HasComponent<EffectReceiverComponent>()) {
```

```
target.AddComponent<EffectReceiverComponent>(source.GetComponent<EffectReceiverComponent>());
        }

        if (source.HasComponent<TextComponent>()) {
            target.AddComponent<TextComponent>(source.GetComponent<TextComponent>());
        }

        if (source.HasComponent<ProjectileComponent>()) {
            target.AddComponent<ProjectileComponent>(source.GetComponent<ProjectileComponent>());
        }

        // EnemyComponent without spawn capability
        if (source.HasComponent<EnemyComponent>()) {
            target.AddComponent<EnemyComponent>(source.GetComponent<EnemyComponent>());
            target.GetComponent<EnemyComponent>().amountToSpawn = 0;
            target.GetComponent<EnemyComponent>().spawnerId = source.GetId();
        }
    }

    glm::vec2 GetRandomSpawnPosition() {
        int x, y;

        // Choose left or right side
        bool leftRight = rand() % 2;
        if (leftRight) {
            x = rand() % 41; // 0 - 40
        } else {
            x = 760 + (rand() % 41); // 760 - 800
        }

        // Choose top or bottom
        bool topBottom = rand() % 2;
        if (topBottom) {
            y = 75 + (rand() % 31); // 75 - 105
        } else {
            y = 560 + (rand() % 41); // 560 - 600
        }

        return glm::vec2(static_cast<float>(x), static_cast<float>(y));
    }
};

#endif // ENEMYSYSTEM_HPP
```

2.37 HealthSystem.hpp

```
#ifndef HEALTHSYSTEM_HPP
#define HEALTHSYSTEM_HPP

#include <memory>
#include "../Components/HealthComponent.hpp"
#include "../Components/EffectReceiverComponent.hpp"
#include "../Components/RigidBodyComponent.hpp"
#include "../ECS/ECS.hpp"

class HealthSystem : public System {
public:
    HealthSystem() {
        RequireComponent<HealthComponent>();
    }

    void Update() {
        auto& registry = Game::GetInstance().registry;
        Entity playerEntity(-1);
        bool foundPlayer = false;

        // Get entities managed by HealthSystem
        auto entities = registry->GetEntitiesFromSystem<HealthSystem>();

        // Find the player entity
        for (auto& entity : entities) {
            if (entity.HasComponent<HealthComponent>()) {
                auto& health = entity.GetComponent<HealthComponent>();
                if (health.isPlayer) {
                    playerEntity = entity;
                    foundPlayer = true;
                    break;
                }
            }
        }

        if (foundPlayer && playerEntity.HasComponent<HealthComponent>()) {
            auto& playerHealth = playerEntity.GetComponent<HealthComponent>();
            auto& playerDamage = playerHealth.damage;
            auto& playerTimeout = playerHealth.attackTimeout;

            // Process all effect-receiving entities
            for (auto& entity : entities) {
                if (entity.HasComponent<HealthComponent>() && entity.HasComponent<EffectReceiverComponent>()) {
                    auto& effectReceiver = entity.GetComponent<EffectReceiverComponent>();
                    auto& entityHealth = entity.GetComponent<HealthComponent>();

                    // Process zone damage ONLY if entity is currently in damage zone
                    if (effectReceiver.takingDamage && !entityHealth.isPlayer) {
                        ProcessZoneDamage(entity, playerDamage, playerTimeout);
                    }
                    ApplySpeedEffect(entity, effectReceiver, entityHealth);
                    // Process other effects
                    if (effectReceiver.slowed && !entityHealth.isPlayer) {
                        // Apply slow logic
                        // std::cout « "Enemy slowed" « std::endl;
                    }

                    if (effectReceiver.speedBoosted && entityHealth.isPlayer) {
                        // Apply speed boost logic
                        // std::cout « "Player speed boosted" « std::endl;
                    }
                }
            }
        }
    }

private:
    void ApplySpeedEffect(Entity& entity, EffectReceiverComponent& effectReceiver, HealthComponent& entityHealth) {
        bool isPlayer = entityHealth.isPlayer;
        if (isPlayer && effectReceiver.speedBoosted) {
            entity.GetComponent<RigidBodyComponent>().velocity *= 1.5f; // Increase speed
        } else if (!isPlayer && effectReceiver.slowed) {
            entity.GetComponent<RigidBodyComponent>().velocity *= 0.3f; // Reduce speed
        }
    }

    void ProcessZoneDamage(Entity entity, int damage, float damageInterval) {
        if (!entity.HasComponent<HealthComponent>()) return;

        auto& targetHealth = entity.GetComponent<HealthComponent>();

        // Get current time
        auto now = std::chrono::steady_clock::now();

        // Calculate time since last damage
        auto elapsed = std::chrono::duration_cast<std::chrono::milliseconds>(now - targetHealth.lastDamageReceived).count();
        int intervalMs = static_cast<int>(damageInterval * 1000); // seconds to ms

        // Skip if not enough time passed
        if (elapsed < intervalMs) {
            return;
        }

        // Verify entity is still taking damage
        if (!entity.HasComponent<EffectReceiverComponent>()) return;
        auto& effectReceiver = entity.GetComponent<EffectReceiverComponent>();
        if (!effectReceiver.takingDamage) return;

        // Update last damage time
        targetHealth.lastDamageReceived = now;

        // Apply damage
        targetHealth.health -= damage;

        // Check for death
        if (targetHealth.health <= 0) {
            targetHealth.health = 0;
            if (!targetHealth.isPlayer) {
                Game::GetInstance().totalPoints += entity.GetComponent<EnemyComponent>().points;
                Game::GetInstance().enemiesLeft--;
                entity.Kill();
                if (Game::GetInstance().enemiesLeft == 0) {
                    Game::GetInstance().finDelNivel = true;
                    Game::GetInstance().win = true;
                }
            }
        }
    }

public:
    void ReduceHP(Entity entity, int damage, Entity attacker) {
        if (!entity.HasComponent<HealthComponent>()) return;
        if (!attacker.HasComponent<HealthComponent>()) return;

        auto& attackerHealth = attacker.GetComponent<HealthComponent>();
        auto& targetHealth = entity.GetComponent<HealthComponent>();

        // Get current time
        auto now = std::chrono::steady_clock::now();

        // Calculate time since attacker's last attack
        auto elapsed = std::chrono::duration_cast<std::chrono::milliseconds>(now - attackerHealth.attackTimeoutDuration).count();
        int timeoutMs = static_cast<int>(attackerHealth.attackTimeout * 1000);

        if (elapsed < timeoutMs) {
            return;
        }

        // Update attacker's last attack time
        attackerHealth.attackTimeoutDuration = now;

        // Apply damage
        targetHealth.health -= damage;

        if (targetHealth.health <= 0) {
            targetHealth.health = 0;
            if (!targetHealth.isPlayer) {
                entity.Kill();
            } else {
                Game::GetInstance().finDelNivel = true;
                Game::GetInstance().win = false;
            }
        }
    }

    void SetHealth(Entity entity, int value) {
        if (entity.HasComponent<HealthComponent>()) {
            auto& health = entity.GetComponent<HealthComponent>();
            health.health = std::max(0, value);

            if (health.health == 0 && !health.isPlayer) {
                entity.Kill();
            }
        }
    }

    void Heal(Entity entity, int amount) {
        if (entity.HasComponent<HealthComponent>()) {
            auto& health = entity.GetComponent<HealthComponent>();
            health.health += amount;
            health.health = std::min(health.health, health.maxHealth);
        }
    }
};

#endif // HEALTHSYSTEM_HPP
```

2.38 MovementSystem.hpp

```
#ifndef MOVEMENTSYSTEM_HPP
#define MOVEMENTSYSTEM_HPP

#include "../Components/RigidBodyComponent.hpp"
#include "../Components/TransformComponent.hpp"
#include "../Components/SpriteComponent.hpp"
#include "../Components/ProjectileComponent.hpp"
#include "../ECS/ECS.hpp"
#include "../Game/Game.hpp"

class MovementSystem : public System {
public:
    MovementSystem() {
        RequireComponent<RigidBodyComponent>();
        RequireComponent<TransformComponent>();
        RequireComponent<SpriteComponent>();
    }

    void Update(double dt) {
        auto& game = Game::GetInstance();

        for (auto entity : GetSystemEntities()) {
            const auto& rigidBody = entity.GetComponent<RigidBodyComponent>();
            auto& transform = entity.GetComponent<TransformComponent>();
            auto& sprite = entity.GetComponent<SpriteComponent>();

            // Skip inactive sprites
            if (!sprite.active) {
                continue;
            }

            // Update position based on velocity and delta time
            transform.position.x += rigidBody.velocity.x * dt;
            transform.position.y += rigidBody.velocity.y * dt;

            bool crash = false;

            // X-axis boundary checks (with sprite width consideration)
            if (transform.position.x < 0) {
                transform.position.x = 0;
                crash = true;
            } else if (transform.position.x > game.windowWidth - (sprite.width * transform.scale.x)) {
                transform.position.x = game.windowWidth - (sprite.width * transform.scale.x);
                crash = true;
            }

            // Y-axis boundary checks (with sprite height consideration)
            if (transform.position.y < 175) {  // Top boundary
                transform.position.y = 175;
                crash = true;
            } else if (transform.position.y > game.windowHeight - (sprite.height * transform.scale.y) - 25) {  // Bottom boundary
                transform.position.y = game.windowHeight - (sprite.height * transform.scale.y) - 25;
                crash = true;
            }

            // Update sprite orientation based on horizontal velocity
            if (rigidBody.velocity.x < 0) {
                sprite.flip = SDL_FLIP_HORIZONTAL;
            } else if (rigidBody.velocity.x > 0) {
                sprite.flip = SDL_FLIP_NONE;
            }

            // Special handling for projectiles
            if (entity.HasComponent<ProjectileComponent>()) {
                sprite.flip = SDL_FLIP_NONE;
            }

            // Destroy projectiles that hit boundaries
            if (entity.HasComponent<ProjectileComponent>() && crash) {
                sprite.flip = SDL_FLIP_NONE;
                entity.Kill();
            }
        }
    }
};

#endif // MOVEMENTSYSTEM_HPP
```

2.39 RenderSystem.hpp

```
#ifndef RENDERSYSTEM_HPP
#define RENDERSYSTEM_HPP

#include <SDL2/SDL.h>
#include "../AssetManager/AssetManager.hpp"
#include "../Components/SpriteComponent.hpp"
#include "../Components/TransformComponent.hpp"
#include "../ECS/ECS.hpp"

class RenderSystem : public System {
public:
    RenderSystem() {
        RequireComponent<TransformComponent>();
        RequireComponent<SpriteComponent>();
    }

    void Update(SDL_Renderer* renderer, std::unique_ptr<AssetManager>& AssetManager) {
        std::vector<Entity> entities = GetSystemEntities();

        size_t startIndex = 0;

        // Special case: skip background if it’s the first entity
        if (!entities.empty()) {
            const auto firstEntity = entities[0];
            if (firstEntity.HasComponent<SpriteComponent>()) {
                const auto sprite = firstEntity.GetComponent<SpriteComponent>();
                if (sprite.textureId.find("background") != std::string::npos) {
                    startIndex = 1;  // Skip first element (background)
                }
            }
        }

        // Render all entities starting from startIndex
        for (size_t i = startIndex; i < entities.size(); ++i) {
            const auto entity = entities[i];
            const auto transform = entity.GetComponent<TransformComponent>();
            const auto sprite = entity.GetComponent<SpriteComponent>();

            // Skip inactive sprites
            if (!sprite.active) {
                continue;
            }

            // Prepare source and destination rectangles
            SDL_Rect srcRect = sprite.srcRect;
            SDL_Rect dstRect = {
                static_cast<int>(transform.position.x),
                static_cast<int>(transform.position.y),
                static_cast<int>(sprite.width * transform.scale.x),
                static_cast<int>(sprite.height * transform.scale.y)
            };

            // Render with optional flip and rotation
            SDL_RenderCopyEx(
                renderer,
                AssetManager->GetTexture(sprite.textureId),
                &srcRect,
                &dstRect,
                transform.rotation,
                NULL,
                sprite.flip
            );
        }
    }

    void UpdateBackground(SDL_Renderer* renderer, std::unique_ptr<AssetManager>& AssetManager) {
        auto& registry = Game::GetInstance().registry;
        auto entities = registry->GetEntitiesFromSystem<RenderSystem>();

        for (auto entity : entities) {
            if (!entity.HasComponent<TransformComponent>()) {
                continue;
            }
            const auto transform = entity.GetComponent<TransformComponent>();
            const auto sprite = entity.GetComponent<SpriteComponent>();

            // Only render active background sprites
            if (!sprite.active || sprite.textureId.find("background") == std::string::npos) {
                continue;
            }

            // Prepare source and destination rectangles
            SDL_Rect srcRect = sprite.srcRect;
            SDL_Rect dstRect = {
                static_cast<int>(transform.position.x),
                static_cast<int>(transform.position.y),
                static_cast<int>(sprite.width * transform.scale.x),
                static_cast<int>(sprite.height * transform.scale.y)
            };

            // Render with optional flip and rotation
            SDL_RenderCopyEx(
                renderer,
                AssetManager->GetTexture(sprite.textureId),
                &srcRect,
                &dstRect,
                transform.rotation,
                NULL,
                sprite.flip
            );
        }
    }
};

#endif // RENDERSYSTEM_HPP
```

2.40 RenderTextSystem.hpp

```
#ifndef RENDERTEXTSYSTEM_HPP
#define RENDERTEXTSYSTEM_HPP

#include <SDL2/SDL.h>
```

```
#include <SDL2/SDL_ttf.h>
#include <memory>
#include "../AssetManager/AssetManager.hpp"
#include "../Components/TextComponent.hpp"
#include "../Components/TransformComponent.hpp"
#include "../Components/HealthComponent.hpp"
#include "../Components/DamageChargeComponent.hpp"
#include "../Components/SprintChargeComponent.hpp"
#include "../Components/SlowChargeComponent.hpp"
#include "../Components/IdentifierComponent.hpp"
#include "../ECS/ECS.hpp"

class RenderTextSystem : public System {
public:
    RenderTextSystem() {
        RequireComponent<TextComponent>();
        RequireComponent<TransformComponent>();
    }

    void Update(SDL_Renderer* renderer, std::unique_ptr<AssetManager>& assetManager) {
        for (auto entity : GetSystemEntities()) {
            auto& text = entity.GetComponent<TextComponent>();
            auto& transform = entity.GetComponent<TransformComponent>();

            // Case 1: Health display (floating above entities)
            if (entity.HasComponent<HealthComponent>()) {
                const auto sprite = entity.GetComponent<SpriteComponent>();
                if (!sprite.active) {
                    continue;
                }

                text.text = std::to_string(entity.GetComponent<HealthComponent>().health);
                SDL_Surface* surface = TTF_RenderText_Blended(assetManager->GetFont(text.fontId), text.text.c_str(), text.color);
                text.width = surface->w;
                text.height = surface->h;
                SDL_Texture* texture = SDL_CreateTextureFromSurface(renderer, surface);
                SDL_FreeSurface(surface);

                SDL_Rect dstrect = {
                    static_cast<int>(transform.position.x),
                    static_cast<int>(transform.position.y - 20),  // Position above entity
                    text.width * static_cast<int>(transform.scale.x) / 2,
                    text.height * static_cast<int>(transform.scale.y) / 2
                };

                SDL_RenderCopy(renderer, texture, NULL, &dstrect);
                SDL_DestroyTexture(texture);
            }
            // Case 2: Charge displays (damage, sprint, slow)
            else if (entity.HasComponent<DamageChargeComponent>() ||
                     entity.HasComponent<SprintChargeComponent>() ||
                     entity.HasComponent<SlowChargeComponent>()) {

                if (entity.HasComponent<DamageChargeComponent>()) {
                    text.text = entity.GetComponent<DamageChargeComponent>().chargeDisplay;
                } else if (entity.HasComponent<SprintChargeComponent>()) {
                    text.text = entity.GetComponent<SprintChargeComponent>().chargeDisplay;
                } else if (entity.HasComponent<SlowChargeComponent>()) {
                    text.text = entity.GetComponent<SlowChargeComponent>().chargeDisplay;
                }

                SDL_Surface* surface = TTF_RenderText_Blended(assetManager->GetFont(text.fontId), text.text.c_str(), text.color);
                text.width = surface->w;
                text.height = surface->h;
                SDL_Texture* texture = SDL_CreateTextureFromSurface(renderer, surface);
                SDL_FreeSurface(surface);

                SDL_Rect dstrect = {
                    static_cast<int>(transform.position.x),
                    static_cast<int>(transform.position.y - 20),  // Position above
                    text.width * static_cast<int>(transform.scale.x) / 2,
                    text.height * static_cast<int>(transform.scale.y) / 2
                };

                SDL_RenderCopy(renderer, texture, NULL, &dstrect);
                SDL_DestroyTexture(texture);
            }
            // Case 3: Score display
            else if (entity.HasComponent<IdentifierComponent>() &&
                     entity.GetComponent<IdentifierComponent>().name == "puntuacion") {

                text.text = "Score: " + std::to_string(Game::GetInstance().totalPoints);

                SDL_Surface* surface = TTF_RenderText_Blended(assetManager->GetFont(text.fontId), text.text.c_str(), text.color);
                text.width = surface->w;
                text.height = surface->h;
                SDL_Texture* texture = SDL_CreateTextureFromSurface(renderer, surface);
                SDL_FreeSurface(surface);

                SDL_Rect dstrect = {
                    static_cast<int>(transform.position.x),
                    static_cast<int>(transform.position.y - 20),  // Position above
                    text.width * static_cast<int>(transform.scale.x) / 2,
                    text.height * static_cast<int>(transform.scale.y) / 2
                };

                SDL_RenderCopy(renderer, texture, NULL, &dstrect);
                SDL_DestroyTexture(texture);
            }
            // Case 4: Default text rendering
            else {
                SDL_Surface* surface = TTF_RenderText_Blended(assetManager->GetFont(text.fontId),
text.text.c_str(), text.color);
                text.width = surface->w;
                text.height = surface->h;
                SDL_Texture* texture = SDL_CreateTextureFromSurface(renderer, surface);
                SDL_FreeSurface(surface);

                SDL_Rect dstrect = {
                    static_cast<int>(transform.position.x),
                    static_cast<int>(transform.position.y),
                    text.width * static_cast<int>(transform.scale.x),
                    text.height * static_cast<int>(transform.scale.y)
                };

                SDL_RenderCopy(renderer, texture, NULL, &dstrect);
                SDL_DestroyTexture(texture);
            }
        }
    }
};

#endif // RENDERTEXTSYSTEM_HPP
```

2.41 ScriptSystem.hpp

```
#ifndef SCRIPt_SYSTEM_HPP
#define SCRIPt_SYSTEM_HPP

#include <memory>
#include <sol/sol.hpp>
#include "../Binding/LuaBinding.hpp"
#include "../Components/ScriptComponent.hpp"
#include "../ECS/ECS.hpp"

class ScriptSystem : public System {
public:
    ScriptSystem() {
        RequireComponent<ScriptComponent>();
    }

    void CreateLuaBinding(sol::state& lua) {
        lua.script("math.randomseed(os.time())");
        lua.new_usertype<Entity>("entity");
        lua.set_function("is_action_activated", IsActionActivated);
        lua.set_function("set_velocity", SetVelocity);
        lua.set_function("go_to_scene", GoToScene);

        lua.set_function("is_mouse_button_down", IsMouseButtonDown);

        lua.set_function("push_draw_point", PushDrawPoint);

        lua.set_function("get_mouse_position", GetMousePosition);
        lua.set_function("get_player_position", GetPlayerPosition);
        lua.set_function("get_enemy_position", GetEnemyPosition);
        lua.set_function("attack_melee", AttackMelee);
        lua.set_function("get_all_enemies", GetAllEnemies);
        lua.set_function("get_enemy_position_by_id", GetEnemyPositionById);
        lua.set_function("attack_ranger", AttackRanger);
        lua.set_function("set_draw_index", CurrentDrawIndex);
        lua.set_function("set_level", SetLevel);
    }

    void Update(sol::state& lua) {
        for (auto entity : GetSystemEntities()) {
            const auto& script = entity.GetComponent<ScriptComponent>();

            if (script.update != sol::lua_nil) {
                lua["this"] = entity;
                script.update();
            }
        }
    }
};

#endif // SCRIPt_SYSTEM_HPP
```

2.42 SoundSystem.hpp

```
#ifndef SOUNDSYSTEM_HPP
#define SOUNDSYSTEM_HPP

#include <SDL2/SDL_mixer.h>
#include <vector>
#include <memory>
#include "../AssetManager/AssetManager.hpp"
#include "../Components/SoundComponent.hpp"
#include "../ECS/ECS.hpp"

class SoundSystem : public System {
public:
    SoundSystem() {
        RequireComponent<SoundComponent>();
    }

    void Update(std::unique_ptr<AssetManager>& assetManager) {
        std::vector<Entity> entities = GetSystemEntities();
        for (auto& entity : entities) {
            auto& sound = entity.GetComponent<SoundComponent>();

            if (!sound.active) {
                continue;
            }
            // Auto-reproducir sonidos marcados para autoPlay
            if (sound.autoPlay && !sound.isPlaying) {
                PlaySound(assetManager, sound);
            }
        }
    }

    void PlaySound(std::unique_ptr<AssetManager>& assetManager, SoundComponent& sound) {
        if (sound.soundId == "none" || !sound.active) {
            return;
        }

        Mix_Chunk* chunk = assetManager->GetSound(sound.soundId);
        if (chunk != nullptr) {
            Mix_VolumeChunk(chunk, sound.volume);
            int channel = Mix_PlayChannel(-1, chunk, sound.loops);
            sound.isPlaying = (channel != -1);
        }
    }

    void StopSound(SoundComponent& sound) {
        sound.isPlaying = false;
        // Nota: Para detener un sonido específico necesitarías trackear el canal
    }

    void PauseSound(SoundComponent& sound) {
        if (sound.isPlaying) {
            Mix_Pause(-1); // Pausa todos los canales, idealmente trackearias el canal específico
        }
    }

    void ResumeSound(SoundComponent& sound) {
        if (!sound.isPlaying) {
            Mix_Resume(-1); // Reanuda todos los canales
        }
    }
};

#endif
```

2.43 UISystem.hpp

```
#ifndef UISYSTEM_HPP
#define UISYSTEM_HPP

#include <SDL2/SDL.h>
```

```
#include <SDL2/SDL_ttf.h>

#include <memory>
#include <iostream>
#include <string>

#include "../Components/ClickableComponent.hpp"
#include "../Components/TransformComponent.hpp"
#include "../Components/ScriptComponent.hpp"
#include "../Components/TextComponent.hpp"
#include "../ECS/ECS.hpp"
#include "../EventManager/EventManager.hpp"
#include "../Events/ClickEvent.hpp"

class UISystem : public System {
    public:
        UISystem() {
            RequireComponent<ClickableComponent>();
            RequireComponent<TransformComponent>();
            RequireComponent<TextComponent>();
        };

        void SubscribeToClickEvent(std::unique_ptr<EventManager>& eventManager) {
            eventManager->SubscribeToEvent<UISystem, ClickEvent>(this, &UISystem::OnClickEvent);
        };

        void OnClickEvent(ClickEvent& e) {
            for (auto entity : GetSystemEntities()) {
                auto& transform = entity.GetComponent<TransformComponent>();
                auto& text = entity.GetComponent<TextComponent>();
                if (transform.position.x < e.x && e.x < transform.position.x + text.width &&
                    transform.position.y < e.y && e.y < transform.position.y + text.height) {

                    if (entity.HasComponent<ClickableComponent>()) {
                        const auto& script = entity.GetComponent<ScriptComponent>();
                        if (script.onClick != sol::lua_nil) {
                            script.onClick();
                        }
                    }
                }
            }
        };
};

#endif // UISYSTEM_HPP
```

2.44 Pool.hpp

```
#ifndef POOL_HPP
#define POOL_HPP
#include <vector>

class IPool {
    public:
        virtual ~IPool() = default;
};

template <typename TComponent>
class Pool : public IPool {
    private:
        std::vector<TComponent> data;

    public:
        Pool(int size = 1000) {
            data.resize(size);
        }

        virtual ~Pool() = default;

        bool IsEmpty() const {
            return data.empty();
        }

        int GetSize() const {
            return static_cast<int>(data.size());
        }

        void Resize(int n) {
            data.resize(n);
        }

        void Clear() {
            data.clear();
        }

        void Add(TComponent object) {
            data.push_back(object);
        }

        void Set(unsigned int index, TComponent object) {
            data[index] = object;
        }

        TComponent& Get(unsigned int index) {
            return static_cast<TComponent&>(data[index]);
        }

        TComponent& operator[](unsigned int index) {
            return static_cast<TComponent&>(data[index]);
        }
};

#endif
```

Index