R-Type

Generated by Doxygen 1.9.8

Chapter 1

Namespace Index

1.1 Namespace List

Here is a list of all namespaces with brief descriptions:

Chapter 2

Hierarchical Index

2.1 Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

Chapter 3

Class Index

3.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

Chapter 4

File Index

4.1 File List

Here is a list of all files with brief descriptions:

Chapter 5

Namespace Documentation

5.1 r_type Namespace Reference

Namespaces

- namespace net

Classes

- class Level

  The Level class template manages the game level, including updating game state, handling collisions, and managing entities.

5.2 r_type::net Namespace Reference

Classes

- class AClient
- class AServer

  AServer class template for managing server operations.
- class Client
- class IClient
- class Server

  A server class that handles client connections and messaging.

Chapter 6

Class Documentation

6.1 AbstractScenes Class Reference

An abstract class that provides a base for managing different scenes in a game.

```
#include <a_scenes.hpp>
```

6.1.1 Detailed Description

An abstract class that provides a base for managing different scenes in a game.

This abstract class implements the ScenesInterface and provides some common functionality.

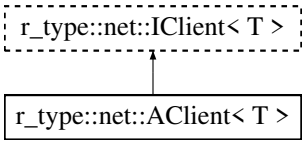
The documentation for this class was generated from the following file:

- /home/runner/work/R-Type/R-Type/ECS/Interface/Include/a_scenes.hpp

6.2 r_type::net::AClient< T > Class Template Reference

```
#include <a_client.hpp>
```

Inheritance diagram for r_type::net::AClient< T >:



Public Member Functions

- AClient ()
- virtual ~AClient ()
- bool Connect (const std::string &host, const uint16_t port)

  Connects to a remote host using UDP protocol.
- void Disconnect ()

  Disconnects the client from the server.
- bool IsConnected ()

  Checks if the client is connected to the server.
- void Send (const Message< T > &msg)

  Send message to server.
- ThreadSafeQueue< OwnedMessage< T > > & Incoming ()

  get incoming messages
- const std::unique_ptr< Connection< T > > & getConnection ()
- void setPlayerId (uint32_t id)
- uint32_t getPlayerId ()
- void setWindowSize (sf::Vector2u size)
- sf::Vector2u getWindowSize ()

Public Member Functions inherited from r_type::net::IClient< T >

- IClient ()
- virtual ~IClient ()

Protected Attributes

- asio::io_context m_context
- std::thread thrContext
- std::unique_ptr< Connection< T > > m_connection

Private Attributes

- ThreadSafeQueue< OwnedMessage< T > > m_qMessagesIn
- uint32_t playerId = 0
- sf::Vector2u windowSize

6.2.1 Constructor & Destructor Documentation

6.2.1.1 AClient()

```
template<typename T >
r_type::net::AClient< T >::AClient ( ) [inline]
```

6.2.1.2 ~AClient()

```
template<typename T >
virtual r_type::net::AClient< T >::~AClient ( ) [inline], [virtual]
```

6.2.2 Member Function Documentation

6.2.2.1 Connect()

```
template<typename T >
bool r_type::net::AClient< T >::Connect (
            const std::string & host,
            const uint16_t port )  [inline], [virtual]
```

Connects to a remote host using UDP protocol.

Parameters

host	The IP address or hostname of the remote host.
port	The port number of the remote host.

Returns

true if the connection is successful, false otherwise.

Implements r_type::net::IClient< T >.

6.2.2.2 Disconnect()

```
template<typename T >
void r_type::net::AClient< T >::Disconnect ( )  [inline], [virtual]
```

Disconnects the client from the server.

This function disconnects the client from the server if it is currently connected. It stops the context and joins the context thread. It also releases the connection resource.

Implements r_type::net::IClient< T >.

6.2.2.3 getConnection()

```
template<typename T >
const std::unique_ptr< Connection< T > > & r_type::net::AClient< T >::getConnection ( )
[inline]
```

6.2.2.4 getPlayerId()

```
template<typename T >
uint32_t r_type::net::AClient< T >::getPlayerId ( )  [inline]
```

6.2.2.5 getWindowSize()

```
template<typename T >
sf::Vector2u r_type::net::AClient< T >::getWindowSize ( )  [inline]
```

6.2.2.6 Incoming()

```
template<typename T >
ThreadSafeQueue< OwnedMessage< T > > & r_type::net::AClient< T >::Incoming ( ) [inline],
[virtual]
```

get incoming messages

Returns

ThreadSafeQueue<OwnedMessage<T>>&

Implements r_type::net::IClient< T >.

6.2.2.7 IsConnected()

```
template<typename T >
bool r_type::net::AClient< T >::IsConnected ( ) [inline], [virtual]
```

Checks if the client is connected to the server.

Returns

true

false

Implements r_type::net::IClient< T >.

6.2.2.8 Send()

```
template<typename T >
void r_type::net::AClient< T >::Send (
            const Message< T > & msg ) [inline], [virtual]
```

Send message to server.

Parameters

msg	

Implements r_type::net::IClient< T >.

6.2.2.9 setPlayerId()

```
template<typename T >
void r_type::net::AClient< T >::setPlayerId (
            uint32_t id ) [inline]
```

6.2.2.10 setWindowSize()

```
template<typename T >
void r_type::net::AClient< T >::setWindowSize (
            sf::Vector2u size )  [inline]
```

6.2.3 Member Data Documentation

6.2.3.1 m_connection

```
template<typename T >
std::unique_ptr<Connection<T> > r_type::net::AClient< T >::m_connection  [protected]
```

6.2.3.2 m_context

```
template<typename T >
asio::io_context r_type::net::AClient< T >::m_context  [protected]
```

6.2.3.3 m_qMessagesIn

```
template<typename T >
ThreadSafeQueue<OwnedMessage<T> > r_type::net::AClient< T >::m_qMessagesIn  [private]
```

6.2.3.4 playerId

```
template<typename T >
uint32_t r_type::net::AClient< T >::playerId = 0  [private]
```

6.2.3.5 thrContext

```
template<typename T >
std::thread r_type::net::AClient< T >::thrContext  [protected]
```

6.2.3.6 windowSize

```
template<typename T >
sf::Vector2u r_type::net::AClient< T >::windowSize  [private]
```

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

- /home/runner/work/R-Type/R-Type/Client/Interface/Include/Net/a_client.hpp

6.3 AllyComponent Struct Reference

```
#include <ally_component.hpp>
```

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

- /home/runner/work/R-Type/R-Type/ECS/Interface/Include/Components/ally_component.hpp

6.4 AllyMissileComponent Struct Reference

```
#include <ally_missile_component.hpp>
```

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

- /home/runner/work/R-Type/R-Type/ECS/Interface/Include/Components/ally_missile_component.hpp

6.5 AnimationComponent Struct Reference

A component that holds animation properties such as offset and dimension.

```
#include <animation_component.hpp>
```

Public Member Functions

- AnimationComponent (vf2d _offset, vf2d _dimension)

  Constructs an AnimationComponent with the given offset and dimension.

Public Attributes

- vf2d offset

  The offset of the animation.
- vf2d dimension

  The dimension of the animation.

6.5.1 Detailed Description

A component that holds animation properties such as offset and dimension.

This component is used to define the properties of an animation, including its offset and dimension.

6.5.2 Constructor & Destructor Documentation

6.5.2.1 AnimationComponent()

```
AnimationComponent::AnimationComponent (
            vf2d _offset,
            vf2d _dimension )  [inline]
```

Constructs an AnimationComponent with the given offset and dimension.

Parameters

_offset	The offset of the animation.
_dimension	The dimension of the animation.

6.5.3 Member Data Documentation

6.5.3.1 dimension

`AnimationComponent::dimension`

The dimension of the animation.

6.5.3.2 offset

`AnimationComponent::offset`

The offset of the animation.

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

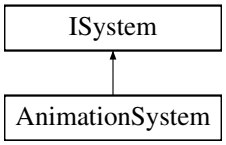
- /home/runner/work/R-Type/R-Type/ECS/Interface/Include/Components/animation_component.hpp

6.6 AnimationSystem Class Reference

A system responsible for animating entities within the ECS framework.

`#include <animation_system.hpp>`

Inheritance diagram for AnimationSystem:



Public Member Functions

- AnimationSystem (ComponentManager &componentManager, EntityManager &entityManager)
- void AnimationEntities (ComponentManager &componentManager, EntityManager &entityManager, float deltaTime, bool &endOfLevel)

  Animates entities.
- void animatePlayer (std::optional< VelocityComponent * > &velocity, std::optional< AnimationComponent * > &animation)

  Animates the player based on their velocity.
- void animateBasicMonster (std::optional< AnimationComponent * > &animation)

  Animates a basic monster entity.
- void animateForceWeapon (std::optional< ForceWeaponComponent * > &forceWeapon, std::optional< AnimationComponent * > &animation, std::optional< HitboxComponent * > &hitbox)

  Animates the force weapon based on its current state.
- void animateForceMissile (std::optional< ForceWeaponComponent * > &forceWeapon, std::optional< AnimationComponent * > &animation, std::optional< HitboxComponent * > &hitbox)

  Animates the force missile based on the provided components.
- void animateBoss (std::optional< BossComponent * > &boss, std::optional< AnimationComponent * > &animation)

Public Member Functions inherited from ISystem

- ISystem ()=default
- virtual ~ISystem ()=default

Private Attributes

- ComponentManager & _componentManager

  Reference to the ComponentManager instance.
- EntityManager & _entityManager

  Reference to the EntityManager instance.

6.6.1 Detailed Description

A system responsible for animating entities within the ECS framework.

The AnimationSystem class provides functionality to animate various entities such as players, basic monsters, force weapons, and force missiles. It interacts with the ComponentManager and EntityManager to access and update the relevant components required for animation.

The AnimationSystem class inherits from the ISystem interface and implements several methods to handle the animation of different types of entities. It processes entities based on their animation components and updates their animation states according to the provided delta time or specific component states.

Note

This class assumes the presence of specific components such as VelocityComponent, AnimationComponent, and ForceWeaponComponent to perform the animations. The methods use optional pointers to these components to ensure that animations are only performed when the components are available.

See also

ISystem

ComponentManager

EntityManager

6.6.2 Constructor & Destructor Documentation

6.6.2.1 AnimationSystem()

```
AnimationSystem::AnimationSystem (
            ComponentManager & componentManager,
            EntityManager & entityManager )  [inline]
```

6.6.3 Member Function Documentation

6.6.3.1 animateBasicMonster()

```
void AnimationSystem::animateBasicMonster (
            std::optional< AnimationComponent * > & animation )
```

Animates a basic monster entity.

This function updates the animation state of a basic monster entity based on the provided AnimationComponent. The animation state is modified to reflect the current frame or sequence in the animation.

Parameters

animation	An optional pointer to the AnimationComponent associated with the basic monster entity. If the optional is empty, no animation will be performed.

6.6.3.2 animateBoss()

```
void AnimationSystem::animateBoss (
            std::optional< BossComponent * > & boss,
            std::optional< AnimationComponent * > & animation )
```

6.6.3.3 animateForceMissile()

```
void AnimationSystem::animateForceMissile (
            std::optional< ForceWeaponComponent * > & forceWeapon,
            std::optional< AnimationComponent * > & animation,
            std::optional< HitboxComponent * > & hitbox )
```

Animates the force missile based on the provided components.

This function updates the animation state of a force missile using the provided ForceWeaponComponent and AnimationComponent. The function ensures that the animation reflects the current state of the force missile.

Parameters

forceWeapon	An optional reference to a ForceWeaponComponent pointer. This component contains the state and properties of the force missile weapon.
animation	An optional reference to an AnimationComponent pointer. This component handles the animation state and frames for the force missile.

6.6.3.4 animateForceWeapon()

```
void AnimationSystem::animateForceWeapon (
            std::optional< ForceWeaponComponent * > & forceWeapon,
            std::optional< AnimationComponent * > & animation,
            std::optional< HitboxComponent * > & hitbox )
```

Animates the force weapon based on its current state.

This function updates the animation of the force weapon component by modifying the associated animation component.

Parameters

forceWeapon	An optional reference to the ForceWeaponComponent.
animation	An optional reference to the AnimationComponent.

6.6.3.5 animatePlayer()

```
void AnimationSystem::animatePlayer (
            std::optional< VelocityComponent * > & velocity,
            std::optional< AnimationComponent * > & animation )
```

Animates the player based on their velocity.

This function updates the player's animation state according to the provided velocity component. If the velocity component indicates movement, the animation component will be updated to reflect the corresponding animation state.

Parameters

velocity	A reference to an optional VelocityComponent pointer. If the pointer is present, it contains the player's current velocity.
animation	A reference to an optional AnimationComponent pointer. If the pointer is present, it contains the player's current animation state.

6.6.3.6 AnimationEntities()

```
void AnimationSystem::AnimationEntities (
            ComponentManager & componentManager,
            EntityManager & entityManager,
            float deltaTime,
            bool & endOfLevel )
```

Animates entities.

Updates the animation states of entities based on their components.

This function animates entities based on their animation components. It processes each entity in the entity manager and updates their animation based on the delta time provided.

Parameters

componentManager	The component manager used to access entity components.
entityManager	The entity manager used to access entities.
deltaTime	The time elapsed since the last update, used to update animations.

This function iterates through all entities and updates their animation states based on the presence and values of specific components such as AnimationComponent, PlayerComponent, VelocityComponent, and BackgroundComponent.

Parameters

componentManager	Reference to the ComponentManager that handles components.
entityManager	Reference to the EntityManager that handles entities.
deltaTime	The time elapsed since the last update, used for time-based animations.

6.6.4 Member Data Documentation

6.6.4.1 _componentManager

```
ComponentManager& AnimationSystem::_componentManager [private]
```

Reference to the ComponentManager instance.

This member variable holds a reference to the ComponentManager, which is responsible for managing all the components within the ECS (Entity Component System). It provides functionality to add, remove, and query components associated with entities.

6.6.4.2 _entityManager

```
EntityManager& AnimationSystem::_entityManager [private]
```

Reference to the EntityManager instance.

This member variable holds a reference to the EntityManager, which is responsible for managing all entities within the ECS (Entity Component System). It provides functionalities such as entity creation, deletion, and retrieval.

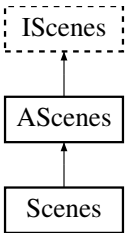
The documentation for this class was generated from the following files:

- /home/runner/work/R-Type/R-Type/Server/Interface/Include/animation_system.hpp
- /home/runner/work/R-Type/R-Type/Server/Src/animation_system.cpp

6.7 AScenes Class Reference

```
#include <a_scenes.hpp>
```

Inheritance diagram for AScenes:



Public Types

- enum class Scene {
  MAIN_MENU , GAME_LOOP , SETTINGS_MENU , IN_GAME_MENU ,
  CHOOSE_DIFFICULTY , CUSTOM_DIFFICULTY , TRANSITION_LEVEL , EXIT }

  Represents the different scenes in the R-Type client application.
- enum class GameMode { EASY , MEDIUM , HARD }

  Enumeration to represent different game difficulty levels.
- enum class DaltonismMode { NORMAL , TRITANOPIA , DEUTERANOPIA , PROTANOPIA }

  Enum representing different modes of color blindness (Daltonism).
- enum class Actions {
  UP , DOWN , LEFT , RIGHT ,
  FIRE , PAUSE , QUIT }

  Enumeration representing possible actions in the game.
- enum class SpriteType {
  BACKGROUND , PLAYER , ALLY , ENEMY ,
  FILTER , WEAPON , POWER_UP , UI ,
  OTHER }

  Enumeration representing the type of sprite in the game.

Public Member Functions

- AScenes (std::string ip, int port)
- ~AScenes ()=default
- void setScene (Scene scene)

  Set the Scene object.
- AScenes::Scene getPreviousScene ()

  Get the Previous Scene object.
- DaltonismMode getDaltonism () const

  Get the Daltonism object.
- void setDaltonism (DaltonismMode const mode)

  Set the Daltonism object.
- void setGameMode (GameParameters const mode)

  Set the Game Mode object.
- GameParameters getGameMode () const

  Get the Game Mode object.
- void setDisplayDaltonismChoice (bool const displayDaltonismChoice)

  Sets the display option for Daltonism mode.
- bool getDisplayDaltonismChoice () const

  Retrieves the current display setting for Daltonism (color blindness) mode.
- void setDisplayGameModeChoice (bool const displayGameModeChoice)

  Sets the display state for the game mode choice.
- bool getDisplayGameModeChoice () const

  Retrieves the current display game mode choice.
- void setDisplayKeyBindsChoice (bool const displayKeyBindsChoice)

  Sets the display status for key binds choice.
- bool getDisplayKeyBindsChoice () const

  Retrieves the current choice for displaying key bindings.
- void setIp (std::string ip)

  Sets the IP address.
- void setPort (int port)

  Sets the port number for the connection.
- std::string getIp () const

  Retrieves the IP address.
- int getPort () const

  Retrieves the port number.
- void SetPlayerReady (bool ready)

  Sets the player ready status.
- bool GetPlayerReady () const

  Retrieves the player ready status.

Public Member Functions inherited from IScenes

- virtual ~IScenes ()=default
- virtual void mainMenu ()=0

  Displays the main menu and creates necessary entities.
- virtual void gameLoop ()=0

  Displays the main game loop and creates necessary entities.
- virtual void settingsMenu ()=0

  Displays the settings menu and creates necessary entities.

- virtual void inGameMenu ()=0

  Displays the in-game menu and creates necessary entities.
- virtual void difficultyChoices ()=0

  Displays the difficulty choices.
- virtual void render ()=0

  Displays the current scene and manages its components.
- virtual bool shouldQuit ()=0

  Checks if the game should quit.
- virtual sf::RenderWindow * getRenderWindow ()=0

  Gets the render window.

Public Attributes

- std::map< Actions, sf::Keyboard::Key > keyBinds

  A map that binds game actions to specific keyboard keys.
- std::vector< std::shared_ptr< Entity > > buttons

  A collection of shared pointers to Entity objects representing buttons.
- std::shared_ptr< Entity > filter

  A shared pointer to an Entity object.

Protected Attributes

- GameParameters _currentGameMode

  Represents the current game mode.
- DaltonismMode _currentDaltonismMode = DaltonismMode::NORMAL

  Enum representing different modes of Daltonism (color blindness).
- Scene _currentScene = Scene::MAIN_MENU

  Represents the current scene in the application.
- Scene _previousScene = Scene::MAIN_MENU

  Represents the previous scene in the application.
- bool _displayDaltonismChoice = false

  Flag to indicate whether the Daltonism choice should be displayed.
- bool _displayGameModeChoice = false

  Flag indicating whether the game mode choice should be displayed.
- bool _displayKeyBindsChoice = false

  Flag indicating whether the key bindings choice should be displayed.
- std::string _ip

  The IP address of the server.
- int _port

  The port number of the server.
- bool _playerReady = false

6.7.1 Member Enumeration Documentation

6.7.1.1 Actions

```
enum class AScenes::Actions [strong]
```

Enumeration representing possible actions in the game.

This enumeration defines the various actions that can be performed by the player in the game. The actions include:

- UP: Move up
- DOWN: Move down
- LEFT: Move left
- RIGHT: Move right
- FIRE: Fire a weapon
- PAUSE: Pause the game
- QUIT: Quit the game

Enumerator

UP	
DOWN	
LEFT	
RIGHT	
FIRE	
PAUSE	
QUIT	

6.7.1.2 DaltonismMode

```
enum class AScenes::DaltonismMode [strong]
```

Enum representing different modes of color blindness (Daltonism).

This enum is used to specify the type of color blindness mode that can be applied.

Enumerator

NORMAL	Represents normal vision without any color blindness.
TRITANOPIA	Represents Tritanopia, a type of color blindness where blue and yellow colors are confused.
DEUTERANOPIA	Represents Deuteranopia, a type of color blindness where green and red colors are confused.
PROTANOPIA	Represents Protanopia, a type of color blindness where red and green colors are confused.

6.7.1.3 GameMode

```
enum class AScenes::GameMode [strong]
```

Enumeration to represent different game difficulty levels.

This enumeration defines the various difficulty levels that can be selected in the game. The available modes are:

- EASY: Represents an easy difficulty level.
- MEDIUM: Represents a medium difficulty level.
- HARD: Represents a hard difficulty level.

Enumerator

EASY	
MEDIUM	
HARD	

6.7.1.4 Scene

```
enum class AScenes::Scene [strong]
```

Represents the different scenes in the R-Type client application.

This enumeration defines the various scenes that the client can be in during its lifecycle.

Enumerator

MAIN_MENU	Represents the main menu scene.
GAME_LOOP	Represents the game loop scene where the main gameplay occurs.
SETTINGS_MENU	Represents the settings menu scene where the user can adjust settings.
IN_GAME_MENU	Represents the in-game menu scene that can be accessed during gameplay.
CHOOSE_DIFFICULTY	
CUSTOM_DIFFICULTY	
TRANSITION_LEVEL	
EXIT	Represents the exit scene where the application is closing.

6.7.1.5 SpriteType

```
enum class AScenes::SpriteType [strong]
```

Enumeration representing the type of sprite in the game.

This enumeration defines the different sprite types that need to be identified in the game. The types include:

- BACKGROUND: Represents a background sprite.

- PLAYER: Represents a player sprite.
- ALLY: Represents an ally sprite.
- ENEMY: Represents an enemy sprite.
- OTHER: Represents any other type of sprite.

Enumerator

BACKGROUND	
PLAYER	
ALLY	
ENEMY	
FILTER	
WEAPON	
POWER_UP	
UI	
OTHER	

6.7.2 Constructor & Destructor Documentation

6.7.2.1 AScenes()

```
AScenes::AScenes (
            std::string ip,
            int port )
```

6.7.2.2 ~AScenes()

```
AScenes::~AScenes ( ) [default]
```

6.7.3 Member Function Documentation

6.7.3.1 getDaltonism()

```
DaltonismMode AScenes::getDaltonism ( ) const [inline]
```

Get the Daltonism object.

Returns

DaltonismMode

6.7.3.2 getDisplayDaltonismChoice()

```
bool AScenes::getDisplayDaltonismChoice ( ) const
```

Retrieves the current display setting for Daltonism (color blindness) mode.

Returns

true if Daltonism mode is enabled, false otherwise.

6.7.3.3 getDisplayGameModeChoice()

```
bool AScenes::getDisplayGameModeChoice ( ) const
```

Retrieves the current display game mode choice.

This function returns a boolean value indicating whether the game mode choice is currently set to be displayed.

Returns

true if the game mode choice is set to be displayed, false otherwise.

6.7.3.4 getDisplayKeyBindsChoice()

```
bool AScenes::getDisplayKeyBindsChoice ( ) const
```

Retrieves the current choice for displaying key bindings.

Returns

true if key bindings should be displayed, false otherwise.

6.7.3.5 getGameMode()

```
GameParameters AScenes::getGameMode ( ) const  [inline]
```

Get the Game Mode object.

Returns

GameParameters

6.7.3.6 getIp()

```
std::string AScenes::getIp ( ) const
```

Retrieves the IP address.

This function returns the IP address as a string.

Returns

std::string The IP address.

6.7.3.7 GetPlayerReady()

```
bool AScenes::GetPlayerReady ( ) const  [inline]
```

Retrieves the player ready status.

This function returns the player ready status.

Returns

bool The player ready status.

6.7.3.8 getPort()

```
int AScenes::getPort ( ) const
```

Retrieves the port number.

Returns

int The port number.

6.7.3.9 getPreviousScene()

```
AScenes::Scene AScenes::getPreviousScene ( )
```

Get the Previous Scene object.

Returns

Scene

6.7.3.10 setDaltonism()

```
void AScenes::setDaltonism (
            DaltonismMode const mode )
```

Set the Daltonism object.

Parameters

mode	The daltonism mode to set

6.7.3.11 setDisplayDaltonismChoice()

```
void AScenes::setDisplayDaltonismChoice (
            bool const displayDaltonismChoice )
```

Sets the display option for Daltonism mode.

This function enables or disables the display option for Daltonism mode based on the provided boolean value.

Parameters

displayDaltonismChoice	A boolean value indicating whether to display the Daltonism mode option (true) or not (false).

6.7.3.12 setDisplayGameModeChoice()

```
void AScenes::setDisplayGameModeChoice (
            bool const displayGameModeChoice )
```

Sets the display state for the game mode choice.

This function allows you to control whether the game mode choice should be displayed or not.

Parameters

displayGameModeChoice	A boolean value indicating whether the game mode choice should be displayed (true) or hidden (false).

6.7.3.13 setDisplayKeyBindsChoice()

```
void AScenes::setDisplayKeyBindsChoice (
            bool const displayKeyBindsChoice )
```

Sets the display status for key binds choice.

This function allows you to enable or disable the display of key binds choice.

Parameters

displayKeyBindsChoice	A boolean value indicating whether to display the key binds choice (true) or not (false).

6.7.3.14 setGameMode()

```
void AScenes::setGameMode (
            GameParameters const mode )
```

Set the Game Mode object.

Parameters

mode	

6.7.3.15 setIp()

```
void AScenes::setIp (
            std::string ip )
```

Sets the IP address.

This function sets the IP address to the specified value.

Parameters

ip	The IP address to set as a string.

6.7.3.16 SetPlayerReady()

```
void AScenes::SetPlayerReady (
            bool ready )  [inline]
```

Sets the player ready status.

This function sets the player ready status to the specified value.

Parameters

ready	The player ready status to set.

6.7.3.17 setPort()

```
void AScenes::setPort (
            int port )
```

Sets the port number for the connection.

This function assigns the specified port number to be used for network communication.

Parameters

port	The port number to be set.

6.7.3.18 setScene()

```
void AScenes::setScene (
            AScenes::Scene scene )
```

Set the Scene object.

Parameters

scene	

6.7.4 Member Data Documentation

6.7.4.1 _currentDaltonismMode

```
DaltonismMode AScenes::_currentDaltonismMode = DaltonismMode::NORMAL  [protected]
```

Enum representing different modes of Daltonism (color blindness).

This enum is used to specify the current Daltonism mode, which can be used to adjust the display settings for users with different types of color blindness.

Possible values:

- NORMAL: No color blindness.
- PROTANOPIA: Red color blindness.
- DEUTERANOPIA: Green color blindness.
- TRITANOPIA: Blue color blindness.

6.7.4.2 _currentGameMode

```
GameParameters AScenes::_currentGameMode  [protected]
```

Represents the current game mode.

This variable holds the current game mode of the game. It is initialized to GameMode::MEDIUM by default.

6.7.4.3 _currentScene

```
Scene AScenes::_currentScene = Scene::MAIN_MENU  [protected]
```

Represents the current scene in the application.

This variable holds the current scene being displayed or interacted with in the application. It is initialized to the MAIN_MENU scene by default.

6.7.4.4 _displayDaltonismChoice

```
bool AScenes::_displayDaltonismChoice = false  [protected]
```

Flag to indicate whether the Daltonism choice should be displayed.

6.7.4.5 _displayGameModeChoice

```
bool AScenes::_displayGameModeChoice = false  [protected]
```

Flag indicating whether the game mode choice should be displayed.

6.7.4.6 _displayKeyBindsChoice

```
bool AScenes::_displayKeyBindsChoice = false  [protected]
```

Flag indicating whether the key bindings choice should be displayed.

6.7.4.7 _ip

```
std::string AScenes::_ip  [protected]
```

The IP address of the server.

This member variable stores the IP address of the server to which the client will connect. It is a string that contains the IP address in the format "xxx.xxx.xxx.xxx".

6.7.4.8 _playerReady

```
bool AScenes::_playerReady = false  [protected]
```

6.7.4.9 _port

```
int AScenes::_port  [protected]
```

The port number of the server.

This member variable stores the port number of the server to which the client will connect. It is an integer that represents the port number on which the server is listening for incoming connections.

6.7.4.10 _previousScene

```
Scene AScenes::_previousScene = Scene::MAIN_MENU  [protected]
```

Represents the previous scene in the application.

This variable holds the previous scene that was active before the current one. It is initialized to the MAIN_MENU scene by default.

6.7.4.11 buttons

```
std::vector<std::shared_ptr<Entity> > AScenes::buttons
```

A collection of shared pointers to Entity objects representing buttons.

This vector holds shared pointers to Entity instances, which are used to represent buttons within the scene. The use of shared pointers ensures that the Entity objects are properly managed and their memory is automatically deallocated when they are no longer in use.

6.7.4.12 filter

```
std::shared_ptr<Entity> AScenes::filter
```

A shared pointer to an Entity object.

This smart pointer manages the lifetime of an Entity instance, ensuring that the Entity is properly deleted when no longer in use. It allows multiple parts of the program to share ownership of the Entity.

6.7.4.13 keyBinds

```
std::map<Actions, sf::Keyboard::Key> AScenes::keyBinds
```

Initial value:

```
= {{Actions::UP, sf::Keyboard::Key::Up},
        {Actions::DOWN, sf::Keyboard::Key::Down}, {Actions::LEFT, sf::Keyboard::Key::Left},
        {Actions::RIGHT, sf::Keyboard::Key::Right}, {Actions::FIRE, sf::Keyboard::Key::Space},
        {Actions::PAUSE, sf::Keyboard::Key::Escape}, {Actions::QUIT, sf::Keyboard::Key::Q}}
```

A map that binds game actions to specific keyboard keys.

This map associates each action defined in the Actions enum with a corresponding key from the sf::Keyboard::Key enumeration. It is used to handle user input by mapping key presses to game actions.

The key bindings are as follows:

- Actions::UP -> sf::Keyboard::Key::Up
- Actions::DOWN -> sf::Keyboard::Key::Down
- Actions::LEFT -> sf::Keyboard::Key::Left
- Actions::RIGHT -> sf::Keyboard::Key::Right
- Actions::FIRE -> sf::Keyboard::Key::Space
- Actions::PAUSE -> sf::Keyboard::Key::Escape
- Actions::QUIT -> sf::Keyboard::Key::Q

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

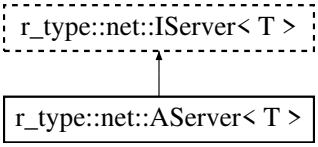
- /home/runner/work/R-Type/R-Type/ECS/Interface/Include/a_scenes.hpp
- /home/runner/work/R-Type/R-Type/ECS/Src/a_scenes.cpp

6.8 r_type::net::AServer< T > Class Template Reference

AServer class template for managing server operations.

```
#include <a_server.hpp>
```

Inheritance diagram for r_type::net::AServer< T >:



Public Member Functions

- AServer (uint16_t port)

  Constructs an AServer object with the specified port.
- ~AServer ()

  Destructor for the AServer class.
- bool Start ()

  Starts the server and begins waiting for client messages.
- void Stop ()

  Stops the server.
- void WaitForClientMessage ()

  Waits for a client message asynchronously.
- void MessageClient (std::shared_ptr< Connection< T > > client, const Message< T > &msg)

  Sends a message to a specific client if the client is connected.
- void MessageAllClients (const Message< T > &msg, std::shared_ptr< Connection< T > > pIgnoreClient=nullptr)

  Sends a message to all connected clients, optionally ignoring a specified client.
- UIEntityInformation UpdateInfoBar (int playerId)

  Updates the information bar for a given player.
- void Update (size_t nMaxMessages=-1, bool bWait=false)

  Updates the server state, processes incoming messages, and updates the game level.
- void UpdatePlayerPosition (PlayerMovement direction, uint32_t entityId) override

  Updates the position of an entity based on the message received and the client ID.
- void SavePlayerScore (uint32_t playerId)

  Saves the score of a player to a file.
- uint32_t GetClientPlayerId (uint32_t id)

  Retrieves the entity ID associated with a client ID.
- uint32_t GetPlayerClientId (uint32_t id)

  Retrieves the client ID associated with a given player ID.
- uint32_t GetClientInfoBarId (uint32_t id)

  Retrieves the client info bar ID associated with a given client ID.
- void RemovePlayer (uint32_t id)

  Removes a player from the server.
- void RemoveEntity (uint32_t id)

  Removes an entity from the server.
- void RemoveInfoBar (uint32_t infoBarId)

  Removes an information bar and its associated entities.

- void RemoveBossTail (int bossId)
- EntityInformation InitiatePlayer (int clientId)

  Initializes a new player entity and assigns a random position.
- UIEntityInformation InitInfoBar (int clientId)
- EntityInformation FormatEntityInformation (uint32_t entityId)

  Formats the information of a given entity into an EntityInformation structure.
- EntityInformation InitiatePlayerMissile (int entityId)

  Initializes a missile entity associated with a player.
- EntityInformation InitiateEnemyMissile (int enemyId)
- EntityInformation InitiateWeaponForce (int entityId)
- void InitBoss (r_type::net::AServer< T > *server)
- std::shared_ptr< Connection< T > > getClientById (const std::deque< std::shared_ptr< Connection< T > > > &connections, uint32_t clientId)
- virtual void OnClientValidated (std::shared_ptr< Connection< T > > client)

  Callback function that is called when a client has been successfully validated.
- ComponentManager GetComponentManager () override

  Retrieves the component manager associated with the server.
- EntityManager & GetEntityManager () override

  Retrieves the entity manager associated with the server.
- EntityFactory & GetEntityFactory () override

  Retrieves the entity factory associated with the server.
- std::chrono::system_clock::time_point GetClock () override

  Retrieves the current clock time of the server.
- void SetClock (std::chrono::system_clock::time_point clock)

  Set the Clock object.

Public Attributes

- ThreadSafeQueue< OwnedMessage< T > > _qMessagesIn

  Thread-safe queue to store incoming messages.
- std::deque< std::shared_ptr< Connection< T > > > _deqConnections

  A deque that holds shared pointers to Connection objects.
- asio::io_context _asioContext

  The io_context object provides I/O services, such as sockets, that the server will use.
- std::thread _threadContext

  Thread object for managing the server's context operations.
- asio::ip::udp::socket _asioSocket

  A socket for sending and receiving UDP datagrams.
- asio::ip::udp::endpoint _clientEndpoint

  Represents the endpoint of a client in a UDP connection.
- std::array< uint8_t, 1024 > _tempBuffer

  Temporary buffer used for storing data.
- uint32_t _nIDCounter = 10000

  Counter for generating unique network IDs.
- ComponentManager _componentManager

  Manages and maintains the lifecycle of various components within the server.
- EntityManager _entityManager

  Manages the lifecycle and operations of entities within the server.
- EntityFactory _entityFactory

  An instance of EntityFactory used to create and manage game entities.
- bool _endOfLevel = false

- bool _bossActive = false
- bool _bossDefeated = false
- bool _watingPlayersReady = false
- std::unordered_map< uint32_t, uint32_t > _clientPlayerID

  A container that maps client IDs to player IDs.
- std::unordered_map< uint32_t, uint32_t > _clientInfoBarID
- int _nbrOfPlayers = 0

  Number of players currently connected to the server.
- std::chrono::system_clock::time_point _clock = std::chrono::system_clock::now()

  Stores the current time point from the system clock.
- bool _playerConnected = false
- EntityInformation _background

  Holds information about the background entity.
- int _port
- r_type::Level< T > _level
- int _playerReady = 0

Protected Member Functions

- virtual bool OnClientConnect (std::shared_ptr< Connection< T > > client)

  on client connect event
- virtual void OnClientDisconnect (std::shared_ptr< Connection< T > > client)

  on client disconnect event
- virtual void OnMessage (std::shared_ptr< Connection< T > > client, Message< T > &msg)

  on message event

6.8.1 Detailed Description

template<typename T>
class r_type::net::AServer< T >

AServer class template for managing server operations.

This class template provides a framework for creating and managing a server that handles client connections, messages, and entity updates. It uses the ASIO library for asynchronous network communication and provides various functions for server operations such as starting, stopping, and updating the server, as well as handling client messages and connections.

Template Parameters

T	The type of data that the server handles.

6.8.2 Constructor & Destructor Documentation

6.8.2.1 AServer()

```
template<typename T >
r_type::net::AServer< T >::AServer (
            uint16_t port )  [inline]
```

Constructs an AServer object with the specified port.

This constructor initializes the server with the given port number and sets up the necessary components for the server to function. It initializes the ASIO socket with the provided port and creates instances of EntityManager, EntityFactory, and ComponentManager. Additionally, it initiates the background process and creates three basic monster entities using the entity factory.

Parameters

port	The port number on which the server will listen for incoming connections.

6.8.2.2 ~AServer()

```
template<typename T >
r_type::net::AServer< T >::~AServer ( ) [inline]
```

Destructor for the AServer class.

This destructor ensures that the server is properly stopped by calling the Stop() method when an instance of AServer is destroyed.

6.8.3 Member Function Documentation

6.8.3.1 FormatEntityInformation()

```
template<typename T >
EntityInformation r_type::net::AServer< T >::FormatEntityInformation (
            uint32_t entityId ) [inline]
```

Formats the information of a given entity into an EntityInformation structure.

This function retrieves the position and sprite data components of the specified entity and populates an EntityInformation structure with this data. If the entity has both position and sprite data components, their values are copied into the EntityInformation structure. If either component is missing, the EntityInformation structure will be returned with default values.

Parameters

entity	The entity whose information is to be formatted.

Returns

EntityInformation The formatted information of the entity.

6.8.3.2 getClientById()

```
template<typename T >
std::shared_ptr< Connection< T > > r_type::net::AServer< T >::getClientById (
            const std::deque< std::shared_ptr< Connection< T > > > & connections,
            uint32_t clientId ) [inline]
```

6.8.3.3 GetClientInfoBarId()

```
template<typename T >
uint32_t r_type::net::AServer< T >::GetClientInfoBarId (
            uint32_t id )  [inline]
```

Retrieves the client info bar ID associated with a given client ID.

Parameters

id	The client ID for which to retrieve the info bar ID.

Returns

uint32_t The info bar ID associated with the specified client ID.

6.8.3.4 GetClientPlayerId()

```
template<typename T >
uint32_t r_type::net::AServer< T >::GetClientPlayerId (
            uint32_t id )  [inline]
```

Retrieves the entity ID associated with a client ID.

Parameters

id	The client ID.

Returns

uint32_t The entity ID associated with the client.

6.8.3.5 GetClock()

```
template<typename T >
std::chrono::system_clock::time_point r_type::net::AServer< T >::GetClock ( )  [inline], [override]
```

Retrieves the current clock time of the server.

This function returns the current time point of the server's clock, which can be used for time-related calculations, such as updating game state, handling animations, or scheduling events. It provides a consistent reference point for the server's operations.

Returns

std::chrono::system_clock::time_point The current time point of the server's clock.

6.8.3.6 GetComponentManager()

```
template<typename T >
ComponentManager r_type::net::AServer< T >::GetComponentManager ( ) [inline], [override]
```

Retrieves the component manager associated with the server.

This function provides access to the component manager, which is responsible for managing the components associated with entities in the game. It allows for the retrieval and manipulation of entity components, enabling the game logic to interact with them as needed.

Returns

ComponentManager& A reference to the component manager instance.

6.8.3.7 GetEntityFactory()

```
template<typename T >
EntityFactory & r_type::net::AServer< T >::GetEntityFactory ( ) [inline], [override]
```

Retrieves the entity factory associated with the server.

This function provides access to the entity factory, which is responsible for creating new entities in the game. The entity factory provides methods to instantiate various types of entities, such as players, missiles, and background elements, ensuring that they are correctly initialized with the necessary components.

Returns

EntityFactory& A reference to the entity factory instance.

6.8.3.8 GetEntityManager()

```
template<typename T >
EntityManager & r_type::net::AServer< T >::GetEntityManager ( ) [inline], [override]
```

Retrieves the entity manager associated with the server.

This function returns the entity manager responsible for creating, managing, and removing entities in the game. The entity manager handles the lifecycle of entities and ensures that they are correctly processed within the game's systems.

Returns

EntityManager& A reference to the entity manager instance.

6.8.3.9 GetPlayerClientId()

```
template<typename T >
uint32_t r_type::net::AServer< T >::GetPlayerClientId (
            uint32_t id ) [inline]
```

Retrieves the client ID associated with a given player ID.

This function searches through the _clientPlayerID map to find the client ID that corresponds to the provided player ID. If the player ID is found, the associated client ID is returned. If the player ID is not found, a playerIdNotFound exception is thrown.

Parameters

id	The player ID for which the client ID is to be retrieved.

Returns

uint32_t The client ID associated with the given player ID.

Exceptions

playerIdNotFound	If the player ID is not found in the map.

6.8.3.10 InitBoss()

```
template<typename T >
void r_type::net::AServer< T >::InitBoss (
            r_type::net::AServer< T > * server )  [inline]
```

6.8.3.11 InitiateEnemyMissile()

```
template<typename T >
EntityInformation r_type::net::AServer< T >::InitiateEnemyMissile (
            int enemyId )  [inline]
```

6.8.3.12 InitiatePlayer()

```
template<typename T >
EntityInformation r_type::net::AServer< T >::InitiatePlayer (
            int clientId )  [inline]
```

Initializes a new player entity and assigns a random position.

The function creates a new player entity, assigns it a random position, and ensures that it does not overlap with any other players.

Parameters

client↩ Id	The client ID of the player being initialized.

Returns

EntityInformation The information of the newly created player entity.

6.8.3.13 InitiatePlayerMissile()

```
template<typename T >
```

```
EntityInformation r_type::net::AServer< T >::InitiatePlayerMissile (
            int entityId )  [inline]
```

Initializes a missile entity associated with a player.

The function creates a missile entity associated with a player and assigns its position based on the player's current position.

Parameters

clientId	The client ID of the player firing the missile.

Returns

EntityInformation The information of the newly created missile entity.

6.8.3.14 InitiateWeaponForce()

```
template<typename T >
EntityInformation r_type::net::AServer< T >::InitiateWeaponForce (
            int entityId )  [inline]
```

6.8.3.15 InitInfoBar()

```
template<typename T >
UIEntityInformation r_type::net::AServer< T >::InitInfoBar (
            int clientId )  [inline]
```

6.8.3.16 MessageAllClients()

```
template<typename T >
void r_type::net::AServer< T >::MessageAllClients (
            const Message< T > & msg,
            std::shared_ptr< Connection< T > > pIgnoreClient = nullptr )  [inline]
```

Sends a message to all connected clients, optionally ignoring a specified client.

This function iterates through all the connections in the server and sends the provided message to each connected client, except for the client specified by `pIgnoreClient`. If a client is found to be disconnected, it triggers the disconnection handler and removes the client from the list of connections.

Template Parameters

T	The type of the message.

Parameters

msg	The message to be sent to all clients.

Parameters

pIgnoreClient	A shared pointer to a client connection that should be ignored. Defaults to nullptr.

6.8.3.17 MessageClient()

```
template<typename T >
void r_type::net::AServer< T >::MessageClient (
            std::shared_ptr< Connection< T > > client,
            const Message< T > & msg )  [inline]
```

Sends a message to a specific client if the client is connected.

If the client is not connected, it handles the client disconnection.

Template Parameters

T	The type of the message.

Parameters

client	A shared pointer to the client connection.
msg	The message to be sent to the client.

6.8.3.18 OnClientConnect()

```
template<typename T >
virtual bool r_type::net::AServer< T >::OnClientConnect (
            std::shared_ptr< Connection< T > > client )  [inline], [protected], [virtual]
```

on client connect event

Parameters

client	

Returns

true

false

6.8.3.19 OnClientDisconnect()

```
template<typename T >
virtual void r_type::net::AServer< T >::OnClientDisconnect (
            std::shared_ptr< Connection< T > > client )  [inline], [protected], [virtual]
```

on client disconnect event

Parameters

client	

6.8.3.20 OnClientValidated()

```
template<typename T >
virtual void r_type::net::AServer< T >::OnClientValidated (
            std::shared_ptr< Connection< T > > client )  [inline], [virtual]
```

Callback function that is called when a client has been successfully validated.

This function is intended to be overridden by derived classes to handle any specific actions that need to be taken when a client is validated.

Parameters

client	A shared pointer to the validated client connection.

6.8.3.21 OnMessage()

```
template<typename T >
virtual void r_type::net::AServer< T >::OnMessage (
            std::shared_ptr< Connection< T > > client,
            Message< T > & msg )  [inline], [protected], [virtual]
```

on message event

Parameters

client	
msg	

6.8.3.22 RemoveBossTail()

```
template<typename T >
void r_type::net::AServer< T >::RemoveBossTail (
            int bossId )  [inline]
```

6.8.3.23 RemoveEntity()

```
template<typename T >
void r_type::net::AServer< T >::RemoveEntity (
            uint32_t id )  [inline]
```

Removes an entity from the server.

This function removes an entity identified by the given ID from the server. It first checks if the entity exists using the entity manager. If the entity is found, it removes the entity from all components using the component manager and then removes the entity itself from the entity manager.

Parameters

id	The unique identifier of the entity to be removed.

6.8.3.24 RemoveInfoBar()

```
template<typename T >
void r_type::net::AServer< T >::RemoveInfoBar (
            uint32_t infoBarId )  [inline]
```

Removes an information bar and its associated entities.

This function removes an information bar identified by the given `infoBarId`. It first checks if the information bar has a `TextDataComponent` and removes all entities associated with the categories listed in the `TextDataComponent`. Finally, it removes the information bar entity itself and erases its ID from the client information bar ID map.

Parameters

infoBarId	The ID of the information bar to be removed.

6.8.3.25 RemovePlayer()

```
template<typename T >
void r_type::net::AServer< T >::RemovePlayer (
            uint32_t id )  [inline]
```

Removes a player from the server.

This function removes a player identified by the given ID from the server's internal player list.

Parameters

id	The unique identifier of the player to be removed.

6.8.3.26 SavePlayerScore()

```
template<typename T >
void r_type::net::AServer< T >::SavePlayerScore (
            uint32_t playerId )  [inline]
```

Saves the score of a player to a file.

This function saves the score of a player identified by the given playerId to a file named "scores.txt" located in the "GameScores" directory. If the directory or file does not exist, they will be created. The score is appended to the file in the format "Player <playerId>: <score>".

Parameters

playerId	The unique identifier of the player whose score is to be saved.

Exceptions

failedToCreateFile	If the file cannot be created.
failedToOpenFile	If the file cannot be opened in append mode.

6.8.3.27 SetClock()

```
template<typename T >
void r_type::net::AServer< T >::SetClock (
            std::chrono::system_clock::time_point clock )  [inline]
```

Set the Clock object.

Parameters

clock	

6.8.3.28 Start()

```
template<typename T >
bool r_type::net::AServer< T >::Start ( )  [inline]
```

Starts the server and begins waiting for client messages.

This function attempts to start the server by waiting for client messages and running the ASIO context in a separate thread. If an exception occurs during this process, it will be caught, an error message will be printed to the standard error stream, and the function will return false.

Returns

true if the server started successfully, false otherwise.

6.8.3.29 Stop()

```
template<typename T >
void r_type::net::AServer< T >::Stop ( )  [inline]
```

Stops the server.

This function stops the server by stopping the ASIO context and joining the thread context. It also prints a message indicating that the server has been stopped.

6.8.3.30 Update()

```
template<typename T >
void r_type::net::AServer< T >::Update (
            size_t nMaxMessages = -1,
            bool bWait = false )  [inline]
```

Updates the server state, processes incoming messages, and updates the game level.

This function performs several tasks:

- If no players are connected, it returns immediately.
- If players are connected and the player connection flag is not set, it sets the flag and updates the clock.
- Spawns a thread to update the game level.
- Processes up to nMaxMessages from the incoming message queue.
- Joins the level update thread and updates the clock if entities were updated.

Parameters

nMaxMessages	The maximum number of messages to process from the incoming message queue. Default is -1 (process all messages).
bWait	A flag indicating whether to wait for messages. Default is false.

6.8.3.31 UpdateInfoBar()

```
template<typename T >
UIEntityInformation r_type::net::AServer< T >::UpdateInfoBar (
            int playerId )  [inline]
```

Updates the information bar for a given player.

This function retrieves the health and score components of the specified player, as well as the sprite and text data components of the player's information bar. It then updates the UIEntityInformation structure with these values.

Parameters

playerId	The ID of the player whose information bar is to be updated.

Returns

UIEntityInformation The updated information for the player's information bar.

6.8.3.32 UpdatePlayerPosition()

```
template<typename T >
void r_type::net::AServer< T >::UpdatePlayerPosition (
```

```
PlayerMovement direction,
uint32_t entityId )  [inline], [override]
```

Updates the position of an entity based on the message received and the client ID.

This function updates the position of an entity. If the entity is not touching any other player, it updates its position and sends a message to all clients about the new position. If it touches another player, a destroy message is sent to all clients.

Parameters

msg	The message containing the new position of the entity.
client↩ Id	The ID of the client sending the update.

6.8.3.33 WaitForClientMessage()

```
template<typename T >
void r_type::net::AServer< T >::WaitForClientMessage ( )  [inline]
```

Waits for a client message asynchronously.

This function waits for a client message by asynchronously receiving data from the socket. When a message is received, it checks if the client endpoint protocol is UDPv4. If the protocol is not UDPv4, it recursively calls itself to wait for another client message. If the protocol is UDPv4 and there are no errors, it prints the client endpoint and checks if a connection already exists. If a connection already exists, it returns without further processing. If a connection does not exist, it creates a new client socket, binds it to a local endpoint, and creates a new connection object. It then calls the OnClientConnect function to check if the client connection is approved. If the connection is approved, it adds the new connection to the list of connections, connects it to the client, and prints the connection ID. If the connection is denied, it prints a message indicating the connection was denied. If there is an error during the receive operation, it prints the error message../

6.8.4 Member Data Documentation

6.8.4.1 _asioContext

```
template<typename T >
asio::io_context r_type::net::AServer< T >::_asioContext
```

The io_context object provides I/O services, such as sockets, that the server will use.

This member variable is responsible for managing asynchronous I/O operations. It is part of the ASIO library, which is used for network programming.

6.8.4.2 _asioSocket

```
template<typename T >
asio::ip::udp::socket r_type::net::AServer< T >::_asioSocket
```

A socket for sending and receiving UDP datagrams.

This member variable represents a UDP socket using the ASIO library. It is used for network communication in the server.

6.8.4.3 _background

```
template<typename T >
EntityInformation r_type::net::AServer< T >::_background
```

Holds information about the background entity.

This member variable stores the details related to the background entity in the game. It includes properties such as position, texture, and other relevant attributes that define the background's appearance and behavior.

6.8.4.4 _bossActive

```
template<typename T >
bool r_type::net::AServer< T >::_bossActive = false
```

6.8.4.5 _bossDefeated

```
template<typename T >
bool r_type::net::AServer< T >::_bossDefeated = false
```

6.8.4.6 _clientEndpoint

```
template<typename T >
asio::ip::udp::endpoint r_type::net::AServer< T >::_clientEndpoint
```

Represents the endpoint of a client in a UDP connection.

This member variable holds the endpoint information (IP address and port) of a client in a UDP connection using the ASIO library.

6.8.4.7 _clientInfoBarID

```
template<typename T >
std::unordered_map<uint32_t, uint32_t> r_type::net::AServer< T >::_clientInfoBarID
```

6.8.4.8 _clientPlayerID

```
template<typename T >
std::unordered_map<uint32_t, uint32_t> r_type::net::AServer< T >::_clientPlayerID
```

A container that maps client IDs to player IDs.

left: client ID right: player ID

This unordered map is used to associate client IDs with their corresponding player IDs. The keys are of type uint32_t representing the client IDs, and the values are also of type uint32_t representing the player IDs.

6.8.4.9 _clock

```
template<typename T >
std::chrono::system_clock::time_point r_type::net::AServer< T >::_clock = std::chrono::system↩
_clock::now()
```

Stores the current time point from the system clock.

This variable is initialized with the current time using std::chrono::system_clock::now() and represents a specific point in time according to the system clock.

6.8.4.10 _componentManager

```
template<typename T >
ComponentManager r_type::net::AServer< T >::_componentManager
```

Manages and maintains the lifecycle of various components within the server.

The ComponentManager is responsible for creating, updating, and destroying components as needed. It ensures that all components are properly managed and that their states are consistent throughout the server's operation.

6.8.4.11 _deqConnections

```
template<typename T >
std::deque<std::shared_ptr<Connection<T> > > r_type::net::AServer< T >::_deqConnections
```

A deque that holds shared pointers to Connection objects.

This member variable is used to manage a collection of active connections. The use of std::shared_ptr ensures that the Connection objects are reference-counted and automatically deallocated when no longer in use.

Template Parameters

T	The type of data that the Connection handles.

6.8.4.12 _endOfLevel

```
template<typename T >
bool r_type::net::AServer< T >::_endOfLevel = false
```

6.8.4.13 _entityFactory

```
template<typename T >
EntityFactory r_type::net::AServer< T >::_entityFactory
```

An instance of EntityFactory used to create and manage game entities.

6.8.4.14 _entityManager

```
template<typename T >
EntityManager r_type::net::AServer< T >::_entityManager
```

Manages the lifecycle and operations of entities within the server.

The EntityManager is responsible for creating, updating, and deleting entities. It ensures that entities are properly managed and synchronized within the server's environment.

6.8.4.15 _level

```
template<typename T >
r_type::Level<T> r_type::net::AServer< T >::_level
```

6.8.4.16 _nbrOfPlayers

```
template<typename T >
int r_type::net::AServer< T >::_nbrOfPlayers = 0
```

Number of players currently connected to the server.

6.8.4.17 _nIDCounter

```
template<typename T >
uint32_t r_type::net::AServer< T >::_nIDCounter = 10000
```

Counter for generating unique network IDs.

This variable is used to keep track of the current ID to be assigned for network-related entities. It starts at 10000 and increments with each new ID generation.

6.8.4.18 _playerConnected

```
template<typename T >
bool r_type::net::AServer< T >::_playerConnected = false
```

6.8.4.19 _playerReady

```
template<typename T >
int r_type::net::AServer< T >::_playerReady = 0
```

6.8.4.20 _port

```
template<typename T >
int r_type::net::AServer< T >::_port
```

6.8.4.21 _qMessagesIn

```
template<typename T >
ThreadSafeQueue<OwnedMessage<T> > r_type::net::AServer< T >::_qMessagesIn
```

Thread-safe queue to store incoming messages.

This member variable is a thread-safe queue that holds messages of type OwnedMessage<T>. It ensures that messages can be safely accessed and modified by multiple threads concurrently.

6.8.4.22 _tempBuffer

```
template<typename T >
std::array<uint8_t, 1024> r_type::net::AServer< T >::_tempBuffer
```

Temporary buffer used for storing data.

This buffer is an array of 1024 bytes (uint8_t) used for temporary storage of data within the server's network interface.

6.8.4.23 _threadContext

```
template<typename T >
std::thread r_type::net::AServer< T >::_threadContext
```

Thread object for managing the server's context operations.

This member variable represents a thread that handles the server's context, allowing for concurrent execution of tasks related to the server's operation. It is used to ensure that the server can perform its duties without blocking the main execution flow.

6.8.4.24 _watingPlayersReady

```
template<typename T >
bool r_type::net::AServer< T >::_watingPlayersReady = false
```

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

- /home/runner/work/R-Type/R-Type/Server/Interface/Include/level.hpp
- /home/runner/work/R-Type/R-Type/Server/Interface/Include/Net/a_server.hpp

6.9 AudioManager Class Reference

Manages and caches sound buffers for efficient audio playback.

```
#include <audio_manager.hpp>
```

Public Member Functions

- sf::SoundBuffer & getSoundBuffer (const std::string &filePath)

  Retrieves a sound buffer from the specified file path.

Private Attributes

- std::unordered_map< std::string, std::shared_ptr< sf::SoundBuffer > > soundBuffers

  A map that associates sound buffer names with their corresponding shared pointers to sf::SoundBuffer objects.

6.9.1 Detailed Description

Manages and caches sound buffers for efficient audio playback.

The AudioManager class is responsible for loading, caching, and retrieving sound buffers. It ensures that sound buffers are loaded only once and reused efficiently throughout the application.

Note

This class uses the SFML library for handling sound buffers.

6.9.2 Member Function Documentation

6.9.2.1 getSoundBuffer()

```
sf::SoundBuffer & AudioManager::getSoundBuffer (
            const std::string & filePath )  [inline]
```

Retrieves a sound buffer from the specified file path.

This function checks if the sound buffer is already cached. If it is, the cached sound buffer is returned. Otherwise, it loads the sound buffer from the file, caches it, and then returns it.

Parameters

filePath	The path to the sound file.

Returns

A reference to the sound buffer.

Exceptions

std::runtime_error	If the sound buffer fails to load from the file.

6.9.3 Member Data Documentation

6.9.3.1 soundBuffers

```
std::unordered_map<std::string, std::shared_ptr<sf::SoundBuffer> > AudioManager::soundBuffers
[private]
```

A map that associates sound buffer names with their corresponding shared pointers to sf::SoundBuffer objects.

This unordered map is used to store and manage sound buffers by their names, allowing for efficient retrieval and usage of sound resources in the application. Each entry in the map consists of a string key representing the name of the sound buffer and a shared pointer to an sf::SoundBuffer object, which ensures proper memory management and resource sharing.

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

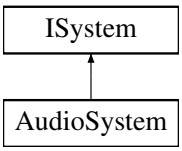
- /home/runner/work/R-Type/R-Type/ECS/Interface/Include/audio_manager.hpp

6.10 AudioSystem Class Reference

Manages audio playback within the application.

```
#include <audio_system.hpp>
```

Inheritance diagram for AudioSystem:



Public Member Functions

- AudioSystem (std::shared_ptr< AudioManager > audioManager)

  Constructs an AudioSystem object.
- void playBackgroundMusic (const std::string &filePath)

  Plays background music from the specified file.
- void stopBackgroundMusic ()

  Stops the background music that is currently playing.
- void playSoundEffect (const std::string &filePath)

  Plays a sound effect from the specified file.

Public Member Functions inherited from ISystem

- ISystem ()=default
- virtual ~ISystem ()=default

Private Attributes

- std::shared_ptr< AudioManager > _audioManager

  A shared pointer to the AudioManager instance.
- sf::Music _backgroundMusic

  A class that provides functionality for playing music.
- std::string _currentMusicFilePath

  Stores the file path of the currently playing music.
- sf::Sound _soundEffect

  Represents a sound effect that can be played in the audio system.

6.10.1 Detailed Description

Manages audio playback within the application.

The AudioSystem class provides functionalities for playing background music and sound effects. It utilizes the AudioManager for managing audio resources and the SFML library for audio playback.

This class is responsible for handling audio playback in the application. It allows for playing background music and sound effects from specified file paths. The class ensures proper management of audio resources through the use of std::shared_ptr for the AudioManager instance.

Note

The AudioSystem class relies on the SFML library for audio playback functionalities. Ensure that the SFML library is properly included and linked in your project.

See also

AudioManager

sf::Music

sf::Sound

6.10.2 Constructor & Destructor Documentation

6.10.2.1 AudioSystem()

```
AudioSystem::AudioSystem (
            std::shared_ptr< AudioManager > audioManager )  [inline]
```

Constructs an AudioSystem object.

Parameters

audioManager	A shared pointer to an AudioManager instance.

6.10.3 Member Function Documentation

6.10.3.1 playBackgroundMusic()

```
void AudioSystem::playBackgroundMusic (
            const std::string & filePath )
```

Plays background music from the specified file.

This function loads and plays background music from the given file path. It is typically used to provide ambient music for the application.

Parameters

filePath	The path to the audio file to be played as background music.

6.10.3.2 playSoundEffect()

```
void AudioSystem::playSoundEffect (
            const std::string & filePath )
```

Plays a sound effect from the specified file.

This function loads and plays a sound effect from the given file path. It is useful for triggering sound effects in response to game events.

Parameters

filePath	The path to the sound effect file to be played.

6.10.3.3 stopBackgroundMusic()

```
void AudioSystem::stopBackgroundMusic ( )
```

Stops the background music that is currently playing.

This function halts any background music that is being played by the audio system. It can be used to stop the music when it is no longer needed or when transitioning to a different scene or state in the application.

6.10.4 Member Data Documentation

6.10.4.1 _audioManager

```
std::shared_ptr<AudioManager> AudioSystem::_audioManager [private]
```

A shared pointer to the AudioManager instance.

This member variable holds a shared pointer to an AudioManager object, which is responsible for managing audio resources and playback within the system. The use of std::shared_ptr ensures that the AudioManager instance is properly managed and deallocated when no longer in use.

6.10.4.2 _backgroundMusic

```
sf::Music AudioSystem::_backgroundMusic [private]
```

A class that provides functionality for playing music.

The sf::Music class allows for streaming audio from a file or memory. It is particularly useful for playing large audio files, such as background music, as it does not load the entire file into memory.

6.10.4.3 _currentMusicFilePath

```
std::string AudioSystem::_currentMusicFilePath [private]
```

Stores the file path of the currently playing music.

6.10.4.4 _soundEffect

```
sf::Sound AudioSystem::_soundEffect [private]
```

Represents a sound effect that can be played in the audio system.

This member variable is an instance of the sf::Sound class from the SFML library, which is used to handle the playback of short sound effects. It provides functionalities to play, pause, stop, and manipulate sound properties such as volume, pitch, and loop status.

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

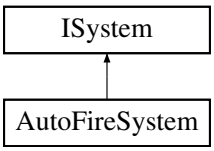
- /home/runner/work/R-Type/R-Type/ECS/Interface/Include/Systems/audio_system.hpp
- /home/runner/work/R-Type/R-Type/ECS/Src/Systems/audio_system.cpp

6.11 AutoFireSystem Class Reference

A system that handles automatic firing mechanisms for entities.

```
#include <auto_fire_system.hpp>
```

Inheritance diagram for AutoFireSystem:



Public Member Functions

- AutoFireSystem (ComponentManager &componentManager, EntityManager &entityManager)
- void handleAutoFire (ComponentManager &componentManager, EntityManager &entityManager)

  Handles the automatic firing mechanism for entities.

Public Member Functions inherited from ISystem

- ISystem ()=default
- virtual ~ISystem ()=default

Private Attributes

- ComponentManager & _componentManager

  Reference to the ComponentManager instance.
- EntityManager & _entityManager

  Reference to the EntityManager instance.

6.11.1 Detailed Description

A system that handles automatic firing mechanisms for entities.

System responsible for handling automatic firing mechanisms in entities.

The AutoFireSystem class is responsible for managing the automatic firing behavior of entities within the ECS framework. It interacts with the ComponentManager and EntityManager to update and control the firing state of entities.

Parameters

componentManager	Reference to the ComponentManager instance.
entityManager	Reference to the EntityManager instance.
componentManager	Reference to the ComponentManager that manages all components.
entityManager	Reference to the EntityManager that manages all entities.

6.11.2 Constructor & Destructor Documentation

6.11.2.1 AutoFireSystem()

```
AutoFireSystem::AutoFireSystem (
            ComponentManager & componentManager,
            EntityManager & entityManager )  [inline]
```

6.11.3 Member Function Documentation

6.11.3.1 handleAutoFire()

```
void AutoFireSystem::handleAutoFire (
            ComponentManager & componentManager,
            EntityManager & entityManager )
```

Handles the automatic firing mechanism for entities.

This function processes entities that have the auto-fire capability and triggers their firing actions based on the game logic and conditions.

Parameters

componentManager	Reference to the ComponentManager that manages all components.
entityManager	Reference to the EntityManager that manages all entities.

6.11.4 Member Data Documentation

6.11.4.1 _componentManager

```
ComponentManager& AutoFireSystem::_componentManager [private]
```

Reference to the ComponentManager instance.

This member variable holds a reference to the ComponentManager, which is responsible for managing all the components within the ECS (Entity Component System). It is used by the system to access and manipulate components associated with entities.

6.11.4.2 _entityManager

```
EntityManager& AutoFireSystem::_entityManager [private]
```

Reference to the EntityManager instance.

This member variable holds a reference to the EntityManager, which is responsible for managing the entities within the ECS (Entity Component System). It is used to perform operations such as adding, removing, and querying entities.

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

- /home/runner/work/R-Type/R-Type/ECS/Interface/Include/Systems/auto_fire_system.hpp
- /home/runner/work/R-Type/R-Type/ECS/Src/Systems/auto_fire_system.cpp

6.12 BackgroundComponent Struct Reference

```
#include <background_component.hpp>
```

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

- /home/runner/work/R-Type/R-Type/ECS/Interface/Include/Components/background_component.hpp

6.13 BasicMonsterComponent Struct Reference

```
#include <basic_monster_component.hpp>
```

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

- /home/runner/work/R-Type/R-Type/ECS/Interface/Include/Components/basic_monster_component.hpp

6.14 BindComponent Struct Reference

A component that binds a function to handle scene transitions.

```
#include <bind_component.hpp>
```

Public Member Functions

- BindComponent (std::function< IScenes *(AScenes *, AScenes::Actions)> bindFunction)

  Constructs a BindComponent with the given bind function.

Public Attributes

- bool isHovered = false

  A boolean flag indicating whether the component is currently hovered.
- std::function< IScenes *(AScenes *, AScenes::Actions)> bind

  A std::function that takes two AScenes pointers and an AScenes::Actions, and returns a pointer to an IScenes.

6.14.1 Detailed Description

A component that binds a function to handle scene transitions.

This component contains a function that takes two scene pointers and an action, and returns a pointer to a new scene. It also has a flag to indicate if the component is currently hovered.

6.14.2 Constructor & Destructor Documentation

6.14.2.1 BindComponent()

```
BindComponent::BindComponent (
            std::function< IScenes *(AScenes *, AScenes::Actions)> bindFunction )  [inline]
```

Constructs a BindComponent with the given bind function.

Parameters

bindFunction	The function to bind for handling scene transitions.

6.14.3 Member Data Documentation

6.14.3.1 bind

```
BindComponent::bind
```

A std::function that takes two AScenes pointers and an AScenes::Actions, and returns a pointer to an IScenes.

6.14.3.2 isHovered

```
BindComponent::isHovered = false
```

A boolean flag indicating whether the component is currently hovered.

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

- /home/runner/work/R-Type/R-Type/ECS/Interface/Include/Components/bind_component.hpp

6.15 BossComponent Struct Reference

```
#include <boss_component.hpp>
```

Public Attributes

- std::vector< int > tailSegmentIds

6.15.1 Member Data Documentation

6.15.1.1 tailSegmentIds

```
std::vector<int> BossComponent::tailSegmentIds
```

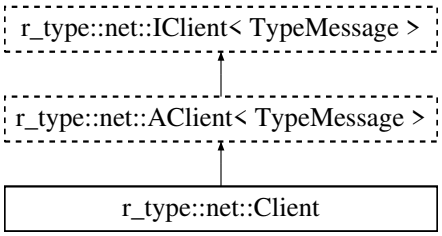
The documentation for this struct was generated from the following file:

- /home/runner/work/R-Type/R-Type/ECS/Interface/Include/Components/boss_component.hpp

6.16 r_type::net::Client Class Reference

```
#include <client.hpp>
```

Inheritance diagram for r_type::net::Client:



Public Member Functions

- void PingServer ()

  Send a message to the server to get the ping.
- void MessageAll ()

  Send a message to the server to all other clients.
- sf::Vector2u initInfoBar (UIEntityInformation entity, ComponentManager &componentManager, TextureManager &textureManager, FontManager &fontManager, sf::Vector2u windowSize)
- void updateInfoBar (UIEntityInformation entity, ComponentManager &componentManager)
- void addEntity (EntityInformation entity, ComponentManager &componentManager, TextureManager &textureManager, sf::Vector2u windowSize)
- void removeEntity (int entityId, ComponentManager &componentManager)
- void moveEntity (uint32_t id, vf2d newPos, ComponentManager &componentManager, sf::Vector2u windowSize)
- void animateEntity (int entityId, AnimationComponent rect, ComponentManager &componentManager)
- void displayEndOfGame (ComponentManager &componentManager, TextureManager &textureManager, FontManager &fontManager, sf::Vector2u windowSize)

Public Member Functions inherited from r_type::net::AClient< TypeMessage >

- AClient ()
- virtual ~AClient ()
- bool Connect (const std::string &host, const uint16_t port)

  Connects to a remote host using UDP protocol.
- void Disconnect ()

  Disconnects the client from the server.
- bool IsConnected ()

  Checks if the client is connected to the server.
- void Send (const Message< TypeMessage > &msg)

  Send message to server.
- ThreadSafeQueue< OwnedMessage< TypeMessage > > & Incoming ()

  get incoming messages
- const std::unique_ptr< Connection< TypeMessage > > & getConnection ()
- void setPlayerId (uint32_t id)
- uint32_t getPlayerId ()
- void setWindowSize (sf::Vector2u size)
- sf::Vector2u getWindowSize ()

Public Member Functions inherited from r_type::net::IClient< T >

- IClient ()
- virtual ~IClient ()
- virtual void Send (const Message< T > &msg)=0

  Send message to server.

Additional Inherited Members

Protected Attributes inherited from r_type::net::AClient< TypeMessage >

- asio::io_context m_context
- std::thread thrContext
- std::unique_ptr< Connection< TypeMessage > > m_connection

6.16.1 Member Function Documentation

6.16.1.1 addEntity()

```
void r_type::net::Client::addEntity (
            EntityInformation entity,
            ComponentManager & componentManager,
            TextureManager & textureManager,
            sf::Vector2u windowSize )  [inline]
```

6.16.1.2 animateEntity()

```
void r_type::net::Client::animateEntity (
            int entityId,
            AnimationComponent rect,
            ComponentManager & componentManager )  [inline]
```

6.16.1.3 displayEndOfGame()

```
void r_type::net::Client::displayEndOfGame (
            ComponentManager & componentManager,
            TextureManager & textureManager,
            FontManager & fontManager,
            sf::Vector2u windowSize )  [inline]
```

6.16.1.4 initInfoBar()

```
sf::Vector2u r_type::net::Client::initInfoBar (
            UIEntityInformation entity,
            ComponentManager & componentManager,
            TextureManager & textureManager,
            FontManager & fontManager,
            sf::Vector2u windowSize )  [inline]
```

6.16.1.5 MessageAll()

```
void r_type::net::Client::MessageAll ( )  [inline]
```

Send a message to the server to all other clients.

6.16.1.6 moveEntity()

```
void r_type::net::Client::moveEntity (
            uint32_t id,
            vf2d newPos,
            ComponentManager & componentManager,
            sf::Vector2u windowSize )  [inline]
```

6.16.1.7 PingServer()

```
void r_type::net::Client::PingServer ( ) [inline]
```

Send a message to the server to get the ping.

6.16.1.8 removeEntity()

```
void r_type::net::Client::removeEntity (
            int entityId,
            ComponentManager & componentManager ) [inline]
```

6.16.1.9 updateInfoBar()

```
void r_type::net::Client::updateInfoBar (
            UIEntityInformation entity,
            ComponentManager & componentManager ) [inline]
```

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

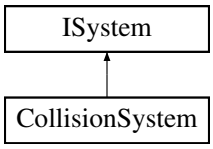
- /home/runner/work/R-Type/R-Type/Client/Interface/Include/Net/client.hpp

6.17 CollisionSystem Class Reference

Manages collision detection and response within the ECS framework.

```
#include <collision_system.hpp>
```

Inheritance diagram for CollisionSystem:



Public Member Functions

- CollisionSystem (ComponentManager &componentManager, EntityManager &entityManager)
- bool checkCollision (ComponentManager &componentManager, int entityId1, int entityId2)

  Checks for a collision between two entities.
- bool checkOffScreen (ComponentManager &componentManager, int entityId)

  Checks if the entity with the given ID is off the screen.

Public Member Functions inherited from ISystem

- ISystem ()=default
- virtual ~ISystem ()=default

Private Attributes

- ComponentManager & _componentManager

  Reference to the ComponentManager instance.
- EntityManager & _entityManager

  Reference to the EntityManager instance.

6.17.1 Detailed Description

Manages collision detection and response within the ECS framework.

This system is responsible for handling all collision-related logic, including detecting collisions between entities and responding to them appropriately.

Parameters

componentManager	Reference to the ComponentManager that handles the components of the entities.
entityManager	Reference to the EntityManager that manages the entities in the system.

6.17.2 Constructor & Destructor Documentation

6.17.2.1 CollisionSystem()

```
CollisionSystem::CollisionSystem (
            ComponentManager & componentManager,
            EntityManager & entityManager )  [inline]
```

6.17.3 Member Function Documentation

6.17.3.1 checkCollision()

```
bool CollisionSystem::checkCollision (
            ComponentManager & componentManager,
            int entityId1,
            int entityId2 )
```

Checks for a collision between two entities.

This function determines whether there is a collision between the components of two specified entities within the component manager.

Parameters

componentManager	Reference to the ComponentManager that holds the components of all entities.
entityId1	The ID of the first entity to check for collision.
entityId2	The ID of the second entity to check for collision.

Returns

true if a collision is detected between the two entities, false otherwise.

6.17.3.2 checkOffScreen()

```
bool CollisionSystem::checkOffScreen (
            ComponentManager & componentManager,
            int entityId )
```

Checks if the entity with the given ID is off the screen.

This function determines whether the specified entity is outside the visible screen area based on its components managed by the ComponentManager.

Parameters

componentManager	Reference to the ComponentManager that manages the entity's components.
entityId	The ID of the entity to check.

Returns

true if the entity is off the screen, false otherwise.

6.17.4 Member Data Documentation

6.17.4.1 _componentManager

```
ComponentManager& CollisionSystem::_componentManager  [private]
```

Reference to the ComponentManager instance.

This member is used to manage and access various components within the ECS (Entity Component System).

6.17.4.2 _entityManager

```
EntityManager& CollisionSystem::_entityManager  [private]
```

Reference to the EntityManager instance.

This member variable holds a reference to the EntityManager, which is responsible for managing the entities within the ECS (Entity Component System). It is used to perform operations such as adding, removing, and querying entities.

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

- /home/runner/work/R-Type/R-Type/ECS/Interface/Include/Systems/collision_system.hpp
- /home/runner/work/R-Type/R-Type/ECS/Src/Systems/collision_system.cpp

6.18 ComponentManager Class Reference

Manages the components of entities in an ECS system.

```
#include <component_manager.hpp>
```

Public Member Functions

- template<typename ComponentType , typename... Args>
  void addComponent (int entityId, Args &&...args)

  Adds a component to an entity.
- template<typename ComponentType >
  std::optional< ComponentType * > getComponent (int entityId)

  Retrieves the component of the specified type associated with the given entity ID.
- template<typename ComponentType >
  std::optional< std::unordered_map< int, std::any > * > getComponentMap ()

  Retrieves the component map for the specified component type.
- template<typename ComponentType >
  void removeEntityFromComponent (int entityId)

  Removes an entity from the specified component type.
- void removeAllComponents ()

  Removes all components from the component manager.
- void removeEntityFromAllComponents (int entityId)

  Removes the specified entity from all components.

Private Attributes

- std::unordered_map< std::type_index, std::unordered_map< int, std::any > > components

  A component manager that stores components in an unordered map.

6.18.1 Detailed Description

Manages the components of entities in an ECS system.

The ComponentManager class provides functionality to add and retrieve components for entities in an ECS system. It uses an unordered map to store the components, where the key is the type of the component and the value is another unordered map that maps entity IDs to their corresponding component values.

6.18.2 Member Function Documentation

6.18.2.1 addComponent()

```
template<typename ComponentType , typename...  Args>
void ComponentManager::addComponent (
            int entityId,
            Args &&...  args )  [inline]
```

Adds a component to an entity.

Template Parameters

ComponentType	The type of the component to add.
Args	The types of the arguments to forward to the component's constructor.

Parameters

entity↩ Id	The ID of the entity to add the component to.
args	The arguments to forward to the component's constructor.

6.18.2.2 getComponent()

```
template<typename ComponentType >
std::optional< ComponentType * > ComponentManager::getComponent (
            int entityId )  [inline]
```

Retrieves the component of the specified type associated with the given entity ID.

Template Parameters

ComponentType	The type of the component to retrieve.

Parameters

entity↩ Id	The ID of the entity.

Returns

An optional pointer to the component if found, otherwise std::nullopt.

6.18.2.3 getComponentMap()

```
template<typename ComponentType >
std::optional< std::unordered_map< int, std::any > * > ComponentManager::getComponentMap ( )
[inline]
```

Retrieves the component map for the specified component type.

Template Parameters

ComponentType	The type of the component.

Returns

std::optional<std::unordered_map<int, std::any>*> The component map if found, otherwise std::nullopt.

6.18.2.4 removeAllComponents()

```
void ComponentManager::removeAllComponents ( ) [inline]
```

Removes all components from the component manager.

6.18.2.5 removeEntityFromAllComponents()

```
void ComponentManager::removeEntityFromAllComponents (
            int entityId ) [inline]
```

Removes the specified entity from all components.

This function iterates through all components and removes the entity with the given ID from each component's collection.

Parameters

entity↩ Id	The ID of the entity to be removed from all components.

6.18.2.6 removeEntityFromComponent()

```
template<typename ComponentType >
void ComponentManager::removeEntityFromComponent (
            int entityId ) [inline]
```

Removes an entity from the specified component type.

This function searches for the component type in the components map using the typeid of the ComponentType. If the component type is found, it removes the entity with the given entityId from the component's entity list.

Template Parameters

ComponentType	The type of the component from which the entity should be removed.

Parameters

entity↩ Id	The ID of the entity to be removed from the component.

6.18.3 Member Data Documentation

6.18.3.1 components

```
std::unordered_map<std::type_index, std::unordered_map<int, std::any> > ComponentManager↩
::components [private]
```

A component manager that stores components in an unordered map.

This component manager uses an unordered map to store components. The keys of the outer map are of type std::type_index, which represents the type of the component. The values of the outer map are inner unordered maps, where the keys are of type int and represent the entity ID, and the values are of type std::any, which allows storing components of any type.

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

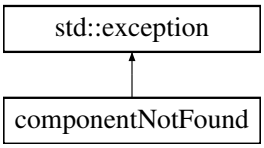
- /home/runner/work/R-Type/R-Type/ECS/Interface/Include/Components/component_manager.hpp

6.19 componentNotFound Class Reference

Exception class for when a component is not found.

```
#include <error_handling.hpp>
```

Inheritance diagram for componentNotFound:



Private Member Functions

- const char ∗ what () const noexcept override

6.19.1 Detailed Description

Exception class for when a component is not found.

This exception is thrown when a component is not found in the system. It inherits from std::exception and overrides the what() method to provide a custom error message.

6.19.2 Member Function Documentation

6.19.2.1 what()

```
const char * componentNotFound::what ( ) const  [inline], [override], [private], [noexcept]
```

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

- /home/runner/work/R-Type/R-Type/ECS/Interface/Include/error_handling.hpp

6.20 EnemyComponent Struct Reference

```
#include <enemy_component.hpp>
```

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

- /home/runner/work/R-Type/R-Type/ECS/Interface/Include/Components/enemy_component.hpp

6.21 EnemyMissileComponent Struct Reference

```
#include <enemy_missile_component.hpp>
```

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

- /home/runner/work/R-Type/R-Type/ECS/Interface/Include/Components/enemy_missile_component.hpp

6.22 Entity Class Reference

Represents an entity in the ECS system.

```
#include <entity.hpp>
```

Public Member Functions

- Entity (int id)

  Constructs an Entity with a specified ID.
- int getId () const

  Retrieves the unique identifier of the entity.

Private Attributes

- int _id

  Unique identifier for the entity.

6.22.1 Detailed Description

Represents an entity in the ECS system.

This class is a concrete implementation of the IEntity interface. It provides functionality to retrieve the ID of the entity.

6.22.2 Constructor & Destructor Documentation

6.22.2.1 Entity()

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

Constructs an Entity with a specified ID.

Parameters

id	The unique identifier for the entity.

6.22.3 Member Function Documentation

6.22.3.1 getId()

```
int Entity::getId ( ) const  [inline]
```

Retrieves the unique identifier of the entity.

Returns

int The unique identifier of the entity.

6.22.4 Member Data Documentation

6.22.4.1 _id

```
int Entity::_id  [private]
```

Unique identifier for the entity.

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

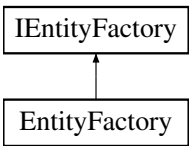
- /home/runner/work/R-Type/R-Type/ECS/Interface/Include/Entities/entity.hpp

6.23 EntityFactory Class Reference

A factory class for creating various types of entities.

```
#include <entity_factory.hpp>
```

Inheritance diagram for EntityFactory:



Public Member Functions

- Entity createBackgroundLevelOne (EntityManager &entityManager, ComponentManager &component↩ Manager) override

  Create a Background Level One object.
- Entity createBackgroundLevelTwo (EntityManager &entityManager, ComponentManager &component↩ Manager) override

  Create a Background Level Two object.
- Entity createBackgroundLevelThree (EntityManager &entityManager, ComponentManager &component↩ Manager) override

  Create a Background Level Three object.
- Entity createBackgroundMenu (EntityManager &entityManager, ComponentManager &componentManager, TextureManager &textureManager) override

  Create a Background Menu object.
- Entity createInfoBar (EntityManager &entityManager, ComponentManager &componentManager) override

  Creates a bar entity.
- Entity createPlayer (EntityManager &entityManager, ComponentManager &componentManager, int nbrOf↩ Players) override

  Creates a player entity.
- Entity createShooterEnemy (EntityManager &entityManager, ComponentManager &componentManager, int posX, int posY) override

  Creates a shooter enemy entity.
- Entity createBasicMonster (EntityManager &entityManager, ComponentManager &componentManager, int posX, int posY) override

  Creates a basic monster entity.
- Entity createPlayerMissile (EntityManager &entityManager, ComponentManager &componentManager, uint32_t entityId) override

  Creates a player missile entity.
- Entity createForceWeapon (EntityManager &entityManager, ComponentManager &componentManager, uint32_t entityId) override

  Creates a force weapon entity.
- Entity createForceMissile (EntityManager &entityManager, ComponentManager &componentManager, uint32_t entityId) override

  Creates a force missile entity.
- Entity createPowerUpBlueLaserCrystal (EntityManager &entityManager, ComponentManager &component↩ Manager, int posX, int posY) override

  Creates a power-up blue laser crystal entity.
- Entity createWall (EntityManager &entityManager, ComponentManager &componentManager, int posX, int posY) override

  Creates a wall entity.
- Entity createButton (EntityManager &entityManager, ComponentManager &componentManager, TextureManager &textureManager, FontManager &fontManager, std::string text, std::function< IScenes *(AScenes *)> *on↩ Click, float x=0, float y=0) override

  Creates a button entity.
- Entity createSmallButton (EntityManager &entityManager, ComponentManager &componentManager, TextureManager &textureManager, FontManager &fontManager, std::string text, std::function< IScenes *(AScenes *, AScenes::Actions)> *onClick, float x=0, float y=0) override

  Creates a small button entity.
- Entity createUpdateButton (EntityManager &entityManager, ComponentManager &componentManager, TextureManager &textureManager, FontManager &fontManager, std::string text, std::function< IScenes *(AScenes *)> *onClick, std::function< std::string(GameParameters)> *updateText, float x, float y) override
- Entity createEnemyMissile (EntityManager &entityManager, ComponentManager &componentManager, uint32_t entityId) override

Creates an enemy missile entity.

- Entity createFilter (EntityManager &entityManager, ComponentManager &componentManager, AScenes::DaltonismMode mode)

  Creates a filter entity.
- Entity backgroundFactory (EntityManager &entityManager, ComponentManager &componentManager, GameState type)
- Entity createBoss (EntityManager &entityManager, ComponentManager &componentManager, EntityFactory &entityFactory)

  Creates a boss entity.
- Entity createTailSegment (EntityManager &entityManager, ComponentManager &componentManager) override
- Entity createTailEnd (EntityManager &entityManager, ComponentManager &componentManager) override

Public Member Functions inherited from IEntityFactory

- virtual ~IEntityFactory ()=default

  Destroy the IEntityFactory object.

Additional Inherited Members

Public Types inherited from IEntityFactory

- enum EnemyType { BasicMonster , ShooterEnemy , Wall , Boss }

  Enumeration representing different types of enemies in the game.

6.23.1 Detailed Description

A factory class for creating various types of entities.

The EntityFactory class provides methods to create different types of entities such as background, player, enemies, missiles, buttons, and more. It utilizes the provided entity manager and component manager to create and initialize the entities with the necessary components.

6.23.2 Member Function Documentation

6.23.2.1 backgroundFactory()

```
Entity EntityFactory::backgroundFactory (
            EntityManager & entityManager,
            ComponentManager & componentManager,
            GameState type )
```

6.23.2.2 createBackgroundLevelOne()

```
Entity EntityFactory::createBackgroundLevelOne (
            EntityManager & entityManager,
            ComponentManager & componentManager )  [override], [virtual]
```

Create a Background Level One object.

Parameters

entityManager	
componentManager	

Returns

Entity

Implements IEntityFactory.

6.23.2.3 createBackgroundLevelThree()

```
Entity EntityFactory::createBackgroundLevelThree (
            EntityManager & entityManager,
            ComponentManager & componentManager )  [override], [virtual]
```

Create a Background Level Three object.

Parameters

entityManager	
componentManager	

Returns

Entity

Implements IEntityFactory.

6.23.2.4 createBackgroundLevelTwo()

```
Entity EntityFactory::createBackgroundLevelTwo (
            EntityManager & entityManager,
            ComponentManager & componentManager )  [override], [virtual]
```

Create a Background Level Two object.

Parameters

entityManager	
componentManager	

Returns

Entity

Implements IEntityFactory.

6.23.2.5 createBackgroundMenu()

```
Entity EntityFactory::createBackgroundMenu (
            EntityManager & entityManager,
            ComponentManager & componentManager,
            TextureManager & textureManager )  [override], [virtual]
```

Create a Background Menu object.

Parameters

entityManager	
componentManager	

Returns

Entity

Implements IEntityFactory.

6.23.2.6 createBasicMonster()

```
Entity EntityFactory::createBasicMonster (
            EntityManager & entityManager,
            ComponentManager & componentManager,
            int posX,
            int posY )  [override], [virtual]
```

Creates a basic monster entity.

This function creates a basic monster entity using the provided entity manager and component manager.

Parameters

entityManager	The entity manager used to create the entity.
componentManager	The component manager used to add components to the entity.
posX	The x-coordinate position of the basic monster.
posY	The y-coordinate position of the basic monster.

Returns

The created basic monster entity.

Implements IEntityFactory.

6.23.2.7 createBoss()

```
Entity EntityFactory::createBoss (
            EntityManager & entityManager,
```

```
            ComponentManager & componentManager,
            EntityFactory & entityFactory )
```

Creates a boss entity.

This function creates a boss entity using the provided entity manager and component manager.

Parameters

entityManager	The entity manager used to create the entity.
componentManager	The component manager used to add components to the entity.

Returns

The created boss entity.

6.23.2.8 createButton()

```
Entity EntityFactory::createButton (
            EntityManager & entityManager,
            ComponentManager & componentManager,
            TextureManager & textureManager,
            FontManager & fontManager,
            std::string text,
            std::function< IScenes *(AScenes *)> * onClick,
            float x = 0,
            float y = 0 )  [override], [virtual]
```

Creates a button entity.

This function creates a button entity with the specified parameters.

Parameters

entityManager	The entity manager to create the entity.
componentManager	The component manager to add components to the entity.
textureManager	The texture manager to load the button texture.
fontManager	The font manager to load the button font.
text	The text to display on the button.
onClick	The function to be called when the button is clicked.
x	The x-coordinate position of the button.
y	The y-coordinate position of the button.

Returns

The created button entity.

Implements IEntityFactory.

6.23.2.9 createEnemyMissile()

```
Entity EntityFactory::createEnemyMissile (
            EntityManager & entityManager,
            ComponentManager & componentManager,
            uint32_t entityId )  [override], [virtual]
```

Creates an enemy missile entity.

This function creates an enemy missile entity using the provided entity manager and component manager.

Parameters

entityManager	The entity manager used to create the entity.
componentManager	The component manager used to add components to the entity.
entityId	The id of the entity that shoots the missile.

Returns

The created enemy missile entity.

Implements IEntityFactory.

6.23.2.10 createFilter()

```
Entity EntityFactory::createFilter (
            EntityManager & entityManager,
            ComponentManager & componentManager,
            AScenes::DaltonismMode mode )
```

Creates a filter entity.

This function creates a filter entity using the provided entity manager and component manager.

Parameters

entityManager	The entity manager to use for creating the entity.
componentManager	The component manager to use for adding components to the entity.
mode	The Daltonism mode for the filter.

Returns

The created filter entity.

6.23.2.11 createForceMissile()

```
Entity EntityFactory::createForceMissile (
            EntityManager & entityManager,
```

```
            ComponentManager & componentManager,
            uint32_t entityId )  [override], [virtual]
```

Creates a force missile entity.

This function creates a force missile entity with the specified player ID and adds it to the entity manager. It also initializes the necessary components for the force missile entity using the component manager.

Parameters

entityManager	The entity manager to add the force missile entity to.
componentManager	The component manager to initialize the components for the force missile entity.
entityId	The id of the entity that shoots the force missile.

Returns

The created force missile entity.

Implements IEntityFactory.

6.23.2.12 createForceWeapon()

```
Entity EntityFactory::createForceWeapon (
            EntityManager & entityManager,
            ComponentManager & componentManager,
            uint32_t entityId )  [override], [virtual]
```

Creates a force weapon entity.

This function creates a force weapon entity with the specified player ID and adds it to the entity manager. It also initializes the necessary components for the force weapon entity using the component manager.

Parameters

entityManager	The entity manager to add the force weapon entity to.
componentManager	The component manager to initialize the components for the force weapon entity.
entityId	The id of the entity that uses the force weapon.

Returns

The created force weapon entity.

Implements IEntityFactory.

6.23.2.13 createInfoBar()

```
Entity EntityFactory::createInfoBar (
            EntityManager & entityManager,
            ComponentManager & componentManager )  [override], [virtual]
```

Creates a bar entity.

This function creates a bar with text for displaying player information like health and score.

Parameters

entityManager	The entity manager to use for creating the entity.
componentManager	The component manager to use for adding components to the entity.

Returns

The created bar entity.

Implements IEntityFactory.

6.23.2.14 createPlayer()

```
Entity EntityFactory::createPlayer (
            EntityManager & entityManager,
            ComponentManager & componentManager,
            int nbrOfPlayers )  [override], [virtual]
```

Creates a player entity.

This function creates a player entity using the provided entity manager and component manager.

Parameters

entityManager	The entity manager to use for creating the entity.
componentManager	The component manager to use for adding components to the entity.
nbrOfPlayers	The number of players to create.

Returns

The created player entity.

Implements IEntityFactory.

6.23.2.15 createPlayerMissile()

```
Entity EntityFactory::createPlayerMissile (
            EntityManager & entityManager,
            ComponentManager & componentManager,
            uint32_t entityId )  [override], [virtual]
```

Creates a player missile entity.

This function creates a player missile entity with the specified player ID and adds it to the entity manager. It also initializes the necessary components for the player missile entity using the component manager.

Parameters

entityManager	The entity manager to add the player missile entity to.
componentManager	The component manager to initialize the components for the player missile entity.
entityId	The id of the entity that shoots the missile.

Returns

The created player missile entity.

Implements IEntityFactory.

6.23.2.16 createPowerUpBlueLaserCrystal()

```
Entity EntityFactory::createPowerUpBlueLaserCrystal (
            EntityManager & entityManager,
            ComponentManager & componentManager,
            int posX,
            int posY )  [override], [virtual]
```

Creates a power-up blue laser crystal entity.

This function creates a power-up blue laser crystal entity using the provided entity manager and component manager.

Parameters

entityManager	The entity manager to use for creating the entity.
componentManager	The component manager to use for adding components to the entity.

Returns

The created power-up blue laser crystal entity.

Implements IEntityFactory.

6.23.2.17 createShooterEnemy()

```
Entity EntityFactory::createShooterEnemy (
            EntityManager & entityManager,
            ComponentManager & componentManager,
            int posX,
            int posY )  [override], [virtual]
```

Creates a shooter enemy entity.

This function creates a shooter enemy entity using the provided entity manager and component manager.

Parameters

entityManager	The entity manager used to create the entity.
componentManager	The component manager used to add components to the entity.
posX	The x-coordinate position of the shooter enemy.
posY	The y-coordinate position of the shooter enemy.

Returns

The created shooter enemy entity.

Implements IEntityFactory.

6.23.2.18 createSmallButton()

```
Entity EntityFactory::createSmallButton (
            EntityManager & entityManager,
            ComponentManager & componentManager,
            TextureManager & textureManager,
            FontManager & fontManager,
            std::string text,
            std::function< IScenes *(AScenes *, AScenes::Actions)> * onClick,
            float x = 0,
            float y = 0 )  [override], [virtual]
```

Creates a small button entity.

This function creates a small button entity with the specified parameters.

Parameters

entityManager	The entity manager to create the entity.
componentManager	The component manager to add components to the entity.
textureManager	The texture manager to load the button texture.
fontManager	The font manager to load the button font.
text	The text to display on the button.
onClick	The function to be called when the button is clicked.
x	The x-coordinate position of the button.
y	The y-coordinate position of the button.

Returns

The created small button entity.

Implements IEntityFactory.

6.23.2.19 createTailEnd()

```
Entity EntityFactory::createTailEnd (
            EntityManager & entityManager,
            ComponentManager & componentManager )  [override], [virtual]
```

Implements IEntityFactory.

6.23.2.20 createTailSegment()

```
Entity EntityFactory::createTailSegment (
            EntityManager & entityManager,
            ComponentManager & componentManager )  [override], [virtual]
```

Implements IEntityFactory.

6.23.2.21 createUpdateButton()

```
Entity EntityFactory::createUpdateButton (
            EntityManager & entityManager,
            ComponentManager & componentManager,
            TextureManager & textureManager,
            FontManager & fontManager,
            std::string text,
            std::function< IScenes *(AScenes *)> * onClick,
            std::function< std::string(GameParameters)> * updateText,
            float x,
            float y )  [override], [virtual]
```

Implements IEntityFactory.

6.23.2.22 createWall()

```
Entity EntityFactory::createWall (
            EntityManager & entityManager,
            ComponentManager & componentManager,
            int posX,
            int posY )  [override], [virtual]
```

Creates a wall entity.

This function creates a wall entity using the provided entity manager and component manager.

Parameters

entityManager	The entity manager to use for creating the entity.
componentManager	The component manager to use for adding components to the entity.
posX	The x-coordinate position of the wall.
posY	The y-coordinate position of the wall.

Returns

The created wall entity.

Implements IEntityFactory.

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

- /home/runner/work/R-Type/R-Type/ECS/Interface/Include/Entities/entity_factory.hpp
- /home/runner/work/R-Type/R-Type/ECS/Src/Entities/entity_factory.cpp

6.24 EntityInformation Struct Reference

Represents information about an entity.

```
#include <entity_struct.hpp>
```

Public Attributes

- uint32_t uniqueID = 0
- vf2d ratio = {0, 0}
- SpriteDataComponent spriteData
- vf2d vPos = {0, 0}
- AnimationComponent animationComponent = {{0, 0}, {0, 0}}

6.24.1 Detailed Description

Represents information about an entity.

6.24.2 Member Data Documentation

6.24.2.1 animationComponent

```
AnimationComponent EntityInformation::animationComponent = {{0, 0}, {0, 0}}
```

6.24.2.2 ratio

```
vf2d EntityInformation::ratio = {0, 0}
```

6.24.2.3 spriteData

```
SpriteDataComponent EntityInformation::spriteData
```

6.24.2.4 uniqueID

```
uint32_t EntityInformation::uniqueID = 0
```

6.24.2.5 vPos

```
vf2d EntityInformation::vPos = {0, 0}
```

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

- /home/runner/work/R-Type/R-Type/ECS/Interface/Include/entity_struct.hpp

6.25 EntityManager Class Reference

Manages the creation, removal, and retrieval of entities.

```
#include <entity_manager.hpp>
```

Public Member Functions

- Entity createEntity ()

  Creates a new entity and adds it to the entity manager.
- void removeEntity (int entityId)

  Remove an entity from the entity manager.
- void removeAllEntities ()

  Remove all entities from the entity manager.
- std::optional< Entity * > getEntity (int entityId)

  Get an entity by its ID.
- const std::vector< Entity > & getAllEntities () const

  Get all entities in the entity manager.

Private Attributes

- int entityNb = 0

  The number of entities in the entity manager.
- std::vector< Entity > entities

  A container that holds a collection of Entity objects.

6.25.1 Detailed Description

Manages the creation, removal, and retrieval of entities.

The EntityManager class is responsible for managing entities within the system. It provides functionality to create new entities, remove existing ones, and retrieve entities by their ID. It also allows access to all entities currently managed by the entity manager.

6.25.2 Member Function Documentation

6.25.2.1 createEntity()

```
Entity EntityManager::createEntity ( )  [inline]
```

Creates a new entity and adds it to the entity manager.

This function increments the entity counter, assigns a new unique ID to the entity, and adds it to the list of managed entities.

Returns

Entity The newly created entity.

6.25.2.2 getAllEntities()

```
const std::vector< Entity > & EntityManager::getAllEntities ( ) const  [inline]
```

Get all entities in the entity manager.

Returns

const std::vector<Entity>& A reference to the vector of entities.

This function returns a reference to the vector of entities in the entity manager.

6.25.2.3 getEntity()

```
std::optional< Entity * > EntityManager::getEntity (
            int entityId )  [inline]
```

Get an entity by its ID.

Parameters

entity↩ Id	The ID of the entity to retrieve.

Returns

Entity& A reference to the entity with the specified ID.

This function retrieves the entity with the specified ID from the entity manager. If the entity is not found, an entityNotFound exception is thrown.

6.25.2.4 removeAllEntities()

```
void EntityManager::removeAllEntities ( ) [inline]
```

Remove all entities from the entity manager.

This function removes all entities from the entity manager.

6.25.2.5 removeEntity()

```
void EntityManager::removeEntity (
            int entityId ) [inline]
```

Remove an entity from the entity manager.

Parameters

entity↩ Id	The ID of the entity to remove.

This function removes the entity with the specified ID from the entity manager. If the entity is not found, an entityNotFound exception is thrown.

6.25.3 Member Data Documentation

6.25.3.1 entities

```
std::vector<Entity> EntityManager::entities [private]
```

A container that holds a collection of Entity objects.

This vector is used to manage and store all the entities within the Entity Component System (ECS). Each Entity represents a unique object within the ECS framework.

6.25.3.2 entityNb

```
int EntityManager::entityNb = 0  [private]
```

The number of entities in the entity manager.

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

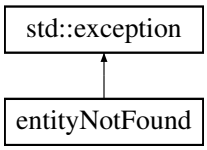
- /home/runner/work/R-Type/R-Type/ECS/Interface/Include/Entities/entity_manager.hpp

6.26 entityNotFound Class Reference

Exception class for entity not found error.

```
#include <error_handling.hpp>
```

Inheritance diagram for entityNotFound:



Private Member Functions

- const char ∗ what () const noexcept override

6.26.1 Detailed Description

Exception class for entity not found error.

This exception is thrown when an entity is not found. It is derived from the std::exception class. The `what()` function is overridden to provide a custom error message.

6.26.2 Member Function Documentation

6.26.2.1 what()

```
const char * entityNotFound::what ( ) const  [inline], [override], [private], [noexcept]
```

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

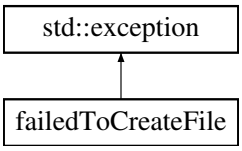
- /home/runner/work/R-Type/R-Type/ECS/Interface/Include/error_handling.hpp

6.27 failedToCreateFile Class Reference

Exception class for handling file creation failures.

```
#include <error_handling.hpp>
```

Inheritance diagram for failedToCreateFile:



Private Member Functions

- const char * what () const noexcept override

6.27.1 Detailed Description

Exception class for handling file creation failures.

This exception is thrown when a file creation operation fails. It inherits from the standard std::exception class.

6.27.2 Member Function Documentation

6.27.2.1 what()

```
const char * failedToCreateFile::what ( ) const  [inline], [override], [private], [noexcept]
```

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

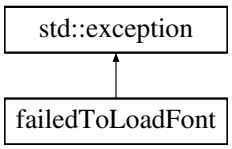
- /home/runner/work/R-Type/R-Type/ECS/Interface/Include/error_handling.hpp

6.28 failedToLoadFont Class Reference

Exception class for handling font loading failures.

```
#include <error_handling.hpp>
```

Inheritance diagram for failedToLoadFont:



Private Member Functions

- const char ∗ what () const noexcept override

6.28.1 Detailed Description

Exception class for handling font loading failures.

This exception is thrown when the application fails to load a font. It inherits from std::exception and overrides the what() method to provide a specific error message.

6.28.2 Member Function Documentation

6.28.2.1 what()

```
const char * failedToLoadFont::what ( ) const  [inline], [override], [private], [noexcept]
```

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

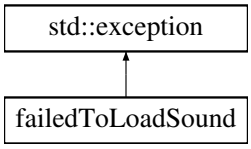
- /home/runner/work/R-Type/R-Type/ECS/Interface/Include/error_handling.hpp

6.29 failedToLoadSound Class Reference

Exception class for handling sound loading failures.

```
#include <error_handling.hpp>
```

Inheritance diagram for failedToLoadSound:



Private Member Functions

- const char ∗ what () const noexcept override

6.29.1 Detailed Description

Exception class for handling sound loading failures.

This exception is thrown when the application fails to load a sound file. It inherits from std::exception and overrides the what() method to provide a specific error message.

6.29.2 Member Function Documentation

6.29.2.1 what()

```
const char * failedToLoadSound::what ( ) const  [inline], [override], [private], [noexcept]
```

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

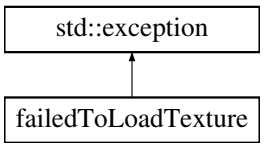
- /home/runner/work/R-Type/R-Type/ECS/Interface/Include/error_handling.hpp

6.30 failedToLoadTexture Class Reference

Exception class for failed texture loading.

```
#include <error_handling.hpp>
```

Inheritance diagram for failedToLoadTexture:



Private Member Functions

- const char * what () const noexcept override

6.30.1 Detailed Description

Exception class for failed texture loading.

This exception is thrown when there is a failure to load a texture. It inherits from the std::exception class and overrides the what() method to provide a custom error message.

6.30.2 Member Function Documentation

6.30.2.1 what()

```
const char * failedToLoadTexture::what ( ) const  [inline], [override], [private], [noexcept]
```

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

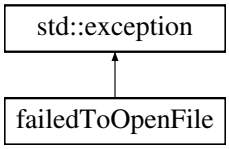
- /home/runner/work/R-Type/R-Type/ECS/Interface/Include/error_handling.hpp

6.31 failedToOpenFile Class Reference

Exception class for handling file opening failures.

```
#include <error_handling.hpp>
```

Inheritance diagram for failedToOpenFile:



Private Member Functions

- const char * what () const noexcept override

6.31.1 Detailed Description

Exception class for handling file opening failures.

This exception is thrown when a file cannot be opened. It inherits from std::exception and overrides the what() method to provide a specific error message.

6.31.2 Member Function Documentation

6.31.2.1 what()

```
const char * failedToOpenFile::what ( ) const  [inline], [override], [private], [noexcept]
```

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

- /home/runner/work/R-Type/R-Type/ECS/Interface/Include/error_handling.hpp

6.32 FontManager Class Reference

Manages the loading and retrieval of font resources.

```
#include <font_manager.hpp>
```

Public Member Functions

- sf::Font & getFont (const std::string &filePath)

  Retrieves a font from the font manager.
- void releaseFont (const std::string &filePath)

  Releases the font associated with the given file path.

Private Attributes

- std::unordered_map< std::string, sf::Font > fonts

  A map that associates font names with their corresponding sf::Font objects.

6.32.1 Detailed Description

Manages the loading and retrieval of font resources.

The FontManager class provides functionality to load, retrieve, and release font resources. It maintains an internal storage of fonts, allowing for efficient management and reuse of font resources.

Example usage:

```
FontManager fontManager;
sf::Font &font = fontManager.getFont("path/to/font.ttf");
// Use the font...
fontManager.releaseFont("path/to/font.ttf");
```

6.32.2 Member Function Documentation

6.32.2.1 getFont()

```
sf::Font & FontManager::getFont (
            const std::string & filePath )  [inline]
```

Retrieves a font from the font manager.

This function attempts to find and return a font associated with the given file path. If the font is not already loaded, it will attempt to load it from the specified file path. If loading the font fails, an exception is thrown.

Parameters

filePath	The path to the font file.

Returns

A reference to the loaded sf::Font object.

Exceptions

failedToLoadFont	if the font cannot be loaded from the specified file path.

6.32.2.2 releaseFont()

```
void FontManager::releaseFont (
            const std::string & filePath )  [inline]
```

Releases the font associated with the given file path.

This function removes the font from the internal storage, effectively releasing any resources associated with it.

Parameters

filePath	The file path of the font to be released.

6.32.3 Member Data Documentation

6.32.3.1 fonts

```
std::unordered_map<std::string, sf::Font> FontManager::fonts  [private]
```

A map that associates font names with their corresponding sf::Font objects.

This unordered map uses strings as keys to store and retrieve sf::Font objects. It allows for efficient lookup, insertion, and deletion of font resources by name.

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

- /home/runner/work/R-Type/R-Type/ECS/Interface/Include/font_manager.hpp

6.33 ForceMissileComponent Struct Reference

Component representing a force missile in the ECS system.

```
#include <force_missile_component.hpp>
```

Public Attributes

- uint32_t forceId

  Unique identifier for the force missile.

6.33.1 Detailed Description

Component representing a force missile in the ECS system.

This component is used to identify and manage force missiles within the Entity-Component-System (ECS) architecture.

6.33.2 Member Data Documentation

6.33.2.1 forceId

```
ForceMissileComponent::forceId
```

Unique identifier for the force missile.

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

- /home/runner/work/R-Type/R-Type/ECS/Interface/Include/Components/force_missile_component.hpp

6.34 ForceWeaponComponent Struct Reference

Represents a component for a force weapon in the game.

```
#include <force_weapon_component.hpp>
```

Public Member Functions

- ForceWeaponComponent (uint32_t _playerId, uint32_t _level, uint32_t _attached)

  Constructs a new ForceWeaponComponent.

Public Attributes

- uint32_t playerId

  The ID of the player who owns the force weapon.
- uint32_t level

  The level of the force weapon.
- bool attached

  A flag indicating whether the force weapon is attached to the player.

6.34.1 Detailed Description

Represents a component for a force weapon in the game.

This component is used to manage the state and properties of a force weapon associated with a player.

6.34.2 Constructor & Destructor Documentation

6.34.2.1 ForceWeaponComponent()

```
ForceWeaponComponent::ForceWeaponComponent (
            uint32_t _playerId,
            uint32_t _level,
            uint32_t _attached )  [inline]
```

Constructs a new ForceWeaponComponent.

Parameters

_playerId	The ID of the player who owns the force weapon.
_level	The level of the force weapon.
_attached	A flag indicating whether the force weapon is attached to the player.

6.34.3 Member Data Documentation

6.34.3.1 attached

`ForceWeaponComponent::attached`

A flag indicating whether the force weapon is attached to the player.

6.34.3.2 level

`ForceWeaponComponent::level`

The level of the force weapon.

6.34.3.3 playerId

`ForceWeaponComponent::playerId`

The ID of the player who owns the force weapon.

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

- /home/runner/work/R-Type/R-Type/ECS/Interface/Include/Components/force_weapon_component.hpp

6.35 FrontComponent Struct Reference

A component that represents the front of an entity.

```
#include <front_component.hpp>
```

Public Member Functions

- FrontComponent (int _targetId)

Public Attributes

- int targetId

6.35.1 Detailed Description

A component that represents the front of an entity.

This component is used to identify the target entity that this component is associated with.

6.35.2 Constructor & Destructor Documentation

6.35.2.1 FrontComponent()

```
FrontComponent::FrontComponent (
            int _targetId )  [inline]
```

6.35.3 Member Data Documentation

6.35.3.1 targetId

```
int FrontComponent::targetId
```

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

- /home/runner/work/R-Type/R-Type/ECS/Interface/Include/Components/front_component.hpp

6.36 HealthComponent Struct Reference

Represents the health attributes of an entity.

```
#include <health_component.hpp>
```

Public Attributes

- int lives

6.36.1 Detailed Description

Represents the health attributes of an entity.

This component is used to store and manage the health-related data of an entity.

6.36.2 Member Data Documentation

6.36.2.1 lives

```
int HealthComponent::lives
```

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

- /home/runner/work/R-Type/R-Type/ECS/Interface/Include/Components/health_component.hpp

6.37 HitboxComponent Struct Reference

Represents the hitbox dimensions of an entity.

```
#include <hitbox_component.hpp>
```

Public Attributes

- int w

  Width of the hitbox.
- int h

  Height of the hitbox.

6.37.1 Detailed Description

Represents the hitbox dimensions of an entity.

This component is used to define the width and height of an entity's hitbox in the game. It is typically used for collision detection purposes.

6.37.2 Member Data Documentation

6.37.2.1 h

`HitboxComponent::h`

Height of the hitbox.

6.37.2.2 w

`HitboxComponent::w`

Width of the hitbox.

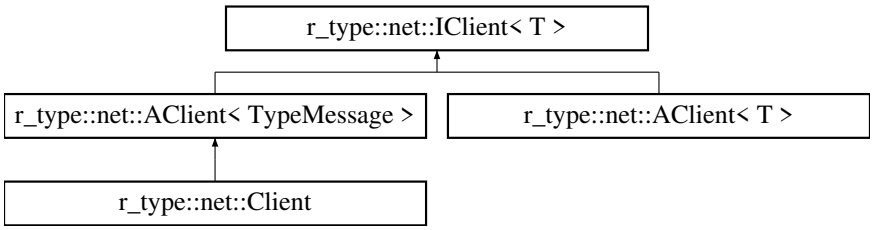
The documentation for this struct was generated from the following file:

- /home/runner/work/R-Type/R-Type/ECS/Interface/Include/Components/hitbox_component.hpp

6.38 r_type::net::IClient< T > Class Template Reference

```
#include <i_client.hpp>
```

Inheritance diagram for r_type::net::IClient< T >:



Public Member Functions

- IClient ()
- virtual ~IClient ()
- virtual bool Connect (const std::string &host, const uint16_t port)=0

  Connects to a remote host using UDP protocol.
- virtual void Disconnect ()=0

  Disconnects the client from the server.
- virtual bool IsConnected ()=0

  Checks if the client is connected to the server.
- virtual void Send (const Message< T > &msg)=0

  Send message to server.
- virtual ThreadSafeQueue< OwnedMessage< T > > & Incoming ()=0

  get incoming messages

6.38.1 Constructor & Destructor Documentation

6.38.1.1 IClient()

```
template<typename T >
r_type::net::IClient< T >::IClient ( ) [inline]
```

6.38.1.2 ~IClient()

```
template<typename T >
virtual r_type::net::IClient< T >::~IClient ( ) [inline], [virtual]
```

6.38.2 Member Function Documentation

6.38.2.1 Connect()

```
template<typename T >
virtual bool r_type::net::IClient< T >::Connect (
            const std::string & host,
            const uint16_t port ) [pure virtual]
```

Connects to a remote host using UDP protocol.

Parameters

host	The IP address or hostname of the remote host.
port	The port number of the remote host.

Returns

true if the connection is successful

false otherwise.

Implemented in r_type::net::AClient< T >, and r_type::net::AClient< TypeMessage >.

6.38.2.2 Disconnect()

```
template<typename T >
virtual void r_type::net::IClient< T >::Disconnect ( ) [pure virtual]
```

Disconnects the client from the server.

This function disconnects the client from the server if it is currently connected. It stops the context and joins the context thread. It also releases the connection resource.

Implemented in r_type::net::AClient< T >, and r_type::net::AClient< TypeMessage >.

6.38.2.3 Incoming()

```
template<typename T >
virtual ThreadSafeQueue< OwnedMessage< T > > & r_type::net::IClient< T >::Incoming ( ) [pure
virtual]
```

get incoming messages

Returns

ThreadSafeQueue<OwnedMessage<T>>&

Implemented in r_type::net::AClient< T >, and r_type::net::AClient< TypeMessage >.

6.38.2.4 IsConnected()

```
template<typename T >
virtual bool r_type::net::IClient< T >::IsConnected ( ) [pure virtual]
```

Checks if the client is connected to the server.

Returns

true

false

Implemented in r_type::net::AClient< T >, and r_type::net::AClient< TypeMessage >.

6.38.2.5 Send()

```
template<typename T >
virtual void r_type::net::IClient< T >::Send (
            const Message< T > & msg ) [pure virtual]
```

Send message to server.

Parameters

msg	

Implemented in r_type::net::AClient< T >.

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

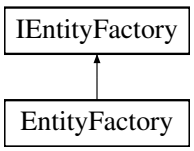
- /home/runner/work/R-Type/R-Type/Client/Interface/Include/Net/i_client.hpp

6.39 IEntityFactory Class Reference

The interface for an entity factory.

```
#include <i_entity_factory.hpp>
```

Inheritance diagram for IEntityFactory:



Public Types

- enum EnemyType { BasicMonster , ShooterEnemy , Wall , Boss }

  Enumeration representing different types of enemies in the game.

Public Member Functions

- virtual ~IEntityFactory ()=default

  Destroy the IEntityFactory object.
- virtual Entity createBackgroundLevelOne (EntityManager &entityManager, ComponentManager &component↩ Manager)=0

  Creates a background entity.
- virtual Entity createBackgroundLevelTwo (EntityManager &entityManager, ComponentManager &component↩ Manager)=0

  Create a Background Level Two object.
- virtual Entity createBackgroundLevelThree (EntityManager &entityManager, ComponentManager &component↩ Manager)=0

  Create a Background Level Three object.
- virtual Entity createBackgroundMenu (EntityManager &entityManager, ComponentManager &component↩ Manager, TextureManager &textureManager)=0

  Create a Background Menu object.
- virtual Entity createInfoBar (EntityManager &entityManager, ComponentManager &componentManager)=0

  Creates a bar entity.
- virtual Entity createPlayer (EntityManager &entityManager, ComponentManager &componentManager, int nbrOfPlayers)=0

Creates a player entity.

- virtual Entity createShooterEnemy (EntityManager &entityManager, ComponentManager &component↩ Manager, int posX, int posY)=0

  Creates a shooter enemy entity.

- virtual Entity createBasicMonster (EntityManager &entityManager, ComponentManager &component↩ Manager, int posX, int posY)=0

  Creates a basic monster entity.

- virtual Entity createPlayerMissile (EntityManager &entityManager, ComponentManager &component↩ Manager, uint32_t entityId)=0

  Creates a player missile entity.

- virtual Entity createForceWeapon (EntityManager &entityManager, ComponentManager &component↩ Manager, uint32_t entityId)=0

  Creates a Force Weapon entity.

- virtual Entity createForceMissile (EntityManager &entityManager, ComponentManager &componentManager, uint32_t entityId)=0

  Creates a Force Missile entity.

- virtual Entity createPowerUpBlueLaserCrystal (EntityManager &entityManager, ComponentManager &componentManager, int posX, int posY)=0

  Creates a Power-Up Blue Laser Crystal entity.

- virtual Entity createWall (EntityManager &entityManager, ComponentManager &componentManager, int posX, int posY)=0

  Creates a wall entity with the specified position.

- virtual Entity createEnemyMissile (EntityManager &entityManager, ComponentManager &component↩ Manager, uint32_t entityId)=0

  Creates an enemy missile entity.

- virtual Entity createButton (EntityManager &entityManager, ComponentManager &componentManager, TextureManager &textureManager, FontManager &fontManager, std::string text, std::function< IScenes *(AScenes *)> *onClick, float x, float y)=0

  Creates a button entity.

- virtual Entity createSmallButton (EntityManager &entityManager, ComponentManager &componentManager, TextureManager &textureManager, FontManager &fontManager, std::string text, std::function< IScenes *(AScenes *, AScenes::Actions)> *onClick, float x=0, float y=0)=0

  Creates a button entity with the specified properties.

- virtual Entity createUpdateButton (EntityManager &entityManager, ComponentManager &component↩ Manager, TextureManager &textureManager, FontManager &fontManager, std::string text, std::function< IScenes *(AScenes *)> *onClick, std::function< std::string(GameParameters)> *updateText, float x, float y)=0
- virtual Entity createTailSegment (EntityManager &entityManager, ComponentManager &component↩ Manager)=0
- virtual Entity createTailEnd (EntityManager &entityManager, ComponentManager &componentManager)=0

6.39.1 Detailed Description

The interface for an entity factory.

This interface defines the methods for creating different types of entities in the game. Each method takes references to the entity manager, component manager, and other necessary parameters, and returns an entity object.

Note

This is an abstract base class and cannot be instantiated directly.

6.39.2 Member Enumeration Documentation

6.39.2.1 EnemyType

```
enum IEntityFactory::EnemyType
```

Enumeration representing different types of enemies in the game.

This enumeration defines the various enemy types that can be instantiated in the game. Each type corresponds to a specific kind of enemy with unique behaviors and characteristics.

Enumerator

BasicMonster	
ShooterEnemy	
Wall	
Boss	

6.39.3 Constructor & Destructor Documentation

6.39.3.1 ~IEntityFactory()

```
virtual IEntityFactory::~IEntityFactory ( ) [virtual], [default]
```

Destroy the IEntityFactory object.

6.39.4 Member Function Documentation

6.39.4.1 createBackgroundLevelOne()

```
virtual Entity IEntityFactory::createBackgroundLevelOne (
            EntityManager & entityManager,
            ComponentManager & componentManager ) [pure virtual]
```

Creates a background entity.

This function creates a background entity using the provided entity manager and component manager.

Parameters

entityManager	The entity manager to use for creating the entity.
componentManager	The component manager to use for adding components to the entity.

Returns

The created background entity.

Implemented in EntityFactory.

6.39.4.2 createBackgroundLevelThree()

```
virtual Entity IEntityFactory::createBackgroundLevelThree (
            EntityManager & entityManager,
            ComponentManager & componentManager )  [pure virtual]
```

Create a Background Level Three object.

Parameters

entityManager	
componentManager	

Returns

Entity

Implemented in EntityFactory.

6.39.4.3 createBackgroundLevelTwo()

```
virtual Entity IEntityFactory::createBackgroundLevelTwo (
            EntityManager & entityManager,
            ComponentManager & componentManager )  [pure virtual]
```

Create a Background Level Two object.

Parameters

entityManager	
componentManager	

Returns

Entity

Implemented in EntityFactory.

6.39.4.4 createBackgroundMenu()

```
virtual Entity IEntityFactory::createBackgroundMenu (
            EntityManager & entityManager,
            ComponentManager & componentManager,
            TextureManager & textureManager )  [pure virtual]
```

Create a Background Menu object.

Parameters

entityManager	
componentManager	

Returns

Entity

Implemented in EntityFactory.

6.39.4.5 createBasicMonster()

```
virtual Entity IEntityFactory::createBasicMonster (
            EntityManager & entityManager,
            ComponentManager & componentManager,
            int posX,
            int posY )  [pure virtual]
```

Creates a basic monster entity.

This function creates a basic monster entity using the provided entity manager and component manager.

Parameters

entityManager	The entity manager used to create the entity.
componentManager	The component manager used to add components to the entity.

Returns

The created basic monster entity.

Implemented in EntityFactory.

6.39.4.6 createButton()

```
virtual Entity IEntityFactory::createButton (
            EntityManager & entityManager,
            ComponentManager & componentManager,
            TextureManager & textureManager,
            FontManager & fontManager,
            std::string text,
            std::function< IScenes *(AScenes *)> * onClick,
            float x,
            float y )  [pure virtual]
```

Creates a button entity.

This function creates a button entity using the provided entity manager, component manager, texture manager, text, and onClick function. The button entity represents a clickable button in the game.

Parameters

entityManager	The entity manager used to create the button entity.
componentManager	The component manager used to manage the components of the button entity.
textureManager	The texture manager used to load the textures for the button entity.
text	The text displayed on the button.
onClick	The function to be called when the button is clicked.

Returns

The created button entity.

Implemented in EntityFactory.

6.39.4.7 createEnemyMissile()

```
virtual Entity IEntityFactory::createEnemyMissile (
            EntityManager & entityManager,
            ComponentManager & componentManager,
            uint32_t entityId )  [pure virtual]
```

Creates an enemy missile entity.

This function creates an enemy missile entity using the provided entity manager and component manager.

Parameters

entityManager	The entity manager used to create the entity.
componentManager	The component manager used to add components to the entity.

Returns

The created enemy missile entity.

Implemented in EntityFactory.

6.39.4.8 createForceMissile()

```
virtual Entity IEntityFactory::createForceMissile (
            EntityManager & entityManager,
            ComponentManager & componentManager,
            uint32_t entityId )  [pure virtual]
```

Creates a Force Missile entity.

This function creates a Force Missile entity and registers it with the given EntityManager and ComponentManager. The entity is identified by the provided entityId.

Parameters

entityManager	Reference to the EntityManager that will manage the entity.
componentManager	Reference to the ComponentManager that will manage the components of the entity.
entityId	The unique identifier for the entity to be created.

Returns

Entity The created Force Missile entity.

Implemented in EntityFactory.

6.39.4.9 createForceWeapon()

```
virtual Entity IEntityFactory::createForceWeapon (
            EntityManager & entityManager,
            ComponentManager & componentManager,
            uint32_t entityId )  [pure virtual]
```

Creates a Force Weapon entity.

This function is responsible for creating a Force Weapon entity and adding it to the provided EntityManager and ComponentManager. The entity is identified by the given entityId.

Parameters

entityManager	Reference to the EntityManager that will manage the entity.
componentManager	Reference to the ComponentManager that will manage the components of the entity.
entityId	The unique identifier for the entity to be created.

Returns

Entity The created Force Weapon entity.

Implemented in EntityFactory.

6.39.4.10 createInfoBar()

```
virtual Entity IEntityFactory::createInfoBar (
            EntityManager & entityManager,
            ComponentManager & componentManager )  [pure virtual]
```

Creates a bar entity.

This function creates a bar with text for displaying player information like health and score.

Parameters

entityManager	The entity manager to use for creating the entity.
componentManager	The component manager to use for adding components to the entity.

Returns

The created bar entity.

Implemented in EntityFactory.

6.39.4.11 createPlayer()

```
virtual Entity IEntityFactory::createPlayer (
            EntityManager & entityManager,
            ComponentManager & componentManager,
            int nbrOfPlayers )  [pure virtual]
```

Creates a player entity.

This function creates a player entity using the provided entity manager and component manager.

Parameters

entityManager	The entity manager used to create the entity.
componentManager	The component manager used to add components to the entity.

Returns

The created player entity.

Implemented in EntityFactory.

6.39.4.12 createPlayerMissile()

```
virtual Entity IEntityFactory::createPlayerMissile (
            EntityManager & entityManager,
            ComponentManager & componentManager,
            uint32_t entityId )  [pure virtual]
```

Creates a player missile entity.

This function creates a player missile entity with the specified player ID and adds it to the entity manager. It also initializes the necessary components for the player missile entity using the component manager.

Parameters

entityId	The ID of the entity that shoot the missile.
entityManager	The entity manager to add the player missile entity to.
componentManager	The component manager to initialize the components for the player missile entity.

Returns

The created player missile entity.

Implemented in EntityFactory.

6.39.4.13 createPowerUpBlueLaserCrystal()

```
virtual Entity IEntityFactory::createPowerUpBlueLaserCrystal (
            EntityManager & entityManager,
            ComponentManager & componentManager,
            int posX,
            int posY )  [pure virtual]
```

Creates a Power-Up Blue Laser Crystal entity.

This function is responsible for creating an entity that represents a Power-Up Blue Laser Crystal in the game. It initializes the entity with the necessary components and registers it with the provided EntityManager and ComponentManager.

Parameters

entityManager	Reference to the EntityManager that will manage the entity.
componentManager	Reference to the ComponentManager that will manage the components of the entity.

Returns

Entity The created Power-Up Blue Laser Crystal entity.

Implemented in EntityFactory.

6.39.4.14 createShooterEnemy()

```
virtual Entity IEntityFactory::createShooterEnemy (
            EntityManager & entityManager,
            ComponentManager & componentManager,
            int posX,
            int posY )  [pure virtual]
```

Creates a shooter enemy entity.

This function creates a shooter enemy entity using the provided entity manager and component manager.

Parameters

entityManager	The entity manager used to create the entity.
componentManager	The component manager used to add components to the entity.

Returns

The created shooter enemy entity.

Implemented in EntityFactory.

6.39.4.15 createSmallButton()

```
virtual Entity IEntityFactory::createSmallButton (
            EntityManager & entityManager,
```

```
            ComponentManager & componentManager,
            TextureManager & textureManager,
            FontManager & fontManager,
            std::string text,
            std::function< IScenes *(AScenes *, AScenes::Actions)> * onClick,
            float x = 0,
            float y = 0 )  [pure virtual]
```

Creates a button entity with the specified properties.

Parameters

entityManager	Reference to the EntityManager responsible for managing entities.
componentManager	Reference to the ComponentManager responsible for managing components.
textureManager	Reference to the TextureManager responsible for managing textures.
fontManager	Reference to the FontManager responsible for managing fonts.
text	The text to be displayed on the button.
onClick	A pointer to a function that will be called when the button is clicked.
x	The x-coordinate position of the button.
y	The y-coordinate position of the button.

Returns

Entity The created button entity.

Implemented in EntityFactory.

6.39.4.16 createTailEnd()

```
virtual Entity IEntityFactory::createTailEnd (
            EntityManager & entityManager,
            ComponentManager & componentManager )  [pure virtual]
```

Implemented in EntityFactory.

6.39.4.17 createTailSegment()

```
virtual Entity IEntityFactory::createTailSegment (
            EntityManager & entityManager,
            ComponentManager & componentManager )  [pure virtual]
```

Implemented in EntityFactory.

6.39.4.18 createUpdateButton()

```
virtual Entity IEntityFactory::createUpdateButton (
            EntityManager & entityManager,
            ComponentManager & componentManager,
            TextureManager & textureManager,
            FontManager & fontManager,
            std::string text,
            std::function< IScenes *(AScenes *)> * onClick,
            std::function< std::string(GameParameters)> * updateText,
            float x,
            float y )  [pure virtual]
```

Implemented in EntityFactory.

6.39.4.19 createWall()

```
virtual Entity IEntityFactory::createWall (
            EntityManager & entityManager,
            ComponentManager & componentManager,
            int posX,
            int posY )  [pure virtual]
```

Creates a wall entity with the specified position.

Parameters

entityManager	Reference to the EntityManager that will manage the new entity.
componentManager	Reference to the ComponentManager that will manage the components of the new entity.
posX	The x-coordinate of the wall's position.
posY	The y-coordinate of the wall's position.

Returns

Entity The created wall entity.

Implemented in EntityFactory.

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

- /home/runner/work/R-Type/R-Type/ECS/Interface/Include/Entities/i_entity_factory.hpp

6.40 InputComponent Struct Reference

Component for handling input actions.

```
#include <input_component.hpp>
```

Public Attributes

- InputType input

  The current input action of the entity.

6.40.1 Detailed Description

Component for handling input actions.

This structure is used to store the current input action of an entity.

6.40.2 Member Data Documentation

6.40.2.1 input

`InputComponent::input`

The current input action of the entity.

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

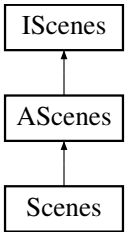
- /home/runner/work/R-Type/R-Type/ECS/Interface/Include/Components/input_component.hpp

6.41 IScenes Class Reference

Interface for managing different scenes in a game.

```
#include <i_scenes.hpp>
```

Inheritance diagram for IScenes:



Public Member Functions

- virtual ~IScenes ()=default
- virtual void mainMenu ()=0

  Displays the main menu and creates necessary entities.
- virtual void gameLoop ()=0

  Displays the main game loop and creates necessary entities.
- virtual void settingsMenu ()=0

  Displays the settings menu and creates necessary entities.
- virtual void inGameMenu ()=0

  Displays the in-game menu and creates necessary entities.
- virtual void difficultyChoices ()=0

  Displays the difficulty choices.
- virtual void render ()=0

  Displays the current scene and manages its components.
- virtual bool shouldQuit ()=0

  Checks if the game should quit.
- virtual sf::RenderWindow * getRenderWindow ()=0

  Gets the render window.

6.41.1 Detailed Description

Interface for managing different scenes in a game.

This interface declares the methods for displaying and managing various scenes in a game, such as the main menu, game loop, settings menu, and in-game menu.

6.41.2 Constructor & Destructor Documentation

6.41.2.1 ~IScenes()

```
virtual IScenes::~IScenes ( )  [virtual], [default]
```

6.41.3 Member Function Documentation

6.41.3.1 difficultyChoices()

```
virtual void IScenes::difficultyChoices ( )  [pure virtual]
```

Displays the difficulty choices.

Implemented in Scenes.

6.41.3.2 gameLoop()

```
virtual void IScenes::gameLoop ( )  [pure virtual]
```

Displays the main game loop and creates necessary entities.

Implemented in Scenes.

6.41.3.3 getRenderWindow()

```
virtual sf::RenderWindow * IScenes::getRenderWindow ( )  [pure virtual]
```

Gets the render window.

Returns

Pointer to the sf::RenderWindow.

Implemented in Scenes.

6.41.3.4 inGameMenu()

```
virtual void IScenes::inGameMenu ( )  [pure virtual]
```

Displays the in-game menu and creates necessary entities.

Implemented in Scenes.

6.41.3.5 mainMenu()

```
virtual void IScenes::mainMenu ( )  [pure virtual]
```

Displays the main menu and creates necessary entities.

Implemented in Scenes.

6.41.3.6 render()

```
virtual void IScenes::render ( )  [pure virtual]
```

Displays the current scene and manages its components.

Implemented in Scenes.

6.41.3.7 settingsMenu()

```
virtual void IScenes::settingsMenu ( )  [pure virtual]
```

Displays the settings menu and creates necessary entities.

Implemented in Scenes.

6.41.3.8 shouldQuit()

```
virtual bool IScenes::shouldQuit ( )  [pure virtual]
```

Checks if the game should quit.

Returns

True if the game should quit, false otherwise.

Implemented in Scenes.

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

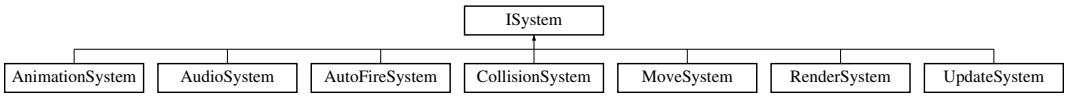
- /home/runner/work/R-Type/R-Type/ECS/Interface/Include/i_scenes.hpp

6.42 ISystem Class Reference

Interface for all systems in the ECS (Entity Component System) architecture.

```
#include <i_system.hpp>
```

Inheritance diagram for ISystem:



Public Member Functions

- ISystem ()=default
- virtual ~ISystem ()=default

6.42.1 Detailed Description

Interface for all systems in the ECS (Entity Component System) architecture.

This class serves as a base class for all systems within the ECS framework. Systems are responsible for processing entities that possess a specific set of components.

Note

This is an abstract class and should not be instantiated directly.

6.42.2 Constructor & Destructor Documentation

6.42.2.1 ISystem()

```
ISystem::ISystem ( ) [default]
```

6.42.2.2 ~ISystem()

```
virtual ISystem::~ISystem ( ) [virtual], [default]
```

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

- /home/runner/work/R-Type/R-Type/ECS/Interface/Include/Systems/i_system.hpp

6.43 labelComponent Struct Reference

Represents a label component with a name and position coordinates.

```
#include <label_component.hpp>
```

Public Attributes

- std::string name

  The name of the label.
- int x

  The x-coordinate of the label's position.
- int y

  The y-coordinate of the label's position.

6.43.1 Detailed Description

Represents a label component with a name and position coordinates.

This structure is used to define a label component in the ECS (Entity Component System). It contains a name and the x and y coordinates for positioning.

6.43.2 Member Data Documentation

6.43.2.1 name

```
labelComponent::name
```

The name of the label.

6.43.2.2 x

```
labelComponent::x
```

The x-coordinate of the label's position.

6.43.2.3 y

```
labelComponent::y
```

The y-coordinate of the label's position.

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

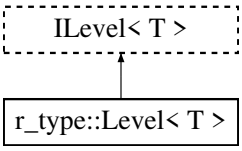
- /home/runner/work/R-Type/R-Type/ECS/Interface/Include/Components/label_component.hpp

6.44 r_type::Level< T > Class Template Reference

The Level class template manages the game level, including updating game state, handling collisions, and managing entities.

```
#include <level.hpp>
```

Inheritance diagram for r_type::Level< T >:



Public Member Functions

- Level ()=default
- ~Level ()=default
- void Update (r_type::net::AServer< T > *server, ComponentManager &componentManager, EntityManager &entityManager, std::chrono::system_clock::time_point newClock, bool *bUpdateEntities) override

  Updates the game state by processing entity movements, handling collisions, and sending messages to clients.
- void SetSystem (ComponentManager &componentManager, EntityManager &entityManager) override

  Initializes and sets up various systems for the level.
- void MoveUpdate (r_type::net::AServer< T > *server, ComponentManager &componentManager, EntityManager &entityManager, std::chrono::system_clock::time_point newClock) override

  Updates the positions of entities and notifies clients of any changes.
- void CollisionUpdate (r_type::net::AServer< T > *server, ComponentManager &componentManager, EntityManager &entityManager, std::chrono::system_clock::time_point newClock) override

  Updates the collision status of entities in the game.
- void AnimationUpdate (r_type::net::AServer< T > *server, ComponentManager &componentManager, EntityManager &entityManager, std::chrono::system_clock::time_point newClock) override

  Updates the animations of entities and sends messages to clients if animations have changed.
- void FireUpdate (r_type::net::AServer< T > *server, ComponentManager &componentManager, EntityManager &entityManager, std::chrono::system_clock::time_point newClock) override

  Updates the firing mechanism of entities in the game.
- void LevelOne (r_type::net::AServer< T > *server, ComponentManager &componentManager, EntityManager &entityManager, std::chrono::system_clock::time_point newClock) override

  Handles the spawning of entities for Level One.
- void LevelTwo (r_type::net::AServer< T > *server, ComponentManager &componentManager, EntityManager &entityManager, std::chrono::system_clock::time_point newClock) override

  Handles the spawning of entities for Level Two.
- void LevelThree (r_type::net::AServer< T > *server, ComponentManager &componentManager, EntityManager &entityManager, std::chrono::system_clock::time_point newClock) override

  Handles the spawning of entities for Level Three.
- void EndOfGame (r_type::net::AServer< T > *server, ComponentManager &componentManager, EntityManager &entityManager) override
- void SpawnEntity (r_type::net::AServer< T > *server, EntityManager &entityManager, ComponentManager &componentManager, int nbrOfEnemy, EntityFactory::EnemyType enemyType)

  Spawns a specified number of enemy entities in the game.
- EntityInformation GetEntityBackGround (r_type::net::AServer< T > *server, EntityManager &entityManager, ComponentManager &componentManager) override

  Get the Entity Back Ground object.
- void ChangeBackground (r_type::net::AServer< T > *server, EntityManager &entityManager, ComponentManager &componentManager) override

  Changes the background in the game by removing the current background entity and creating a new one.
- GameState GetLevel () override
- EntityInformation InitiateBackground (r_type::net::AServer< T > *server, EntityManager &entityManager, ComponentManager &componentManager) override

  Initializes a background entity.
- void SetGameParameters (GameParameters gameParameters)

  Sets the game difficulty based on the provided game parameters.
- void ChangeLevel (GameState state) override

  Changes the level of the game based on the provided game state.

Static Public Member Functions

- static bool collisionAction (r_type::net::AServer< T > *server, ComponentManager &componentManager, EntityManager &entityManager, std::vector< int > &entitiesToRemove, std::vector< int > &entitiesToAdd, uint32_t entityId1, uint32_t entityId2)

  Handles the collision action between two entities in the game.

Protected Attributes

- std::shared_ptr< MoveSystem > _moveSystem

  A shared pointer to the MoveSystem instance.
- std::shared_ptr< CollisionSystem > _collisionSystem

  A shared pointer to the CollisionSystem.
- std::shared_ptr< AnimationSystem > _animationSystem

  A shared pointer to the AnimationSystem.
- std::shared_ptr< AutoFireSystem > _autoFireSystem

  A shared pointer to an instance of AutoFireSystem.
- std::chrono::system_clock::time_point _basicMonsterSpawnTime

  Represents the time point at which a basic monster is spawned.
- std::chrono::system_clock::time_point _shooterEnemySpawnTime

  Represents the time point when the shooter enemy is spawned.
- std::chrono::system_clock::time_point _WallSpawnTime = std::chrono::system_clock::now()

  Stores the time point when the wall was spawned.
- std::chrono::system_clock::time_point _spawnTimeMonsterThree

  The time point at which the third type of monster is spawned.
- GameParameters _gameParameters

  Holds the parameters for the game configuration.

6.44.1 Detailed Description

template<typename T>
class r_type::Level< T >

The Level class template manages the game level, including updating game state, handling collisions, and managing entities.

This class template provides methods to update the game state, handle collisions, manage animations, and control the firing mechanisms of entities. It also includes functionality to spawn entities and set game parameters.

Template Parameters

T	The type of the server.

6.44.2 Constructor & Destructor Documentation

6.44.2.1 Level()

```
template<typename T >
r_type::Level< T >::Level ( ) [default]
```

6.44.2.2 ~Level()

```
template<typename T >
r_type::Level< T >::~Level ( ) [default]
```

6.44.3 Member Function Documentation

6.44.3.1 AnimationUpdate()

```
template<typename T >
void r_type::Level< T >::AnimationUpdate (
            r_type::net::AServer< T > * server,
            ComponentManager & componentManager,
            EntityManager & entityManager,
            std::chrono::system_clock::time_point newClock ) [inline], [override]
```

Updates the animations of entities and sends messages to clients if animations have changed.

This function performs the following steps:

1. Retrieves the current animation components from the component manager.
2. Saves the current state of animations.
3. Updates the animations using the animation system.
4. Compares the new state of animations with the previous state.
5. Sends messages to all clients if any animations have changed.

Parameters

server	Pointer to the server instance.
componentManager	Reference to the component manager.
entityManager	Reference to the entity manager.
newClock	The current time point.

6.44.3.2 ChangeBackground()

```
template<typename T >
void r_type::Level< T >::ChangeBackground (
            r_type::net::AServer< T > * server,
            EntityManager & entityManager,
            ComponentManager & componentManager ) [inline], [override]
```

Changes the background in the game by removing the current background entity and creating a new one.

This function sends messages to all clients to destroy the current background entity and create a new one.

Template Parameters

T	The type of the server.

Parameters

server	Pointer to the server instance.
entityManager	Reference to the EntityManager instance.
componentManager	Reference to the ComponentManager instance.

6.44.3.3 ChangeLevel()

```
template<typename T >
void r_type::Level< T >::ChangeLevel (
            GameState state )  [inline], [override]
```

Changes the level of the game based on the provided game state.

This function changes the level of the game based on the provided game state.

Parameters

state	The game state to change the level to.

6.44.3.4 collisionAction()

```
template<typename T >
static bool r_type::Level< T >::collisionAction (
            r_type::net::AServer< T > * server,
            ComponentManager & componentManager,
            EntityManager & entityManager,
            std::vector< int > & entitiesToRemove,
            std::vector< int > & entitiesToAdd,
            uint32_t entityId1,
            uint32_t entityId2 )  [inline], [static]
```

Handles the collision action between two entities in the game.

This function determines the type of collision between two entities and performs the appropriate actions based on the components of the entities involved. It updates the health, score, and other relevant components, and manages the addition and removal of entities from the game.

Template Parameters

T	The type of the server.

Parameters

server	Pointer to the server instance.

Parameters

componentManager	Reference to the ComponentManager.
entityManager	Reference to the EntityManager.
entitiesToRemove	Vector of entity IDs to be removed from the game.
entitiesToAdd	Vector of entity IDs to be added to the game.
entityId1	The ID of the first entity involved in the collision.
entityId2	The ID of the second entity involved in the collision.

Returns

True if a collision was handled, false otherwise.

6.44.3.5 CollisionUpdate()

```
template<typename T >
void r_type::Level< T >::CollisionUpdate (
            r_type::net::AServer< T > * server,
            ComponentManager & componentManager,
            EntityManager & entityManager,
            std::chrono::system_clock::time_point newClock )  [inline], [override]
```

Updates the collision status of entities in the game.

This function checks for collisions between entities and handles the consequences of those collisions, such as updating health, removing entities, and adding new entities. It also handles entities that go off-screen.

Parameters

server	Pointer to the server instance.
componentManager	Reference to the component manager.
entityManager	Reference to the entity manager.
newClock	The current time point for the update.

6.44.3.6 EndOfGame()

```
template<typename T >
void r_type::Level< T >::EndOfGame (
            r_type::net::AServer< T > * server,
            ComponentManager & componentManager,
            EntityManager & entityManager )  [inline], [override]
```

6.44.3.7 FireUpdate()

```
template<typename T >
void r_type::Level< T >::FireUpdate (
            r_type::net::AServer< T > * server,
```

```
            ComponentManager & componentManager,
            EntityManager & entityManager,
            std::chrono::system_clock::time_point newClock )  [inline], [override]
```

Updates the firing mechanism of entities in the game.

This function handles the automatic firing system and processes the firing logic for entities. It retrieves all entities and checks if they can shoot. If an entity can shoot, it sends a message to all clients to create an enemy missile and sets the entity's canShoot flag to false.

Parameters

server	Pointer to the server instance.
componentManager	Reference to the ComponentManager handling components.
entityManager	Reference to the EntityManager handling entities.
newClock	The current time point used for timing events.

6.44.3.8 GetEntityBackGround()

```
template<typename T >
EntityInformation r_type::Level< T >::GetEntityBackGround (
            r_type::net::AServer< T > * server,
            EntityManager & entityManager,
            ComponentManager & componentManager )  [inline], [override]
```

Get the Entity Back Ground object.

Parameters

server	
entityManager	
componentManager	

Returns

EntityInformation

6.44.3.9 GetLevel()

```
template<typename T >
GameState r_type::Level< T >::GetLevel ( )  [inline], [override]
```

6.44.3.10 InitiateBackground()

```
template<typename T >
EntityInformation r_type::Level< T >::InitiateBackground (
            r_type::net::AServer< T > * server,
            EntityManager & entityManager,
            ComponentManager & componentManager )  [inline], [override]
```

Initializes a background entity.

The function creates and returns information about the background entity.

Returns

EntityInformation The information of the background entity.

6.44.3.11 LevelOne()

```
template<typename T >
void r_type::Level< T >::LevelOne (
            r_type::net::AServer< T > * server,
            ComponentManager & componentManager,
            EntityManager & entityManager,
            std::chrono::system_clock::time_point newClock )  [inline], [override]
```

Handles the spawning of entities for Level One.

This function is responsible for spawning basic monsters and shooter enemies at specific intervals defined by the game parameters. It checks the elapsed time since the last spawn of each entity type and spawns new entities if the required time has passed.

Parameters

server	Pointer to the server instance.
componentManager	Reference to the ComponentManager instance.
entityManager	Reference to the EntityManager instance.
newClock	The current time point used for timing calculations.

6.44.3.12 LevelThree()

```
template<typename T >
void r_type::Level< T >::LevelThree (
            r_type::net::AServer< T > * server,
            ComponentManager & componentManager,
            EntityManager & entityManager,
            std::chrono::system_clock::time_point newClock )  [inline], [override]
```

Handles the spawning of entities for Level Three.

This function is responsible for spawning basic monsters and shooter enemies at specific intervals defined by the game parameters. It checks the elapsed time since the last spawn of each entity type and spawns new entities if the required time has passed.

Parameters

server	Pointer to the server instance.
componentManager	Reference to the ComponentManager instance.
entityManager	Reference to the EntityManager instance.
newClock	The current time point used for timing calculations.

6.44.3.13 LevelTwo()

```
template<typename T >
void r_type::Level< T >::LevelTwo (
            r_type::net::AServer< T > * server,
            ComponentManager & componentManager,
            EntityManager & entityManager,
            std::chrono::system_clock::time_point newClock )  [inline], [override]
```

Handles the spawning of entities for Level Two.

This function is responsible for spawning basic monsters and shooter enemies at specific intervals defined by the game parameters. It checks the elapsed time since the last spawn of each entity type and spawns new entities if the required time has passed.

Parameters

server	Pointer to the server instance.
componentManager	Reference to the ComponentManager instance.
entityManager	Reference to the EntityManager instance.
newClock	The current time point used for timing calculations.

6.44.3.14 MoveUpdate()

```
template<typename T >
void r_type::Level< T >::MoveUpdate (
            r_type::net::AServer< T > * server,
            ComponentManager & componentManager,
            EntityManager & entityManager,
            std::chrono::system_clock::time_point newClock )  [inline], [override]
```

Updates the positions of entities and notifies clients of any changes.

This function performs the following steps:

1. Retrieves the current positions of entities and stores them.
2. Moves the entities using the move system.
3. Compares the new positions with the previous positions.
4. If an entity's position has changed, sends an update message to all clients.

Parameters

server	Pointer to the server instance.
componentManager	Reference to the ComponentManager.
entityManager	Reference to the EntityManager.
newClock	The current time point.

6.44.3.15 SetGameParameters()

```
template<typename T >
void r_type::Level< T >::SetGameParameters (
            GameParameters gameParameters )  [inline]
```

Sets the game difficulty based on the provided game parameters.

This function sets the game difficulty based on the provided game parameters.

Parameters

gameParameters	The game parameters to set the difficulty.

6.44.3.16 SetSystem()

```
template<typename T >
void r_type::Level< T >::SetSystem (
            ComponentManager & componentManager,
            EntityManager & entityManager )  [inline], [override]
```

Initializes and sets up various systems for the level.

This function overrides a base class method to initialize and set up the MoveSystem, CollisionSystem, AnimationSystem, and AutoFireSystem using the provided ComponentManager and EntityManager.

Parameters

componentManager	Reference to the ComponentManager used to manage components.
entityManager	Reference to the EntityManager used to manage entities.

6.44.3.17 SpawnEntity()

```
template<typename T >
void r_type::Level< T >::SpawnEntity (
            r_type::net::AServer< T > * server,
            EntityManager & entityManager,
            ComponentManager & componentManager,
            int nbrOfEnemy,
            EntityFactory::EnemyType enemyType )  [inline]
```

Spawns a specified number of enemy entities in the game.

This function creates and spawns a specified number of enemy entities of a given type at random positions within the game world. The enemy entities are then broadcasted to all connected clients.

Template Parameters

T	The type of the server.

Parameters

server	A pointer to the server instance.
entityManager	Reference to the EntityManager responsible for managing entities.
componentManager	Reference to the ComponentManager responsible for managing components.
nbrOfEnemy	The number of enemy entities to spawn.
enemyType	The type of enemy to spawn (e.g., BasicMonster, ShooterEnemy).

6.44.3.18 Update()

```
template<typename T >
void r_type::Level< T >::Update (
            r_type::net::AServer< T > * server,
            ComponentManager & componentManager,
            EntityManager & entityManager,
            std::chrono::system_clock::time_point newClock,
            bool * bUpdateEntities )  [inline], [override]
```

Updates the game state by processing entity movements, handling collisions, and sending messages to clients.

This function performs several tasks to update the game state:

- Moves entities based on the elapsed time.
- Handles collisions between entities.
- Sends messages to clients about destroyed entities.
- Updates animations and firing mechanisms.

Parameters

server	Pointer to the server instance.
componentManager	Reference to the ComponentManager handling game components.
entityManager	Reference to the EntityManager handling game entities.
newClock	The current time point used to calculate elapsed time.
bUpdateEntities	Pointer to a boolean flag indicating whether entities should be updated.

6.44.4 Member Data Documentation

6.44.4.1 _animationSystem

```
template<typename T >
std::shared_ptr<AnimationSystem> r_type::Level< T >::_animationSystem  [protected]
```

A shared pointer to the AnimationSystem.

This member variable holds a shared pointer to an instance of the AnimationSystem. The AnimationSystem is responsible for managing and updating animations within the game. Using a shared pointer ensures that the AnimationSystem instance is properly managed and its lifetime is controlled, preventing memory leaks and dangling pointers.

6.44.4.2 _autoFireSystem

```
template<typename T >
std::shared_ptr<AutoFireSystem> r_type::Level< T >::_autoFireSystem [protected]
```

A shared pointer to an instance of AutoFireSystem.

This member variable holds a shared pointer to an AutoFireSystem object, which is responsible for managing the automatic firing mechanism in the game. The use of std::shared_ptr ensures that the AutoFireSystem instance is properly managed and deallocated when no longer in use.

6.44.4.3 _basicMonsterSpawnTime

```
template<typename T >
std::chrono::system_clock::time_point r_type::Level< T >::_basicMonsterSpawnTime [protected]
```

Initial value:

```
=
        std::chrono::system_clock::now()
```

Represents the time point at which a basic monster is spawned.

This variable is initialized to the current system time using std::chrono::system_clock::now(). It is used to track the spawn time of a basic monster in the game.

6.44.4.4 _collisionSystem

```
template<typename T >
std::shared_ptr<CollisionSystem> r_type::Level< T >::_collisionSystem [protected]
```

A shared pointer to the CollisionSystem.

This member variable holds a shared pointer to an instance of the CollisionSystem, which is responsible for handling collision detection and response within the game. Using a shared pointer ensures that the CollisionSystem instance is properly managed and its memory is automatically deallocated when no longer in use.

6.44.4.5 _gameParameters

```
template<typename T >
GameParameters r_type::Level< T >::_gameParameters [protected]
```

Holds the parameters for the game configuration.

This member variable stores an instance of the GameParameters class, which contains various settings and configurations for the game.

6.44.4.6 _moveSystem

```
template<typename T >
std::shared_ptr<MoveSystem> r_type::Level< T >::_moveSystem [protected]
```

A shared pointer to the MoveSystem instance.

This member variable holds a shared pointer to an instance of the MoveSystem class, which is responsible for handling movement logic within the system. Using a shared pointer ensures that the MoveSystem instance is properly managed and its lifetime is tied to the number of references to it.

6.44.4.7 _shooterEnemySpawnTime

```
template<typename T >
std::chrono::system_clock::time_point r_type::Level< T >::_shooterEnemySpawnTime [protected]
```

Initial value:

```
=
        std::chrono::system_clock::now()
```

Represents the time point when the shooter enemy is spawned.

This variable is initialized to the current system time using std::chrono::system_clock::now(). It is used to track the exact moment when the shooter enemy is spawned in the game.

6.44.4.8 _spawnTimeMonsterThree

```
template<typename T >
std::chrono::system_clock::time_point r_type::Level< T >::_spawnTimeMonsterThree [protected]
```

The time point at which the third type of monster is spawned.

This member variable holds the spawn time for the third type of monster using the system clock's time point. It is used to schedule or track when the third type of monster should appear in the game.

6.44.4.9 _WallSpawnTime

```
template<typename T >
std::chrono::system_clock::time_point r_type::Level< T >::_WallSpawnTime = std::chrono::system↩
_clock::now() [protected]
```

Stores the time point when the wall was spawned.

This member variable holds the time point, using the system clock, at which the wall was spawned. It is initialized to the current time when the object is created.

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

- /home/runner/work/R-Type/R-Type/Server/Interface/Include/level.hpp

6.45 LinkForceComponent Struct Reference

Component that links an entity to a target entity by ID.

```
#include <link_force_component.hpp>
```

Public Member Functions

- LinkForceComponent (int _targetId)

  Constructs a LinkForceComponent with the specified target entity ID.

Public Attributes

- int targetId

  The ID of the target entity to which this entity is linked.

6.45.1 Detailed Description

Component that links an entity to a target entity by ID.

This component is used to establish a link between the current entity and a target entity identified by the targetId. This can be useful in scenarios where entities need to interact or be aware of each other.

6.45.2 Constructor & Destructor Documentation

6.45.2.1 LinkForceComponent()

```
LinkForceComponent::LinkForceComponent (
            int _targetId )  [inline]
```

Constructs a LinkForceComponent with the specified target entity ID.

Parameters

_↩ *targetId*	The ID of the target entity.

6.45.3 Member Data Documentation

6.45.3.1 targetId

```
LinkForceComponent::targetId
```

The ID of the target entity to which this entity is linked.

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

- /home/runner/work/R-Type/R-Type/ECS/Interface/Include/Components/link_force_component.hpp

6.46 MovementComponent Struct Reference

Represents a component that handles movement in the ECS system.

```
#include <movement_component.hpp>
```

Public Member Functions

- MovementComponent ()
- MovementComponent (MovementType movementType, uint32_t index, bool move)

  Constructs a MovementComponent with the specified parameters.

Public Attributes

- MovementType movementType

  The type of movement to be applied.
- uint32_t index

  An index used to identify the movement component.
- bool move

  A boolean flag indicating whether the entity should move.

6.46.1 Detailed Description

Represents a component that handles movement in the ECS system.

This component is used to define the movement behavior of an entity.

6.46.2 Constructor & Destructor Documentation

6.46.2.1 MovementComponent() [1/2]

```
MovementComponent::MovementComponent ( ) [inline]
```

6.46.2.2 MovementComponent() [2/2]

```
MovementComponent::MovementComponent (
            MovementType movementType,
            uint32_t index,
            bool move ) [inline]
```

Constructs a MovementComponent with the specified parameters.

Parameters

movementType	The type of movement to be applied.
index	An index used to identify the movement component.
move	A boolean flag indicating whether the entity should move.

6.46.3 Member Data Documentation

6.46.3.1 index

`MovementComponent::index`

An index used to identify the movement component.

6.46.3.2 move

`MovementComponent::move`

A boolean flag indicating whether the entity should move.

6.46.3.3 movementType

`MovementComponent::movementType`

The type of movement to be applied.

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

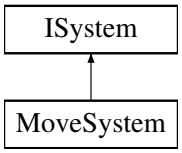
- /home/runner/work/R-Type/R-Type/ECS/Interface/Include/Components/movement_component.hpp

6.47 MoveSystem Class Reference

System responsible for moving entities within the ECS framework.

`#include <move_system.hpp>`

Inheritance diagram for MoveSystem:



Public Member Functions

- MoveSystem (ComponentManager &componentManager, EntityManager &entityManager)

  Constructs a new MoveSystem object.
- void moveEntities (ComponentManager &componentManager, EntityManager &entityManager)

  Moves all entities managed by the system.
- void moveEntity (ComponentManager &componentManager, int entityId)

  Moves a single entity.

Public Member Functions inherited from ISystem

- ISystem ()=default
- virtual ~ISystem ()=default

Private Attributes

- ComponentManager & _componentManager

  Reference to the ComponentManager instance.
- EntityManager & _entityManager

  Reference to the EntityManager instance.

6.47.1 Detailed Description

System responsible for moving entities within the ECS framework.

The MoveSystem class handles the movement logic for entities in the game. It interacts with the ComponentManager and EntityManager to update the positions of entities based on their movement components.

6.47.2 Constructor & Destructor Documentation

6.47.2.1 MoveSystem()

```
MoveSystem::MoveSystem (
            ComponentManager & componentManager,
            EntityManager & entityManager )  [inline]
```

Constructs a new MoveSystem object.

Parameters

componentManager	Reference to the ComponentManager.
entityManager	Reference to the EntityManager.

6.47.3 Member Function Documentation

6.47.3.1 moveEntities()

```
void MoveSystem::moveEntities (
            ComponentManager & componentManager,
            EntityManager & entityManager )
```

Moves all entities managed by the system.

This function iterates through all entities and updates their positions based on their movement components.

Parameters

componentManager	Reference to the ComponentManager.
entityManager	Reference to the EntityManager.

6.47.3.2 moveEntity()

```
void MoveSystem::moveEntity (
            ComponentManager & componentManager,
            int entityId )
```

Moves a single entity.

This function updates the position of a specific entity based on its movement component.

Parameters

componentManager	Reference to the ComponentManager.
entityId	The ID of the entity to be moved.

6.47.4 Member Data Documentation

6.47.4.1 _componentManager

```
ComponentManager& MoveSystem::_componentManager [private]
```

Reference to the ComponentManager instance.

This member variable holds a reference to the ComponentManager, which is responsible for managing all the components within the ECS (Entity Component System). It is used by the system to access and manipulate components associated with entities.

6.47.4.2 _entityManager

```
EntityManager& MoveSystem::_entityManager [private]
```

Reference to the EntityManager instance.

This member variable holds a reference to the EntityManager, which is responsible for managing all entities within the ECS (Entity Component System). It provides functionalities to create, destroy, and manage entities and their components.

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

- /home/runner/work/R-Type/R-Type/ECS/Interface/Include/Systems/move_system.hpp
- /home/runner/work/R-Type/R-Type/ECS/Src/Systems/move_system.cpp

6.48 OffsetComponent Struct Reference

Component that represents an offset value.

```
#include <offset_component.hpp>
```

Public Attributes

- float offset

6.48.1 Detailed Description

Component that represents an offset value.

This component is used to store a floating-point offset value.

6.48.2 Member Data Documentation

6.48.2.1 offset

```
float OffsetComponent::offset
```

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

- /home/runner/work/R-Type/R-Type/ECS/Interface/Include/Components/offset_component.hpp

6.49 OnClickComponent Struct Reference

Component that handles click events.

```
#include <on_click_component.hpp>
```

Public Member Functions

- OnClickComponent (std::function< IScenes *(AScenes *)> onClickFunction)

  Constructs an OnClickComponent with the specified click handler function.

Public Attributes

- bool isClicked = false

  Indicates whether the entity has been clicked.
- std::function< IScenes *(AScenes *)> onClick

  A function that is executed when the entity is clicked.

6.49.1 Detailed Description

Component that handles click events.

This component is used to determine if an entity has been clicked and to execute a specified function when the entity is clicked.

6.49.2 Constructor & Destructor Documentation

6.49.2.1 OnClickComponent()

```
OnClickComponent::OnClickComponent (
            std::function< IScenes *(AScenes *)> onClickFunction )  [inline]
```

Constructs an OnClickComponent with the specified click handler function.

Parameters

onClickFunction	The function to be executed when the entity is clicked.

6.49.3 Member Data Documentation

6.49.3.1 isClicked

```
OnClickComponent::isClicked = false
```

Indicates whether the entity has been clicked.

6.49.3.2 onClick

```
OnClickComponent::onClick
```

A function that is executed when the entity is clicked.

The function takes a pointer to an AScenes object and returns a pointer to an IScenes object.

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

- /home/runner/work/R-Type/R-Type/ECS/Interface/Include/Components/on_click_component.hpp

6.50 PlayerComponent Struct Reference

```
#include <player_component.hpp>
```

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

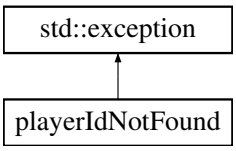
- /home/runner/work/R-Type/R-Type/ECS/Interface/Include/Components/player_component.hpp

6.51 playerIdNotFound Class Reference

Exception class for handling cases where a player ID is not found.

```
#include <error_handling.hpp>
```

Inheritance diagram for playerIdNotFound:



Private Member Functions

- const char ∗ what () const noexcept override

6.51.1 Detailed Description

Exception class for handling cases where a player ID is not found.

This exception is thrown when a requested player ID cannot be found in the system. It inherits from the standard std::exception class and overrides the what() method to provide a specific error message.

6.51.2 Member Function Documentation

6.51.2.1 what()

```
const char * playerIdNotFound::what ( ) const  [inline], [override], [private], [noexcept]
```

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

- /home/runner/work/R-Type/R-Type/ECS/Interface/Include/error_handling.hpp

6.52 PlayerMissileComponent Struct Reference

Component that represents a missile belonging to a player.

```
#include <player_missile_component.hpp>
```

Public Attributes

- uint32_t playerId

  The unique identifier of the player who fired the missile.

6.52.1 Detailed Description

Component that represents a missile belonging to a player.

This component is used to identify missiles that are fired by players in the game.

6.52.2 Member Data Documentation

6.52.2.1 playerId

```
PlayerMissileComponent::playerId
```

The unique identifier of the player who fired the missile.

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

- /home/runner/work/R-Type/R-Type/ECS/Interface/Include/Components/player_missile_component.hpp

6.53 PositionComponent Struct Reference

A component that represents the position of an entity in 2D space.

```
#include <position_component.hpp>
```

Public Member Functions

- PositionComponent (float _x, float _y)

Public Attributes

- float x
- float y

6.53.1 Detailed Description

A component that represents the position of an entity in 2D space.

6.53.2 Constructor & Destructor Documentation

6.53.2.1 PositionComponent()

```
PositionComponent::PositionComponent (
            float _x,
            float _y )  [inline]
```

6.53.3 Member Data Documentation

6.53.3.1 x

```
float PositionComponent::x
```

6.53.3.2 y

```
float PositionComponent::y
```

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

- /home/runner/work/R-Type/R-Type/ECS/Interface/Include/Components/position_component.hpp

6.54 PowerUpComponent Struct Reference

```
#include <power_up_component.hpp>
```

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

- /home/runner/work/R-Type/R-Type/ECS/Interface/Include/Components/power_up_component.hpp

6.55 RectangleShapeComponent Struct Reference

A component that holds an sf::RectangleShape.

```
#include <rectangleShapeComponent.hpp>
```

Public Member Functions

- RectangleShapeComponent (sf::RectangleShape &rectangleShape)

  Constructs a new RectangleShapeComponent.

Public Attributes

- sf::RectangleShape rectangleShape

6.55.1 Detailed Description

A component that holds an sf::RectangleShape.

This component is used to store and manage an sf::RectangleShape object.

6.55.2 Constructor & Destructor Documentation

6.55.2.1 RectangleShapeComponent()

```
RectangleShapeComponent::RectangleShapeComponent (
            sf::RectangleShape & rectangleShape )  [inline]
```

Constructs a new RectangleShapeComponent.

Parameters

rectangleShape	A reference to an sf::RectangleShape object.

6.55.3 Member Data Documentation

6.55.3.1 rectangleShape

`sf::RectangleShape RectangleShapeComponent::rectangleShape`

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

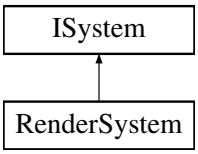
- /home/runner/work/R-Type/R-Type/ECS/Interface/Include/Components/rectangleShapeComponent.hpp

6.56 RenderSystem Class Reference

A system responsible for rendering entities in the ECS framework.

`#include <render_system.hpp>`

Inheritance diagram for RenderSystem:



Public Member Functions

- RenderSystem (sf::RenderWindow &window, ComponentManager &componentManager)
- void render (ComponentManager &componentManager)

  Renders the components managed by the ComponentManager.

Public Member Functions inherited from ISystem

- ISystem ()=default
- virtual ~ISystem ()=default

Private Attributes

- sf::RenderWindow & _window

  Reference to the SFML RenderWindow used for rendering.
- ComponentManager & _componentManager

  Reference to the ComponentManager instance.
- sf::Font _font

  A font object used for drawing text in SFML.

6.56.1 Detailed Description

A system responsible for rendering entities in the ECS framework.

This class handles the rendering of entities using the SFML library. It requires a reference to an SFML Render↩ Window and a ComponentManager.

Parameters

window	Reference to the SFML RenderWindow where entities will be rendered.
componentManager	Reference to the ComponentManager that manages entity components.

Exceptions

failedToLoadFont	Exception thrown if the font file cannot be loaded.

6.56.2 Constructor & Destructor Documentation

6.56.2.1 RenderSystem()

```
RenderSystem::RenderSystem (
            sf::RenderWindow & window,
            ComponentManager & componentManager )  [inline]
```

6.56.3 Member Function Documentation

6.56.3.1 render()

```
void RenderSystem::render (
            ComponentManager & componentManager )
```

Renders the components managed by the ComponentManager.

This function iterates through the components in the provided ComponentManager and performs rendering operations on them. It is typically called once per frame to update the visual representation of the components.

Parameters

componentManager	A reference to the ComponentManager that holds the components to be rendered.

6.56.4 Member Data Documentation

6.56.4.1 _componentManager

```
ComponentManager& RenderSystem::_componentManager  [private]
```

Reference to the ComponentManager instance.

This member variable holds a reference to the ComponentManager, which is responsible for managing all the components within the ECS (Entity Component System). It provides functionalities to add, remove, and access components associated with entities.

6.56.4.2 _font

```
sf::Font RenderSystem::_font [private]
```

A font object used for drawing text in SFML.

This member variable holds an instance of sf::Font, which is used to load and manage font resources for rendering text in the application. The sf::Font class provides functionality to load fonts from files, memory, or streams, and to retrieve font metrics and glyphs for text rendering.

6.56.4.3 _window

```
sf::RenderWindow& RenderSystem::_window [private]
```

Reference to the SFML RenderWindow used for rendering.

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

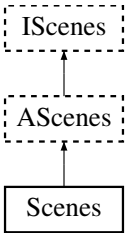
- /home/runner/work/R-Type/R-Type/ECS/Interface/Include/Systems/render_system.hpp
- /home/runner/work/R-Type/R-Type/ECS/Src/Systems/render_system.cpp

6.57 Scenes Class Reference

Represents a class that manages different scenes in a game.

```
#include <scenes.hpp>
```

Inheritance diagram for Scenes:



Public Member Functions

- Scenes (std::string ip, int port)

  Construct a new Scenes object.
- ~Scenes ()=default

  Destroy the Scenes object.
- void mainMenu ()

  displays the main menu, creates all the necessary entities
- void gameLoop ()

  displays the main game loop, creates all the necessary entities

- void HandleMessage (r_type::net::Message< TypeMessage > &msg, ComponentManager &component↩ Manager, TextureManager &textureManager, FontManager &fontManager, std::shared_ptr< AudioSystem > &audioSystem)
- void StopGameLoop (std::shared_ptr< AudioSystem > &audioSystem)
- void settingsMenu ()

  displays the settings menu, creates all the necessary entities
- void inGameMenu ()

  displays the in game menu, creates all the necessary entities
- void difficultyChoices ()

  displays the difficulty choices, creates all the necessary entities
- void difficultyChoicesCustomization ()
- void render ()

  display what must be displayed (main menu, game loop, settings menu, in game menu), creates all the components needed and manages them
- bool shouldQuit ()

  check if game should stop running
- sf::RenderWindow * getRenderWindow ()

  Get the RenderWindow object.
- void TransitionLevel ()
- void HandleTransitionLevelMessage (r_type::net::Message< TypeMessage > &msg, ComponentManager &componentManager, TextureManager &textureManager)
- void run ()

Public Member Functions inherited from AScenes

- AScenes (std::string ip, int port)
- ~AScenes ()=default
- void setScene (Scene scene)

  Set the Scene object.
- AScenes::Scene getPreviousScene ()

  Get the Previous Scene object.
- DaltonismMode getDaltonism () const

  Get the Daltonism object.
- void setDaltonism (DaltonismMode const mode)

  Set the Daltonism object.
- void setGameMode (GameParameters const mode)

  Set the Game Mode object.
- GameParameters getGameMode () const

  Get the Game Mode object.
- void setDisplayDaltonismChoice (bool const displayDaltonismChoice)

  Sets the display option for Daltonism mode.
- bool getDisplayDaltonismChoice () const

  Retrieves the current display setting for Daltonism (color blindness) mode.
- void setDisplayGameModeChoice (bool const displayGameModeChoice)

  Sets the display state for the game mode choice.
- bool getDisplayGameModeChoice () const

  Retrieves the current display game mode choice.
- void setDisplayKeyBindsChoice (bool const displayKeyBindsChoice)

  Sets the display status for key binds choice.
- bool getDisplayKeyBindsChoice () const

  Retrieves the current choice for displaying key bindings.

- void setIp (std::string ip)

  Sets the IP address.
- void setPort (int port)

  Sets the port number for the connection.
- std::string getIp () const

  Retrieves the IP address.
- int getPort () const

  Retrieves the port number.
- void SetPlayerReady (bool ready)

  Sets the player ready status.
- bool GetPlayerReady () const

  Retrieves the player ready status.

Public Member Functions inherited from IScenes

- virtual ~IScenes ()=default

Public Attributes

- sf::RenderWindow _window
- r_type::net::Client _networkClient

Public Attributes inherited from AScenes

- std::map< Actions, sf::Keyboard::Key > keyBinds

  A map that binds game actions to specific keyboard keys.
- std::vector< std::shared_ptr< Entity > > buttons

  A collection of shared pointers to Entity objects representing buttons.
- std::shared_ptr< Entity > filter

  A shared pointer to an Entity object.

Additional Inherited Members

Public Types inherited from AScenes

- enum class Scene {
  MAIN_MENU , GAME_LOOP , SETTINGS_MENU , IN_GAME_MENU ,
  CHOOSE_DIFFICULTY , CUSTOM_DIFFICULTY , TRANSITION_LEVEL , EXIT }

  Represents the different scenes in the R-Type client application.
- enum class GameMode { EASY , MEDIUM , HARD }

  Enumeration to represent different game difficulty levels.
- enum class DaltonismMode { NORMAL , TRITANOPIA , DEUTERANOPIA , PROTANOPIA }

  Enum representing different modes of color blindness (Daltonism).
- enum class Actions {
  UP , DOWN , LEFT , RIGHT ,
  FIRE , PAUSE , QUIT }

  Enumeration representing possible actions in the game.
- enum class SpriteType {
  BACKGROUND , PLAYER , ALLY , ENEMY ,
  FILTER , WEAPON , POWER_UP , UI ,
  OTHER }

  Enumeration representing the type of sprite in the game.

Protected Attributes inherited from AScenes

- GameParameters _currentGameMode

  Represents the current game mode.
- DaltonismMode _currentDaltonismMode = DaltonismMode::NORMAL

  Enum representing different modes of Daltonism (color blindness).
- Scene _currentScene = Scene::MAIN_MENU

  Represents the current scene in the application.
- Scene _previousScene = Scene::MAIN_MENU

  Represents the previous scene in the application.
- bool _displayDaltonismChoice = false

  Flag to indicate whether the Daltonism choice should be displayed.
- bool _displayGameModeChoice = false

  Flag indicating whether the game mode choice should be displayed.
- bool _displayKeyBindsChoice = false

  Flag indicating whether the key bindings choice should be displayed.
- std::string _ip

  The IP address of the server.
- int _port

  The port number of the server.
- bool _playerReady = false

6.57.1 Detailed Description

Represents a class that manages different scenes in a game.

The Scenes class provides functionality to display and manage various scenes in a game, such as the main menu, game loop, settings menu, and in-game menu. It also allows setting the game mode and daltonism mode.

6.57.2 Constructor & Destructor Documentation

6.57.2.1 Scenes()

```
Scenes::Scenes (
            std::string ip,
            int port )
```

Construct a new Scenes object.

Parameters

window	

6.57.2.2 ~Scenes()

```
Scenes::~Scenes ( ) [default]
```

Destroy the Scenes object.

6.57.3 Member Function Documentation

6.57.3.1 difficultyChoices()

```
void Scenes::difficultyChoices ( ) [virtual]
```

displays the difficulty choices, creates all the necessary entities

Implements IScenes.

6.57.3.2 difficultyChoicesCustomization()

```
void Scenes::difficultyChoicesCustomization ( )
```

6.57.3.3 gameLoop()

```
void Scenes::gameLoop ( ) [virtual]
```

displays the main game loop, creates all the necessary entities

Implements IScenes.

6.57.3.4 getRenderWindow()

```
sf::RenderWindow * Scenes::getRenderWindow ( ) [inline], [virtual]
```

Get the RenderWindow object.

Returns

sf::RenderWindow*

Implements IScenes.

6.57.3.5 HandleMessage()

```
void Scenes::HandleMessage (
            r_type::net::Message< TypeMessage > & msg,
            ComponentManager & componentManager,
            TextureManager & textureManager,
            FontManager & fontManager,
            std::shared_ptr< AudioSystem > & audioSystem )
```

6.57.3.6 HandleTransitionLevelMessage()

```
void Scenes::HandleTransitionLevelMessage (
            r_type::net::Message< TypeMessage > & msg,
            ComponentManager & componentManager,
            TextureManager & textureManager )
```

6.57.3.7 inGameMenu()

```
void Scenes::inGameMenu ( ) [virtual]
```

displays the in game menu, creates all the necessary entities

This function handles the main game loop for the Scenes class.

It contains the logic for connecting to a server, updating entities, handling user input, and rendering the game.

The game loop performs the following steps:

1. Connects to a server using the r_type::net::Client class.
2. Initializes the ComponentManager, TextureManager, and EntityManager.
3. Creates a background entity and sets its sprite component.
4. Defines lambda functions for updating player position and firing missiles.
5. Enters the main loop, which continues until the window is closed.
6. Within the loop, it checks for user input events and handles them accordingly.
7. If the server is connected, it processes incoming messages and updates entities accordingly.
8. It then updates the entities using the UpdateSystem and renders them using the RenderSystem.

Note

This code assumes the presence of the r_type::net::Client, ComponentManager, TextureManager, EntityManager, UpdateSystem, and RenderSystem classes.

See also

r_type::net::Client
ComponentManager
TextureManager
EntityManager
UpdateSystem
RenderSystem

Displays the in-game menu.

Implements IScenes.

6.57.3.8 mainMenu()

```
void Scenes::mainMenu ( ) [virtual]
```

displays the main menu, creates all the necessary entities

Displays the main menu scene.

This function creates the main menu scene, including the background, buttons, and event handling. The main menu scene allows the user to navigate to different scenes by clicking on the buttons. The buttons include "Play", "↩ Settings", and "Quit". The function continuously updates and renders the scene until the user closes the window or navigates to a different scene.

Returns

void

Implements IScenes.

6.57.3.9 render()

```
void Scenes::render ( ) [virtual]
```

display what must be displayed (main menu, game loop, settings menu, in game menu), creates all the components needed and manages them

Renders the current scene based on the value of currentScene.

The render function uses a switch statement to determine which scene to render. It calls the corresponding member function based on the value of currentScene.

Note

The currentScene variable must be set before calling this function.

Implements IScenes.

6.57.3.10 run()

```
void Scenes::run ( )
```

6.57.3.11 settingsMenu()

```
void Scenes::settingsMenu ( ) [virtual]
```

displays the settings menu, creates all the necessary entities

Displays the settings menu.

This function is responsible for displaying the settings menu in the game. It does not return any value.

Implements IScenes.

6.57.3.12 shouldQuit()

```
bool Scenes::shouldQuit ( ) [inline], [virtual]
```

check if game should stop running

Returns

true

false

Implements IScenes.

6.57.3.13 StopGameLoop()

```
void Scenes::StopGameLoop (
            std::shared_ptr< AudioSystem > & audioSystem )
```

6.57.3.14 TransitionLevel()

```
void Scenes::TransitionLevel ( )
```

6.57.4 Member Data Documentation

6.57.4.1 _networkClient

```
r_type::net::Client Scenes::_networkClient
```

6.57.4.2 _window

```
sf::RenderWindow Scenes::_window
```

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

- /home/runner/work/R-Type/R-Type/Client/Interface/Include/scenes.hpp
- /home/runner/work/R-Type/R-Type/Client/Src/scenes.cpp

6.58 ScoreComponent Struct Reference

Component that holds the score of an entity.

```
#include <score_component.hpp>
```

Public Attributes

- int score

6.58.1 Detailed Description

Component that holds the score of an entity.

The ScoreComponent is used within the ECS framework to keep track of the score associated with a particular entity.

6.58.2 Member Data Documentation

6.58.2.1 score

```
int ScoreComponent::score
```

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

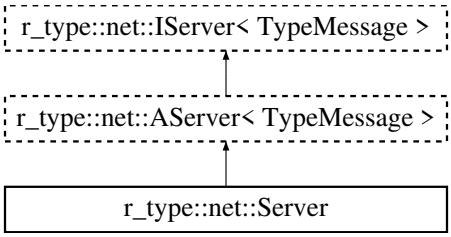
- /home/runner/work/R-Type/R-Type/ECS/Interface/Include/Components/score_component.hpp

6.59 r_type::net::Server Class Reference

A server class that handles client connections and messaging.

```
#include <server.hpp>
```

Inheritance diagram for r_type::net::Server:



Public Member Functions

- Server (uint16_t nPort)

  Constructs a new Server object with the specified port number.
- ~Server ()

  Destructor for the Server class.

Public Member Functions inherited from r_type::net::AServer< TypeMessage >

- AServer (uint16_t port)

  Constructs an AServer object with the specified port.
- ~AServer ()

  Destructor for the AServer class.
- bool Start ()

  Starts the server and begins waiting for client messages.
- void Stop ()

  Stops the server.
- void WaitForClientMessage ()

  Waits for a client message asynchronously.
- void MessageClient (std::shared_ptr< Connection< TypeMessage > > client, const Message< TypeMessage > &msg)

  Sends a message to a specific client if the client is connected.
- void MessageAllClients (const Message< TypeMessage > &msg, std::shared_ptr< Connection< TypeMessage > > pIgnoreClient=nullptr)

  Sends a message to all connected clients, optionally ignoring a specified client.
- UIEntityInformation UpdateInfoBar (int playerId)

  Updates the information bar for a given player.
- void Update (size_t nMaxMessages=-1, bool bWait=false)

  Updates the server state, processes incoming messages, and updates the game level.
- void UpdatePlayerPosition (PlayerMovement direction, uint32_t entityId) override

  Updates the position of an entity based on the message received and the client ID.
- void SavePlayerScore (uint32_t playerId)

  Saves the score of a player to a file.
- uint32_t GetClientPlayerId (uint32_t id)

Retrieves the entity ID associated with a client ID.

- uint32_t GetPlayerClientId (uint32_t id)

  Retrieves the client ID associated with a given player ID.
- uint32_t GetClientInfoBarId (uint32_t id)

  Retrieves the client info bar ID associated with a given client ID.
- void RemovePlayer (uint32_t id)

  Removes a player from the server.
- void RemoveEntity (uint32_t id)

  Removes an entity from the server.
- void RemoveInfoBar (uint32_t infoBarId)

  Removes an information bar and its associated entities.
- void RemoveBossTail (int bossId)
- EntityInformation InitiatePlayer (int clientId)

  Initializes a new player entity and assigns a random position.
- UIEntityInformation InitInfoBar (int clientId)
- EntityInformation FormatEntityInformation (uint32_t entityId)

  Formats the information of a given entity into an EntityInformation structure.
- EntityInformation InitiatePlayerMissile (int entityId)

  Initializes a missile entity associated with a player.
- EntityInformation InitiateEnemyMissile (int enemyId)
- EntityInformation InitiateWeaponForce (int entityId)
- void InitBoss (r_type::net::AServer< TypeMessage > *server)
- std::shared_ptr< Connection< TypeMessage > > getClientById (const std::deque< std::shared_ptr< Connection< TypeMessage > > &connections, uint32_t clientId)
- virtual void OnClientValidated (std::shared_ptr< Connection< TypeMessage > > client)

  Callback function that is called when a client has been successfully validated.
- ComponentManager GetComponentManager () override

  Retrieves the component manager associated with the server.
- EntityManager & GetEntityManager () override

  Retrieves the entity manager associated with the server.
- EntityFactory & GetEntityFactory () override

  Retrieves the entity factory associated with the server.
- std::chrono::system_clock::time_point GetClock () override

  Retrieves the current clock time of the server.
- void SetClock (std::chrono::system_clock::time_point clock)

  Set the Clock object.

Protected Member Functions

- bool OnClientConnect (std::shared_ptr< r_type::net::Connection< TypeMessage > > client)

  Handles the event when a client attempts to connect to the server.
- void OnClientDisconnect (std::shared_ptr< r_type::net::Connection< TypeMessage > > client, r_type::net↩::Message< TypeMessage > &msg)

  Handles the event when a client disconnects from the server.
- void OnMessage (std::shared_ptr< r_type::net::Connection< TypeMessage > > client, r_type::net::↩Message< TypeMessage > &msg)

  Handles incoming messages from a client.

Protected Member Functions inherited from r_type::net::AServer< TypeMessage >

- virtual bool OnClientConnect (std::shared_ptr< Connection< TypeMessage > > client)

  on client connect event
- virtual void OnClientDisconnect (std::shared_ptr< Connection< TypeMessage > > client)

  on client disconnect event
- virtual void OnMessage (std::shared_ptr< Connection< TypeMessage > > client, Message< TypeMessage > &msg)

  on message event

Additional Inherited Members

Public Attributes inherited from r_type::net::AServer< TypeMessage >

- ThreadSafeQueue< OwnedMessage< TypeMessage > > _qMessagesIn

  Thread-safe queue to store incoming messages.
- std::deque< std::shared_ptr< Connection< TypeMessage > > > _deqConnections

  A deque that holds shared pointers to Connection objects.
- asio::io_context _asioContext

  The io_context object provides I/O services, such as sockets, that the server will use.
- std::thread _threadContext

  Thread object for managing the server's context operations.
- asio::ip::udp::socket _asioSocket

  A socket for sending and receiving UDP datagrams.
- asio::ip::udp::endpoint _clientEndpoint

  Represents the endpoint of a client in a UDP connection.
- std::array< uint8_t, 1024 > _tempBuffer

  Temporary buffer used for storing data.
- uint32_t _nIDCounter

  Counter for generating unique network IDs.
- ComponentManager _componentManager

  Manages and maintains the lifecycle of various components within the server.
- EntityManager _entityManager

  Manages the lifecycle and operations of entities within the server.
- EntityFactory _entityFactory

  An instance of EntityFactory used to create and manage game entities.
- bool _endOfLevel
- bool _bossActive
- bool _bossDefeated
- bool _watingPlayersReady
- std::unordered_map< uint32_t, uint32_t > _clientPlayerID

  A container that maps client IDs to player IDs.
- std::unordered_map< uint32_t, uint32_t > _clientInfoBarID
- int _nbrOfPlayers

  Number of players currently connected to the server.
- std::chrono::system_clock::time_point _clock

  Stores the current time point from the system clock.
- bool _playerConnected
- EntityInformation _background

  Holds information about the background entity.
- int _port
- r_type::Level< TypeMessage > _level
- int _playerReady

6.59.1 Detailed Description

A server class that handles client connections and messaging.

This class inherits from r_type::net::AServer<TypeMessage> and provides implementations for handling client connections, disconnections, and message reception.

Template Parameters

TypeMessage	The type of message that the server will handle.

6.59.2 Constructor & Destructor Documentation

6.59.2.1 Server()

```
r_type::net::Server::Server (
            uint16_t nPort )  [inline]
```

Constructs a new Server object with the specified port number.

This constructor initializes the Server object by calling the constructors of the base classes r_type::net::IServer<↩ TypeMessage> and r_type::net::AServer<TypeMessage> with the provided port number.

Parameters

nPort	The port number on which the server will listen for incoming connections.

6.59.2.2 ~Server()

```
r_type::net::Server::~Server ( )  [inline]
```

Destructor for the Server class.

This destructor is responsible for cleaning up any resources allocated by the Server class. Currently, it does not perform any specific actions.

6.59.3 Member Function Documentation

6.59.3.1 OnClientConnect()

```
bool r_type::net::Server::OnClientConnect (
            std::shared_ptr< r_type::net::Connection< TypeMessage > > client )  [protected]
```

Handles the event when a client attempts to connect to the server.

Parameters

client	A shared pointer to the client's connection object.

Returns

true if the client is allowed to connect, false otherwise.

This function checks if the maximum number of players (4) has been reached. If so, it sends a denial message to the client and returns false. Otherwise, it sends an acceptance message to the client, increments the number of players, sets the client's status to INITIALISATION, assigns the last message sent to the client, and initializes the client's entities.

Parameters

client	A shared pointer to the client connection attempting to connect.

Returns

true if the client is accepted, false if the client is denied.

6.59.3.2 OnClientDisconnect()

```
void r_type::net::Server::OnClientDisconnect (
            std::shared_ptr< r_type::net::Connection< TypeMessage > > client,
            r_type::net::Message< TypeMessage > & msg )  [protected]
```

Handles the event when a client disconnects from the server.

Handles the disconnection of a client from the server.

Parameters

client	A shared pointer to the client's connection object.
msg	A reference to the message object containing information about the disconnection.

This function is called when a client disconnects from the server. It performs several tasks including removing the client, saving the player's score, removing associated entities, and notifying all other clients about the disconnection.

Parameters

client	A shared pointer to the connection object representing the client.
msg	A reference to the message object containing information about the disconnection.

6.59.3.3 OnMessage()

```
void r_type::net::Server::OnMessage (
            std::shared_ptr< r_type::net::Connection< TypeMessage > > client,
            r_type::net::Message< TypeMessage > & msg )  [protected]
```

Handles incoming messages from a client.

Handles the reception of a message from a client.

This function is called whenever a message is received from a client. It processes the message and performs the necessary actions based on the message content.

Parameters

client	A shared pointer to the client connection that sent the message.
msg	The message received from the client.

This function is called when a message is received from a client. It processes the message based on the client's status and the message's ID. The function performs different actions based on the message ID, such as sending a response message, updating player positions, creating entities, or destroying entities.

Parameters

client	A shared pointer to the connection object representing the client.
msg	A reference to the message object containing information sent by the client.

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

- /home/runner/work/R-Type/R-Type/Server/Interface/Include/Net/server.hpp
- /home/runner/work/R-Type/R-Type/Server/Src/server.cpp

6.60 ShaderComponent Struct Reference

A component that holds a shader.

```
#include <shader_component.hpp>
```

Public Member Functions

- ShaderComponent (std::string path)

  Constructs a ShaderComponent and loads a shader from a file.

Public Attributes

- std::shared_ptr< sf::Shader > shader

  The shader object.

6.60.1 Detailed Description

A component that holds a shader.

This component is used to manage a shader in the ECS (Entity Component System). It loads a shader from a file and stores it in a shared pointer.

6.60.2 Constructor & Destructor Documentation

6.60.2.1 ShaderComponent()

```
ShaderComponent::ShaderComponent (
            std::string path )  [inline]
```

Constructs a ShaderComponent and loads a shader from a file.

Parameters

path	The file path to the shader.

This constructor creates a new sf::Shader object and attempts to load a shader from the specified file path. If the shader fails to load, an error message is printed to the standard error stream.

6.60.3 Member Data Documentation

6.60.3.1 shader

```
std::shared_ptr<sf::Shader> ShaderComponent::shader
```

The shader object.

This is a shared pointer to an sf::Shader object.

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

- /home/runner/work/R-Type/R-Type/ECS/Interface/Include/Components/shader_component.hpp

6.61 ShootComponent Struct Reference

Component that handles shooting mechanics for an entity.

```
#include <shoot_component.hpp>
```

Public Member Functions

- ShootComponent (std::chrono::milliseconds cooldown)

  Constructs a ShootComponent with a specified cooldown time.

Public Attributes

- std::chrono::system_clock::time_point nextShootTime

  The time point when the entity is next allowed to shoot.
- std::chrono::milliseconds cooldownTime

  The cooldown duration between consecutive shots.
- bool canShoot

  A flag indicating whether the entity is currently allowed to shoot.

6.61.1 Detailed Description

Component that handles shooting mechanics for an entity.

This component keeps track of the next allowed shooting time, the cooldown period between shots, and whether the entity can shoot.

6.61.2 Constructor & Destructor Documentation

6.61.2.1 ShootComponent()

```
ShootComponent::ShootComponent (
            std::chrono::milliseconds cooldown )  [inline]
```

Constructs a ShootComponent with a specified cooldown time.

Initializes the nextShootTime to the current time and sets the cooldownTime to the provided value.

Parameters

cooldown	The cooldown duration between consecutive shots.

6.61.3 Member Data Documentation

6.61.3.1 canShoot

`ShootComponent::canShoot`

A flag indicating whether the entity is currently allowed to shoot.

6.61.3.2 cooldownTime

`ShootComponent::cooldownTime`

The cooldown duration between consecutive shots.

6.61.3.3 nextShootTime

`ShootComponent::nextShootTime`

The time point when the entity is next allowed to shoot.

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

- /home/runner/work/R-Type/R-Type/ECS/Interface/Include/Components/shoot_component.hpp

6.62 SpriteComponent Struct Reference

A component that represents a sprite in the ECS (Entity Component System).

```
#include <sprite_component.hpp>
```

Public Member Functions

- SpriteComponent (sf::Texture &texture, const float posX, float posY, const sf::Vector2f &scale, AScenes::SpriteType typeNb, sf::IntRect rect=sf::IntRect(0, 0, 0, 0))

  Constructs a SpriteComponent with the given parameters.

Public Attributes

- sf::Sprite sprite

  The SFML sprite object.
- AScenes::SpriteType type

  The type of the sprite, defined by the AScenes namespace.
- int hitboxX

  The width of the sprite's hitbox.
- int hitboxY

  The height of the sprite's hitbox.

6.62.1 Detailed Description

A component that represents a sprite in the ECS (Entity Component System).

This component holds a sprite, its type, and hitbox dimensions. It provides functionality to initialize the sprite with a texture, position, scale, and optional texture rectangle.

6.62.2 Constructor & Destructor Documentation

6.62.2.1 SpriteComponent()

```
SpriteComponent::SpriteComponent (
            sf::Texture & texture,
            const float posX,
            float posY,
            const sf::Vector2f & scale,
            AScenes::SpriteType typeNb,
            sf::IntRect rect = sf::IntRect(0, 0, 0, 0) )  [inline]
```

Constructs a SpriteComponent with the given parameters.

Parameters

texture	The texture to be used for the sprite.
posX	The X position of the sprite.
posY	The Y position of the sprite.
scale	The scale of the sprite.
typeNb	The type of the sprite.
rect	The texture rectangle to be used for the sprite (default is an empty rectangle).

6.62.3 Member Data Documentation

6.62.3.1 hitboxX

```
int SpriteComponent::hitboxX
```

The width of the sprite's hitbox.

6.62.3.2 hitboxY

```
int SpriteComponent::hitboxY
```

The height of the sprite's hitbox.

6.62.3.3 sprite

```
sf::Sprite SpriteComponent::sprite
```

The SFML sprite object.

6.62.3.4 type

```
AScenes::SpriteType SpriteComponent::type
```

The type of the sprite, defined by the AScenes namespace.

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

- /home/runner/work/R-Type/R-Type/ECS/Interface/Include/Components/sprite_component.hpp

6.63 SpriteDataComponent Struct Reference

Component that holds data related to a sprite.

```
#include <sprite_data_component.hpp>
```

Public Attributes

- SpritePath spritePath

  Path to the sprite resource.
- vf2d scale

  Scale factor for the sprite.
- AScenes::SpriteType type

  Type of the sprite as defined in AScenes::SpriteType.

6.63.1 Detailed Description

Component that holds data related to a sprite.

This component contains information about the sprite's path, scale, and type.

6.63.2 Member Data Documentation

6.63.2.1 scale

`SpriteDataComponent::scale`

Scale factor for the sprite.

6.63.2.2 spritePath

`SpriteDataComponent::spritePath`

Path to the sprite resource.

6.63.2.3 type

`SpriteDataComponent::type`

Type of the sprite as defined in AScenes::SpriteType.

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

- /home/runner/work/R-Type/R-Type/ECS/Interface/Include/Components/sprite_data_component.hpp

6.64 TailComponent Struct Reference

```
#include <tail_component.hpp>
```

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

- /home/runner/work/R-Type/R-Type/ECS/Interface/Include/Components/tail_component.hpp

6.65 TextComponent Struct Reference

A component that encapsulates an SFML text object.

```
#include <text_component.hpp>
```

Public Member Functions

- TextComponent (sf::Font &font, const std::string &string, float posX, float posY, int size=30)

  Constructs a TextComponent with the specified parameters.

Public Attributes

- sf::Text text

  The SFML text object that this component encapsulates.

6.65.1 Detailed Description

A component that encapsulates an SFML text object.

This component is used to manage and render text in an SFML application.

6.65.2 Constructor & Destructor Documentation

6.65.2.1 TextComponent()

```
TextComponent::TextComponent (
            sf::Font & font,
            const std::string & string,
            float posX,
            float posY,
            int size = 30 )  [inline]
```

Constructs a TextComponent with the specified parameters.

Parameters

font	The font to be used for the text.
string	The string to be displayed.
posX	The x-coordinate of the text's position.
posY	The y-coordinate of the text's position.
size	The character size of the text. Default is 30.

6.65.3 Member Data Documentation

6.65.3.1 text

```
sf::Text TextComponent::text
```

The SFML text object that this component encapsulates.

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

- /home/runner/work/R-Type/R-Type/ECS/Interface/Include/Components/text_component.hpp

6.66 TextDataComponent Struct Reference

Component that holds text-related data for an entity.

```
#include <text_data_component.hpp>
```

Public Attributes

- FontPath fontPath

  Path to the font file used for rendering the text.
- uint32_t charSize = 0

  Size of the characters to be rendered.
- uint32_t categoryIds [5] = {0}

  Array of category IDs associated with the text.
- GameText categoryTexts [5]

  Array of GameText objects representing the text for each category.
- uint32_t categorySize = 0

  Number of categories available.

6.66.1 Detailed Description

Component that holds text-related data for an entity.

This component is used to store information about text that can be rendered in the game, including font path, character size, category IDs, and category texts.

6.66.2 Member Data Documentation

6.66.2.1 categoryIds

```
TextDataComponent::categoryIds = {0}
```

Array of category IDs associated with the text.

6.66.2.2 categorySize

```
TextDataComponent::categorySize = 0
```

Number of categories available.

6.66.2.3 categoryTexts

```
TextDataComponent::categoryTexts
```

Array of GameText objects representing the text for each category.

6.66.2.4 charSize

```
TextDataComponent::charSize = 0
```

Size of the characters to be rendered.

6.66.2.5 fontPath

```
TextDataComponent::fontPath
```

Path to the font file used for rendering the text.

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

- /home/runner/work/R-Type/R-Type/ECS/Interface/Include/Components/text_data_component.hpp

6.67 TextureManager Class Reference

```
#include <texture_manager.hpp>
```

Public Member Functions

- sf::Texture & getTexture (const std::string &filePath)

  Retrieves a texture from the texture manager.
- void releaseTexture (const std::string &filePath)

  Releases the texture associated with the given file path.

Private Attributes

- std::unordered_map< std::string, sf::Texture > textures

  A container for storing textures with string keys.

6.67.1 Member Function Documentation

6.67.1.1 getTexture()

```
sf::Texture & TextureManager::getTexture (
            const std::string & filePath )  [inline]
```

Retrieves a texture from the texture manager.

This function attempts to find the texture associated with the given file path in the texture manager. If the texture is found, it is returned. Otherwise, a new texture is loaded from the file path and added to the texture manager before being returned.

Exceptions

failedToLoadTexture	If the texture fails to load from the file path.

Parameters

filePath	The file path of the texture to retrieve.

Returns

sf::Texture& A reference to the retrieved texture.

6.67.1.2 releaseTexture()

```
void TextureManager::releaseTexture (
            const std::string & filePath )  [inline]
```

Releases the texture associated with the given file path.

This function removes the texture from the internal texture storage, effectively releasing any resources associated with it.

Parameters

filePath	The file path of the texture to be released.

6.67.2 Member Data Documentation

6.67.2.1 textures

```
std::unordered_map<std::string, sf::Texture> TextureManager::textures  [private]
```

A container for storing textures with string keys.

This unordered map allows you to associate a string key with an sf::Texture object. It provides fast access to textures based on their keys.

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

- /home/runner/work/R-Type/R-Type/ECS/Interface/Include/texture_manager.hpp

6.68 UIEntityInformation Struct Reference

Represents the information of a UI entity in the game.

```
#include <entity_struct.hpp>
```

Public Attributes

- uint32_t uniqueID = 0

  Unique identifier for the UI entity.
- uint32_t lives = 0

  Number of lives the UI entity has.
- uint32_t score = 0

  Score associated with the UI entity.
- SpriteDataComponent spriteData

  Data related to the sprite of the UI entity.
- TextDataComponent textData

  Data related to the text of the UI entity.

6.68.1 Detailed Description

Represents the information of a UI entity in the game.

This structure holds various attributes related to a UI entity, including its unique identifier, lives, score, and associated sprite and text data components.

6.68.2 Member Data Documentation

6.68.2.1 lives

```
uint32_t UIEntityInformation::lives = 0
```

Number of lives the UI entity has.

6.68.2.2 score

```
uint32_t UIEntityInformation::score = 0
```

Score associated with the UI entity.

6.68.2.3 spriteData

```
SpriteDataComponent UIEntityInformation::spriteData
```

Data related to the sprite of the UI entity.

6.68.2.4 textData

```
TextDataComponent UIEntityInformation::textData
```

Data related to the text of the UI entity.

6.68.2.5 uniqueID

```
uint32_t UIEntityInformation::uniqueID = 0
```

Unique identifier for the UI entity.

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

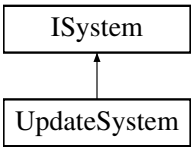
- /home/runner/work/R-Type/R-Type/ECS/Interface/Include/entity_struct.hpp

6.69 UpdateSystem Class Reference

A system responsible for updating sprite positions in the game.

```
#include <update_system.hpp>
```

Inheritance diagram for UpdateSystem:



Public Member Functions

- UpdateSystem (sf::RenderWindow &window, ComponentManager &componentManager, EntityManager &entityManager)

  Manages the update logic for entities within the ECS framework.
- void updateSpritePositions (ComponentManager &componentManager, EntityManager &entityManager)

  Updates the positions of all sprite components in the game.

Public Member Functions inherited from ISystem

- ISystem ()=default
- virtual ~ISystem ()=default

Private Attributes

- sf::RenderWindow & _window

  Reference to the SFML RenderWindow used for rendering.
- ComponentManager & _componentManager

  Reference to the ComponentManager instance.
- EntityManager & _entityManager

  Reference to the EntityManager instance.

6.69.1 Detailed Description

A system responsible for updating sprite positions in the game.

The UpdateSystem class inherits from the ISystem interface and is responsible for updating the positions of sprites in the game. It interacts with the ComponentManager and EntityManager to manage and update the components and entities.

Parameters

window	Reference to the SFML RenderWindow object.
componentManager	Reference to the ComponentManager object.
entityManager	Reference to the EntityManager object.

6.69.2 Constructor & Destructor Documentation

6.69.2.1 UpdateSystem()

```
UpdateSystem::UpdateSystem (
            sf::RenderWindow & window,
            ComponentManager & componentManager,
            EntityManager & entityManager )  [inline]
```

Manages the update logic for entities within the ECS framework.

Parameters

window	Reference to the SFML RenderWindow used for rendering.
componentManager	Reference to the ComponentManager that handles components.
entityManager	Reference to the EntityManager that handles entities.

6.69.3 Member Function Documentation

6.69.3.1 updateSpritePositions()

```
void UpdateSystem::updateSpritePositions (
            ComponentManager & componentManager,
            EntityManager & entityManager )
```

Updates the positions of all sprite components in the game.

This function iterates through all entities that have sprite components and updates their positions based on their current velocities and other relevant factors.

Parameters

componentManager	Reference to the ComponentManager that manages all components.
entityManager	Reference to the EntityManager that manages all entities.

6.69.4 Member Data Documentation

6.69.4.1 _componentManager

```
ComponentManager& UpdateSystem::_componentManager  [private]
```

Reference to the ComponentManager instance.

This member is used to manage and access various components within the ECS (Entity Component System).

6.69.4.2 _entityManager

```
EntityManager& UpdateSystem::_entityManager [private]
```

Reference to the EntityManager instance.

This member variable holds a reference to the EntityManager, which is responsible for managing all entities within the ECS (Entity Component System). It provides functionalities to create, destroy, and query entities.

6.69.4.3 _window

```
sf::RenderWindow& UpdateSystem::_window [private]
```

Reference to the SFML RenderWindow used for rendering.

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

- /home/runner/work/R-Type/R-Type/ECS/Interface/Include/Systems/update_system.hpp
- /home/runner/work/R-Type/R-Type/ECS/Src/Systems/update_system.cpp

6.70 UpdateTextComponent Struct Reference

```
#include <update_text_component.hpp>
```

Public Member Functions

- UpdateTextComponent (std::function< std::string(GameParameters)> updateTextFunction)

Public Attributes

- std::function< std::string(GameParameters)> updateText

6.70.1 Constructor & Destructor Documentation

6.70.1.1 UpdateTextComponent()

```
UpdateTextComponent::UpdateTextComponent (
            std::function< std::string(GameParameters)> updateTextFunction ) [inline]
```

6.70.2 Member Data Documentation

6.70.2.1 updateText

```
std::function<std::string(GameParameters)> UpdateTextComponent::updateText
```

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

- /home/runner/work/R-Type/R-Type/ECS/Interface/Include/Components/update_text_component.hpp

6.71 VelocityComponent Struct Reference

Represents the velocity of an entity in 2D space.

```
#include <velocity_component.hpp>
```

Public Attributes

- float x

  The velocity along the x-axis.
- float y

  The velocity along the y-axis.

6.71.1 Detailed Description

Represents the velocity of an entity in 2D space.

This component stores the velocity of an entity along the x and y axes. It can be used to update the position of the entity based on its speed and direction.

6.71.2 Member Data Documentation

6.71.2.1 x

```
VelocityComponent::x
```

The velocity along the x-axis.

6.71.2.2 y

```
VelocityComponent::y
```

The velocity along the y-axis.

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

- /home/runner/work/R-Type/R-Type/ECS/Interface/Include/Components/velocity_component.hpp

6.72 vf2d Struct Reference

Represents a 2D vector with x and y coordinates.

```
#include <macros.hpp>
```

Public Attributes

- float x = 0
- float y = 0

6.72.1 Detailed Description

Represents a 2D vector with x and y coordinates.

6.72.2 Member Data Documentation

6.72.2.1 x

```
float vf2d::x = 0
```

6.72.2.2 y

```
float vf2d::y = 0
```

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

- /home/runner/work/R-Type/R-Type/ECS/Interface/Include/macros.hpp

6.73 WallComponent Struct Reference

```
#include <wall_component.hpp>
```

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

- /home/runner/work/R-Type/R-Type/ECS/Interface/Include/Components/wall_component.hpp

Chapter 7

File Documentation

7.1 /home/runner/work/R-Type/R-Type/Client/Interface/↩ Include/mainmenu.hpp File Reference

```
#include <SFML/Graphics.hpp>
#include <r_type_client.hpp>
```

Functions

- int MainMenu (sf::RenderWindow ∗window, Rtype ∗rtype)

7.1.1 Function Documentation

7.1.1.1 MainMenu()

```
int MainMenu (
            sf::RenderWindow * window,
            Rtype * rtype )
```

7.2 mainmenu.hpp

Go to the documentation of this file.

```
/*
** EPITECH PROJECT, 2024
** R-Type
** File description:
** mainmenu
*/

#pragma once

#include <SFML/Graphics.hpp>
#include <r_type_client.hpp>

int MainMenu(sf::RenderWindow *window, Rtype *rtype);
```

7.3 /home/runner/work/R-Type/R-Type/Client/Interface/Include/Net/a_client.hpp File Reference

```
#include <Components/component_manager.hpp>
#include <Components/components.hpp>
#include <Net/i_client.hpp>
#include <SFML/Graphics.hpp>
#include <entity_struct.hpp>
#include <font_manager.hpp>
#include <texture_manager.hpp>
#include <unordered_map>
```

Classes

- class r_type::net::AClient< T >

Namespaces

- namespace r_type
- namespace r_type::net

7.4 a_client.hpp

Go to the documentation of this file.

```
/*
** EPITECH PROJECT, 2024
** R-Type
** File description:
** netClient
*/

#pragma once

#include <Components/component_manager.hpp>
#include <Components/components.hpp>
#include <Net/i_client.hpp>
#include <SFML/Graphics.hpp>
#include <entity_struct.hpp>
#include <font_manager.hpp>
#include <texture_manager.hpp>
#include <unordered_map>

namespace r_type {
namespace net {
template <typename T> class AClient : virtual public IClient<T> {
  public:
    AClient() { m_connection = nullptr; }

    virtual ~AClient() { Disconnect(); }

  public:
    bool Connect(const std::string &host, const uint16_t port)
    {
        try {
            asio::ip::udp::endpoint remote_endpoint =
                asio::ip::udp::endpoint(asio::ip::make_address(host), port);
            // std::cout « "Remote endpoint: " « remote_endpoint « std::endl;

            asio::ip::udp::socket socket(
                m_context, asio::ip::udp::endpoint(asio::ip::udp::v4(), 0));
            m_connection = std::make_unique<Connection<T>(Connection<T>::owner::client, m_context,
                std::move(socket), std::move(remote_endpoint), m_qMessagesIn);
            m_connection->ConnectToServer();

```

```
            // std::cout « "Connection: " « *(m_connection.get()) « std::endl;

            thrContext = std::thread([this]() { m_context.run(); });
        } catch (std::exception &e) {
            std::cerr « "Client Exception: " « e.what() « std::endl;
            return false;
        }
        return true;
    }

    void Disconnect()
    {
        if (IsConnected()) {
            m_connection->Disconnect();
        }

        m_context.stop();
        if (thrContext.joinable())
            thrContext.join();

        m_connection.release();
    }

    bool IsConnected()
    {
        if (m_connection)
            return m_connection->IsConnected();
        else
            return false;
    }

  public:
    void Send(const Message<T> &msg)
    {
        if (IsConnected())
            m_connection->Send(msg);
    }

    ThreadSafeQueue<OwnedMessage<T» &Incoming() { return m_qMessagesIn; }

    const std::unique_ptr<Connection<T» &getConnection() { return m_connection; }

    void setPlayerId(uint32_t id) { playerId = id; }
    uint32_t getPlayerId() { return playerId; }

    void setWindowSize(sf::Vector2u size) { windowSize = size; }
    sf::Vector2u getWindowSize() { return windowSize; }

  protected:
    asio::io_context m_context;
    std::thread thrContext;
    std::unique_ptr<Connection<T» m_connection;

  private:
    ThreadSafeQueue<OwnedMessage<T» m_qMessagesIn;
    uint32_t playerId = 0;
    sf::Vector2u windowSize;
};
} // namespace net
} // namespace r_type
```

7.5 /home/runner/work/R-Type/R-Type/Client/Interface/Include/↩ Net/client.hpp File Reference

```
#include <Net/a_client.hpp>
#include <SFML/Graphics.hpp>
#include <fstream>
#include <iostream>
```

Classes

- class r_type::net::Client

Namespaces

- namespace r_type
- namespace r_type::net

7.6 client.hpp

Go to the documentation of this file.

```
/*
** EPITECH PROJECT, 2024
** R-Type
** File description:
** simpleClient
*/

#pragma once

#include <Net/a_client.hpp>
#include <SFML/Graphics.hpp>
#include <fstream>
#include <iostream>

namespace r_type {
namespace net {
class Client : virtual public r_type::net::AClient<TypeMessage> {
  public:
    void PingServer()
    {
        r_type::net::Message<TypeMessage> msg;
        msg.header.id = TypeMessage::ServerPing;

        std::chrono::system_clock::time_point timeNow = std::chrono::system_clock::now();

        msg « timeNow;
        Send(msg);
    }

    void MessageAll()
    {
        r_type::net::Message<TypeMessage> msg;
        msg.header.id = TypeMessage::MessageAll;
        Send(msg);
    }

    sf::Vector2u initInfoBar(UIEntityInformation entity, ComponentManager &componentManager,
        TextureManager &textureManager, FontManager &fontManager, sf::Vector2u windowSize)
    {
        float windowWidth = static_cast<float>(windowSize.x);
        float windowHeight = static_cast<float>(windowSize.y);

        sf::Texture &texture =
            textureManager.getTexture(SpriteFactory(entity.spriteData.spritePath));
        float desiredWidth = windowWidth;
        float desiredHeight = windowHeight * 0.10f;
        sf::Vector2f scale(
            desiredWidth / texture.getSize().x, desiredHeight / texture.getSize().y);
        SpriteComponent spriteComponent(texture, 0, 0, scale, entity.spriteData.type);
        componentManager.addComponent<SpriteComponent>(entity.uniqueID, spriteComponent);

        if (auto spriteEntity = componentManager.getComponent<SpriteComponent>(entity.uniqueID)) {
            spriteEntity.value()->sprite.setPosition(windowWidth / 2, windowHeight * 0.95f);
            sf::Font &font = fontManager.getFont(FontFactory(entity.textData.fontPath));

            sf::FloatRect spriteBounds = spriteEntity.value()->sprite.getLocalBounds();
            float barWidth = spriteBounds.width * scale.x;
            float barHeight = spriteBounds.height * scale.y;
            float barPosX = spriteEntity.value()->sprite.getPosition().x;
            float barPosY = spriteEntity.value()->sprite.getPosition().y;

            for (size_t i = 0; i < entity.textData.categorySize; i++) {
                std::string displayText = GameTextFactory(entity.textData.categoryTexts[i]);
                TextComponent textComponent(font, displayText, 0, 0, entity.textData.charSize);
                componentManager.addComponent<TextComponent>(
                    entity.textData.categoryIds[i], textComponent);

                if (auto textComponent = componentManager.getComponent<TextComponent>(
                        entity.textData.categoryIds[i])) {
                    sf::FloatRect textBounds = textComponent.value()->text.getLocalBounds();

                    float posX = barPosX;
```

```
                    if (entity.textData.categoryTexts[i] == GameText::Lives) {
                        posX -= (barWidth / 4);
                        displayText += std::to_string(entity.lives);
                    } else if (entity.textData.categoryTexts[i] == GameText::Score) {
                        posX += (barWidth / 4);
                        displayText += std::to_string(entity.score);
                    }
                    float posY =
                        barPosY - (barHeight / 2) + (barHeight / 2) - (textBounds.height / 2);

                    textComponent.value()->text.setPosition(posX, posY);
                    textComponent.value()->text.setString(displayText);
                }
            }
            sf::Vector2u newWindowSize = {
                windowSize.x, static_cast<unsigned int>(windowHeight - desiredHeight)};
            return newWindowSize;
        }
        return windowSize;
    }

    void updateInfoBar(UIEntityInformation entity, ComponentManager &componentManager)
    {
        for (size_t i = 0; i < entity.textData.categorySize; i++) {
            if (auto textComponent =
                    componentManager.getComponent<TextComponent>(entity.textData.categoryIds[i])) {
                std::string displayText = GameTextFactory(entity.textData.categoryTexts[i]);
                if (entity.textData.categoryTexts[i] == GameText::Lives) {
                    displayText += std::to_string(entity.lives);
                } else if (entity.textData.categoryTexts[i] == GameText::Score) {
                    displayText += std::to_string(entity.score);
                }
                textComponent.value()->text.setString(displayText);
            }
        }
    }

    void addEntity(EntityInformation entity, ComponentManager &componentManager,
        TextureManager &textureManager, sf::Vector2u windowSize)
    {
        if (entity.spriteData.type == AScenes::SpriteType::UI) {
            return;
        }
        float posX = windowSize.x * (entity.vPos.x / 100.0f);
        float posY = windowSize.y * (entity.vPos.y / 100.0f);
        float scaleX = (entity.ratio.x * windowSize.x) / entity.animationComponent.dimension.x;
        float scaleY = (entity.ratio.y * windowSize.y) / entity.animationComponent.dimension.y;
        sf::Texture &texture =
            textureManager.getTexture(SpriteFactory(entity.spriteData.spritePath));
        sf::Vector2f scale(scaleX, scaleY);
        sf::IntRect rect(entity.animationComponent.offset.x, entity.animationComponent.offset.y,
            entity.animationComponent.dimension.x, entity.animationComponent.dimension.y);
        SpriteComponent sprite(texture, posX, posY, scale, entity.spriteData.type, rect);
        componentManager.addComponent<SpriteComponent>(entity.uniqueID, sprite);
    }

    void removeEntity(int entityId, ComponentManager &componentManager)
    {
        componentManager.removeEntityFromComponent<SpriteComponent>(entityId);
    }

    void moveEntity(
        uint32_t id, vf2d newPos, ComponentManager &componentManager, sf::Vector2u windowSize)
    {
        auto spriteEntity = componentManager.getComponent<SpriteComponent>(id);
        if (spriteEntity) {
            float posX = windowSize.x * (newPos.x / 100.0f);
            float posY = windowSize.y * (newPos.y / 100.0f);
            spriteEntity.value()->sprite.setPosition(posX, posY);
        } else {
            std::cerr « "Entity not found, id: " « id « std::endl;
        }
    }

    void animateEntity(int entityId, AnimationComponent rect, ComponentManager &componentManager)
    {
        if (auto spritesOpt = componentManager.getComponentMap<SpriteComponent>()) {
            auto &sprites = **spritesOpt;
            auto entitySpriteIt = sprites.find(entityId);
            if (entitySpriteIt != sprites.end()) {
                auto &spriteComponent = entitySpriteIt->second;
                if (auto entitySprite = std::any_cast<SpriteComponent>(&spriteComponent)) {
                    sf::IntRect newRect(
                        rect.offset.x, rect.offset.y, rect.dimension.x, rect.dimension.y);
                    entitySprite->sprite.setTextureRect(newRect);
                }
            }
```

```
        }
    }

    void displayEndOfGame(ComponentManager &componentManager, TextureManager &textureManager,
        FontManager &fontManager, sf::Vector2u windowSize)
    {
        float yPos = 50;
        float xPos = windowSize.x / 2;
        sf::Font &font = fontManager.getFont(FontFactory(FontPath::MAIN));
        const std::string winText = "You Win!";
        TextComponent textComponent(font, winText, xPos, yPos, 80);
        componentManager.addComponent<TextComponent>(3, textComponent);

        std::vector<std::string> scores;
        std::ifstream file("GameScores/scores.txt");
        if (file.is_open()) {
            std::string line;
            while (std::getline(file, line)) {
                scores.push_back(line);
            }
            file.close();
        } else {
            throw failedToOpenFile();
        }
        int id = 4;
        for (const auto &score : scores) {
            yPos += 80;
            TextComponent textComponent(font, score, xPos, yPos, 50);
            componentManager.addComponent<TextComponent>(id, textComponent);
            id++;
        }
    }
};
} // namespace net
} // namespace r_type
```

7.7 /home/runner/work/R-Type/R-Type/Client/Interface/Include/Net/i_client.hpp File Reference

```
#include <Net/common.hpp>
#include <Net/connection.hpp>
#include <Net/thread_safe_queue.hpp>
```

Classes

- class r_type::net::IClient< T >

Namespaces

- namespace r_type
- namespace r_type::net

7.8 i_client.hpp

Go to the documentation of this file.

```
/*
** EPITECH PROJECT, 2024
** R-Type
** File description:
** netClient
*/

#pragma once
```

```

#include <Net/common.hpp>
#include <Net/connection.hpp>
#include <Net/thread_safe_queue.hpp>

namespace r_type {
namespace net {
template <typename T> class IClient {
  public:
    IClient() {}

    virtual ~IClient() {}

  public:
    virtual bool Connect(const std::string &host, const uint16_t port) = 0;

    virtual void Disconnect() = 0;

    virtual bool IsConnected() = 0;

  public:
    virtual void Send(const Message<T> &msg) = 0;

    virtual ThreadSafeQueue<OwnedMessage<T>> &Incoming() = 0;
};
} // namespace net
} // namespace r_type
```

7.9 /home/runner/work/R-Type/R-Type/Client/Interface/↩ Include/scenes.hpp File Reference

```
#include <Entities/entity.hpp>
#include <Net/client.hpp>
#include <SFML/Graphics.hpp>
#include <Systems/systems.hpp>
#include <a_scenes.hpp>
#include <memory>
#include <vector>
```

Classes

- class Scenes

  Represents a class that manages different scenes in a game.

Functions

- std::string keyToString (sf::Keyboard::Key key)

7.9.1 Function Documentation

7.9.1.1 keyToString()

```
std::string keyToString (
            sf::Keyboard::Key key )
```

7.10 scenes.hpp

Go to the documentation of this file.

```
/*
** EPITECH PROJECT, 2024
** R-Type
** File description:
** scenes
*/

#pragma once

#include <Entities/entity.hpp>
#include <Net/client.hpp>
#include <SFML/Graphics.hpp>
#include <Systems/systems.hpp>
#include <a_scenes.hpp>
#include <memory>
#include <vector>

std::string keyToString(sf::Keyboard::Key key);
class Scenes : virtual public AScenes {

  public:
    Scenes(std::string ip, int port);

    ~Scenes() = default;

    void mainMenu();

    void gameLoop();

    void HandleMessage(r_type::net::Message<TypeMessage> &msg, ComponentManager &componentManager,
        TextureManager &textureManager, FontManager &fontManager,
        std::shared_ptr<AudioSystem> &audioSystem);

    void StopGameLoop(std::shared_ptr<AudioSystem> &audioSystem);

    void settingsMenu();

    void inGameMenu();

    void difficultyChoices();

    void difficultyChoicesCustomization();

    void render();

    bool shouldQuit() { return _currentScene == Scene::EXIT; }

    sf::RenderWindow *getRenderWindow() { return &_window; }

    void TransitionLevel();

    void HandleTransitionLevelMessage(r_type::net::Message<TypeMessage> &msg,
        ComponentManager &componentManager, TextureManager &textureManager);

    void run();

    sf::RenderWindow _window;

    r_type::net::Client _networkClient;
};
```

7.11 /home/runner/work/R-Type/R-Type/Client/Src/keyToString.cpp File Reference

```
#include <SFML/Window/Keyboard.hpp>
#include <iostream>
```

Functions

- std::string keyToString (sf::Keyboard::Key key)

7.11.1 Function Documentation

7.11.1.1 keyToString()

```
std::string keyToString (
            sf::Keyboard::Key key )
```

7.12 /home/runner/work/R-Type/R-Type/Client/Src/main.cpp File Reference

```
#include <iostream>
#include <macro.hpp>
#include <scenes.hpp>
#include <sstream>
```

Functions

- static bool isValidIPv4 (const std::string &ip)
- static bool isValidPort (const std::string &portStr)
- int main (int const argc, char const ∗const ∗argv)

  The entry point of the program.

7.12.1 Function Documentation

7.12.1.1 isValidIPv4()

```
static bool isValidIPv4 (
            const std::string & ip )  [static]
```

7.12.1.2 isValidPort()

```
static bool isValidPort (
            const std::string & portStr )  [static]
```

7.12.1.3 main()

```
int main (
            int const argc,
            char const *const * argv )
```

The entry point of the program.

This function initializes the Rtype object and runs the game.

Returns

0 indicating successful program execution.

int

7.13 /home/runner/work/R-Type/R-Type/Server/Src/main.cpp File Reference

```
#include <Net/server.hpp>
#include <iostream>
#include <errno.h>
#include <signal.h>
#include <stdio.h>
```

Functions

- void signal_handler (int signal)

  Signal handler for SIGINT.
- static bool isValidPort (const std::string &portStr)

  Validates if a given string represents a valid port number.
- int main (int const argc, char const ∗const ∗const argv)

  Entry point for the server application.

Variables

- static bool loopRunning = true

  A static boolean flag to control the main loop execution.

7.13.1 Function Documentation

7.13.1.1 isValidPort()

```
static bool isValidPort (
            const std::string & portStr )  [static]
```

Validates if a given string represents a valid port number.

This function checks if the provided string is a valid port number within the range of 1024 to 65535. It performs the following checks:

- The string is not empty and does not exceed 5 characters in length.
- The string contains only digit characters.
- The integer value of the string is within the valid port range.

Parameters

portStr	The string representation of the port number to validate.

Returns

true if the string is a valid port number within the range 1024-65535, false otherwise.

7.13.1.2 main()

```
int main (
            int const argc,
            char const *const *const argv )
```

Entry point for the server application.

This function initializes the server, sets up signal handling, and enters the main loop.

Parameters

argc	The number of command-line arguments.
argv	The array of command-line arguments. The first argument should be the port number on which the server will listen.

Returns

Returns an error code if the usage is incorrect or the port number is invalid. Returns OK upon successful execution.

7.13.1.3 signal_handler()

```
void signal_handler (
            int signal )
```

Signal handler for SIGINT.

This function is called when the program receives a SIGINT signal (usually generated by pressing Ctrl+C). It sets the global variable `loopRunning` to false, which can be used to gracefully terminate a running loop.

Parameters

signal	The signal number received by the handler.

7.13.2 Variable Documentation

7.13.2.1 loopRunning

```
bool loopRunning = true  [static]
```

A static boolean flag to control the main loop execution.

This variable is used to determine whether the main loop should continue running. It is set to true initially, and can be modified to false to stop the loop.

7.14 /home/runner/work/R-Type/R-Type/Client/Src/scenes.cpp File Reference

```
#include <Components/components.hpp>
#include <Entities/entity_factory.hpp>
#include <Entities/entity_manager.hpp>
#include <Net/client.hpp>
#include <Systems/systems.hpp>
#include <audio_manager.hpp>
#include <chrono>
#include <creatable_client_object.hpp>
#include <font_manager.hpp>
#include <functional>
#include <iostream>
#include <scenes.hpp>
#include <sound_path.hpp>
#include <texture_manager.hpp>
```

Functions

- void reloadFilter (sf::RectangleShape &rectangle, AScenes::DaltonismMode mode)
- void handleEvents (sf::Event event, ComponentManager &componentManager, sf::RenderWindow *_window, std::vector< std::shared_ptr< Entity > > buttons, Scenes *scenes)

  Handles events for the scene, including window close and mouse button press events.
- void createDaltonismChoiceButtons (std::vector< std::shared_ptr< Entity > > &buttons, ComponentManager &componentManager, EntityManager &entityManager, TextureManager &textureManager, FontManager &fontManager, EntityFactory &entityFactory)
- sf::Keyboard::Key waitForKey (sf::RenderWindow *_window)
- void createKeyBindingButtons (std::vector< std::shared_ptr< Entity > > &buttons, ComponentManager &componentManager, EntityManager &entityManager, TextureManager &textureManager, FontManager &fontManager, EntityFactory &entityFactory, std::map< Scenes::Actions, sf::Keyboard::Key > &keyBinds)

7.14.1 Function Documentation

7.14.1.1 createDaltonismChoiceButtons()

```
void createDaltonismChoiceButtons (
            std::vector< std::shared_ptr< Entity > > & buttons,
            ComponentManager & componentManager,
            EntityManager & entityManager,
            TextureManager & textureManager,
            FontManager & fontManager,
            EntityFactory & entityFactory )
```

7.14.1.2 createKeyBindingButtons()

```
void createKeyBindingButtons (
            std::vector< std::shared_ptr< Entity > > & buttons,
            ComponentManager & componentManager,
            EntityManager & entityManager,
            TextureManager & textureManager,
            FontManager & fontManager,
            EntityFactory & entityFactory,
            std::map< Scenes::Actions, sf::Keyboard::Key > & keyBinds )
```

7.14.1.3 handleEvents()

```
void handleEvents (
            sf::Event event,
            ComponentManager & componentManager,
            sf::RenderWindow * _window,
            std::vector< std::shared_ptr< Entity > > buttons,
            Scenes * scenes )
```

Handles events for the scene, including window close and mouse button press events.

This function processes events from the given RenderWindow and performs actions based on the type of event. It handles window close events and mouse button press events. For mouse button press events, it checks if the left mouse button was pressed and if the click occurred within the bounds of any button entities. If a button is clicked, it triggers the associated OnClickComponent or BindComponent actions.

Parameters

event	The event to handle.
componentManager	Reference to the ComponentManager to access components of entities.
_window	Pointer to the RenderWindow where events are polled from.
buttons	Vector of shared pointers to Entity objects representing buttons.

7.14.1.4 reloadFilter()

```
void reloadFilter (
            sf::RectangleShape & rectangle,
            AScenes::DaltonismMode mode )
```

7.14.1.5 waitForKey()

```
sf::Keyboard::Key waitForKey (
            sf::RenderWindow * _window )
```

7.15 /home/runner/work/R-Type/R-Type/ECS/Interface/Include/a_↩ scenes.hpp File Reference

```
#include "Entities/entity.hpp"
#include "i_scenes.hpp"
#include <game_struct.hpp>
#include <memory>
```

Classes

- class AScenes

7.16 a_scenes.hpp

Go to the documentation of this file.

```
/*
** EPITECH PROJECT, 2024
** R-Type
** File description:
** a_scenes
*/

#pragma once

#include "Entities/entity.hpp"
#include "i_scenes.hpp"
#include <game_struct.hpp>
#include <memory>

class AScenes : virtual public IScenes {
  public:
    AScenes(std::string ip, int port);
    ~AScenes() = default;

    enum class Scene
    {
        MAIN_MENU,
        GAME_LOOP,
        SETTINGS_MENU,
        IN_GAME_MENU,
        CHOOSE_DIFFICULTY,
        CUSTOM_DIFFICULTY,
        TRANSITION_LEVEL,
        EXIT
    };

    enum class GameMode
    {
        EASY,
        MEDIUM,
        HARD
    };

    enum class DaltonismMode
    {
        NORMAL,
        TRITANOPIA,
        DEUTERANOPIA,
        PROTANOPIA
    };

    enum class Actions
    {
        UP,
        DOWN,
        LEFT,
        RIGHT,
        FIRE,
        PAUSE,
        QUIT
    };

    enum class SpriteType
    {
        BACKGROUND,
        PLAYER,
        ALLY,
        ENEMY,
        FILTER,
        WEAPON,
        POWER_UP,
        UI,
        OTHER
    };

    std::map<Actions, sf::Keyboard::Key> keyBinds = {{Actions::UP, sf::Keyboard::Key::Up},
        {Actions::DOWN, sf::Keyboard::Key::Down}, {Actions::LEFT, sf::Keyboard::Key::Left},
        {Actions::RIGHT, sf::Keyboard::Key::Right}, {Actions::FIRE, sf::Keyboard::Key::Space},
        {Actions::PAUSE, sf::Keyboard::Key::Escape}, {Actions::QUIT, sf::Keyboard::Key::Q}};

    void setScene(Scene scene);

    AScenes::Scene getPreviousScene();

    DaltonismMode getDaltonism() const { return _currentDaltonismMode; };

    void setDaltonism(DaltonismMode const mode);
```

```
    void setGameMode(GameParameters const mode);

    GameParameters getGameMode() const { return _currentGameMode; };

    void setDisplayDaltonismChoice(bool const displayDaltonismChoice);

    bool getDisplayDaltonismChoice() const;

    void setDisplayGameModeChoice(bool const displayGameModeChoice);

    bool getDisplayGameModeChoice() const;

    void setDisplayKeyBindsChoice(bool const displayKeyBindsChoice);

    bool getDisplayKeyBindsChoice() const;

    std::vector<std::shared_ptr<Entity>> buttons;
    std::shared_ptr<Entity> filter;
    void setIp(std::string ip);
    void setPort(int port);

    std::string getIp() const;
    int getPort() const;

    void SetPlayerReady(bool ready) { _playerReady = ready; };

    bool GetPlayerReady() const { return _playerReady; };

  protected:
    GameParameters _currentGameMode;
    DaltonismMode _currentDaltonismMode = DaltonismMode::NORMAL;
    Scene _currentScene = Scene::MAIN_MENU;
    Scene _previousScene = Scene::MAIN_MENU;
    bool _displayDaltonismChoice = false;
    bool _displayGameModeChoice = false;
    bool _displayKeyBindsChoice = false;

    std::string _ip;
    int _port;

    bool _playerReady = false;
};
```

7.17 /home/runner/work/R-Type/R-Type/ECS/Interface/Include/audio_manager.hpp File Reference

```
#include "error_handling.hpp"
#include <SFML/Audio.hpp>
#include <memory>
#include <string>
#include <unordered_map>
```

Classes

- class AudioManager

  Manages and caches sound buffers for efficient audio playback.

7.18 audio_manager.hpp

Go to the documentation of this file.

```
/*
** EPITECH PROJECT, 2024
** R-Type
** File description:
** audio_manager
```

```
*/

#pragma once

#include "error_handling.hpp"
#include <SFML/Audio.hpp>
#include <memory>
#include <string>
#include <unordered_map>

class AudioManager {
  public:
    sf::SoundBuffer &getSoundBuffer(const std::string &filePath)
    {
        // Check if sound buffer is already cached
        auto it = soundBuffers.find(filePath);
        if (it != soundBuffers.end()) {
            return *it->second;
        }

        // Load and cache the sound buffer
        auto buffer = std::make_shared<sf::SoundBuffer>();
        if (!buffer->loadFromFile(filePath)) {
            throw std::runtime_error("Failed to load sound from " + filePath);
        }
        soundBuffers[filePath] = buffer;
        return *buffer;
    }

  private:
    std::unordered_map<std::string, std::shared_ptr<sf::SoundBuffer>> soundBuffers;
};
```

7.19 /home/runner/work/R-Type/R-Type/ECS/Interface/Include/↩ Components/ally_component.hpp File Reference

Defines the AllyComponent structure.

Classes

- struct AllyComponent

7.19.1 Detailed Description

Defines the AllyComponent structure.

The AllyComponent is used to mark entities as allies within the ECS framework.

7.20 ally_component.hpp

Go to the documentation of this file.

```
/*
** EPITECH PROJECT, 2024
** R-Type
** File description:
** ally_component
*/

#pragma once

struct AllyComponent {};
```

7.21 /home/runner/work/R-Type/R-Type/ECS/Interface/Include/↩ Components/ally_missile_component.hpp File Reference

Defines the AllyMissileComponent structure.

Classes

- struct AllyMissileComponent

7.21.1 Detailed Description

Defines the AllyMissileComponent structure.

The AllyMissileComponent is used to represent a missile fired by an ally in the game. This component can be attached to an entity to give it the behavior and properties of an ally missile.

7.22 ally_missile_component.hpp

Go to the documentation of this file.

```
00001 /*
00002 ** EPITECH PROJECT, 2024
00003 ** R-Type
00004 ** File description:
00005 ** ally_missile_component
00006 */
00007
00008 #pragma once
00009
00018 struct AllyMissileComponent {};
```

7.23 /home/runner/work/R-Type/R-Type/ECS/Interface/Include/↩ Components/animation_component.hpp File Reference

```
#include <macros.hpp>
```

Classes

- struct AnimationComponent

  A component that holds animation properties such as offset and dimension.

Functions

- bool operator!= (AnimationComponent animation, AnimationComponent other)

  Inequality operator for AnimationComponent.

7.23.1 Function Documentation

7.23.1.1 operator"!=()

```
bool operator!= (
            AnimationComponent animation,
            AnimationComponent other )
```

Inequality operator for AnimationComponent.

This operator checks if two AnimationComponent instances are not equal.

Parameters

animation	The first AnimationComponent instance.
other	The second AnimationComponent instance.

Returns

true if the two AnimationComponent instances are not equal, false otherwise.

This operator compares two AnimationComponent objects to determine if they are not equal. Two AnimationComponent objects are considered not equal if any of their respective offset or dimension coordinates differ.

Parameters

animation	The first AnimationComponent to compare.
other	The second AnimationComponent to compare.

Returns

true if the AnimationComponent objects are not equal, false otherwise.

7.24 animation_component.hpp

Go to the documentation of this file.

```
/*
** EPITECH PROJECT, 2024
** R-Type
** File description:
** velocity_component
*/

#pragma once

#include <macros.hpp>

struct AnimationComponent {
    vf2d offset;
    vf2d dimension;

    AnimationComponent(vf2d _offset, vf2d _dimension) : offset(_offset), dimension(_dimension) {}
};

bool operator!=(AnimationComponent animation, AnimationComponent other);
```

7.25 /home/runner/work/R-Type/R-Type/ECS/Interface/Include/↩ Components/background_component.hpp File Reference

Defines the BackgroundComponent structure.

Classes

- struct BackgroundComponent

7.25.1 Detailed Description

Defines the BackgroundComponent structure.

The BackgroundComponent is used to represent the background in the ECS (Entity Component System). This component can be attached to entities that require a background.

7.26 background_component.hpp

Go to the documentation of this file.

```
00001 /*
00002 ** EPITECH PROJECT, 2024
00003 ** R-Type
00004 ** File description:
00005 ** background_component
00006 */
00007
00008 #pragma once
00009
00017 struct BackgroundComponent {};
```

7.27 /home/runner/work/R-Type/R-Type/ECS/Interface/Include/↩ Components/basic_monster_component.hpp File Reference

Defines the BasicMonsterComponent structure.

Classes

- struct BasicMonsterComponent

7.27.1 Detailed Description

Defines the BasicMonsterComponent structure.

This component is used to represent basic monster entities in the game.

7.28 basic_monster_component.hpp

Go to the documentation of this file.

```
00001 /*
00002 ** EPITECH PROJECT, 2024
00003 ** R-Type
00004 ** File description:
00005 ** basic_monster_component
00006 */
00007
00008 #pragma once
00009
00016 struct BasicMonsterComponent {};
```

7.29 /home/runner/work/R-Type/R-Type/ECS/Interface/Include/↩ Components/bind_component.hpp File Reference

```
#include "a_scenes.hpp"
#include "i_scenes.hpp"
#include <functional>
```

Classes

- struct BindComponent

  A component that binds a function to handle scene transitions.

7.30 bind_component.hpp

Go to the documentation of this file.

```
00001 /*
00002 ** EPITECH PROJECT, 2024
00003 ** R-Type
00004 ** File description:
00005 ** bind_component
00006 */
00007
00008 #pragma once
00009
00010 #include "a_scenes.hpp"
00011 #include "i_scenes.hpp"
00012 #include <functional>
00013
00035 struct BindComponent {
00036     bool isHovered = false;
00037     std::function<IScenes *(AScenes *, AScenes::Actions)> bind;
00038
00039     BindComponent(std::function<IScenes *(AScenes *, AScenes::Actions)> bindFunction)
00040         : bind(bindFunction){};
00041 };
```

7.31 /home/runner/work/R-Type/R-Type/ECS/Interface/Include/↩ Components/boss_component.hpp File Reference

```
#include <vector>
```

Classes

- struct BossComponent

7.32 boss_component.hpp

Go to the documentation of this file.

```
00001 /*
00002 ** EPITECH PROJECT, 2024
00003 ** R-Type
00004 ** File description:
00005 ** boss_component
00006 */
00007
00008 #pragma once
00009
00010 #include <vector>
00011
00012 struct BossComponent {
00013     std::vector<int> tailSegmentIds;
00014 };
```

7.33 /home/runner/work/R-Type/R-Type/ECS/Interface/Include/↩ Components/component_manager.hpp File Reference

```
#include "components.hpp"
#include "texture_manager.hpp"
#include <any>
#include <iostream>
#include <memory>
#include <optional>
#include <typeindex>
#include <unordered_map>
```

Classes

- class ComponentManager

  Manages the components of entities in an ECS system.

7.34 component_manager.hpp

Go to the documentation of this file.

```
00001 /*
00002 ** EPITECH PROJECT, 2024
00003 ** R-Type
00004 ** File description:
00005 ** component_manager
00006 */
00007
00008 #pragma once
00009
00010 #include "components.hpp"
00011 #include "texture_manager.hpp"
00012 #include <any>
00013 #include <iostream>
00014 #include <memory>
00015 #include <optional>
00016 #include <typeindex>
00017 #include <unordered_map>
00018
00028 class ComponentManager {
00029   public:
00030     // Usage ex: CMName.addComponent<NameOfComponent>(entityId, arg1, arg2);
00039     template <typename ComponentType, typename... Args>
00040     void addComponent(int entityId, Args &&...args)
00041     {
00042         ComponentType component(std::forward<Args>(args)...);
00043         components[typeid(ComponentType)][entityId] =
00044             std::make_any<ComponentType>(std::move(component));
00045     }
00046
00054     template <typename ComponentType> std::optional<ComponentType *> getComponent(int entityId)
00055     {
00056         if (components.find(typeid(ComponentType)) != components.end()) {
00057             auto &entityComponents = components[typeid(ComponentType)];
00058             if (entityComponents.find(entityId) != entityComponents.end()) {
00059                 return std::any_cast<ComponentType>(&entityComponents[entityId]);
00060             }
00061         }
00062         return std::nullopt; // Return nullopt if not found
00063     }
00064
00072     template <typename ComponentType>
00073     std::optional<std::unordered_map<int, std::any> *> getComponentMap()
00074     {
00075         auto it = components.find(typeid(ComponentType));
00076         if (it != components.end()) {
00077             return &it->second;
00078         }
00079         return std::nullopt;
```

```
    }

    template <typename ComponentType> void removeEntityFromComponent(int entityId)
    {
        auto it = components.find(typeid(ComponentType));
        if (it != components.end()) {
            auto &entityComponents = it->second;
            entityComponents.erase(entityId);
        }
    }

    void removeAllComponents()
    {
        if (components.empty())
            return;
        std::cout « "Removing all components" « std::endl;
        components.clear();
    }

    void removeEntityFromAllComponents(int entityId)
    {
        for (auto &component : components) {
            component.second.erase(entityId);
        }
    }

  private:
    // unordered map of <componentType, it's unordered map of <entityId, it's
    // values for the componentType»
    std::unordered_map<std::type_index, std::unordered_map<int, std::any» components;
};
```

7.35 /home/runner/work/R-Type/R-Type/ECS/Interface/Include/↩ Components/components.hpp File Reference

```
#include "ally_component.hpp"
#include "ally_missile_component.hpp"
#include "animation_component.hpp"
#include "background_component.hpp"
#include "basic_monster_component.hpp"
#include "bind_component.hpp"
#include "boss_component.hpp"
#include "enemy_component.hpp"
#include "enemy_missile_component.hpp"
#include "force_missile_component.hpp"
#include "force_weapon_component.hpp"
#include "front_component.hpp"
#include "health_component.hpp"
#include "hitbox_component.hpp"
#include "input_component.hpp"
#include "link_force_component.hpp"
#include "movement_component.hpp"
#include "offset_component.hpp"
#include "on_click_component.hpp"
#include "player_component.hpp"
#include "player_missile_component.hpp"
#include "position_component.hpp"
#include "power_up_component.hpp"
#include "rectangleShapeComponent.hpp"
#include "score_component.hpp"
#include "shoot_component.hpp"
#include "sprite_component.hpp"
#include "sprite_data_component.hpp"
#include "tail_component.hpp"
#include "text_component.hpp"
```

```
#include "text_data_component.hpp"
#include "update_text_component.hpp"
#include "velocity_component.hpp"
#include "wall_component.hpp"
```

7.36 components.hpp

Go to the documentation of this file.

```
/*
** EPITECH PROJECT, 2024
** R-Type
** File description:
** components
*/

#pragma once

#include "ally_component.hpp"
#include "ally_missile_component.hpp"
#include "animation_component.hpp"
#include "background_component.hpp"
#include "basic_monster_component.hpp"
#include "bind_component.hpp"
#include "boss_component.hpp"
#include "enemy_component.hpp"
#include "enemy_missile_component.hpp"
#include "force_missile_component.hpp"
#include "force_weapon_component.hpp"
#include "front_component.hpp"
#include "health_component.hpp"
#include "hitbox_component.hpp"
#include "input_component.hpp"
#include "link_force_component.hpp"
#include "movement_component.hpp"
#include "offset_component.hpp"
#include "on_click_component.hpp"
#include "player_component.hpp"
#include "player_missile_component.hpp"
#include "position_component.hpp"
#include "power_up_component.hpp"
#include "rectangleShapeComponent.hpp"
#include "score_component.hpp"
#include "shoot_component.hpp"
#include "sprite_component.hpp"
#include "sprite_data_component.hpp"
#include "tail_component.hpp"
#include "text_component.hpp"
#include "text_data_component.hpp"
#include "update_text_component.hpp"
#include "velocity_component.hpp"
#include "wall_component.hpp"
```

7.37 /home/runner/work/R-Type/R-Type/ECS/Interface/Include/↩ Components/enemy_component.hpp File Reference

Defines the EnemyComponent structure.

Classes

- struct EnemyComponent

7.37.1 Detailed Description

Defines the EnemyComponent structure.

This file contains the definition of the EnemyComponent structure, which is used to represent an enemy entity in the game. The structure itself is currently empty, but it can be extended in the future to include properties and behaviors specific to enemies.

7.38 enemy_component.hpp

Go to the documentation of this file.

```
/*
** EPITECH PROJECT, 2024
** R-Type
** File description:
** enemy_component
*/

#pragma once

struct EnemyComponent {};
```

7.39 /home/runner/work/R-Type/R-Type/ECS/Interface/Include/↩ Components/enemy_missile_component.hpp File Reference

Defines the EnemyMissileComponent structure.

Classes

- struct EnemyMissileComponent

7.39.1 Detailed Description

Defines the EnemyMissileComponent structure.

This component is used to represent an enemy missile in the game.

7.40 enemy_missile_component.hpp

Go to the documentation of this file.

```
/*
** EPITECH PROJECT, 2024
** R-Type
** File description:
** enemy_missile_component
*/

#pragma once

struct EnemyMissileComponent {};
```

7.41 /home/runner/work/R-Type/R-Type/ECS/Interface/Include/↩ Components/force_missile_component.hpp File Reference

```
#include <cstdint>
```

Classes

- struct ForceMissileComponent

  Component representing a force missile in the ECS system.

7.42 force_missile_component.hpp

Go to the documentation of this file.

```
00001 /*
00002 ** EPITECH PROJECT, 2024
00003 ** R-Type
00004 ** File description:
00005 ** missile_component
00006 */
00007
00008 #pragma once
00009
00010 #include <cstdint>
00011
00022 struct ForceMissileComponent {
00023     uint32_t forceId;
00024 };
```

7.43 /home/runner/work/R-Type/R-Type/ECS/Interface/Include/↩ Components/force_weapon_component.hpp File Reference

```
#include <cstdint>
```

Classes

- struct ForceWeaponComponent

  Represents a component for a force weapon in the game.

7.44 force_weapon_component.hpp

Go to the documentation of this file.

```
00001 /*
00002 ** EPITECH PROJECT, 2024
00003 ** R-Type
00004 ** File description:
00005 ** weapon_component
00006 */
00007
00008 #pragma once
00009
00010 #include <cstdint>
00011
00036 struct ForceWeaponComponent {
00037     uint32_t playerId;
00038     uint32_t level;
00039     bool attached;
00040
00041     ForceWeaponComponent(uint32_t _playerId, uint32_t _level, uint32_t _attached)
00042         : playerId(_playerId), level(_level), attached(_attached)
00043     {
00044     }
00045 };
```

7.45 /home/runner/work/R-Type/R-Type/ECS/Interface/Include/↩ Components/front_component.hpp File Reference

```
#include <Entities/entity.hpp>
#include <memory>
```

Classes

- struct FrontComponent

  A component that represents the front of an entity.

7.46 front_component.hpp

Go to the documentation of this file.

```
/*
** EPITECH PROJECT, 2024
** R-Type
** File description:
** ally_component
*/

#pragma once

#include <Entities/entity.hpp>
#include <memory>

struct FrontComponent {
    int targetId;

    FrontComponent(int _targetId) : targetId(_targetId) {}
};
```

7.47 /home/runner/work/R-Type/R-Type/ECS/Interface/Include/↩ Components/health_component.hpp File Reference

Classes

- struct HealthComponent

  Represents the health attributes of an entity.

7.48 health_component.hpp

Go to the documentation of this file.

```
/*
** EPITECH PROJECT, 2024
** R-Type
** File description:
** health_component
*/

#pragma once

struct HealthComponent {
    int lives;
};
```

7.49 /home/runner/work/R-Type/R-Type/ECS/Interface/Include/↩ Components/hitbox_component.hpp File Reference

Classes

- struct HitboxComponent

  Represents the hitbox dimensions of an entity.

7.50 hitbox_component.hpp

Go to the documentation of this file.

```
/*
** EPITECH PROJECT, 2024
** R-Type
** File description:
** hitbox_component
*/

#pragma once

struct HitboxComponent {
    int w;
    int h;
};
```

7.51 /home/runner/work/R-Type/R-Type/ECS/Interface/Include/↩ Components/input_component.hpp File Reference

Classes

- struct InputComponent

  Component for handling input actions.

Enumerations

- enum class InputType {
  UP , DOWN , LEFT , RIGHT ,
  SHOOT , QUIT , NONE }

  Enumeration of possible input actions.

7.51.1 Enumeration Type Documentation

7.51.1.1 InputType

```
enum class InputType  [strong]
```

Enumeration of possible input actions.

This enumeration defines the different types of inputs that can be handled by the InputComponent.

Enumerator

UP	Represents the "up" input action.
DOWN	Represents the "down" input action.
LEFT	Represents the "left" input action.
RIGHT	Represents the "right" input action.
SHOOT	Represents the "shoot" input action.
QUIT	Represents the "quit" input action.
NONE	Represents no input action.

7.52 input_component.hpp

Go to the documentation of this file.

```
00001 /*
00002 ** EPITECH PROJECT, 2024
00003 ** R-Type
00004 ** File description:
00005 ** input_component
00006 */
00007
00008 #pragma once
00009
00038 enum class InputType
00039 {
00040     UP,
00041     DOWN,
00042     LEFT,
00043     RIGHT,
00044     SHOOT,
00045     QUIT,
00046     NONE
00047 };
00048
00058 struct InputComponent {
00059     InputType input;
00060 };
```

7.53 /home/runner/work/R-Type/R-Type/ECS/Interface/Include/↩ Components/label_component.hpp File Reference

```
#include <string>
```

Classes

- struct labelComponent

  Represents a label component with a name and position coordinates.

7.54 label_component.hpp

Go to the documentation of this file.

```
/*
** EPITECH PROJECT, 2024
** R-Type
** File description:
** label_component
*/

#pragma once

#include <string>

struct labelComponent {
    std::string name;
    int x;
    int y;
};
```

7.55 /home/runner/work/R-Type/R-Type/ECS/Interface/Include/↩ Components/link_force_component.hpp File Reference

Classes

- struct LinkForceComponent

  Component that links an entity to a target entity by ID.

7.56 link_force_component.hpp

Go to the documentation of this file.

```
/*
** EPITECH PROJECT, 2024
** R-Type
** File description:
** ally_component
*/

#pragma once

struct LinkForceComponent {
    int targetId;

    LinkForceComponent(int _targetId) : targetId(_targetId) {}
};
```

7.57 /home/runner/work/R-Type/R-Type/ECS/Interface/Include/↩ Components/movement_component.hpp File Reference

```
#include <cstdint>
```

Classes

- struct MovementComponent

  Represents a component that handles movement in the ECS system.

Enumerations

- enum class MovementType {
  WIGGLE , DIAGONAL , CIRCLE , STRAIGHT ,
  SWEEPING , NONE }

  Enumeration of different types of movement behaviors.

7.57.1 Enumeration Type Documentation

7.57.1.1 MovementType

```
enum class MovementType  [strong]
```

Enumeration of different types of movement behaviors.

Enumerator

WIGGLE	Represents a wiggling movement pattern.
DIAGONAL	Represents a diagonal movement pattern.
CIRCLE	Represents a circular movement pattern.
STRAIGHT	
SWEEPING	
NONE	

7.58 movement_component.hpp

Go to the documentation of this file.

```
/*
** EPITECH PROJECT, 2024
** R-Type
** File description:
** player_component
*/

#pragma once

#include <cstdint>

enum class MovementType
{
    WIGGLE,
    DIAGONAL,
    CIRCLE,
    STRAIGHT,
    SWEEPING,
    NONE
};

struct MovementComponent {
    MovementType movementType;
    uint32_t index;
    bool move;

    MovementComponent() : movementType(MovementType::STRAIGHT), index(0), move(true) {}
    MovementComponent(MovementType movementType, uint32_t index, bool move)
        : movementType(movementType), index(index), move(move)
    {
    }
};
```

7.59 /home/runner/work/R-Type/R-Type/ECS/Interface/Include/↩ Components/offset_component.hpp File Reference

Classes

- struct OffsetComponent

  Component that represents an offset value.

7.60 offset_component.hpp

Go to the documentation of this file.

```
/*
** EPITECH PROJECT, 2024
** R-Type
** File description:
** offset_component
*/

#pragma once

struct OffsetComponent {
    float offset;
};
```

7.61 /home/runner/work/R-Type/R-Type/ECS/Interface/Include/↩ Components/on_click_component.hpp File Reference

Defines the OnClickComponent structure used for handling click events in the ECS system.

```
#include <a_scenes.hpp>
#include <functional>
#include <i_scenes.hpp>
```

Classes

- struct OnClickComponent

  Component that handles click events.

7.61.1 Detailed Description

Defines the OnClickComponent structure used for handling click events in the ECS system.

7.62 on_click_component.hpp

Go to the documentation of this file.

```
/*
** EPITECH PROJECT, 2024
** R-Type
** File description:
** on_click_component
*/

#pragma once
#include <a_scenes.hpp>
#include <functional>
#include <i_scenes.hpp>

struct OnClickComponent {
    bool isClicked = false;
    std::function<IScenes *(AScenes *)> onClick;

    OnClickComponent(std::function<IScenes *(AScenes *)> onClickFunction)
        : onClick(onClickFunction){};
};
```

7.63 /home/runner/work/R-Type/R-Type/ECS/Interface/Include/↩ Components/player_component.hpp File Reference

Defines the PlayerComponent structure.

Classes

- struct PlayerComponent

7.63.1 Detailed Description

Defines the PlayerComponent structure.

The PlayerComponent structure is used to represent a player entity within the ECS (Entity Component System) framework of the R-Type project.

7.64 player_component.hpp

Go to the documentation of this file.

```
/*
** EPITECH PROJECT, 2024
** R-Type
** File description:
** player_component
*/

#pragma once

struct PlayerComponent {};
```

7.65 /home/runner/work/R-Type/R-Type/ECS/Interface/Include/↩ Components/player_missile_component.hpp File Reference

```
#include <cstdint>
```

Classes

- struct PlayerMissileComponent

  Component that represents a missile belonging to a player.

7.66 player_missile_component.hpp

Go to the documentation of this file.

```
/*
** EPITECH PROJECT, 2024
** R-Type
** File description:
** missile_component
*/

#pragma once

#include <cstdint>

struct PlayerMissileComponent {
    uint32_t playerId;
};
```

7.67 /home/runner/work/R-Type/R-Type/ECS/Interface/Include/↩ Components/position_component.hpp File Reference

Classes

- struct PositionComponent

  A component that represents the position of an entity in 2D space.

7.68 position_component.hpp

Go to the documentation of this file.

```
/*
** EPITECH PROJECT, 2024
** R-Type
** File description:
** position_component
*/

#pragma once

struct PositionComponent {
    float x;
    float y;

    PositionComponent(float _x, float _y) : x(_x), y(_y) {}
};
```

7.69 /home/runner/work/R-Type/R-Type/ECS/Interface/Include/↩ Components/power_up_component.hpp File Reference

Defines the PowerUpComponent structure.

Classes

- struct PowerUpComponent

7.69.1 Detailed Description

Defines the PowerUpComponent structure.

The PowerUpComponent structure is used to represent a power-up in the game. It can be attached to entities to give them special abilities or enhancements.

7.70 power_up_component.hpp

Go to the documentation of this file.

```
00001 /*
00002 ** EPITECH PROJECT, 2024
00003 ** R-Type
00004 ** File description:
00005 ** player_component
00006 */
00007
00008 #pragma once
00009
00017 struct PowerUpComponent {};
```

7.71 /home/runner/work/R-Type/R-Type/ECS/Interface/Include/↩ Components/rectangleShapeComponent.hpp File Reference

```
#include <SFML/Graphics.hpp>
```

Classes

- struct RectangleShapeComponent

  A component that holds an sf::RectangleShape.

7.72 rectangleShapeComponent.hpp

Go to the documentation of this file.

```
00001 /*
00002 ** EPITECH PROJECT, 2024
00003 ** R-Type
00004 ** File description:
00005 ** rectangleShapeComponent
00006 */
00007
00008 #pragma once
00009 #include <SFML/Graphics.hpp>
00010
00017 struct RectangleShapeComponent {
00018     sf::RectangleShape rectangleShape;
00019
00025     RectangleShapeComponent(sf::RectangleShape &rectangleShape)
00026     {
00027         this->rectangleShape = rectangleShape;
00028     }
00029 };
```

7.73 /home/runner/work/R-Type/R-Type/ECS/Interface/Include/↩ Components/score_component.hpp File Reference

Defines the ScoreComponent struct used to store the score of an entity.

Classes

- struct ScoreComponent

  Component that holds the score of an entity.

7.73.1 Detailed Description

Defines the ScoreComponent struct used to store the score of an entity.

7.74 score_component.hpp

Go to the documentation of this file.

```
00001 /*
00002 ** EPITECH PROJECT, 2024
00003 ** R-Type
00004 ** File description:
00005 ** score_component
00006 */
00007
00008 #pragma once
00009
00022 struct ScoreComponent {
00023     int score;
00024 };
```

7.75 /home/runner/work/R-Type/R-Type/ECS/Interface/Include/↩ Components/shader_component.hpp File Reference

```
#include <SFML/Graphics.hpp>
#include <iostream>
#include <memory>
```

Classes

- struct ShaderComponent

  A component that holds a shader.

7.76 shader_component.hpp

Go to the documentation of this file.

```
/*
** EPITECH PROJECT, 2024
** R-Type
** File description:
** shader_component
*/

#pragma once
#include <SFML/Graphics.hpp>
#include <iostream>
#include <memory>

struct ShaderComponent {
    std::shared_ptr<sf::Shader> shader;

    ShaderComponent(std::string path)
    {
        shader = std::make_shared<sf::Shader>();
        if (!shader->loadFromFile(path, sf::Shader::Fragment)) {
            std::cerr « "Error loading shader" « std::endl;
        }
    }
};
```

7.77 /home/runner/work/R-Type/R-Type/ECS/Interface/Include/↩ Components/shoot_component.hpp File Reference

```
#include <chrono>
```

Classes

- struct ShootComponent

  Component that handles shooting mechanics for an entity.

7.78 shoot_component.hpp

Go to the documentation of this file.

```
/*
** EPITECH PROJECT, 2024
** R-Type
** File description:
** shoot_component
*/

#pragma once

#include <chrono>

struct ShootComponent {
    std::chrono::system_clock::time_point nextShootTime;
    std::chrono::milliseconds cooldownTime;
    bool canShoot;

    ShootComponent(std::chrono::milliseconds cooldown)
        : nextShootTime(std::chrono::system_clock::now()), cooldownTime(cooldown)
    {
    }
};
```

7.79 /home/runner/work/R-Type/R-Type/ECS/Interface/Include/↩ Components/sprite_component.hpp File Reference

```
#include "a_scenes.hpp"
#include <SFML/Graphics.hpp>
#include <string>
```

Classes

- struct SpriteComponent

  A component that represents a sprite in the ECS (Entity Component System).

7.80 sprite_component.hpp

Go to the documentation of this file.

```
/*
** EPITECH PROJECT, 2024
** R-Type
** File description:
** sprite_component
*/

#pragma once

#include "a_scenes.hpp"
#include <SFML/Graphics.hpp>
#include <string>

struct SpriteComponent {
    sf::Sprite sprite;
    AScenes::SpriteType type;
    int hitboxX;
    int hitboxY;

    SpriteComponent(sf::Texture &texture, const float posX, float posY, const sf::Vector2f &scale,
        AScenes::SpriteType typeNb, sf::IntRect rect = sf::IntRect(0, 0, 0, 0))
    {
        type = typeNb;
        sprite.setTexture(texture);
        sprite.setPosition(posX, posY);
        sprite.setScale(scale);
        if (rect != sf::IntRect(0, 0, 0, 0))
            sprite.setTextureRect(rect);
        sprite.setOrigin(sprite.getLocalBounds().width / 2, sprite.getLocalBounds().height / 2);
        hitboxX = rect.width;
        hitboxY = rect.height;
    }
};
```

7.81 /home/runner/work/R-Type/R-Type/ECS/Interface/Include/↩ Components/sprite_data_component.hpp File Reference

```
#include "../error_handling.hpp"
#include "../sprite_path.hpp"
#include "animation_component.hpp"
#include "position_component.hpp"
#include <SFML/Graphics.hpp>
#include <a_scenes.hpp>
#include <cstdint>
#include <macros.hpp>
#include <string>
```

Classes

- struct SpriteDataComponent

  Component that holds data related to a sprite.

Functions

- std::ostream & operator<< (std::ostream &os, const SpriteDataComponent &spriteData)

  Overloads the << operator to output the contents of a SpriteDataComponent to an ostream.

7.81.1 Function Documentation

7.81.1.1 operator<<()

```
std::ostream & operator<< (
            std::ostream & os,
            const SpriteDataComponent & spriteData )
```

Overloads the << operator to output the contents of a SpriteDataComponent to an ostream.

Parameters

os	The output stream to which the SpriteDataComponent will be written.
spriteData	The SpriteDataComponent instance to be written to the output stream.

Returns

std::ostream& The output stream after writing the SpriteDataComponent.

7.82 sprite_data_component.hpp

Go to the documentation of this file.

```
/*
** EPITECH PROJECT, 2024
** R-Type
** File description:
** sprite_component
*/

#pragma once

#include "../error_handling.hpp"
#include "../sprite_path.hpp"
#include "animation_component.hpp"
#include "position_component.hpp"
#include <SFML/Graphics.hpp>
#include <a_scenes.hpp>
#include <cstdint>
#include <macros.hpp>
#include <string>

struct SpriteDataComponent {
    SpritePath spritePath;
    vf2d scale;
    AScenes::SpriteType type;
};

std::ostream &operator«(std::ostream &os, const SpriteDataComponent &spriteData);
```

7.83 /home/runner/work/R-Type/R-Type/ECS/Interface/Include/↩ Components/tail_component.hpp File Reference

Classes

- struct TailComponent

7.84 tail_component.hpp

Go to the documentation of this file.

```
/*
** EPITECH PROJECT, 2024
** R-Type
** File description:
** tail_component
*/

#pragma once

struct TailComponent {};
```

7.85 /home/runner/work/R-Type/R-Type/ECS/Interface/Include/↩ Components/text_component.hpp File Reference

```
#include <SFML/Graphics.hpp>
```

Classes

- struct TextComponent

  A component that encapsulates an SFML text object.

7.86 text_component.hpp

Go to the documentation of this file.

```
/*
** EPITECH PROJECT, 2024
** R-Type
** File description:
** text_component
*/

#pragma once

#include <SFML/Graphics.hpp>

struct TextComponent {
    sf::Text text;

    TextComponent(sf::Font &font, const std::string &string, float posX, float posY, int size = 30)
    {
        text.setCharacterSize(size);
        text.setFont(font);
        text.setString(string);
        text.setPosition(posX, posY);
        text.setFillColor(sf::Color::White);
        text.setStyle(sf::Text::Bold);
    }
};
```

7.87 /home/runner/work/R-Type/R-Type/ECS/Interface/Include/↩ Components/text_data_component.hpp File Reference

```
#include "../font_path.hpp"
#include "../game_text.hpp"
```

Classes

- struct TextDataComponent

  Component that holds text-related data for an entity.

7.88 text_data_component.hpp

Go to the documentation of this file.

```
00001 /*
00002 ** EPITECH PROJECT, 2024
00003 ** R-Type
00004 ** File description:
00005 ** text_data_component
00006 */
00007
00008 #pragma once
00009
00010 #include "../font_path.hpp"
00011 #include "../game_text.hpp"
00012
00035 struct TextDataComponent {
00036     FontPath fontPath;
00037     uint32_t charSize = 0;
00038     uint32_t categoryIds[5] = {0};
00039     GameText categoryTexts[5];
00040     uint32_t categorySize = 0;
00041 };
```

7.89 /home/runner/work/R-Type/R-Type/ECS/Interface/Include/↩ Components/update_text_component.hpp File Reference

```
#include <a_scenes.hpp>
#include <functional>
#include <i_scenes.hpp>
#include <string>
```

Classes

- struct UpdateTextComponent

7.90 update_text_component.hpp

Go to the documentation of this file.

```
/*
** EPITECH PROJECT, 2024
** R-Type
** File description:
** on_click_component
*/

#pragma once
#include <a_scenes.hpp>
#include <functional>
#include <i_scenes.hpp>
#include <string>

struct UpdateTextComponent {
    std::function<std::string(GameParameters)> updateText;

    UpdateTextComponent(std::function<std::string(GameParameters)> updateTextFunction)
        : updateText(updateTextFunction){};
};
```

7.91 /home/runner/work/R-Type/R-Type/ECS/Interface/Include/↩ Components/velocity_component.hpp File Reference

Classes

- struct VelocityComponent

  Represents the velocity of an entity in 2D space.

7.92 velocity_component.hpp

Go to the documentation of this file.

```
/*
** EPITECH PROJECT, 2024
** R-Type
** File description:
** velocity_component
*/

#pragma once

struct VelocityComponent {
    float x;
    float y;
};
```

7.93 /home/runner/work/R-Type/R-Type/ECS/Interface/Include/↩ Components/wall_component.hpp File Reference

Defines the WallComponent structure.

Classes

- struct WallComponent

7.93.1 Detailed Description

Defines the WallComponent structure.

The WallComponent is a marker component used to identify entities that represent walls in the ECS (Entity Component System).

7.94 wall_component.hpp

Go to the documentation of this file.

```
/*
** EPITECH PROJECT, 2024
** R-Type
** File description:
** enemy_missile_component
*/

#pragma once

struct WallComponent {};
```

7.95 /home/runner/work/R-Type/R-Type/ECS/Interface/Include/creatable_client_object.hpp File Reference

```
#include <cstdint>
```

Enumerations

- enum class CreatableClientObject : uint32_t { PLAYERMISSILE , NONE }

  Enum representing the types of client objects that can be created.

7.95.1 Enumeration Type Documentation

7.95.1.1 CreatableClientObject

```
enum class CreatableClientObject : uint32_t [strong]
```

Enum representing the types of client objects that can be created.

This enum is used to specify the different types of objects that can be instantiated on the client side in the R-Type game.

Enumerator

PLAYERMISSILE	Represents a missile fired by the player.
NONE	Represents the absence of a creatable client object.

7.96 creatable_client_object.hpp

Go to the documentation of this file.

```
/*
** EPITECH PROJECT, 2024
** R-Type
** File description:
** creatable_client_object
*/

#pragma once

#include <cstdint>

enum class CreatableClientObject : uint32_t
{
    PLAYERMISSILE,
    NONE
};
```

7.97 /home/runner/work/R-Type/R-Type/ECS/Interface/Include/↩ Entities/entity.hpp File Reference

Classes

- class Entity

  Represents an entity in the ECS system.

7.98 entity.hpp

Go to the documentation of this file.

```
/*
** EPITECH PROJECT, 2024
** R-Type
** File description:
** entity
*/

#pragma once

class Entity {
  public:
    explicit Entity(int id) : _id(id) {}

    int getId() const { return _id; }

  private:
    int _id;
};
```

7.99 /home/runner/work/R-Type/R-Type/ECS/Interface/Include/↩ Entities/entity_factory.hpp File Reference

```
#include "a_scenes.hpp"
#include "i_entity_factory.hpp"
#include "i_scenes.hpp"
#include <functional>
#include <game_struct.hpp>
```

Classes

- class EntityFactory

  A factory class for creating various types of entities.

7.100 entity_factory.hpp

Go to the documentation of this file.

```
/*
** EPITECH PROJECT, 2024
** R-Type
** File description:
** entity_factory
*/

#pragma once

#include "a_scenes.hpp"
#include "i_entity_factory.hpp"
#include "i_scenes.hpp"
#include <functional>
#include <game_struct.hpp>

class EntityFactory : public IEntityFactory {
  public:
    Entity createBackgroundLevelOne(
        EntityManager &entityManager, ComponentManager &componentManager) override;

    Entity createBackgroundLevelTwo(
        EntityManager &entityManager, ComponentManager &componentManager) override;

    Entity createBackgroundLevelThree(
        EntityManager &entityManager, ComponentManager &componentManager) override;

    Entity createBackgroundMenu(EntityManager &entityManager, ComponentManager &componentManager,
        TextureManager &textureManager) override;

    Entity createInfoBar(
        EntityManager &entityManager, ComponentManager &componentManager) override;

    Entity createPlayer(EntityManager &entityManager, ComponentManager &componentManager,
        int nbrOfPlayers) override;

    Entity createShooterEnemy(EntityManager &entityManager, ComponentManager &componentManager,
        int posX, int posY) override;

    Entity createBasicMonster(EntityManager &entityManager, ComponentManager &componentManager,
        int posX, int posY) override;

    Entity createPlayerMissile(EntityManager &entityManager, ComponentManager &componentManager,
        uint32_t entityId) override;

    Entity createForceWeapon(EntityManager &entityManager, ComponentManager &componentManager,
        uint32_t entityId) override;

    Entity createForceMissile(EntityManager &entityManager, ComponentManager &componentManager,
        uint32_t entityId) override;

    Entity createPowerUpBlueLaserCrystal(EntityManager &entityManager,
        ComponentManager &componentManager, int posX, int posY) override;

    Entity createWall(EntityManager &entityManager, ComponentManager &componentManager, int posX,
        int posY) override;

    Entity createButton(EntityManager &entityManager, ComponentManager &componentManager,
        TextureManager &textureManager, FontManager &fontManager, std::string text,
        std::function<IScenes *(AScenes *)> *onClick, float x = 0, float y = 0) override;

    Entity createSmallButton(EntityManager &entityManager, ComponentManager &componentManager,
        TextureManager &textureManager, FontManager &fontManager, std::string text,
        std::function<IScenes *(AScenes *, AScenes::Actions)> *onClick, float x = 0,
        float y = 0) override;

    Entity createUpdateButton(EntityManager &entityManager, ComponentManager &componentManager,
        TextureManager &textureManager, FontManager &fontManager, std::string text,
        std::function<IScenes *(AScenes *)> *onClick,
        std::function<std::string(GameParameters)> *updateText, float x, float y) override;

    Entity createEnemyMissile(EntityManager &entityManager, ComponentManager &componentManager,
```

```
        uint32_t entityId) override;

    Entity createFilter(EntityManager &entityManager, ComponentManager &componentManager,
        AScenes::DaltonismMode mode);

    Entity backgroundFactory(
        EntityManager &entityManager, ComponentManager &componentManager, GameState type);

    Entity createBoss(EntityManager &entityManager, ComponentManager &componentManager,
        EntityFactory &entityFactory);

    Entity createTailSegment(
        EntityManager &entityManager, ComponentManager &componentManager) override;

    Entity createTailEnd(
        EntityManager &entityManager, ComponentManager &componentManager) override;
};
```

7.101 /home/runner/work/R-Type/R-Type/ECS/Interface/Include/↩ Entities/entity_manager.hpp File Reference

```
#include "../error_handling.hpp"
#include "entity.hpp"
#include <algorithm>
#include <memory>
#include <optional>
#include <vector>
```

Classes

- class EntityManager

  Manages the creation, removal, and retrieval of entities.

7.102 entity_manager.hpp

Go to the documentation of this file.

```
/*
** EPITECH PROJECT, 2024
** R-Type
** File description:
** entity_manager
*/

#pragma once

#include "../error_handling.hpp"
#include "entity.hpp"
#include <algorithm>
#include <memory>
#include <optional>
#include <vector>

class EntityManager {
  public:
    Entity createEntity()
    {
        int id = (entityNb += 1);
        entities.emplace_back(id);
        return entities.back();
    }

    void removeEntity(int entityId)
    {
        auto it = std::remove_if(entities.begin(), entities.end(),
            [entityId](const Entity &entity) { return entity.getId() == entityId; });
```

```

        if (it != entities.end()) {
            entities.erase(it, entities.end());
        } else
            throw entityNotFound();
    }

    void removeAllEntities()
    {
        if (entities.empty())
            return;
        std::cout « "Removing all entities" « std::endl;
        entities.clear();
        entityNb = 0;
    }

    std::optional<Entity *> getEntity(int entityId)
    {
        for (auto &entity : entities) {
            if (entity.getId() == entityId) {
                return &entity;
            }
        }
        return std::nullopt;
    }

    const std::vector<Entity> &getAllEntities() const { return entities; }

  private:
    int entityNb = 0;
    std::vector<Entity> entities;
};
```

7.103 /home/runner/work/R-Type/R-Type/ECS/Interface/Include/↩ Entities/i_entity_factory.hpp File Reference

```
#include "Components/component_manager.hpp"
#include "entity.hpp"
#include "entity_manager.hpp"
#include "font_manager.hpp"
#include "texture_manager.hpp"
```

Classes

- class IEntityFactory

  The interface for an entity factory.

7.104 i_entity_factory.hpp

Go to the documentation of this file.

```
/*
** EPITECH PROJECT, 2024
** R-Type
** File description:
** i_entity_factory
*/

#pragma once

#include "Components/component_manager.hpp"
#include "entity.hpp"
#include "entity_manager.hpp"
#include "font_manager.hpp"
#include "texture_manager.hpp"

// Abstract Entity Factory
```

```
class IEntityFactory {
  public:
    virtual ~IEntityFactory() = default;

    virtual Entity createBackgroundLevelOne(
        EntityManager &entityManager, ComponentManager &componentManager) = 0;

    virtual Entity createBackgroundLevelTwo(
        EntityManager &entityManager, ComponentManager &componentManager) = 0;

    virtual Entity createBackgroundLevelThree(
        EntityManager &entityManager, ComponentManager &componentManager) = 0;

    virtual Entity createBackgroundMenu(EntityManager &entityManager,
        ComponentManager &componentManager, TextureManager &textureManager) = 0;

    virtual Entity createInfoBar(
        EntityManager &entityManager, ComponentManager &componentManager) = 0;

    virtual Entity createPlayer(
        EntityManager &entityManager, ComponentManager &componentManager, int nbrOfPlayers) = 0;

    virtual Entity createShooterEnemy(
        EntityManager &entityManager, ComponentManager &componentManager, int posX, int posY) = 0;

    virtual Entity createBasicMonster(
        EntityManager &entityManager, ComponentManager &componentManager, int posX, int posY) = 0;

    virtual Entity createPlayerMissile(
        EntityManager &entityManager, ComponentManager &componentManager, uint32_t entityId) = 0;

    virtual Entity createForceWeapon(
        EntityManager &entityManager, ComponentManager &componentManager, uint32_t entityId) = 0;

    virtual Entity createForceMissile(
        EntityManager &entityManager, ComponentManager &componentManager, uint32_t entityId) = 0;

    virtual Entity createPowerUpBlueLaserCrystal(
        EntityManager &entityManager, ComponentManager &componentManager, int posX, int posY) = 0;

    virtual Entity createWall(
        EntityManager &entityManager, ComponentManager &componentManager, int posX, int posY) = 0;

    virtual Entity createEnemyMissile(
        EntityManager &entityManager, ComponentManager &componentManager, uint32_t entityId) = 0;

    virtual Entity createButton(EntityManager &entityManager, ComponentManager &componentManager,
        TextureManager &textureManager, FontManager &fontManager, std::string text,
        std::function<IScenes *(AScenes *)> *onClick, float x, float y) = 0;

    virtual Entity createSmallButton(EntityManager &entityManager,
        ComponentManager &componentManager, TextureManager &textureManager,
        FontManager &fontManager, std::string text,
        std::function<IScenes *(AScenes *, AScenes::Actions)> *onClick, float x = 0,
        float y = 0) = 0;

    virtual Entity createUpdateButton(EntityManager &entityManager,
        ComponentManager &componentManager, TextureManager &textureManager,
        FontManager &fontManager, std::string text, std::function<IScenes *(AScenes *)> *onClick,
        std::function<std::string(GameParameters)> *updateText, float x, float y) = 0;

    virtual Entity createTailSegment(
        EntityManager &entityManager, ComponentManager &componentManager) = 0;

    virtual Entity createTailEnd(
        EntityManager &entityManager, ComponentManager &componentManager) = 0;

    enum EnemyType
    {
        BasicMonster,
        ShooterEnemy,
        Wall,
        Boss,
    };
};
```

7.105 /home/runner/work/R-Type/R-Type/ECS/Interface/Include/entity_struct.hpp File Reference

```
#include "Components/sprite_data_component.hpp"
#include "Components/text_data_component.hpp"
```

```
#include <cstdint>
#include <macros.hpp>
```

Classes

- struct EntityInformation

  Represents information about an entity.
- struct UIEntityInformation

  Represents the information of a UI entity in the game.

7.106 entity_struct.hpp

Go to the documentation of this file.

```
00001 /*
00002 ** EPITECH PROJECT, 2024
00003 ** R-Type
00004 ** File description:
00005 ** entities_struct
00006 */
00007
00008 #pragma once
00009
00010 #include "Components/sprite_data_component.hpp"
00011 #include "Components/text_data_component.hpp"
00012 #include <cstdint>
00013 #include <macros.hpp>
00014
00022 struct EntityInformation {
00023     uint32_t uniqueID = 0;
00024     vf2d ratio = {0, 0};
00025     SpriteDataComponent spriteData;
00026     vf2d vPos = {0, 0};
00027     AnimationComponent animationComponent = {{0, 0}, {0, 0}};
00028 };
00029
00053 struct UIEntityInformation {
00054     uint32_t uniqueID = 0;
00055     uint32_t lives = 0;
00056     uint32_t score = 0;
00057     SpriteDataComponent spriteData;
00058     TextDataComponent textData;
00059 };
```

7.107 /home/runner/work/R-Type/R-Type/ECS/Interface/Include/error_handling.hpp File Reference

```
#include <exception>
```

Classes

- class componentNotFound

  Exception class for when a component is not found.
- class entityNotFound

  Exception class for entity not found error.
- class failedToLoadTexture

  Exception class for failed texture loading.

- class failedToLoadSound

  Exception class for handling sound loading failures.
- class failedToLoadFont

  Exception class for handling font loading failures.
- class playerIdNotFound

  Exception class for handling cases where a player ID is not found.
- class failedToCreateFile

  Exception class for handling file creation failures.
- class failedToOpenFile

  Exception class for handling file opening failures.

7.108 error_handling.hpp

Go to the documentation of this file.

```
/*
** EPITECH PROJECT, 2024
** R-Type
** File description:
** error_handling
*/

#pragma once

#include <exception>

class componentNotFound : public std::exception {
    const char *what() const noexcept override { return "Component not found"; }
};

class entityNotFound : public std::exception {
    const char *what() const noexcept override { return "Entity not found"; }
};

class failedToLoadTexture : public std::exception {
    const char *what() const noexcept override { return "Failed to load texture"; }
};

class failedToLoadSound : public std::exception {
    const char *what() const noexcept override { return "Failed to load sound"; }
};

class failedToLoadFont : public std::exception {
    const char *what() const noexcept override { return "Failed to load font"; }
};

class playerIdNotFound : public std::exception {
    const char *what() const noexcept override { return "Player ID not found"; }
};

class failedToCreateFile : public std::exception {
    const char *what() const noexcept override { return "Failed to create file"; }
};

class failedToOpenFile : public std::exception {
    const char *what() const noexcept override { return "Failed to open file"; }
};
```

7.109 /home/runner/work/R-Type/R-Type/ECS/Interface/Include/font_manager.hpp File Reference

```
#include "error_handling.hpp"
#include <SFML/Graphics.hpp>
#include <string>
#include <unordered_map>
```

Classes

- class FontManager

  Manages the loading and retrieval of font resources.

7.110 font_manager.hpp

Go to the documentation of this file.

```
/*
** EPITECH PROJECT, 2024
** R-Type
** File description:
** font_manager
*/

#pragma once

#include "error_handling.hpp"
#include <SFML/Graphics.hpp>
#include <string>
#include <unordered_map>

class FontManager {
  public:
    sf::Font &getFont(const std::string &filePath)
    {
        auto it = fonts.find(filePath);
        if (it != fonts.end()) {
            return it->second;
        }

        auto &font = fonts[filePath];
        if (!font.loadFromFile(filePath)) {
            fonts.erase(filePath);
            throw failedToLoadFont();
        }

        return fonts[filePath];
    }

    void releaseFont(const std::string &filePath) { fonts.erase(filePath); }

  private:
    std::unordered_map<std::string, sf::Font> fonts;
};
```

7.111 /home/runner/work/R-Type/R-Type/ECS/Interface/Include/font_path.hpp File Reference

```
#include <cstdint>
#include <string>
```

Enumerations

- enum class FontPath : uint32_t { MAIN , NONE }

  Enumeration of font paths.

Functions

- std::string FontFactory (FontPath font)

  Creates a font object from the given font path.
- std::ostream & operator<< (std::ostream &os, const FontPath &fontPath)

  Overloads the stream insertion operator to output the FontPath object.

7.111.1 Enumeration Type Documentation

7.111.1.1 FontPath

```
enum class FontPath : uint32_t [strong]
```

Enumeration of font paths.

The FontPath enumeration contains a list of font paths that can be used to

specify the location of a font resource. Each font path corresponds to a specific font file that can be loaded and used by the application.

Example usage:

```
FontPath fontPath = FontPath::MAIN;
std::string font = FontFactory(fontPath);
```

See also

FontFactory

operator<<

FontManager

FontPath

Note

The NONE font path is used to indicate that no font should be loaded.

Enumerator

MAIN	
NONE	

7.111.2 Function Documentation

7.111.2.1 FontFactory()

```
std::string FontFactory (
            FontPath font )
```

Creates a font object from the given font path.

This function takes a FontPath object and returns a string representation of the font. The FontPath object should contain the necessary information to locate and load the font.

Parameters

font	The FontPath object containing the path to the font.

Returns

std::string The string representation of the font.

7.111.2.2 operator<<()

```
std::ostream & operator<< (
            std::ostream & os,
            const FontPath & fontPath )
```

Overloads the stream insertion operator to output the FontPath object.

This function allows the FontPath object to be output to an ostream, such as std::cout or any other output stream, by using the << operator.

Parameters

os	The output stream to which the FontPath object will be written.
fontPath	The FontPath object to be written to the output stream.

Returns

A reference to the output stream after the FontPath object has been written.

7.112 font_path.hpp

Go to the documentation of this file.

```
/*
** EPITECH PROJECT, 2024
** R-Type
** File description:
** font_path
*/

#pragma once

#include <cstdint>
#include <string>

enum class FontPath : uint32_t
{
    MAIN,
    NONE
};

std::string FontFactory(FontPath font);

std::ostream &operator«(std::ostream &os, const FontPath &fontPath);
```

7.113 /home/runner/work/R-Type/R-Type/ECS/Interface/Include/game_text.hpp File Reference

```
#include <cstdint>
#include <string>
```

Enumerations

- enum class GameText : uint32_t { Lives , Score , NONE }

  Enumeration for different types of game text.

Functions

- std::string GameTextFactory (GameText text)

  Factory function to convert GameText enum to a string.
- std::ostream & operator<< (std::ostream &os, const GameText &text)

  Overloaded stream insertion operator for GameText.

7.113.1 Enumeration Type Documentation

7.113.1.1 GameText

```
enum class GameText : uint32_t [strong]
```

Enumeration for different types of game text.

This enumeration defines the different types of text that can be displayed in the game.

Enumerator

Lives	Represents the number of lives left.
Score	Represents the player's score.
NONE	Represents no text.

7.113.2 Function Documentation

7.113.2.1 GameTextFactory()

```
std::string GameTextFactory (
            GameText text )
```

Factory function to convert GameText enum to a string.

Parameters

text	The GameText enum value.

Returns

A string representation of the GameText value.

7.113.2.2 operator<<()

```
std::ostream & operator<< (
            std::ostream & os,
            const GameText & text )
```

Overloaded stream insertion operator for GameText.

Parameters

os	The output stream.
text	The GameText enum value.

Returns

The output stream with the GameText value inserted.

7.114 game_text.hpp

Go to the documentation of this file.

```
00001 /*
00002 ** EPITECH PROJECT, 2024
00003 ** R-Type
00004 ** File description:
00005 ** game_text
00006 */
00007
00008 #pragma once
00009
00010 #include <cstdint>
00011 #include <string>
00012
00019 enum class GameText : uint32_t
00020 {
00021     Lives,
00022     Score,
00023     NONE,
00024 };
00025
00032 std::string GameTextFactory(GameText text);
00033
00041 std::ostream &operator«(std::ostream &os, const GameText &text);
```

7.115 /home/runner/work/R-Type/R-Type/ECS/Interface/Include/hitbox_↩ tmp.hpp File Reference

```
#include <Components/component_manager.hpp>
#include <Entities/entity.hpp>
#include <Entities/entity_manager.hpp>
#include <entity_struct.hpp>
```

Functions

- int CheckEntityPosition (uint32_t entityId, ComponentManager componentManager, EntityManager entity↩ Manager)

  Checks the position of an entity within the game world.
- int CheckEntityMovement (EntityInformation desc, ComponentManager componentManager, EntityManager entityManager)

  Checks the movement of an entity within the game.

7.115.1 Function Documentation

7.115.1.1 CheckEntityMovement()

```
int CheckEntityMovement (
            EntityInformation desc,
            ComponentManager componentManager,
            EntityManager entityManager )
```

Checks the movement of an entity within the game.

Parameters

desc	An EntityInformation object containing details about the entity.
componentManager	A ComponentManager object to manage the components of entities.
entityManager	An EntityManager object to manage the entities.

Returns

An integer indicating the result of the movement check.

7.115.1.2 CheckEntityPosition()

```
int CheckEntityPosition (
            uint32_t entityId,
            ComponentManager componentManager,
            EntityManager entityManager )
```

Checks the position of an entity within the game world.

This function retrieves and checks the position of the specified entity using the provided component and entity managers.

Parameters

entityId	The unique identifier of the entity whose position is to be checked.
componentManager	The manager responsible for handling components of entities.
entityManager	The manager responsible for handling entities.

Returns

An integer representing the status or result of the position check.

7.116 hitbox_tmp.hpp

Go to the documentation of this file.

```
00001 /*
00002 ** EPITECH PROJECT, 2024
00003 ** R-Type
```

```
** File description:
** hitbox_tmp
*/

#pragma once
#include <Components/component_manager.hpp>
#include <Entities/entity.hpp>
#include <Entities/entity_manager.hpp>
#include <entity_struct.hpp>

int CheckEntityPosition(
    uint32_t entityId, ComponentManager componentManager, EntityManager entityManager);

int CheckEntityMovement(
    EntityInformation desc, ComponentManager componentManager, EntityManager entityManager);
```

7.117 /home/runner/work/R-Type/R-Type/ECS/Interface/Include/i_scenes.hpp File Reference

```
#include <SFML/Graphics.hpp>
```

Classes

- class IScenes

  Interface for managing different scenes in a game.

7.118 i_scenes.hpp

Go to the documentation of this file.

```
/*
** EPITECH PROJECT, 2024
** R-Type
** File description:
** i_scenes
*/

#pragma once

#include <SFML/Graphics.hpp>

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

    virtual void mainMenu() = 0;

    virtual void gameLoop() = 0;

    virtual void settingsMenu() = 0;

    virtual void inGameMenu() = 0;

    virtual void difficultyChoices() = 0;

    virtual void render() = 0;

    virtual bool shouldQuit() = 0;

    virtual sf::RenderWindow *getRenderWindow() = 0;
};
```

7.119 /home/runner/work/R-Type/R-Type/ECS/Interface/↩ Include/macros.hpp File Reference

Classes

- struct vf2d

  Represents a 2D vector with x and y coordinates.

Macros

- #define SCREEN_WIDTH 1920
- #define SCREEN_HEIGHT 1080

7.119.1 Macro Definition Documentation

7.119.1.1 SCREEN_HEIGHT

```
#define SCREEN_HEIGHT 1080
```

7.119.1.2 SCREEN_WIDTH

```
#define SCREEN_WIDTH 1920
```

7.120 macros.hpp

Go to the documentation of this file.

```
/*
** EPITECH PROJECT, 2024
** R-Type
** File description:
** macros
*/

#pragma once

#define SCREEN_WIDTH 1920
#define SCREEN_HEIGHT 1080

struct vf2d {
    float x = 0;
    float y = 0;
};
```

7.121 /home/runner/work/R-Type/R-Type/ECS/Interface/Include/sound_↩ path.hpp File Reference

```
#include <cstdint>
#include <string>
```

Enumerations

- enum class ActionType : uint32_t {
  Win , Shot , Boss , PowerUp ,
  GameOver , BossDeath , Explosion , Background ,
  NONE }

  This header file defines the ActionType enumeration and declares the SoundFactory function.

Functions

- std::string SoundFactory (ActionType action)

  Generates the file path for a sound based on the given action type.

7.121.1 Enumeration Type Documentation

7.121.1.1 ActionType

```
enum class ActionType : uint32_t [strong]
```

This header file defines the ActionType enumeration and declares the SoundFactory function.

ActionType: An enumeration representing different types of actions that can trigger sounds in the game. The possible values are:

- Win: Represents a winning action.
- Shot: Represents a shooting action.
- Boss: Represents a boss-related action.
- PowerUp: Represents a power-up action.
- GameOver: Represents a game over action.
- BossDeath: Represents a boss death action.
- Explosion: Represents an explosion action.
- Background: Represents background music or sound.
- NONE: Represents no action.

SoundFactory: A function that takes an ActionType as a parameter and returns a string representing the path to the corresponding sound file.

Parameters

action	The ActionType for which the sound path is required.

Returns

A string representing the path to the sound file corresponding to the given action.

Enumerator

Win	
Shot	
Boss	
PowerUp	
GameOver	
BossDeath	
Explosion	
Background	
NONE	

7.121.2 Function Documentation

7.121.2.1 SoundFactory()

```
std::string SoundFactory (
            ActionType action )
```

Generates the file path for a sound based on the given action type.

This function takes an ActionType enumeration value and returns a corresponding file path as a string. The file path points to the sound file associated with the specified action.

Parameters

action	The action type for which the sound file path is needed.

Returns

std::string The file path of the sound associated with the given action.

7.122 sound_path.hpp

Go to the documentation of this file.

```
/*
** EPITECH PROJECT, 2024
** R-Type
** File description:
** sound_path
*/

#pragma once

#include <cstdint>
#include <string>

enum class ActionType : uint32_t
{
    Win,
    Shot,
    Boss,
    PowerUp,
    GameOver,
    BossDeath,
    Explosion,
```

```
    Background,
    NONE,
};

std::string SoundFactory(ActionType action);
```

7.123 /home/runner/work/R-Type/R-Type/ECS/Interface/Include/sprite_path.hpp File Reference

```
#include <cstdint>
#include <string>
```

Enumerations

- enum class SpritePath : uint32_t {
  Ship1 , Ship2 , Ship3 , Ship4 ,
  Enemy1 , Enemy2 , Enemy3 , Missile ,
  ForceWeapon , ForceMissile , BlueLaserCrystal , Background1 ,
  Background2 , Background3 , Boss , BossBullet ,
  Bar , Wall }

  Enum class representing various sprite paths used in the game.

Functions

- std::string SpriteFactory (SpritePath sprite)

  Factory function to get the string representation of a sprite path.
- std::ostream & operator<< (std::ostream &os, const SpritePath &spritePath)

  Overloaded output stream operator for SpritePath.

7.123.1 Enumeration Type Documentation

7.123.1.1 SpritePath

```
enum class SpritePath :  uint32_t  [strong]
```

Enum class representing various sprite paths used in the game.

This enum class defines a set of constants representing different sprite paths that can be used in the game. Each constant corresponds to a specific sprite.

Enumerator

Ship1	Represents the path for the first ship sprite.
Ship2	Represents the path for the second ship sprite.
Ship3	Represents the path for the third ship sprite.
Ship4	Represents the path for the fourth ship sprite.
Enemy1	Represents the path for the first enemy sprite.
Enemy2	Represents the path for the second enemy sprite.
Enemy3	Represents the path for the third enemy sprite.

Enumerator

Missile	Represents the path for the missile sprite.
ForceWeapon	Represents the path for the force weapon sprite.
ForceMissile	Represents the path for the force missile sprite.
BlueLaserCrystal	Represents the path for the blue laser crystal sprite.
Background1	Represents the path for the first background sprite.
Background2	Represents the path for the second background sprite.
Background3	Represents the path for the third background sprite.
Boss	Represents the path for the boss sprite.
BossBullet	Represents the path for the boss bullet sprite.
Bar	Represents the path for the bar sprite.
Wall	Represents the path for the wall sprite.

7.123.2 Function Documentation

7.123.2.1 operator<<()

```
std::ostream & operator<< (
            std::ostream & os,
            const SpritePath & spritePath )
```

Overloaded output stream operator for SpritePath.

This operator allows the SpritePath enum value to be output to an output stream, such as std::cout.

Parameters

os	The output stream.
spritePath	The SpritePath enum value.

Returns

std::ostream& The output stream with the sprite path written to it.

7.123.2.2 SpriteFactory()

```
std::string SpriteFactory (
            SpritePath sprite )
```

Factory function to get the string representation of a sprite path.

This function takes a SpritePath enum value and returns the corresponding string representation of the sprite path.

Parameters

sprite	The SpritePath enum value.

Returns

std::string The string representation of the sprite path.

7.124 sprite_path.hpp

Go to the documentation of this file.

```
/*
** EPITECH PROJECT, 2024
** R-Type
** File description:
** spriteData
*/

#pragma once

#include <cstdint>
#include <string>

enum class SpritePath : uint32_t
{
    Ship1,
    Ship2,
    Ship3,
    Ship4,
    Enemy1,
    Enemy2,
    Enemy3,
    Missile,
    ForceWeapon,
    ForceMissile,
    BlueLaserCrystal,
    Background1,
    Background2,
    Background3,
    // Explosion,
    // PowerUp,
    Boss,
    BossBullet,
    Bar,
    Wall,
};

std::string SpriteFactory(SpritePath sprite);

std::ostream &operator«(std::ostream &os, const SpritePath &spritePath);
```

7.125 /home/runner/work/R-Type/R-Type/ECS/Interface/Include/↩ Systems/audio_system.hpp File Reference

```
#include <SFML/Audio.hpp>
#include <Systems/i_system.hpp>
#include <audio_manager.hpp>
#include <error_handling.hpp>
#include <memory>
#include <string>
```

Classes

- class AudioSystem

  Manages audio playback within the application.

7.126 audio_system.hpp

Go to the documentation of this file.

```
/*
** EPITECH PROJECT, 2024
** R-Type
** File description:
** audio_system
*/

#pragma once

#include <SFML/Audio.hpp>
#include <Systems/i_system.hpp>
#include <audio_manager.hpp>
#include <error_handling.hpp>
#include <memory>
#include <string>

class AudioSystem : public ISystem {
  public:
    AudioSystem(std::shared_ptr<AudioManager> audioManager) : _audioManager(audioManager) {}

    void playBackgroundMusic(const std::string &filePath);

    void stopBackgroundMusic();

    void playSoundEffect(const std::string &filePath);

  private:
    std::shared_ptr<AudioManager> _audioManager;
    sf::Music _backgroundMusic;
    std::string _currentMusicFilePath;
    sf::Sound _soundEffect;
};
```

7.127 /home/runner/work/R-Type/R-Type/ECS/Interface/Include/↩ Systems/auto_fire_system.hpp File Reference

```
#include "Systems/i_system.hpp"
```

Classes

- class AutoFireSystem

  A system that handles automatic firing mechanisms for entities.

7.128 auto_fire_system.hpp

Go to the documentation of this file.

```
/*
** EPITECH PROJECT, 2024
** R-Type
** File description:
** auto_fire_system
*/

#pragma once

#include "Systems/i_system.hpp"

class AutoFireSystem : public ISystem {
  public:
    AutoFireSystem(ComponentManager &componentManager, EntityManager &entityManager)
        : _componentManager(componentManager), _entityManager(entityManager)
    {
```

```
    }

    void handleAutoFire(ComponentManager &componentManager, EntityManager &entityManager);

  private:
    ComponentManager &_componentManager;
    EntityManager &_entityManager;
};
```

7.129 /home/runner/work/R-Type/R-Type/ECS/Interface/Include/Systems/collision_system.hpp File Reference

```
#include "Systems/i_system.hpp"
```

Classes

- class CollisionSystem

  Manages collision detection and response within the ECS framework.

7.130 collision_system.hpp

Go to the documentation of this file.

```
/*
** EPITECH PROJECT, 2024
** R-Type
** File description:
** collision_system
*/

#pragma once

#include "Systems/i_system.hpp"

class CollisionSystem : public ISystem {
  public:
    CollisionSystem(ComponentManager &componentManager, EntityManager &entityManager)
        : _componentManager(componentManager), _entityManager(entityManager)
    {
    }

    bool checkCollision(ComponentManager &componentManager, int entityId1, int entityId2);

    bool checkOffScreen(ComponentManager &componentManager, int entityId);

  private:
    ComponentManager &_componentManager;
    EntityManager &_entityManager;
};
```

7.131 /home/runner/work/R-Type/R-Type/ECS/Interface/Include/Systems/i_system.hpp File Reference

```
#include "Components/component_manager.hpp"
#include "Entities/entity_manager.hpp"
#include <SFML/Graphics.hpp>
```

Classes

- class ISystem

  Interface for all systems in the ECS (Entity Component System) architecture.

7.132 i_system.hpp

Go to the documentation of this file.

```
/*
** EPITECH PROJECT, 2024
** R-Type
** File description:
** i_system
*/

#pragma once

#include "Components/component_manager.hpp"
#include "Entities/entity_manager.hpp"
#include <SFML/Graphics.hpp>

class ISystem {
  public:
    ISystem() = default;
    virtual ~ISystem() = default;
};
```

7.133 /home/runner/work/R-Type/R-Type/ECS/Interface/Include/↩ Systems/move_system.hpp File Reference

```
#include "i_system.hpp"
```

Classes

- class MoveSystem

  System responsible for moving entities within the ECS framework.

7.134 move_system.hpp

Go to the documentation of this file.

```
/*
** EPITECH PROJECT, 2024
** R-Type
** File description:
** move_system
*/

#pragma once

#include "i_system.hpp"

class MoveSystem : public ISystem {
  public:
    MoveSystem(ComponentManager &componentManager, EntityManager &entityManager)
        : _componentManager(componentManager), _entityManager(entityManager){};
    void moveEntities(ComponentManager &componentManager, EntityManager &entityManager);

    void moveEntity(ComponentManager &componentManager, int entityId);

  private:
    ComponentManager &_componentManager;
    EntityManager &_entityManager;
};
```

7.135 /home/runner/work/R-Type/R-Type/ECS/Interface/Include/↩ Systems/render_system.hpp File Reference

```
#include "Systems/i_system.hpp"
#include <error_handling.hpp>
```

Classes

- class RenderSystem

  A system responsible for rendering entities in the ECS framework.

7.136 render_system.hpp

Go to the documentation of this file.

```
00001 /*
00002 ** EPITECH PROJECT, 2024
00003 ** R-Type
00004 ** File description:
00005 ** render_system
00006 */
00007
00008 #pragma once
00009
00010 #include "Systems/i_system.hpp"
00011 #include <error_handling.hpp>
00012
00013 class RenderSystem : public ISystem {
00014   public:
00027     RenderSystem(sf::RenderWindow &window, ComponentManager &componentManager)
00028         : _window(window), _componentManager(componentManager)
00029     {
00030     }
00031
00042     void render(ComponentManager &componentManager);
00043
00044   private:
00048     sf::RenderWindow &_window;
00056     ComponentManager &_componentManager;
00065     sf::Font _font;
00066 };
```

7.137 /home/runner/work/R-Type/R-Type/ECS/Interface/Include/↩ Systems/systems.hpp File Reference

```
#include <Systems/audio_system.hpp>
#include <Systems/auto_fire_system.hpp>
#include <Systems/collision_system.hpp>
#include <Systems/move_system.hpp>
#include <Systems/render_system.hpp>
#include <Systems/update_system.hpp>
```

7.138 systems.hpp

Go to the documentation of this file.

```
00001 /*
00002 ** EPITECH PROJECT, 2024
00003 ** R-Type
00004 ** File description:
00005 ** system
00006 */
00007
00008 #pragma once
00009
00010 #include <Systems/audio_system.hpp>
00011 #include <Systems/auto_fire_system.hpp>
00012 #include <Systems/collision_system.hpp>
00013 #include <Systems/move_system.hpp>
00014 #include <Systems/render_system.hpp>
00015 #include <Systems/update_system.hpp>
```

7.139 /home/runner/work/R-Type/R-Type/ECS/Interface/Include/↩ Systems/update_system.hpp File Reference

```
#include "Systems/i_system.hpp"
```

Classes

- class UpdateSystem

  A system responsible for updating sprite positions in the game.

7.140 update_system.hpp

Go to the documentation of this file.

```
00001 /*
00002 ** EPITECH PROJECT, 2024
00003 ** R-Type
00004 ** File description:
00005 ** update_system
00006 */
00007
00008 #pragma once
00009
00010 #include "Systems/i_system.hpp"
00011
00024 class UpdateSystem : public ISystem {
00025   public:
00033     UpdateSystem(
00034         sf::RenderWindow &window, ComponentManager &componentManager, EntityManager &entityManager)
00035         : _window(window), _componentManager(componentManager), _entityManager(entityManager)
00036     {
00037     }
00038
00049     void updateSpritePositions(ComponentManager &componentManager, EntityManager &entityManager);
00050
00051   private:
00055     sf::RenderWindow &_window;
00062     ComponentManager &_componentManager;
00070     EntityManager &_entityManager;
00071 };
```

7.141 /home/runner/work/R-Type/R-Type/ECS/Interface/Include/texture_manager.hpp File Reference

```
#include "error_handling.hpp"
#include <SFML/Graphics.hpp>
#include <string>
#include <unordered_map>
```

Classes

- class TextureManager

7.142 texture_manager.hpp

Go to the documentation of this file.

```
/*
** EPITECH PROJECT, 2024
** R-Type
** File description:
** texture_manager
*/

#pragma once

#include "error_handling.hpp"
#include <SFML/Graphics.hpp>
#include <string>
#include <unordered_map>

class TextureManager {
  public:
    sf::Texture &getTexture(const std::string &filePath)
    {
        auto it = textures.find(filePath);
        if (it != textures.end()) {
            return it->second;
        }

        auto &texture = textures[filePath];
        if (!texture.loadFromFile(filePath)) {
            textures.erase(filePath);
            throw failedToLoadTexture();
        }

        return textures[filePath];
    }

    void releaseTexture(const std::string &filePath) { textures.erase(filePath); }

  private:
    std::unordered_map<std::string, sf::Texture> textures;
};
```

7.143 /home/runner/work/R-Type/R-Type/ECS/Src/a_scenes.cpp File Reference

```
#include <a_scenes.hpp>
```

7.144 /home/runner/work/R-Type/R-Type/ECS/Src/Entities/entity_factory.cpp File Reference

```
#include "hitbox_tmp.hpp"
#include <Components/components.hpp>
#include <Entities/entity_factory.hpp>
#include <SFML/Graphics.hpp>
#include <cstdint>
#include <cstdlib>
#include <macros.hpp>
```

Functions

- std::ostream & operator<< (std::ostream &os, const SpritePath &spritePath)

  Overloaded output stream operator for SpritePath.
- std::ostream & operator<< (std::ostream &os, const AScenes::SpriteType &spriteType)
- std::ostream & operator<< (std::ostream &os, const GameState &gameState)
- std::ostream & operator<< (std::ostream &os, const SpriteDataComponent &spriteData)

  Overloads the << operator to output the contents of a SpriteDataComponent to an ostream.

7.144.1 Function Documentation

7.144.1.1 operator<<() [1/4]

```
std::ostream & operator<< (
            std::ostream & os,
            const AScenes::SpriteType & spriteType )
```

7.144.1.2 operator<<() [2/4]

```
std::ostream & operator<< (
            std::ostream & os,
            const GameState & gameState )
```

7.144.1.3 operator<<() [3/4]

```
std::ostream & operator<< (
            std::ostream & os,
            const SpriteDataComponent & spriteData )
```

Overloads the << operator to output the contents of a SpriteDataComponent to an ostream.

Parameters

os	The output stream to which the SpriteDataComponent will be written.
spriteData	The SpriteDataComponent instance to be written to the output stream.

Returns

std::ostream& The output stream after writing the SpriteDataComponent.

7.144.1.4 operator<<() [4/4]

```
std::ostream & operator<< (
            std::ostream & os,
            const SpritePath & spritePath )
```

Overloaded output stream operator for SpritePath.

This operator allows the SpritePath enum value to be output to an output stream, such as std::cout.

Parameters

os	The output stream.
spritePath	The SpritePath enum value.

Returns

std::ostream& The output stream with the sprite path written to it.

7.145 /home/runner/work/R-Type/R-Type/ECS/Src/font_path.cpp File Reference

```
#include <font_path.hpp>
```

Functions

- std::string FontFactory (FontPath font)

  Creates a font object from the given font path.

7.145.1 Function Documentation

7.145.1.1 FontFactory()

```
std::string FontFactory (
            FontPath font )
```

Creates a font object from the given font path.

This function takes a FontPath object and returns a string representation of the font. The FontPath object should contain the necessary information to locate and load the font.

Parameters

font	The FontPath object containing the path to the font.

Returns

std::string The string representation of the font.

7.146 /home/runner/work/R-Type/R-Type/ECS/Src/game_text.cpp File Reference

```
#include <game_text.hpp>
```

Functions

- std::string GameTextFactory (GameText text)

  Factory function to convert GameText enum to a string.

7.146.1 Function Documentation

7.146.1.1 GameTextFactory()

```
std::string GameTextFactory (
            GameText text )
```

Factory function to convert GameText enum to a string.

Parameters

text	The GameText enum value.

Returns

A string representation of the GameText value.

7.147 /home/runner/work/R-Type/R-Type/ECS/Src/hitbox_tmp.cpp File Reference

```
#include "hitbox_tmp.hpp"
#include <macros.hpp>
```

Functions

- static int CheckCollisionLogic (float descLeft, float descRight, float descTop, float descBottom, ComponentManager componentManager, EntityManager entityManager, int entityId)
- int CheckEntityPosition (uint32_t entityId, ComponentManager componentManager, EntityManager entity↩ Manager)

  Checks the position of an entity within the game world.
- int CheckEntityMovement (EntityInformation desc, ComponentManager componentManager, EntityManager entityManager)

  Checks the movement of an entity within the game.

7.147.1 Function Documentation

7.147.1.1 CheckCollisionLogic()

```
static int CheckCollisionLogic (
            float descLeft,
            float descRight,
            float descTop,
            float descBottom,
            ComponentManager componentManager,
            EntityManager entityManager,
            int entityId )  [static]
```

7.147.1.2 CheckEntityMovement()

```
int CheckEntityMovement (
            EntityInformation desc,
            ComponentManager componentManager,
            EntityManager entityManager )
```

Checks the movement of an entity within the game.

Parameters

desc	An EntityInformation object containing details about the entity.
componentManager	A ComponentManager object to manage the components of entities.
entityManager	An EntityManager object to manage the entities.

Returns

An integer indicating the result of the movement check.

7.147.1.3 CheckEntityPosition()

```
int CheckEntityPosition (
            uint32_t entityId,
            ComponentManager componentManager,
            EntityManager entityManager )
```

Checks the position of an entity within the game world.

This function retrieves and checks the position of the specified entity using the provided component and entity managers.

Parameters

entityId	The unique identifier of the entity whose position is to be checked.
componentManager	The manager responsible for handling components of entities.
entityManager	The manager responsible for handling entities.

Returns

An integer representing the status or result of the position check.

7.148 /home/runner/work/R-Type/R-Type/ECS/Src/sound_path.cpp File Reference

```
#include <sound_path.hpp>
```

Functions

- std::string SoundFactory (ActionType action)

  Generates the file path for a sound based on the given action type.

7.148.1 Function Documentation

7.148.1.1 SoundFactory()

```
std::string SoundFactory (
            ActionType action )
```

Generates the file path for a sound based on the given action type.

This function takes an ActionType enumeration value and returns a corresponding file path as a string. The file path points to the sound file associated with the specified action.

Parameters

action	The action type for which the sound file path is needed.

Returns

std::string The file path of the sound associated with the given action.

7.149 /home/runner/work/R-Type/R-Type/ECS/Src/sprite_path.cpp File Reference

```
#include <sprite_path.hpp>
```

Functions

- std::string SpriteFactory (SpritePath sprite)

  Factory function to get the string representation of a sprite path.

7.149.1 Function Documentation

7.149.1.1 SpriteFactory()

```
std::string SpriteFactory (
            SpritePath sprite )
```

Factory function to get the string representation of a sprite path.

This function takes a SpritePath enum value and returns the corresponding string representation of the sprite path.

Parameters

sprite	The SpritePath enum value.

Returns

std::string The string representation of the sprite path.

7.150 /home/runner/work/R-Type/R-Type/ECS/Src/Systems/audio_system.cpp File Reference

```
#include <Systems/audio_system.hpp>
```

7.151 /home/runner/work/R-Type/R-Type/ECS/Src/Systems/auto_fire_system.cpp File Reference

```
#include <Systems/auto_fire_system.hpp>
```

7.152 /home/runner/work/R-Type/R-Type/ECS/Src/Systems/collision_system.cpp File Reference

```
#include <Systems/collision_system.hpp>
#include <macros.hpp>
#include <vector>
```

7.153 /home/runner/work/R-Type/R-Type/ECS/Src/Systems/move_system.cpp File Reference

```
#include <Systems/move_system.hpp>
#include <cmath>
```

7.154 /home/runner/work/R-Type/R-Type/ECS/Src/Systems/render_system.cpp File Reference

```
#include <Systems/render_system.hpp>
```

7.155 /home/runner/work/R-Type/R-Type/ECS/Src/Systems/update_system.cpp File Reference

```
#include "Systems/update_system.hpp"
```

7.156 /home/runner/work/R-Type/R-Type/Server/Interface/Include/animation_system.hpp File Reference

```
#include "Systems/i_system.hpp"
#include <entity_struct.hpp>
```

Classes

- class AnimationSystem

  A system responsible for animating entities within the ECS framework.

Enumerations

- enum class AnimationShip : uint32_t { SHIP_DOWN , SHIP_FLIP_DOWN , SHIP_STRAIT , SHIP_FLIP_UP , SHIP_UP }
- enum class AnimationBasicMonster : uint32_t { BASIC_MONSTER_DEFAULT , BASIC_MONSTER_1 , BASIC_MONSTER_2 , BASIC_MONSTER_3 , BASIC_MONSTER_4 , BASIC_MONSTER_5 , BASIC_MONSTER_6 , BASIC_MONSTER_7 }
- enum class AnimationForceWeapon1 : uint32_t { FORCE_WEAPON_DEFAULT , FORCE_WEAPON_1 , FORCE_WEAPON_2 , FORCE_WEAPON_3 , FORCE_WEAPON_4 , FORCE_WEAPON_5 }
- enum class AnimationForceWeapon2 : uint32_t { FORCE_WEAPON_DEFAULT , FORCE_WEAPON_1 , FORCE_WEAPON_2 , FORCE_WEAPON_3 , FORCE_WEAPON_4 , FORCE_WEAPON_5 }
- enum class AnimationForceWeapon3 : uint32_t { FORCE_WEAPON_DEFAULT , FORCE_WEAPON_1 , FORCE_WEAPON_2 , FORCE_WEAPON_3 }
- enum class AnimationForceMissile1 : uint32_t { FORCE_MISSILE_DEFAULT }
- enum class AnimationForceMissile2 : uint32_t { FORCE_MISSILE_DEFAULT }
- enum class AnimationForceMissile3 : uint32_t { FORCE_MISSILE_DEFAULT , FORCE_MISSILE_1 , FORCE_MISSILE_2 , FORCE_MISSILE_3 , FORCE_MISSILE_4 , FORCE_MISSILE_5 , FORCE_MISSILE_6 , FORCE_MISSILE_7 }
- enum class AnimationBoss : uint32_t { BOSS_DEFAULT , BOSS_1 , BOSS_2 , BOSS_3 }

Functions

- bool operator!= (AnimationComponent animation, AnimationComponent other)

  get if two animations are different.

7.156.1 Enumeration Type Documentation

7.156.1.1 AnimationBasicMonster

```
enum class AnimationBasicMonster : uint32_t [strong]
```

Enumerator

BASIC_MONSTER_DEFAULT	
BASIC_MONSTER_1	
BASIC_MONSTER_2	
BASIC_MONSTER_3	
BASIC_MONSTER_4	
BASIC_MONSTER_5	
BASIC_MONSTER_6	
BASIC_MONSTER_7	

7.156.1.2 AnimationBoss

```
enum class AnimationBoss : uint32_t [strong]
```

Enumerator

BOSS_DEFAULT	
BOSS_1	
BOSS_2	
BOSS_3	

7.156.1.3 AnimationForceMissile1

```
enum class AnimationForceMissile1 : uint32_t [strong]
```

Enumerator

FORCE_MISSILE_DEFAULT	

7.156.1.4 AnimationForceMissile2

```
enum class AnimationForceMissile2 : uint32_t [strong]
```

Enumerator

FORCE_MISSILE_DEFAULT	

7.156.1.5 AnimationForceMissile3

```
enum class AnimationForceMissile3 : uint32_t [strong]
```

Enumerator

FORCE_MISSILE_DEFAULT	
FORCE_MISSILE_1	
FORCE_MISSILE_2	
FORCE_MISSILE_3	
FORCE_MISSILE_4	
FORCE_MISSILE_5	
FORCE_MISSILE_6	
FORCE_MISSILE_7	

7.156.1.6 AnimationForceWeapon1

```
enum class AnimationForceWeapon1 : uint32_t [strong]
```

Enumerator

FORCE_WEAPON_DEFAULT	

Enumerator

FORCE_WEAPON_1	
FORCE_WEAPON_2	
FORCE_WEAPON_3	
FORCE_WEAPON_4	
FORCE_WEAPON_5	

7.156.1.7 AnimationForceWeapon2

```
enum class AnimationForceWeapon2 : uint32_t [strong]
```

Enumerator

FORCE_WEAPON_DEFAULT	
FORCE_WEAPON_1	
FORCE_WEAPON_2	
FORCE_WEAPON_3	
FORCE_WEAPON_4	
FORCE_WEAPON_5	

7.156.1.8 AnimationForceWeapon3

```
enum class AnimationForceWeapon3 : uint32_t [strong]
```

Enumerator

FORCE_WEAPON_DEFAULT	
FORCE_WEAPON_1	
FORCE_WEAPON_2	
FORCE_WEAPON_3	

7.156.1.9 AnimationShip

```
enum class AnimationShip : uint32_t [strong]
```

Enumerator

SHIP_DOWN	Ship animation when going down.
SHIP_FLIP_DOWN	Ship animation when flipping down.
SHIP_STRAIT	Ship animation when going strait.
SHIP_FLIP_UP	Ship animation when flipping up.
SHIP_UP	Ship animation when going up.

7.156.2 Function Documentation

7.156.2.1 operator"!=()

```
bool operator!= (
            AnimationComponent animation,
            AnimationComponent other )
```

get if two animations are different.

Parameters

animation	The first animation.
other	The second animation.

Returns

bool true if the animations are different, false otherwise.

get if two animations are different.

This operator compares two AnimationComponent objects to determine if they are not equal. Two AnimationComponent objects are considered not equal if any of their respective offset or dimension coordinates differ.

Parameters

animation	The first AnimationComponent to compare.
other	The second AnimationComponent to compare.

Returns

true if the AnimationComponent objects are not equal, false otherwise.

7.157 animation_system.hpp

Go to the documentation of this file.

```
/*
** EPITECH PROJECT, 2024
** R-Type
** File description:
** move_system
*/

#pragma once

#include "Systems/i_system.hpp"
#include <entity_struct.hpp>

enum class AnimationShip : uint32_t
{
    SHIP_DOWN,
    SHIP_FLIP_DOWN,
    SHIP_STRAIT,
    SHIP_FLIP_UP,
    SHIP_UP
};

```

```
enum class AnimationBasicMonster : uint32_t
{
    BASIC_MONSTER_DEFAULT,
    BASIC_MONSTER_1,
    BASIC_MONSTER_2,
    BASIC_MONSTER_3,
    BASIC_MONSTER_4,
    BASIC_MONSTER_5,
    BASIC_MONSTER_6,
    BASIC_MONSTER_7
};

enum class AnimationForceWeapon1 : uint32_t
{
    FORCE_WEAPON_DEFAULT,
    FORCE_WEAPON_1,
    FORCE_WEAPON_2,
    FORCE_WEAPON_3,
    FORCE_WEAPON_4,
    FORCE_WEAPON_5
};

enum class AnimationForceWeapon2 : uint32_t
{
    FORCE_WEAPON_DEFAULT,
    FORCE_WEAPON_1,
    FORCE_WEAPON_2,
    FORCE_WEAPON_3,
    FORCE_WEAPON_4,
    FORCE_WEAPON_5
};

enum class AnimationForceWeapon3 : uint32_t
{
    FORCE_WEAPON_DEFAULT,
    FORCE_WEAPON_1,
    FORCE_WEAPON_2,
    FORCE_WEAPON_3
};

enum class AnimationForceMissile1 : uint32_t
{
    FORCE_MISSILE_DEFAULT
};

enum class AnimationForceMissile2 : uint32_t
{
    FORCE_MISSILE_DEFAULT
};

enum class AnimationForceMissile3 : uint32_t
{
    FORCE_MISSILE_DEFAULT,
    FORCE_MISSILE_1,
    FORCE_MISSILE_2,
    FORCE_MISSILE_3,
    FORCE_MISSILE_4,
    FORCE_MISSILE_5,
    FORCE_MISSILE_6,
    FORCE_MISSILE_7
};

enum class AnimationBoss : uint32_t
{
    BOSS_DEFAULT,
    BOSS_1,
    BOSS_2,
    BOSS_3
};

bool operator!=(AnimationComponent animation, AnimationComponent other);

class AnimationSystem : public ISystem {
  public:
    AnimationSystem(ComponentManager &componentManager, EntityManager &entityManager)
        : _componentManager(componentManager), _entityManager(entityManager){};

    void AnimationEntities(ComponentManager &componentManager, EntityManager &entityManager,
        float deltaTime, bool &endOfLevel);

    void animatePlayer(std::optional<VelocityComponent *> &velocity,
        std::optional<AnimationComponent *> &animation);

    void animateBasicMonster(std::optional<AnimationComponent *> &animation);

    void animateForceWeapon(std::optional<ForceWeaponComponent *> &forceWeapon,
        std::optional<AnimationComponent *> &animation, std::optional<HitboxComponent *> &hitbox);
```

```

    void animateForceMissile(std::optional<ForceWeaponComponent *> &forceWeapon,
        std::optional<AnimationComponent *> &animation, std::optional<HitboxComponent *> &hitbox);

    void animateBoss(
        std::optional<BossComponent *> &boss, std::optional<AnimationComponent *> &animation);

  private:
    ComponentManager &_componentManager;
    EntityManager &_entityManager;
};
```

7.158 /home/runner/work/R-Type/R-Type/Server/Interface/↩ Include/level.hpp File Reference

```
#include <Components/component_manager.hpp>
#include <Components/components.hpp>
#include <animation_system.hpp>
#include <cmath>
#include <game_struct.hpp>
#include <i_level.hpp>
```

Classes

- class r_type::Level< T >

  The Level class template manages the game level, including updating game state, handling collisions, and managing entities.

Namespaces

- namespace r_type
- namespace r_type::net

7.159 level.hpp

Go to the documentation of this file.

```
/*
** EPITECH PROJECT, 2024
** R-Type
** File description:
** level
*/

#pragma once

#include <Components/component_manager.hpp>
#include <Components/components.hpp>
#include <animation_system.hpp>
#include <cmath>
#include <game_struct.hpp>

#include <i_level.hpp>

namespace r_type {
namespace net {
template <typename T> class AServer;
}
template <typename T> class Level : virtual public ILevel<T> {
  public:
    Level() = default;
    ~Level() = default;
```

```

    void Update(r_type::net::AServer<T> *server, ComponentManager &componentManager,
        EntityManager &entityManager, std::chrono::system_clock::time_point newClock,
        bool *bUpdateEntities) override
    {
        if (server->_watingPlayersReady)
            return;
        while (std::chrono::duration_cast<std::chrono::milliseconds>(newClock - server->GetClock())
                    .count() > 100) {
            *bUpdateEntities = true;

            MoveUpdate(server, componentManager, entityManager, newClock);
            CollisionUpdate(server, componentManager, entityManager, newClock);
            AnimationUpdate(server, componentManager, entityManager, newClock);
            FireUpdate(server, componentManager, entityManager, newClock);
            if (server->_endOfLevel == false) {
                switch (_gameParameters.levelType) {
                case GameState::LevelOne:
                    LevelOne(server, componentManager, entityManager, newClock);
                    break;
                case GameState::LevelTwo:
                    LevelTwo(server, componentManager, entityManager, newClock);
                    break;
                case GameState::LevelThree:
                    LevelThree(server, componentManager, entityManager, newClock);
                    break;
                case GameState::Win:
                    EndOfGame(server, componentManager, entityManager);
                    break;
                default:
                    break;
                }
            } else {
                if (server->_bossActive == false) {
                    SpawnEntity(server, entityManager, componentManager, 0,
                        EntityFactory::EnemyType::Boss);
                }
            }
            server->SetClock(server->GetClock() + std::chrono::milliseconds(500));
        }
    }

    void SetSystem(ComponentManager &componentManager, EntityManager &entityManager) override
    {
        _moveSystem = std::make_shared<MoveSystem>(componentManager, entityManager);
        _collisionSystem = std::make_shared<CollisionSystem>(componentManager, entityManager);
        _animationSystem = std::make_shared<AnimationSystem>(componentManager, entityManager);
        _autoFireSystem = std::make_shared<AutoFireSystem>(componentManager, entityManager);
    }

    void MoveUpdate(r_type::net::AServer<T> *server, ComponentManager &componentManager,
        EntityManager &entityManager, std::chrono::system_clock::time_point newClock) override
    {
        if (auto positionsBefore = componentManager.getComponentMap<PositionComponent>()) {
            std::unordered_map<int, PositionComponent> previousPositions;
            // Save previous positions
            for (auto &pair : **positionsBefore) {
                int entityId = pair.first;
                auto positionComponent = pair.second;
                auto position = std::any_cast<PositionComponent>(&positionComponent);
                if (position) {
                    previousPositions.insert({entityId, *position});
                }
            }
            // Move entities
            _moveSystem.get()->moveEntities(componentManager, entityManager);
            // Compare new positions
            if (auto positionsAfter = componentManager.getComponentMap<PositionComponent>()) {
                for (auto &pair : **positionsAfter) {
                    int entityId = pair.first;
                    auto &newPositionComponent = pair.second;
                    if (auto newPosition =
                            std::any_cast<PositionComponent>(&newPositionComponent)) {
                        auto it = previousPositions.find(entityId);
                        if (it != previousPositions.end()) {
                            auto &oldPosition = it->second;
                            if (oldPosition.x != newPosition->x ||
                                oldPosition.y != newPosition->y) {
                                if (auto spriteData =
                                        componentManager.getComponent<SpriteDataComponent>(
                                            entityId)) {
                                    r_type::net::Message<TypeMessage> msg;
                                    vf2d newPos(newPosition->x, newPosition->y);
                                    msg.header.id = TypeMessage::MoveEntityMessage;
                                    msg « entityId « newPos;
                                    server->MessageAllClients(msg);
                                }
```

```
                            }
                        }
                    }
                }
            }
        }
    }

    static bool collisionAction(r_type::net::AServer<T> *server,
        ComponentManager &componentManager, EntityManager &entityManager,
        std::vector<int> &entitiesToRemove, std::vector<int> &entitiesToAdd, uint32_t entityId1,
        uint32_t entityId2)
    {
        // component of entity 1
        auto player1 = componentManager.getComponent<PlayerComponent>(entityId1);
        auto playerMissile1 = componentManager.getComponent<PlayerMissileComponent>(entityId1);
        auto enemyMissile1 = componentManager.getComponent<EnemyMissileComponent>(entityId1);
        auto playerHealth1 = componentManager.getComponent<HealthComponent>(entityId1);
        auto powerUp1 = componentManager.getComponent<PowerUpComponent>(entityId1);
        auto enemy1 = componentManager.getComponent<EnemyComponent>(entityId1);
        auto forceWeapon1 = componentManager.getComponent<ForceWeaponComponent>(entityId1);
        auto forceMissile1 = componentManager.getComponent<ForceMissileComponent>(entityId1);
        auto boss1 = componentManager.getComponent<BossComponent>(entityId1);
        auto tail1 = componentManager.getComponent<TailComponent>(entityId1);

        // component of entity 2
        auto enemyMissile2 = componentManager.getComponent<EnemyMissileComponent>(entityId2);
        auto player2 = componentManager.getComponent<PlayerComponent>(entityId2);
        auto playerHealth2 = componentManager.getComponent<HealthComponent>(entityId2);
        auto playerMissile2 = componentManager.getComponent<PlayerMissileComponent>(entityId2);
        auto powerUp2 = componentManager.getComponent<PowerUpComponent>(entityId2);
        auto enemy2 = componentManager.getComponent<EnemyComponent>(entityId2);
        auto forceWeapon2 = componentManager.getComponent<ForceWeaponComponent>(entityId2);
        auto forceMissile2 = componentManager.getComponent<ForceMissileComponent>(entityId2);
        auto boss2 = componentManager.getComponent<BossComponent>(entityId2);
        auto tail2 = componentManager.getComponent<TailComponent>(entityId2);

        // Handle collision
        if (player1) {
            if (playerHealth1) {
                if (enemy2 || enemyMissile2) {
                    if (std::find(entitiesToRemove.begin(), entitiesToRemove.end(), entityId2) ==
                        entitiesToRemove.end()) {
                        entitiesToRemove.push_back(entityId2);
                        playerHealth1.value()->lives -= 1;
                    }
                }
                if (boss2) {
                    playerHealth1.value()->lives -= 1;
                    if (auto bossHealth =
                            componentManager.getComponent<HealthComponent>(entityId2)) {
                        bossHealth.value()->lives -= 1;
                        if (bossHealth.value()->lives <= 0) {
                            if (std::find(entitiesToRemove.begin(), entitiesToRemove.end(),
                                    entityId2) == entitiesToRemove.end()) {
                                entitiesToRemove.push_back(entityId2);
                            }
                        }
                    }
                }
                if (tail2) {
                    playerHealth1.value()->lives -= 1;
                }
                r_type::net::Message<TypeMessage> updLivesMsg;
                updLivesMsg.header.id = TypeMessage::UpdateInfoBar;
                updLivesMsg « server->UpdateInfoBar(entityId1);
                server->MessageClient(server->getClientById(server->_deqConnections,
                                        server->GetPlayerClientId(entityId1)),
                    updLivesMsg);
                if (playerHealth1.value()->lives <= 0) {
                    if (std::find(entitiesToRemove.begin(), entitiesToRemove.end(), entityId1) ==
                        entitiesToRemove.end()) {
                        entitiesToRemove.push_back(entityId1);
                    }
                }
            }
            // when player collides with power up
            if (powerUp2) {
                if (std::find(entitiesToRemove.begin(), entitiesToRemove.end(), entityId2) ==
                    entitiesToRemove.end()) {
                    entitiesToRemove.push_back(entityId2);
                }
                auto linkForceComponent =
                    componentManager.getComponent<LinkForceComponent>(entityId1);
                if (linkForceComponent.value()->targetId != -1) {
                    auto forceWeapon = componentManager.getComponent<ForceWeaponComponent>(
                        linkForceComponent.value()->targetId);
```

```
                    if (forceWeapon) {
                        if (forceWeapon.value()->level < 3) {
                            forceWeapon.value()->level += 1;
                        }
                    }
                } else {
                    Entity weapon = server->GetEntityFactory().createForceWeapon(
                        entityManager, componentManager, entityId1);
                    entitiesToAdd.push_back(weapon.getId());
                    auto linkForceComponent =
                        componentManager.getComponent<LinkForceComponent>(entityId1);
                    linkForceComponent.value()->targetId = weapon.getId();
                }
            }
            if (forceWeapon2) {
                auto frontComponent = componentManager.getComponent<FrontComponent>(entityId1);
                if (frontComponent) {
                    frontComponent.value()->targetId = entityId2;
                    auto forceWeapon = componentManager.getComponent<ForceWeaponComponent>(
                        frontComponent.value()->targetId);
                    if (forceWeapon) {
                        forceWeapon.value()->attached = true;
                    }
                    auto forceWeaponMovementComponent =
                        componentManager.getComponent<MovementComponent>(
                            frontComponent.value()->targetId);
                    if (forceWeaponMovementComponent) {
                        forceWeaponMovementComponent.value()->move = false;
                    }
                }
            }
            return true;
        } else if (playerMissile1) {
            if (enemy2 || enemyMissile2) {
                if (std::find(entitiesToRemove.begin(), entitiesToRemove.end(), entityId1) ==
                    entitiesToRemove.end()) {
                    entitiesToRemove.push_back(entityId1);
                }
                if (std::find(entitiesToRemove.begin(), entitiesToRemove.end(), entityId2) ==
                    entitiesToRemove.end()) {
                    if (rand() % 10 == 0) {
                        auto posEnemy =
                            componentManager.getComponent<PositionComponent>(entityId2);
                        if (posEnemy) {
                            Entity weapon =
                                server->GetEntityFactory().createPowerUpBlueLaserCrystal(
                                    entityManager, componentManager, posEnemy.value()->x,
                                    posEnemy.value()->y);
                            entitiesToAdd.push_back(weapon.getId());
                        }
                    }
                    entitiesToRemove.push_back(entityId2);
                }
                int playerId = playerMissile1.value()->playerId;
                if (auto playerScore = componentManager.getComponent<ScoreComponent>(playerId)) {
                    playerScore.value()->score += 100;
                }
                r_type::net::Message<TypeMessage> updScoreMsg;
                updScoreMsg.header.id = TypeMessage::UpdateInfoBar;
                updScoreMsg « server->UpdateInfoBar(playerId);
                server->MessageClient(server->getClientById(server->_deqConnections,
                                      server->GetPlayerClientId(playerId)),
                    updScoreMsg);
            }
            if (boss2) {
                if (std::find(entitiesToRemove.begin(), entitiesToRemove.end(), entityId1) ==
                    entitiesToRemove.end()) {
                    entitiesToRemove.push_back(entityId1);
                }
                if (auto bossHealth = componentManager.getComponent<HealthComponent>(entityId2)) {
                    bossHealth.value()->lives -= 1;
                    if (bossHealth.value()->lives <= 0) {
                        if (std::find(entitiesToRemove.begin(), entitiesToRemove.end(),
                                entityId2) == entitiesToRemove.end()) {
                            entitiesToRemove.push_back(entityId2);
                        }
                    }
                }
                int playerId = playerMissile1.value()->playerId;
                if (auto playerScore = componentManager.getComponent<ScoreComponent>(playerId)) {
                    playerScore.value()->score += 200;
                }
                r_type::net::Message<TypeMessage> updScoreMsg;
                updScoreMsg.header.id = TypeMessage::UpdateInfoBar;
                updScoreMsg « server->UpdateInfoBar(playerId);
                server->MessageClient(server->getClientById(server->_deqConnections,
                                      server->GetPlayerClientId(playerId)),
```

```
                    updScoreMsg);
            }
            if (tail2) {
                if (std::find(entitiesToRemove.begin(), entitiesToRemove.end(), entityId1) ==
                    entitiesToRemove.end()) {
                    entitiesToRemove.push_back(entityId1);
                }
            }
            return true;
        } else if (forceMissile1) {
            if (enemy2 || enemyMissile2) {
                if (std::find(entitiesToRemove.begin(), entitiesToRemove.end(), entityId1) ==
                    entitiesToRemove.end()) {
                    entitiesToRemove.push_back(entityId1);
                }
                if (std::find(entitiesToRemove.begin(), entitiesToRemove.end(), entityId2) ==
                    entitiesToRemove.end()) {
                    auto posEnemy = componentManager.getComponent<PositionComponent>(entityId2);
                    if (rand() % 10 == 0) {
                        auto posEnemy =
                            componentManager.getComponent<PositionComponent>(entityId2);
                        if (posEnemy) {
                            Entity weapon =
                                server->GetEntityFactory().createPowerUpBlueLaserCrystal(
                                    entityManager, componentManager, posEnemy.value()->x,
                                    posEnemy.value()->y);
                            entitiesToAdd.push_back(weapon.getId());
                        }
                    }
                    entitiesToRemove.push_back(entityId2);
                }
            }
            auto weapon = componentManager.getComponent<ForceWeaponComponent>(
                forceMissile1.value()->forceId);
            if (weapon) {
                int playerId = weapon.value()->playerId;
                if (auto playerScore = componentManager.getComponent<ScoreComponent>(playerId)) {
                    if (enemy2 || enemyMissile2) {
                        if (std::find(entitiesToRemove.begin(), entitiesToRemove.end(),
                                entityId1) == entitiesToRemove.end()) {
                            entitiesToRemove.push_back(entityId1);
                        }
                        if (std::find(entitiesToRemove.begin(), entitiesToRemove.end(),
                                entityId2) == entitiesToRemove.end()) {
                            auto posEnemy =
                                componentManager.getComponent<PositionComponent>(entityId2);
                            if (rand() % 10 == 0) {
                                auto posEnemy =
                                    componentManager.getComponent<PositionComponent>(entityId2);
                                if (posEnemy) {
                                    Entity weapon =
                                        server->GetEntityFactory().createPowerUpBlueLaserCrystal(
                                            entityManager, componentManager, posEnemy.value()->x,
                                            posEnemy.value()->y);
                                    entitiesToAdd.push_back(weapon.getId());
                                }
                            }
                            entitiesToRemove.push_back(entityId2);
                        }
                        playerScore.value()->score += 100;
                    }
                    if (boss2) {
                        if (std::find(entitiesToRemove.begin(), entitiesToRemove.end(),
                                entityId1) == entitiesToRemove.end()) {
                            entitiesToRemove.push_back(entityId1);
                        }
                        if (auto bossHealth =
                                componentManager.getComponent<HealthComponent>(entityId2)) {
                            bossHealth.value()->lives -= 2;
                            if (bossHealth.value()->lives <= 0) {
                                if (std::find(entitiesToRemove.begin(), entitiesToRemove.end(),
                                        entityId2) == entitiesToRemove.end()) {
                                    entitiesToRemove.push_back(entityId2);
                                }
                            }
                        }
                        playerScore.value()->score += 200;
                    }

                    if (tail2) {
                        if (std::find(entitiesToRemove.begin(), entitiesToRemove.end(),
                                entityId1) == entitiesToRemove.end()) {
                            entitiesToRemove.push_back(entityId1);
                        }
                    }
                }
                r_type::net::Message<TypeMessage> updScoreMsg;
```

```
                updScoreMsg.header.id = TypeMessage::UpdateInfoBar;
                updScoreMsg « server->UpdateInfoBar(playerId);
                server->MessageClient(server->getClientById(server->_deqConnections,
                                        server->GetPlayerClientId(playerId)),
                    updScoreMsg);
            }
            return true;
        } else if (forceWeapon1) {
            if (enemyMissile2) {
                if (std::find(entitiesToRemove.begin(), entitiesToRemove.end(), entityId2) ==
                    entitiesToRemove.end()) {
                    entitiesToRemove.push_back(entityId2);
                }
                std::cout « "force weapon collided with enemy missile" « std::endl;
            }
            return true;
        }
        return false;
    }

    void CollisionUpdate(r_type::net::AServer<T> *server, ComponentManager &componentManager,
        EntityManager &entityManager, std::chrono::system_clock::time_point newClock) override
    {
        std::vector<int> entitiesToRemove;
        std::vector<int> entitiesToAdd;
        auto beforeCollisionEntities = entityManager.getAllEntities();

        for (size_t i = 0; i < beforeCollisionEntities.size(); ++i) {
            int entityId1 = beforeCollisionEntities[i].getId();
            for (size_t j = i + 1; j < beforeCollisionEntities.size(); ++j) {
                int entityId2 = beforeCollisionEntities[j].getId();
                // Check for collision
                if (_collisionSystem.get()->checkCollision(
                        componentManager, entityId1, entityId2)) {
                    if (collisionAction(server, componentManager, entityManager, entitiesToRemove,
                            entitiesToAdd, entityId1, entityId2) == false) {
                        collisionAction(server, componentManager, entityManager, entitiesToRemove,
                            entitiesToAdd, entityId2, entityId1);
                    }
                }
            }
        }

        // Add entities
        for (int entityId : entitiesToAdd) {
            r_type::net::Message<TypeMessage> msg;
            msg.header.id = TypeMessage::CreateEntityMessage;
            msg « server->InitiateWeaponForce(entityId);
            server->MessageAllClients(msg);
        }
        // Remove entities
        for (int entityId : entitiesToRemove) {
            r_type::net::Message<TypeMessage> msg;
            msg.header.id = TypeMessage::DestroyEntityMessage;
            msg « entityId;
            server->MessageAllClients(msg);
            if (auto playerComponent = componentManager.getComponent<PlayerComponent>(entityId)) {
                continue;
            }
            if (auto bossComponent = componentManager.getComponent<BossComponent>(entityId)) {
                server->RemoveBossTail(entityId);
            }
            componentManager.removeEntityFromAllComponents(entityId);
            entityManager.removeEntity(entityId);
        }
        // Remove entities when they go off-screen
        auto afterCollisionEntities = entityManager.getAllEntities();
        for (const auto &entity : afterCollisionEntities) {
            int entityId = entity.getId();
            if (_collisionSystem.get()->checkOffScreen(componentManager, entityId)) {
                r_type::net::Message<TypeMessage> msg;
                msg.header.id = TypeMessage::DestroyEntityMessage;
                msg « entityId;
                server->MessageAllClients(msg);
                componentManager.removeEntityFromAllComponents(entityId);
                entityManager.removeEntity(entityId);
            }
        }
    }

    void AnimationUpdate(r_type::net::AServer<T> *server, ComponentManager &componentManager,
        EntityManager &entityManager, std::chrono::system_clock::time_point newClock) override
    {
        if (auto animationsBefore = componentManager.getComponentMap<AnimationComponent>()) {

            std::unordered_map<int, AnimationComponent> previousAnimations;

```

```
            // Save previous animations
            for (const auto &pair : **animationsBefore) {
                int entityId = pair.first;
                const auto animationComponent = pair.second;
                auto animation = std::any_cast<AnimationComponent>(&animationComponent);
                if (animation) {
                    previousAnimations.insert({entityId, *animation});
                }
            }
            _animationSystem->AnimationEntities(
                componentManager, entityManager, 0.2, server->_endOfLevel);
            // Compare new Animations
            if (auto animationsAfter = componentManager.getComponentMap<AnimationComponent>()) {
                for (const auto &pair : **animationsAfter) {
                    int entityId = pair.first;
                    const auto &newAnimationComponent = pair.second;
                    auto newAnimation = std::any_cast<AnimationComponent>(&newAnimationComponent);
                    if (newAnimation) {
                        auto it = previousAnimations.find(entityId);
                        if (it != previousAnimations.end()) {
                            const auto &oldAnimation = it->second;
                            if (oldAnimation != *newAnimation) {
                                r_type::net::Message<TypeMessage> msg;
                                msg.header.id = TypeMessage::AnimateEntityMessage;
                                msg « entityId « newAnimation->dimension « newAnimation->offset;
                                server->MessageAllClients(msg);
                            }
                        }
                    }
                }
            }
        }
    }

    void FireUpdate(r_type::net::AServer<T> *server, ComponentManager &componentManager,
        EntityManager &entityManager, std::chrono::system_clock::time_point newClock) override
    {
        // auto fire system
        _autoFireSystem->handleAutoFire(componentManager, entityManager);
        auto shootComponentMap = componentManager.getComponentMap<ShootComponent>();
        if (shootComponentMap) {
            for (auto &pair : **shootComponentMap) {
                int entityId = pair.first;
                auto &shootComponent = pair.second;
                if (auto shootInfo = std::any_cast<ShootComponent>(&shootComponent)) {
                    if (shootInfo->canShoot) {
                        auto weapon =
                            componentManager.getComponent<ForceWeaponComponent>(entityId);
                        if (weapon) {
                            if (!weapon.value()->attached) {
                                Entity missile = server->GetEntityFactory().createForceMissile(
                                    entityManager, componentManager, entityId);
                                r_type::net::Message<TypeMessage> weaponMissileMsg;
                                weaponMissileMsg.header.id = TypeMessage::CreateEntityMessage;
                                weaponMissileMsg « server->InitiatePlayerMissile(missile.getId());
                                server->MessageAllClients(weaponMissileMsg);
                            }
                            shootInfo->canShoot = false;
                        } else {
                            r_type::net::Message<TypeMessage> enemyMissileMsg;
                            enemyMissileMsg.header.id = TypeMessage::CreateEntityMessage;
                            enemyMissileMsg « server->InitiateEnemyMissile(entityId);
                            server->MessageAllClients(enemyMissileMsg);
                            shootInfo->canShoot = false;
                        }
                    }
                }
            }
        }
    }

    void LevelOne(r_type::net::AServer<T> *server, ComponentManager &componentManager,
        EntityManager &entityManager, std::chrono::system_clock::time_point newClock) override
    {
        if (std::chrono::duration_cast<std::chrono::seconds>(
                server->GetClock() - _basicMonsterSpawnTime)
                .count() > _gameParameters.spawnTimeBasicMonster) {
            SpawnEntity(server, entityManager, componentManager, _gameParameters.nbrOfBasicMonster,
                EntityFactory::EnemyType::BasicMonster);
            _basicMonsterSpawnTime = server->GetClock();
        }
        if (std::chrono::duration_cast<std::chrono::seconds>(
                server->GetClock() - _shooterEnemySpawnTime)
                .count() > _gameParameters.spawnTimeShooterEnemy) {
            SpawnEntity(server, entityManager, componentManager, _gameParameters.nbrOfShooterEnemy,
                EntityFactory::EnemyType::ShooterEnemy);
            _shooterEnemySpawnTime = server->GetClock();
```

```
        }
        if (std::chrono::duration_cast<std::chrono::seconds>(server->GetClock() - _WallSpawnTime)
                .count() > _gameParameters.spawnTimeWall) {
            SpawnEntity(server, entityManager, componentManager, _gameParameters.nbrOfWall,
                EntityFactory::EnemyType::Wall);
            _WallSpawnTime = server->GetClock();
        }
    }

    void LevelTwo(r_type::net::AServer<T> *server, ComponentManager &componentManager,
        EntityManager &entityManager, std::chrono::system_clock::time_point newClock) override
    {
        server->_bossActive = false;
        if (std::chrono::duration_cast<std::chrono::seconds>(
                server->GetClock() - _basicMonsterSpawnTime)
                .count() > _gameParameters.spawnTimeBasicMonster) {
            SpawnEntity(server, entityManager, componentManager, _gameParameters.nbrOfBasicMonster,
                EntityFactory::EnemyType::BasicMonster);
            _basicMonsterSpawnTime = server->GetClock();
        }
        if (std::chrono::duration_cast<std::chrono::seconds>(
                server->GetClock() - _shooterEnemySpawnTime)
                .count() > _gameParameters.spawnTimeShooterEnemy) {
            SpawnEntity(server, entityManager, componentManager, _gameParameters.nbrOfShooterEnemy,
                EntityFactory::EnemyType::ShooterEnemy);
            _shooterEnemySpawnTime = server->GetClock();
        }
        if (std::chrono::duration_cast<std::chrono::seconds>(server->GetClock() - _WallSpawnTime)
                .count() > _gameParameters.spawnTimeWall) {
            SpawnEntity(server, entityManager, componentManager, _gameParameters.nbrOfWall,
                EntityFactory::EnemyType::Wall);
            _WallSpawnTime = server->GetClock();
        }
    }

    void LevelThree(r_type::net::AServer<T> *server, ComponentManager &componentManager,
        EntityManager &entityManager, std::chrono::system_clock::time_point newClock) override
    {
        server->_bossActive = false;
        if (std::chrono::duration_cast<std::chrono::seconds>(
                server->GetClock() - _basicMonsterSpawnTime)
                .count() > _gameParameters.spawnTimeBasicMonster) {
            SpawnEntity(server, entityManager, componentManager, _gameParameters.nbrOfBasicMonster,
                EntityFactory::EnemyType::BasicMonster);
            _basicMonsterSpawnTime = server->GetClock();
        }
        if (std::chrono::duration_cast<std::chrono::seconds>(
                server->GetClock() - _shooterEnemySpawnTime)
                .count() > _gameParameters.spawnTimeShooterEnemy) {
            SpawnEntity(server, entityManager, componentManager, _gameParameters.nbrOfShooterEnemy,
                EntityFactory::EnemyType::ShooterEnemy);
            _shooterEnemySpawnTime = server->GetClock();
        }
        if (std::chrono::duration_cast<std::chrono::seconds>(server->GetClock() - _WallSpawnTime)
                .count() > _gameParameters.spawnTimeWall) {
            SpawnEntity(server, entityManager, componentManager, _gameParameters.nbrOfWall,
                EntityFactory::EnemyType::Wall);
            _WallSpawnTime = server->GetClock();
        }
    }

    void EndOfGame(r_type::net::AServer<T> *server, ComponentManager &componentManager,
        EntityManager &entityManager) override
    {
        auto entities = entityManager.getAllEntities();
        for (const auto &entity : entities) {
            int entityId = entity.getId();
            if (auto playerComponent = componentManager.getComponent<PlayerComponent>(entityId)) {
                server->SavePlayerScore(entityId);
            }
        }
        r_type::net::Message<TypeMessage> msg;
        msg.header.id = TypeMessage::EndOfGame;
        server->MessageAllClients(msg);
        server->_endOfLevel = true;
        server->_bossActive = true;
    }

    void SpawnEntity(r_type::net::AServer<T> *server, EntityManager &entityManager,
        ComponentManager &componentManager, int nbrOfEnemy, EntityFactory::EnemyType enemyType)
    {
        switch (enemyType) {
        case EntityFactory::EnemyType::BasicMonster: {
            int i = 0;
            int posX = 100;
            int posY = static_cast<int>((rand() % 70) + 10);
            while (i < nbrOfEnemy) {
```

```
                Entity Monster = server->GetEntityFactory().createBasicMonster(
                    entityManager, componentManager, posX, posY);
                posY += (static_cast<int>(rand() % 10) - static_cast<int>(rand() % 10));
                if (posY > 90)
                    posY = static_cast<int>((rand() % 70) + 10);
                posX += 5;
                r_type::net::Message<TypeMessage> msg;
                msg.header.id = TypeMessage::CreateEntityMessage;
                msg « server->FormatEntityInformation(Monster.getId());
                server->MessageAllClients(msg);
                i++;
            }
        } break;
        case EntityFactory::EnemyType::ShooterEnemy: {
            int i = 0;
            int posX = 99;
            int posY = static_cast<int>((rand() % 70) + 10);
            while (i < nbrOfEnemy) {
                Entity ShooterEnemy = server->GetEntityFactory().createShooterEnemy(
                    entityManager, componentManager, posX, posY);
                posX += 5;
                posY += (static_cast<int>(rand() % 20) - static_cast<int>(rand() % 10));

                if (posY > 90)
                    posY = static_cast<int>((rand() % 70) + 10);
                r_type::net::Message<TypeMessage> msg;
                msg.header.id = TypeMessage::CreateEntityMessage;
                msg « server->FormatEntityInformation(ShooterEnemy.getId());
                server->MessageAllClients(msg);
                i++;
            }
        } break;
        case EntityFactory::EnemyType::Wall: {
            int i = 0;
            int posX = 99;
            int posY = static_cast<int>((rand() % 70) + 10);

            while (i < nbrOfEnemy) {
                Entity wall = server->GetEntityFactory().createWall(
                    entityManager, componentManager, posX, posY);
                posX += 5;
                posY += (static_cast<int>(rand() % 20) - static_cast<int>(rand() % 10));

                if (posY > 90)
                    posY = static_cast<int>((rand() % 70) + 10);
                r_type::net::Message<TypeMessage> msg;
                msg.header.id = TypeMessage::CreateEntityMessage;
                msg « server->FormatEntityInformation(wall.getId());
                server->MessageAllClients(msg);
                i++;
            }
        } break;
        case EntityFactory::EnemyType::Boss: {
            server->InitBoss(server);
        } break;
        default:
            break;
        }
    }

    EntityInformation GetEntityBackGround(r_type::net::AServer<T> *server,
        EntityManager &entityManager, ComponentManager &componentManager) override
    {
        EntityInformation entityInfo;
        auto background = componentManager.getComponentMap<BackgroundComponent>();
        if (background) {
            for (auto &pair : **background) {
                int entityId = pair.first;
                auto &backgroundComponent = pair.second;
                if (auto backgroundInfo =
                        std::any_cast<BackgroundComponent>(&backgroundComponent)) {
                    return server->FormatEntityInformation(entityId);
                }
            }
        } else {
            return InitiateBackground(server, entityManager, componentManager);
        }
        return entityInfo;
    }

    void ChangeBackground(r_type::net::AServer<T> *server, EntityManager &entityManager,
        ComponentManager &componentManager) override
    {
        r_type::net::Message<TypeMessage> msg;
        msg.header.id = TypeMessage::DestroyEntityMessage;
        auto background = componentManager.getComponentMap<BackgroundComponent>();
        if (background) {
```

```
            for (auto &pair : **background) {
                int entityId = pair.first;
                auto &backgroundComponent = pair.second;
                if (auto backgroundInfo =
                        std::any_cast<BackgroundComponent>(&backgroundComponent)) {
                    msg « entityId;
                    server->MessageAllClients(msg);
                    componentManager.removeEntityFromAllComponents(entityId);
                    entityManager.removeEntity(entityId);
                    break;
                }
            }
        }
        r_type::net::Message<TypeMessage> newMsg;
        newMsg.header.id = TypeMessage::CreateEntityMessage;
        EntityInformation entityInfo = InitiateBackground(server, entityManager, componentManager);
        newMsg « entityInfo;
        server->MessageAllClients(newMsg);
    }

    GameState GetLevel() override { return _gameParameters.levelType; }

    EntityInformation InitiateBackground(r_type::net::AServer<T> *server,
        EntityManager &entityManager, ComponentManager &componentManager) override
    {
        EntityInformation entityInfo;
        Entity background = server->GetEntityFactory().backgroundFactory(
            entityManager, componentManager, _gameParameters.levelType);
        entityInfo.uniqueID = background.getId();
        auto sprite = componentManager.getComponent<SpriteDataComponent>(entityInfo.uniqueID);
        auto backgroundPos = componentManager.getComponent<PositionComponent>(entityInfo.uniqueID);
        auto animation = componentManager.getComponent<AnimationComponent>(entityInfo.uniqueID);
        if (sprite) {
            entityInfo.spriteData = *(sprite.value());
            entityInfo.vPos.x = backgroundPos.value()->x;
            entityInfo.vPos.y = backgroundPos.value()->y;
            if (animation) {
                entityInfo.ratio.x =
                    (animation.value()->dimension.x * sprite.value()->scale.x) / SCREEN_WIDTH;
                entityInfo.ratio.y =
                    (animation.value()->dimension.y * sprite.value()->scale.y) / SCREEN_HEIGHT;
                entityInfo.animationComponent.dimension = animation.value()->dimension;
                entityInfo.animationComponent.offset = animation.value()->offset;
            }
        }
        return entityInfo;
    }

    void SetGameParameters(GameParameters gameParameters) { _gameParameters = gameParameters; }

    void ChangeLevel(GameState state) override { _gameParameters.levelType = state; }

  protected:
    std::shared_ptr<MoveSystem> _moveSystem;
    std::shared_ptr<CollisionSystem> _collisionSystem;
    std::shared_ptr<AnimationSystem> _animationSystem;
    std::shared_ptr<AutoFireSystem> _autoFireSystem;

    std::chrono::system_clock::time_point _basicMonsterSpawnTime =
        std::chrono::system_clock::now();
    std::chrono::system_clock::time_point _shooterEnemySpawnTime =
        std::chrono::system_clock::now();
    std::chrono::system_clock::time_point _WallSpawnTime = std::chrono::system_clock::now();
    std::chrono::system_clock::time_point _spawnTimeMonsterThree;

    GameParameters _gameParameters;
};
} // namespace r_type
```

7.160 /home/runner/work/R-Type/R-Type/Server/Interface/Include/Net/a↩ _server.hpp File Reference

```
#include <Components/component_manager.hpp>
#include <Components/components.hpp>
#include <Entities/entity_factory.hpp>
#include <Entities/entity_manager.hpp>
#include <Net/i_server.hpp>
#include <Systems/systems.hpp>
```

```
#include <cmath>
#include <entity_struct.hpp>
#include <error_handling.hpp>
#include <filesystem>
#include <fstream>
#include <game_struct.hpp>
#include <iostream>
#include <level.hpp>
#include <macros.hpp>
#include <unordered_map>
```

Classes

- class r_type::net::AServer< T >

  AServer class template for managing server operations.

Namespaces

- namespace r_type
- namespace r_type::net

7.161 a_server.hpp

Go to the documentation of this file.

```
00001 /*
00002 ** EPITECH PROJECT, 2024
00003 ** R-Type
00004 ** File description:
00005 ** netServer
00006 */
00007
00008 #pragma once
00009
00010 #include <Components/component_manager.hpp>
00011 #include <Components/components.hpp>
00012 #include <Entities/entity_factory.hpp>
00013 #include <Entities/entity_manager.hpp>
00014 #include <Net/i_server.hpp>
00015 #include <Systems/systems.hpp>
00016 #include <cmath>
00017 #include <entity_struct.hpp>
00018 #include <error_handling.hpp>
00019 #include <filesystem>
00020 #include <fstream>
00021 #include <game_struct.hpp>
00022 #include <iostream>
00023 #include <level.hpp>
00024 #include <macros.hpp>
00025 #include <unordered_map>
00026
00027 namespace r_type {
00028 namespace net {
00029
00041 template <typename T> class AServer : virtual public r_type::net::IServer<T> {
00042   public:
00054     AServer(uint16_t port)
00055         : r_type::net::IServer<T>(),
00056           _asioSocket(_asioContext, asio::ip::udp::endpoint(asio::ip::udp::v4(), port)),
00057           _port(port)
00058     {
00059         srand(time(NULL));
00060         _componentManager = ComponentManager();
00061         _entityManager = EntityManager();
00062         _entityFactory = EntityFactory();
00063         _level = r_type::Level<T>();
00064         _level.SetSystem(_componentManager, _entityManager);
```

```
    }

    ~AServer() { Stop(); }

    bool Start()
    {
        try {
            WaitForClientMessage();

            _threadContext = std::thread([this]() { _asioContext.run(); });
        } catch (std::exception &e) {
            std::cerr « "[SERVER] Exception: " « e.what() « std::endl;
            return false;
        }
        std::cout « "[SERVER] Started on port " « _port « std::endl;
        return true;
    }

    void Stop()
    {
        _asioContext.stop();

        if (_threadContext.joinable())
            _threadContext.join();

        if (_asioSocket.is_open())
            _asioSocket.close();

        std::cout « "[SERVER] Stopped!\n";
    }

    void WaitForClientMessage()
    {
        _asioSocket.async_receive_from(asio::buffer(_tempBuffer.data(), _tempBuffer.size()),
            _clientEndpoint, [this](std::error_code ec, std::size_t bytes_recvd) {
                if (_clientEndpoint.protocol() != asio::ip::udp::v4())
                    return WaitForClientMessage();
                if (!ec) {
                    std::cout « "[SERVER] New Connection: " « _clientEndpoint « std::endl;
                    // check if connection already exists
                    std::cout « "Client endpoint connection: " « _clientEndpoint « std::endl;
                    for (auto &conn : _deqConnections) {
                        std::cout « "Client endpoint: " « conn->getEndpoint() « std::endl;
                        if (conn->getEndpoint() == _clientEndpoint) {
                            std::cout « "[" « conn->GetID() « "] Connection Approved"
                                      « std::endl;
                            std::cout « "[SERVER] Connection already exists" « std::endl;
                            if (OnClientConnect(conn)) {
                                std::cout « "[SERVER] Connection already exists" « std::endl;
                            } else {
                                std::cout « "[-----] Connection Denied" « std::endl;
                            }
                            return;
                        }
                    }

                    asio::ip::udp::socket newSocket(
                        _asioContext, asio::ip::udp::endpoint(asio::ip::udp::v4(), 0));

                    // create client socket
                    std::shared_ptr<Connection<T» newConn =
                        std::make_shared<Connection<T»(Connection<T>::owner::server, _asioContext,
                            std::move(newSocket), _clientEndpoint, _qMessagesIn);

                    if (OnClientConnect(newConn)) {
                        _deqConnections.push_back(std::move(newConn));
                        _deqConnections.back()->ConnectToClient(this, _nIDCounter++);
                        std::cout « "[" « _deqConnections.back()->GetID()
                                  « "] Connection Approved" « std::endl;
                    } else {
                        std::cout « "[-----] Connection Denied" « std::endl;
                    }
                } else {
                    std::cout « "[SERVER] New Connection Error: " « ec.message() « std::endl;
                }
                WaitForClientMessage();
            });
    }

    void MessageClient(std::shared_ptr<Connection<T>> client, const Message<T> &msg)
    {
        if (client && client->IsConnected()) {
            client->Send(msg);
        } else {
            OnClientDisconnect(client);

            client.reset();
```

```

            _deqConnections.erase(
                std::remove(_deqConnections.begin(), _deqConnections.end(), client),
                _deqConnections.end());
        }
    }

    void MessageAllClients(
        const Message<T> &msg, std::shared_ptr<Connection<T>> pIgnoreClient = nullptr)
    {
        bool bInvalidClientExists = false;

        for (auto &client : _deqConnections) {
            if (client && client->IsConnected()) {
                if (client != pIgnoreClient) {
                    client->Send(msg);
                    if (msg.header.id == TypeMessage::DestroyEntityMessage) {
                        client->_lastMsg = msg;
                        client->SetStatus(ServerStatus::WAITING);
                    }
                    if (msg.header.id == TypeMessage::GameTransitionMode) {
                        client->_lastMsg = msg;
                        client->SetStatus(ServerStatus::TRANSITION);
                    }
                }
            } else {
                OnClientDisconnect(client);
                client.reset();
                bInvalidClientExists = true;
            }
        }

        if (bInvalidClientExists)
            _deqConnections.erase(
                std::remove(_deqConnections.begin(), _deqConnections.end(), nullptr),
                _deqConnections.end());
    }

    UIEntityInformation UpdateInfoBar(int playerId)
    {
        UIEntityInformation entity;
        int clientId = GetPlayerClientId(playerId);
        int infoBarId = GetClientInfoBarId(clientId);
        auto playerHealth = _componentManager.getComponent<HealthComponent>(playerId);
        auto playerScore = _componentManager.getComponent<ScoreComponent>(playerId);
        auto barSpriteData = _componentManager.getComponent<SpriteDataComponent>(infoBarId);
        auto barTextData = _componentManager.getComponent<TextDataComponent>(infoBarId);
        if (playerHealth && barSpriteData && barTextData) {
            entity.uniqueID = infoBarId;
            entity.spriteData = *(barSpriteData.value());
            entity.textData = *(barTextData.value());
            entity.lives = playerHealth.value()->lives;
            entity.score = playerScore.value()->score;
        }
        return entity;
    }

    void Update(size_t nMaxMessages = -1, bool bWait = false)
    {
        if (_nbrOfPlayers == 0) {
            _entityManager.removeAllEntities();
            _componentManager.removeAllComponents();
            _playerConnected = false;
            _bossActive = false;
            _bossDefeated = false;
            _endOfLevel = false;
            return;
        }
        if (_nbrOfPlayers > 0 && !_playerConnected) {
            _playerConnected = true;
            _clock = std::chrono::system_clock::now();
        }
        _level.SetSystem(_componentManager, _entityManager);

        bool bUpdateEntities = false;
        std::chrono::system_clock::time_point newClock = std::chrono::system_clock::now();

        std::thread t_level([this, newClock, &bUpdateEntities]() {
            _level.Update(this, _componentManager, _entityManager, newClock, &bUpdateEntities);
        });

        auto setToAllClients = [this](ServerStatus type) {
            for (auto &client : _deqConnections) {
                if (client && client->IsConnected()) {
                    client->SetStatus(type);
                }
            }
```

```
        };

        if (_bossDefeated) {
            if (!_watingPlayersReady) {
                switch (_level.GetLevel()) {
                case GameState::LevelOne: {
                    _level.ChangeLevel(GameState::LevelTwo);
                } break;
                case GameState::LevelTwo: {
                    _level.ChangeLevel(GameState::LevelThree);
                } break;
                case GameState::LevelThree: {
                    _level.ChangeLevel(GameState::Win);
                } break;

                default:
                    break;
                }
                r_type::net::Message<T> msg;
                _watingPlayersReady = true;
                msg.header.id = TypeMessage::GameTransitionMode;
                MessageAllClients(msg);
            }
            if (_playerReady == _nbrOfPlayers) {
                _endOfLevel = false;
                _bossDefeated = false;
                _clock = std::chrono::system_clock::now();
                r_type::net::Message<T> msg;
                msg.header.id = TypeMessage::FinishInitialization;
                MessageAllClients(msg);
                setToAllClients(ServerStatus::RUNNING);
                _watingPlayersReady = false;
                _playerReady = 0;
                _level.ChangeBackground(this, _entityManager, _componentManager);
            }
        }

        size_t nMessageCount = 0;
        while (nMessageCount < nMaxMessages && !_qMessagesIn.empty()) {
            auto msg = _qMessagesIn.pop_front();

            OnMessage(msg.remote, msg.msg);

            nMessageCount++;
        }
        t_level.join();
        if (bUpdateEntities)
            _clock = newClock;
    }

    void UpdatePlayerPosition(PlayerMovement direction, uint32_t entityId) override
    {
        auto entitySpriteData = _componentManager.getComponent<SpriteDataComponent>(entityId);
        EntityInformation entity;

        auto hitbox = _componentManager.getComponent<HitboxComponent>(entityId);
        auto pos = _componentManager.getComponent<PositionComponent>(entityId);

        vf2d newPos = {pos.value()->x, pos.value()->y};
        switch (direction) {
        case PlayerMovement::UP: {
            newPos.y -= 2;
        } break;
        case PlayerMovement::DOWN: {
            newPos.y += 2;
        } break;
        case PlayerMovement::LEFT: {
            newPos.x -= 2;
        } break;
        case PlayerMovement::RIGHT: {
            newPos.x += 2;
        } break;
        }

        if (hitbox && pos) {
            float halfWidth = hitbox.value()->w / 2;
            float halfHeight = hitbox.value()->h / 2;
            float minX, maxX, minY, maxY;

            maxX = ((newPos.x / 100) * SCREEN_WIDTH) + halfWidth;
            minX = ((newPos.x / 100) * SCREEN_WIDTH) - halfWidth;
            maxY = ((newPos.y / 100) * SCREEN_HEIGHT) + halfHeight;
            minY = ((newPos.y / 100) * SCREEN_HEIGHT) - halfHeight;

            if (maxX > SCREEN_WIDTH || minX < 0 || maxY > (SCREEN_HEIGHT - 30) || minY < 0) {
                return;
            }
```

```
            auto vel = _componentManager.getComponent<VelocityComponent>(entityId);
            if (pos && vel) {
                // player go down
                if (pos.value()->y < newPos.y) {
                    vel.value()->y -= 0.1;
                    if (vel.value()->y < -1) {
                        vel.value()->y = -1;
                    }
                } else if (pos.value()->y > newPos.y) {
                    vel.value()->y += 0.1;
                    if (vel.value()->y > 1) {
                        vel.value()->y = 1;
                    }
                }
                pos.value()->x = newPos.x;
                pos.value()->y = newPos.y;

                auto frontComponent = _componentManager.getComponent<FrontComponent>(entityId);
                if (frontComponent.value()->targetId != -1) {
                    std::cout « "frontComponent.targetId: " « frontComponent.value()->targetId
                              « std::endl;
                    auto posFront = _componentManager.getComponent<PositionComponent>(
                        frontComponent.value()->targetId);
                    posFront.value()->x = newPos.x + 2;
                    posFront.value()->y = newPos.y;
                    vf2d newPosFront = {posFront.value()->x, posFront.value()->y};
                    uint32_t entityIdFront = frontComponent.value()->targetId;
                    r_type::net::Message<TypeMessage> moveMsg;
                    moveMsg.header.id = TypeMessage::MoveEntityMessage;
                    moveMsg « entityIdFront « newPosFront;
                    MessageAllClients(moveMsg);
                }
            }

            // std::cout « "Pos: " « newPos.x « " " « newPos.y « std::endl;
            // ////////////////////////////////////////////////

            // Update entity information and send to all clients
            r_type::net::Message<TypeMessage> moveMsg;
            moveMsg.header.id = TypeMessage::MoveEntityMessage;
            moveMsg « entityId « newPos;
            MessageAllClients(moveMsg);
        }
    }

    void SavePlayerScore(uint32_t playerId)
    {
        std::string directory = "GameScores";
        std::string filePath = directory + "/scores.txt";
        if (!std::filesystem::exists(directory)) {
            std::filesystem::create_directory(directory);
        }
        if (!std::filesystem::exists(filePath)) {
            std::ofstream outFile(filePath); // Creating the file
            if (!outFile) {
                throw failedToCreateFile();
            }
            outFile.close();
        }
        std::ofstream outFile(filePath, std::ios_base::app); // Open in append mode
        if (!outFile) {
            throw failedToOpenFile();
        }
        if (auto scoreComponent = _componentManager.getComponent<ScoreComponent>(playerId)) {
            std::string playerID = "Player " + std::to_string(playerId);
            outFile « playerID « ": " « scoreComponent.value()->score « "\n";
            outFile.close();
        }
    }

    uint32_t GetClientPlayerId(uint32_t id) { return _clientPlayerID[id]; }

    uint32_t GetPlayerClientId(uint32_t id)
    {
        for (const auto &pair : _clientPlayerID) {
            if (pair.second == id) {
                return pair.first;
            }
        }
        throw playerIdNotFound();
    }

    uint32_t GetClientInfoBarId(uint32_t id) { return _clientInfoBarID[id]; }

    void RemovePlayer(uint32_t id) { _clientPlayerID.erase(id); }

    void RemoveEntity(uint32_t id)
```

```
    {
        if (auto entity = _entityManager.getEntity(id)) {
            _componentManager.removeEntityFromAllComponents(id);
            _entityManager.removeEntity(id);
        }
    }

    void RemoveInfoBar(uint32_t infoBarId)
    {
        if (auto textDataComponent =
                _componentManager.getComponent<TextDataComponent>(infoBarId)) {
            for (uint32_t categoryId : textDataComponent.value()->categoryIds) {
                if (auto entity = _entityManager.getEntity(categoryId)) {
                    _entityManager.removeEntity(categoryId);
                }
            }
            RemoveEntity(infoBarId);
        }
        _clientInfoBarID.erase(infoBarId);
    }

    void RemoveBossTail(int bossId)
    {
        if (auto bossComp = _componentManager.getComponent<BossComponent>(bossId)) {
            for (size_t i = 0; i < bossComp.value()->tailSegmentIds.size(); i++) {
                int tailSegId = bossComp.value()->tailSegmentIds[i];
                if (auto entity = _entityManager.getEntity(tailSegId)) {
                    r_type::net::Message<TypeMessage> msg;
                    msg.header.id = TypeMessage::DestroyEntityMessage;
                    msg « tailSegId;
                    MessageAllClients(msg);
                    RemoveEntity(tailSegId);
                    // std::cout « "Tail segment removed" « std::endl;
                    // ////////////////////////////////////
                }
            }
        }
        _bossDefeated = true;
    }

    EntityInformation InitiatePlayer(int clientId)
    {
        EntityInformation entityInfo;
        Entity player =
            _entityFactory.createPlayer(_entityManager, _componentManager, _nbrOfPlayers);
        entityInfo.uniqueID = player.getId();
        auto playerSprite =
            _componentManager.getComponent<SpriteDataComponent>(entityInfo.uniqueID);
        auto playerPos = _componentManager.getComponent<PositionComponent>(entityInfo.uniqueID);
        auto animation = _componentManager.getComponent<AnimationComponent>(entityInfo.uniqueID);
        if (playerSprite && playerPos && animation) {
            entityInfo.ratio.x =
                (animation.value()->dimension.x * playerSprite.value()->scale.x) / SCREEN_WIDTH;
            entityInfo.ratio.y =
                (animation.value()->dimension.y * playerSprite.value()->scale.y) / SCREEN_HEIGHT;
            entityInfo.spriteData = *(playerSprite.value());
            entityInfo.vPos.x = playerPos.value()->x;
            entityInfo.vPos.y = playerPos.value()->y;
            entityInfo.animationComponent.dimension = animation.value()->dimension;
            entityInfo.animationComponent.offset = animation.value()->offset;
        }
        _clientPlayerID.insert_or_assign(clientId, entityInfo.uniqueID);
        return entityInfo;
    }

    UIEntityInformation InitInfoBar(int clientId)
    {
        UIEntityInformation entityInfo;
        Entity infoBar = _entityFactory.createInfoBar(_entityManager, _componentManager);
        entityInfo.uniqueID = infoBar.getId();
        std::cout « "GameBarInformation" « std::endl;
        std::cout « "Entity ID: " « entityInfo.uniqueID « std::endl;
        std::cout « "Info: " « entityInfo.lives « ", " « entityInfo.score « std::endl;
        std::cout « "SpriteData: " « entityInfo.spriteData.spritePath « ", "
                  « entityInfo.spriteData.scale.x « ", " « entityInfo.spriteData.scale.y « ", "
                  « std::endl;
        auto spriteData = _componentManager.getComponent<SpriteDataComponent>(entityInfo.uniqueID);
        auto textData = _componentManager.getComponent<TextDataComponent>(entityInfo.uniqueID);
        auto health = _componentManager.getComponent<HealthComponent>(GetClientPlayerId(clientId));
        if (spriteData && textData && health) {
            entityInfo.spriteData = *(spriteData.value());
            entityInfo.textData = *(textData.value());
            entityInfo.textData.categorySize = textData.value()->categorySize;
            entityInfo.lives = health.value()->lives;
        }
        _clientInfoBarID.insert_or_assign(clientId, entityInfo.uniqueID);
        return entityInfo;
```

```
    }

    EntityInformation FormatEntityInformation(uint32_t entityId)
    {
        EntityInformation entityInfo;
        auto entityPos = _componentManager.getComponent<PositionComponent>(entityId);
        auto sprite = _componentManager.getComponent<SpriteDataComponent>(entityId);
        auto animation = _componentManager.getComponent<AnimationComponent>(entityId);
        if (entityPos && sprite) {
            entityInfo.uniqueID = entityId;
            entityInfo.vPos.x = entityPos.value()->x;
            entityInfo.vPos.y = entityPos.value()->y;
            entityInfo.spriteData = *(sprite.value());
            if (animation) {
                entityInfo.ratio.x =
                    (animation.value()->dimension.x * sprite.value()->scale.x) / SCREEN_WIDTH;
                entityInfo.ratio.y =
                    (animation.value()->dimension.y * sprite.value()->scale.y) / SCREEN_HEIGHT;
                entityInfo.animationComponent.dimension = animation.value()->dimension;
                entityInfo.animationComponent.offset = animation.value()->offset;
            }
        }
        return entityInfo;
    }

    EntityInformation InitiatePlayerMissile(int entityId)
    {
        EntityInformation entityInfo;
        entityInfo.uniqueID = entityId;
        auto missilePos = _componentManager.getComponent<PositionComponent>(entityInfo.uniqueID);
        auto sprite = _componentManager.getComponent<SpriteDataComponent>(entityInfo.uniqueID);
        auto animation = _componentManager.getComponent<AnimationComponent>(entityInfo.uniqueID);
        if (missilePos && sprite) {
            entityInfo.vPos.x = missilePos.value()->x;
            entityInfo.vPos.y = missilePos.value()->y;
            entityInfo.spriteData = *(sprite.value());
            if (animation) {
                entityInfo.ratio.x =
                    (animation.value()->dimension.x * sprite.value()->scale.x) / SCREEN_WIDTH;
                entityInfo.ratio.y =
                    (animation.value()->dimension.y * sprite.value()->scale.y) / SCREEN_HEIGHT;
                entityInfo.animationComponent.dimension = animation.value()->dimension;
                entityInfo.animationComponent.offset = animation.value()->offset;
            }
        }
        return entityInfo;
    }

    EntityInformation InitiateEnemyMissile(int enemyId)
    {
        EntityInformation entityInfo;
        Entity missile =
            _entityFactory.createEnemyMissile(_entityManager, _componentManager, enemyId);
        entityInfo.uniqueID = missile.getId();
        auto missilePos = _componentManager.getComponent<PositionComponent>(entityInfo.uniqueID);
        auto sprite = _componentManager.getComponent<SpriteDataComponent>(entityInfo.uniqueID);
        auto animation = _componentManager.getComponent<AnimationComponent>(entityInfo.uniqueID);
        if (missilePos && sprite) {
            entityInfo.vPos.x = missilePos.value()->x;
            entityInfo.vPos.y = missilePos.value()->y;
            if (animation) {
                entityInfo.ratio.x =
                    (animation.value()->dimension.x * sprite.value()->scale.x) / SCREEN_WIDTH;
                entityInfo.ratio.y =
                    (animation.value()->dimension.y * sprite.value()->scale.y) / SCREEN_HEIGHT;
                entityInfo.animationComponent.dimension = animation.value()->dimension;
                entityInfo.animationComponent.offset = animation.value()->offset;
            }
            entityInfo.spriteData = *(sprite.value());
        }
        return entityInfo;
    }

    EntityInformation InitiateWeaponForce(int entityId)
    {
        EntityInformation entityInfo;
        entityInfo.uniqueID = entityId;
        auto position = _componentManager.getComponent<PositionComponent>(entityInfo.uniqueID);
        auto sprite = _componentManager.getComponent<SpriteDataComponent>(entityInfo.uniqueID);
        auto animation = _componentManager.getComponent<AnimationComponent>(entityInfo.uniqueID);
        if (position && sprite) {
            entityInfo.vPos.x = position.value()->x;
            entityInfo.vPos.y = position.value()->y;
            entityInfo.spriteData = *(sprite.value());
            if (animation) {
                entityInfo.ratio.x =
                    (animation.value()->dimension.x * sprite.value()->scale.x) / SCREEN_WIDTH;
```

```
                entityInfo.ratio.y =
                    (animation.value()->dimension.y * sprite.value()->scale.y) / SCREEN_HEIGHT;
                entityInfo.animationComponent.dimension = animation.value()->dimension;
                entityInfo.animationComponent.offset = animation.value()->offset;
            }
        }
        return entityInfo;
    }

    void InitBoss(r_type::net::AServer<T> *server)
    {
        server->_bossActive = true;
        // r_type::net::Message<TypeMessage> msg;
        // msg.header.id = TypeMessage::ChangeBackgroundMusic;
        // msg « 1;
        // server->MessageAllClients(msg);
        Entity boss = _entityFactory.createBoss(_entityManager, _componentManager, _entityFactory);
        int bossId = boss.getId();
        float segmentOffsetX = -2.0f;
        float segmentOffsetY = -2.0f;
        auto bossComp = _componentManager.getComponent<BossComponent>(bossId);
        auto bossPos = _componentManager.getComponent<PositionComponent>(bossId);
        auto bossSpriteData = _componentManager.getComponent<SpriteDataComponent>(bossId);
        auto bossAnimation = _componentManager.getComponent<AnimationComponent>(bossId);

        if (bossComp && bossPos && bossSpriteData && bossAnimation) {
            int firstTailSegId = bossComp.value()->tailSegmentIds[0];
            if (auto firstTailSegPos =
                    _componentManager.getComponent<PositionComponent>(firstTailSegId)) {
                firstTailSegPos.value()->x = 76;
                firstTailSegPos.value()->y = 88;
            }

            for (size_t i = 1; i < bossComp.value()->tailSegmentIds.size(); i++) {
                int tailSegId = bossComp.value()->tailSegmentIds[i];
                if (auto tailSegPos =
                        _componentManager.getComponent<PositionComponent>(tailSegId)) {
                    int precedingTailSegId = bossComp.value()->tailSegmentIds[i - 1];
                    if (auto precedingTailSegPos =
                            _componentManager.getComponent<PositionComponent>(
                                precedingTailSegId)) {
                        tailSegPos.value()->x = precedingTailSegPos.value()->x + segmentOffsetX;
                        tailSegPos.value()->y = precedingTailSegPos.value()->y + segmentOffsetY;
                    }
                }
            }

            EntityInformation bossEntityInfo;
            bossEntityInfo.uniqueID = bossId;

            bossEntityInfo.vPos.x = bossPos.value()->x;
            bossEntityInfo.vPos.y = bossPos.value()->y;
            bossEntityInfo.spriteData = *(bossSpriteData.value());
            bossEntityInfo.ratio.x =
                (bossAnimation.value()->dimension.x * bossSpriteData.value()->scale.x) /
                SCREEN_WIDTH;
            bossEntityInfo.ratio.y =
                (bossAnimation.value()->dimension.y * bossSpriteData.value()->scale.y) /
                SCREEN_HEIGHT;

            bossEntityInfo.animationComponent.dimension = bossAnimation.value()->dimension;
            bossEntityInfo.animationComponent.offset = bossAnimation.value()->offset;

            r_type::net::Message<TypeMessage> bossMsg;
            bossMsg.header.id = TypeMessage::CreateEntityMessage;
            bossMsg « bossEntityInfo;
            server->MessageAllClients(bossMsg);

            for (int i = 0; i != bossComp.value()->tailSegmentIds.size(); i++) {
                EntityInformation tailSegEntityInfo;
                tailSegEntityInfo.uniqueID = bossComp.value()->tailSegmentIds[i];
                auto tailSpriteData = _componentManager.getComponent<SpriteDataComponent>(
                    tailSegEntityInfo.uniqueID);
                auto tailPos =
                    _componentManager.getComponent<PositionComponent>(tailSegEntityInfo.uniqueID);
                auto tailAnimation =
                    _componentManager.getComponent<AnimationComponent>(tailSegEntityInfo.uniqueID);
                if (tailSpriteData && tailPos && tailAnimation) {
                    tailSegEntityInfo.vPos.x = tailPos.value()->x;
                    tailSegEntityInfo.vPos.y = tailPos.value()->y;
                    tailSegEntityInfo.spriteData = *(tailSpriteData.value());
                    tailSegEntityInfo.ratio.x =
                        (tailAnimation.value()->dimension.x * tailSpriteData.value()->scale.x) /
                        SCREEN_WIDTH;
                    tailSegEntityInfo.ratio.y =
                        (tailAnimation.value()->dimension.y * tailSpriteData.value()->scale.y) /
                        SCREEN_HEIGHT;
```

```
                    tailSegEntityInfo.animationComponent.dimension =
                        tailAnimation.value()->dimension;
                    tailSegEntityInfo.animationComponent.offset = tailAnimation.value()->offset;
                }
                r_type::net::Message<TypeMessage> tailMsg;
                tailMsg.header.id = TypeMessage::CreateEntityMessage;
                tailMsg « tailSegEntityInfo;
                server->MessageAllClients(tailMsg);
            }
        }
    }

    std::shared_ptr<Connection<T» getClientById(
        const std::deque<std::shared_ptr<Connection<T>» &connections, uint32_t clientId)
    {
        for (const auto &client : connections) {
            if (client && client->GetID() == clientId) {
                return client;
            }
        }
        return nullptr;
    }

    virtual void OnClientValidated(std::shared_ptr<Connection<T>> client) {}

    ComponentManager GetComponentManager() override { return _componentManager; }

    EntityManager &GetEntityManager() override { return _entityManager; }

    EntityFactory &GetEntityFactory() override { return _entityFactory; }

    std::chrono::system_clock::time_point GetClock() override { return _clock; }

    void SetClock(std::chrono::system_clock::time_point clock) { _clock = clock; }

  protected:
    virtual bool OnClientConnect(std::shared_ptr<Connection<T>> client) { return false; }

    virtual void OnClientDisconnect(std::shared_ptr<Connection<T>> client) {}

    virtual void OnMessage(std::shared_ptr<Connection<T>> client, Message<T> &msg) {}

  public:
    ThreadSafeQueue<OwnedMessage<T>> _qMessagesIn;

    std::deque<std::shared_ptr<Connection<T»> _deqConnections;

    asio::io_context _asioContext;
    std::thread _threadContext;

    asio::ip::udp::socket _asioSocket;
    asio::ip::udp::endpoint _clientEndpoint;

    std::array<uint8_t, 1024> _tempBuffer;

    uint32_t _nIDCounter = 10000;

    ComponentManager _componentManager;
    EntityManager _entityManager;
    EntityFactory _entityFactory;

    bool _endOfLevel = false;
    bool _bossActive = false;
    bool _bossDefeated = false;
    bool _watingPlayersReady = false;

    std::unordered_map<uint32_t, uint32_t> _clientPlayerID;

    std::unordered_map<uint32_t, uint32_t> _clientInfoBarID;

    int _nbrOfPlayers = 0;

    std::chrono::system_clock::time_point _clock = std::chrono::system_clock::now();
    bool _playerConnected = false;

    EntityInformation _background;

    int _port;

    r_type::Level<T> _level;

    int _playerReady = 0;
};
} // namespace net
} // namespace r_type
```

7.162 /home/runner/work/R-Type/R-Type/Server/Interface/Include/↩ Net/server.hpp File Reference

```
#include "a_server.hpp"
```

Classes

- class r_type::net::Server

  A server class that handles client connections and messaging.

Namespaces

- namespace r_type
- namespace r_type::net

7.163 server.hpp

Go to the documentation of this file.

```
00001 /*
00002 ** EPITECH PROJECT, 2024
00003 ** R-Type
00004 ** File description:
00005 ** main
00006 */
00007
00008 #pragma once
00009
00010 #include "a_server.hpp"
00011
00012 namespace r_type {
00013 namespace net {
00024 class Server : virtual public r_type::net::AServer<TypeMessage> {
00025   public:
00035     Server(uint16_t nPort)
00036         : r_type::net::IServer<TypeMessage>(), r_type::net::AServer<TypeMessage>(nPort)
00037     {
00038     }
00039
00047     ~Server() {}
00048
00049   protected:
00056     bool OnClientConnect(std::shared_ptr<r_type::net::Connection<TypeMessage» client);
00057
00064     void OnClientDisconnect(std::shared_ptr<r_type::net::Connection<TypeMessage» client,
00065         r_type::net::Message<TypeMessage> &msg);
00066
00077     void OnMessage(std::shared_ptr<r_type::net::Connection<TypeMessage» client,
00078         r_type::net::Message<TypeMessage> &msg);
00079 };
00080 } // namespace net
00081 } // namespace r_type
```

7.164 /home/runner/work/R-Type/R-Type/Server/Src/animation_↩ system.cpp File Reference

```
#include <Systems/systems.hpp>
#include <animation_system.hpp>
```

Functions

- bool operator== (const vf2d &lhs, const vf2d &rhs)
- static vf2d animationShipFactory (AnimationShip animation)

  Generates a vector representing the animation state of a ship.
- static vf2d animationBasicMonsterFactory (AnimationBasicMonster animation)
- static vf2d animationForceWeapon1Factory (AnimationForceWeapon1 animation)
- static vf2d animationForceWeapon2Factory (AnimationForceWeapon2 animation)
- static vf2d animationForceWeapon3Factory (AnimationForceWeapon3 animation)
- static vf2d animationForceMissile1Factory (AnimationForceMissile1 animation)
- static vf2d animationForceMissile2Factory (AnimationForceMissile2 animation)
- static vf2d animationForceMissile3Factory (AnimationForceMissile3 animation)
- bool operator!= (AnimationComponent animation, AnimationComponent other)

  Inequality operator for AnimationComponent.
- static void animateForceWeaponLevel1 (std::optional< AnimationComponent * > &animation)
- static void animateForceWeaponLevel2 (std::optional< AnimationComponent * > &animation)
- static void animateForceWeaponLevel3 (std::optional< AnimationComponent * > &animation)
- static void animateForceMissileLevel1 (std::optional< AnimationComponent * > &animation)
- static void animateForceMissileLevel2 (std::optional< AnimationComponent * > &animation)
- static void animateForceMissileLevel3 (std::optional< AnimationComponent * > &animation)
- vf2d animationBossFactory (AnimationBoss animation)

7.164.1 Function Documentation

7.164.1.1 animateForceMissileLevel1()

```
static void animateForceMissileLevel1 (
            std::optional< AnimationComponent * > & animation )  [static]
```

7.164.1.2 animateForceMissileLevel2()

```
static void animateForceMissileLevel2 (
            std::optional< AnimationComponent * > & animation )  [static]
```

7.164.1.3 animateForceMissileLevel3()

```
static void animateForceMissileLevel3 (
            std::optional< AnimationComponent * > & animation )  [static]
```

7.164.1.4 animateForceWeaponLevel1()

```
static void animateForceWeaponLevel1 (
            std::optional< AnimationComponent * > & animation )  [static]
```

7.164.1.5 animateForceWeaponLevel2()

```
static void animateForceWeaponLevel2 (
            std::optional< AnimationComponent * > & animation )  [static]
```

7.164.1.6 animateForceWeaponLevel3()

```
static void animateForceWeaponLevel3 (
            std::optional< AnimationComponent * > & animation )  [static]
```

7.164.1.7 animationBasicMonsterFactory()

```
static vf2d animationBasicMonsterFactory (
            AnimationBasicMonster animation )  [static]
```

7.164.1.8 animationBossFactory()

```
vf2d animationBossFactory (
            AnimationBoss animation )
```

7.164.1.9 animationForceMissile1Factory()

```
static vf2d animationForceMissile1Factory (
            AnimationForceMissile1 animation )  [static]
```

7.164.1.10 animationForceMissile2Factory()

```
static vf2d animationForceMissile2Factory (
            AnimationForceMissile2 animation )  [static]
```

7.164.1.11 animationForceMissile3Factory()

```
static vf2d animationForceMissile3Factory (
            AnimationForceMissile3 animation )  [static]
```

7.164.1.12 animationForceWeapon1Factory()

```
static vf2d animationForceWeapon1Factory (
            AnimationForceWeapon1 animation )  [static]
```

7.164.1.13 animationForceWeapon2Factory()

```
static vf2d animationForceWeapon2Factory (
            AnimationForceWeapon2 animation )  [static]
```

7.164.1.14 animationForceWeapon3Factory()

```
static vf2d animationForceWeapon3Factory (
            AnimationForceWeapon3 animation )  [static]
```

7.164.1.15 animationShipFactory()

```
static vf2d animationShipFactory (
            AnimationShip animation )  [static]
```

Generates a vector representing the animation state of a ship.

This function takes an AnimationShip enumeration value and returns a vf2d vector that corresponds to the animation state of the ship.

Parameters

animation	The animation state of the ship, represented by the AnimationShip enumeration.

Returns

vf2d A vector representing the animation state of the ship. The x-coordinate of the vector corresponds to the frame position, and the y-coordinate is always -1 for valid states. If the animation state is not recognized, the function returns {0, 0}.

7.164.1.16 operator"!=()

```
bool operator!= (
            AnimationComponent animation,
            AnimationComponent other )
```

Inequality operator for AnimationComponent.

get if two animations are different.

This operator compares two AnimationComponent objects to determine if they are not equal. Two AnimationComponent objects are considered not equal if any of their respective offset or dimension coordinates differ.

Parameters

animation	The first AnimationComponent to compare.
other	The second AnimationComponent to compare.

Returns

true if the AnimationComponent objects are not equal, false otherwise.

7.164.1.17 operator==()

```
bool operator== (
            const vf2d & lhs,
            const vf2d & rhs )
```

7.165 /home/runner/work/R-Type/R-Type/Server/Src/server.cpp File Reference

```
#include <Net/server.hpp>
#include <creatable_client_object.hpp>
```

Index