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

soundId	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.
textureId	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

fontId	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

soundId	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

textureId	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 > &asset↩ Manager, 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 src↩ RectY=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: "")
font↩ Id	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
```

```
00002 #define ASSETMANAGER_HPP
00003
00004 #include <SDL2/SDL.h>
00005 #include <SDL2/SDL_image.h>
00006 #include <SDL2/SDL_ttf.h>
00007 #include <map>
00008 #include <string>
00009 #include <SDL2/SDL_mixer.h>
00010 #include <unordered_map>
00011
00016 class AssetManager {
00017 private:
00019     std::map<std::string, SDL_Texture*> textures;
00020
00022     std::map<std::string, TTF_Font*> fonts;
00023
00025     std::unordered_map<std::string, Mix_Chunk*> sounds;
00026
00027 public:
00031     AssetManager();
00032
00036     ~AssetManager();
00037
00041     void ClearAssets();
00042
00049     void AddTexture(SDL_Renderer* renderer, const std::string& textureId, const std::string&
      filePath);
00050
00056     SDL_Texture* GetTexture(const std::string& textureId);
00057
00064     void AddFont(const std::string& fontId, const std::string& filePath, int fontSize);
00065
00071     TTF_Font* GetFont(const std::string& fontId);
00072
00078     void AddSound(const std::string& soundId, const std::string& filePath);
00079
00085     Mix_Chunk* GetSound(const std::string& soundId);
00086 };
00087
00088 #endif
```

2.2 LuaBinding.hpp

```
00001 #ifndef LUABINDING_HPP
00002 #define LUABINDING_HPP
00003
00004 #include <string>
00005 #include "../Game/Game.hpp"
00006 #include "../ECS/ECS.hpp"
00007 #include "../Components/RigidBodyComponent.hpp"
00008 #include "../Components/DrawableComponent.hpp"
00009 #include "../Components/HealthComponent.hpp"
00010 #include "../Components/TransformComponent.hpp"
00011 #include "../Components/ProjectileComponent.hpp"
00012 #include "../Components/SpriteComponent.hpp"
00013 #include "../Systems/EnemySystem.hpp"
00014 #include "../Systems/HealthSystem.hpp"
00015 #include "../Systems/EnemySystem.hpp"
00016 #include "../Systems/ChargeManageSystem.hpp"
00017 #include "../Systems/DrawingEffectSystem.hpp"
00018 #include "../Systems/RenderTextSystem.hpp"
00019 #include <chrono>
00020
00026 bool IsActionActivated(const std::string& action) {
00027     return Game::GetInstance().controllerManager->IsActionActivated(action);
00028 }
00029
00035 bool IsMouseButtonDown(const std::string& button_name) {
00036     return Game::GetInstance().controllerManager->IsMouseButtonDown(button_name);
00037 }
00042 std::tuple<int, int> GetMousePosition() {
00043     return Game::GetInstance().controllerManager->GetMousePosition();
00044 }
00049 std::tuple<int, int> GetPlayerPosition() {
00050     auto& registry = Game::GetInstance().registry;
00051
00052     // Obtener todas las entidades gestionadas por HealthSystem
00053     auto entities = registry->GetEntitiesFromSystem<HealthSystem>();
00054
00055     for (auto& entity : entities) {
00056         if (entity.HasComponent<HealthComponent>()) {
00057             auto& health = entity.GetComponent<HealthComponent>();
00058             if (health.isPlayer && entity.HasComponent<TransformComponent>()) {
00059                 auto& transform = entity.GetComponent<TransformComponent>();
```

```
00060                 return std::make_tuple(
00061                     static_cast<int>(transform.position.x),
00062                     static_cast<int>(transform.position.y)
00063                 );
00064             }
00065         }
00066     }
00067 
00068     // Si no se encuentra el jugador, retornar 0,0
00069     return std::make_tuple(0, 0);
00070 }
00071 
00077 std::tuple<int, int> GetEnemyPosition(Entity self) {
00078     if (self.HasComponent<TransformComponent>()) {
00079         auto& transform = self.GetComponent<TransformComponent>();
00080         return std::make_tuple(
00081             static_cast<int>(transform.position.x),
00082             static_cast<int>(transform.position.y)
00083         );
00084     }
00085 
00086     // Si no tiene componente de posición, devolver por defecto
00087     return std::make_tuple(0, 0);
00088 }
00094 std::tuple<int, int> GetEnemyPositionById(int id) {
00095     auto& registry = Game::GetInstance().registry;
00096     auto entities = registry->GetEntitiesFromSystem<EnemySystem>();
00097 
00098     for (const auto& entity : entities) {
00099         if (entity.GetId() == id) {
00100             if (entity.HasComponent<TransformComponent>()) {
00101                 auto& transform = entity.GetComponent<TransformComponent>();
00102                 return std::make_tuple(
00103                     static_cast<int>(transform.position.x),
00104                     static_cast<int>(transform.position.y)
00105                 );
00106             }
00107             break; // Encontramos la entidad, pero no tiene TransformComponent
00108         }
00109     }
00110     // No se encontró la entidad o no tiene TransformComponent
00111     return std::make_tuple(0, 0);
00112 }
00118 int GetAllEnemies(lua_State* L) {
00119     auto& registry = Game::GetInstance().registry;
00120     auto entities = registry->GetEntitiesFromSystem<EnemySystem>();
00121 
00122     lua_newtable(L);
00123     int index = 1;
00124     for (const auto& entity : entities) {
00125         lua_pushinteger(L, static_cast<lua_Integer>(entity.GetId()));
00126         lua_rawseti(L, -2, index);
00127         index++;
00128     }
00129 
00130     return 1;
00131 }
00136 void AttackMelee(Entity attacker) {
00137     auto& registry = Game::GetInstance().registry;
00138 
00139     if (!attacker.HasComponent<HealthComponent>()) return;
00140 
00141     Entity playerEntity(-1);
00142     bool foundPlayer = false;
00143 
00144     // Obtener entidades manejadas por HealthSystem
00145     auto entities = registry->GetEntitiesFromSystem<HealthSystem>();
00146 
00147     for (auto& entity : entities) {
00148         if (entity.HasComponent<HealthComponent>()) {
00149             auto& health = entity.GetComponent<HealthComponent>();
00150             if (health.isPlayer) {
00151                 playerEntity = entity;
00152                 foundPlayer = true;
00153                 break;
00154             }
00155         }
00156     }
00157 
00158     if (foundPlayer && playerEntity.HasComponent<HealthComponent>()) {
00159         auto& attackerHealth = attacker.GetComponent<HealthComponent>();
00160         auto& healthSystem = registry->GetSystem<HealthSystem>();
00161         healthSystem.ReduceHP(playerEntity, attackerHealth.damage, attacker);
00162     }
00163 }
00168 void AttackRanger(Entity attacker) {
00169     auto& registry = Game::GetInstance().registry;
```

```
00170     if (!attacker.HasComponent<HealthComponent>()) return;
00171
00172     // Obtener entidades manejadas por HealthSystem
00173     auto entities = registry->GetEntitiesFromSystem<EnemySystem>();
00174
00175     for (auto& entity : entities) { // buggeado
00176         if (entity.HasComponent<ProjectileComponent>() &&
      entity.GetComponent<SpriteComponent>().active == false) {
00177
00178             auto& health = attacker.GetComponent<HealthComponent>();
00179             float damageInterval = health.attackTimeout;
00180             // Obtener tiempo actual
00181             auto now = std::chrono::steady_clock::now();
00182
00183             // Calcular tiempo transcurrido desde el último daño recibido por esta entidad específica
00184             auto elapsed = std::chrono::duration_cast<std::chrono::milliseconds>(now -
      attacker.GetComponent<HealthComponent>().attackTimeoutDuration).count();
00185             int intervalMs = static_cast<int>(damageInterval * 1000); //damageInterval es tiempos de
      disparos
00186
00187             // Si no ha pasado suficiente tiempo, no aplicar daño
00188             if (elapsed < intervalMs) {
00189                 return;
00190             }
00191             attacker.GetComponent<HealthComponent>().attackTimeoutDuration = now;
00192             auto& ene = registry->GetSystem<EnemySystem>();
00193             std::tuple<int, int> playerPos = GetPlayerPosition();
00194             glm::vec2 enemyPosition = attacker.GetComponent<TransformComponent>().position;
00195
00196             // Convertir posición del jugador a vec2
00197             glm::vec2 playerPosition = glm::vec2(std::get<0>(playerPos), std::get<1>(playerPos));
00198
00199             // Calcular el vector dirección del enemigo hacia el jugador
00200             glm::vec2 direction = playerPosition - enemyPosition;
00201
00202             // Normalizar la dirección para obtener el vector unitario
00203             glm::vec2 normalizedDirection = glm::normalize(direction);
00204
00205             // Definir la velocidad del proyectil (ajusta este valor según tu juego)
00206             float projectileSpeed = 100.0f; // pixels per second, por ejemplo
00207
00208             // Calcular la velocidad final
00209             glm::vec2 velocity = normalizedDirection * projectileSpeed;
00210
00211             // Calcular la rotación en radianes usando atan2
00212             double arrowRotation = atan2(direction.y, direction.x);
00213
00214             // Si necesitas la rotación en grados en lugar de radianes:
00215             double arrowRotationDegrees = glm::degrees(arrowRotation);
00216             ene.CreateEnemyProjectile(registry, velocity, enemyPosition, arrowRotationDegrees,
      attacker.GetComponent<HealthComponent>().damage);
00217         }
00218     }
00219 }
00224 void SetLevel(int level) {
00225     Game::GetInstance().currentLevel = level;
00226 }
00232 void CurrentDrawIndex(Entity entity, int index) {
00233
00234     if (Game::GetInstance().drawIndex == -1) {
00235         Game::GetInstance().drawIndex = index;
00236         auto& registry = Game::GetInstance().registry;
00237         for (auto entity : registry->GetSystem<RenderTextSystem>().GetSystemEntities()) {
00238             if (entity.HasComponent<DamageChargeComponent>() ||
      entity.HasComponent<SprintChargeComponent>() || entity.HasComponent<SlowChargeComponent>() ) {
00239                 if (entity.HasComponent<DamageChargeComponent>() && index == 0) {
00240                     entity.GetComponent<SpriteComponent>().active = true;
00241                 } else if (entity.HasComponent<SprintChargeComponent>() && index == 1) {
00242                     entity.GetComponent<SpriteComponent>().active = true;
00243                 } else if (entity.HasComponent<SlowChargeComponent>() && index == 2) {
00244                     entity.GetComponent<SpriteComponent>().active = true;
00245                 }
00246
00247             }
00248         }
00249     } else {
00250         int prevIndex = Game::GetInstance().drawIndex;
00251         Game::GetInstance().drawIndex = index;
00252         auto& registry = Game::GetInstance().registry;
00253
00254         for (auto entity : registry->GetSystem<RenderTextSystem>().GetSystemEntities()) {
00255             if (entity.HasComponent<DamageChargeComponent>() ||
      entity.HasComponent<SprintChargeComponent>() || entity.HasComponent<SlowChargeComponent>() ) {
00256                 if (entity.HasComponent<DamageChargeComponent>() && index == 0) {
00257
00258                     entity.GetComponent<SpriteComponent>().active = true;
00259                 } else if (entity.HasComponent<SprintChargeComponent>() && index == 1) {
```

```
00260                     entity.GetComponent<SpriteComponent>().active = true;
00261                 } else if (entity.HasComponent<SlowChargeComponent>() && index == 2) {
00262                     entity.GetComponent<SpriteComponent>().active = true;
00263                 } else if (entity.HasComponent<DamageChargeComponent>() && prevIndex == 0) {
00264 
00265                     entity.GetComponent<SpriteComponent>().active = false;
00266                 } else if (entity.HasComponent<SprintChargeComponent>() && prevIndex == 1) {
00267                     entity.GetComponent<SpriteComponent>().active = false;
00268                 } else if (entity.HasComponent<SlowChargeComponent>() && prevIndex == 2) {
00269                     entity.GetComponent<SpriteComponent>().active = false;
00270                 }
00271 
00272             }
00273         }
00274     }
00275 
00276 
00277 }
00278 
00279 
00280 
00288 void SetVelocity(Entity entity, float x, float y) {
00289     auto& rigidBody = entity.GetComponent<RigidBodyComponent>();
00290     rigidBody.velocity.x = x;
00291     rigidBody.velocity.y = y;
00292 }
00297 void GoToScene(const std::string& sceneName) {
00298     Game::GetInstance().sceneManager->SetNextScene(sceneName);
00299     Game::GetInstance().sceneManager->StopScene();
00300 }
00308 void PushDrawPoint(Entity entity, int index, int x, int y) {
00309     auto& draw = entity.GetComponent<DrawableComponent>();
00310     if (index >= 0 && index < (int)draw.colorPoints.size() &&
    Game::GetInstance().registry->GetSystem<ChargeManageSystem>().HasSufficientCharge(index) == true) {
00311         // TODO: aunque no se dibujen igual se cuentan (hacer chequeo de la posicion (menor a 70/75 en
    Y))
00312         draw.colorPoints[index].emplace_back(glm::vec2(x, y), std::chrono::steady_clock::now());
00313         Game::GetInstance().registry->GetSystem<ChargeManageSystem>().ConsumeChargeForDrawing(index);
00314     }
00315 }
00316 
00317 #endif // LUABINDING_HPP
```

2.3 AnimationComponent.hpp

```
00001 #ifndef ANIMATIONCOMPONENT_HPP
00002 #define ANIMATIONCOMPONENT_HPP
00003 
00004 #include <glm/glm.hpp>
00005 #include <SDL2/SDL.h>
00006 
00011 struct AnimationComponent {
00013     int numFrames;
00014 
00016     int currentFrame;
00017 
00019     int frameSpeedRate;
00020 
00022     bool isLoop;
00023 
00025     int startTime;
00026 
00033     AnimationComponent(int numFrames = 1, int frameSpeedRate = 1, bool isLoop = true) {
00034         this->numFrames = numFrames;
00035         this->currentFrame = 1;
00036         this->frameSpeedRate = frameSpeedRate;
00037         this->isLoop = isLoop;
00038         this->startTime = SDL_GetTicks();
00039     }
00040 };
00041 
00042 #endif
```

2.4 CircleColliderComponent.hpp

```
00001 #ifndef CIRCLECOLLIDERCOMPONENT_HPP
00002 #define CIRCLECOLLIDERCOMPONENT_HPP
00003 
00008 struct CircleColliderComponent {
00010     int radius;
00011 
```

```
00013     int width;
00014
00016     int height;
00017
00024     CircleColliderComponent(int radius = 0, int width = 0, int height = 0) {
00025         this->width = width;
00026         this->height = height;
00027         this->radius = radius;
00028     }
00029 };
00030
00031 #endif
```

2.5 ClickableComponent.hpp

```
00001 #ifndef CLICKABLECOMPONENT_HPP
00002 #define CLICKABLECOMPONENT_HPP
00003
00008 struct ClickableComponent {
00010     bool isClicked;
00011
00015     ClickableComponent() {
00016         isClicked = false;
00017     }
00018 };
00019
00020 #endif // CLICKABLECOMPONENT_HPP
```

2.6 DamageChargeComponent.hpp

```
00001 #ifndef DAMAGECHARGECOMPONENT_HPP
00002 #define DAMAGECHARGECOMPONENT_HPP
00003
00004 #include <string>
00005
00010 struct DamageChargeComponent {
00012     int totalCharge;
00013
00015     int currentCharge;
00016
00018     std::string chargeDisplay;
00019
00025     DamageChargeComponent(int total = 100, int initialCharge = 100) {
00026         totalCharge = total;
00027         currentCharge = initialCharge;
00028         updateChargeDisplay();
00029     }
00030
00034     void updateChargeDisplay() {
00035         chargeDisplay = std::to_string(currentCharge) + "/" + std::to_string(totalCharge);
00036     }
00037
00041     void Recharge() {
00042         currentCharge = totalCharge;
00043         updateChargeDisplay();
00044     }
00045
00050     void Charge(int amount) {
00051         currentCharge += amount;
00052         if (currentCharge > totalCharge) {
00053             currentCharge = totalCharge;
00054         }
00055         updateChargeDisplay();
00056     }
00057
00062     void Discharge(int amount) {
00063         currentCharge -= amount;
00064         if (currentCharge < 0) {
00065             currentCharge = 0;
00066         }
00067         updateChargeDisplay();
00068     }
00069
00074     float GetPercentage() const {
00075         if (totalCharge == 0) return 0.0f;
00076         return (static_cast<float>(currentCharge) / static_cast<float>(totalCharge)) * 100.0f;
00077     }
00078
00083     bool IsFullyCharged() const {
00084         return currentCharge == totalCharge;
00085     }
```

```
00086
00091     bool IsEmpty() const {
00092         return currentCharge == 0;
00093     }
00094 };
00095
00096 #endif // DAMAGECHARGECOMPONENT_HPP
```

2.7 DrawableComponent.hpp

```
00001 #ifndef DRAWABLE_COMPONENT_HPP
00002 #define DRAWABLE_COMPONENT_HPP
00003
00004 #include <SDL2/SDL.h>
00005 #include <vector>
00006 #include <glm/vec2.hpp>
00007 #include <chrono>
00008
00013 struct DrawableComponent {
00015     SDL_Color color;
00016
00018     std::vector<std::vector<std::pair<glm::vec2, std::chrono::steady_clock::time_point>> colorPoints;
00019
00023     DrawableComponent() {
00024         color = {255, 255, 255, 255}; // White
00025         colorPoints.resize(3); // Red, blue, green
00026     }
00027
00032     DrawableComponent(SDL_Color col) {
00033         color = col;
00034         colorPoints.resize(3); // Red, blue, green
00035     }
00036 };
00037
00038 #endif // DRAWABLE_COMPONENT_HPP
```

2.8 EffectReceiverComponent.hpp

```
00001 #ifndef EFFECTRECEIVERCOMPONENT_HPP
00002 #define EFFECTRECEIVERCOMPONENT_HPP
00003
00008 struct EffectReceiverComponent {
00010     bool takingDamage = false;
00011
00013     bool slowed = false;
00014
00016     bool speedBoosted = false;
00017
00021     EffectReceiverComponent() = default;
00022 };
00023
00024 #endif // EFFECTRECEIVERCOMPONENT_HPP
```

2.9 EnemyComponent.hpp

```
00001 #ifndef ENEMYCOMPONENT_HPP
00002 #define ENEMYCOMPONENT_HPP
00003
00008 struct EnemyComponent {
00010     int amountToSpawn;
00011
00013     int spawnerId;
00014
00016     int totalAmount;
00017
00019     int points;
00020
00028     EnemyComponent(int amountToSpawn = 0, int spawnerId = 0, int totalAmount = 0, int points = 0)
00029         : amountToSpawn(amountToSpawn), spawnerId(spawnerId), totalAmount(totalAmount), points(points)
      {}
00030 };
00031
00032 #endif // ENEMYCOMPONENT_HPP
```

2.10 HealthComponent.hpp

```
00001 #ifndef HEALTHCOMPONENT_HPP
00002 #define HEALTHCOMPONENT_HPP
00003
00004 #include <chrono>
00005
00010 struct HealthComponent {
00012     int health;
00013
00015     int maxHealth;
00016
00018     bool isPlayer;
00019
00021     int damage;
00022
00024     float attackTimeout;
00025
00027     std::chrono::steady_clock::time_point attackTimeoutDuration;
00028
00030     std::chrono::steady_clock::time_point lastDamageReceived;
00031
00040     HealthComponent(int health = 100, int maxHealth = 100, bool isPlayer = false, int damage = 0,
      float attackTimeout = 0.0f)
00041     {
00042         this->health = health;
00043         this->maxHealth = maxHealth;
00044         this->isPlayer = isPlayer;
00045         this->damage = damage;
00046         this->attackTimeout = attackTimeout;
00047         // Initialize timestamps to allow immediate damage or attack
00048         this->attackTimeoutDuration = std::chrono::steady_clock::now() - std::chrono::seconds(1);
00049         this->lastDamageReceived = std::chrono::steady_clock::now() - std::chrono::seconds(-1);
00050     }
00051 };
00052
00053 #endif
```

2.11 IdentifierComponent.hpp

```
00001 #ifndef IDENTIFIERCOMPONENT_HPP
00002 #define IDENTIFIERCOMPONENT_HPP
00003
00004 #include <string>
00005
00010 struct IdentifierComponent {
00012     int id;
00013
00015     std::string name;
00016
00020     IdentifierComponent() = default;
00021
00027     IdentifierComponent(int id, const std::string& name)
00028         : id(id), name(name) {}
00029 };
00030
00031 #endif
```

2.12 ProjectileComponent.hpp

```
00001 #ifndef PROJECTILECOMPONENT_HPP
00002 #define PROJECTILECOMPONENT_HPP
00003
00004 #include <../libs/glm/glm.hpp>
00005
00010 struct ProjectileComponent {
00012     glm::vec2 velocity;
00013
00015     glm::vec2 position;
00016
00018     glm::vec2 scale;
00019
00021     double rotation;
00022
00024     bool hasHit = false;
00025
00027     int damage;
00028
00036     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)) {
00037         this->velocity = velocity;
```

```
00038         this->position = position;
00039         this->scale = scale;
00040         this->rotation = rotation;
00041     }
00042 };
00043
00044 #endif
```

2.13 RigidBodyComponent.hpp

```
00001 #ifndef RIGIDBODYCOMPONENT_HPP
00002 #define RIGIDBODYCOMPONENT_HPP
00003
00004 #include <glm/glm.hpp>
00005
00010 struct RigidBodyComponent {
00012     glm::vec2 velocity;
00013
00018     RigidBodyComponent(const glm::vec2& velocity = glm::vec2(0.0f, 0.0f)) {
00019         this->velocity = velocity;
00020     }
00021 };
00022
00023 #endif
```

2.14 ScriptComponent.hpp

```
00001 #ifndef SCRIPT_COMPONENT_HPP
00002 #define SCRIPT_COMPONENT_HPP
00003
00004 #include <sol/sol.hpp>
00005
00010 struct ScriptComponent {
00012     sol::function update;
00013
00015     sol::function onClick;
00016
00022     ScriptComponent(sol::function update = sol::lua_nil,
00023                     sol::function onClick = sol::lua_nil) {
00024         this->update = update;
00025         this->onClick = onClick;
00026     }
00027 };
00028
00029 #endif // SCRIPT_COMPONENT_HPP
```

2.15 SlowChargeComponent.hpp

```
00001 #ifndef SLOWCHARGECOMPONENT_HPP
00002 #define SLOWCHARGECOMPONENT_HPP
00003
00004 #include <string>
00005
00010 struct SlowChargeComponent {
00012     int totalCharge;
00013
00015     int currentCharge;
00016
00018     std::string chargeDisplay;
00019
00025     SlowChargeComponent(int total = 100, int initialCharge = 100) {
00026         totalCharge = total;
00027         currentCharge = initialCharge;
00028         updateChargeDisplay();
00029     }
00030
00034     void updateChargeDisplay() {
00035         chargeDisplay = std::to_string(currentCharge) + "/" + std::to_string(totalCharge);
00036     }
00037
00041     void Recharge() {
00042         currentCharge = totalCharge;
00043         updateChargeDisplay();
00044     }
00045
00050     void Charge(int amount) {
00051         currentCharge += amount;
00052         if (currentCharge > totalCharge) {
```

```
00053             currentCharge = totalCharge;
00054         }
00055         updateChargeDisplay();
00056     }
00057
00062     void Discharge(int amount) {
00063         currentCharge -= amount;
00064         if (currentCharge < 0) {
00065             currentCharge = 0;
00066         }
00067         updateChargeDisplay();
00068     }
00069
00074     float GetPercentage() const {
00075         if (totalCharge == 0) return 0.0f;
00076         return (static_cast<float>(currentCharge) / static_cast<float>(totalCharge)) * 100.0f;
00077     }
00078
00083     bool IsFullyCharged() const {
00084         return currentCharge == totalCharge;
00085     }
00086
00091     bool IsEmpty() const {
00092         return currentCharge == 0;
00093     }
00094 };
00095
00096 #endif // SLOWCHARGECOMPONENT_HPP
```

2.16 SoundComponent.hpp

```
00001 #ifndef SOUNDCOMPONENT_HPP
00002 #define SOUNDCOMPONENT_HPP
00003
00004 #include <string>
00005
00010 struct SoundComponent {
00012     std::string soundId;
00013
00015     int volume;
00016
00018     int loops;
00019
00021     bool isPlaying;
00022
00024     bool active;
00025
00027     bool autoPlay;
00028
00037     SoundComponent(const std::string& soundId = "none", int volume = 128, int loops = -1, bool
      autoPlay = true, bool active = true) {
00038         this->soundId = soundId;
00039         this->volume = volume;
00040         this->loops = loops;
00041         this->isPlaying = false;
00042         this->active = active;
00043         this->autoPlay = autoPlay;
00044     }
00045 };
00046
00047 #endif
```

2.17 SprintChargeComponent.hpp

```
00001 #ifndef SPRINTCHARGECOMPONENT_HPP
00002 #define SPRINTCHARGECOMPONENT_HPP
00003 #include <string>
00004
00011 struct SprintChargeComponent {
00012     int totalCharge;
00013     int currentCharge;
00014     std::string chargeDisplay;
00015
00022     SprintChargeComponent(int total = 100, int initialCharge = 100) {
00023         totalCharge = total;
00024         currentCharge = initialCharge;
00025         updateChargeDisplay();
00026     }
00027
00031     void updateChargeDisplay() {
00032         chargeDisplay = std::to_string(currentCharge) + "/" + std::to_string(totalCharge);
```

```
00033     }
00034
00038     void Recharge() {
00039         currentCharge = totalCharge;
00040         updateChargeDisplay();
00041     }
00042
00048     void Charge(int amount) {
00049         currentCharge += amount;
00050         if (currentCharge > totalCharge) {
00051             currentCharge = totalCharge;
00052         }
00053         updateChargeDisplay();
00054     }
00055
00061     void Discharge(int amount) {
00062         currentCharge -= amount;
00063         if (currentCharge < 0) {
00064             currentCharge = 0;
00065         }
00066         updateChargeDisplay();
00067     }
00068
00074     float GetPercentage() const {
00075         if (totalCharge == 0) return 0.0f;
00076         return (static_cast<float>(currentCharge) / static_cast<float>(totalCharge)) * 100.0f;
00077     }
00078
00085     bool IsFullyCharged() const {
00086         return currentCharge == totalCharge;
00087     }
00088
00095     bool IsEmpty() const {
00096         return currentCharge == 0;
00097     }
00098 };
00099 #endif // SPRINTCHARGECOMPONENT_HPP
```

2.18 SpriteComponent.hpp

```
00001 #ifndef SPRITECOMPONENT_HPP
00002 #define SPRITECOMPONENT_HPP
00003 #include <SDL2/SDL.h>
00004 #include <string>
00005
00011 struct SpriteComponent {
00012     SDL_Rect srcRect;
00013     std::string textureId;
00014     int width;
00015     int height;
00016     SDL_RendererFlip flip;
00017     bool active;
00018
00029     SpriteComponent(const std::string& textureId = "none",
00030                     int width = 0,
00031                     int height = 0,
00032                     int srcRectX = 0,
00033                     int srcRectY = 0,
00034                     bool active = true)
00035     {
00036         this->textureId = textureId;
00037         this->width = width;
00038         this->height = height;
00039         this->srcRect = { srcRectX, srcRectY, width, height };
00040         this->flip = SDL_FLIP_NONE;
00041         this->active = active;
00042     }
00043 };
00044 #endif // SPRITECOMPONENT_HPP
```

2.19 TextComponent.hpp

```
00001 #ifndef TEXT_COMPONENT_HPP
00002 #define TEXT_COMPONENT_HPP
00003 #include <SDL2/SDL.h>
00004 #include <SDL2/SDL_ttf.h>
00005 #include <string>
00006
00012 struct TextComponent {
00013     std::string text;
00014     std::string fontId;
```

```
00015     SDL_Color color;
00016     int width;
00017     int height;
00018 
00029     TextComponent(const std::string& text = "",
00030                   const std::string& fontId = "",
00031                   u_char r = 0,
00032                   u_char g = 0,
00033                   u_char b = 0,
00034                   u_char a = 0) {
00035         this->text = text;
00036         this->fontId = fontId;
00037         this->color.r = r;
00038         this->color.g = g;
00039         this->color.b = b;
00040         this->color.a = a;
00041         this->width = 0;
00042         this->height = 0;
00043     }
00044 };
00045 
00046 #endif // TEXT_COMPONENT_HPP
```

2.20 TransformComponent.hpp

```
00001 #ifndef TRANSFORM_COMPONENT_HPP
00002 #define TRANSFORM_COMPONENT_HPP
00003 #include <../libs/glm/glm.hpp>
00004 
00010 struct TransformComponent {
00011     glm::vec2 position;
00012     glm::vec2 scale;
00013     double rotation;
00014 
00022     TransformComponent(glm::vec2 position = glm::vec2(0.0,0.0),
00023                        glm::vec2 scale = glm::vec2(1.0,1.0),
00024                        double rotation = 0.0) {
00025         this->position = position;
00026         this->scale = scale;
00027         this->rotation = rotation;
00028     }
00029 };
00030 
00031 #endif // TRANSFORM_COMPONENT_HPP
```

2.21 ControllerManager.hpp

```
00001 #ifndef CONTROLLER_MANAGER_HPP
00002 #define CONTROLLER_MANAGER_HPP
00003 
00004 #include <SDL2/SDL.h>
00005 #include <map>
00006 #include <string>
00007 #include <tuple>
00008 
00015 class ControllerManager {
00016 private:
00017     std::map<std::string, int> actionKeyName;
00018     std::map<int, bool> keyDown;
00019 
00020     std::map<std::string, int> mouseButtonName;
00021     std::map<int, bool> mouseButtonDown;
00022 
00023     int mousePosX;
00024     int mousePosY;
00025 
00026 public:
00030     ControllerManager();
00031 
00035     ~ControllerManager();
00036 
00040     void Clear();
00041 
00042     // Keyboard related methods
00043 
00049     void AddActionKey(const std::string& action, int keyCode);
00050 
00055     void KeyDown(int keyCode);
00056 
00061     void KeyUp(int keyCode);
00062 
```

```
00069     bool IsActionActivated(const std::string& action);
00070
00071     // Mouse related methods
00072
00078     void AddMouseButton(const std::string& action, int buttonCode);
00079
00084     void MouseButtonDown(int buttonCode);
00085
00090     void MouseButtonUp(int buttonCode);
00091
00098     bool IsMouseButtonDown(const std::string& name);
00099
00105     void SetMousePosition(int x, int y);
00106
00111     std::tuple<int, int> GetMousePosition();
00112 };
00113
00114 #endif // CONTROLLER_MANAGER_HPP
```

2.22 ECS.hpp

```
00001 #ifndef ECS_HPP
00002 #define ECS_HPP
00003
00004 #include <cstddef>
00005 #include <bitset>
00006 #include <memory>
00007 #include <vector>
00008 #include <set>
00009 #include <deque>
00010 #include <typeindex>
00011 #include <unordered_map>
00012 #include <iostream>
00013 #include "../Utils/Pool.hpp"
00014
00016 const unsigned int MAX_COMPONENTS = 64;
00017
00019 typedef std::bitset<MAX_COMPONENTS> Signature;
00020
00025 struct IComponent {
00026 protected:
00028     static int nextId;
00029 };
00030
00036 template <typename TComponent>
00037 class Component : public IComponent {
00038 public:
00043     static int GetId() {
00044         static int id = nextId++;
00045         return id;
00046     }
00047 };
00048
00053 class Entity {
00054 private:
00056     int id;
00057
00058 public:
00063     Entity(int id) : id(id) {}
00064
00069     int GetId() const;
00070
00074     void Kill();
00075
00081     bool operator==(const Entity& other) const { return id == other.id; }
00082
00088     bool operator!=(const Entity& other) const { return id != other.id; }
00089
00095     bool operator>(const Entity& other) const { return id > other.id; }
00096
00102     bool operator<(const Entity& other) const { return id < other.id; }
00103
00110     template <typename TComponent, typename... TArgs>
00111     void AddComponent(TArgs&&... args);
00112
00117     template <typename TComponent>
00118     void RemoveComponent();
00119
00125     template <typename TComponent>
00126     bool HasComponent() const;
00127
00133     template <typename TComponent>
00134     TComponent& GetComponent() const;
00135
```

```
00137     class Registry* registry;
00138 };
00139
00144 class System {
00145 private:
00147     Signature componentSignature;
00148
00150     std::vector<Entity> entities;
00151
00152 public:
00156     System() = default;
00157
00161     ~System() = default;
00162
00167     void AddEntityToSystem(Entity entity);
00168
00173     void RemoveEntityFromSystem(Entity entity);
00174
00179     std::vector<Entity> GetSystemEntities() const;
00180
00185     const Signature& GetComponentSignature() const;
00186
00191     template <typename TComponent>
00192     void RequireComponent();
00193 };
00194
00199 class Registry {
00200 private:
00202     int numEntity = 0;
00203
00205     std::vector<std::shared_ptr<IPool>> componentsPools;
00206
00208     std::vector<Signature> entityComponentSignatures;
00209
00211     std::unordered_map<std::type_index, std::shared_ptr<System>> systems;
00212
00214     std::set<Entity> entitiesToBeAdded;
00215
00217     std::set<Entity> entitiesToBeKilled;
00218
00220     std::deque<int> freeIds;
00221
00222 public:
00226     Registry();
00227
00231     ~Registry();
00232
00236     void Update();
00237
00242     Entity CreateEntity();
00243
00248     void KillEntity(Entity entity);
00249
00257     template <typename TComponent, typename... TArgs>
00258     void AddComponent(Entity entity, TArgs&&... args);
00259
00265     template <typename TComponent>
00266     void RemoveComponent(Entity entity);
00267
00274     template <typename TComponent>
00275     bool HasComponent(Entity entity) const;
00276
00283     template <typename TComponent>
00284     TComponent& GetComponent(Entity entity) const;
00285
00292     template <typename TSystem, typename... TArgs>
00293     void AddSystem(TArgs&&... args);
00294
00299     template <typename TSystem>
00300     void RemoveSystem();
00301
00307     template <typename TSystem>
00308     bool HasSystem() const;
00309
00315     template <typename TSystem>
00316     TSystem& GetSystem() const;
00317
00322     void AddEntityToSystems(Entity entity);
00323
00328     void RemoveEntityFromSystems(Entity entity);
00329
00335     template<typename T>
00336     std::vector<Entity> GetEntitiesFromSystem();
00337
00341     void ClearAllEntities();
00342 };
00343
```

```
00348 template <typename TComponent>
00349 void System::RequireComponent() {
00350     const int componentId = Component<TComponent>::GetId();
00351     componentSignature.set(componentId);
00352 }
00353
00359 template<typename T>
00360 std::vector<Entity> Registry::GetEntitiesFromSystem() {
00361     if (!HasSystem<T>()) return {};
00362
00363     auto& systemDerived = GetSystem<T>();
00364     return systemDerived.GetSystemEntities();
00365 }
00366
00374 template <typename TComponent, typename... TArgs>
00375 void Registry::AddComponent(Entity entity, TArgs&&... args) {
00376     const int componentId = Component<TComponent>::GetId();
00377     const int entityId = entity.GetId();
00378
00379     if (static_cast<long unsigned int>(componentId) >= componentsPools.size()) {
00380         componentsPools.resize(componentId + 10, nullptr);
00381     }
00382
00383     if (!componentsPools[componentId]) {
00384         std::shared_ptr<Pool<TComponent>> newComponentPool = std::make_shared<Pool<TComponent>>();
00385         componentsPools[componentId] = newComponentPool;
00386     }
00387
00388     std::shared_ptr<Pool<TComponent>> componentPool
00389         = std::static_pointer_cast<Pool<TComponent>>(componentsPools[componentId]);
00390
00391     if (entityId >= componentPool->GetSize()) {
00392         componentPool->Resize(numEntity + 100);
00393     }
00394
00395     TComponent newComponent(std::forward<TArgs>(args)...);
00396     componentPool->Set(entityId, newComponent);
00397     entityComponentSignatures[entityId].set(componentId);
00398 }
00399
00405 template <typename TComponent>
00406 void Registry::RemoveComponent(Entity entity) {
00407     const int componentId = Component<TComponent>::GetId();
00408     const int entityId = entity.GetId();
00409
00410     entityComponentSignatures[entityId].set(componentId, false);
00411 }
00412
00419 template <typename TComponent>
00420 bool Registry::HasComponent(Entity entity) const {
00421     const int componentId = Component<TComponent>::GetId();
00422     const int entityId = entity.GetId();
00423
00424     return entityComponentSignatures[entityId].test(componentId);
00425 }
00426
00433 template <typename TComponent>
00434 TComponent& Registry::GetComponent(Entity entity) const {
00435     const int componentId = Component<TComponent>::GetId();
00436     const int entityId = entity.GetId();
00437
00438     auto componentPool =
00439         std::static_pointer_cast<Pool<TComponent>>(componentsPools[componentId]);
00440     return componentPool->Get(entityId);
00441 }
00442
00449 template <typename TSystem, typename... TArgs>
00450 void Registry::AddSystem(TArgs&&... args) {
00451     std::shared_ptr<TSystem> newSystem = std::make_shared<TSystem>(std::forward<TArgs>(args)...);
00452     systems.insert(std::make_pair(std::type_index(typeid(TSystem)), newSystem));
00453 }
00454
00459 template <typename TSystem>
00460 void Registry::RemoveSystem() {
00461     auto system = systems.find(std::type_index(typeid(TSystem)));
00462     systems.erase(system);
00463 }
00464
00470 template <typename TSystem>
00471 bool Registry::HasSystem() const {
00472     return systems.find(std::type_index(typeid(TSystem))) != systems.end();
00473 }
00474
00480 template <typename TSystem>
00481 TSystem& Registry::GetSystem() const {
00482     auto system = systems.find(std::type_index(typeid(TSystem)));
00483     return *(std::static_pointer_cast<TSystem>(system->second));
```

```
00484 }
00485
00492 template <typename TComponent, typename... TArgs>
00493 void Entity::AddComponent(TArgs&&... args) {
00494     registry->AddComponent<TComponent>(*this, std::forward<TArgs>(args)...);
00495 }
00496
00501 template <typename TComponent>
00502 void Entity::RemoveComponent() {
00503     registry->RemoveComponent<TComponent>(*this);
00504 }
00505
00511 template <typename TComponent>
00512 bool Entity::HasComponent() const {
00513     return registry->HasComponent<TComponent>(*this);
00514 }
00515
00521 template <typename TComponent>
00522 TComponent& Entity::GetComponent() const {
00523     return registry->GetComponent<TComponent>(*this);
00524 }
00525
00526 #endif
```

2.23 Event.hpp

```
00001 #ifndef EVENT_HPP
00002 #define EVENT_HPP
00003
00011 class Event {
00012 public:
00016     Event() = default;
00017
00021     virtual ~Event() = default;
00022
00026     //virtual void execute() = 0;
00027
00031     //virtual void undo() = 0;
00032
00036     //virtual void redo() = 0;
00037
00038 };
00039
00040 #endif // EVENT_HPP
```

2.24 EventManager.hpp

```
00001 #ifndef EVENTMANAGER_HPP
00002 #define EVENTMANAGER_HPP
00003
00004 #include "Event.hpp"
00005 #include <list>
00006 #include <memory>
00007 #include <functional>
00008 #include <iostream>
00009 #include <map>
00010 #include <typeindex>
00011
00017 class IEventCallback {
00018 public:
00019     virtual ~IEventCallback() = default;
00020
00025     void Excute(Event& event) {
00026         Call(event);
00027     }
00028
00029 private:
00034     virtual void Call(Event& event) = 0;
00035 };
00036
00042 template<typename TOwner, typename TEvent>
00043 class EventCallback : public IEventCallback {
00044 private:
00045     typedef void (TOwner::*CallbackFunction)(TEvent&);
00046     TOwner* ownerInstance;
00047     CallbackFunction callbackFunction;
00048
00053     virtual void Call(Event& event) override {
00054         std::invoke(callbackFunction, ownerInstance, static_cast<TEvent&>(event));
00055     }
00056
```

```
00057 public:
00063     EventCallback(TOwner* owner, CallbackFunction callback) {
00064         this->ownerInstance = owner;
00065         this->callbackFunction = callback;
00066     }
00067 };
00068
00069 typedef std::list<std::unique_ptr<IEventCallback>> HandlerList;
00070
00076 class EventManager {
00077 private:
00078     std::map<std::type_index, std::unique_ptr<HandlerList>> subscribers;
00079
00080 public:
00084     EventManager() {
00085         std::cout << "[EventManager] Se ejecuta constructor" << std::endl;
00086     };
00087
00091     ~EventManager() {
00092         std::cout << "[EventManager] Se ejecuta destructor" << std::endl;
00093     };
00094
00098     void Restart() {
00099         subscribers.clear();
00100     };
00101
00109     template<typename TOwner, typename TEvent>
00110     void SubscribeToEvent(TOwner* owner, void (TOwner::*callback)(TEvent&)) {
00111         if (!subscribers[typeid(TEvent)].get()) {
00112             subscribers[typeid(TEvent)] = std::make_unique<HandlerList>();
00113         }
00114         auto subscriber = std::make_unique<EventCallback<TOwner, TEvent>>(owner, callback);
00115         subscribers[typeid(TEvent)]->push_back(std::move(subscriber));
00116     };
00117
00124     template<typename TEvent, typename... TArgs>
00125     void EmitEvent(TArgs&&... args) {
00126         auto handlers = subscribers[typeid(TEvent)].get();
00127         if (handlers) {
00128             for (auto it = handlers->begin(); it != handlers->end(); ++it) {
00129                 auto handler = it->get();
00130                 TEvent event(std::forward<TArgs>(args)...);
00131                 handler->Excute(event);
00132             }
00133         }
00134     };
00135 };
00136
00137 #endif // EVENTMANAGER_HPP
```

2.25 ClickEvent.hpp

```
00001 #ifndef CLICKEVENT_HPP
00002 #define CLICKEVENT_HPP
00003
00004 #include "../ECS/ECS.hpp"
00005 #include "../EventManager/Event.hpp"
00006
00014 class ClickEvent : public Event {
00015 public:
00016     int buttonCode;
00017     int x;
00018     int y;
00019
00027     ClickEvent(int buttonCode = 0, int x = 0, int y = 0) {
00028         this->buttonCode = buttonCode;
00029         this->x = x;
00030         this->y = y;
00031     }
00032 };
00033
00034 #endif // CLICKEVENT_HPP
```

2.26 CollisionEvent.hpp

```
00001 #ifndef COLLISIONEVENT_HPP
00002 #define COLLISIONEVENT_HPP
00003
00004 #include "../ECS/ECS.hpp"
00005 #include "../EventManager/Event.hpp"
00006
```

```
00013 class CollisionEvent : public Event {
00014 public:
00015     Entity entityA;
00016     Entity entityB;
00017
00023     CollisionEvent(Entity entityA, Entity entityB)
00024         : entityA(entityA), entityB(entityB) {
00025     };
00026
00030     ~CollisionEvent() {
00031     };
00032 };
00033
00034 #endif // COLLISIONEVENT_HPP
```

2.27 Game.hpp

```
00001 #ifndef GAME_HPP
00002 #define GAME_HPP
00003
00004 #include <SDL2/SDL.h>
00005 #include <SDL2/SDL_image.h>
00006 #include <SDL2/SDL_ttf.h>
00007 #include <SDL2/SDL_mixer.h>
00008 #include <glm/glm.hpp>
00009 #include <iostream>
00010 #include <sol/sol.hpp>  // Sol2 al final
00011 #include <string>
00012 #include <fstream>
00013 #include <vector>
00014 #include <memory>
00015 #include "../AssetManager/AssetManager.hpp"
00016 #include "../ControllerManager/ControllerManager.hpp"
00017 #include "../EventManager/EventManager.hpp"
00018 #include "../ECS/ECS.hpp"
00019 #include "../SceneManager/SceneManager.hpp"
00020
00022 const int FPS = 30;
00023
00025 const int MILLISECS_PER_FRAME = 1000 / FPS;
00026
00031 class Game {
00032 private:
00034     SDL_Window* window = nullptr;
00035
00037     int millisecsPreviousFrame = 0;
00038
00040     bool isRunning = false;
00041
00043     int mPreviousFrame = 0;
00044
00045 public:
00047     SDL_Renderer* renderer = nullptr;
00048
00050     size_t windowWidth = 0;
00051
00053     size_t windowHeight = 0;
00054
00056     std::unique_ptr<ControllerManager> controllerManager;
00057
00059     std::unique_ptr<AssetManager> assetManager;
00060
00062     std::unique_ptr<EventManager> eventManager;
00063
00065     std::unique_ptr<Registry> registry;
00066
00068     std::unique_ptr<SceneManager> sceneManager;
00069
00071     sol::state lua;
00072
00074     int enemiesLeftToSpawn = 0;
00075
00077     int enemiesLeft = 0;
00078
00080     int totalPoints = 0;
00081
00083     int totalPointsPrev = 0;
00084
00086     bool finDelNivel = false;
00087
00089     bool win = false;
00090
00092     bool isPaused = false;
00093
```

```
00095     int drawIndex = -1;
00096
00098     int currentLevel = 0;
00099
00100 private:
00104     void Setup();
00105
00109     void RunScene();
00110
00114     void processInput();
00115
00119     void update();
00120
00124     void render();
00125
00129     void readConfig();
00130
00134     Game();
00135
00139     ~Game();
00140
00141 public:
00146     static Game& GetInstance();
00147
00151     void init();
00152
00156     void run();
00157
00161     void destroy();
00162 };
00163
00164 #endif
```

2.28 SceneLoader.hpp

```
00001 #ifndef SCENELOADER_HPP
00002 #define SCENELOADER_HPP
00003
00004 #include <SDL2/SDL.h>
00005 #include <memory>
00006 #include <sol/sol.hpp>
00007 #include <string>
00008 #include "../AssetManager/AssetManager.hpp"
00009 #include "../ControllerManager/ControllerManager.hpp"
00010 #include "../ECS/ECS.hpp"
00011
00018 class SceneLoader {
00019 private:
00026     void LoadBackground(SDL_Renderer* renderer, const sol::table& background,
      std::unique_ptr<AssetManager>& assetManager);
00027
00034     void LoadSprites(SDL_Renderer* renderer, const sol::table& sprites, std::unique_ptr<AssetManager>&
      assetManager);
00035
00041     void LoadFonts(const sol::table& fonts, std::unique_ptr<AssetManager>& assetManager);
00042
00048     void LoadSounds(const sol::table& sounds, std::unique_ptr<AssetManager>& assetManager);
00049
00055     void LoadKey(const sol::table& keys, std::unique_ptr<ControllerManager>& controllerManager);
00056
00062     void LoadButtons(const sol::table& buttons, std::unique_ptr<ControllerManager>&
      controllerManager);
00063
00070     void LoadEntities(sol::state& lua, const sol::table& entities, std::unique_ptr<Registry>&
      registry);
00071
00072 public:
00076     SceneLoader();
00077
00081     ~SceneLoader();
00082
00092     void LoadScene(const std::string& scenePath, sol::state& lua,
00093         std::unique_ptr<AssetManager>& assetManager, std::unique_ptr<ControllerManager>&
      controllerManager,
00094         std::unique_ptr<Registry>& registry, SDL_Renderer* renderer);
00095 };
00096
00097 #endif // SCENELOADER_HPP
```

2.29 SceneManager.hpp

```
00001 #ifndef SCENEMANAGER_HPP
```

```
00002 #define SCENEMANAGER_HPP
00003
00004 #include <map>
00005 #include <memory>
00006 #include <sol/sol.hpp>
00007 #include <string>
00008 #include "../SceneManager/SceneLoader.hpp"
00009
00016 class SceneManager {
00017 private:
00018     std::map<std::string, std::string> scenes;
00019     std::string nextScene;
00020     bool isSceneRunning = false;
00021     std::unique_ptr<SceneLoader> sceneLoader;
00022
00023 public:
00027     SceneManager();
00028
00032     ~SceneManager();
00033
00039     void LoadSceneFromScript(const std::string& scenePath, sol::state& lua);
00040
00044     void LoadScene();
00045
00050     std::string GetNextScene() const;
00051
00056     void SetNextScene(const std::string& nextScene);
00057
00063     bool IsSceneRunning() const;
00064
00068     void StartScene();
00069
00073     void StopScene();
00074 };
00075
00076 #endif // SCENEMANAGER_HPP
```

2.30 AnimationSystem.hpp

```
00001 #ifndef ANIMATIONSYSTEM_HPP
00002 #define ANIMATIONSYSTEM_HPP
00003
00004 #include "../ECS/ECS.hpp"
00005 #include "../Components/AnimationComponent.hpp"
00006 #include "../Components/SpriteComponent.hpp"
00007 #include <SDL2/SDL.h>
00008
00015 class AnimationSystem : public System {
00016 public:
00022     AnimationSystem() {
00023         RequireComponent<AnimationComponent>();
00024         RequireComponent<SpriteComponent>();
00025     }
00026
00036     void Update() {
00037         for (auto entity : GetSystemEntities()) {
00038             auto& animation = entity.GetComponent<AnimationComponent>();
00039             auto& sprite = entity.GetComponent<SpriteComponent>();
00040
00041             // Calculate current frame based on elapsed time and animation speed
00042             animation.currentFrame = ((SDL_GetTicks() - animation.startTime) *
00043                                       animation.frameSpeedRate / 1000) %
00044                                       animation.numFrames;
00045
00046             // Update sprite source rectangle to show current frame
00047             sprite.srcRect.x = animation.currentFrame * sprite.width;
00048         }
00049     }
00050 };
00051
00052 #endif // ANIMATIONSYSTEM_HPP
```

2.31 ChargeManageSystem.hpp

```
00001 #ifndef CHARGEMANAGESYSTEM_HPP
00002 #define CHARGEMANAGESYSTEM_HPP
00003
00004 #include "../ECS/ECS.hpp"
00005 #include "../Components/DamageChargeComponent.hpp"
00006 #include "../Components/SprintChargeComponent.hpp"
00007 #include "../Components/SlowChargeComponent.hpp"
```

```
00008 #include "../Game/Game.hpp"
00009 #include <chrono>
00010
00011 const int MINIMUM_CHARGE = 1;
00012
00022 class ChargeManageSystem : public System {
00023 private:
00024     const int NATURAL_RECHARGE_RATE = 5;
00025     const float RECHARGE_INTERVAL = 1.0f;
00026
00027     std::chrono::steady_clock::time_point lastRechargeTime;
00028
00029 public:
00035     ChargeManageSystem() {
00036         lastRechargeTime = std::chrono::steady_clock::now();
00037     }
00038
00045     void Update() {
00046         auto now = std::chrono::steady_clock::now();
00047         auto timeSinceLastRecharge = std::chrono::duration_cast<std::chrono::milliseconds>(now -
      lastRechargeTime).count() / 1000.0f;
00048
00049         // Only recharge if enough time has passed
00050         if (timeSinceLastRecharge >= RECHARGE_INTERVAL) {
00051             auto allEntities = GetSystemEntities();
00052
00053             for (auto entity : allEntities) {
00054                 // Recharge DamageCharge
00055                 if (entity.HasComponent<DamageChargeComponent>()) {
00056                     auto& charge = entity.GetComponent<DamageChargeComponent>();
00057                     if (!charge.IsFullyCharged()) {
00058                         charge.Charge(NATURAL_RECHARGE_RATE);
00059                     }
00060                 }
00061
00062                 // Recharge SprintCharge
00063                 if (entity.HasComponent<SprintChargeComponent>()) {
00064                     auto& charge = entity.GetComponent<SprintChargeComponent>();
00065                     if (!charge.IsFullyCharged()) {
00066                         charge.Charge(NATURAL_RECHARGE_RATE);
00067                     }
00068                 }
00069
00070                 // Recharge SlowCharge
00071                 if (entity.HasComponent<SlowChargeComponent>()) {
00072                     auto& charge = entity.GetComponent<SlowChargeComponent>();
00073                     if (!charge.IsFullyCharged()) {
00074                         charge.Charge(NATURAL_RECHARGE_RATE);
00075                     }
00076                 }
00077             }
00078
00079             lastRechargeTime = now;
00080         }
00081     }
00082
00089     bool HasSufficientCharge(int colorIndex) {
00090         for (auto& entity : GetSystemEntities()) {
00091             if (entity.HasComponent<DamageChargeComponent>() ||
00092                 entity.HasComponent<SprintChargeComponent>() ||
00093                 entity.HasComponent<SlowChargeComponent>()) {
00094
00095                 // Check specific charge type
00096                 if (entity.HasComponent<DamageChargeComponent>() && colorIndex == 0) {
00097                     return entity.GetComponent<DamageChargeComponent>().currentCharge >=
      MINIMUM_CHARGE;
00098                 } else if (entity.HasComponent<SprintChargeComponent>() && colorIndex == 1) {
00099                     return entity.GetComponent<SprintChargeComponent>().currentCharge >=
      MINIMUM_CHARGE;
00100                 } else if (entity.HasComponent<SlowChargeComponent>() && colorIndex == 2) {
00101                     return entity.GetComponent<SlowChargeComponent>().currentCharge >= MINIMUM_CHARGE;
00102                 }
00103             }
00104         }
00105         return false;
00106     }
00107
00114     bool ConsumeChargeForDrawing(int colorIndex) {
00115         for (auto& entity : GetSystemEntities()) {
00116             if (entity.HasComponent<DamageChargeComponent>() ||
00117                 entity.HasComponent<SprintChargeComponent>() ||
00118                 entity.HasComponent<SlowChargeComponent>()) {
00119
00120                 // Check and consume specific charge type
00121                 if (entity.HasComponent<DamageChargeComponent>() && colorIndex == 0) {
00122                     auto& charge = entity.GetComponent<DamageChargeComponent>();
00123                     if (charge.currentCharge >= MINIMUM_CHARGE) {
```

```
00124                         charge.Discharge(MINIMUM_CHARGE);
00125                         return true;
00126                     }
00127                 } else if (entity.HasComponent<SprintChargeComponent>() && colorIndex == 1) {
00128                     auto& charge = entity.GetComponent<SprintChargeComponent>();
00129                     if (charge.currentCharge >= MINIMUM_CHARGE) {
00130                         charge.Discharge(MINIMUM_CHARGE);
00131                         return true;
00132                     }
00133                 } else if (entity.HasComponent<SlowChargeComponent>() && colorIndex == 2) {
00134                     auto& charge = entity.GetComponent<SlowChargeComponent>();
00135                     if (charge.currentCharge >= MINIMUM_CHARGE) {
00136                         charge.Discharge(MINIMUM_CHARGE);
00137                         return true;
00138                     }
00139                 }
00140             }
00141         }
00142         return false; // Failed to consume charge
00143     }
00144 };
00145
00146 #endif // CHARGEMANAGESYSTEM_HPP
```

2.32 CollisionSystem.hpp

```
00001 #ifndef COLLISIONSYSTEM_HPP
00002 #define COLLISIONSYSTEM_HPP
00003
00004 #include "../ECS/ECS.hpp"
00005 #include "../Components/CircleColliderComponent.hpp"
00006 #include "../Components/TransformComponent.hpp"
00007 #include "../Components/HealthComponent.hpp"
00008 #include "../Components/ProjectileComponent.hpp"
00009 #include "../EventManager/EventManager.hpp"
00010 #include "../Events/CollisionEvent.hpp"
00011 #include <iostream>
00012 #include <memory>
00013
00020 class CollisionSystem : public System {
00021 public:
00027     CollisionSystem() {
00028         RequireComponent<CircleColliderComponent>();
00029         RequireComponent<TransformComponent>();
00030     }
00031
00036     void Update(std::unique_ptr<EventManager>& eventManager) {
00037         auto entities = GetSystemEntities();
00038
00039         // Check all entity pairs for collisions (n^2/2 checks)
00040         for (auto i = entities.begin(); i != entities.end(); ++i) {
00041             auto entityA = *i;
00042             auto transformA = entityA.GetComponent<TransformComponent>();
00043             auto colliderA = entityA.GetComponent<CircleColliderComponent>();
00044
00045             for (auto j = std::next(i); j != entities.end(); ++j) {
00046                 auto entityB = *j;
00047
00048                 auto transformB = entityB.GetComponent<TransformComponent>();
00049                 auto colliderB = entityB.GetComponent<CircleColliderComponent>();
00050
00051                 // Calculate world-space centers including scale and offset
00052                 glm::vec2 centerA = glm::vec2(
00053                     transformA.position.x + (colliderA.width / 2.0f) * transformA.scale.x,
00054                     transformA.position.y + (colliderA.height / 2.0f) * transformA.scale.y
00055                 );
00056
00057                 glm::vec2 centerB = glm::vec2(
00058                     transformB.position.x + (colliderB.width / 2.0f) * transformB.scale.x,
00059                     transformB.position.y + (colliderB.height / 2.0f) * transformB.scale.y
00060                 );
00061
00062                 // Calculate scaled radii
00063                 float aRadius = colliderA.radius * transformA.scale.x;
00064                 float bRadius = colliderB.radius * transformB.scale.x;
00065
00066                 // Check for collision
00067                 bool collision = CheckCircularCollision(aRadius, bRadius, centerA, centerB);
00068
00069                 if (collision) {
00070                     eventManager->EmitEvent<CollisionEvent>(entityA, entityB);
00071                 }
00072             }
00073         }
```

```
00074     }
00075
00076 private:
00086     bool CheckCircularCollision(float aRadius, float bRadius, const glm::vec2& aPos, const glm::vec2&
      bPos) {
00087         glm::vec2 diff = aPos - bPos;
00088         float distanceSquared = diff.x * diff.x + diff.y * diff.y;
00089         float radiusSum = aRadius + bRadius;
00090
00091         // Compare squared distances to avoid sqrt operation
00092         return (radiusSum * radiusSum) >= distanceSquared;
00093     }
00094 };
00095
00096 #endif // COLLISIONSYSTEM_HPP
```

2.33 DamageSystem.hpp

```
00001 #ifndef DAMAGESYSTEM_HPP
00002 #define DAMAGESYSTEM_HPP
00003
00004 #include <memory>
00005 #include "../Components/CircleColliderComponent.hpp"
00006 #include "../Components/HealthComponent.hpp"
00007 #include "../Components/ProjectileComponent.hpp"
00008 #include "../ECS/ECS.hpp"
00009 #include "../EventManager/EventManager.hpp"
00010 #include "../Events/CollisionEvent.hpp"
00011
00020 class DamageSystem : public System {
00021     public:
00027     DamageSystem() {
00028         RequireComponent<CircleColliderComponent>();
00029     }
00034     void SubscribeToCollisionEvent(std::unique_ptr<EventManager>& eventManager) {
00035         eventManager->SubscribeToEvent<DamageSystem, CollisionEvent>(this,
      &DamageSystem::OnCollision);
00036     }
00037
00042     void OnCollision(CollisionEvent& event) {
00043         if (event.entityA.HasComponent<HealthComponent>() &&
      event.entityB.HasComponent<ProjectileComponent>()) {
00044             auto& health = event.entityA.GetComponent<HealthComponent>();
00045             auto& arrow = event.entityB.GetComponent<HealthComponent>();
00046
00047
00048
00049             if (health.isPlayer) {
00050                 health.health -= arrow.damage;
00051                 if (health.health <= 0) {
00052                     health.health = 0;
00053                     Game::GetInstance().finDelNivel = true;
00054                     Game::GetInstance().win = false;
00055                 }
00056                 event.entityB.Kill();
00057             }
00058         } else if (event.entityB.HasComponent<HealthComponent>() &&
      event.entityA.HasComponent<ProjectileComponent>()) {
00059             auto& health = event.entityB.GetComponent<HealthComponent>();
00060             auto& arrow = event.entityA.GetComponent<HealthComponent>();
00061
00062             if (health.isPlayer) {
00063                 health.health -= arrow.damage;
00064                 if (health.health <= 0) {
00065                     health.health = 0;
00066                     Game::GetInstance().finDelNivel = true;
00067                     Game::GetInstance().win = false;
00068                 }
00069                 event.entityA.Kill();
00070             }
00071         }
00072     }
00073
00074
00075
00076 };
00077
00078 #endif
```

2.34 DrawingEffectSystem.hpp

```
00001 #ifndef DRAWINGEFFECTSYSTEM_HPP
```

```
00002 #define DRAWINGEFFECTSYSTEM_HPP
00003
00004 #include "../ECS/ECS.hpp"
00005 #include "../Components/DrawableComponent.hpp"
00006 #include "../Components/EffectReceiverComponent.hpp"
00007 #include "../Components/TransformComponent.hpp"
00008 #include "../Components/CircleColliderComponent.hpp"
00009 #include "../Components/EnemyComponent.hpp"
00010 #include "../Systems/CollisionSystem.hpp"
00011 #include <glm/vec2.hpp>
00012 #include "../Game/Game.hpp"
00013
00023 class DrawingEffectSystem : public System {
00024 private:
00025     const int EFFECT_RADIUS = 15; // Radio de detección aumentado para mejor cobertura
00026     const float DAMAGE_INTERVAL = 1.0f; // Intervalo de daño en segundos
00027
00028 public:
00034     DrawingEffectSystem() {
00035         RequireComponent<DrawableComponent>();
00036     }
00042     void Update() {
00043         for (auto drawingEntity : GetSystemEntities()) {
00044             auto& drawable = drawingEntity.GetComponent<DrawableComponent>();
00045
00046             if (!drawable.colorPoints.empty() && !drawable.colorPoints[0].empty()) {
00047                 ProcessDamageEffect(drawable.colorPoints[0]);
00048             }
00049
00050             for (size_t colorIndex = 1; colorIndex < drawable.colorPoints.size(); ++colorIndex) {
00051                 ProcessOtherEffects(drawable.colorPoints[colorIndex], colorIndex);
00052             }
00053         }
00054     }
00055
00056 private:
00061     void ProcessDamageEffect(const std::vector<std::pair<glm::vec2,
    std::chrono::steady_clock::time_point>& points) {
00062         auto now = std::chrono::steady_clock::now();
00063         auto& registry = Game::GetInstance().registry;
00064         auto entitiesWithCollider = registry->GetEntitiesFromSystem<CollisionSystem>();
00065
00066         int validPointsCount = 0;
00067         for (const auto& point : points) {
00068             auto duration = std::chrono::duration_cast<std::chrono::seconds>(now - point.second);
00069             if (duration.count() <= 4 && point.first.y > 75) {
00070                 validPointsCount++;
00071             }
00072         }
00073
00074         for (auto entity : entitiesWithCollider) {
00075             if (!entity.HasComponent<EffectReceiverComponent>() ||
00076                 !entity.HasComponent<TransformComponent>() ||
00077                 !entity.HasComponent<EnemyComponent>()) { // Solo enemigos reciben daño
00078                 continue;
00079             }
00080
00081             auto& transform = entity.GetComponent<TransformComponent>();
00082             auto& collider = entity.GetComponent<CircleColliderComponent>();
00083             auto& effectReceiver = entity.GetComponent<EffectReceiverComponent>();
00084
00085             glm::vec2 entityCenter = glm::vec2(
00086                 transform.position.x + (collider.width * transform.scale.x / 2),
00087                 transform.position.y + (collider.height * transform.scale.y / 2)
00088             );
00089
00090             int entityRadius = collider.radius * std::max(transform.scale.x, transform.scale.y) / 2;
00091
00092             // Verificar si la entidad está actualmente sobre algún trazo rojo válido
00093             bool isOnDamageTrace = false;
00094             int collisionCount = 0;
00095
00096             for (const auto& point : points) {
00097                 auto duration = std::chrono::duration_cast<std::chrono::seconds>(now - point.second);
00098                 if (duration.count() > 4) continue;
00099
00100                 if (point.first.y <= 75) continue;
00101
00102                 if (CheckPointToCircleCollision(point.first, entityCenter, entityRadius +
    EFFECT_RADIUS)) {
00103                     isOnDamageTrace = true;
00104                     collisionCount++;
00105                 }
00106             }
00107
00108             effectReceiver.takingDamage = isOnDamageTrace;
00109
```

```
00110         }
00111     }
00117     void ProcessOtherEffects(const std::vector<std::pair<glm::vec2,
      std::chrono::steady_clock::time_point>& points, int colorIndex) {
00118         auto now = std::chrono::steady_clock::now();
00119         auto& registry = Game::GetInstance().registry;
00120         auto entitiesWithCollider = registry->GetEntitiesFromSystem<CollisionSystem>();
00121 
00122         for (auto entity : entitiesWithCollider) {
00123             if (!entity.HasComponent<EffectReceiverComponent>() ||
      !entity.HasComponent<TransformComponent>()) {
00124                 continue;
00125             }
00126 
00127             auto& transform = entity.GetComponent<TransformComponent>();
00128             auto& collider = entity.GetComponent<CircleColliderComponent>();
00129             auto& effectReceiver = entity.GetComponent<EffectReceiverComponent>();
00130 
00131             glm::vec2 entityCenter = glm::vec2(
00132                 transform.position.x + (collider.width * transform.scale.x / 2),
00133                 transform.position.y + (collider.height * transform.scale.y / 2)
00134             );
00135 
00136             int entityRadius = collider.radius * transform.scale.x / 2;
00137             bool isOnTrace = false;
00138 
00139             for (const auto& point : points) {
00140                 auto duration = std::chrono::duration_cast<std::chrono::seconds>(now - point.second);
00141                 if (duration.count() > 4) continue;
00142                 if (point.first.y <= 75) continue;
00143 
00144                 if (CheckPointToCircleCollision(point.first, entityCenter, entityRadius +
      EFFECT_RADIUS)) {
00145                     isOnTrace = true;
00146                     break;
00147                 }
00148             }
00149 
00150             // Aplicar efectos según el color
00151             bool isEnemy = entity.HasComponent<EnemyComponent>();
00152 
00153             switch (colorIndex) {
00154                 case 1: // Azul - Speed (solo jugador)
00155                     if (!isEnemy) {
00156                         effectReceiver.speedBoosted = isOnTrace;
00157                     } else {
00158                         effectReceiver.speedBoosted = false;
00159                     }
00160                     break;
00161 
00162                 case 2: // Verde - Slow (solo enemigos)
00163                     if (isEnemy) {
00164                         effectReceiver.slowed = isOnTrace;
00165                     } else {
00166                         effectReceiver.slowed = false;
00167                     }
00168                     break;
00169             }
00170         }
00171     }
00179     bool CheckPointToCircleCollision(const glm::vec2& point, const glm::vec2& circleCenter, int
      radius) {
00180         glm::vec2 diff = point - circleCenter;
00181         double distance = glm::sqrt((diff.x * diff.x) + (diff.y * diff.y));
00182         return distance <= radius;
00183     }
00184 };
00185 
00186 #endif // DRAWINGEFFECTSYSTEM_HPP
```

2.35 DrawSystem.hpp

```
00001 #ifndef DRAWSYSTEM_HPP
00002 #define DRAWSYSTEM_HPP
00003 
00004 #include <SDL2/SDL.h>
00005 #include <vector>
00006 #include "../ECS/ECS.hpp"
00007 #include "../Components/DrawableComponent.hpp"
00008 #include <chrono>
00009 #include "../Game/Game.hpp"
00010 #include "../Systems/ChargeManageSystem.hpp"
00011 
00017 class DrawSystem : public System {
```

```
00018 public:
00024     DrawSystem() {
00025         RequireComponent<DrawableComponent>();
00026     }
00027 
00032     void Update(SDL_Renderer* renderer) {
00033         // Process each entity with drawable component
00034         for (auto entity : GetSystemEntities()) {
00035             auto& drawable = entity.GetComponent<DrawableComponent>();
00036 
00037             // Process each color channel
00038             for (size_t i = 0; i < drawable.colorPoints.size(); ++i) {
00039                 // Set color based on channel index
00040                 SDL_Color color;
00041                 switch (i) {
00042                     case 0: color = {255, 0, 0, 255}; break;   // Red
00043                     case 1: color = {0, 0, 255, 255}; break;   // Blue
00044                     case 2: color = {0, 255, 0, 255}; break;   // Green
00045                     default: color = {255, 255, 255, 255}; break; // White
00046                 }
00047 
00048                 SDL_SetRenderDrawColor(renderer, color.r, color.g, color.b, color.a);
00049 
00050                 // Remove points older than 4 seconds when game is not paused
00051                 if (!Game::GetInstance().isPaused) {
00052                     auto now = std::chrono::steady_clock::now();
00053                     for (auto it = drawable.colorPoints[i].begin(); it !=
    drawable.colorPoints[i].end(); ) {
00054                         auto duration = std::chrono::duration_cast<std::chrono::seconds>(now -
    it->second);
00055                         if (duration.count() > 4) {
00056                             it = drawable.colorPoints[i].erase(it); // Remove expired point
00057                         } else {
00058                             ++it;
00059                         }
00060                     }
00061                 }
00062 
00063                 // Draw remaining points in gameplay area (below y=175)
00064                 for (const auto& point : drawable.colorPoints[i]) {
00065                     if (point.first.y > 175) {
00066                         int size = 10; // Stroke size
00067                         SDL_Rect drawRect = {
00068                             static_cast<int>(point.first.x) - size / 2,
00069                             static_cast<int>(point.first.y) - size / 2,
00070                             size,
00071                             size
00072                         };
00073                         SDL_RenderFillRect(renderer, &drawRect);
00074                     }
00075                 }
00076             }
00077         }
00078     }
00079 };
00080 
00081 #endif // DRAWSYSTEM_HPP
```

2.36 EnemySystem.hpp

```
00001 #ifndef ENEMYSYSTEM_HPP
00002 #define ENEMYSYSTEM_HPP
00003 
00004 #include "../ECS/ECS.hpp"
00005 #include "../Components/EnemyComponent.hpp"
00006 #include "../Components/AnimationComponent.hpp"
00007 #include "../Components/CircleColliderComponent.hpp"
00008 #include "../Components/HealthComponent.hpp"
00009 #include "../Components/RigidBodyComponent.hpp"
00010 #include "../Components/SpriteComponent.hpp"
00011 #include "../Components/TransformComponent.hpp"
00012 #include "../Components/ScriptComponent.hpp"
00013 #include "../Components/DrawableComponent.hpp"
00014 #include "../Components/EnemyComponent.hpp"
00015 #include "../Components/EffectReceiverComponent.hpp"
00016 #include "../Components/TextComponent.hpp"
00017 #include "../Components/ProjectileComponent.hpp"
00018 #include <memory>
00019 #include <cstdlib>
00020 
00027 class EnemySystem : public System {
00028 public:
00034     EnemySystem() {
00035         RequireComponent<EnemyComponent>();
```

```
00036     }
00037
00042     void Update(std::unique_ptr<Registry>& registry) {
00043         Game& game = Game::GetInstance();
00044         for (auto spawner : GetSystemEntities()) {
00045             auto& enemySpawner = spawner.GetComponent<EnemyComponent>();
00046
00047             int currentAlive = CountClonesFrom(spawner.GetId());
00048             if (currentAlive < enemySpawner.amountToSpawn && enemySpawner.totalAmount > 0) {
00049                 Entity newEnemy = registry->CreateEntity();
00050                 CloneEntityFromTemplate(spawner, newEnemy);
00051                 enemySpawner.totalAmount--;
00052                 game.enemiesLeftToSpawn--;
00053             }
00054         }
00055     }
00056
00065     void CreateEnemyProjectile(std::unique_ptr<Registry>& registry, glm::vec2 velocity, glm::vec2
      position, double rotation, int damage) {
00066         for (auto spawner : GetSystemEntities()) {
00067             if (spawner.HasComponent<ProjectileComponent>()) {
00068                 Entity newEnemy = registry->CreateEntity();
00069                 CloneEntityFromTemplate(spawner, newEnemy);
00070                 newEnemy.GetComponent<TransformComponent>().position = position;
00071                 newEnemy.GetComponent<TransformComponent>().rotation = rotation;
00072                 newEnemy.GetComponent<RigidBodyComponent>().velocity = velocity;
00073                 newEnemy.GetComponent<HealthComponent>().damage = damage;
00074                 break;
00075             }
00076         }
00077     }
00078
00079 private:
00085     int CountClonesFrom(int spawnerId) {
00086         int count = 0;
00087         for (auto entity : GetSystemEntities()) {
00088             auto& enemy = entity.GetComponent<EnemyComponent>();
00089             if (entity.GetId() != spawnerId && enemy.spawnerId == spawnerId) {
00090                 count++;
00091             }
00092         }
00093         return count;
00094     }
00095
00101     void CloneEntityFromTemplate(Entity source, Entity target) {
00102         // Animation
00103         if (source.HasComponent<AnimationComponent>()) {
00104             target.AddComponent<AnimationComponent>(source.GetComponent<AnimationComponent>());
00105         }
00106
00107         // Collider
00108         if (source.HasComponent<CircleColliderComponent>()) {
00109
      target.AddComponent<CircleColliderComponent>(source.GetComponent<CircleColliderComponent>());
00110         }
00111
00112         // Health
00113         if (source.HasComponent<HealthComponent>()) {
00114             target.AddComponent<HealthComponent>(source.GetComponent<HealthComponent>());
00115         }
00116
00117         // Rigidbody
00118         if (source.HasComponent<RigidBodyComponent>()) {
00119             target.AddComponent<RigidBodyComponent>(source.GetComponent<RigidBodyComponent>());
00120         }
00121
00122         // Script
00123         if (source.HasComponent<ScriptComponent>()) {
00124             auto script = source.GetComponent<ScriptComponent>();
00125             target.AddComponent<ScriptComponent>(script);
00126         }
00127
00128         // Sprite
00129         if (source.HasComponent<SpriteComponent>()) {
00130             auto sprite = source.GetComponent<SpriteComponent>();
00131             sprite.active = true;  // Activate visibility for clone
00132             target.AddComponent<SpriteComponent>(sprite);
00133         }
00134
00135         // Transform with random position
00136         if (source.HasComponent<TransformComponent>()) {
00137             auto transform = source.GetComponent<TransformComponent>();
00138             transform.position = GetRandomSpawnPosition();
00139             target.AddComponent<TransformComponent>(transform);
00140         }
00141
00142         if (source.HasComponent<EffectReceiverComponent>()) {
```

```
00143
      target.AddComponent<EffectReceiverComponent>(source.GetComponent<EffectReceiverComponent>());
00144         }
00145
00146         if (source.HasComponent<TextComponent>()) {
00147             target.AddComponent<TextComponent>(source.GetComponent<TextComponent>());
00148         }
00149
00150         if (source.HasComponent<ProjectileComponent>()) {
00151             target.AddComponent<ProjectileComponent>(source.GetComponent<ProjectileComponent>());
00152         }
00153
00154         // EnemyComponent without spawn capability
00155         if (source.HasComponent<EnemyComponent>()) {
00156             target.AddComponent<EnemyComponent>(source.GetComponent<EnemyComponent>());
00157             target.GetComponent<EnemyComponent>().amountToSpawn = 0;
00158             target.GetComponent<EnemyComponent>().spawnerId = source.GetId();
00159         }
00160     }
00161
00166     glm::vec2 GetRandomSpawnPosition() {
00167         int x, y;
00168
00169         // Choose left or right side
00170         bool leftRight = rand() % 2;
00171         if (leftRight) {
00172             x = rand() % 41; // 0 - 40
00173         } else {
00174             x = 760 + (rand() % 41); // 760 - 800
00175         }
00176
00177         // Choose top or bottom
00178         bool topBottom = rand() % 2;
00179         if (topBottom) {
00180             y = 75 + (rand() % 31); // 75 - 105
00181         } else {
00182             y = 560 + (rand() % 41); // 560 - 600
00183         }
00184
00185         return glm::vec2(static_cast<float>(x), static_cast<float>(y));
00186     }
00187 };
00188
00189 #endif // ENEMYSYSTEM_HPP
```

2.37 HealthSystem.hpp

```
00001 #ifndef HEALTHSYSTEM_HPP
00002 #define HEALTHSYSTEM_HPP
00003
00004 #include <memory>
00005 #include "../Components/HealthComponent.hpp"
00006 #include "../Components/EffectReceiverComponent.hpp"
00007 #include "../Components/RigidBodyComponent.hpp"
00008 #include "../ECS/ECS.hpp"
00009
00016 class HealthSystem : public System {
00017 public:
00023     HealthSystem() {
00024         RequireComponent<HealthComponent>();
00025     }
00026
00033     void Update() {
00034         auto& registry = Game::GetInstance().registry;
00035         Entity playerEntity(-1);
00036         bool foundPlayer = false;
00037
00038         // Get entities managed by HealthSystem
00039         auto entities = registry->GetEntitiesFromSystem<HealthSystem>();
00040
00041         // Find the player entity
00042         for (auto& entity : entities) {
00043             if (entity.HasComponent<HealthComponent>()) {
00044                 auto& health = entity.GetComponent<HealthComponent>();
00045                 if (health.isPlayer) {
00046                     playerEntity = entity;
00047                     foundPlayer = true;
00048                     break;
00049                 }
00050             }
00051         }
00052
00053         if (foundPlayer && playerEntity.HasComponent<HealthComponent>()) {
00054             auto& playerHealth = playerEntity.GetComponent<HealthComponent>();
```

```
00055             auto& playerDamage = playerHealth.damage;
00056             auto& playerTimeout = playerHealth.attackTimeout;
00057 
00058             // Process all effect-receiving entities
00059             for (auto& entity : entities) {
00060                 if (entity.HasComponent<HealthComponent>() &&
      entity.HasComponent<EffectReceiverComponent>()) {
00061                     auto& effectReceiver = entity.GetComponent<EffectReceiverComponent>();
00062                     auto& entityHealth = entity.GetComponent<HealthComponent>();
00063 
00064                     // Process zone damage ONLY if entity is currently in damage zone
00065                     if (effectReceiver.takingDamage && !entityHealth.isPlayer) {
00066                         ProcessZoneDamage(entity, playerDamage, playerTimeout);
00067                     }
00068                     ApplySpeedEffect(entity, effectReceiver, entityHealth);
00069                     // Process other effects
00070                     if (effectReceiver.slowed && !entityHealth.isPlayer) {
00071                         // Apply slow logic
00072                         // std::cout « "Enemy slowed" « std::endl;
00073                     }
00074 
00075                     if (effectReceiver.speedBoosted && entityHealth.isPlayer) {
00076                         // Apply speed boost logic
00077                         // std::cout « "Player speed boosted" « std::endl;
00078                     }
00079                 }
00080             }
00081         }
00082     }
00083 
00084 private:
00091     void ApplySpeedEffect(Entity& entity, EffectReceiverComponent& effectReceiver, HealthComponent&
      entityHealth) {
00092         bool isPlayer = entityHealth.isPlayer;
00093         if (isPlayer && effectReceiver.speedBoosted) {
00094             entity.GetComponent<RigidBodyComponent>().velocity *= 1.5f; // Increase speed
00095         } else if (!isPlayer && effectReceiver.slowed) {
00096             entity.GetComponent<RigidBodyComponent>().velocity *= 0.3f; // Reduce speed
00097         }
00098     }
00099 
00106     void ProcessZoneDamage(Entity entity, int damage, float damageInterval) {
00107         if (!entity.HasComponent<HealthComponent>()) return;
00108 
00109         auto& targetHealth = entity.GetComponent<HealthComponent>();
00110 
00111         // Get current time
00112         auto now = std::chrono::steady_clock::now();
00113 
00114         // Calculate time since last damage
00115         auto elapsed = std::chrono::duration_cast<std::chrono::milliseconds>(now -
      targetHealth.lastDamageReceived).count();
00116         int intervalMs = static_cast<int>(damageInterval * 1000); // seconds to ms
00117 
00118         // Skip if not enough time passed
00119         if (elapsed < intervalMs) {
00120             return;
00121         }
00122 
00123         // Verify entity is still taking damage
00124         if (!entity.HasComponent<EffectReceiverComponent>()) return;
00125         auto& effectReceiver = entity.GetComponent<EffectReceiverComponent>();
00126         if (!effectReceiver.takingDamage) return;
00127 
00128         // Update last damage time
00129         targetHealth.lastDamageReceived = now;
00130 
00131         // Apply damage
00132         targetHealth.health -= damage;
00133 
00134         // Check for death
00135         if (targetHealth.health <= 0) {
00136             targetHealth.health = 0;
00137             if (!targetHealth.isPlayer) {
00138                 Game::GetInstance().totalPoints += entity.GetComponent<EnemyComponent>().points;
00139                 Game::GetInstance().enemiesLeft--;
00140                 entity.Kill();
00141                 if (Game::GetInstance().enemiesLeft == 0) {
00142                     Game::GetInstance().finDelNivel = true;
00143                     Game::GetInstance().win = true;
00144                 }
00145             }
00146         }
00147     }
00148 
00149 public:
00156     void ReduceHP(Entity entity, int damage, Entity attacker) {
```

```
00157         if (!entity.HasComponent<HealthComponent>()) return;
00158         if (!attacker.HasComponent<HealthComponent>()) return;
00159 
00160         auto& attackerHealth = attacker.GetComponent<HealthComponent>();
00161         auto& targetHealth = entity.GetComponent<HealthComponent>();
00162 
00163         // Get current time
00164         auto now = std::chrono::steady_clock::now();
00165 
00166         // Calculate time since attacker's last attack
00167         auto elapsed = std::chrono::duration_cast<std::chrono::milliseconds>(now -
      attackerHealth.attackTimeoutDuration).count();
00168         int timeoutMs = static_cast<int>(attackerHealth.attackTimeout * 1000);
00169 
00170         if (elapsed < timeoutMs) {
00171             return;
00172         }
00173 
00174         // Update attacker's last attack time
00175         attackerHealth.attackTimeoutDuration = now;
00176 
00177         // Apply damage
00178         targetHealth.health -= damage;
00179 
00180         if (targetHealth.health <= 0) {
00181             targetHealth.health = 0;
00182             if (!targetHealth.isPlayer) {
00183                 entity.Kill();
00184             } else {
00185                 Game::GetInstance().finDelNivel = true;
00186                 Game::GetInstance().win = false;
00187             }
00188         }
00189     }
00190 
00196     void SetHealth(Entity entity, int value) {
00197         if (entity.HasComponent<HealthComponent>()) {
00198             auto& health = entity.GetComponent<HealthComponent>();
00199             health.health = std::max(0, value);
00200 
00201             if (health.health == 0 && !health.isPlayer) {
00202                 entity.Kill();
00203             }
00204         }
00205     }
00206 
00212     void Heal(Entity entity, int amount) {
00213         if (entity.HasComponent<HealthComponent>()) {
00214             auto& health = entity.GetComponent<HealthComponent>();
00215             health.health += amount;
00216             health.health = std::min(health.health, health.maxHealth);
00217         }
00218     }
00219 };
00220 
00221 #endif // HEALTHSYSTEM_HPP
```

2.38 MovementSystem.hpp

```
00001 #ifndef MOVEMENTSYSTEM_HPP
00002 #define MOVEMENTSYSTEM_HPP
00003 
00004 #include "../Components/RigidBodyComponent.hpp"
00005 #include "../Components/TransformComponent.hpp"
00006 #include "../Components/SpriteComponent.hpp"
00007 #include "../Components/ProjectileComponent.hpp"
00008 #include "../ECS/ECS.hpp"
00009 #include "../Game/Game.hpp"
00010 
00017 class MovementSystem : public System {
00018 public:
00027     MovementSystem() {
00028         RequireComponent<RigidBodyComponent>();
00029         RequireComponent<TransformComponent>();
00030         RequireComponent<SpriteComponent>();
00031     }
00032 
00037     void Update(double dt) {
00038         auto& game = Game::GetInstance();
00039 
00040         for (auto entity : GetSystemEntities()) {
00041             const auto& rigidBody = entity.GetComponent<RigidBodyComponent>();
00042             auto& transform = entity.GetComponent<TransformComponent>();
00043             auto& sprite = entity.GetComponent<SpriteComponent>();
```

```
00044
00045             // Skip inactive sprites
00046             if (!sprite.active) {
00047                 continue;
00048             }
00049
00050             // Update position based on velocity and delta time
00051             transform.position.x += rigidBody.velocity.x * dt;
00052             transform.position.y += rigidBody.velocity.y * dt;
00053
00054             bool crash = false;
00055
00056             // X-axis boundary checks (with sprite width consideration)
00057             if (transform.position.x < 0) {
00058                 transform.position.x = 0;
00059                 crash = true;
00060             } else if (transform.position.x > game.windowWidth - (sprite.width * transform.scale.x)) {
00061                 transform.position.x = game.windowWidth - (sprite.width * transform.scale.x);
00062                 crash = true;
00063             }
00064
00065             // Y-axis boundary checks (with sprite height consideration)
00066             if (transform.position.y < 175) {  // Top boundary
00067                 transform.position.y = 175;
00068                 crash = true;
00069             } else if (transform.position.y > game.windowHeight - (sprite.height * transform.scale.y)
      - 25) {  // Bottom boundary
00070                 transform.position.y = game.windowHeight - (sprite.height * transform.scale.y) - 25;
00071                 crash = true;
00072             }
00073
00074             // Update sprite orientation based on horizontal velocity
00075             if (rigidBody.velocity.x < 0) {
00076                 sprite.flip = SDL_FLIP_HORIZONTAL;
00077             } else if (rigidBody.velocity.x > 0) {
00078                 sprite.flip = SDL_FLIP_NONE;
00079             }
00080
00081             // Special handling for projectiles
00082             if (entity.HasComponent<ProjectileComponent>()) {
00083                 sprite.flip = SDL_FLIP_NONE;
00084             }
00085
00086             // Destroy projectiles that hit boundaries
00087             if (entity.HasComponent<ProjectileComponent>() && crash) {
00088                 sprite.flip = SDL_FLIP_NONE;
00089                 entity.Kill();
00090             }
00091         }
00092     }
00093 };
00094
00095 #endif // MOVEMENTSYSTEM_HPP
```

2.39 RenderSystem.hpp

```
00001 #ifndef RENDERSYSTEM_HPP
00002 #define RENDERSYSTEM_HPP
00003
00004 #include <SDL2/SDL.h>
00005 #include "../AssetManager/AssetManager.hpp"
00006 #include "../Components/SpriteComponent.hpp"
00007 #include "../Components/TransformComponent.hpp"
00008 #include "../ECS/ECS.hpp"
00009
00016 class RenderSystem : public System {
00017 public:
00023     RenderSystem() {
00024         RequireComponent<TransformComponent>();
00025         RequireComponent<SpriteComponent>();
00026     }
00027
00033     void Update(SDL_Renderer* renderer, std::unique_ptr<AssetManager>& AssetManager) {
00034         std::vector<Entity> entities = GetSystemEntities();
00035
00036         size_t startIndex = 0;
00037
00038         // Special case: skip background if it's the first entity
00039         if (!entities.empty()) {
00040             const auto firstEntity = entities[0];
00041             if (firstEntity.HasComponent<SpriteComponent>()) {
00042                 const auto sprite = firstEntity.GetComponent<SpriteComponent>();
00043                 if (sprite.textureId.find("background") != std::string::npos) {
00044                     startIndex = 1;  // Skip first element (background)
```

```
00045                 }
00046             }
00047         }
00048 
00049         // Render all entities starting from startIndex
00050         for (size_t i = startIndex; i < entities.size(); ++i) {
00051             const auto entity = entities[i];
00052             const auto transform = entity.GetComponent<TransformComponent>();
00053             const auto sprite = entity.GetComponent<SpriteComponent>();
00054 
00055             // Skip inactive sprites
00056             if (!sprite.active) {
00057                 continue;
00058             }
00059 
00060             // Prepare source and destination rectangles
00061             SDL_Rect srcRect = sprite.srcRect;
00062             SDL_Rect dstRect = {
00063                 static_cast<int>(transform.position.x),
00064                 static_cast<int>(transform.position.y),
00065                 static_cast<int>(sprite.width * transform.scale.x),
00066                 static_cast<int>(sprite.height * transform.scale.y)
00067             };
00068 
00069             // Render with optional flip and rotation
00070             SDL_RenderCopyEx(
00071                 renderer,
00072                 AssetManager->GetTexture(sprite.textureId),
00073                 &srcRect,
00074                 &dstRect,
00075                 transform.rotation,
00076                 NULL,
00077                 sprite.flip
00078             );
00079         }
00080     }
00081 
00087     void UpdateBackground(SDL_Renderer* renderer, std::unique_ptr<AssetManager>& AssetManager) {
00088         auto& registry = Game::GetInstance().registry;
00089         auto entities = registry->GetEntitiesFromSystem<RenderSystem>();
00090 
00091         for (auto entity : entities) {
00092             if (!entity.HasComponent<TransformComponent>()) {
00093                 continue;
00094             }
00095             const auto transform = entity.GetComponent<TransformComponent>();
00096             const auto sprite = entity.GetComponent<SpriteComponent>();
00097 
00098             // Only render active background sprites
00099             if (!sprite.active || sprite.textureId.find("background") == std::string::npos) {
00100                 continue;
00101             }
00102 
00103             // Prepare source and destination rectangles
00104             SDL_Rect srcRect = sprite.srcRect;
00105             SDL_Rect dstRect = {
00106                 static_cast<int>(transform.position.x),
00107                 static_cast<int>(transform.position.y),
00108                 static_cast<int>(sprite.width * transform.scale.x),
00109                 static_cast<int>(sprite.height * transform.scale.y)
00110             };
00111 
00112             // Render with optional flip and rotation
00113             SDL_RenderCopyEx(
00114                 renderer,
00115                 AssetManager->GetTexture(sprite.textureId),
00116                 &srcRect,
00117                 &dstRect,
00118                 transform.rotation,
00119                 NULL,
00120                 sprite.flip
00121             );
00122         }
00123     }
00124 };
00125 
00126 #endif // RENDERSYSTEM_HPP
```

2.40 RenderTextSystem.hpp

```
00001 #ifndef RENDERTEXTSYSTEM_HPP
00002 #define RENDERTEXTSYSTEM_HPP
00003 
00004 #include <SDL2/SDL.h>
```

```
00005 #include <SDL2/SDL_ttf.h>
00006 #include <memory>
00007 #include "../AssetManager/AssetManager.hpp"
00008 #include "../Components/TextComponent.hpp"
00009 #include "../Components/TransformComponent.hpp"
00010 #include "../Components/HealthComponent.hpp"
00011 #include "../Components/DamageChargeComponent.hpp"
00012 #include "../Components/SprintChargeComponent.hpp"
00013 #include "../Components/SlowChargeComponent.hpp"
00014 #include "../Components/IdentifierComponent.hpp"
00015 #include "../ECS/ECS.hpp"
00016
00026 class RenderTextSystem : public System {
00027 public:
00033     RenderTextSystem() {
00034         RequireComponent<TextComponent>();
00035         RequireComponent<TransformComponent>();
00036     }
00037
00043     void Update(SDL_Renderer* renderer, std::unique_ptr<AssetManager>& assetManager) {
00044         for (auto entity : GetSystemEntities()) {
00045             auto& text = entity.GetComponent<TextComponent>();
00046             auto& transform = entity.GetComponent<TransformComponent>();
00047
00048             // Case 1: Health display (floating above entities)
00049             if (entity.HasComponent<HealthComponent>()) {
00050                 const auto sprite = entity.GetComponent<SpriteComponent>();
00051                 if (!sprite.active) {
00052                     continue;
00053                 }
00054
00055                 text.text = std::to_string(entity.GetComponent<HealthComponent>().health);
00056                 SDL_Surface* surface = TTF_RenderText_Blended(assetManager->GetFont(text.fontId),
    text.text.c_str(), text.color);
00057                 text.width = surface->w;
00058                 text.height = surface->h;
00059                 SDL_Texture* texture = SDL_CreateTextureFromSurface(renderer, surface);
00060                 SDL_FreeSurface(surface);
00061
00062                 SDL_Rect dstrect = {
00063                     static_cast<int>(transform.position.x),
00064                     static_cast<int>(transform.position.y - 20),  // Position above entity
00065                     text.width * static_cast<int>(transform.scale.x) / 2,
00066                     text.height * static_cast<int>(transform.scale.y) / 2
00067                 };
00068
00069                 SDL_RenderCopy(renderer, texture, NULL, &dstrect);
00070                 SDL_DestroyTexture(texture);
00071             }
00072             // Case 2: Charge displays (damage, sprint, slow)
00073             else if (entity.HasComponent<DamageChargeComponent>() ||
00074                      entity.HasComponent<SprintChargeComponent>() ||
00075                      entity.HasComponent<SlowChargeComponent>()) {
00076
00077                 if (entity.HasComponent<DamageChargeComponent>()) {
00078                     text.text = entity.GetComponent<DamageChargeComponent>().chargeDisplay;
00079                 } else if (entity.HasComponent<SprintChargeComponent>()) {
00080                     text.text = entity.GetComponent<SprintChargeComponent>().chargeDisplay;
00081                 } else if (entity.HasComponent<SlowChargeComponent>()) {
00082                     text.text = entity.GetComponent<SlowChargeComponent>().chargeDisplay;
00083                 }
00084
00085                 SDL_Surface* surface = TTF_RenderText_Blended(assetManager->GetFont(text.fontId),
    text.text.c_str(), text.color);
00086                 text.width = surface->w;
00087                 text.height = surface->h;
00088                 SDL_Texture* texture = SDL_CreateTextureFromSurface(renderer, surface);
00089                 SDL_FreeSurface(surface);
00090
00091                 SDL_Rect dstrect = {
00092                     static_cast<int>(transform.position.x),
00093                     static_cast<int>(transform.position.y - 20),  // Position above
00094                     text.width * static_cast<int>(transform.scale.x) / 2,
00095                     text.height * static_cast<int>(transform.scale.y) / 2
00096                 };
00097
00098                 SDL_RenderCopy(renderer, texture, NULL, &dstrect);
00099                 SDL_DestroyTexture(texture);
00100             }
00101             // Case 3: Score display
00102             else if (entity.HasComponent<IdentifierComponent>() &&
00103                      entity.GetComponent<IdentifierComponent>().name == "puntuacion") {
00104
00105                 text.text = "Score: " + std::to_string(Game::GetInstance().totalPoints);
00106
00107                 SDL_Surface* surface = TTF_RenderText_Blended(assetManager->GetFont(text.fontId),
    text.text.c_str(), text.color);
```

```
00108                 text.width = surface->w;
00109                 text.height = surface->h;
00110                 SDL_Texture* texture = SDL_CreateTextureFromSurface(renderer, surface);
00111                 SDL_FreeSurface(surface);
00112 
00113                 SDL_Rect dstrect = {
00114                     static_cast<int>(transform.position.x),
00115                     static_cast<int>(transform.position.y - 20),  // Position above
00116                     text.width * static_cast<int>(transform.scale.x) / 2,
00117                     text.height * static_cast<int>(transform.scale.y) / 2
00118                 };
00119 
00120                 SDL_RenderCopy(renderer, texture, NULL, &dstrect);
00121                 SDL_DestroyTexture(texture);
00122             }
00123             // Case 4: Default text rendering
00124             else {
00125                 SDL_Surface* surface = TTF_RenderText_Blended(assetManager->GetFont(text.fontId),
      text.text.c_str(), text.color);
00126                 text.width = surface->w;
00127                 text.height = surface->h;
00128                 SDL_Texture* texture = SDL_CreateTextureFromSurface(renderer, surface);
00129                 SDL_FreeSurface(surface);
00130 
00131                 SDL_Rect dstrect = {
00132                     static_cast<int>(transform.position.x),
00133                     static_cast<int>(transform.position.y),
00134                     text.width * static_cast<int>(transform.scale.x),
00135                     text.height * static_cast<int>(transform.scale.y)
00136                 };
00137 
00138                 SDL_RenderCopy(renderer, texture, NULL, &dstrect);
00139                 SDL_DestroyTexture(texture);
00140             }
00141         }
00142     }
00143 };
00144 
00145 #endif // RENDERTEXTSYSTEM_HPP
```

2.41 ScriptSystem.hpp

```
00001 #ifndef SCRIPt_SYSTEM_HPP
00002 #define SCRIPt_SYSTEM_HPP
00003 
00004 #include <memory>
00005 #include <sol/sol.hpp>
00006 #include "../Binding/LuaBinding.hpp"
00007 #include "../Components/ScriptComponent.hpp"
00008 #include "../ECS/ECS.hpp"
00009 
00017 class ScriptSystem : public System {
00018 public:
00022     ScriptSystem() {
00023         RequireComponent<ScriptComponent>();
00024     }
00025 
00036     void CreateLuaBinding(sol::state& lua) {
00037         lua.script("math.randomseed(os.time())");
00038         lua.new_usertype<Entity>("entity");
00039         lua.set_function("is_action_activated", IsActionActivated);
00040         lua.set_function("set_velocity", SetVelocity);
00041         lua.set_function("go_to_scene", GoToScene);
00042 
00043         lua.set_function("is_mouse_button_down", IsMouseButtonDown);
00044 
00045         lua.set_function("push_draw_point", PushDrawPoint);
00046 
00047         lua.set_function("get_mouse_position", GetMousePosition);
00048         lua.set_function("get_player_position", GetPlayerPosition);
00049         lua.set_function("get_enemy_position", GetEnemyPosition);
00050         lua.set_function("attack_melee", AttackMelee);
00051         lua.set_function("get_all_enemies", GetAllEnemies);
00052         lua.set_function("get_enemy_position_by_id", GetEnemyPositionById);
00053         lua.set_function("attack_ranger", AttackRanger);
00054         lua.set_function("set_draw_index", CurrentDrawIndex);
00055         lua.set_function("set_level", SetLevel);
00056     }
00057 
00065     void Update(sol::state& lua) {
00066         for (auto entity : GetSystemEntities()) {
00067             const auto& script = entity.GetComponent<ScriptComponent>();
00068 
00069             if (script.update != sol::lua_nil) {
```

```
00070                 lua["this"] = entity;
00071                 script.update();
00072             }
00073         }
00074     }
00075 };
00076
00077 #endif // SCRIPt_SYSTEM_HPP
```

2.42 SoundSystem.hpp

```
00001 #ifndef SOUNDSYSTEM_HPP
00002 #define SOUNDSYSTEM_HPP
00003
00004 #include <SDL2/SDL_mixer.h>
00005 #include <vector>
00006 #include <memory>
00007 #include "../AssetManager/AssetManager.hpp"
00008 #include "../Components/SoundComponent.hpp"
00009 #include "../ECS/ECS.hpp"
00010
00018 class SoundSystem : public System {
00019 public:
00023     SoundSystem() {
00024         RequireComponent<SoundComponent>();
00025     }
00026
00034     void Update(std::unique_ptr<AssetManager>& assetManager) {
00035         std::vector<Entity> entities = GetSystemEntities();
00036         for (auto& entity : entities) {
00037             auto& sound = entity.GetComponent<SoundComponent>();
00038
00039             if (!sound.active) {
00040                 continue;
00041             }
00042             // Auto-reproducir sonidos marcados para autoPlay
00043             if (sound.autoPlay && !sound.isPlaying) {
00044                 PlaySound(assetManager, sound);
00045             }
00046         }
00047     }
00048
00057     void PlaySound(std::unique_ptr<AssetManager>& assetManager, SoundComponent& sound) {
00058         if (sound.soundId == "none" || !sound.active) {
00059             return;
00060         }
00061
00062         Mix_Chunk* chunk = assetManager->GetSound(sound.soundId);
00063         if (chunk != nullptr) {
00064             Mix_VolumeChunk(chunk, sound.volume);
00065             int channel = Mix_PlayChannel(-1, chunk, sound.loops);
00066             sound.isPlaying = (channel != -1);
00067         }
00068     }
00069
00075     void StopSound(SoundComponent& sound) {
00076         sound.isPlaying = false;
00077         // Nota: Para detener un sonido específico necesitarías trackear el canal
00078     }
00079
00085     void PauseSound(SoundComponent& sound) {
00086         if (sound.isPlaying) {
00087             Mix_Pause(-1); // Pausa todos los canales, idealmente trackearias el canal específico
00088         }
00089     }
00090
00096     void ResumeSound(SoundComponent& sound) {
00097         if (!sound.isPlaying) {
00098             Mix_Resume(-1); // Reanuda todos los canales
00099         }
00100     }
00101 };
00102
00103 #endif
```

2.43 UISystem.hpp

```
00001 #ifndef UISYSTEM_HPP
00002 #define UISYSTEM_HPP
00003
00004 #include <SDL2/SDL.h>
```

```
00005 #include <SDL2/SDL_ttf.h>
00006
00007 #include <memory>
00008 #include <iostream>
00009 #include <string>
00010
00011 #include "../Components/ClickableComponent.hpp"
00012 #include "../Components/TransformComponent.hpp"
00013 #include "../Components/ScriptComponent.hpp"
00014 #include "../Components/TextComponent.hpp"
00015 #include "../ECS/ECS.hpp"
00016 #include "../EventManager/EventManager.hpp"
00017 #include "../Events/ClickEvent.hpp"
00018
00026 class UISystem : public System {
00027     public:
00034         UISystem() {
00035             RequireComponent<ClickableComponent>();
00036             RequireComponent<TransformComponent>();
00037             RequireComponent<TextComponent>();
00038         };
00039
00046         void SubscribeToClickEvent(std::unique_ptr<EventManager>& eventManager) {
00047             eventManager->SubscribeToEvent<UISystem, ClickEvent>(this, &UISystem::OnClickEvent);
00048         };
00049
00057         void OnClickEvent(ClickEvent& e) {
00058             for (auto entity : GetSystemEntities()) {
00059                 auto& transform = entity.GetComponent<TransformComponent>();
00060                 auto& text = entity.GetComponent<TextComponent>();
00061                 if (transform.position.x < e.x && e.x < transform.position.x + text.width &&
00062                     transform.position.y < e.y && e.y < transform.position.y + text.height) {
00063
00064                     if (entity.HasComponent<ClickableComponent>()) {
00065                         const auto& script = entity.GetComponent<ScriptComponent>();
00066                         if (script.onClick != sol::lua_nil) {
00067                             script.onClick();
00068                         }
00069                     }
00070                 }
00071             }
00072         };
00073 };
00074
00075 #endif // UISYSTEM_HPP
```

2.44 Pool.hpp

```
00001 #ifndef POOL_HPP
00002 #define POOL_HPP
00003 #include <vector>
00004
00011 class IPool {
00012     public:
00013         virtual ~IPool() = default;
00014 };
00015
00024 template <typename TComponent>
00025 class Pool : public IPool {
00026     private:
00027         std::vector<TComponent> data;
00028
00029     public:
00034         Pool(int size = 1000) {
00035             data.resize(size);
00036         }
00037
00038         virtual ~Pool() = default;
00039
00044         bool IsEmpty() const {
00045             return data.empty();
00046         }
00047
00052         int GetSize() const {
00053             return static_cast<int>(data.size());
00054         }
00055
00060         void Resize(int n) {
00061             data.resize(n);
00062         }
00063
00067         void Clear() {
00068             data.clear();
00069         }
```

```
00070
00075         void Add(TComponent object) {
00076             data.push_back(object);
00077         }
00078
00084         void Set(unsigned int index, TComponent object) {
00085             data[index] = object;
00086         }
00087
00093         TComponent& Get(unsigned int index) {
00094             return static_cast<TComponent&>(data[index]);
00095         }
00096
00102         TComponent& operator[](unsigned int index) {
00103             return static_cast<TComponent&>(data[index]);
00104         }
00105 };
00106
00107 #endif
```

Index