Orcs n Towers

Generated by Doxygen 1.9.8

Chapter 1

Orcs n Towers

1.1 Overview

Orcs n Towers is a tower defense game set in a fantasy setting, where orcs and other such monsters try to reach and destroy the player's castle, the player must defend against the monsters by placing different towers with specific roles. The player will have a set number of hitpoints that are depleted when enemies reach the castle. When all hitpoints are lost the player loses.

The monsters traverse a path, along which the player can place their towers. There is three different paths of which one is chosen at random for the duration of the game. Once a monster is inside a towers range, depending on the towers it will either create a projectile that matches the towers type, or apply a slowing or poison effect on the monster. Certain towers can only affect certain monsters.

The player can buy as many Towers as they can afford throughout the game, as well as upgrade them to increase the damage the tower will cause the monster and sell them. The player earns money by killing enemies as well as by progressing through the levels.

The game has 5 different levels of increasing difficulty, by introducing more monsters in amount and type at quicker intervals. The game is won once the player has defeated all levels. The player loses HP every time a monster reaches the castle, and once the HP is zero, the game is lost.

1.2 Instructions

Once you have started the game, to place towers on map, drag and drop them from the side bar to an appropriate place. Note that towers cannot be built on the path. To cancel the purchase of a tower drag it back onto the side bar. The towers range can be seen while dragging it as well as by clicking on it once on map. To upgrade or sell a tower, click on the tower and choose the wanted action from the menu that appeared on the bottom of the screen, there you can also see the towers specifications.

A level is completed once all enemies from that level have been killed. To move on to the next level, press the "next level" button that appears on the screen. The game can be paused by pressing the "pause" button on the side bar, there the player can also see their current level and how much money and HP they have.

Custom levels and paths can be created in levels.csv and paths.csv respectively, found in assets folder, read formatting instructions carefully.

1.3 How to compile the program

To compile the game, as taken from git, on the command line:

```
1. create an empty directory where the build files will be written
2. change directory to that directory
3. run: cmake ..
4. run: make
5. run: ./TD
```

SFML multi-media library (minimum version 2.5) is required.

1.4 Testing

Testing was mostly done directly in the source files, in either a project branch or on master branch. The first kinds of tests were simply rendering the game objects to be able to see them on the screen, once that was at least partially working, it was easier to gauge what exactly the game objects were doing and testing how the objects interacted with each other was started. Print statements were used as well to make it easer to follow which part of the code was being executed, and if it was the expected part.

Enemies movement was initially tested by hardcoding waypoints, to see that the logic worked, and enemies travesed the path that they were supposed to. Once towers and enemies were able to be rendered on the screen, their interactions with each other could be tested, namely that towers recognized that enemies we're within their range and could pick one to target.

When towers could clock on to enemies, the creation of projectiles by towers could be tested. Initially there were some minor issues with initializing projectiles due to different ideas on what should be passed to constructor, but that was easily solved with some adjustments. Once projectiles could be created their movement and ability to hit enemies was tested, initially they didn't seem to move; with some adjustments to their values that dictated how far they could move from their tower, it could be determined that projectiles were able to move towards enemies and hit them, and therefore cause damage to them.

When projectiles could hit enemies, the killing of enemies could be better tested, to see that enemies would actually take damage from projectiles, and once their HP would reach zero, they would die and be deleted, which they did. The testing of enemies causing damage to the player by reaching the castle, and dying when they do so, was done by allowing the enemies to reach the castle.

Not being able to buy towers if player didn't have enough money or, place towers on top of each other or on the path was simply tested by trying to do so. User interactions with the game, like the ability to pause/unpause, displaying tower information, upgrading or selling towers and moving on to the next level were tested by executing the action and observing the outcome.

Reading both levels and paths from file was tested by reading the content into containers and printing the contents as well as the status of the reading success. Firstly with correctly formatted input to see that the reading logic worked, and then with incorrectly formatted input, to test the error handling. As expected, incorrectly formatted input caused reading success to return false, and thus indicating reading failed. Once it was determined the reading of levels worked, the level execution was tested by playing through the whole game.

Additionally, we had encountered segmentation faults on different stages of development. These were addressed by running the executable with GNU Debugger.

1.5 Work log

Division of work / main responisbilities:

Pavel Filippov:

- Tower class and it's derived classes (bullet-, bomb-, missile-, poison-, and freezing tower)
- Game class

Otto Litkey:

- Graphics (buttons, textures)
- User interaction
- Menu class
- Resource container template class

Ellen Molin:

- Projectile class and it's derived classes (bullet-, bomb-, and missile projectile)
- LevelManager class
- Reading paths from file
- Player class

Leo Saied-Ahmad:

- Enemy class
- Path class

Tuan Vu:

- Map class
- Some graphics related to map and path of the game.

Weekly breakdown

Week 1

Pavel Filippov:

- Initialised implementation of base tower class.
- Estimated workload: 10 hours

Otto Litkey:

- Initialised implementation of class(es) responsible for graphics.
- Estimated workload: 6 h

Ellen Molin:

- Initialised implementation of Player and base Projectile classes.
- Estimated workload: 5h

Leo Saied-Ahmad:

- Intialised implementation of base Enemy class.
- Estimated workload:

Tuan Vu:

- Initialised implementation of Map class.
- Estimated workload: 6h

Week 2

Pavel Filippov:

- Continued implementation of base tower class.
- Estimated workload: 10 hours

Otto Litkey:

- Initialised game class and ResourceContainer template class.
- Estimated workload: 7 h

Ellen Molin:

- Continued implementation of Player and base Projectile classes.

- Estimated workload: 8h

Leo Saied-Ahmad:

- Continued implementation of base Enemy class.
- Estimated workload:

Tuan Vu:

- Continued implementation of Map class.
- Estimated workload: 8h

Week 3

Pavel Filippov:

- Implemented update function for game class, as well as for towers.
- Estimated workload: 10 h

Otto Litkey:

- Tested rendering, beginnings of dragging and dropping functionality for creating towers.
- Estimated workload: 10 h

Ellen Molin:

- Improved projectile class and added functionality.
- Estimated workload: 10h

Leo Saied-Ahmad:

- Improved enemy class functionality, specifically kill and death functions.
- Estimated workload:

Tuan Vu:

- Finished implementing loading map from file, worked on drawing and being able to sell towers
- Estimated workload: 10h

Week 4

Pavel Filippov:

- Created derived classes bulletTower and bombTower from tower. Moved tower update logic to it's own function.

- Estimated workload: 15 h

Otto Litkey:

- Continued testing rendering, finished drag and drop functionality for creating towers, and added way to pause game.
- Estimated workload: 10 h

Ellen Molin:

- Created derived classes bullet and bomb from projectile. Improved projectile methods to better work with derived classes.
- Estimated workload: 12h

Leo Saied-Ahmad:

- Created path class for directing enemy movement, and updated enemy's move function to make use of it.
- Estimated workload:

Tuan Vu:

- Continued working on map class so that it worked well with other classes.
- Estimated workload: 10h

Week 5

Pavel Filippov:

- Implemented missile tower. Worked on rendering projectiles.
- Estimated workload: 15 h

Otto Litkey:

- Implemented a level system in player class. Created menu class, migrated UI elements to work from here. Worked on rendering projectiles.
- Estimated workload: 15 h

Ellen Molin:

- Created a missile projectile that follows enemy.
- Estimated workload: 12h

Leo Saied-Ahmad:

- Worked on enemies that split when hit. Worked on graphics: show players state, end screen when player loses.
- Estimated workload:

Tuan Vu:

- Worked on map class, loading background from file
- Estimated workload: 10h

Week 6

Pavel Filippov:

- Implemented freezing effect tower and poison effect tower. Refined tower logic.
- Estimated workload: 15 h

Otto Litkey:

- Implemented explosions class to visualise bombs' explosions. Worked on graphics: created textures, show tower ranges. Improved logic behind user interactions with game objects.
- Estimated workload: 12 h

Ellen Molin:

- Implemented a levelManager that handles creating and managing levels, reads from file. Added functionality to load paths from file.
- Estimated workload: 15h

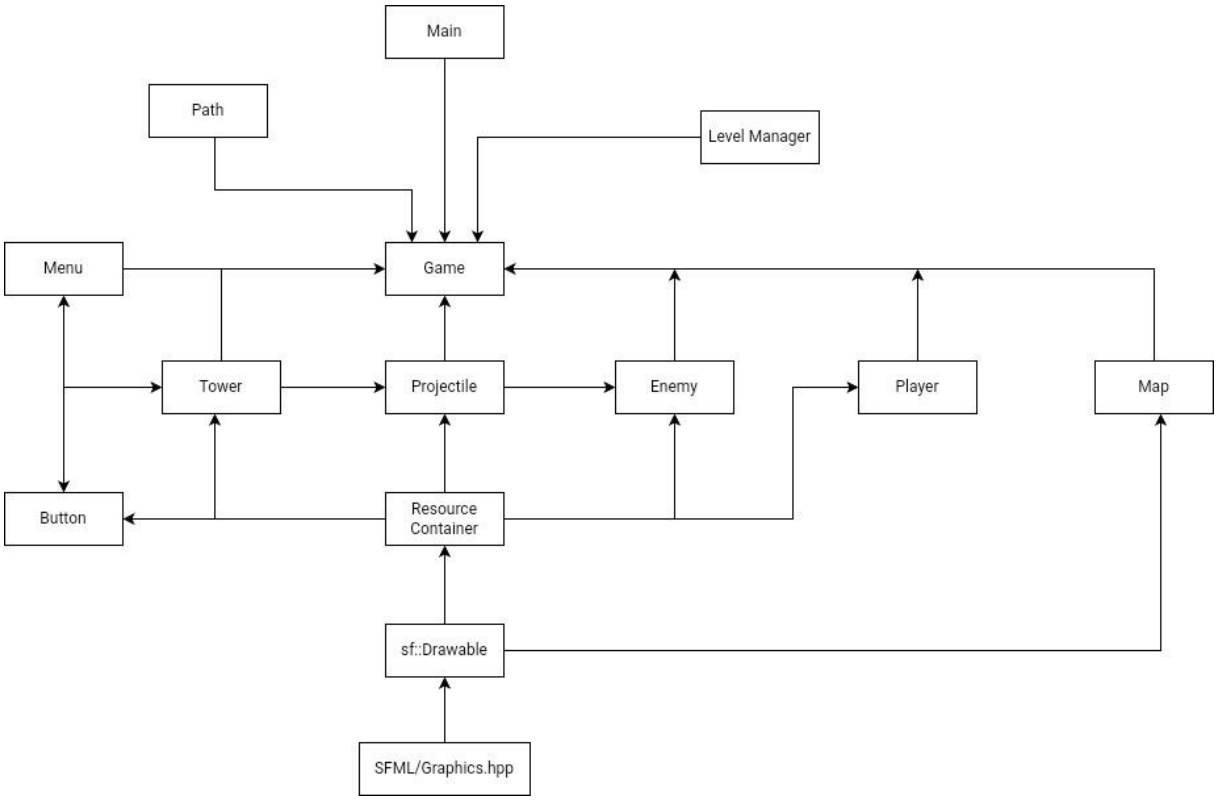
Leo Saied-Ahmad:

- Implemented slowing effect on enemies and refined enemy movement. Added health status over enemies.
- Estimated workload:

Tuan Vu:

- Implemented a path class that facilitates the creation of paths, incorporating functionality to prevent towers from being built on the designated path.
- Estimated workload: 10h

1.6 Software structure



The above image describes the relations between the main classes in the program.

The main function creates an instance of Game, which handles running the game logic and rendering. It creates and stores a sf::RenderWindow object, which handles user input and displaying graphics. The Game class stores lists of Tower, Projectile and Enemy objects. All of these inherit from sf::Sprite, which enables easy drawing and moving. Towers are added by the Menu class, which stores multiple Button objects the user can interact with. The Tower objects create Projectile objects, which the game stores and updates. A Projectile can damage an Enemy object. These are initially added to the list storing Enemy objects by the LevelManager class. The classes Path and Map handle creating the background and the path along which enemies follow. ResourceContainer is used for loading and storing sf::Texture objects.

Chapter 2

Source content

This folder should contain only hpp/cpp files of your implementation. You can also place hpp files in a separate directory `include`.

You can create a summary of files here. It might be useful to describe file relations, and brief summary of their content.

Chapter 3

Hierarchical Index

3.1 Class Hierarchy

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

Chapter 4

Class Index

4.1 Class List

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

Chapter 5

File Index

5.1 File List

Here is a list of all documented files with brief descriptions:

Chapter 6

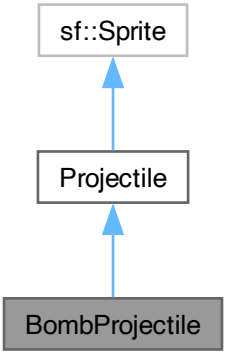
Class Documentation

6.1 BombProjectile Class Reference

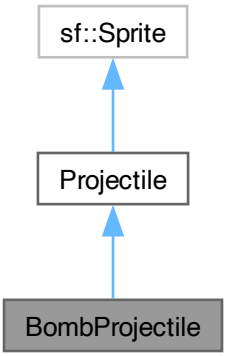
a projectile that causes damage to multiple enemies

`#include <bombProjectile.hpp>`

Inheritance diagram for BombProjectile:



Collaboration diagram for BombProjectile:



Public Member Functions

- BombProjectile (sf::Vector2f shootDirection, sf::Vector2f position, int damage, float range)
- bool hasHitEnemy (std::shared_ptr< Enemy > &enemy) override
- void update (Game &game) override
- Textures::ProjectileID textureType () override

Public Member Functions inherited from Projectile

- Projectile (sf::Vector2f shootDirection, sf::Vector2f position, int damage, float speed, std::string type, float maxDistance)

  Construcs a projectile and sets it's initial position.
- virtual ~**Projectile** ()

  Destroy the Projectile object.
- float getSpeed () const
- const std::string & getType () const
- int getDamage () const
- sf::Vector2f getShootDir () const
- void destroy ()
- bool **isDestroyed** ()

  Returns wheter the projectile is destroyed, and needs to be deleted, or not.
- bool distToTower ()

  Calculates the projectiles distance from the tower that created it.

Private Attributes

- int **blastRange_**

6.1.1 Detailed Description

a projectile that causes damage to multiple enemies

6.1.2 Constructor & Destructor Documentation

6.1.2.1 BombProjectile()

```
BombProjectile::BombProjectile (
            sf::Vector2f shootDirection,
            sf::Vector2f position,
            int damage,
            float range )  [inline]
```

Parameters

*blast↩ Range_*	the blast radius of the bomb

6.1.3 Member Function Documentation

6.1.3.1 hasHitEnemy()

```
bool BombProjectile::hasHitEnemy (
            std::shared_ptr< Enemy > & enemy )  [override], [virtual]
```

Calculates the distance between the bomb and an enemy. If the enemy is within the blast range, cause damage to it because it has been hit.

Returns

true if bomb has hit an enemy.

Parameters

enemy	is a reference to an Enemy object

Implements Projectile.

6.1.3.2 textureType()

```
Textures::ProjectileID BombProjectile::textureType ( )  [inline], [override], [virtual]
```

Returns

the texture ID of the type this derived class uses

Implements Projectile.

6.1.3.3 update()

```
void BombProjectile::update (
            Game & game )  [override], [virtual]
```

If the bomb has reached it's maximum distance, it goes through all the enemies in the game to see if it hits any, and once done with that, is destroyed. If the bomb hasn't yet reached it's maximum distance, it is moved.

Parameters

game	is a reference to the running game instance

Implements Projectile.

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

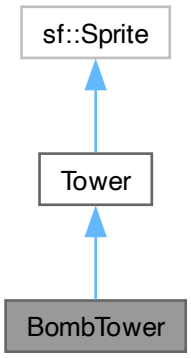
- src/bombProjectile.hpp
- src/bombProjectile.cpp

6.2 BombTower Class Reference

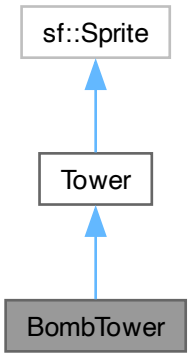
Represents the BombTower class.

`#include <bombTower.hpp>`

Inheritance diagram for BombTower:



Collaboration diagram for BombTower:



Public Member Functions

- BombTower (sf::Vector2f)

  Constructs a BombTower object at the specified position.
- void update (std::list< std::shared_ptr< Enemy > > &enemies, sf::Time time) override

  Override of the base class method.
- BombProjectile * shoot () override

  Override of the base class method to produce a BombProjectile.

Public Member Functions inherited from Tower

- Tower (sf::Vector2f position, const std::string &type, int baseCost, float range, sf::Time fireRate, int damage, int upgradeCost)

  Constructor for abstract tower is used in constructor for derived tower classes.
- const std::string & **getType** () const
- const int **getBaseCost** () const
- sf::Time **getFireRate** () const
- const float **getRange** () const
- int **getDamage** () const
- std::shared_ptr< Enemy > **getLockedEnemy** () const
- bool **isMaxLevelReached** () const
- int **getCurrentLvl** () const
- const int **getUpgradeCost** () const
- sf::Time **getFireTimer** ()
- bool enemyWithinRange (std::shared_ptr< Enemy > enemy)

  Check if the enemy is within the range of the tower.
- virtual void upgradeTower ()

  `upgradeTower()` *method is virtual as upgrade logic is same for all types of towers except FreezingTower*
- void updateFireTimer (sf::Time &dt)

  Increments fireTimer_ by dt.
- void **setLevel** (int level)
- void **setMaxLevelFlag** ()
- void **setLockedEnemy** (std::shared_ptr< Enemy > enemy)
- void **resetFireTimer** ()

6.2.1 Detailed Description

Represents the BombTower class.

The BombTower is a specialized tower that shoots BombProjectile -projectiles. It is derived from the base Tower class and inherits common tower functionalities. Bomb tower can only lock enemies of ground type. BombProjectiles can, however, damage enemies of any type within explosion range of a bomb projectile (explosion range is dictated solely by bomb projectile objects).

6.2.2 Constructor & Destructor Documentation

6.2.2.1 BombTower()

```
BombTower::BombTower (
            sf::Vector2f position )
```

Constructs a BombTower object at the specified position.

Parameters

position	The initial position of the BombTower (mouse position passed by the caller).

Uses base Tower constructor.

6.2.3 Member Function Documentation

6.2.3.1 shoot()

```
BombProjectile * BombTower::shoot ( )  [override], [virtual]
```

Override of the base class method to produce a BombProjectile.

Returns

BombProjectile* A pointer to the created BombProjectile object.

`shoot()` method calculates the direction towards locked enemy, normalizes it, and creates a BombProjectile that takes normalized direction, tower's position, damage, and locking range of the tower as arguments.

Implements Tower.

6.2.3.2 update()

```
void BombTower::update (
            std::list< std::shared_ptr< Enemy > > & enemies,
            sf::Time time )  [override], [virtual]
```

Override of the base class method.

Parameters

enemies	List of enemies passed from calling `Game::update` method.
time	Argument passed from calling `Game::update` method and is used to update `fireTimer_`.

This override for `update()` is very similar to `update()` method of base Tower class. The only difference is that it also checks EnemyType of an enemy as BombTower can only lock on enemies of `EnemyType::Ground`.

Reimplemented from Tower.

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

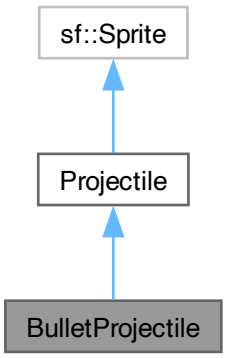
- src/bombTower.hpp
- src/bombTower.cpp

6.3 BulletProjectile Class Reference

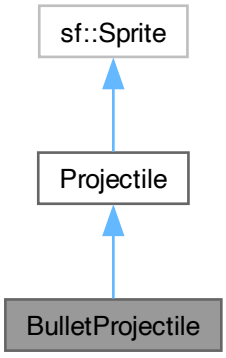
a projectile that travels in a straight line and can hit only one enemy

`#include <bulletProjectile.hpp>`

Inheritance diagram for BulletProjectile:



Collaboration diagram for BulletProjectile:



Public Member Functions

- **BulletProjectile** (sf::Vector2f shootDirection, sf::Vector2f position, int damage, float range)
- bool hasHitEnemy (std::shared_ptr< Enemy > &enemy) override

  Checks if the bullet has hit an enemy. If the bullet's and enemy's sprites intersect, there has been a hit and the bullet causes damage to the enemy.
- void update (Game &game) override
- Textures::ProjectileID textureType () override
- float **rotationAngle** () const

  Calculates the rotation angle of the bullet based on its shooting direction !!! what is it used for.

Public Member Functions inherited from Projectile

- Projectile (sf::Vector2f shootDirection, sf::Vector2f position, int damage, float speed, std::string type, float maxDistance)

  Construcs a projectile and sets it's initial position.
- virtual ~**Projectile** ()

  Destroy the Projectile object.
- float getSpeed () const
- const std::string & getType () const
- int getDamage () const
- sf::Vector2f getShootDir () const
- void destroy ()
- bool **isDestroyed** ()

  Returns wheter the projectile is destroyed, and needs to be deleted, or not.
- bool distToTower ()

  Calculates the projectiles distance from the tower that created it.

6.3.1 Detailed Description

a projectile that travels in a straight line and can hit only one enemy

6.3.2 Member Function Documentation

6.3.2.1 hasHitEnemy()

```
bool BulletProjectile::hasHitEnemy (
            std::shared_ptr< Enemy > & enemy )  [override], [virtual]
```

Checks if the bullet has hit an enemy. If the bullet's and enemy's sprites intersect, there has been a hit and the bullet causes damage to the enemy.

Returns

true if bullet has hit an enemy.

Parameters

enemy	is a reference to an Enemy object

Implements Projectile.

6.3.2.2 textureType()

```
Textures::ProjectileID BulletProjectile::textureType ( )  [inline], [override], [virtual]
```

Returns

the texture ID of the type this derived class uses.

Implements Projectile.

6.3.2.3 update()

```
void BulletProjectile::update (
            Game & game )  [override], [virtual]
```

If the bullet has gone out of range (exceeded its maximum distance), it's destroyed. Otherwise it goes through all enemies in the game to see if it has hit any one. If it has hit an enemy, the bullet is destroyed and the checking is stopped. If nothing of the before mentioned has happened, the bullet is moved.

Parameters

game	is a reference to the running game instance

Implements Projectile.

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

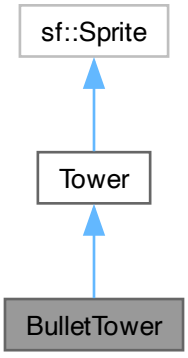
- src/bulletProjectile.hpp
- src/bulletProjectile.cpp

6.4 BulletTower Class Reference

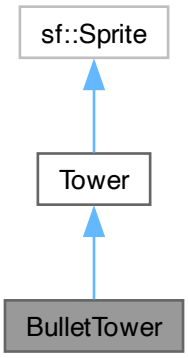
Represents the BulletTower class.

```
#include <bulletTower.hpp>
```

Inheritance diagram for BulletTower:



Collaboration diagram for BulletTower:



Public Member Functions

- BulletTower (sf::Vector2f position)

  Constructs a BulletTower object at the specified position.
- BulletProjectile * shoot () override

  Override of the base class method to produce a BulletProjectile.

Public Member Functions inherited from Tower

- Tower (sf::Vector2f position, const std::string &type, int baseCost, float range, sf::Time fireRate, int damage, int upgradeCost)

  Constructor for abstract tower is used in constructor for derived tower classes.
- const std::string & **getType** () const
- const int **getBaseCost** () const
- sf::Time **getFireRate** () const
- const float **getRange** () const
- int **getDamage** () const
- std::shared_ptr< Enemy > **getLockedEnemy** () const
- bool **isMaxLevelReached** () const
- int **getCurrentLvl** () const
- const int **getUpgradeCost** () const
- sf::Time **getFireTimer** ()
- bool enemyWithinRange (std::shared_ptr< Enemy > enemy)

  Check if the enemy is within the range of the tower.
- virtual void upgradeTower ()

  `upgradeTower()` *method is virtual as upgrade logic is same for all types of towers except FreezingTower*
- virtual void update (std::list< std::shared_ptr< Enemy > > &enemies, sf::Time time)

  `update()` *method is virtual as some types of towers use method of the base class.*
- void updateFireTimer (sf::Time &dt)

  Increments fireTimer_ by dt.
- void **setLevel** (int level)
- void **setMaxLevelFlag** ()
- void **setLockedEnemy** (std::shared_ptr< Enemy > enemy)
- void **resetFireTimer** ()

6.4.1 Detailed Description

Represents the BulletTower class.

The BulletTower is a specialized tower that shoots BulletProjectile -projectiles. It is derived from the base Tower class and inherits common tower functionalities.

6.4.2 Constructor & Destructor Documentation

6.4.2.1 BulletTower()

```
BulletTower::BulletTower (
            sf::Vector2f position )
```

Constructs a BulletTower object at the specified position.

Parameters

position	The initial position of the BulletTower (mouse position passed by the caller).

Uses base Tower constructor.

6.4.3 Member Function Documentation

6.4.3.1 shoot()

```
BulletProjectile * BulletTower::shoot ( )  [override], [virtual]
```

Override of the base class method to produce a BulletProjectile.

Returns

BulletProjectile∗ A pointer to the created BulletProjectile object.

`shoot()` method calculates the direction towards locked enemy, normalizes it, and creates a BulletProjectile that takes normalized direction, tower's position, damage, and slightly increased locking range of the tower as arguments.

Implements Tower.

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

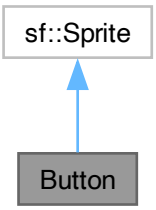
- src/bulletTower.hpp
- src/bulletTower.cpp

6.5 Button Class Reference

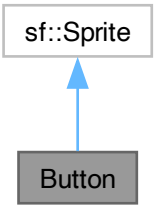
Represents a clickable button.

`#include <button.hpp>`

Inheritance diagram for Button:



Collaboration diagram for Button:



Public Member Functions

- Button (Actions action, sf::Texture &texture, sf::Vector2f position, std::string text, sf::Font &font)

  Constructs a button.
- bool isClicked (sf::Vector2f mousePos) const

  checks if the button has been clicked.
- Actions **getAction** () const
- sf::Text **getLabel** () const

Private Attributes

- Actions **action_**
- sf::Text **label_**

6.5.1 Detailed Description

Represents a clickable button.

6.5.2 Constructor & Destructor Documentation

6.5.2.1 Button()

```
Button::Button (
            Actions action,
            sf::Texture & texture,
            sf::Vector2f position,
            std::string text,
            sf::Font & font )  [inline]
```

Constructs a button.

Parameters

action	is the Actions enum determining the button type
texture	is the texture for the button
position	is the button position
text	is the button lable text
font	is the font used for the button

6.5.3 Member Function Documentation

6.5.3.1 isClicked()

```
bool Button::isClicked (
            sf::Vector2f mousePos ) const  [inline]
```

checks if the button has been clicked.

Returns

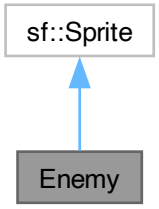
true if button was clicked, false otherwise.

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

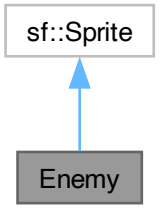
- src/button.hpp

6.6 Enemy Class Reference

Inheritance diagram for Enemy:



Collaboration diagram for Enemy:



Public Member Functions

- Enemy (int hp, int speed, EnemyType type, int money, std::queue< sf::Vector2f > waypoints)
- void **update** (sf::Time time)

  Update function for enemies, updates enemy positions based on movement, and manages/applies status effects.
- sf::Vector2f getCenter ()
- sf::Vector2f getLocation ()
- bool dead ()
- int hp ()
- int initialHp ()
- float speed ()
- int poisonStatus ()
- sf::Time slowedStatus ()
- EnemyType type ()
- void **takeDamage** (int damage)

  //damages the enemy, takes in a damage value as a parameter, if the damage is higher than the health the enemy is a automatically killed
- void **kill** ()

kills the enemy, sets dead variable to true

- void **applyPoison** (int stacksOfPoison, int damagePerStack)

  applies poison status effect to enemies
- void **applySlowed** (sf::Time duration, float slowCoefficient)

  applies slowed status effect to enemies
- void **slowedDamage** ()
- void **setVelocity** ()

  sets the enemy velocity based on where the current waypoint is
- bool isWaypointPassed (sf::Vector2f movement)

  checks to see if the enemies current waypoint will be passed, this is determined by the movement variable of the enemy
- void **findNewWaypoint** ()

  finds a newwaypoint for the enemy, this function goes through the waypoints qeue and sets the current waypoint as the next waypoint in the qeue if waypoints are empty it means the enemy has reached the castle and the enemy is set to state dead
- std::queue< sf::Vector2f > getWaypoints ()
- void **moveEnemy** (sf::Vector2f movement)
- int getMoney () const
- void **updateHealthText** (const sf::Font &font)

  updates the health text above enemies with the enemies current health
- const sf::Text & getHealthText () const

Private Attributes

- int **hp_**
- int **initialHp_**
- bool **dead_** = false
- float **speed_**
- float **actualSpeed_**
- float **effectiveSpeed_**
- sf::Text **healthText_**
- EnemyType **type_**
- int **poison_** = 0
- sf::Time **slowed_** = sf::Time::Zero
- int **money_**
- sf::Vector2f **velocity_**
- std::queue< sf::Vector2f > **waypoints_**
- sf::Vector2f **currentWaypoint_**
- int **direction_**
- int **poisonDamage** = 0
- sf::Time **poisonTimer_**
- float **slowCoefficient_** = 0.f

6.6.1 Constructor & Destructor Documentation

6.6.1.1 Enemy()

```
Enemy::Enemy (
            int hp,
            int speed,
            EnemyType type,
            int money,
            std::queue< sf::Vector2f > waypoints )  [inline]
```

Initialises an enemy

Parameters

hp	reference to the health of the enemy
speed	reference to the speed of the enemy
type	reference to the enemy type
money	reference to the amount of money the enemy is worth
waypoints	reference to the waypoints for the enemy to take

6.6.2 Member Function Documentation

6.6.2.1 dead()

```
bool Enemy::dead ( )
```

Returns

returns boolean on the sate of the enemy, false if alive true if dead

6.6.2.2 getCenter()

```
sf::Vector2f Enemy::getCenter ( )
```

Returns

returns an sf::Vector2f corresponding to the enemies positional centre

6.6.2.3 getHealthText()

```
const sf::Text & Enemy::getHealthText ( ) const
```

Returns

returns the healthText

6.6.2.4 getLocation()

```
sf::Vector2f Enemy::getLocation ( )
```

Returns

returns the enemies location as a sf::Vector2f

6.6.2.5 getMoney()

```
int Enemy::getMoney ( ) const
```

Returns

returns the amount of money this enemy provides when killed

6.6.2.6 getWaypoints()

```
std::queue< sf::Vector2f > Enemy::getWaypoints ( )
```

Returns

returns waypoints

6.6.2.7 hp()

```
int Enemy::hp ( )
```

Returns

returns enemy hp

6.6.2.8 initialHp()

```
int Enemy::initialHp ( )
```

Returns

returns enemies initialHP, this is used for the health text, as it displays the enemies health as a fraction over the initial health

6.6.2.9 isWaypointPassed()

```
bool Enemy::isWaypointPassed (
            sf::Vector2f movement )
```

checks to see if the enemies current waypoint will be passed, this is determined by the movement variable of the enemy

Returns

returns a bool

6.6.2.10 poisonStatus()

```
int Enemy::poisonStatus ( )
```

Returns

returns the duration of poison status effect

6.6.2.11 slowedStatus()

```
sf::Time Enemy::slowedStatus ( )
```

Returns

returns the duration of slowed status effect

6.6.2.12 speed()

```
float Enemy::speed ( )
```

Returns

returns enemies speed

6.6.2.13 type()

```
EnemyType Enemy::type ( )
```

Returns

returns enemy type

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

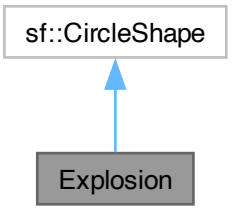
- src/enemy.hpp
- src/enemy.cpp

6.7 Explosion Class Reference

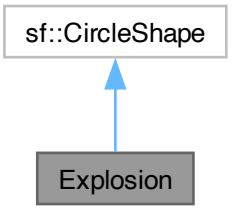
Small class for drawing bomb explosions.

`#include <explosion.hpp>`

Inheritance diagram for Explosion:



Collaboration diagram for Explosion:



Public Member Functions

- Explosion (int blastRange, sf::Vector2f pos)
  Constructs an explosion.
- void update (sf::Time inputtime)
  Updates the explosion.
- bool **isDone** ()
  Return done_ which tells if the explosion is done.

Private Attributes

- sf::Time **time_**
- int **blastRange_**
- bool **done_**

6.7.1 Detailed Description

Small class for drawing bomb explosions.

See also

BombProjectile

6.7.2 Constructor & Destructor Documentation

6.7.2.1 Explosion()

```
Explosion::Explosion (
            int blastRange,
            sf::Vector2f pos )  [inline]
```

Constructs an explosion.

Parameters

blastRange	Stores the bomb's blast range
pos	The bomb's position

6.7.3 Member Function Documentation

6.7.3.1 update()

```
void Explosion::update (
            sf::Time inputtime )  [inline]
```

Updates the explosion.

Scales the circle and reduces time left. If the time (1 second) is over, sets the flag done_

Parameters

inputtime	Time between frames from Game::getTime()

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

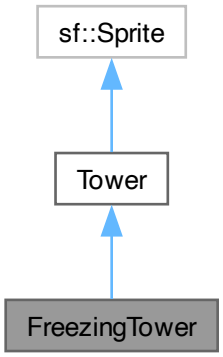
- src/explosion.hpp

6.8 FreezingTower Class Reference

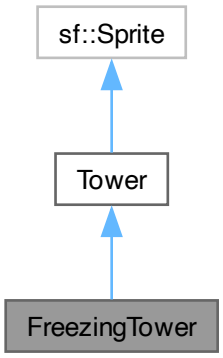
Represents the Freezing Tower class.

`#include <freezingTower.hpp>`

Inheritance diagram for FreezingTower:



Collaboration diagram for FreezingTower:



Public Member Functions

- FreezingTower (sf::Vector2f)

  Constructs a FreezingTower object at the specified position.
- void update (std::list< std::shared_ptr< Enemy > > &enemies, sf::Time time) override

  Override of the base class method.
- Projectile * shoot () override

  Override of the base class method to that applies the slowing effect on enemies.
- void upgradeTower () override

  Override of the base class upgradeTower() method.

Public Member Functions inherited from Tower

- Tower (sf::Vector2f position, const std::string &type, int baseCost, float range, sf::Time fireRate, int damage, int upgradeCost)

  Constructor for abstract tower is used in constructor for derived tower classes.
- const std::string & **getType** () const
- const int **getBaseCost** () const
- sf::Time **getFireRate** () const
- const float **getRange** () const
- int **getDamage** () const
- std::shared_ptr< Enemy > **getLockedEnemy** () const
- bool **isMaxLevelReached** () const
- int **getCurrentLvl** () const
- const int **getUpgradeCost** () const
- sf::Time **getFireTimer** ()
- bool enemyWithinRange (std::shared_ptr< Enemy > enemy)

  Check if the enemy is within the range of the tower.
- void updateFireTimer (sf::Time &dt)

  Increments fireTimer_ by dt.
- void **setLevel** (int level)
- void **setMaxLevelFlag** ()
- void **setLockedEnemy** (std::shared_ptr< Enemy > enemy)
- void **resetFireTimer** ()

Private Attributes

- std::list< std::shared_ptr< Enemy > > lockedEnemies_

  List of enemies currently locked by the FreezingTower.
- float slowCoefficient_ = 0.2

  Determines the strength of the slowing effect.

6.8.1 Detailed Description

Represents the Freezing Tower class.

The Freezing Tower is a specialized non-damaging tower that slows down all the enemies within its range. The slowing effect is accomplished by applying it directly on enemies (rather than creating a projectile). Slowing effect affects all types of enemies.

6.8.2 Constructor & Destructor Documentation

6.8.2.1 FreezingTower()

```
FreezingTower::FreezingTower (
            sf::Vector2f position )
```

Constructs a FreezingTower object at the specified position.

Parameters

position	The initial position of the FreezingTower (mouse position passed by the caller).

Uses base Tower constructor.

6.8.3 Member Function Documentation

6.8.3.1 shoot()

```
Projectile * FreezingTower::shoot ( )  [override], [virtual]
```

Override of the base class method to that applies the slowing effect on enemies.

Returns

Projectile* This override of shoot() method always returns `nullptr`.

Applies the slowing effect on every enemy within `lockedEnemies_` container. As this method doesn't actually produce a projectile and the slowing effect is applied directly on the enemy, return value of this method is always `nullptr`.

Implements Tower.

6.8.3.2 update()

```
void FreezingTower::update (
            std::list< std::shared_ptr< Enemy > > & enemies,
            sf::Time time )  [override], [virtual]
```

Override of the base class method.

Parameters

enemies	List of enemies passed from calling `Game::update` method.
time	Argument passed from calling `Game::update` method and is used to update `fireTimer_`.

This override for `update()` first updates `fireTimer_` . Then `lockedEnemies_` container is cleared and `lockedEnemy_` is set to `nullptr` . After that `enemies` container is iterated through and enemies within tower's range are added to `lockedEnemies_` container. If at this point `lockedEnemies_` is not empty, `lockedEnemy_` is set with first pointer in `std::list<std::shared_ptr<Enemy>> lockedEnemies_` .

Reimplemented from Tower.

6.8.3.3 upgradeTower()

```
void FreezingTower::upgradeTower ( )  [override], [virtual]
```

Override of the base class upgradeTower() method.

Since FreezingTower is a non-damaging tower class its upgrade has to be overriden. As opposed to base `upgradeTower()` method, upgrade of a FreezingTower increases `slowCoefficient_` rather than `damage_`.

Reimplemented from Tower.

6.8.4 Member Data Documentation

6.8.4.1 lockedEnemies_

```
std::list<std::shared_ptr<Enemy> > FreezingTower::lockedEnemies_  [private]
```

List of enemies currently locked by the FreezingTower.

This list holds shared pointers to Enemy objects that the slowing effect will be applied to.

6.8.4.2 slowCoefficient_

```
float FreezingTower::slowCoefficient_ = 0.2  [private]
```

Determines the strength of the slowing effect.

The slowCoefficient_ represents the factor by which the movement speed of enemies is reduced when affected by the slowing effect.

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

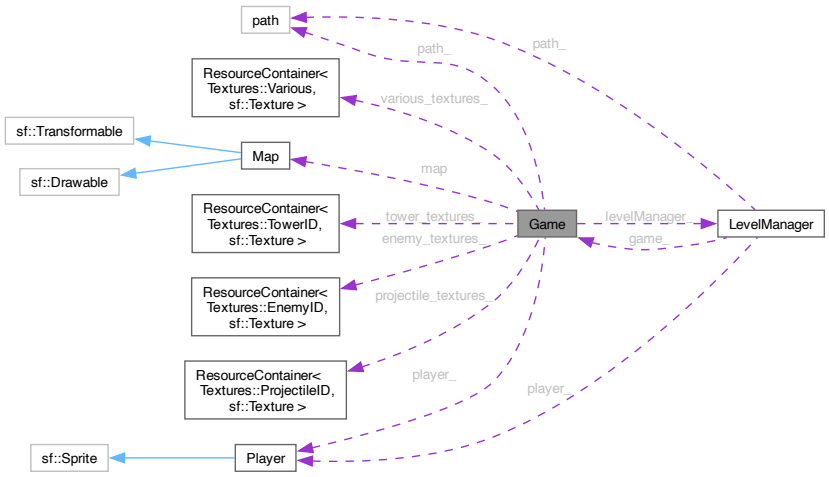
- src/freezingTower.hpp
- src/freezingTower.cpp

6.9 Game Class Reference

This class runs the game logic.

`#include <game.hpp>`

Collaboration diagram for Game:



Public Member Functions

- void run ()

  this function is called from the main function to run the game.
- path & getPath ()

  Returns the path, which enemies follow.

Public Attributes

- Map **map**

Private Member Functions

- void processEvents ()

  processes user input
- void update ()

  Updates the state of objects in the game.
- void render ()

  Renders all objects onto the window.
- void **loadTextures** ()

  Helper function called in constructor, loads all textures.
- void **createPath** ()

  Used for testing the game, creates a hardcoded path.
- void checkTowers ()

  Check if a tower has been clicked.
- void **testEnemy** ()
- void **testEnemySplit** (sf::Vector2f position, std::queue< sf::Vector2f > waypoints)
- void updateMenus ()

  Helper function for updating the menus in game, called in update().
- sf::Time **getTime** () const

Private Attributes

- sf::Clock **clock_**
- sf::Time **time_**
- sf::RenderWindow **window_**
- std::list< std::shared_ptr< Tower > > **towers_**
- std::list< std::shared_ptr< Enemy > > **enemies_**
- std::list< Projectile * > **projectiles_**
- std::list< Explosion * > **explosions_**
- path **path_**
- bool **dragged_**

  Indicates if a tower is currently being dragged into place.
- bool **paused_**

  Is the game paused.
- bool **isGameOver_** =false

  Is the game over because the player has died to an enemy.
- bool **isGameFinished_** = false

  Completed game.
- sf::Font **font_**

Stores text font.

- sf::Text **gameOverText**
- sf::Text **gameFinishedText**
- sf::Sprite **castle_sprite_**
- std::unique_ptr< Menu > **shop_**

  Shop on left side.
- std::unique_ptr< Menu > **alternativeMenu_**

  stores menu for upgrading, beginning game, and advancing to next level
- std::shared_ptr< Tower > **activeTower_**

  Pointer to tower that is being upgraded or dragged into place.
- bool **menuInactive** = false

  Indicates if the alternative menu is closed and needs to be deleted.
- ResourceContainer< Textures::TowerID, sf::Texture > **tower_textures_**
- ResourceContainer< Textures::EnemyID, sf::Texture > **enemy_textures_**
- ResourceContainer< Textures::ProjectileID, sf::Texture > **projectile_textures_**
- ResourceContainer< Textures::Various, sf::Texture > **various_textures_**
- Player **player_**
- LevelManager **levelManager_**

Friends

- class **Tower**
- class **BulletTower**
- class **BombTower**
- class **MissileTower**
- class **FreezingTower**
- class **BombProjectile**
- class **BulletProjectile**
- class **MissileProjectile**
- class **PoisonTower**
- class **Menu**
- class **LevelManager**

6.9.1 Detailed Description

This class runs the game logic.

6.9.2 Member Function Documentation

6.9.2.1 checkTowers()

```
void Game::checkTowers ( )  [private]
```

Check if a tower has been clicked.

If the mouse button has been pressed but no Button object was clicked, this checks if a purchased tower has been clicked. If a tower has been clicked, creates an upgrade menu, for upgrading or selling the tower.

6.9.2.2 getPath()

```
path & Game::getPath ( )
```

Returns the path, which enemies follow.

Returns

path& the path

6.9.2.3 processEvents()

```
void Game::processEvents ( )  [private]
```

processes user input

Gets widow events from SFML and checks if the window has been closed, or if the mouse button has been pressed. If the mouse button has been pressed checks if a button has been pressed by using Menu::checkButtons() and checks if a tower has been clicked to open the upgrade menu.

See also

checkTowers()

6.9.2.4 render()

```
void Game::render ( )  [private]
```

Renders all objects onto the window.

Clears window then draws objects. First draws the background and path, then iterates over towers, projectiles, enemies and explosions. Then draws some miscalennous things, like the tower being dragged if it exists and it's range. Menus are drawn last so they do not end up under anything.

6.9.2.5 run()

```
void Game::run ( )
```

this function is called from the main function to run the game.

If the window remains open, calls processEvents(), update(), and render() in this order.

See also

processEvents()

update()

render()

6.9.2.6 update()

```
void Game::update ( )  [private]
```

Updates the state of objects in the game.

First resets the timer, then handles updating objects by using their update functions.

6.9.2.7 updateMenus()

```
void Game::updateMenus ( )  [private]
```

Helper function for updating the menus in game, called in update().

If a tower is being dragged calls Menu::drag() to update it's position. Then updates the texts on screen. If an alternative menu has been closed deletes the alternative menu.

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

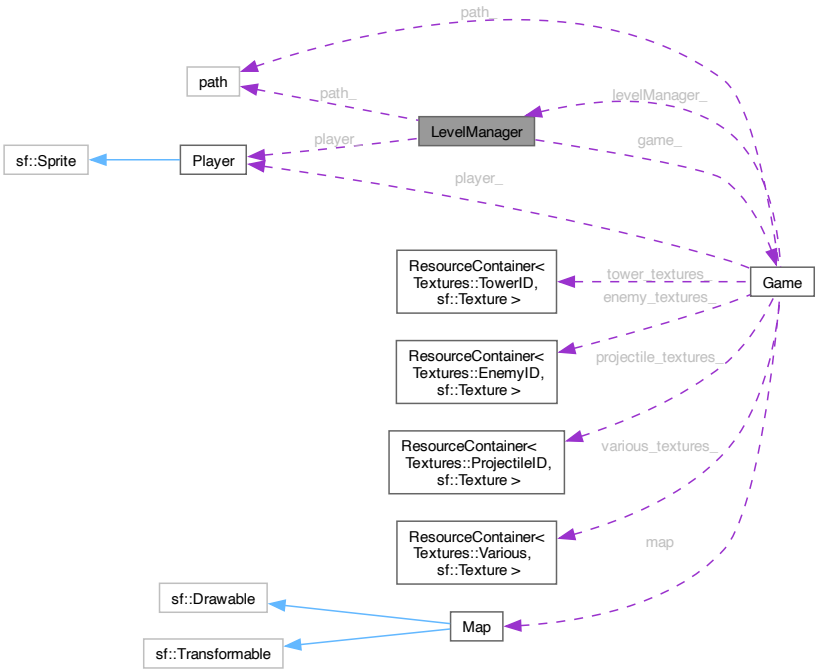
- src/game.hpp
- src/game.cpp

6.10 LevelManager Class Reference

Handles the creation and managing of levels.

```
#include <levelManager.hpp>
```

Collaboration diagram for LevelManager:



Public Types

- using **variantData** = std::variant< int, float, std::vector< int > >

  To allow the map holding level information to use different types.

Public Member Functions

- LevelManager (const std::string &src, path &path, Game &game, Player &player)
- int getCurrentLevel () const
- int getLevelTotal () const
- void update ()
- bool readingSuccessfull ()

Private Member Functions

- void readLevels ()
- void **initiateEnemies** ()

  Initiates the amount of enemies that is allowed for the level. Randomly chooses which type of enemy to initiate based on the allowed types for the level. Uses a switch case to initiate the right kind of enemy and adds it to the container of enemies. Resets the wait time and decreases waves.

Private Attributes

- std::vector< std::map< std::string, variantData > > levelSpecs_

  Container to hold all the levels. One entry in the outer container (vector) is one level, meaning index 0 is level one. The inner map holds all information regarding the specific level.
  Keys:
  .
- int **currLevel_**
- const std::string & **src_**
- bool **readingSuccess_**
- int **levelTotal_**
- float **waitTime_**
- path & **path_**
- Game & **game_**
- Player & **player_**

6.10.1 Detailed Description

Handles the creation and managing of levels.

6.10.2 Constructor & Destructor Documentation

6.10.2.1 LevelManager()

```
LevelManager::LevelManager (
            const std::string & src,
            path & path,
            Game & game,
            Player & player )  [inline]
```

Initialises a levelManager and reads the level information from file. Intitial current level is zero (= level one) to follow indexing convention of level specifications container, to allow easier accessing

Parameters

src	is the source of the level information file that is to be read
path	is a reference to the path instance that creates the path of the game
game	is a reference to the running game instance
player	is a reference to the player instance of the game

See also

readLevels()

6.10.3 Member Function Documentation

6.10.3.1 getCurrentLevel()

```
int LevelManager::getCurrentLevel ( ) const
```

Returns

the current level

6.10.3.2 getLevelTotal()

```
int LevelManager::getLevelTotal ( ) const
```

Returns

the total number of levels definend

6.10.3.3 readingSuccessfull()

```
bool LevelManager::readingSuccessfull ( )
```

returns status flag for reading level info from file.

Returns

True if reading was successfull, false if not

6.10.3.4 readLevels()

```
void LevelManager::readLevels ( )  [private]
```

Reads from the source file provided in constructor. Disregards first line of file as it is the formatting example. Then reads one line at a time:

- number of enemies per wave, number of waves, wait time between waves into variables
- allowed enemy types into a vector
  Adds the collected values into a map which gets pushed into the vector container that holds all levels.

6.10.3.5 update()

```
void LevelManager::update ( )
```

Updates the level manager, called while game is running. Counts down the wait time between waves of enemies. Initiates more enemies once wait time becomes zero, if there are waves left for the level. Moves to a new level once previous is complete and there are no enemies left.

See also

initiateEnemies()

6.10.4 Member Data Documentation

6.10.4.1 levelSpecs_

```
std::vector<std::map<std::string, variantData> > LevelManager::levelSpecs_  [private]
```

Container to hold all the levels. One entry in the outer container (vector) is one level, meaning index 0 is level one. The inner map holds all information regarding the specific level.
Keys:
.

- "enemyAmount" : the number of enemies allowed per wave (int)
- "waves" : the number of waves of enemeis allowed per level (int)
- "waitTime" : the time (in seconds) between waves (float)
- "enemyTypes" : a vector containing the types of enemies allowed for the level

  See also

  Enemy class' type enum EnemyType
  variantData The container that stores all levels information

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

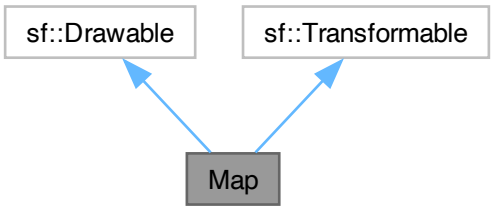
- src/levelManager.hpp
- src/levelManager.cpp

6.11 Map Class Reference

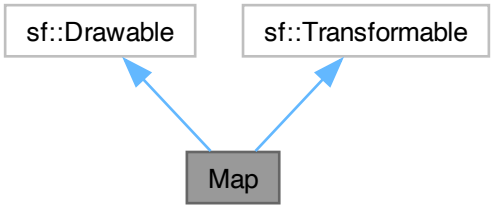
Class representing the game map.

`#include <map.hpp>`

Inheritance diagram for Map:



Collaboration diagram for Map:



Public Member Functions

- void loadMap (const std::string &fileName)

  Loads a map texture from the specified file.

Public Attributes

- sf::Texture **texture**
- sf::Sprite **background**
- std::vector< sf::FloatRect > **unBuildable**

Private Member Functions

- void draw (sf::RenderTarget &target, sf::RenderStates states) const override

  Draws the map on the specified rendering target.

6.11.1 Detailed Description

Class representing the game map.

The Map class extends sf::Drawable and sf::Transformable to provide rendering and transformation functionality. It loads a map texture and handles unbuildable areas, which prevents tower placement in specified regions.

6.11.2 Member Function Documentation

6.11.2.1 draw()

```
void Map::draw (
            sf::RenderTarget & target,
            sf::RenderStates states ) const  [override], [private]
```

Draws the map on the specified rendering target.

Parameters

target	The rendering target on which the map is drawn.
states	The rendering states to apply (overrides sf::Drawable).

6.11.2.2 loadMap()

```
void Map::loadMap (
            const std::string & fileName )
```

Loads a map texture from the specified file.

Parameters

fileName	The name of the file containing the map texture.

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

- src/map.hpp
- src/map.cpp

6.12 Menu Class Reference

Class for storing a collection of buttons, a menu.

```
#include <menu.hpp>
```

Public Member Functions

- void draw (sf::RenderWindow &window)

  Draws all the objects in the menu.
- void checkButtons (Game *game)

  Checks if a button in the menu has been pressed.
- void createMenu (MenuType menu, Game *game)

  Creates the buttons and texts of a menu.
- void update (Player &player)

  Updates the status of the menu.
- void drag (Game *game)

  Implements drag&drop placing of towers.
- void drawRange (Game *game)

  Draws active tower range.

Private Member Functions

- void newTower (std::shared_ptr< Tower > tower, Game *game)

  Adds a new tower to the game, called in checkButtons.
- bool canBePlaced (Game *game)

  Checks if a tower can be placed in its current location.

Private Attributes

- std::list< Button > **buttons_**
- std::vector< sf::Text > **texts_**
- sf::RectangleShape **bg_**

6.12.1 Detailed Description

Class for storing a collection of buttons, a menu.

6.12.2 Member Function Documentation

6.12.2.1 canBePlaced()

```
bool Menu::canBePlaced (
            Game * game )  [private]
```

Checks if a tower can be placed in its current location.

Parameters

game	Pointer to the game object

Returns

true, if the tower can be placed

6.12.2.2 checkButtons()

```
void Menu::checkButtons (
            Game * game )
```

Checks if a button in the menu has been pressed.

Checks if the mouse has clicked a button. If a button has been clicked calls getAction() on the button and does the corresponding action

Parameters

game	Pointer to the game object

6.12.2.3 createMenu()

```
void Menu::createMenu (
            MenuType menu,
            Game * game )
```

Creates the buttons and texts of a menu.

Parameters

menu	Enumerator which tells the type of menu being created
game	Poiner to the game object

6.12.2.4 drag()

```
void Menu::drag (
            Game * game )
```

Implements drag&drop placing of towers.

If the mouse button is still pressed, moves the tower so it follows the mouse if the button is no longer pressed, checks if the player has enough money for the tower and if it can be placed, and if the conditions are met adds the tower to the game object

Parameters

game	pointer to the game object

See also

canBePlaced()

6.12.2.5 draw()

```
void Menu::draw (
            sf::RenderWindow & window )
```

Draws all the objects in the menu.

Parameters

window	window onto which the objects get drawn

6.12.2.6 drawRange()

```
void Menu::drawRange (
            Game * game )
```

Draws active tower range.

Parameters

game	pointer to the game object

6.12.2.7 newTower()

```
void Menu::newTower (
            std::shared_ptr< Tower > tower,
            Game * game )  [private]
```

Adds a new tower to the game, called in checkButtons.

Parameters

tower	Pointer to new tower being built
game	Pointer to game

6.12.2.8 update()

```
void Menu::update (
            Player & player )
```

Updates the status of the menu.

Updates the texts containing the money the player has and the health

Parameters

player	Reference to the player object

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

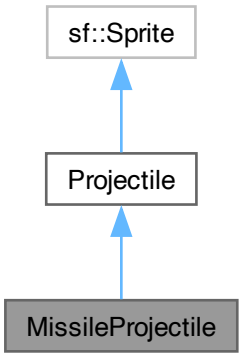
- src/menu.hpp
- src/menu.cpp

6.13 MissileProjectile Class Reference

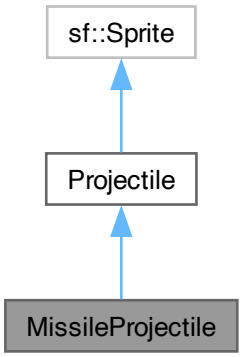
A projectile that targets (follows) a specific enemy.

`#include <missileProjectile.hpp>`

Inheritance diagram for MissileProjectile:



Collaboration diagram for MissileProjectile:



Public Member Functions

- MissileProjectile (sf::Vector2f position, int damage, std::shared_ptr< Enemy > targetEnemy)
- bool hasHitEnemy (std::shared_ptr< Enemy > &enemy) override
- void update (Game &game) override
- Textures::ProjectileID textureType () override

Public Member Functions inherited from Projectile

- Projectile (sf::Vector2f shootDirection, sf::Vector2f position, int damage, float speed, std::string type, float maxDistance)

  Construcs a projectile and sets it's initial position.
- virtual ~**Projectile** ()

  Destroy the Projectile object.
- float getSpeed () const
- const std::string & getType () const
- int getDamage () const
- sf::Vector2f getShootDir () const
- void destroy ()
- bool **isDestroyed** ()

  Returns wheter the projectile is destroyed, and needs to be deleted, or not.
- bool distToTower ()

  Calculates the projectiles distance from the tower that created it.

Private Attributes

- std::shared_ptr< Enemy > **targetEnemy_**

6.13.1 Detailed Description

A projectile that targets (follows) a specific enemy.

6.13.2 Constructor & Destructor Documentation

6.13.2.1 MissileProjectile()

```
MissileProjectile::MissileProjectile (
            sf::Vector2f position,
            int damage,
            std::shared_ptr< Enemy > targetEnemy )  [inline]
```

Missile does not need a pre-calculated directional vector, as its direction needs to be re-calculated everytime before it moves, hence the `shootDirection` is (0,0).

Parameters

targetEnemy	is the enemy that the missile is targeting (following).

6.13.3 Member Function Documentation

6.13.3.1 hasHitEnemy()

```
bool MissileProjectile::hasHitEnemy (
            std::shared_ptr< Enemy > & enemy )  [override], [virtual]
```

Checks whether the missile has hit its target or not. If the missile's and enemy's sprites intersect, there has been a hit and the missile causes damage to the enemy.

Returns

True if missile has hit it's target, otherwise false.

Parameters

enemy	is a reference to an Enemy object, the missiles target.

Implements Projectile.

6.13.3.2 textureType()

```
Textures::ProjectileID MissileProjectile::textureType ( )  [inline], [override], [virtual]
```

Returns

the texture ID of the type this derived class uses.

Implements Projectile.

6.13.3.3 update()

```
void MissileProjectile::update (
            Game & game )  [override], [virtual]
```

Firstly makes sure that the target enemy still exists, if it doesn't the missile is destroyed. If the enemy still exists it checks whether or not the missile has hit it, if there's been a hit, the missile is destroyed. If the missile has not hit the enemy, it re-calculates its directional vector, based on its and the target enemy's current positions, and moves towards the target.

Parameters

game	is a reference to the running game instance.

Implements Projectile.

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

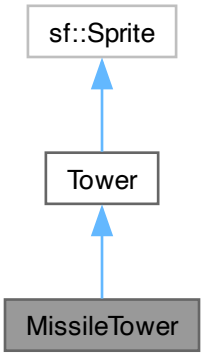
- src/missileProjectile.hpp
- src/missileProjectile.cpp

6.14 MissileTower Class Reference

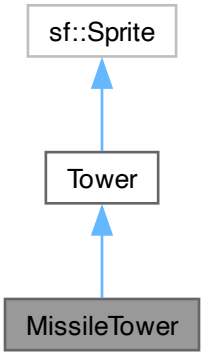
Represents the MissileTower class.

`#include <missileTower.hpp>`

Inheritance diagram for MissileTower:



Collaboration diagram for MissileTower:



Public Member Functions

- MissileTower (sf::Vector2f)

  Constructs a MissileTower object at the specified position.
- MissileProjectile * shoot () override

  Override of the base class method to produce a MissileProjectile.

Public Member Functions inherited from Tower

- Tower (sf::Vector2f position, const std::string &type, int baseCost, float range, sf::Time fireRate, int damage, int upgradeCost)

  Constructor for abstract tower is used in constructor for derived tower classes.
- const std::string & **getType** () const
- const int **getBaseCost** () const
- sf::Time **getFireRate** () const
- const float **getRange** () const
- int **getDamage** () const
- std::shared_ptr< Enemy > **getLockedEnemy** () const
- bool **isMaxLevelReached** () const
- int **getCurrentLvl** () const
- const int **getUpgradeCost** () const
- sf::Time **getFireTimer** ()
- bool enemyWithinRange (std::shared_ptr< Enemy > enemy)

  Check if the enemy is within the range of the tower.
- virtual void upgradeTower ()

  `upgradeTower()` *method is virtual as upgrade logic is same for all types of towers except FreezingTower*
- virtual void update (std::list< std::shared_ptr< Enemy > > &enemies, sf::Time time)

  `update()` *method is virtual as some types of towers use method of the base class.*
- void updateFireTimer (sf::Time &dt)

  Increments fireTimer_ by dt.
- void **setLevel** (int level)
- void **setMaxLevelFlag** ()
- void **setLockedEnemy** (std::shared_ptr< Enemy > enemy)
- void **resetFireTimer** ()

6.14.1 Detailed Description

Represents the MissileTower class.

The MissileTower is a specialized tower that shoots MissileProjectile -projectiles. It is derived from the base Tower class and inherits common tower functionalities.

6.14.2 Constructor & Destructor Documentation

6.14.2.1 MissileTower()

```
MissileTower::MissileTower (
            sf::Vector2f position )
```

Constructs a MissileTower object at the specified position.

Parameters

position	The initial position of the MissileTower (mouse position passed by the caller).

Uses base Tower constructor.

6.14.3 Member Function Documentation

6.14.3.1 shoot()

```
MissileProjectile * MissileTower::shoot ( )  [override], [virtual]
```

Override of the base class method to produce a MissileProjectile.

Returns

MissileProjectile* A pointer to the created MissileProjectile object.

shoot() method creates a MissileProjectile that takes tower's position, damage and lockedEnemy as arguments.

Implements Tower.

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

- src/missileTower.hpp
- src/missileTower.cpp

6.15 path Class Reference

Public Member Functions

- path (const std::string &src)
- void readPath ()
- bool readingSuccessfull ()
- void **addWaypoint** (const sf::Vector2f &point)

  populates the waypoint queue with all the waypoints required for the enemy class to traverse the path
- std::queue< sf::Vector2f > getWaypoints () const
- void makeUnBuildablePath ()

Public Attributes

- std::queue< sf::Vector2f > **waypoints_**
- std::vector< sf::Vector2f > **wayPoints**
- std::vector< sf::FloatRect > **unBuildable**
- std::vector< std::vector< sf::Vector2f > > **paths_**

  The container that stores all of the paths coordinates.

Static Public Attributes

- static const float **width** = 60.f

Private Attributes

- const std::string & **src_**
- bool **readingSuccess_**

Friends

- class **enemy**

6.15.1 Constructor & Destructor Documentation

6.15.1.1 path()

```
path::path (
            const std::string & src )  [inline]
```

Constructs a path by reading coordinate values from a file, randomly chooses one of the paths and adds the coordinates to the `waypoints` containers.

Parameters

src	is the source of the path information file to be read

See also

readPath()

6.15.2 Member Function Documentation

6.15.2.1 getWaypoints()

```
std::queue< sf::Vector2f > path::getWaypoints ( ) const
```

Returns

returns std::queue of waypoints stored in waypoints_ variable

6.15.2.2 makeUnBuildablePath()

```
void path::makeUnBuildablePath ( )
```

Creates unbuildable areas along the path based on waypoints and a specified width

See also

width

6.15.2.3 readingSuccessfull()

```
bool path::readingSuccessfull ( )
```

returns status flag for reading level info from file.

Returns

True if reading was successfull, false if not

6.15.2.4 readPath()

```
void path::readPath ( )
```

Reads the source file provided in the constructor. Disregards the first line as it is the formatting example.
Reads the values into a vector of SFML vector coordinates, and then adds that vector containing the path into a a vector that contains all the paths from the file.

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

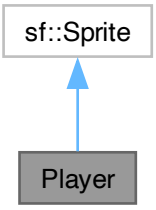
- src/path.hpp
- src/path.cpp

6.16 Player Class Reference

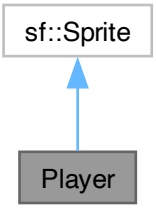
Class representing the player.

```
#include <player.hpp>
```

Inheritance diagram for Player:



Collaboration diagram for Player:



Public Member Functions

- Player ()

  Initialises a player with default values.
- int getWallet () const
- int getHP () const
- int getLevel () const
- void **levelUp** ()

  increases the players level by one
- void addMoney (int amount)

  adds money to the players wallet
- void removeMoney (int cost)

  removes money from the players wallet
- void removeHP (int amount)

  removes health points from the player

Private Attributes

- int **hp_**
- int **wallet_**
- int **level_**

6.16.1 Detailed Description

Class representing the player.

The class handles player health and money and stores the current level number.

6.16.2 Constructor & Destructor Documentation

6.16.2.1 Player()

```
Player::Player ( )  [inline]
```

Initialises a player with default values.

Parameters

*hp_*	is the health points of the player
*wallet_*	is how much money the player has
*level_*	is the level of the player

6.16.3 Member Function Documentation

6.16.3.1 addMoney()

```
void Player::addMoney (
            int amount )
```

adds money to the players wallet

Parameters

amount	is how much money is to be added

6.16.3.2 getHP()

```
int Player::getHP ( ) const
```

Returns

how many health points the player has

6.16.3.3 getLevel()

```
int Player::getLevel ( ) const
```

Returns

the current level of the player

6.16.3.4 getWallet()

```
int Player::getWallet ( ) const
```

Returns

how much money the player has

6.16.3.5 removeHP()

```
void Player::removeHP (
            int amount )
```

removes health points from the player

Parameters

amount	is how much hp is to be removed

6.16.3.6 removeMoney()

```
void Player::removeMoney (
            int cost )
```

removes money from the players wallet

Parameters

cost	is how much money is to be removed

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

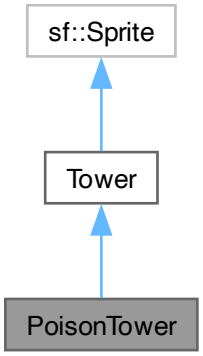
- src/player.hpp
- src/player.cpp

6.17 PoisonTower Class Reference

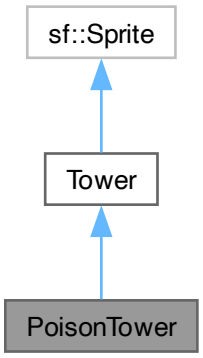
Represents the Poison Tower class.

`#include <poisonTower.hpp>`

Inheritance diagram for PoisonTower:



Collaboration diagram for PoisonTower:



Public Member Functions

- PoisonTower (sf::Vector2f)

  Constructs a PoisonTower object at the specified position.
- void update (std::list< std::shared_ptr< Enemy > > &enemies, sf::Time time) override

  Override of the base class method.
- Projectile * shoot () override

  Override of the base class method to that applies the poison effect on enemies.

Public Member Functions inherited from Tower

- Tower (sf::Vector2f position, const std::string &type, int baseCost, float range, sf::Time fireRate, int damage, int upgradeCost)

  Constructor for abstract tower is used in constructor for derived tower classes.
- const std::string & **getType** () const
- const int **getBaseCost** () const
- sf::Time **getFireRate** () const
- const float **getRange** () const
- int **getDamage** () const
- std::shared_ptr< Enemy > **getLockedEnemy** () const
- bool **isMaxLevelReached** () const
- int **getCurrentLvl** () const
- const int **getUpgradeCost** () const
- sf::Time **getFireTimer** ()
- bool enemyWithinRange (std::shared_ptr< Enemy > enemy)

  Check if the enemy is within the range of the tower.
- virtual void upgradeTower ()

  `upgradeTower()` *method is virtual as upgrade logic is same for all types of towers except FreezingTower*
- void updateFireTimer (sf::Time &dt)

  Increments fireTimer_ by dt.
- void **setLevel** (int level)
- void **setMaxLevelFlag** ()
- void **setLockedEnemy** (std::shared_ptr< Enemy > enemy)
- void **resetFireTimer** ()

Private Attributes

- std::list< std::shared_ptr< Enemy > > lockedEnemies_

  List of enemies currently locked by the PoisonTower.

6.17.1 Detailed Description

Represents the Poison Tower class.

The PoisonTower is a specialized tower that applies the poison effect on all the enemies within its range. The poison effect is accomplished by applying it directly on enemies (rather than creating a projectile) and it deals damage over time (x damage every y units of time for the duration of z seconds). The poison effect affects all types of enemies.

6.17.2 Constructor & Destructor Documentation

6.17.2.1 PoisonTower()

```
PoisonTower::PoisonTower (
            sf::Vector2f position )
```

Constructs a PoisonTower object at the specified position.

Parameters

position	The initial position of the PoisonTower (mouse position passed by the caller).

Uses base Tower constructor.

6.17.3 Member Function Documentation

6.17.3.1 shoot()

```
Projectile * PoisonTower::shoot ( )  [override], [virtual]
```

Override of the base class method to that applies the poison effect on enemies.

Returns

Projectile* This override of shoot() method always returns `nullptr`.

Applies the poison effect on every enemy within `lockedEnemies_` container. As this method doesn't actually produce a projectile and the poison effect is applied directly on the enemy, return value of this method is always `nullptr`.

Implements Tower.

6.17.3.2 update()

```
void PoisonTower::update (
            std::list< std::shared_ptr< Enemy > > & enemies,
            sf::Time time )  [override], [virtual]
```

Override of the base class method.

Parameters

enemies	List of enemies passed from calling `Game::update` method.
time	Argument passed from calling `Game::update` method and is used to update `fireTimer_`.

This override for `update()` first updates `fireTimer_`. Then `lockedEnemies_` container is cleared and `lockedEnemy_` is set to `nullptr`. After that `enemies` container is iterated through and enemies within tower's range are added to `lockedEnemies_` container. If at this point `lockedEnemies_` is not empty, `lockedEnemy_` is set with first pointer in `std::list<std::shared_ptr<Enemy>> lockedEnemies_`.

Reimplemented from Tower.

6.17.4 Member Data Documentation

6.17.4.1 lockedEnemies_

```
std::list<std::shared_ptr<Enemy> > PoisonTower::lockedEnemies_  [private]
```

List of enemies currently locked by the PoisonTower.

This list holds shared pointers to Enemy objects that the poison effect will be applied to.

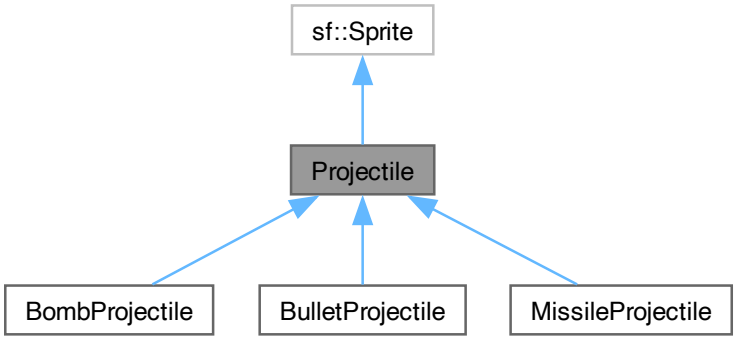
The documentation for this class was generated from the following files:

- src/poisonTower.hpp
- src/poisonTower.cpp

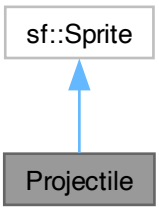
6.18 Projectile Class Reference

```
#include <projectile.hpp>
```

Inheritance diagram for Projectile:



Collaboration diagram for Projectile:



Public Member Functions

- Projectile (sf::Vector2f shootDirection, sf::Vector2f position, int damage, float speed, std::string type, float maxDistance)

  Construcs a projectile and sets it's initial position.
- virtual ~**Projectile** ()

  Destroy the Projectile object.
- float getSpeed () const
- const std::string & getType () const
- int getDamage () const
- sf::Vector2f getShootDir () const
- void destroy ()
- bool **isDestroyed** ()

  Returns wheter the projectile is destroyed, and needs to be deleted, or not.
- bool distToTower ()

  Calculates the projectiles distance from the tower that created it.
- virtual bool hasHitEnemy (std::shared_ptr< Enemy > &)=0

  checks if the projectile has hit an enemy. Overridden in each derived class.
- virtual void update (Game &)=0

  updates the projectiles state as is defiened in each derived class
- virtual Textures::ProjectileID textureType ()=0

Private Attributes

- float **speed_**
- std::string **type_**
- int **damage_**
- sf::Vector2f **position_**
- float **maxDistance_**
- sf::Vector2f **shootDirection_**
- bool **isDestroyed_**

6.18.1 Detailed Description

An abstract class for deriving projectile like, "flying", objects.

6.18.2 Constructor & Destructor Documentation

6.18.2.1 Projectile()

```
Projectile::Projectile (
            sf::Vector2f shootDirection,
            sf::Vector2f position,
            int damage,
            float speed,
            std::string type,
            float maxDistance )  [inline]
```

Construcs a projectile and sets it's initial position.

Parameters

shootDirection	is the normalised directional vector used to move the projectile, determined by the creating tower
position	is position of the tower that created the projectile, is used as a starting position
damage	is the amount of damage that the projectile will cause the enemy it hits, determined by the creating tover
speed	is the speed at which the projectile moves, pre-defiened for each derived type
type	is the type of the projectile, pre-defiened for each derived type
maxDistance	is the maximum distance the projectile is allowed to move from it's tower, based on the towers range

6.18.3 Member Function Documentation

6.18.3.1 destroy()

```
void Projectile::destroy ( )
```

Sets the `isDestroyed_` flag to true when the projectile has hit an enemy, and fullfilled its purpose, or when it has gone out of range (exceeded its max distance), and needs to be destroyed.

6.18.3.2 distToTower()

```
bool Projectile::distToTower ( )
```

Calculates the projectiles distance from the tower that created it.

Returns

true if the projectile is at, or has exceeded, its maximum distance. False otherwise

6.18.3.3 getDamage()

```
int Projectile::getDamage ( ) const
```

Returns

the damage of the projectile

6.18.3.4 getShootDir()

```
sf::Vector2f Projectile::getShootDir ( ) const
```

Returns

the directional vector of the projectile

6.18.3.5 getSpeed()

```
float Projectile::getSpeed ( ) const
```

Returns

the speed of the projectile

6.18.3.6 getType()

```
const std::string & Projectile::getType ( ) const
```

Returns

the type of the projectile

6.18.3.7 hasHitEnemy()

```
virtual bool Projectile::hasHitEnemy (
            std::shared_ptr< Enemy > &  )  [pure virtual]
```

checks if the projectile has hit an enemy. Overridden in each derived class.

Implemented in BombProjectile, BulletProjectile, and MissileProjectile.

6.18.3.8 textureType()

```
virtual Textures::ProjectileID Projectile::textureType ( )  [pure virtual]
```

Returns

the ID of the texture the projectile type uses The return value is directly hardcoded in derived classes.

Implemented in BombProjectile, BulletProjectile, and MissileProjectile.

6.18.3.9 update()

```
virtual void Projectile::update (
            Game &  )  [pure virtual]
```

updates the projectiles state as is defiened in each derived class

Implemented in BombProjectile, BulletProjectile, and MissileProjectile.

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

- src/projectile.hpp
- src/projectile.cpp

6.19 ResourceContainer< T_enum, T_resource > Class Template Reference

Template container for textures etc resources.

`#include <resource_container.hpp>`

Public Member Functions

- void load (T_enum type, std::string filename)
  Loads and stores a resource.
- T_resource & get (T_enum type) const
  Find and return requested resource.

Private Attributes

- std::map< T_enum, std::unique_ptr< T_resource > > **resources_**

6.19.1 Detailed Description

template<typename T_enum, typename T_resource>
class ResourceContainer< T_enum, T_resource >

Template container for textures etc resources.

6.19.2 Member Function Documentation

6.19.2.1 get()

```
template<typename T_enum , typename T_resource >
T_resource & ResourceContainer< T_enum, T_resource >::get (
            T_enum type ) const  [inline]
```

Find and return requested resource.

Parameters

type	Enumerator defining which texture is wanted

Returns

Returns reference to recource if found

6.19.2.2 load()

```
template<typename T_enum , typename T_resource >
void ResourceContainer< T_enum, T_resource >::load (
            T_enum type,
            std::string filename )  [inline]
```

Loads and stores a resource.

Parameters

type	Enumerator which defines the type of this resource.
filename	path to file containing the resource.

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

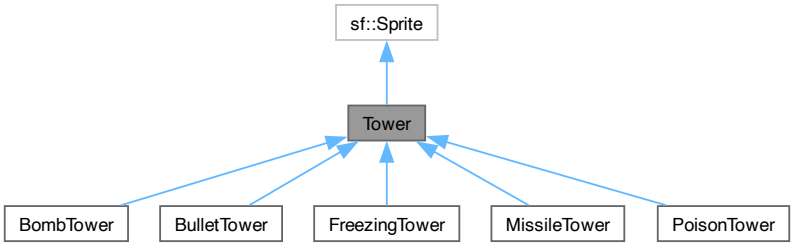
- src/resource_container.hpp

6.20 Tower Class Reference

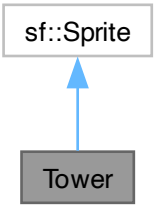
Represents abstract tower class.

`#include <tower.hpp>`

Inheritance diagram for Tower:



Collaboration diagram for Tower:



Public Member Functions

- Tower (sf::Vector2f position, const std::string &type, int baseCost, float range, sf::Time fireRate, int damage, int upgradeCost)

  Constructor for abstract tower is used in constructor for derived tower classes.
- const std::string & **getType** () const
- const int **getBaseCost** () const
- sf::Time **getFireRate** () const
- const float **getRange** () const
- int **getDamage** () const
- std::shared_ptr< Enemy > **getLockedEnemy** () const
- bool **isMaxLevelReached** () const
- int **getCurrentLvl** () const
- const int **getUpgradeCost** () const
- sf::Time **getFireTimer** ()
- bool enemyWithinRange (std::shared_ptr< Enemy > enemy)

  Check if the enemy is within the range of the tower.
- virtual Projectile * shoot ()=0

  `shoot()` *method is pure virtual as different types of towers produce different types of projectiles (or no projectiles at all as is the case with PoisonTower and FreezingTower).*
- virtual void upgradeTower ()

  `upgradeTower()` *method is virtual as upgrade logic is same for all types of towers except FreezingTower*
- virtual void update (std::list< std::shared_ptr< Enemy > > &enemies, sf::Time time)

  `update()` *method is virtual as some types of towers use method of the base class.*
- void updateFireTimer (sf::Time &dt)

  Increments fireTimer_ by dt.
- void **setLevel** (int level)
- void **setMaxLevelFlag** ()
- void **setLockedEnemy** (std::shared_ptr< Enemy > enemy)
- void **resetFireTimer** ()

Private Attributes

- const std::string type_
- const int **baseCost_**
- const float **range_**
- int **damage_**
- int **currentLvl_**
- const int **upgradeCost_**
- std::shared_ptr< Enemy > **lockedEnemy_**
- sf::Time **fireTimer_**
- sf::Time **fireRate_**
- bool **maxLevelReached_**

6.20.1 Detailed Description

Represents abstract tower class.

The Tower class is a base class for various types of towers, each with its unique characteristics. Towers can lock onto enemies within a specified range, shoot projectiles, and be upgraded to increase their effectiveness. This class acts as a common interface for managing towers and a foundation for derived tower classes.

6.20.2 Constructor & Destructor Documentation

6.20.2.1 Tower()

```
Tower::Tower (
            sf::Vector2f position,
            const std::string & type,
            int baseCost,
            float range,
            sf::Time fireRate,
            int damage,
            int upgradeCost )
```

Constructor for abstract tower is used in constructor for derived tower classes.

Parameters

position	Determined by constructor of derived tower class
type	Determined by constructor of derived tower class
baseCost	Determined by constructor of derived tower class
range	Determined by constructor of derived tower class
fireRate	Determined by constructor of derived tower class
damage	Determined by constructor of derived tower class
upgradeCost	Determined by constructor of derived tower class

6.20.3 Member Function Documentation

6.20.3.1 enemyWithinRange()

```
bool Tower::enemyWithinRange (
            std::shared_ptr< Enemy > enemy )
```

Check if the enemy is within the range of the tower.

Parameters

enemy	A shared pointer to an `Enemy` object passed from calling `Tower::update` method.

Returns

`true` if locking range of the tower is more or equal to distance between the enemy and the tower.

`false` otherwise.

6.20.3.2 shoot()

```
virtual Projectile * Tower::shoot ( )  [pure virtual]
```

`shoot()` method is pure virtual as different types of towers produce different types of projectiles (or no projectiles at all as is the case with PoisonTower and FreezingTower).

Returns

Projectile∗

Implemented in BombTower, BulletTower, FreezingTower, MissileTower, and PoisonTower.

6.20.3.3 update()

```
void Tower::update (
            std::list< std::shared_ptr< Enemy > > & enemies,
            sf::Time time )  [virtual]
```

`update()` method is virtual as some types of towers use method of the base class.

Main logic of tower.

First, we check whether currently locked enemy is not `nullptr`, not dead and still within tower's range. If this condition is satisfied nothing else is done. Otherwise, locked enemy is set to `nullptr` and `enemies` container is iterated through to find the fastest enemy which is within tower's range and alive. If there is no enemies alive within tower's range, lockedEnemy_ member stays `nullptr`. Otherwise, `lockedEnemy_` is set to the pointer to the fastest, alive enemy within tower's range.

Parameters

enemies	List argument passed from calling `Game::update` method.
time	Argument passed from calling `Game::update` method and is used to update `Tower::fireTimer_`.

Reimplemented in BombTower, FreezingTower, and PoisonTower.

6.20.3.4 updateFireTimer()

```
void Tower::updateFireTimer (
            sf::Time & dt )
```

Increments fireTimer_ by dt.

Parameters

dt	Time since last frame and is passed from `Game::update()` .

6.20.3.5 upgradeTower()

```
void Tower::upgradeTower ( )  [virtual]
```

`upgradeTower()` method is virtual as upgrade logic is same for all types of towers except FreezingTower

FreezingTower

This method upgrades tower by one level, increases its damage_ member by 1.5 times and sets the maximum level flag to true.

Note

If the maximum level has already been reached, this method has no effect

Reimplemented in FreezingTower.

6.20.4 Member Data Documentation

6.20.4.1 type_

```
const std::string Tower::type_  [private]
```

Parameters

*type_*	A string representing type of the tower
*baseCost_*	The base cost for the type of tower
*range_*	The enemy locking range of the tower
*damage_*	Damage of the tower that is passed as a parameter to projectile constructor
*currentLvl_*	Current level of the tower, initially set 1 and can be upgraded up to level 2
*upgradeCost_*	Set at 1.5 ∗ base cost of tower for all types of towers.
*lockedEnemy_*	The locked enemy of the tower; initially set to nullptr.
*fireTimer_*	Member used to count how much time has passed since last shot.
*fireRate_*	Member that dictates how often tower can shoot projectiles (or apply other effect on enemies).
*maxLevel↩ Reached_*	Flag used to check whether tower is already at max level.

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

- src/tower.hpp
- src/tower.cpp

Chapter 7

File Documentation

7.1 bombProjectile.hpp

```
00001 #ifndef BOMB_PROJECTILE
00002 #define BOMB_PROJECTILE
00003
00004 #include "projectile.hpp"
00005 #include <list>
00006
00010 class BombProjectile : public Projectile
00011 {
00012 private:
00013     int blastRange_;
00014 public:
00015
00019     BombProjectile(sf::Vector2f shootDirection, sf::Vector2f position, int damage, float range) // <-
      tbd
00020     : Projectile(shootDirection, position, damage, 60.0, "bomb", range), blastRange_(1000) {}
00021
00028     bool hasHitEnemy(std::shared_ptr<Enemy>& enemy) override;
00029
00036     void update(Game& game) override;
00037
00041     Textures::ProjectileID textureType() override { return Textures::Bomb; }
00042 };
00043
00044
00045 #endif
```

7.2 bombTower.hpp

```
00001 #ifndef BOMB_TOWER_H
00002 #define BOMB_TOWER_H
00003 #include "tower.hpp"
00004 #include "bombProjectile.hpp"
00015 class BombTower : public Tower {
00016 public:
00022     BombTower(sf::Vector2f);
00029     void update(std::list<std::shared_ptr<Enemy>> &enemies, sf::Time time) override;
00035     BombProjectile* shoot() override;
00036 };
00037 #endif //BOMB_TOWER
```

7.3 bulletProjectile.hpp

```
00001 #ifndef BULLET_PROJECTILE
00002 #define BULLET_PROJECTILE
00003
00004 #include "projectile.hpp"
00005
00009 class BulletProjectile : public Projectile
00010 {
```

```
00011 public:
00012     BulletProjectile(sf::Vector2f shootDirection, sf::Vector2f position, int damage, float range)
00013     : Projectile(shootDirection, position, damage, 500, "bullet", range) {}
00014
00022     bool hasHitEnemy(std::shared_ptr<Enemy>& enemy) override;
00023
00031     void update(Game& game) override;
00032
00036     Textures::ProjectileID textureType() override { return Textures::Bullet; }
00037
00042     float rotationAngle() const; //this one is used to calculate rotation angle of a projectile.
00043 };
00044
00045
00046 #endif
```

7.4 bulletTower.hpp

```
00001 #ifndef BULLET_TOWER_H
00002 #define BULLET_TOWER_H
00003 #include "tower.hpp"
00004 #include "bulletProjectile.hpp"
00012 class BulletTower : public Tower {
00013 public:
00019     BulletTower(sf::Vector2f position);
00025     BulletProjectile* shoot() override;
00026 };
00027 #endif //BULLET_TOWER
```

7.5 button.hpp

```
00001 #ifndef BUTTON
00002 #define BUTTON
00003 #include <SFML/Graphics.hpp>
00004
00009 enum class Actions{
00010     Tower1,
00011     Tower2,
00012     Tower3,
00013     Tower4,
00014     Tower5,
00015     Pause,
00016     Upgrade,
00017     Sell,
00018     Close,  // In upgrade menu, closes upgrade menu.
00019     Level   // Click to start level
00020 };
00021
00026 class Button : public sf::Sprite {
00027 public:
00037     Button(Actions action, sf::Texture& texture, sf::Vector2f position, std::string text, sf::Font&
      font) : action_(action) {
00038         setTexture(texture);
00039         setPosition(position);
00040         label_ = sf::Text(text, font, 15);
00041         label_.setPosition(position.x, position.y+20);
00042     }
00043
00049     bool isClicked(sf::Vector2f mousePos) const {
00050         return getGlobalBounds().contains(mousePos);
00051     }
00052
00053     Actions getAction() const { return action_; }
00054     sf::Text getLabel() const { return label_; }
00055
00056 private:
00057     Actions action_;
00058     sf::Text label_;
00059
00060 };
00061
00062
00063 #endif
```

7.6 enemy.hpp

```
00001 #ifndef ENEMY_HPP
```

```
00002 #define ENEMY_HPP
00003 #include <string>
00004 #include "path.hpp"
00005 #include <queue>
00006 #include "player.hpp"
00007 #include <SFML/System/Vector2.hpp>
00008 #include <SFML/Graphics.hpp>
00009 #include <random>
00010
00011 enum class EnemyType {
00012     Ground,
00013     Flying,
00014     Split,
00015 };
00016
00017 class Enemy :public sf::Sprite {
00018 public:
00019
00029     Enemy(int hp, int speed, EnemyType type, int money, std::queue<sf::Vector2f> waypoints)
00030         : hp_(hp), actualSpeed_(speed), speed_(speed), effectiveSpeed_(speed), type_(type),
      money_(money), waypoints_(waypoints), initialHp_(hp) {
00031
00032         // Random y value of starting pos, gets set as a negative value
00033         // So enemies spawn outside window and then move in
00034         //int rand_y = std::rand() % 40;
00035
00036         //tries to avoid enemies being on top of eachother
00037         std::random_device rd;
00038         std::uniform_int_distribution range(1,40);
00039         int x = range(rd);
00040         int y = range(rd);
00041         setPosition(waypoints_.front() - sf::Vector2f(x,y));
00042
00043         if (!waypoints_.empty()) {
00044             currentWaypoint_ = waypoints_.front();
00045         }
00046         setVelocity();
00047     }
00048
00049
00050
00051     ~Enemy() {}
00056     void update(sf::Time time);
00061     sf::Vector2f getCenter();
00066     sf::Vector2f getLocation();
00070     bool dead();
00074     int hp();
00078     int initialHp();
00082     float speed();
00086     int poisonStatus();
00090     sf::Time slowedStatus();
00094     EnemyType type();
00099     void takeDamage(int damage); //decreases the hp_ variable and if hp reaches 0 than the enemy is
      automatically destroyed
00103     void kill();
00107     void applyPoison(int stacksOfPoison, int damagePerStack);
00108
00112     void applySlowed(sf::Time duration, float slowCoefficient);
00113
00114     void slowedDamage();
00119     void setVelocity();
00124     bool isWaypointPassed(sf::Vector2f movement);
00129     void findNewWaypoint();
00133     std::queue<sf::Vector2f> getWaypoints();
00134
00135     void moveEnemy(sf::Vector2f movement);
00139     int getMoney() const;
00143     void updateHealthText(const sf::Font& font);
00147     const sf::Text& getHealthText() const;
00148
00149 private:
00150     int hp_;
00151
00152     int initialHp_;
00153
00154     bool dead_= false;
00155
00156     float speed_;
00157
00158     float actualSpeed_;
00159
00160     float effectiveSpeed_;
00161
00162     sf::Text healthText_;
00163
00164     EnemyType type_;
00165
```

```
00166     int poison_ = 0; //If poison is larger than 0 that means that the enemy is poisoned
00167     // the length of time that the enemy is poisoned for depends on how large the poison
00168     //value is as the number decreases incrimently until 0
00169     sf::Time slowed_ = sf::Time::Zero;
00170     //How much money the player recieves for killing the monster
00171     int money_;
00172     //waypoint based movement, the path class provides a queue of waypoints that take the enemies
      through the path to the end
00173
00174     sf::Vector2f velocity_;
00175
00176     std::queue<sf::Vector2f> waypoints_;
00177
00178     sf::Vector2f currentWaypoint_;
00179
00180     int direction_; //0 = down, 1= left, 2= right, 3 = up
00181
00182     int poisonDamage = 0;
00183
00184     sf::Time poisonTimer_;
00185
00186     float slowCoefficient_ = 0.f;
00187 };
00188
00189 #endif
```

7.7 explosion.hpp

```
00001 #ifndef EXPLOSION
00002 #define EXPLOSION
00003 #include <SFML/Graphics.hpp>
00004 #include <SFML/System.hpp>
00005 #include <stdio.h>
00006 #define BOMB_SIZE_HALF 24
00007
00014 class Explosion : public sf::CircleShape {
00015 public:
00022     Explosion(int blastRange, sf::Vector2f pos) : blastRange_(blastRange), done_(false) {
00023         time_ = sf::seconds(1);
00024         setPosition(pos.x + BOMB_SIZE_HALF, pos.y + BOMB_SIZE_HALF);
00025         setRadius(2);
00026         setOrigin(2, 2);
00027         setFillColor(sf::Color(255, 64, 0, 150));
00028     }
00029
00038     void update(sf::Time inputtime) {
00039         time_ -= inputtime;
00040         if (time_ < sf::microseconds(0)) {
00041             done_ = true;
00042             //std::cout « "The explosion is done" « std::endl;
00043             return;
00044         }
00045         if (time_ >= sf::seconds(0.5)) {
00046             setScale(getScale().x + 1, getScale().y + 1);
00047         } else {
00048             setScale(getScale().x - 1, getScale().y - 1);
00049         }
00050
00051     }
00052
00056     bool isDone(){ return done_; }
00057
00058
00059 private:
00060     sf::Time time_;
00061     int blastRange_;
00062     bool done_;
00063
00064 };
00065
00066 #endif
```

7.8 freezingTower.hpp

```
00001 #ifndef FREEZING_TOWER
00002 #define FREEZING_TOWER
00003 #include "tower.hpp"
00004 #include "enemy.hpp"
00005 #include <list>
```

```
00006 #include <memory>
00016 class FreezingTower : public Tower{
00017 public:
00023     FreezingTower(sf::Vector2f);
00030     void update(std::list<std::shared_ptr<Enemy>> &enemies, sf::Time time) override;
00036     Projectile* shoot() override;
00041     void upgradeTower() override;
00042 private:
00050     std::list<std::shared_ptr<Enemy>> lockedEnemies_;
00058     float slowCoefficient_ = 0.2;
00059
00060 };
00061 #endif //FREEZING_TOWER
```

7.9 game.hpp

```
00001 #ifndef GAME_HPP
00002 #define GAME_HPP
00003
00004 #include <SFML/Graphics.hpp>
00005 #include <list>
00006 #include "tower.hpp"
00007 #include "path.hpp"
00008 #include "enemy.hpp"
00009 #include "projectile.hpp"
00010 #include "resource_container.hpp"
00011 #include "player.hpp"
00012 #include <memory> //for shared_ptr
00013 #include "bulletTower.hpp"
00014 #include "button.hpp"
00015 #include "map.hpp"
00016 #include "missileProjectile.hpp"
00017 #include "menu.hpp"
00018 #include <vector>
00019 #include "levelManager.hpp"
00020 #include "explosion.hpp"
00021
00022 class Menu;
00023 // Class for running the game logic
00024
00029 class Game {
00030
00031     friend class Tower;
00032     friend class BulletTower;
00033     friend class BombTower;
00034     friend class MissileTower;
00035     friend class FreezingTower;
00036     friend class BombProjectile;
00037     friend class BulletProjectile;
00038     friend class MissileProjectile;
00039     friend class PoisonTower;
00040     friend class Menu;
00041     friend class LevelManager;
00042
00043 public:
00044     Map map;
00045     Game();
00046
00056     void run();
00057
00058     ~Game(){
00059
00060         enemies_.clear();
00061
00062         for(auto i : projectiles_){
00063             delete i;
00064         }
00065         projectiles_.clear();
00066
00067         towers_.clear();
00068
00069         // Menus deleted by unique_ptr
00070     }
00071
00077     path& getPath();
00078 private:
00089     void processEvents();
00090
00096     void update();
00097
00106     void render();
00107
00111     void loadTextures();
```

```
00112
00116     void createPath(); //this will create the path that the enemies will traverse (this should also be rendered visually in the game)
00117
00124     void checkTowers();
00125
00126
00127     void testEnemy();
00128     void testEnemySplit(sf::Vector2f position, std::queue<sf::Vector2f> waypoints);
00129
00136     void updateMenus();
00137
00138     //adding a function to return the elapsed time
00139     sf::Time getTime() const;
00140     //I am adding a clock and time functionality that will need to be used for enemy movement and updating and other game logic
00141     sf::Clock clock_;
00142     sf::Time time_;
00143     sf::RenderWindow window_;
00144
00145     std::list<std::shared_ptr<Tower» towers_;
00146     std::list<std::shared_ptr<Enemy» enemies_;
00147     std::list<Projectile*> projectiles_;
00148     std::list<Explosion*> explosions_;
00149     path path_;
00150
00151     bool dragged_;
00152     bool paused_;
00153     bool isGameOver_=false;
00154     bool isGameFinished_ = false;
00155     sf::Font font_;
00156     sf::Text gameOverText;
00157     sf::Text gameFinishedText;
00158     sf::Sprite castle_sprite_;
00159
00160     std::unique_ptr<Menu> shop_;
00161     std::unique_ptr<Menu> alternativeMenu_;
00162     std::shared_ptr<Tower> activeTower_;
00163     bool menuInactive = false;
00164
00165     ResourceContainer<Textures::TowerID, sf::Texture> tower_textures_;
00166     ResourceContainer<Textures::EnemyID, sf::Texture> enemy_textures_;
00167     ResourceContainer<Textures::ProjectileID, sf::Texture> projectile_textures_;
00168     ResourceContainer<Textures::Various, sf::Texture> various_textures_;
00169
00170     Player player_;
00171
00172     LevelManager levelManager_;
00173 };
00174
00175 #endif
```

7.10 levelManager.hpp

```
00001 #ifndef LEVELMANAGER
00002 #define LEVELMANAGER
00003
00004 #pragma once
00005
00006 #include <iostream>
00007 #include <string>
00008 #include <vector>
00009 #include <map>
00010 #include <variant>
00011 #include <fstream>
00012 #include <sstream>
00013 #include <random>
00014
00015 #include "enemy.hpp"
00016 #include "path.hpp"
00017
00018 class Game;
00019
00023 class LevelManager {
00024
00025     public:
00029     using variantData = std::variant<int, float, std::vector<int»;
00030
00041     LevelManager(const std::string& src, path& path, Game& game, Player& player) : src_(src), path_(path), game_(game), player_(player) {
00042         readLevels();
00043
00044         currLevel_ = 0;
```

```
00045         waitTime_ = 0;
00046         levelTotal_ = levelSpecs_.size();
00047     }
00048     ~LevelManager(){}
00049 
00053     int getCurrentLevel() const;
00054 
00058     int getLevelTotal() const;
00059 
00067     void update();
00068 
00074     bool readingSuccessfull();
00075 
00076     private:
00077 
00086     void readLevels();
00087 
00094     void initiateEnemies();
00095 
00096 
00109     std::vector<std::map<std::string, variantData» levelSpecs_;
00110 
00111     int currLevel_;
00112     const std::string& src_;
00113     bool readingSuccess_;
00114     int levelTotal_;
00115     float waitTime_;
00116 
00117     path& path_;
00118     Game& game_;
00119     Player& player_;
00120 };
00121 
00122 #endif
```

7.11 map.hpp

```
00001 #ifndef MAP_HPP
00002 #define MAP_HPP
00003 
00004 #include <SFML/Graphics.hpp>
00005 #include <memory>
00006 #include <string>
00007 #include <vector>
00008 #include "tower.hpp"
00009 
00010 class Tower; // Forward declaration
00011 
00020 class Map : public sf::Drawable, public sf::Transformable {
00021 public:
00022     sf::Texture texture; // Texture used for the map.
00023     sf::Sprite background; // Sprite representing the map background.
00024     std::vector<sf::FloatRect> unBuildable; // Vector of rectangles representing unbuildable areas on
    the map.
00025 
00031     void loadMap(const std::string& fileName);
00032 
00033 
00034 private:
00040     void draw(sf::RenderTarget& target, sf::RenderStates states) const override;
00041 };
00042 
00043 #endif // MAP_HPP
```

7.12 menu.hpp

```
00001 #ifndef MENU
00002 #define MENU
00003 #include <SFML/Graphics.hpp>
00004 #include <list>
00005 #include "button.hpp"
00006 #include "game.hpp"
00007 #include "tower.hpp"
00008 
00009 // These are used in createMenu()
00010 // the enum determines what type of menu is created:
00011 // Which buttons are added etc.
00012 enum class MenuType{
00013     Shop,
```

```
00014     Upgrade,
00015     Begin,
00016     Level
00017 };
00022 class Menu {
00023 public:
00029     void draw(sf::RenderWindow& window);
00030
00039     void checkButtons(Game* game);
00040
00047     void createMenu(MenuType menu, Game* game);
00048
00056     void update(Player& player);
00057
00069     void drag(Game* game);
00070
00076     void drawRange(Game* game);
00077 private:
00078
00085     void newTower(std::shared_ptr<Tower> tower, Game* game);
00086
00093     bool canBePlaced(Game* game);
00094
00095     std::list<Button> buttons_;
00096     std::vector<sf::Text> texts_;
00097     sf::RectangleShape bg_;
00098 };
00099
00100 #endif
```

7.13 missileProjectile.hpp

```
00001 #ifndef MISSILE_PROJECTILE
00002 #define MISSILE_PROJECTILE
00003
00004 #include "projectile.hpp"
00005
00009 class MissileProjectile : public Projectile
00010 {
00011 private:
00012     std::shared_ptr<Enemy> targetEnemy_;
00013
00014 public:
00021     MissileProjectile(sf::Vector2f position, int damage, std::shared_ptr<Enemy> targetEnemy)
00022     : Projectile(sf::Vector2f(0,0), position, damage, 280.f, "missile", 400),
      targetEnemy_(targetEnemy) {}
00023
00031     bool hasHitEnemy(std::shared_ptr<Enemy>& enemy) override;
00032
00041     void update(Game& game) override;
00042
00046     Textures::ProjectileID textureType() override { return Textures::Missile; }
00047 };
00048
00049
00050 #endif
```

7.14 missileTower.hpp

```
00001 #ifndef MISSILE_TOWER
00002 #define MISSILE_TOWER
00003 #include "tower.hpp"
00004 #include "missileProjectile.hpp"
00012 class MissileTower : public Tower {
00013 public:
00018     MissileTower(sf::Vector2f);
00024     MissileProjectile* shoot() override;
00025 };
00026 #endif
```

7.15 path.hpp

```
00001 #ifndef PATH_HPP
00002 #define PATH_HPP
00003 #include <queue>
```

```
00004 #include <SFML/System/Vector2.hpp>
00005 #include <SFML/Graphics.hpp>
00006 #include <vector>
00007 #include <random>
00008
00009 class path {
00010     friend class enemy;
00011 public:
00018     path(const std::string& src) : src_(src) {
00019         readPath();
00020
00021         std::random_device rd;
00022         std::uniform_int_distribution<int> range(0, paths_.size()-1);
00023
00024         auto gamePath = paths_[range(rd)];
00025
00026         for(const auto& point: gamePath){
00027             addWaypoint(point);
00028         }
00029
00030     }
00031
00032     ~path() {}
00033
00039     void readPath();
00040
00045     bool readingSuccessfull();
00046
00050     void addWaypoint(const sf::Vector2f& point);
00051
00055     std::queue<sf::Vector2f> getWaypoints() const;
00056
00061     void makeUnBuildablePath();
00062
00063     static const float width;
00064     std::queue<sf::Vector2f> waypoints_;
00065     std::vector <sf::Vector2f> wayPoints;
00066     std::vector <sf::FloatRect> unBuildable;
00067
00068     std::vector<std::vector<sf::Vector2f» paths_;
00069
00070 private:
00071     const std::string& src_;
00072     bool readingSuccess_;
00073
00074 };
00075
00076 #endif
```

7.16 player.hpp

```
00001 #ifndef PLAYER
00002 #define PLAYER
00003
00004 #include <string>
00005 #include <list>
00006 #include "enemy.hpp"
00007 #include "tower.hpp"
00008 #include <SFML/System/Vector2.hpp>
00009 #include <SFML/Graphics/Transformable.hpp>
00010 #include <memory>
00011 #include "resource_container.hpp"
00012
00013 class Tower;
00014 class Enemy;
00015
00022 class Player : public sf::Sprite
00023 {
00024     private:
00025         int hp_;
00026         int wallet_;
00027         int level_;
00028
00029     public:
00036         Player() : hp_(500), wallet_(1000), level_(0){}
00037
00038         ~Player() {}
00039
00043         int getWallet() const;
00044
00048         int getHP() const;
00049
00053         int getLevel() const;
```

```
00054
00058         void levelUp();
00059
00064         void addMoney(int amount);
00065
00070         void removeMoney(int cost);
00071
00076         void removeHP(int amount);
00077 };
00078
00079 #endif
```

7.17 poisonTower.hpp

```
00001 #ifndef POISON_TOWER
00002 #define POISON_TOWER
00003 #include "tower.hpp"
00004 #include "enemy.hpp"
00005 #include <list>
00006 #include <memory>
00017 class PoisonTower : public Tower{
00018 public:
00024     PoisonTower(sf::Vector2f);
00031     void update(std::list<std::shared_ptr<Enemy>> &enemies, sf::Time time) override;
00037     Projectile* shoot() override;
00038 private:
00046     std::list<std::shared_ptr<Enemy>> lockedEnemies_;
00047 };
00048 #endif
```

7.18 projectile.hpp

```
00001 #ifndef PROJECTILE
00002 #define PROJECTILE
00003
00004 #include "tower.hpp"
00005 #include "player.hpp"
00006 #include "enemy.hpp"
00007 #include "resource_container.hpp"
00008 #include <SFML/System/Vector2.hpp>
00009 #include <SFML/Graphics/Transformable.hpp>
00010 #include <SFML/Graphics.hpp>
00011 #include <memory>
00012 #include <iostream>
00013
00014 class Game;
00015 class Enemy;
00016
00020 class Projectile : public sf::Sprite
00021 {
00022     private:
00023         float speed_;
00024         std::string type_;
00025         int damage_;
00026         sf::Vector2f position_; // of tower that created
00027         float maxDistance_;
00028         sf::Vector2f shootDirection_;
00029         bool isDestroyed_;
00030
00031     public:
00032
00042         Projectile(sf::Vector2f shootDirection, sf::Vector2f position, int damage, float speed,
      std::string type, float maxDistance)
00043         : shootDirection_(shootDirection), position_(position), damage_(damage), speed_(speed),
      type_(type), maxDistance_(maxDistance),
00044         isDestroyed_(false){
00045             this->setPosition(position_);
00046         }
00047
00051         virtual ~Projectile() {}
00052
00056         float getSpeed() const;
00057
00061         const std::string& getType() const;
00062
00066         int getDamage() const;
00067
00071         sf::Vector2f getShootDir() const;
00072         //sf::Vector2f getVelocity() const;
```

```
00073
00078         void destroy();
00079
00083         bool isDestroyed();
00084
00089         bool distToTower();
00090
00094         virtual bool hasHitEnemy(std::shared_ptr<Enemy>&) = 0;
00095
00099        virtual void update(Game&) = 0;
00100
00105        virtual Textures::ProjectileID textureType() = 0;
00106 };
00107 #endif
00108
```

7.19 resource_container.hpp

```
00001 #ifndef RESOURCE_CONTAINER
00002 #define RESOURCE_CONTAINER
00003 #include <SFML/Graphics.hpp>
00004 #include <string>
00005 #include <memory>
00006
00007 // Enums for different textures
00008 namespace Textures{
00009
00010     // NOTE: these could also be stored in one big enum...
00011     enum TowerID {BulletTower, BombTower, MissileTower, FreezingTower, PoisonTower};
00012     enum EnemyID {Enemy1, Enemy2, Enemy3, Enemy4, Enemy5};
00013     enum ProjectileID{Bullet, Bomb, Missile};
00014     enum Various {Pause, Castle, Dirt, Upgrade, Sell, Continue};
00015 }
00016
00023 template <typename T_enum, typename T_resource>
00024 class ResourceContainer {
00025 public:
00026
00033     void load(T_enum type, std::string filename){
00034         std::unique_ptr<T_resource> resource(new T_resource());
00035
00036         if (!resource->loadFromFile(filename)){
00037             //TODO: Handle texture loading error
00038         }
00039         // The function move should avoid creating a copy of the object recource, when inserting it
      into the map
00040         resources_.insert(std::make_pair(type, std::move(resource)));
00041         }
00042
00050     T_resource& get(T_enum type) const {
00051         auto wanted = resources_.find(type);
00052         return *wanted->second;
00053     }
00054
00055
00056 private:
00057     std::map<T_enum, std::unique_ptr<T_resource» resources_;
00058
00059
00060 };
00061
00062 #endif
```

7.20 tower.hpp

```
00001 #ifndef TOWER_H
00002 #define TOWER_H
00003 #include <string>
00004 #include <array>
00005 #include <SFML/System/Vector2.hpp>
00006 #include <SFML/System/Clock.hpp>
00007 #include <SFML/Graphics.hpp>
00008 #include "projectile.hpp"
00009 #include "enemy.hpp"
00010 #include <memory>
00011
00012
00013 class Projectile;
00024 class Tower : public sf::Sprite {
```

```
00025 public:
00026     Tower(sf::Vector2f position, const std::string& type,  int baseCost, float range, sf::Time
      fireRate,
00027           int damage, int upgradeCost);
00028     const std::string& getType() const {return type_;}
00029     const int getBaseCost() const {return baseCost_;}
00030     sf::Time getFireRate() const {return fireRate_;}
00031     const float getRange() const {return range_;}
00032     int getDamage() const {return damage_;}
00033     std::shared_ptr<Enemy> getLockedEnemy() const {return lockedEnemy_;}
00034     bool isMaxLevelReached() const {return maxLevelReached_;};
00035     int getCurrentLvl() const {return currentLvl_;}
00036     const int getUpgradeCost() const {return upgradeCost_;};
00037     sf::Time getFireTimer() {return fireTimer_;}
00038     bool enemyWithinRange(std::shared_ptr<Enemy> enemy);
00044     virtual Projectile* shoot() = 0;
00052     virtual void upgradeTower();
00057     virtual void update(std::list<std::shared_ptr<Enemy>> &enemies, sf::Time time);
00058     void updateFireTimer(sf::Time &dt);
00059     void setLevel(int level) {currentLvl_ = level;}
00060     void setMaxLevelFlag() {maxLevelReached_ = true;}
00061     void setLockedEnemy(std::shared_ptr<Enemy> enemy) {lockedEnemy_ = enemy;}
00062     void resetFireTimer() {fireTimer_ = sf::Time::Zero;}
00075 private:
00076     const std::string type_;
00077     const int baseCost_;
00078     const float range_;
00079     int damage_;
00080     int currentLvl_;
00081     const int upgradeCost_;
00082     std::shared_ptr<Enemy> lockedEnemy_;
00083     sf::Time fireTimer_;
00084     sf::Time fireRate_;
00085     bool maxLevelReached_;
00086 };
00087 #endif //TOWER_H
```

Index