Silesian University of Technology

Computer Programming

Project: Worms Clone

Silesian University of Technology

Faculty of Automatic Control, Electronics and Computer Science

author	Dawid Grobert
instructor	mgr inż. Wojciech Dudzik
year	2020/2021

1 Topic description

1.1 Introduction

Worms are turn-based artillery games presented in 2D environment (few version were made in 3D and 2.5D). Each player has his own worm team consisting of X worms. Subsequently each player in his turn can move one worm from his team (there is no choice here, they are chosen in turn). Each worm has its own arsenal of weapons that it can replenish over the course of the game. To win a player have to defeat all opponent's worms.

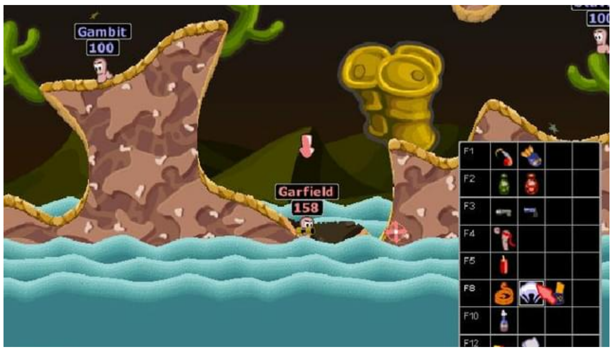


Figure 1: An exemplary *Worms 2* screenshot

In the original worms, the terrain is gradually flooded by water forcing the worms to flee up the map, and the terrain is fully destructible by an arsenal of worm weapons.

1.2 History

Worms 2 was released in 1997 as a sequel to the game *Worms*, which makes us assume that its creation took nearly 2 years most likely based on the code of Worms 1. These games were produced by the British studio *Team17* and are sometimes categorised as an *artillery game* in addition to a *turn-based arcade game*.

Artillery game is a genre of game that was usually designed for two or more players – usually with turn-based game mechanics. These games usually had objects that tried to destroy each other with projectiles running in a parabolic curve. Often these games also had destructible terrain.

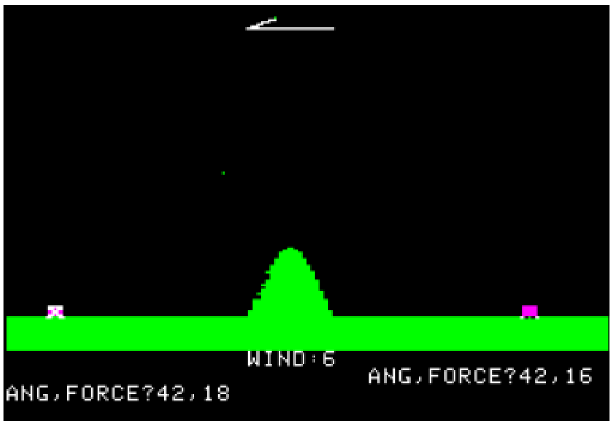
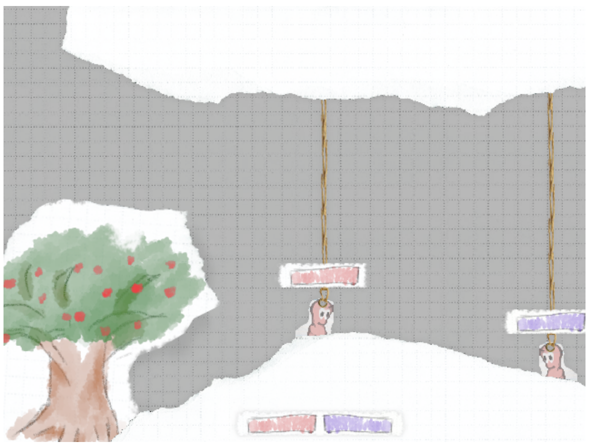


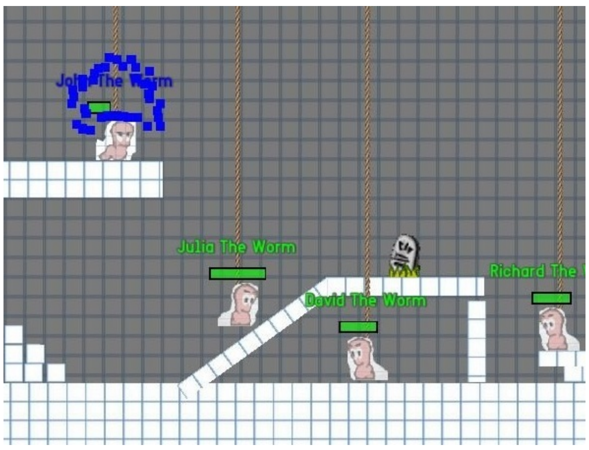
Figure 2: Artillery Simulator for the Apple II

1.3 My version of Worms

My version of the game is intended to resemble a paper theatre. The initial sketch of the game differs a little from the final effect. This can be seen in the images below.



Early sketch of the game



The end result of the game

The squareness of the editor's map objects has prevented it from fully capturing the spirit of paper theatre, but it at least tries to resemble paper cut-outs.

Most importantly, however, it has been preserved. The worms still have their equipment and the arsenal available in it. The worms can move around the map and, most importantly, shoot at each other. Some objects (not all) are fully destructive. We can say that the spirit of Worms has been somewhat preserved, but it just lacks the polish and production time that the original had.

Even with a small arsenal of weapons, there is still nothing to stop two/three teams from getting together and starting a battle between each other on a self-made map. The game is fully playable and can be started and completed.

1.3.1 Editor

It's worth mentioning here that at the production stage, the game additionally got a map editor, which allows player to create a map at the his own discretion. The editor was not initially foreseen.

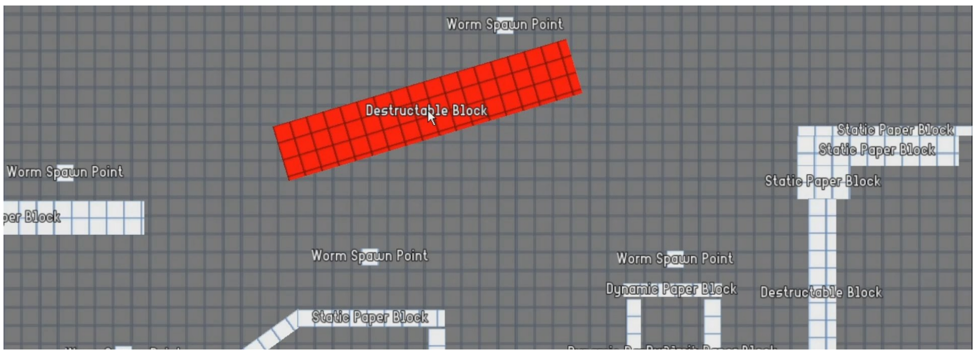


Figure 3: In-game world editor

1.4 Source of knowledge

My main knowledge comes from the documentation of individual libraries. In the process of learning (started some time ago) I also read two books: *C++ Primer 5th edition* and *SFML Game Development*. I also get a lot of help from the official SFML forum. There are many helpful threads there.

2 Topic Analysis

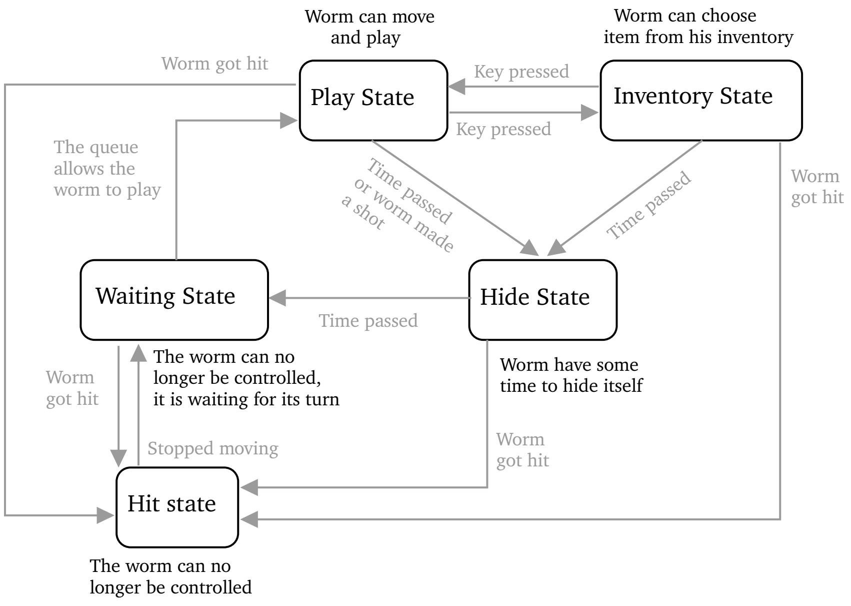
Even before proceeding with the implementation, I was able to set up a couple of mechanisms that should be inside the game.

2.1 The worm

The worm object has the ability to take on a great many states. It is quite a complex object and only one worm can be controlled at a time. However, this does not make other worms completely disabled as they can still be hit. This means roughly that they should physically react – that is, be moved (pushed back) when they are hit. They should then be dealt damage. However, they should not be able to be moved by the other worm. To sum up:

1. A worm whose turn has not occurred should not be able to be moved by a worm that has its turn. This would cause the worms to be pushed by the player.
2. A worm whose turn has not occurred cannot be fully blocked, as still hitting should move it. However, after some time after the hit it should be blocked again so that the player cannot move it.

It will look like this: a worm whose waiting for turn will be locked in the X axis. This will ensure that it is not moved by another worm, but in the event that the ground under its feet is removed it will still fall down. In the event of an given damage the worm will be unlocked, pushed away and when it is feet on the ground and its speed is low enough it will be locked again.



Worm can move and play
Worm can choose item from his inventory
Worm got hit
Play State
Key pressed
Inventory State
Key pressed
The queue allows the worm to play
Time passed or worm made a shot
Time passed
Worm got hit
Waiting State
Hide State
Time passed
The worm can no longer be controlled, it is waiting for its turn
Worm have some time to hide itself
Worm got hit
Stopped moving
Worm got hit
Hit state
The worm can no longer be controlled

Figure 4: Worm states shown as *Finite State Machine*

2.1.1 Problem of jumping

Another problem with this worm is that it must be able to jump. That is, I need to know if it is touching the ground from which it could bounce and then allow it to jump. I can't just check if he's colliding with something at the current moment and allow him to jump, because he could jump by touching the ceiling with his head. Therefore, I will create a special collision check block in his feet. This block will intentionally be shorter than the width of the worm, so that he cannot jump by constantly bouncing off the wall. More details can be found in the *Internal Specification*[1].

[1]4.6. Solving the jumping problem

2.2 The Application

It is important to remember that, in a way, the game is only one state. The application consists of many more intermediate states such as:

- The Title
- The Menu
- Pause
- Many mores such as settings, information about authors

In the case of my application, there will be several such states and they are best represented as a *Finite State Machine*.

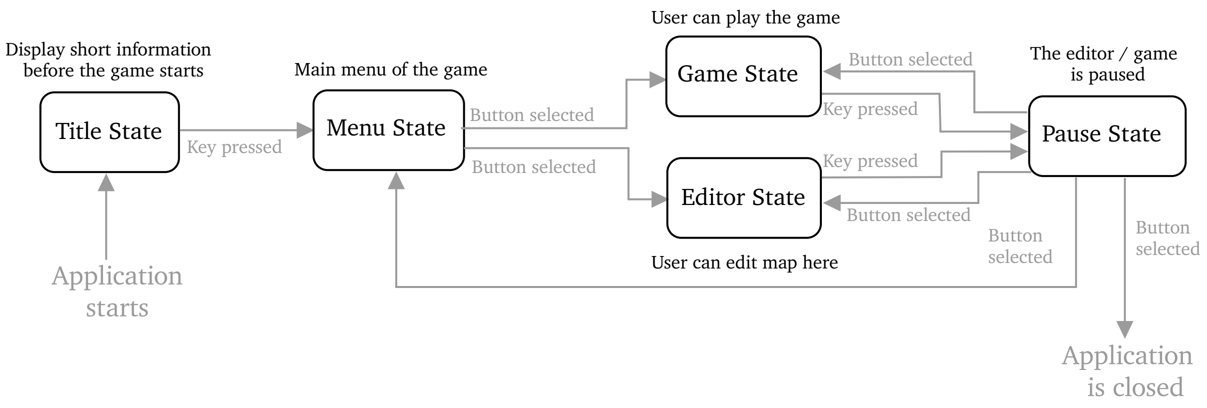


Figure 5: Flow of the application shown as *Finite State Machine*

As can be seen in the above figure (Fig. 5) the application flow resembles a typical flow for a game. The title with the studio logo, the main menu with the possibility to configure the game and to start the game. The possibility to pause and to quit the game. I would like to mention here two states:

- **Game State** allows to play directly in the game. This is the target state in which the game starts and can be paused and resumed at any time. *Pause State* blocks updates to the *Game State*, so it is effectively stopped.
- **Editor State** is another very powerful part of the program that allows to edit the map. Place different objects, rotate them, save them, and finally in the *Game State* play on the newly created map.

This space has been intentionally left blank.

2.3 Worm Queue

The turn order must be set up in such a way that each team can make two moves regardless of how many worms they have. Worms can die at any time and be removed from the list and still the order of moves cannot be disturbed. For this purpose, I present the following structure:

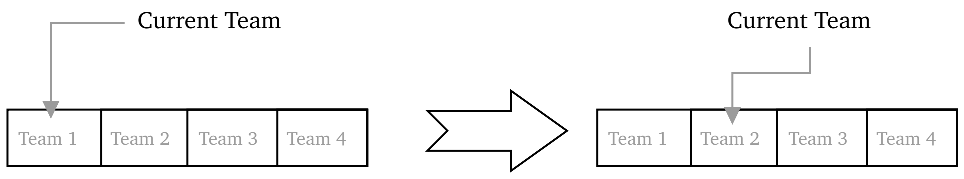


Figure 6: In general, the sequence of movements presented

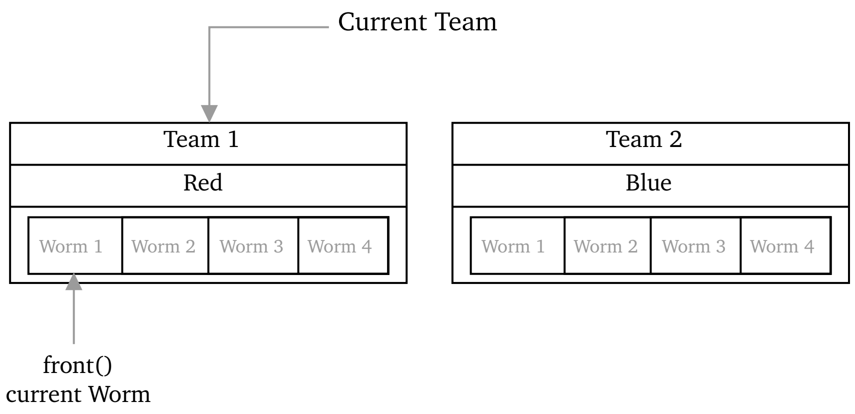


Figure 7: Queue of moves during a turn

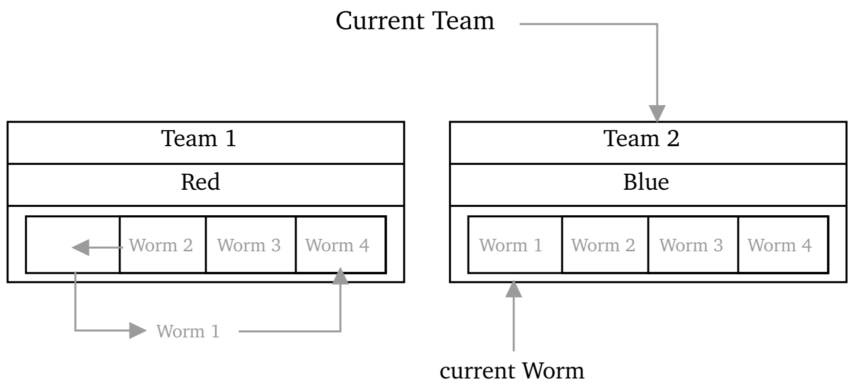


Figure 8: Queue of moves during a turn change.

I think it is a good enough structure to handle it inside the game. The problem of doing this with worms inside a typical container is that it would be quite unreadable and difficult, as it would require to repair the queue on the fly with each worm death.

2.4 Libraries Used

My project turned out to be quite large, so it needed as many as four external libraries.

2.4.1 Simple and Fast Multimedia Library (SFML)

SFML is free and open-source cross-platform software development library designed to create games and multimedia programs. Thanks to five modules that SFML provides:

- System – that handles time and threads,
- Window – that handles windows, and interaction with the user,
- Graphics – that handles rendering graphics,
- Audio – that provides interface for playing music and sounds,
- Network – that handles TCP and UDP netowrk sockets, and data encapsulation facilities.

the game development can be much simpler using the high-level language. In my game SFML is the basis on which the code is based. With this library I display all objects in the game, and handle user input.

2.4.2 Box2D

Box2D is a two-dimensional physics engine. This engine is limited to the simulation of rigid solids. The basic unit of computation for length is the meter and mass is the kilogram. This means that I have to convert meters to pixels and vice versa on the fly. I display objects using the SFML library.

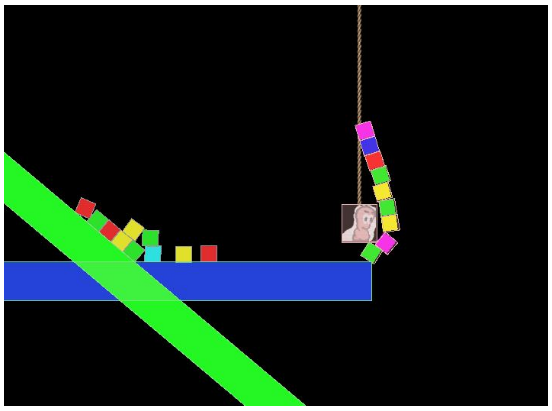


Figure 9: Exemplary use of Box2D inside a very early version of the game.

2.4.3 Clipper

The Clipper Library can perform clipping, and offsetting of lines, and polygons. The library is based on *Vatti's clipping algorithm*. The library is used to create destructible objects.

2.4.4 Poly2Tri

Poly2Tri allows to turn polygons into triangles. Many graphics libraries including SFML allow to create and draw only convex figures. Because of this, it does not allow to draw more complex figures, including concave ones. For this reason, the initial object will be broken into smaller polygons, and these will undergo a triangulation process to draw final triangles which, when joined together, will form a figure.

3 External specification

The program is created using a graphical library, so everything user need to play comes down to running the executable file (.exe). Sample gameplay can be watched on the recording available under the QR code. QR Code is also a clickable link, so everyone can click on it. Included in the recording is an example of creating a map, setting up the game in terms of the number of teams and worms, and gameplay leading to a victory for one of the teams.

Video

Exemplary gameplay

3.1 How to play

In this section I will describe step by step all the necessary information on how to play this game. Subsections are broken down into individual application states such as *map editor*, *game* or *menu*.

Info: The camera inside the game can be moved using the middle mouse button. It can also be zoomed in/out using the scroll wheel.

3.1.1 Menu

Inside the menu and in many other cases, the buttons visible on the screen are clickable when hovering the mouse. Some of them can be clicked only with the left button, and some additionally with the right button

How many worms per team?
4 worms per team
How many teams?
2 teams
How many worms per team?
4 worms per team
How many teams?
2 teams

The button zooms in when hovered over

We set the settings for teams and worms using:

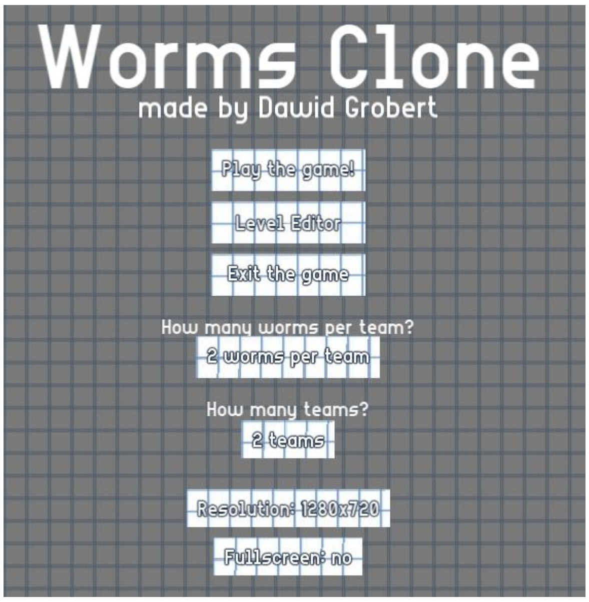
- **Left Mouse Button** when we want to decrease its value
- **Right Mouse Button** when we want o increase its value

How many worms per team?
4 worms per team
How many teams?
2 teams
How many worms per team?
5 worms per team
How many teams?
2 teams

The right mouse button increases the value

How many worms per team?
4 worms per team
How many teams?
2 teams
How many worms per team?
2 worms per team
How many teams?
2 teams

The left mouse button decreases the value



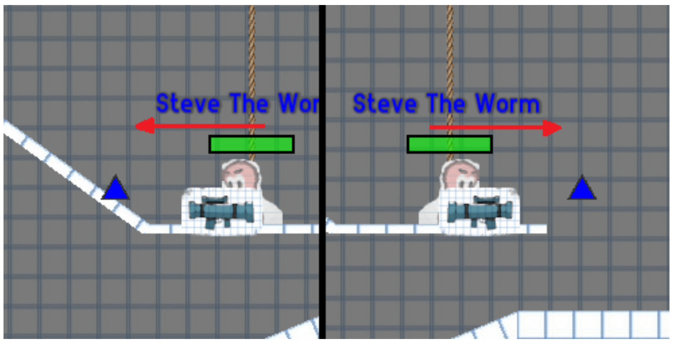
The first three buttons in turn allow:

- **Play the game!** – starts the game with the current settings and the current map inside the *Level editor*.
- **Level Editor** – allows to edit the current map and save its status. Changes are saved even after the game is closed.
- **Exit the game** – Ends program operation

The next buttons allows to configure game settings.

- **The number of teams** can be set between 2 and 4.
- **The number of worms inside one team** can be set between 2 and 6 worms.
- **Available resolutions** are: `1024x768`, `1280x720`, `1366x768`, `1680x1050`, `1920x1080`.
- **The full screen** can be switched between 'yes' and 'no'.

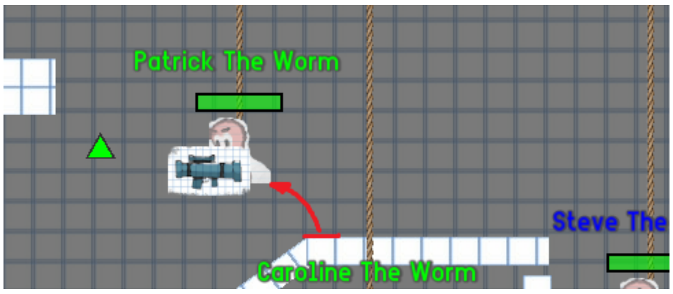
3.1.2 In-Game controls



Moving the worm

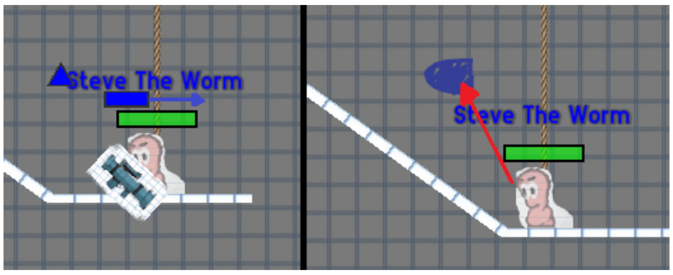
Two buttons are needed to move the worm around the map.

- The **A** button makes the worm **move to the left**.
- The **D** button makes the worm **move to the right**.



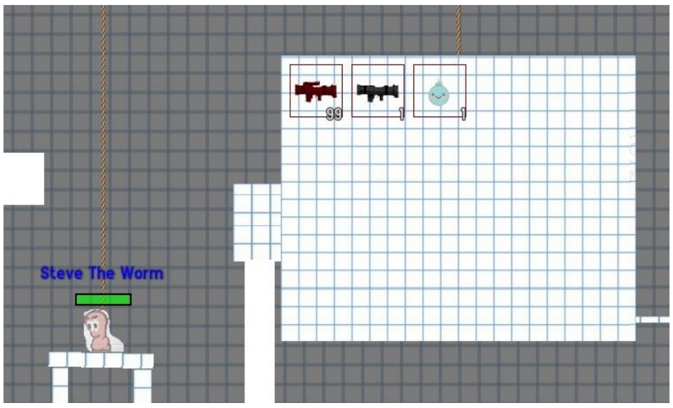
Jumping

To jump, **simply press the spacebar**. The worm will jump in the direction it is facing.



Shooting – to get rid of your enemies

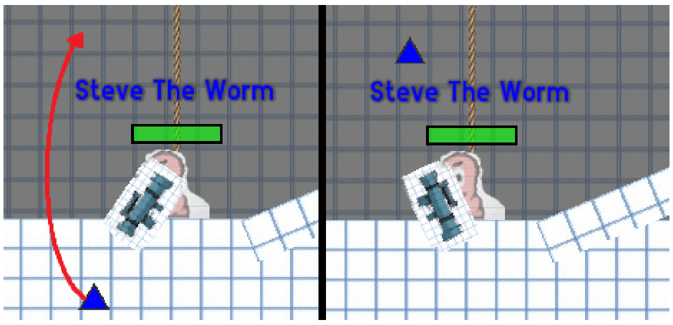
To shoot, simply hold down the ENTER. Holding the enter down long enough increases the power with which the bullet will be fired. The higher the power, the further and faster the bullet will fly.



Equipment – weapon selection and weapon change

If there is a **need to check inventory or change weapon during a round**, it is possible **by pressing the E button**.

To the right of the worm, his equipment will slide out at his height. The available number of uses is displayed next to each weapon. **Weapons can be selected/changed using the left mouse button.**

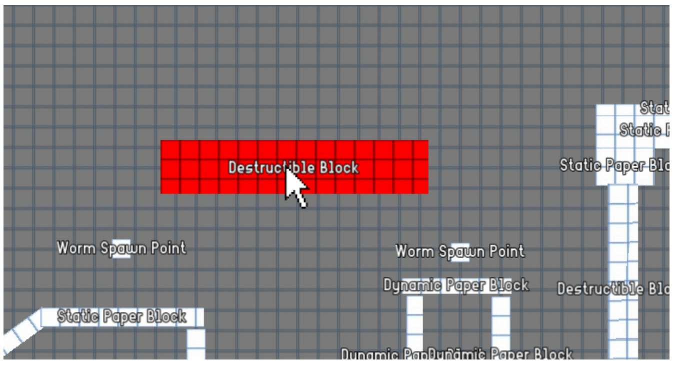


Aiming

Two buttons are needed to aim.

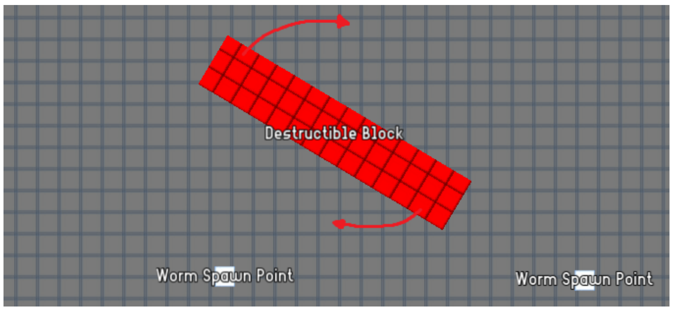
- The left button (←) rotates the pointer in clockwise manner.
- The right button (→) rotates the pointer in counter–clockwise manner.

3.1.3 Editor



Moving the objects

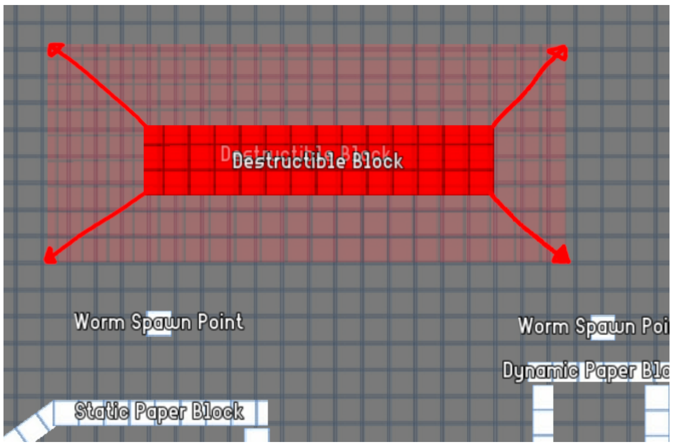
To move objects within the editor, simply **hover the mouse over them and hold down the left mouse button** to move them as desired.



Rotating the objects

To rotate objects inside the editor you use two buttons:

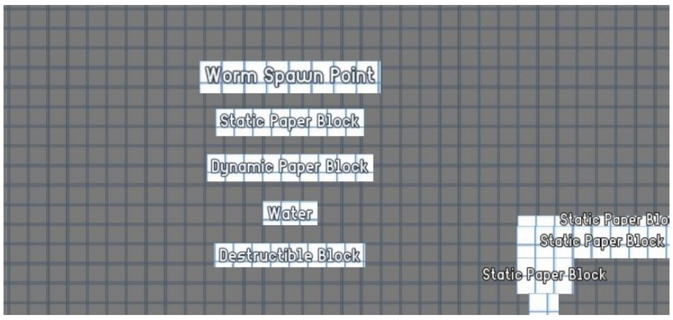
- **Q button** rotates object **to the left**.
- **E button** rotates object **to the right**.



Scaling the objects

Objects can be resized by using four buttons

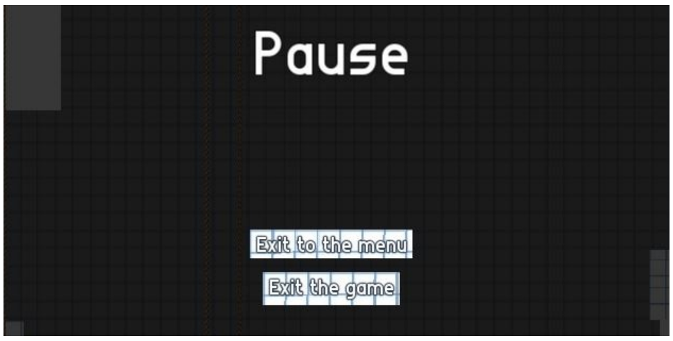
- Use the **left arrow** button (←) to reduce the width of the object.
- The **right arrow** button (→) increases the width of the object.
- The height of the object is decreased using the **upper arrow** button (↑).
- The height of the object is increased by the **bottom arrow** button (↓).



Creating the objects

Objects can be created with **the right mouse button**. A menu of available objects is then displayed from which it is possible to select the object to be created. The object is selected with **the left mouse button.**

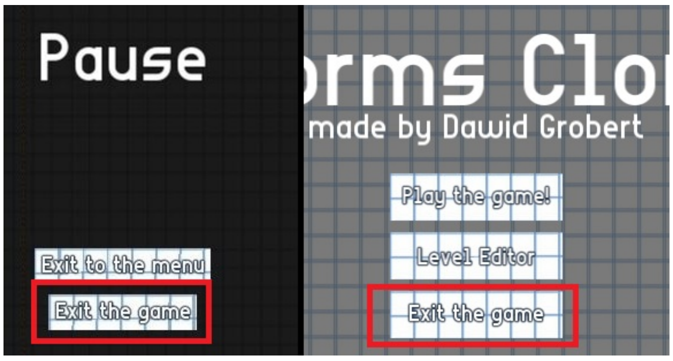
3.1.4 How to pause



Pausing the game

To pause the application (whether in the editor or in the middle of the game), **simply press the ESCAPE key (ESC).** Then, using the **left mouse button**, we can confirm one of the available options, **or return back by clicking ESC again**.

3.1.5 How to quit

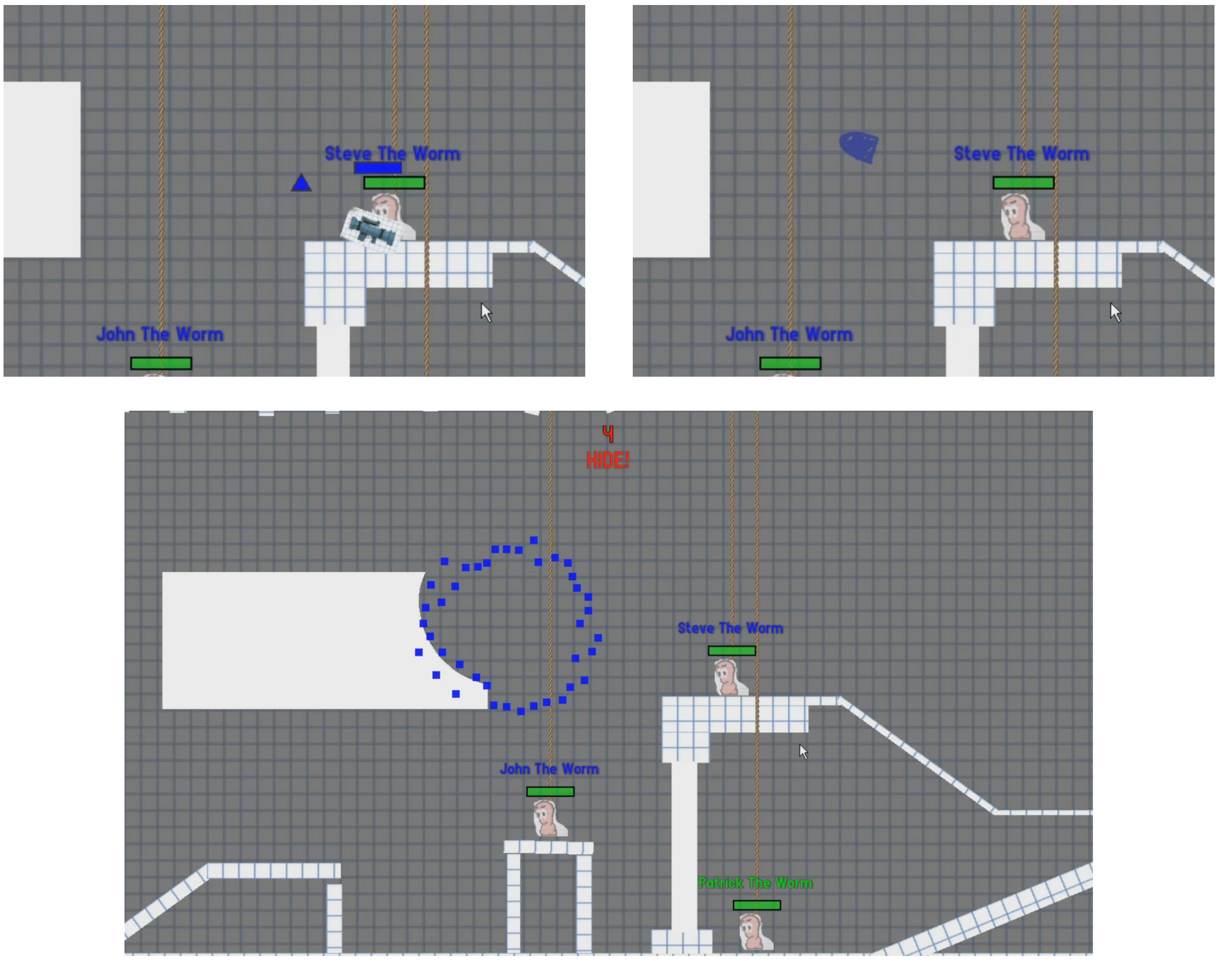


Exiting the game

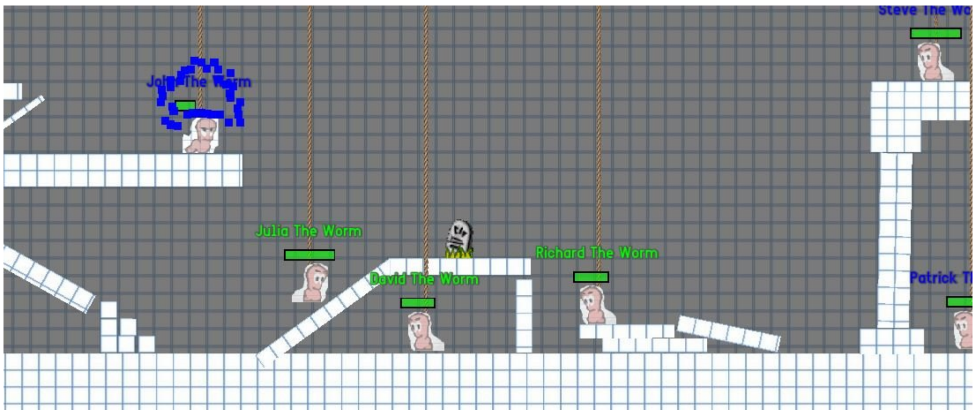
It is possible to quit the game in two ways.

- By selecting the available button during pause (see *3.1.4. How to pause*).
- By selecting an available button in the main menu.

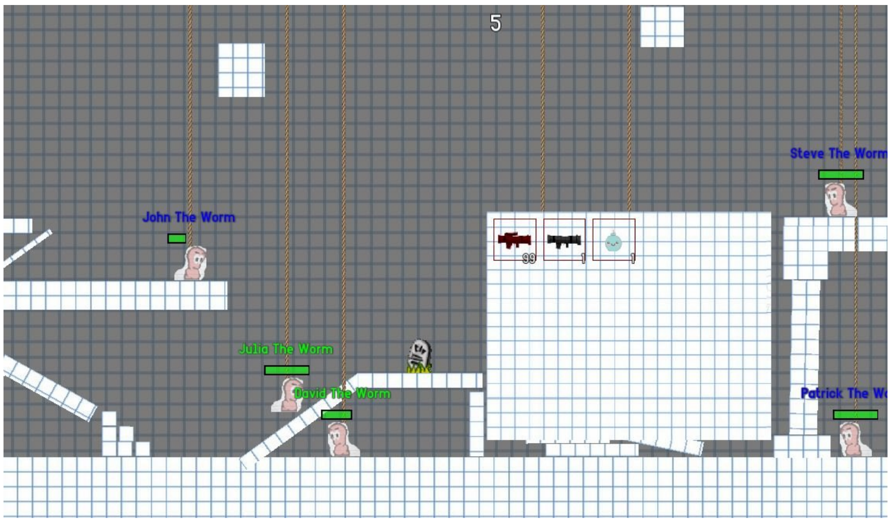
3.2 Screenshots of the game



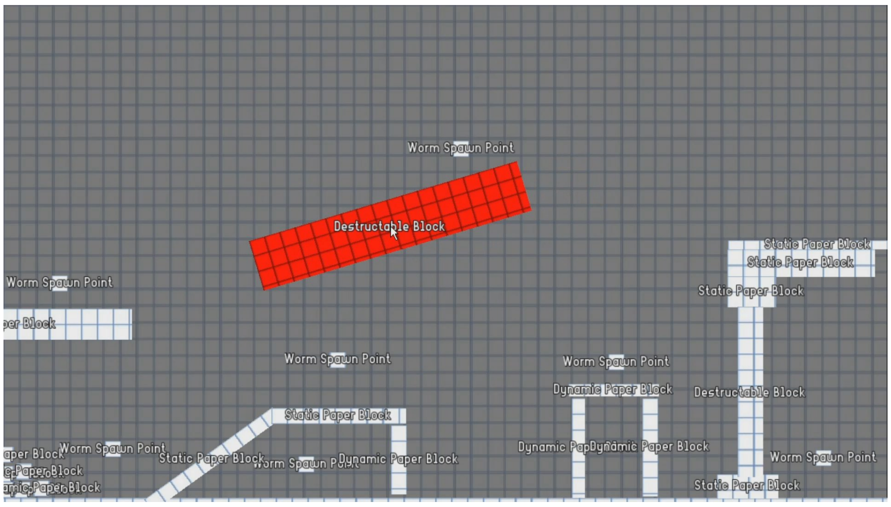
Destruction system



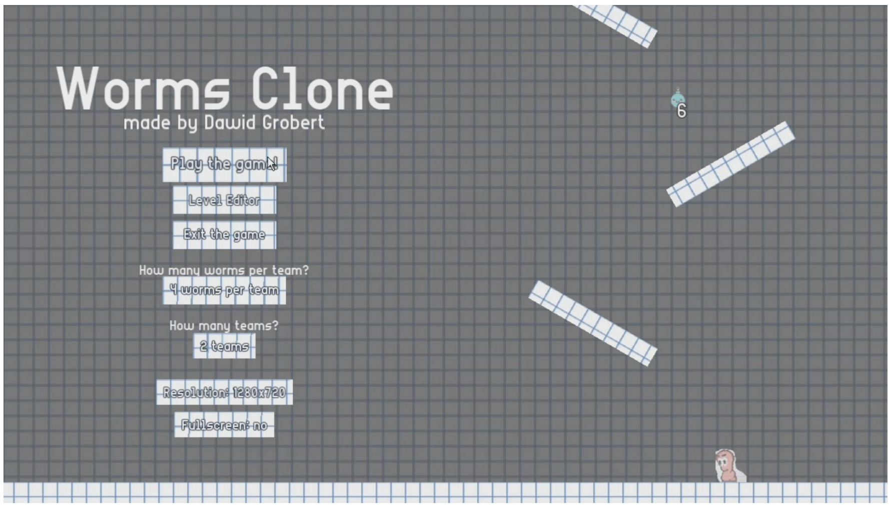
Exemplary gameplay



In-game inventory



Map-Editor



Main Menu

4 Internal specification

Many games use some well-known mechanisms that I will also implement in my project.

4.1 Scenes & Nodes (like in Godot Engine)

This mechanism is supposed to resemble something in the Godot Engine called "Scenes & Nodes". A bit much to say, as it mainly resembles a simplified version of it.

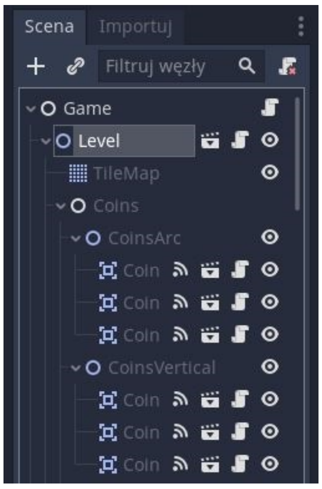


Figure 10: Tree hierarchy of nodes in Godot Engine

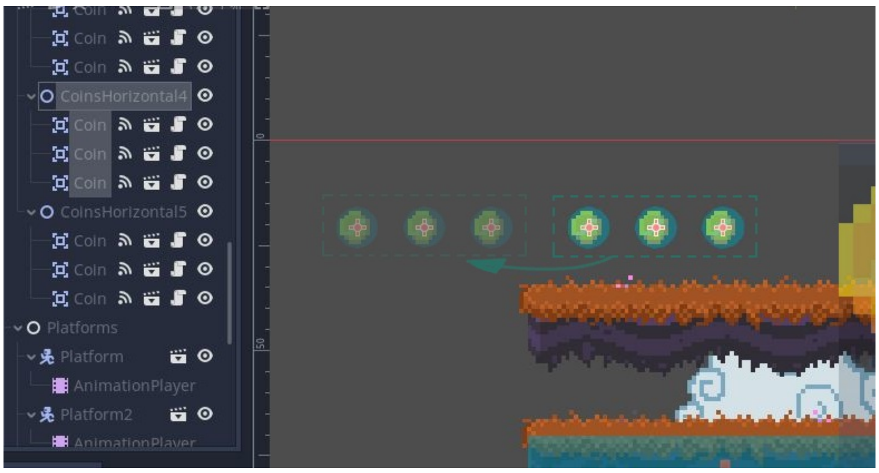


Figure 11: By moving parent node we move all pinned nodes in Godot Engine

Individual nodes (objects inside the game) can be parents of other nodes. Thus, moving the parent will make the relativistic position of the nodes (children) pinned to it also to change. We can observe this in Fig 11. This makes it very easy to create any hierarchies and scenes within the game. Nodes that are pinned to a parent node at the time of drawing receive *transform* of node they are pinned to, so that they can draw themselves relative to their parent.

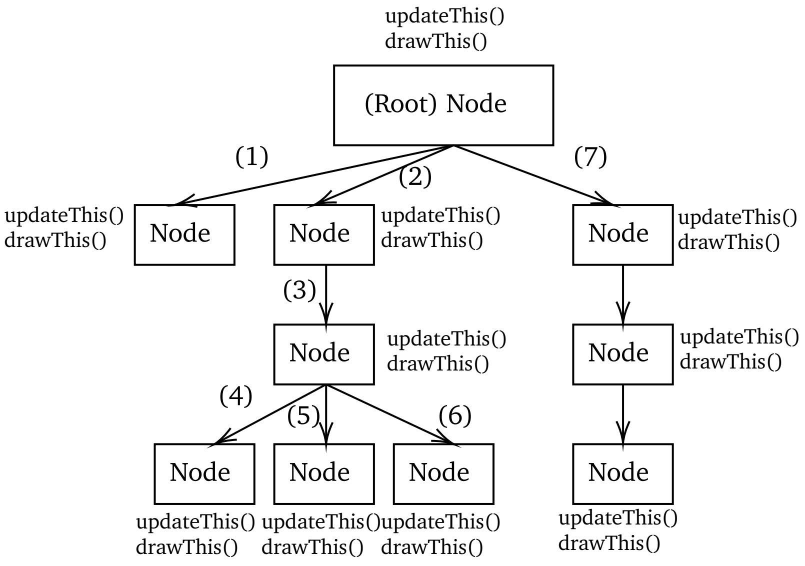


Figure 12: The order and method of displaying and updating individual nodes.

We can say that the way the nodes are executed resembles "Depth-first search". It starts at the root node, and it explores as far as possible along each branch before backtracking. This also means that objects higher up in the hierarchy (parents) will be drawn on the screen underneath their pinned nodes (children).

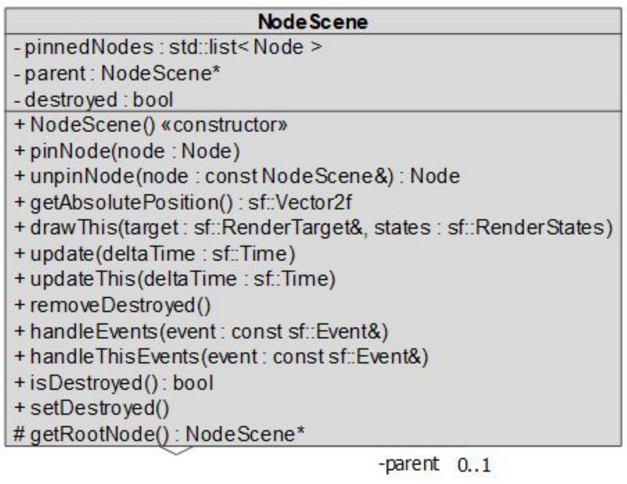

NodeScene
- pinnedNodes : std::list< Node >
- parent : NodeScene*
- destroyed : bool
+ NodeScene() «constructor»
+ pinNode(node : Node)
+ unpinNode(node : const NodeScene&) : Node
+ getAbsolutePosition() : sf::Vector2f
+ drawThis(target : sf::RenderTarget&, states : sf::RenderStates)
+ update(deltaTime : sf::Time)
+ updateThis(deltaTime : sf::Time)
+ removeDestroyed()
+ handleEvents(event : const sf::Event&)
+ handleThisEvents(event : const sf::Event&)
+ isDestroyed() : bool
+ setDestroyed()
# getRootNode() : NodeScene*
-parent 0..1

Figure 13: The base class from which other nodes will inherit

Similar systems also appear in other engines and probably many games because of their simplicity and usefulness.

4.2 Statestack

Statestack is a mechanism that appears in the vast majority of games. For larger SFML-based games, Statestack is already a sort of tradition, as it is then implemented in some form (depending on needs) almost always.

As can be seen from the given figure (see Fig. 14) statestack is used to control the flow of the game.

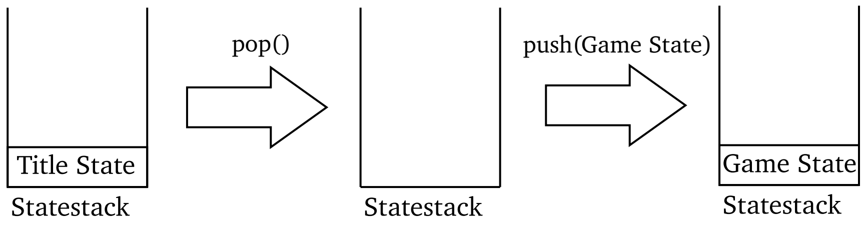
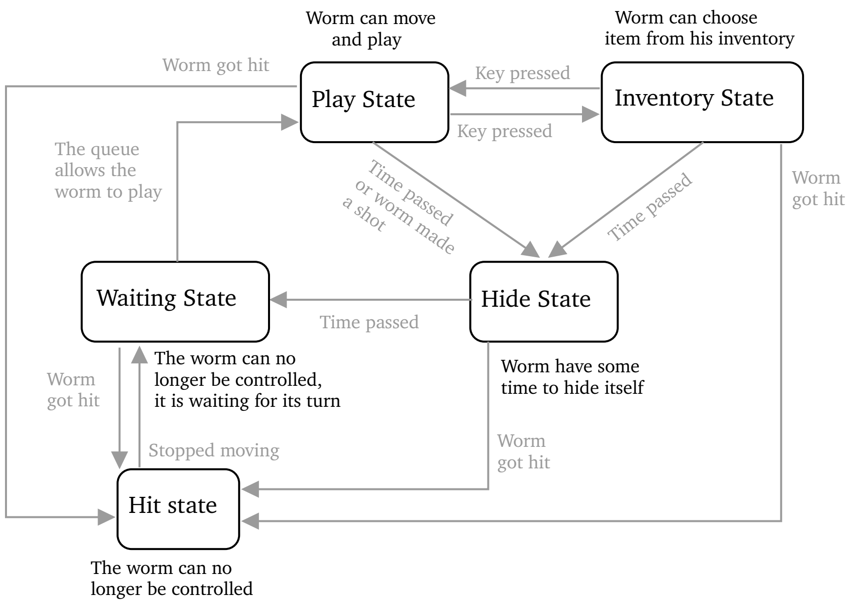


Figure 14: Changing state from one to the other

Change between states not only is possible in the context of the application itself, but also, for example, in the context of a single object. For this game, StateStack was also used to implement the worm object. It allowed the diagram below (see Fig, 15) to be reproduced very faithfully.



Worm can move and play
Worm can choose item from his inventory
Worm got hit
Play State
Key pressed
Inventory State
Key pressed
The queue allows the worm to play
Time passed or worm made a shot
Time passed
Worm got hit
Waiting State
Hide State
Time passed
The worm can no longer be controlled, it is waiting for its turn
Worm have some time to hide itself
Worm got hit
Stopped moving
Worm got hit
Hit state
The worm can no longer be controlled

Figure 15: Statestack flow of the worm shown as Finite State Machine

I've given examples of changing state to other state, but I haven't yet mentioned the most important aspect of Statestack – namely, the genesis of its name, specifically the "stack" part.

Statestack allows consecutive states to overlap and in its basic version, only the state at the top of the stack is executed. This allows to block certain functions of the program (states) by pushing other state and then restore them seamlessly by using *pop()* on the stack.

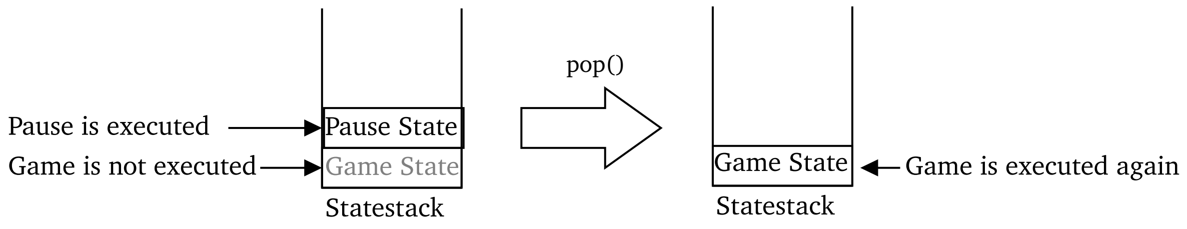


Figure 16: An example of how pause can stop processes running inside the game.

Some Statestacks additionally implement the ability to make certain states "transparent". These are states that do not block some of the layers below and allow them to execute. This is the implementation I use and that takes away the ability to use *std::stack*. For this reason in my implemenation I use a *std::vector*.

```cpp
void StateStack::update(sf::Time deltaTime)
{
    // Performs all queued operations / changes
    applyChanges();

    // Iterate from the highest state to the lowest state,
    // and stop iterating if any state returns false
    for (auto state = stack.rbegin(), end = stack.rend(); state != end; ++state)
    {
        // If a state is of "Transparent" type, then it returns true
        // But if state returns false, then it stop iterating through the lower layers
        if (!(*state)->update(deltaTime))
            return;
    }
}
```

Figure 17: Example of how StateStack updates its states

Please note that *Statestack* have the *update()* function, which can, for example, call the *update()* function of the *GameState*. In turn, this function can perform an *update()* for a *Root node*, and as we know from **4.1. Scenes & Nodes (like in Godot Engine)** this one will call *update()* for itself and its children. This gives us a great hierarchy of calls of the game world.

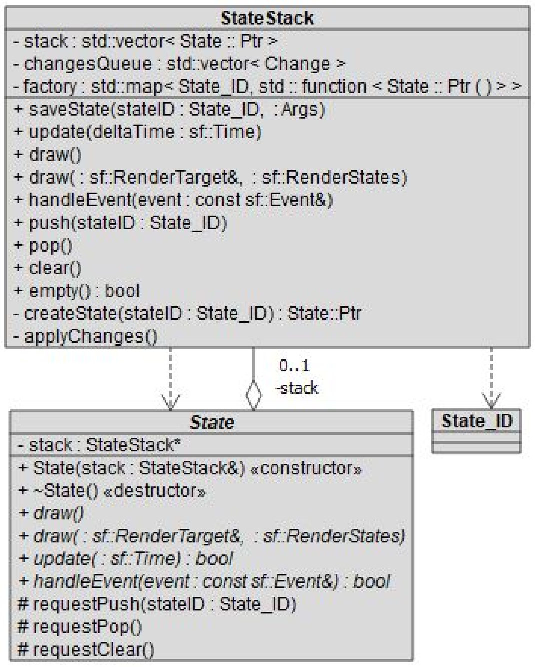

StateStack
- stack : std::vector< State :: Ptr >
- changesQueue : std::vector< Change >
- factory : std::map< State_ID, std :: function < State :: Ptr ( ) > >
+ saveState(stateID : State_ID, : Args)
+ update(deltaTime : sf::Time)
+ draw()
+ draw(: sf::RenderTarget&, : sf::RenderStates)
+ handleEvent(event : const sf::Event&)
+ push(stateID : State_ID)
+ pop()
+ clear()
+ empty() : bool
- createState(stateID : State_ID) : State::Ptr
- applyChanges()
0..1
-stack
State
- stack : StateStack*
+ State(stack : StateStack&) «constructor»
+ ~State() «destructor»
+ draw()
+ draw(: sf::RenderTarget&, : sf::RenderStates)
+ update(: sf::Time) : bool
+ handleEvent(event : const sf::Event&) : bool
# requestPush(stateID : State_ID)
# requestPop()
# requestClear()
State_ID

Figure 18: The implementation of the Statestack

4.2.1 The change queue

The *changeQueue* (previously used in update of the stack, see Fig. 17) was added due to the fact that when one state decides to remove itself from the stack using pop() then it loses the ability to push out another state immediately. For this reason I use the queue to delay this operation until the beginning of the next iteration during which the *applyChanges()* function performs all stored operations on the queue. This allows a state to pop itself from the stack and push another state onto the stack.

4.2.2 The factory

It would be too expensive to hold objects of all states at once. For this reason, *Statestack* is also a kind of factory that creates and removes states when they are no longer needed.

Each state must be properly saved. Information such as variables needed to create the object are stored in the class creating the state and must persist for the duration of the existence of these states. Pointer to the stack object is passed too, so the state can request various operations.

```cpp
template <typename State, typename... Args>
void StateStack::saveState(State_ID stateID, Args&&... args)
{
        // args holds all the variables and data that state might need to use
        factory[stateID] = [&args..., this]()
        {
                return State::Ptr(std::make_unique<State>(*this, args...));
        };
}
```

Figure 19: Example of how a function that saves states looks like

```
// Register
appStack.saveState<PauseState>(State_ID::PauseState, gameWindow);

// Later in code:
if(event.type == sf::Event::KeyPressed && event.key.code == sf::Keyboard::Escape)
        requestPush(State_ID::PauseState);
```

Figure 20: Exemplary usage

When the queue is checked for any new changes at the beginning of an iteration, it will fulfill the request to push new object onto the stack (see Fig. 20). Before pushing the object onto the stack, a special function creates the object by translating the registered identifier, to the lambda that returns the ready object (see Fig. 19). It is done using the map of identifiers to lambdas returning *std::unique_ptr* to the new object. This function is of course called to create the object.

```
State::Ptr StateStack::createState(State_ID stateID)
{
    auto found = factory.find(stateID);
    assert(found != factory.end());

    return found->second();
}
```

Of course, if such an object is popped from the stack, thanks to the fact that *std::unique_ptr* follows the RAII design principle, the object is safely removed.

4.3 General scheme of the program operation

Like it was shown in **4.1. Scenes & Nodes (like in Godot Engine)** the root node updates and draws the nodes pinned to it, and the nodes attached to them do the same to nodes pinned to them. As we also saw in **4.2. Statestack**, the statestack updates and draws states on its stack. All this is controlled by a loop in a main function that makes sure that objects are updated only 60 times per second. All objects follow a mathematical formula:

$$distance = speed \cdot time$$

which makes sure that the speed of objects is not dependent on how powerful the computer is. For example, if we were to move an object by 1 pixel every one iteration of the loop (1 pixel per frame) – then if the computer can do 160 iterations per second (160 frames per second), it will move the object farther in one second than a less powerful computer that can do 90 iterations per second (90 frames per second). To avoid making the distance dependent on the number of frames, all things related to distance are multiplied by time.

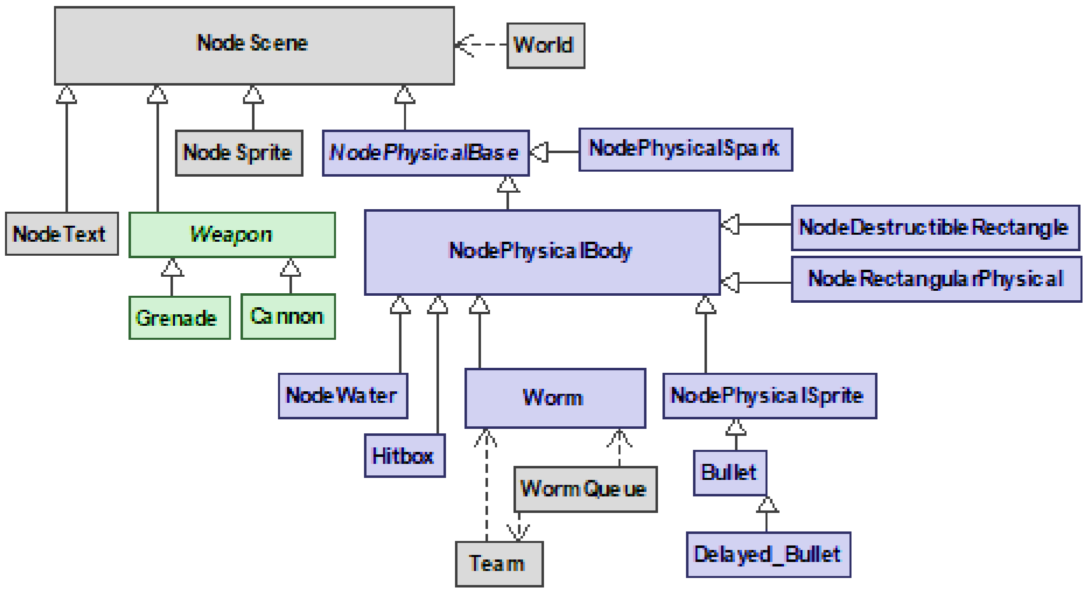
Algorithm 1: `Main loop`

```
Initialize $time\_per\_frame$ with 1/60 of second
Initialize $frame\_time\_elapsed$ with 0 seconds
Start $clock$ with 0 seconds
while Game Window is opened do
    $frame\_time\_elapsed \leftarrow frame\_time\_elapsed + clock$
    Restart the $clock$
    while $frame\_time\_elapsed > time\_per\_frame$ do
        $frame\_time\_elapsed \leftarrow frame\_time\_elapsed - time\_per\_frame$
        Update the game using $time\_per\_frame$ as a time
    end
    Render the Game
end
```

4.4 Class Diagram

The most important two diagrams will revolve around, of course, the two most elaborate elements: *Node-scene*, and *States*

4.4.1 NodeScene inheritance hierarchy



Hierarchy of the class Nodescene

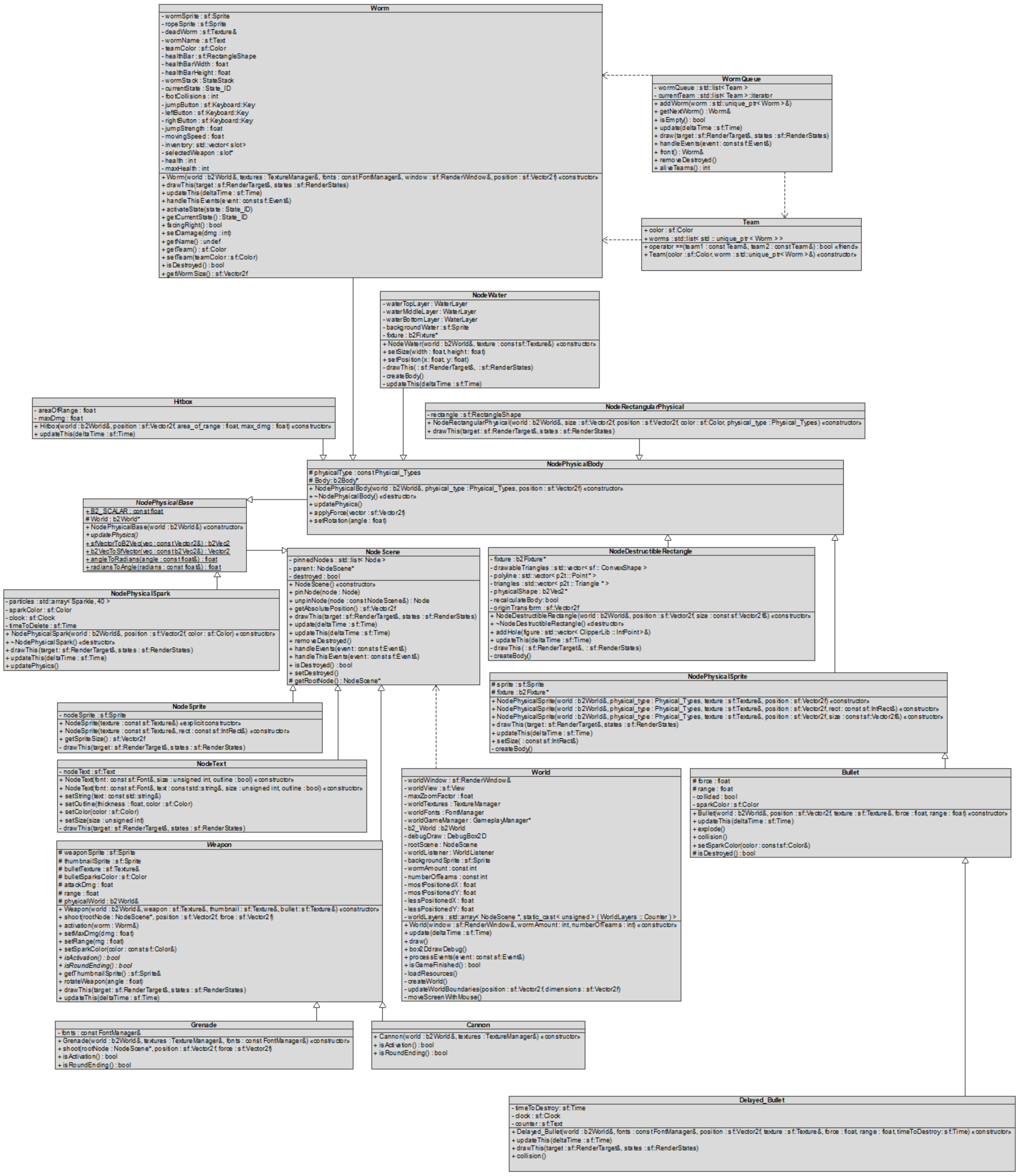
In the above diagram, it can be seen the hierarchy of classes that inherit from NodeScene. Classes related to the physics of the game are highlighted in blue. They derive from *NodePhysicalBase*, which contains very basic data such as functions to convert units between meters and pixels, and a pointer to the game's physical world. Most classes inherit from *NodePhysicalBody*, which adds the physical body and physical types of the object. Of course, *NodeScene* also inherits from the SFML library classes. For the purity of the diagram this is not included, but it does inherit from three classes:

- *sf::Drawable* – as it is suppossed to be drawn on the screen.
- *sf::Transformable* – which gives all members related with position, rotation and scale
- *sf::NonCopyable* – as NodeScene should not be copied

A full diagram is hard to present in this format. It has been inserted a page below, but requires a large zoom. In addition, you can find it by scanning the QR Code next to this text or by clicking on the QR Code with a mouse.

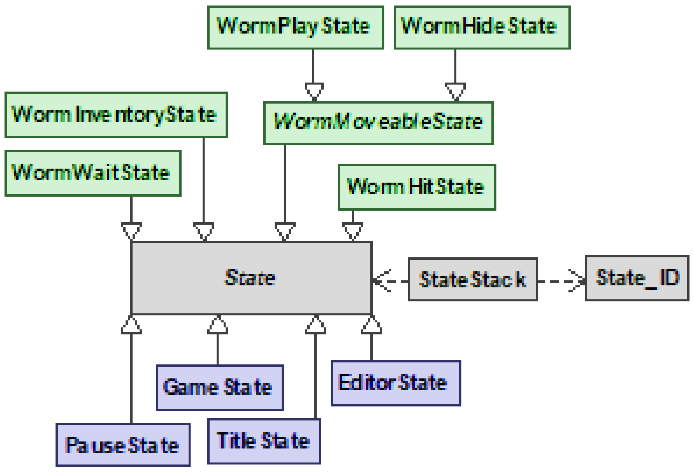
Image

Full NodeScene diagram



NodeScene inheritance hierarchy diagram

4.4.2 States inheritance hierarchy



Hierarchy of the class States

In the diagram, the states are divided into two parts:

- The green part is the states that the worm can have
- The blue part is the states that are used for flow of application

As it can be seen, the same *Statestack* can be used both to create the application flow and even to create the states that an object can have inside the game. A worm is a very complex class, so it needed something to effectively separate it into smaller parts.

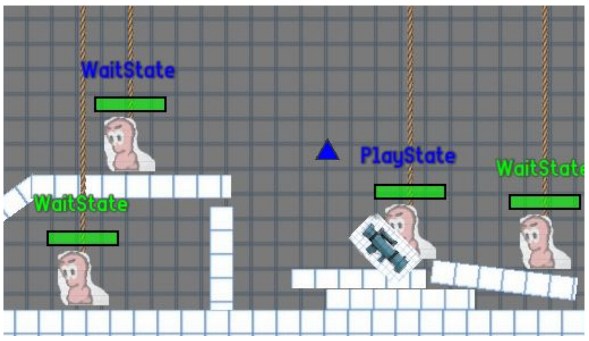


Figure 21: States of worms during the game

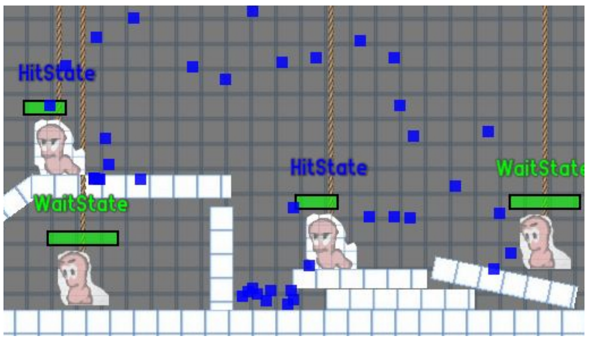


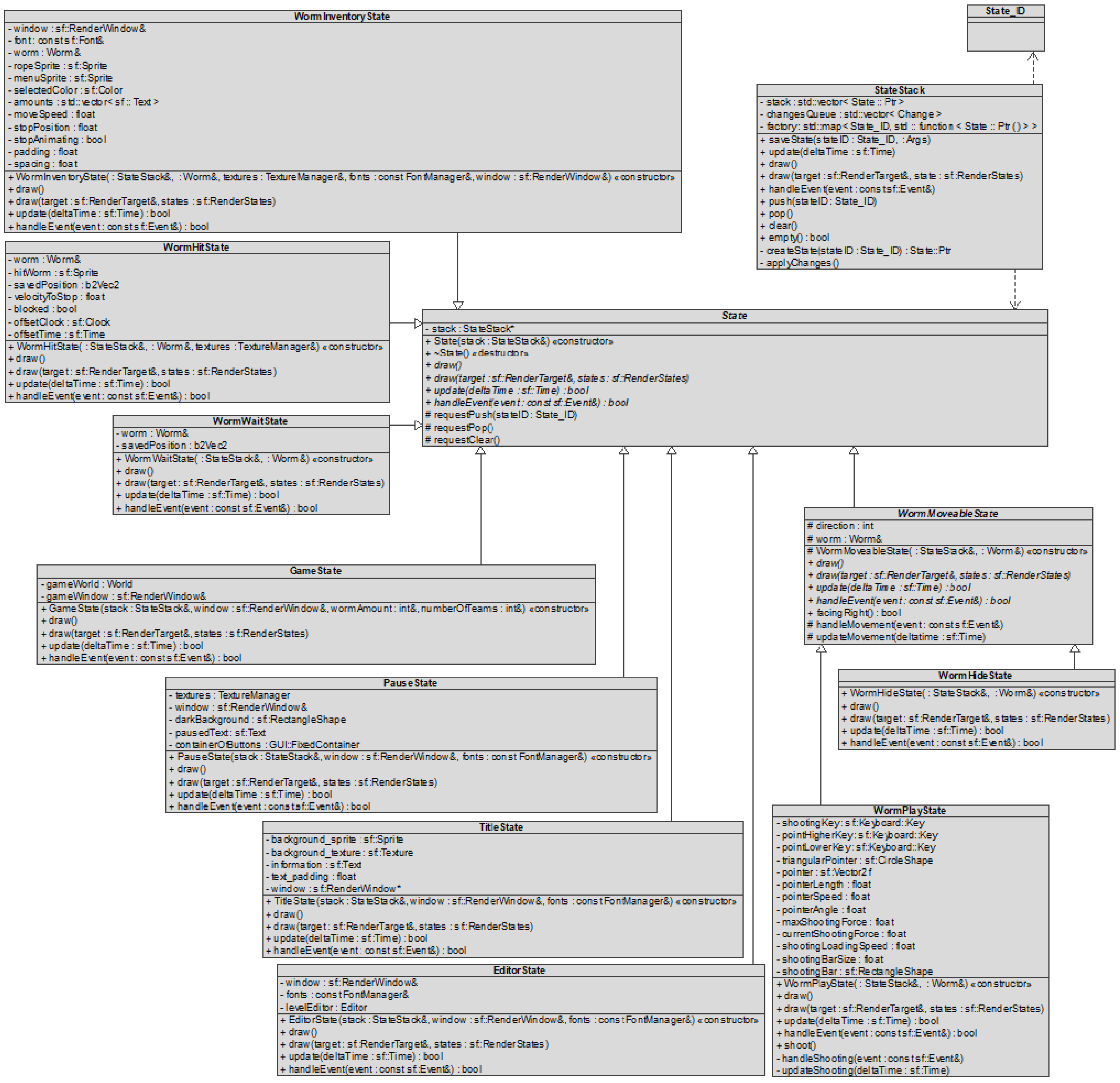
Figure 22: States of worms after impact

For this reason, Statestack found an ideal application to handle such a complex class. An example of how it works can be seen in the images above (see Fig. 21 and Fig. 22).

A full diagram is hard to present in this format. It has been inserted a page below, but requires a large zoom. In addition, you can find it by scanning the QR Code next to this text or by clicking on the QR Code with a mouse.

Image

Full States diagram

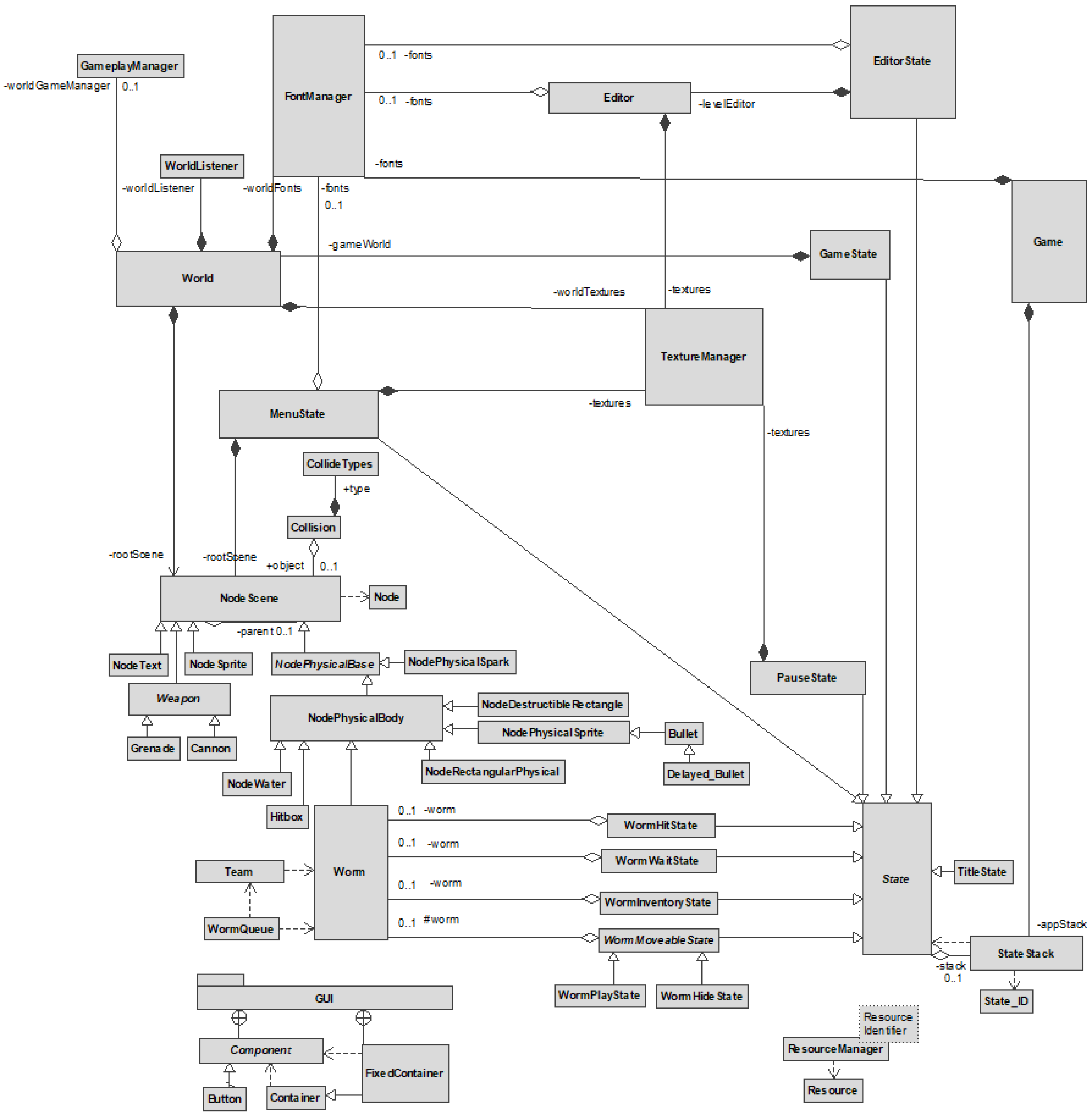


WormInventoryState
- window : sf::RenderWindow&
- font: const sf::Font&
- worm : Worm&
- ropeSprite : sf::Sprite
- menuSprite : sf::Sprite
- selectedColor : sf::Color
- amounts : std::vector< sf :: Text >
- moveSpeed : float
- stopPosition : float
- stopAnimating : bool
- padding : float
- spacing : float
+ WormInventoryState(: StateStack&, : Worm&, textures : TextureManager&, fonts : const FontManager&, window : sf::RenderWindow&) «constructor»
+ draw()
+ draw(target : sf::RenderTarget&, states : sf::RenderStates)
+ update(deltaTime : sf::Time) : bool
+ handleEvent(event: const sf::Event&) : bool
State_ID
StateStack
- stack : std::vector< State :: Ptr >
- changesQueue : std::vector< Change >
- factory: std::map< State_ID, std :: function < State :: Ptr () > >
+ saveState(stateID : State_ID, : Args)
+ update(deltaTime : sf::Time)
+ draw()
+ draw(target : sf::RenderTarget&, state : sf::RenderStates)
+ handleEvent(event : const sf::Event&)
+ push(stateID : State_ID)
+ pop()
+ clear()
+ empty() : bool
- createState(stateID : State_ID) : State::Ptr
- applyChanges()
WormHitState
- worm : Worm&
- hitWorm : sf::Sprite
- savedPosition : b2Vec2
- velocityToStop : float
- blocked : bool
- offsetClock : sf::Clock
- offsetTime : sf::Time
+ WormHitState(: StateStack&, : Worm&, textures : TextureManager&) «constructor»
+ draw()
+ draw(target : sf::RenderTarget&, states : sf::RenderStates)
+ update(deltaTime : sf::Time) : bool
+ handleEvent(event : const sf::Event&) : bool
State
- stack : StateStack*
+ State(stack : StateStack&) «constructor»
+ ~State() «destructor»
+ draw()
+ draw(target : sf::RenderTarget&, states : sf::RenderStates)
+ update(deltaTime : sf::Time) : bool
+ handleEvent(event : const sf::Event&) : bool
# requestPush(stateID : State_ID)
# requestPop()
# requestClear()
WormWaitState
- worm : Worm&
- savedPosition : b2Vec2
+ WormWaitState(: StateStack&, : Worm&) «constructor»
+ draw()
+ draw(target : sf::RenderTarget&, states : sf::RenderStates)
+ update(deltaTime : sf::Time) : bool
+ handleEvent(event : const sf::Event&) : bool
WormMoveableState
# direction : int
# worm : Worm&
# WormMoveableState(: StateStack&, : Worm&) «constructor»
+ draw()
+ draw(target : sf::RenderTarget&, states : sf::RenderStates)
+ update(deltaTime : sf::Time) : bool
+ handleEvent(event : const sf::Event&) : bool
+ facingRight() : bool
# handleMovement(event : const sf::Event&)
# updateMovement(deltatime : sf::Time)
GameState
- gameWorld : World
- gameWindow : sf::RenderWindow&
+ GameState(stack : StateStack&, window : sf::RenderWindow&, wormAmount : int&, numberOfTeams : int&) «constructor»
+ draw()
+ draw(target : sf::RenderTarget&, states : sf::RenderStates)
+ update(deltaTime : sf::Time) : bool
+ handleEvent(event: const sf::Event&) : bool
WormHideState
+ WormHideState(: StateStack&, : Worm&) «constructor»
+ draw()
+ draw(target : sf::RenderTarget&, states : sf::RenderStates)
+ update(deltaTime : sf::Time) : bool
+ handleEvent(event : const sf::Event&) : bool
PauseState
- textures : TextureManager
- window : sf::RenderWindow&
- darkBackground : sf::RectangleShape
- pausedText : sf::Text
- containerOfButtons : GUI::FixedContainer
+ PauseState(stack : StateStack&, window : sf::RenderWindow&, fonts : const FontManager&) «constructor»
+ draw()
+ draw(target : sf::RenderTarget&, states : sf::RenderStates)
+ update(deltaTime : sf::Time) : bool
+ handleEvent(event : const sf::Event&) : bool
WormPlayState
- shootingKey : sf::Keyboard::Key
- pointHigherKey : sf::Keyboard::Key
- pointLowerKey : sf::Keyboard::Key
- triangularPointer : sf::CircleShape
- pointer : sf::Vector2f
- pointerLength : float
- pointerSpeed : float
- pointerAngle : float
- maxShootingForce : float
- currentShootingForce : float
- shootingLoadingSpeed : float
- shootingBarSize : float
- shootingBar : sf::RectangleShape
+ WormPlayState(: StateStack&, : Worm&) «constructor»
+ draw()
+ draw(target : sf::RenderTarget&, states : sf::RenderStates)
+ update(deltaTime : sf::Time) : bool
+ handleEvent(event : const sf::Event&) : bool
+ shoot()
- handleShooting(event : const sf::Event&)
- updateShooting(deltaTime : sf::Time)
TitleState
- background_sprite : sf::Sprite
- background_texture : sf::Texture
- information : sf::Text
- text_padding : float
- window : sf::RenderWindow*
+ TitleState(stack : StateStack&, window : sf::RenderWindow&, fonts : const FontManager&) «constructor»
+ draw()
+ draw(target : sf::RenderTarget&, states : sf::RenderStates)
+ update(deltaTime : sf::Time) : bool
+ handleEvent(event : const sf::Event&) : bool
EditorState
- window : sf::RenderWindow&
- fonts : const FontManager&
- levelEditor : Editor
+ EditorState(stack : StateStack&, window : sf::RenderWindow&, fonts : const FontManager&) «constructor»
+ draw()
+ draw(target : sf::RenderTarget&, states : sf::RenderStates)
+ update(deltaTime : sf::Time) : bool
+ handleEvent(event : const sf::Event&) : bool

States inheritance hierarchy diagram

4.4.3 General Class Diagram

Due to the excessive complexity of the application, this diagram has been simplified for readability.

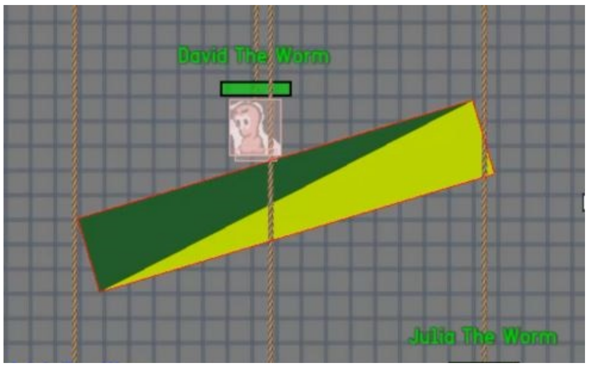


General class diagram

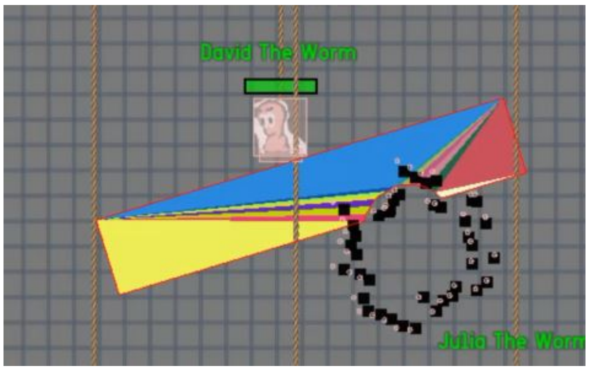
4.5 Destructible Objects

The game features so-called „destructible objects" that can be destroyed using an arsenal of worms. I am using two libraries here: *Clipper*, and *Poly2Tri*.

In my implementation of this problem I store the points forming a figure. The holes created in it are cut out using the *Clipper* library. However, the problem arises when displaying the figure. *SFML* does not allow to display concave figures, it supports only convex figures. For this purpose, the library *Poly2Tri* turned out to be helpful, which transforms the figure created by *Clipper* into triangles, which are drawn by *SFML* on the screen. Each such transformation also updates the physical shell of the object.



Object just before the impact (with triangles building an object)



Object just after the impact (with colored triangles building an object)

4.6 Solving the jumping problem

The problem contained in **Section 2.1.1. Problem of jumping** was solved by using a special sensor to check the worm's contact with the ground. The sensor is smaller than the real width of the worm to prevent it from jumping by sticking to walls.

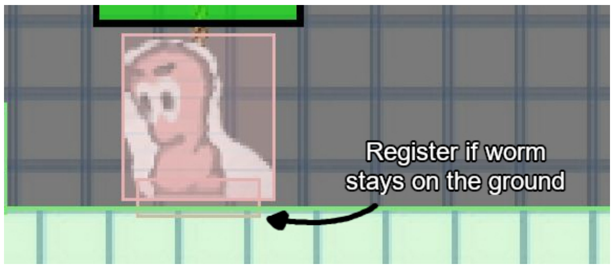


Figure 23: Sensor used to check contact with the ground

4.7 Two types of containers

My game offers two types of containers holding GUI objects (just buttons at the moment).

- **Container** – A standard container that allows to place buttons **inside** the game world. These are buttons that take up space in the game world like any other object, so moving the camera will not make the button follow it.
- **Fixed Container** – It is a container that occupies a special place on the screen. This means that if user move the camera inside the game, the button will move with the camera.

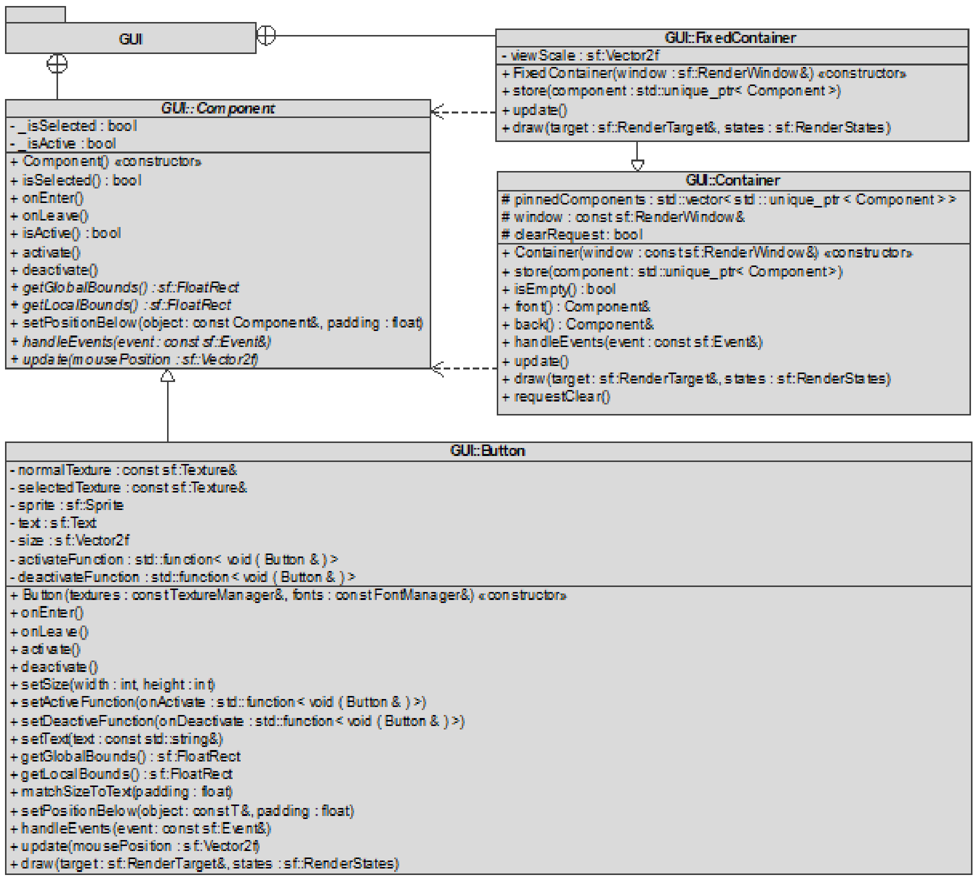

GUI
GUI::FixedContainer
- viewScale : sf::Vector2f
+ FixedContainer(window : sf::RenderWindow&) «constructor»
+ store(component : std::unique_ptr< Component >)
+ update()
+ draw(target : sf::RenderTarget&, states : sf::RenderStates)
GUI::Component
- _isSelected : bool
- _isActive : bool
+ Component() «constructor»
+ isSelected() : bool
+ onEnter()
+ onLeave()
+ isActive() : bool
+ activate()
+ deactivate()
+ getGlobalBounds() : sf::FloatRect
+ getLocalBounds() : sf::FloatRect
+ setPositionBelow(object : const Component&, padding : float)
+ handleEvents(event : const sf::Event&)
+ update(mousePosition : sf::Vector2f)
GUI::Container
# pinnedComponents : std::vector< std :: unique_ptr < Component > >
# window : const sf::RenderWindow&
# clearRequest : bool
+ Container(window : const sf::RenderWindow&) «constructor»
+ store(component : std::unique_ptr< Component>)
+ isEmpty() : bool
+ front() : Component&
+ back() : Component&
+ handleEvents(event : const sf::Event&)
+ update()
+ draw(target : sf::RenderTarget&, states : sf::RenderStates)
+ requestClear()
GUI::Button
- normalTexture : const sf::Texture&
- selectedTexture : const sf::Texture&
- sprite : sf::Sprite
- text : sf::Text
- size : sf::Vector2f
- activateFunction : std::function< void (Button &) >
- deactivateFunction : std::function< void (Button &) >
+ Button(textures : const TextureManager&, fonts : const FontManager&) «constructor»
+ onEnter()
+ onLeave()
+ activate()
+ deactivate()
+ setSize(width : int, height : int)
+ setActiveFunction(onActivate : std::function< void (Button &) >)
+ setDeactiveFunction(onDeactivate : std::function< void (Button &) >)
+ setText(text : const std::string&)
+ getGlobalBounds() : sf::FloatRect
+ getLocalBounds() : sf::FloatRect
+ matchSizeToText(padding : float)
+ setPositionBelow(object : const T&, padding : float)
+ handleEvents(event : const sf::Event&)
+ update(mousePosition : sf::Vector2f)
+ draw(target : sf::RenderTarget&, states : sf::RenderStates)

Figure 24: Elements that build the graphical user interface

4.8 Techniques from the thematic classes

The project is large, and it to use information from all classes – except just Class 13 which are regular expressions, and classes 12 (Threads) are unused too (at least in this state, as they can be later useful for creating loading screens). However, knowledge from other classes will certainly be used:

1. Class declaration
2. Operator overloading
3. Inheritance
4. Polymorphism
5. Multiple inheritance
6. RTTI
7. Exception mechanism
8. Templates
9. STL containers
10. STL algorithms and iterators
11. Smart pointers

4.9 Description of types and functions

Description of types and functions is moved to the appendix (page 26).

5 Testing and debugging

Testing covered a large part of the project work. Games were tested repeatedly with each change. During the testing process, bugs such as:

- Failure to lock the worms' positions led to them being pushed off the map by the currently controlled worm.
- Death of the last two different worms simultaneously – initially led to the end of the application, now it leads to a draw.
- A misconstrued vector after the explosion that pushed the furthest away worms the most and the closest away worms the least.
- Incorrect order in which objects were drawn, allowing other worms to cover the equipment
- Initially a worm-length ground contact detection sensor made that the worm could jump/walk while clinging to walls.
- Complete destruction of the destructive object caused the application to shut down
- A bullet fired during the game would initially add itself to the vector on which the current iteration was running, thus destroying the current iterators – the vector has been changed to a map, although there is still an option to create a queue for the vector.
- It turned out that in the *Box2D* library, two static objects do not report to the listener when a collision occurs between them, so that, one of the two objects: *Hitbox*, or *Destructive object* had to be changed to a dynamic object.
- The speed of movement of the pointer indicating the direction of the shot was initially dependent on the frames per second displayed.
- Before using triangulation I tried to display a transformed concave figure forgetting that SFML only supports convex shapes. At first it looked fine until the figure became concave.
- The counter of available ammunition for the weapon displayed in reverse order. For example, the ammunition for the far right weapon was displayed for the far left weapon.

The best way of testing, of course, was to test every newly introduced mechanic. It often takes a long time, but no one can always see everything from the code. That's why the game industry developed the job of game tester.

The game was also checked for memory leaks, which did not occur. The game almost always used smart pointers when there was a need of a dynamic allocation. The only exception was collision information, which was dynamically allocated without smart pointers, but was checked in detail for this.

The game also seemed to work correctly on several different computers.

6 Remarks and conclusions

I am glad that I managed to bring the project to the end. Personally, I treat these projects as another entry in my portfolio and an extension of my skills. Of course, the project is given up at this stage, but that does not mean that it will stop being developed. The project will get its own repository on which it is still possible that it will be developed. I am glad that the classes looked the way they did. It also helps a lot that the current block allowed to focus on the project work without cramming all the subjects into one block.

If I start to develop this project further without a deadline, I will definitely try to add a network model, graphical improvements and, above all, sound design. As standard, however, before I get down to another project I am going to finish the book *SFML Game Development* written by *Laurent Gomila*, which has helped me a lot, and the book *Effective Modern C++*, which I hope will further develop me in terms of writing better code.

Appendix
Description of types and functions

Worms Clone

Generated by Doxygen 1.8.17

1 Class Documentation **1**
1.1 b2ContactListener Class Reference 1
1.2 b2Draw Class Reference 1
1.3 Bazooka Class Reference 2
1.3.1 Detailed Description 3
1.3.2 Member Function Documentation 3
1.3.2.1 isActivation() 3
1.3.2.2 isRoundEnding() 3
1.4 Bullet Struct Reference 4
1.4.1 Detailed Description 6
1.4.2 Member Function Documentation 6
1.4.2.1 isDestroyed() 6
1.4.2.2 setSparkColor() 6
1.4.2.3 updateThis() 6
1.5 GUI::Button Class Reference 7
1.5.1 Detailed Description 9
1.5.2 Member Function Documentation 9
1.5.2.1 draw() 9
1.5.2.2 getGlobalBounds() 9
1.5.2.3 getLocalBounds() 9
1.5.2.4 handleEvents() 10
1.5.2.5 matchSizeToText() 11
1.5.2.6 setActiveFunction() 11
1.5.2.7 setDeactiveFunction() 11
1.5.2.8 setPositionBelow() 12
1.5.2.9 setSize() 12
1.5.2.10 setText() 12
1.5.2.11 update() 13
1.6 Cannon Class Reference 13
1.6.1 Detailed Description 14
1.6.2 Member Function Documentation 15
1.6.2.1 isActivation() 15
1.6.2.2 isRoundEnding() 15
1.7 Collision Struct Reference 15
1.7.1 Detailed Description 16
1.8 GUI::Component Class Reference 16
1.8.1 Detailed Description 17
1.8.2 Member Function Documentation 17
1.8.2.1 getGlobalBounds() 18
1.8.2.2 getLocalBounds() 18
1.8.2.3 handleEvents() 18
1.8.2.4 isActive() 18

1.8.2.5 isSelected() 19
1.8.2.6 setPositionBelow() 19
1.8.2.7 update() 19
1.9 GUI::Container Class Reference 20
1.9.1 Detailed Description 21
1.9.2 Member Function Documentation 21
1.9.2.1 back() 21
1.9.2.2 draw() 21
1.9.2.3 front() 22
1.9.2.4 handleEvents() 22
1.9.2.5 isEmpty() 22
1.9.2.6 store() 22
1.10 DebugBox2D Class Reference 23
1.11 Delayed_Bullet Struct Reference 24
1.11.1 Detailed Description 26
1.11.2 Member Function Documentation 26
1.11.2.1 drawThis() 26
1.11.2.2 updateThis() 27
1.12 Editor Class Reference 27
1.12.1 Detailed Description 28
1.12.2 Member Function Documentation 28
1.12.2.1 handleEvent() 28
1.12.2.2 update() 28
1.13 EditorState Class Reference 28
1.13.1 Detailed Description 29
1.13.2 Member Function Documentation 30
1.13.2.1 draw() 30
1.13.2.2 handleEvent() 30
1.13.2.3 update() 30
1.14 GUI::FixedContainer Class Reference 31
1.14.1 Detailed Description 32
1.14.2 Member Function Documentation 32
1.14.2.1 draw() 32
1.14.2.2 store() 32
1.15 Game Class Reference 33
1.15.1 Detailed Description 33
1.15.2 Member Function Documentation 33
1.15.2.1 run() 33
1.16 GameplayManager Class Reference 34
1.16.1 Detailed Description 35
1.16.2 Member Function Documentation 35
1.16.2.1 addTime() 35

1.16.2.2 addWorm() . . . 36
1.16.2.3 anyBullet() . . . 36
1.16.2.4 drawThis() . . . 36
1.16.2.5 getRootNode() [1/2] . . . 36
1.16.2.6 getRootNode() [2/2] . . . 37
1.16.2.7 handleThisEvents() . . . 37
1.16.2.8 isGameFinished() . . . 37
1.16.2.9 setWorldMessage() . . . 38
1.16.2.10 updateThis() . . . 38
1.17 GameState Class Reference . . . 38
1.17.1 Detailed Description . . . 39
1.17.2 Member Function Documentation . . . 40
1.17.2.1 draw() . . . 40
1.17.2.2 handleEvent() . . . 40
1.17.2.3 update() . . . 40
1.18 Grenade Class Reference . . . 41
1.18.1 Detailed Description . . . 42
1.18.2 Member Function Documentation . . . 42
1.18.2.1 isActivation() . . . 42
1.18.2.2 isRoundEnding() . . . 42
1.18.2.3 shoot() . . . 42
1.19 Hitbox Class Reference . . . 43
1.19.1 Detailed Description . . . 44
1.19.2 Member Function Documentation . . . 44
1.19.2.1 updateThis() . . . 44
1.20 MenuState Class Reference . . . 45
1.20.1 Detailed Description . . . 46
1.20.2 Member Function Documentation . . . 46
1.20.2.1 createBackgroundWorld() . . . 46
1.20.2.2 createGrenades() . . . 46
1.20.2.3 draw() . . . 47
1.20.2.4 handleEvent() . . . 47
1.20.2.5 update() . . . 47
1.21 NodeDestructibleRectangle Class Reference . . . 48
1.21.1 Detailed Description . . . 49
1.21.2 Member Function Documentation . . . 49
1.21.2.1 addHole() . . . 49
1.21.2.2 updateThis() . . . 50
1.22 NodeEditorObject Class Reference . . . 50
1.22.1 Detailed Description . . . 52
1.22.2 Member Function Documentation . . . 52
1.22.2.1 draw() . . . 52

1.22.2.2 getGlobalBounds() 52
1.22.2.3 getId() 52
1.22.2.4 getSize() 53
1.22.2.5 handleEvent() 53
1.22.2.6 isActivated() 53
1.22.2.7 isDestroyed() 53
1.22.2.8 isSelected() 54
1.22.2.9 setId() 54
1.22.2.10 setName() 54
1.22.2.11 setSize() 54
1.22.2.12 update() 55
1.22.2.13 updateMouse() 55
1.23 NodePhysicalBase Class Reference 55
1.23.1 Detailed Description 57
1.23.2 Member Function Documentation 57
1.23.2.1 angleToRadians() 57
1.23.2.2 b2VecToSfVector() 57
1.23.2.3 radiansToAngle() 58
1.23.2.4 sfVectorToB2Vec() 58
1.23.2.5 update() 59
1.24 NodePhysicalBody Class Reference 59
1.24.1 Detailed Description 60
1.24.2 Member Enumeration Documentation 61
1.24.2.1 Physical_Types 61
1.24.3 Member Function Documentation 61
1.24.3.1 applyForce() 61
1.24.3.2 setRotation() 61
1.25 NodePhysicalSpark Class Reference 62
1.25.1 Detailed Description 63
1.25.2 Member Function Documentation 63
1.25.2.1 drawThis() 63
1.25.2.2 updateThis() 64
1.26 NodePhysicalSprite Class Reference 64
1.26.1 Detailed Description 66
1.26.2 Member Function Documentation 67
1.26.2.1 drawThis() 67
1.26.2.2 updateThis() 67
1.27 NodeRectangularPhysical Class Reference 67
1.27.1 Detailed Description 69
1.27.2 Member Function Documentation 69
1.27.2.1 drawThis() 69
1.28 NodeScene Class Reference 69

1.28.1 Detailed Description 71
1.28.2 Member Function Documentation 72
1.28.2.1 draw() 72
1.28.2.2 drawThis() 72
1.28.2.3 getAbsolutePosition() 72
1.28.2.4 getRootNode() [1/2] 73
1.28.2.5 getRootNode() [2/2] 73
1.28.2.6 handleEvents() 73
1.28.2.7 handleThisEvents() 73
1.28.2.8 isDestroyed() 74
1.28.2.9 pinNode() 74
1.28.2.10 removeDestroyed() 74
1.28.2.11 unpinNode() 75
1.28.2.12 update() 75
1.28.2.13 updateThis() 75
1.29 NodeSprite Class Reference 76
1.29.1 Detailed Description 77
1.29.2 Constructor & Destructor Documentation 77
1.29.2.1 NodeSprite() [1/2] 77
1.29.2.2 NodeSprite() [2/2] 77
1.29.3 Member Function Documentation 78
1.29.3.1 getSpriteSize() 78
1.30 NodeText Class Reference 78
1.30.1 Detailed Description 79
1.30.2 Constructor & Destructor Documentation 79
1.30.2.1 NodeText() [1/2] 79
1.30.2.2 NodeText() [2/2] 80
1.30.3 Member Function Documentation 80
1.30.3.1 setColor() 80
1.30.3.2 setOutline() 80
1.30.3.3 setSize() 81
1.30.3.4 setString() 81
1.31 NodeWater Class Reference 81
1.31.1 Detailed Description 83
1.31.2 Member Function Documentation 83
1.31.2.1 setPosition() 83
1.31.2.2 setSize() 83
1.32 PauseState Class Reference 84
1.32.1 Detailed Description 85
1.32.2 Member Function Documentation 85
1.32.2.1 draw() 85
1.32.2.2 handleEvent() 85

1.32.2.3 update() 86
1.33 ResourceManager< Resource, Identifier > Class Template Reference 86
1.33.1 Member Function Documentation 86
1.33.1.1 getResourceReference() [1/2] 86
1.33.1.2 getResourceReference() [2/2] 87
1.33.1.3 storeResource() [1/2] 87
1.33.1.4 storeResource() [2/2] 87
1.34 State Class Reference 88
1.34.1 Detailed Description 90
1.34.2 Member Function Documentation 90
1.34.2.1 draw() 90
1.34.2.2 handleEvent() 90
1.34.2.3 requestPush() 91
1.34.2.4 update() 91
1.35 StateStack Class Reference 91
1.35.1 Detailed Description 93
1.35.2 Member Function Documentation 93
1.35.2.1 draw() [1/2] 93
1.35.2.2 draw() [2/2] 93
1.35.2.3 empty() 94
1.35.2.4 handleEvent() 94
1.35.2.5 pop() 94
1.35.2.6 push() 94
1.35.2.7 saveState() 95
1.35.2.8 update() 95
1.36 Team Struct Reference 96
1.36.1 Detailed Description 96
1.37 TitleState Class Reference 97
1.37.1 Detailed Description 98
1.37.2 Member Function Documentation 98
1.37.2.1 draw() 98
1.37.2.2 handleEvent() 98
1.37.2.3 update() 98
1.38 NodeWater::WaterLayer Struct Reference 99
1.38.1 Detailed Description 100
1.39 Weapon Class Reference 100
1.39.1 Detailed Description 101
1.39.2 Member Function Documentation 102
1.39.2.1 activation() 102
1.39.2.2 drawThis() 102
1.39.2.3 getThumbnailSprite() 102
1.39.2.4 isActivation() 103

1.39.2.5 isRoundEnding() . . . 103
1.39.2.6 rotateWeapon() . . . 103
1.39.2.7 setMaxDmg() . . . 103
1.39.2.8 setRange() . . . 104
1.39.2.9 setSparkColor() . . . 104
1.39.2.10 shoot() . . . 104
1.39.2.11 updateThis() . . . 105
1.40 World Class Reference . . . 105
1.40.1 Detailed Description . . . 106
1.40.2 Member Function Documentation . . . 106
1.40.2.1 processEvents() . . . 106
1.40.2.2 update() . . . 107
1.41 WorldListener Class Reference . . . 107
1.41.1 Detailed Description . . . 108
1.42 Worm Class Reference . . . 108
1.42.1 Detailed Description . . . 110
1.42.2 Constructor & Destructor Documentation . . . 111
1.42.2.1 Worm() . . . 111
1.42.3 Member Function Documentation . . . 111
1.42.3.1 activateState() . . . 111
1.42.3.2 drawThis() . . . 111
1.42.3.3 facingRight() . . . 112
1.42.3.4 getCurrentState() . . . 112
1.42.3.5 getName() . . . 112
1.42.3.6 getTeam() . . . 113
1.42.3.7 getWormSize() . . . 113
1.42.3.8 handleThisEvents() . . . 113
1.42.3.9 isDestroyed() . . . 113
1.42.3.10 setDamage() . . . 114
1.42.3.11 setName() . . . 114
1.42.3.12 setTeam() . . . 114
1.42.3.13 updateThis() . . . 114
1.43 WormHideState Class Reference . . . 115
1.43.1 Detailed Description . . . 116
1.43.2 Member Function Documentation . . . 117
1.43.2.1 draw() . . . 117
1.43.2.2 handleEvent() . . . 117
1.43.2.3 update() . . . 117
1.44 WormHitState Class Reference . . . 118
1.44.1 Detailed Description . . . 119
1.44.2 Member Function Documentation . . . 119
1.44.2.1 draw() . . . 119

1.44.2.2 handleEvent() 119
1.44.2.3 update() 120
1.45 WormInventoryState Class Reference 120
1.45.1 Detailed Description 121
1.45.2 Member Function Documentation 121
1.45.2.1 draw() 121
1.45.2.2 handleEvent() 121
1.45.2.3 update() 122
1.46 WormMoveableState Class Reference 122
1.46.1 Detailed Description 124
1.46.2 Member Function Documentation 124
1.46.2.1 draw() 124
1.46.2.2 facingRight() 124
1.46.2.3 handleEvent() 124
1.46.2.4 handleMovement() 125
1.46.2.5 update() 125
1.46.2.6 updateMovement() 125
1.46.3 Member Data Documentation 126
1.46.3.1 direction 126
1.47 WormPlayState Class Reference 126
1.47.1 Detailed Description 127
1.47.2 Member Function Documentation 128
1.47.2.1 draw() 128
1.47.2.2 handleEvent() 128
1.47.2.3 shoot() 128
1.47.2.4 update() 128
1.48 WormQueue Class Reference 129
1.48.1 Detailed Description 129
1.48.2 Member Function Documentation 130
1.48.2.1 addWorm() 130
1.48.2.2 aliveTeams() 130
1.48.2.3 draw() 130
1.48.2.4 front() 131
1.48.2.5 getNextWorm() 131
1.48.2.6 handleEvents() 132
1.48.2.7 isEmpty() 132
1.48.2.8 removeDestroyed() 132
1.48.2.9 update() 132
1.49 WormWaitState Class Reference 133
1.49.1 Member Function Documentation 134
1.49.1.1 draw() 134
1.49.1.2 handleEvent() 134

1.49.1.3 update() 134

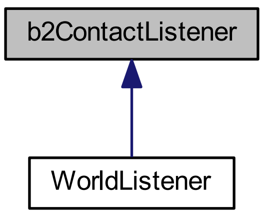
Index **137**

Chapter 1

Class Documentation

1.1 b2ContactListener Class Reference

Inheritance diagram for b2ContactListener:

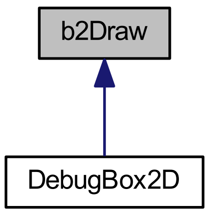


The documentation for this class was generated from the following file:

- Worms-Clone/Worms-Clone/WorldListener.h

1.2 b2Draw Class Reference

Inheritance diagram for b2Draw:



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

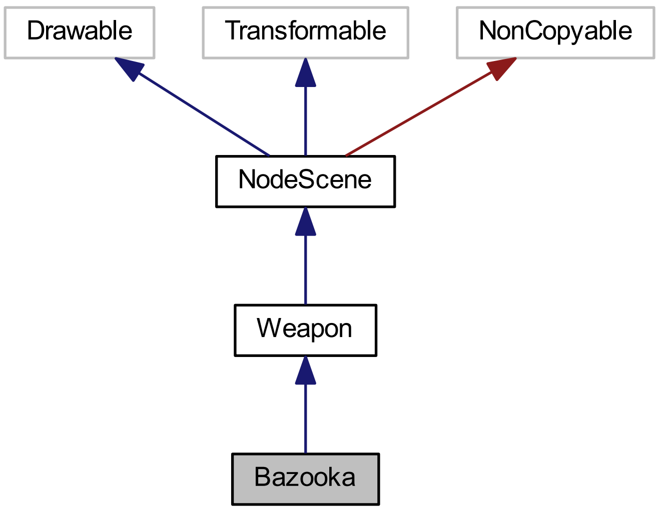
- Worms-Clone/Worms-Clone/DebugBox2D.h

1.3 Bazooka Class Reference

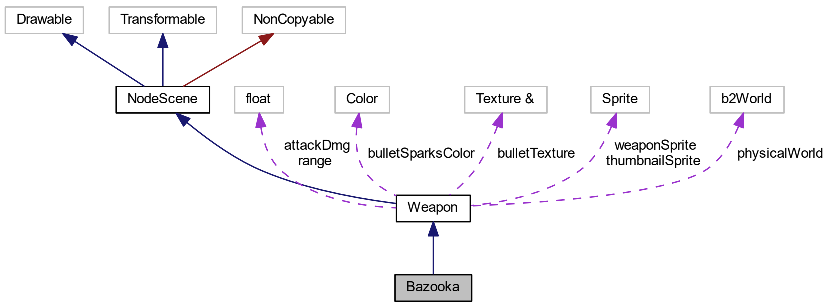
The bazooka is a long-range weapon with medium damage.

```
#include <Bazooka.h>
```

Inheritance diagram for Bazooka:



Collaboration diagram for Bazooka:



Public Member Functions

- **Bazooka** (b2World &world, TextureManager &textures)
- bool isActivation () override

  Is the bazooka an activated weapon or a loaded weapon (via the shooting bar).
- bool isRoundEnding () override

  Does using the Bazooka end the round.

Additional Inherited Members

1.3.1 Detailed Description

The bazooka is a long-range weapon with medium damage.

Its projectiles explode on contact with a physical object.

1.3.2 Member Function Documentation

1.3.2.1 isActivation()

```
bool Bazooka::isActivation ( )  [override], [virtual]
```

Is the bazooka an activated weapon or a loaded weapon (via the shooting bar).

Returns

True if it is activated weapon, false if it need to be loaded

Implements Weapon.

1.3.2.2 isRoundEnding()

```
bool Bazooka::isRoundEnding ( )  [override], [virtual]
```

Does using the Bazooka end the round.

Returns

True usage of bazooka ends the round, false otherwise

Implements Weapon.

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

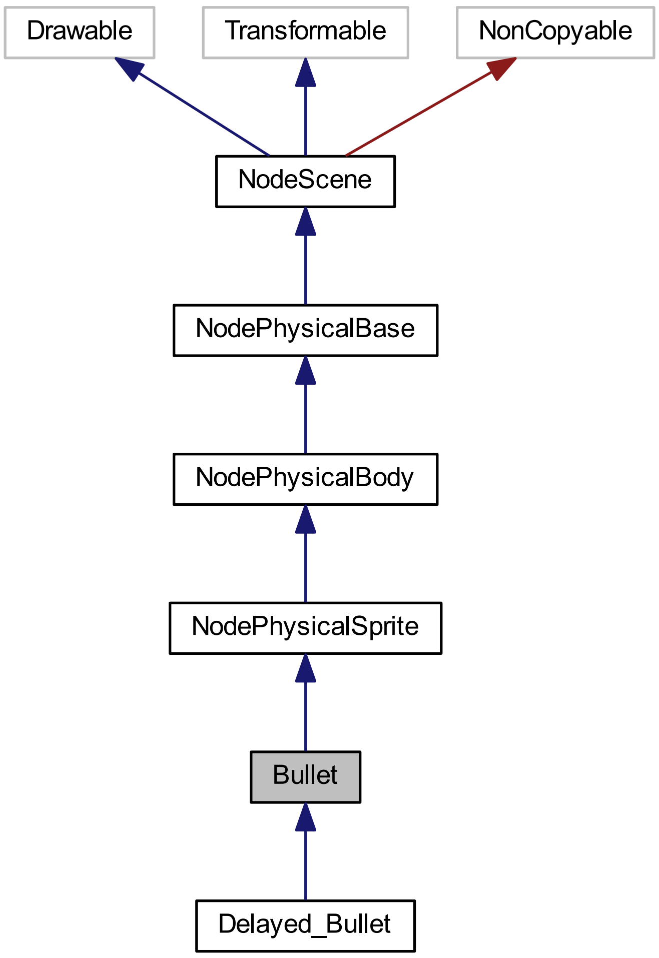
- Worms-Clone/Worms-Clone/Nodes/Physical/Specified/Worm/Weapons/Bazooka.h

1.4 Bullet Struct Reference

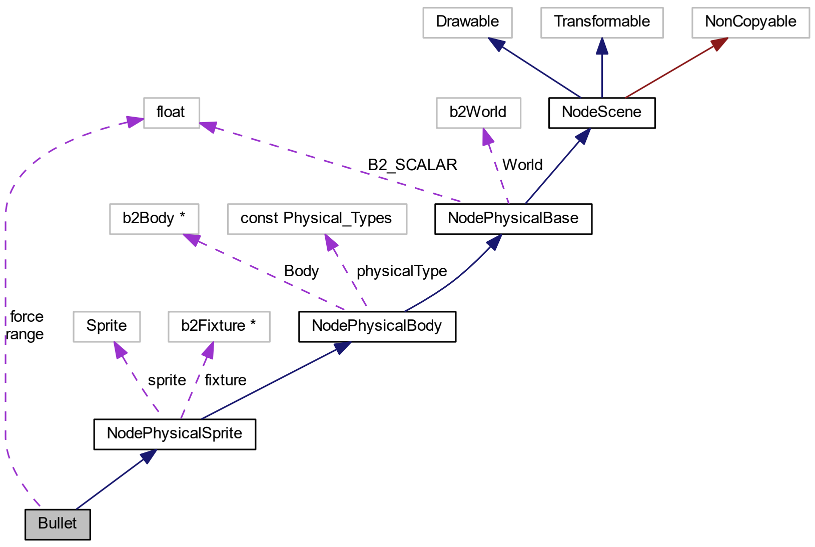
This is a standard button that, when it collides into place, creates a Hitbox class object that checks the worms in its range and deals damage to them.

```
#include <Bullet.h>
```

Inheritance diagram for Bullet:



Collaboration diagram for Bullet:



Public Member Functions

- **Bullet** (b2World &world, sf::Vector2f position, sf::Texture &texture, float force, float range)
- void updateThis (sf::Time deltaTime) override

  Updates the current state of the sprite.
- void explode ()

  A function that executes at the moment of explosion (immediately after the collision).
- virtual void collision ()

  A function that executes upon collision with a physical object.
- void setSparkColor (const sf::Color &color)

  Sets the color of the particles that will disperse when the projectile explodes.

Protected Member Functions

- bool isDestroyed () override

  A function that checks if an object is marked as ready for deletion.

Protected Attributes

- float force

  The force with which the bullet will strike (the power of the damage inflicted).
- float range

  The range that the bullet will cover after the explosion.

Additional Inherited Members

1.4.1 Detailed Description

This is a standard button that, when it collides into place, creates a Hitbox class object that checks the worms in its range and deals damage to them.

1.4.2 Member Function Documentation

1.4.2.1 isDestroyed()

```
bool Bullet::isDestroyed ( )  [override], [protected], [virtual]
```

A function that checks if an object is marked as ready for deletion.

Additionally, it applies visual effects where the object is removable.

Returns

True if it is ready to be removed, false otherwise.

Reimplemented from NodeScene.

1.4.2.2 setSparkColor()

```
void Bullet::setSparkColor (
             const sf::Color & color )
```

Sets the color of the particles that will disperse when the projectile explodes.

Parameters

color	Color of the particles

1.4.2.3 updateThis()

```
void Bullet::updateThis (
             sf::Time deltaTime )  [override], [virtual]
```

Updates the current state of the sprite.

Parameters

deltaTime	Time passed since the last frame

It synchronizes the sprite with the physics world

Reimplemented from NodePhysicalSprite.

Reimplemented in Delayed_Bullet.

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

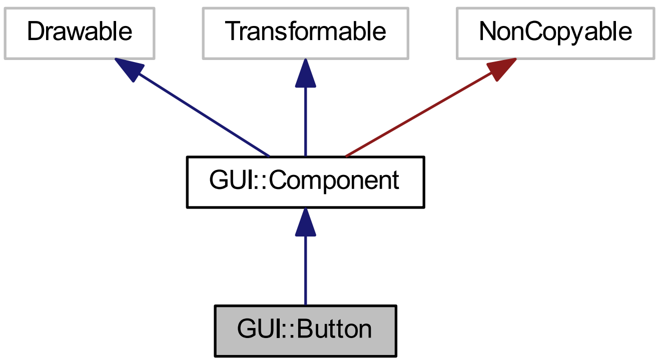
- Worms-Clone/Worms-Clone/Nodes/Physical/Specified/Worm/Weapons/Bullet.h

1.5 GUI::Button Class Reference

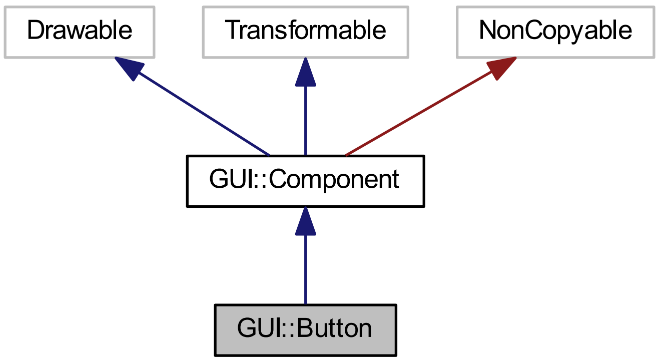
A button that can be pressed in two ways depending on the settings (right and left mouse button).

```
#include <Button.h>
```

Inheritance diagram for GUI::Button:



Collaboration diagram for GUI::Button:



Public Member Functions

- **Button** (const TextureManager &textures, const FontManager &fonts)
- void onEnter () override

  Code that executes everytime when the player enters the button area.
- void onLeave () override

  Code that executes everytime when the player leaves the button area.
- void activate () override

  Function executed when a button is activated (pressed).
- void deactivate () override

  Function executed when a button is deactivated (for example pressed with other button).
- void setSize (int width, int height)

  Sets a new button dimension.
- void setActiveFunction (std::function< void(Button &) > onActivate)

  Sets the function that is performed when the button is activated.
- void setDeactiveFunction (std::function< void(Button &)> onDeactivate)

  Sets the function that is performed when the button is deactivated.
- void setText (const std::string &text)

  Sets the text inside the button.
- sf::FloatRect getGlobalBounds () const override

  Get the global bounding rectangle of the button.
- sf::FloatRect getLocalBounds () const override

  Get the local bounding rectangle of the button.
- void matchSizeToText (float padding=0.f)

  Resizes the button to fit the text it contains.
- template<typename T >
  void setPositionBelow (const T &object, float padding=0.f)

  Sets button under another object.
- void handleEvents (const sf::Event &event) override

  It takes input (event) from the user and interprets it.
- void update (sf::Vector2f mousePosition) override

  Updates the status/logic of the button.
- void draw (sf::RenderTarget &target, sf::RenderStates states) const override

  Draws only this state to the passed target.

1.5.1 Detailed Description

A button that can be pressed in two ways depending on the settings (right and left mouse button).

1.5.2 Member Function Documentation

1.5.2.1 draw()

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

Draws only this state to the passed target.

Parameters

target	where it should be drawn to
states	provides information about rendering process (transform, shader, blend mode)

1.5.2.2 getGlobalBounds()

```
sf::FloatRect GUI::Button::getGlobalBounds ( ) const  [override], [virtual]
```

Get the global bounding rectangle of the button.

Returns

The global bounds of the button

Implements GUI::Component.

1.5.2.3 getLocalBounds()

```
sf::FloatRect GUI::Button::getLocalBounds ( ) const  [override], [virtual]
```

Get the local bounding rectangle of the button.

Returns

The local bounds of the button

Implements GUI::Component.

1.5.2.4 handleEvents()

```
void GUI::Button::handleEvents (
             const sf::Event & event )  [override], [virtual]
```

It takes input (event) from the user and interprets it.

Parameters

event	user input

Implements GUI::Component.

1.5.2.5 matchSizeToText()

```
void GUI::Button::matchSizeToText (
             float padding = 0.f )
```

Resizes the button to fit the text it contains.

Parameters

padding	Additional distance of the button edge from the text

1.5.2.6 setActiveFunction()

```
void GUI::Button::setActiveFunction (
             std::function< void(Button &) > onActivate )
```

Sets the function that is performed when the button is activated.

Parameters

onActivate	Function to execute

1.5.2.7 setDeactiveFunction()

```
void GUI::Button::setDeactiveFunction (
             std::function< void(Button &)> onDeactivate )
```

Sets the function that is performed when the button is deactivated.

Parameters

onDeactivate	Function to execute

1.5.2.8 setPositionBelow()

```
template<typename T >
void GUI::Button::setPositionBelow (
             const T & object,
             float padding = 0.f )
```

Sets button under another object.

Function to place a button under another object.

Template Parameters

T	Type of object on the basis of which we set the button

Parameters

object	The object on the basis of which we set the button
padding	Additional distance between reference point and button

Template Parameters

T	Other object type

Parameters

object	The object under which the button should be placed
padding	Additional spacing that the button should maintain

1.5.2.9 setSize()

```
void GUI::Button::setSize (
             int width,
             int height )
```

Sets a new button dimension.

Parameters

width	Width of button
height	Height of button

1.5.2.10 setText()

```
void GUI::Button::setText (
            const std::string & text )
```

Sets the text inside the button.

Parameters

text	Text to be placed inside the button

1.5.2.11 update()

```
void GUI::Button::update (
            sf::Vector2f mousePosition )  [override], [virtual]
```

Updates the status/logic of the button.

Parameters

mousePosition	Current mouse position

Implements GUI::Component.

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

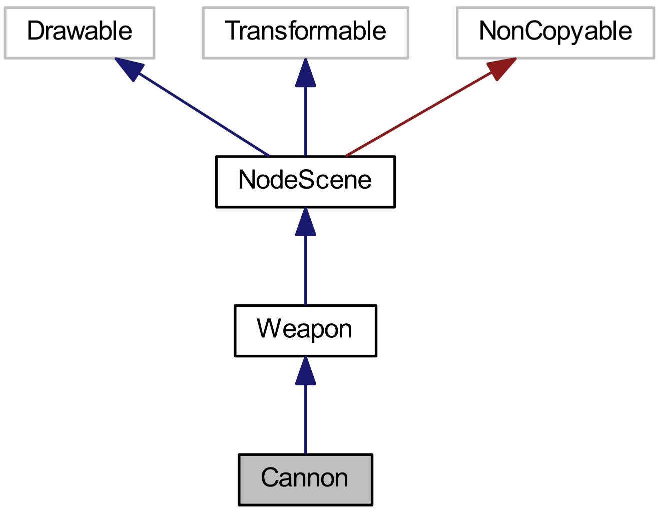
- Worms-Clone/Worms-Clone/GUI/Button.h

1.6 Cannon Class Reference

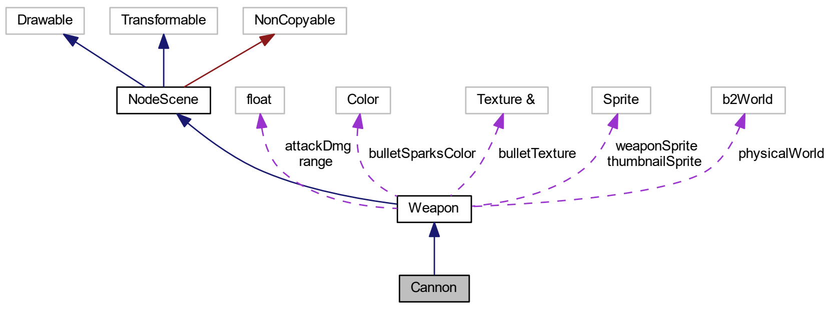
A weapon with a short range of fire but high damage.

```
#include <Cannon.h>
```

Inheritance diagram for Cannon:



Collaboration diagram for Cannon:



Public Member Functions

- **Cannon** (b2World &world, TextureManager &textures)
- bool isActivation () override

  Is the cannon an activated weapon or a loaded weapon (via the shooting bar).
- bool isRoundEnding () override

  Does using the Cannon end the round.

Additional Inherited Members

1.6.1 Detailed Description

A weapon with a short range of fire but high damage.

1.6.2 Member Function Documentation

1.6.2.1 isActivation()

```
bool Cannon::isActivation ( )  [override], [virtual]
```

Is the cannon an activated weapon or a loaded weapon (via the shooting bar).

Returns

True if it is activated weapon, false if it need to be loaded

Implements Weapon.

1.6.2.2 isRoundEnding()

```
bool Cannon::isRoundEnding ( )  [override], [virtual]
```

Does using the Cannon end the round.

Returns

True usage of cannon ends the round, false otherwise

Implements Weapon.

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

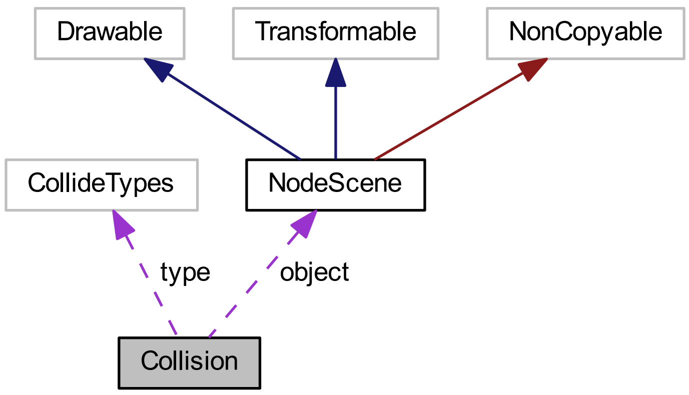
- Worms-Clone/Worms-Clone/Nodes/Physical/Specified/Worm/Weapons/Cannon.h

1.7 Collision Struct Reference

A collision object that contains information about what object is colliding, and stores a pointer to that object.

`#include <CollideTypes.h>`

Collaboration diagram for Collision:



Public Member Functions

- **Collision** (CollideTypes type, NodeScene &object)
- **operator Collision** ∗ ()

Public Attributes

- CollideTypes **type**
- NodeScene ∗ **object**

1.7.1 Detailed Description

A collision object that contains information about what object is colliding, and stores a pointer to that object.

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

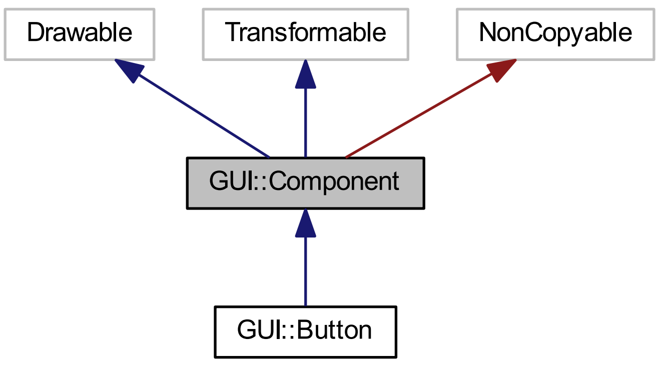
- Worms-Clone/Worms-Clone/Nodes/Physical/CollideTypes.h

1.8 GUI::Component Class Reference

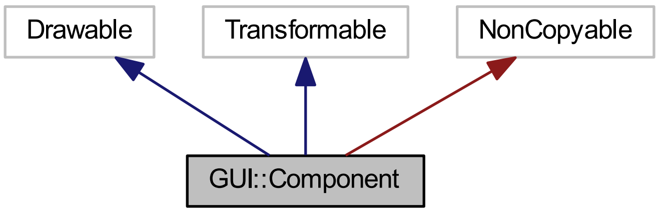
A GUI object that the player can interact with.

`#include <Component.h>`

Inheritance diagram for GUI::Component:



Collaboration diagram for GUI::Component:



Public Member Functions

- bool isSelected () const

  Checks whether the user moved the mouse over the component.
- virtual void onEnter ()

  Code that executes everytime when the player enters the component area.
- virtual void onLeave ()

  Code that executes everytime when the player leaves the component area.
- virtual bool isActive () const

  Checks whether a component has been pressed/activated.
- virtual void activate ()

  Function executed when a component is activated (pressed).
- virtual void deactivate ()

  Function executed when a component is deactivated (for example pressed with other button).
- virtual sf::FloatRect getGlobalBounds () const =0

  Get the global bounding rectangle of the button.
- virtual sf::FloatRect getLocalBounds () const =0

  Get the local bounding rectangle of the button.
- virtual void setPositionBelow (const Component &object, float padding=0.f)

  Sets position of a component under another component.
- virtual void handleEvents (const sf::Event &event)=0

  It takes input (event) from the user and interprets it.
- virtual void update (sf::Vector2f mousePosition)=0

  Updates the status/logic of the component.

1.8.1 Detailed Description

A GUI object that the player can interact with.

1.8.2 Member Function Documentation

1.8.2.1 getGlobalBounds()

```
virtual sf::FloatRect GUI::Component::getGlobalBounds ( ) const  [pure virtual]
```

Get the global bounding rectangle of the button.

Returns

The global bounds of the button

Implemented in GUI::Button.

1.8.2.2 getLocalBounds()

```
virtual sf::FloatRect GUI::Component::getLocalBounds ( ) const  [pure virtual]
```

Get the local bounding rectangle of the button.

Returns

The local bounds of the button

Implemented in GUI::Button.

1.8.2.3 handleEvents()

```
virtual void GUI::Component::handleEvents (
             const sf::Event & event )  [pure virtual]
```

It takes input (event) from the user and interprets it.

Parameters

event	user input

Implemented in GUI::Button.

1.8.2.4 isActive()

```
virtual bool GUI::Component::isActive ( ) const  [virtual]
```

Checks whether a component has been pressed/activated.

Returns

True, if activated, false otherwise

1.8.2.5 isSelected()

```
bool GUI::Component::isSelected ( ) const
```

Checks whether the user moved the mouse over the component.

Returns

True if the user hovers the mouse over the object, false otherwise

1.8.2.6 setPositionBelow()

```
virtual void GUI::Component::setPositionBelow (
             const Component & object,
             float padding = 0.f )  [virtual]
```

Sets position of a component under another component.

Parameters

object	Another component against which we set the new position
padding	Additional distance between components

1.8.2.7 update()

```
virtual void GUI::Component::update (
             sf::Vector2f mousePosition )  [pure virtual]
```

Updates the status/logic of the component.

Parameters

mousePosition	Current mouse position

Implemented in GUI::Button.

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

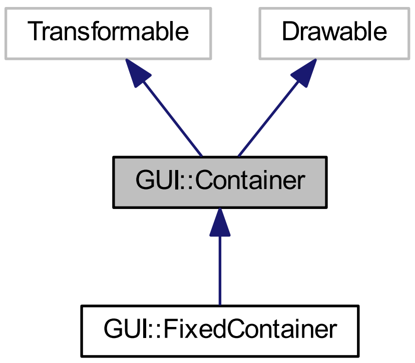
- Worms-Clone/Worms-Clone/GUI/Component.h

1.9 GUI::Container Class Reference

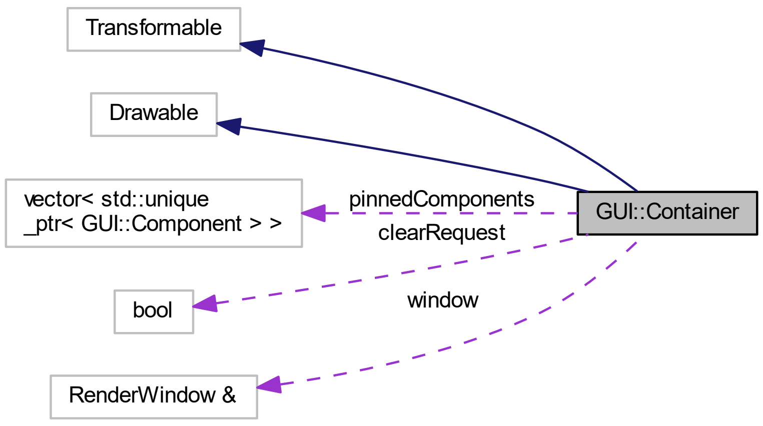
A container for storing all kinds of interactive elements such as button.

```
#include <Container.h>
```

Inheritance diagram for GUI::Container:



Collaboration diagram for GUI::Container:



Public Member Functions

- **Container** (const sf::RenderWindow &window)
- virtual void store (std::unique_ptr< Component > component)

    Adds a component to the container.

- bool isEmpty ()

  Checks and returns information whether a container is empty.
- Component & front ()

  Returns the reference to the element at the front of the container.
- Component & back ()

  Returns the reference to the element at the back of the container.
- void handleEvents (const sf::Event &event)

  It takes input (event) from the user and interprets it, and then passes it to all components inside the container.
- void update ()

  Updates the logic/status of all components inside the container.
- void draw (sf::RenderTarget &target, sf::RenderStates states) const override

  Draws all components inside the container to the passed target.
- void requestClear ()

  Request for clearing the container – the request is only processed at the beginning of the next iteration.

Protected Attributes

- std::vector< std::unique_ptr< Component > > pinnedComponents

  All components that have been pinned to this container.
- const sf::RenderWindow & window

  Window into which the content of the container is displayed.
- bool clearRequest = false

  A flag indicating whether there is a request to clear the container.

1.9.1 Detailed Description

A container for storing all kinds of interactive elements such as button.

The container is inside the game world and does not follow the screen.

1.9.2 Member Function Documentation

1.9.2.1 back()

```
Component& GUI::Container::back ( )
```

Returns the reference to the element at the back of the container.

Returns

Reference to the element at the back of the container

1.9.2.2 draw()

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

Draws all components inside the container to the passed target.

Parameters

target	where it should be drawn to
states	provides information about rendering process (transform, shader, blend mode)

1.9.2.3 front()

```
Component& GUI::Container::front ( )
```

Returns the reference to the element at the front of the container.

Returns

Reference to the element at the front of the container

1.9.2.4 handleEvents()

```
void GUI::Container::handleEvents (
             const sf::Event & event )
```

It takes input (event) from the user and interprets it, and then passes it to all components inside the container.

Parameters

event	user input

1.9.2.5 isEmpty()

```
bool GUI::Container::isEmpty ( )
```

Checks and returns information whether a container is empty.

Returns

True if container is empty, false otherwise

1.9.2.6 store()

```
virtual void GUI::Container::store (
             std::unique_ptr< Component > component )  [virtual]
```

Adds a component to the container.

Parameters

component	Component to add

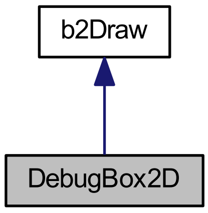
Reimplemented in GUI::FixedContainer.

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

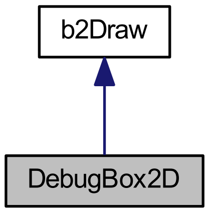
- Worms-Clone/Worms-Clone/GUI/Container.h

1.10 DebugBox2D Class Reference

Inheritance diagram for DebugBox2D:



Collaboration diagram for DebugBox2D:



Public Member Functions

- **DebugBox2D** (sf::RenderWindow &)
- sf::Color **b2ColorConvert** (const b2Color &, sf::Uint8 alpha=255)
- sf::Vector2f **b2VecConvert** (const b2Vec2 &)
- void **DrawPolygon** (const b2Vec2 ∗vertices, int32 vertices_number, const b2Color &color)
- void **DrawSolidPolygon** (const b2Vec2 ∗vertices, int32 vertices_number, const b2Color &color)
- void **DrawCircle** (const b2Vec2 &center, float radius, const b2Color &color)
- void **DrawSolidCircle** (const b2Vec2 &center, float radius, const b2Vec2 &axis, const b2Color &color)
- void **DrawSegment** (const b2Vec2 &p1, const b2Vec2 &p2, const b2Color &color)
- void **DrawTransform** (const b2Transform &xf)
- void **DrawPoint** (const b2Vec2 &p, float size, const b2Color &color)

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

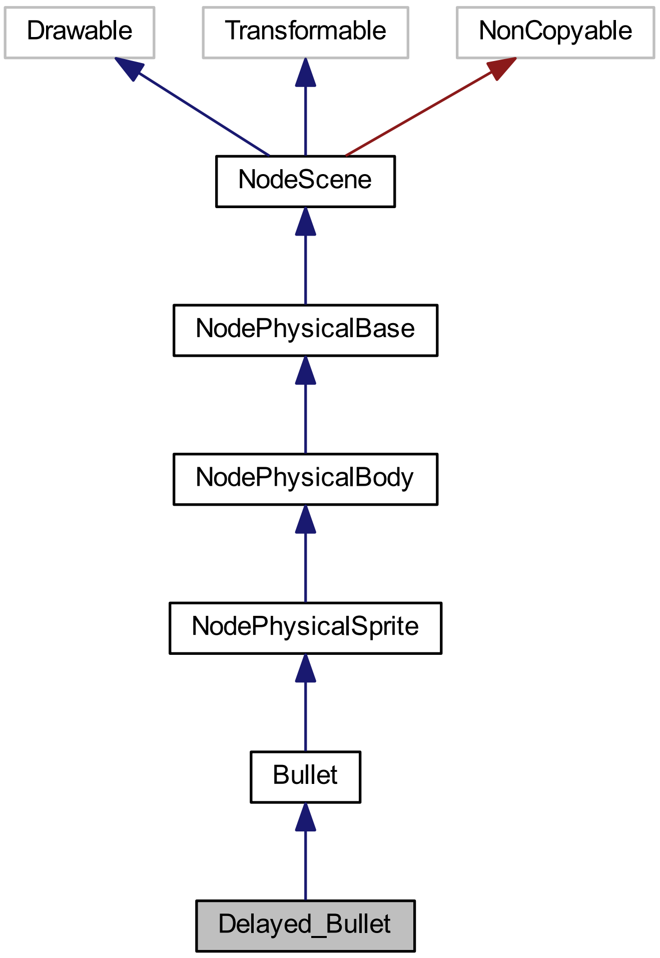
- Worms-Clone/Worms-Clone/DebugBox2D.h

1.11 Delayed_Bullet Struct Reference

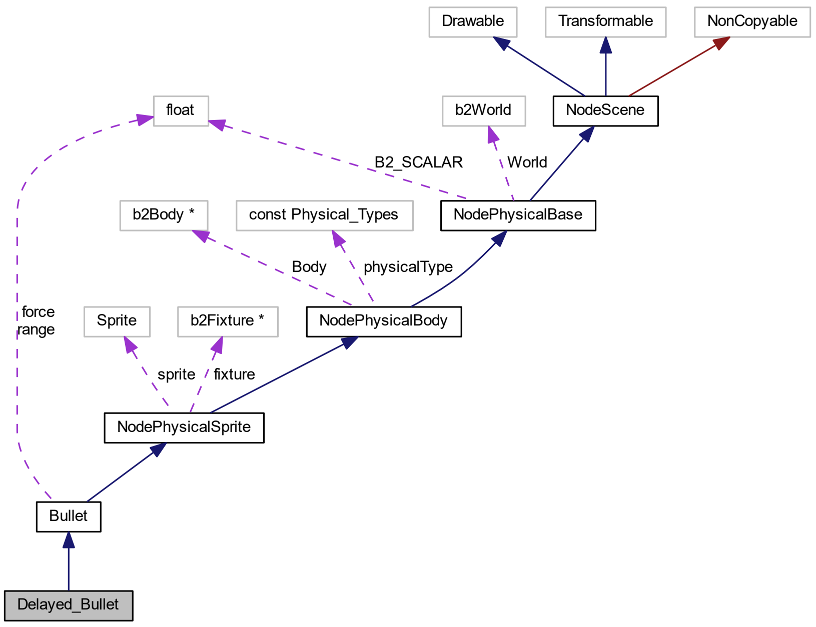
A bullet that does not explode on impact but after a sufficient amount of time has passed.

```
#include <Delayed_Bullet.h>
```

Inheritance diagram for Delayed_Bullet:



Collaboration diagram for Delayed_Bullet:



Public Member Functions

- **Delayed_Bullet** (b2World &world, const FontManager &fonts, sf::Vector2f position, sf::Texture &texture, float force, float range, sf::Time timeToDestroy)
- void updateThis (sf::Time deltaTime) override

  A function that updates the projectile status and checks if enough time has passed for the bullet to explode.
- void drawThis (sf::RenderTarget &target, sf::RenderStates states) const override

  Draws only this Bullet to the passed target.
- void collision () override

  An empty function specifying that nothing happens at the time of the collision.

Additional Inherited Members

1.11.1 Detailed Description

A bullet that does not explode on impact but after a sufficient amount of time has passed.

1.11.2 Member Function Documentation

1.11.2.1 drawThis()

```
void Delayed_Bullet::drawThis (
            sf::RenderTarget & target,
            sf::RenderStates states ) const  [override], [virtual]
```

Draws only this Bullet to the passed target.

Parameters

target	where it should be drawn to
states	provides information about rendering process (transform, shader, blend mode)

Reimplemented from NodePhysicalSprite.

1.11.2.2 updateThis()

```
void Delayed_Bullet::updateThis (
            sf::Time deltaTime )  [override], [virtual]
```

A function that updates the projectile status and checks if enough time has passed for the bullet to explode.

Parameters

deltaTime	Time passed since the last frame

Reimplemented from Bullet.

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

- Worms-Clone/Worms-Clone/Nodes/Physical/Specified/Worm/Weapons/Delayed_Bullet.h

1.12 Editor Class Reference

An in-game editor that allows to edit the map.

```
#include <Editor.h>
```

Public Member Functions

- **Editor** (sf::RenderWindow &window, const FontManager &fonts)
- void update (sf::Time deltaTime)

  Updates the status/logic of the editor.
- void updateMouse ()

  Updates editor logic and status related to the mouse.
- void handleEvent (const sf::Event &event)

  It takes input (event) from the user and interprets it.
- void draw () const

  Draws the editor and all its contents into the application window.
- void saveWorld ()

  Saves the created world to a file.
- void removeDestroyed ()

  Deletes objects mark as ready to be deleted.

1.12.1 Detailed Description

An in-game editor that allows to edit the map.

1.12.2 Member Function Documentation

1.12.2.1 handleEvent()

```
void Editor::handleEvent (
             const sf::Event & event )
```

It takes input (event) from the user and interprets it.

Parameters

event	user input

1.12.2.2 update()

```
void Editor::update (
             sf::Time deltaTime )
```

Updates the status/logic of the editor.

Parameters

deltaTime	the time that has passed since the last frame.

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

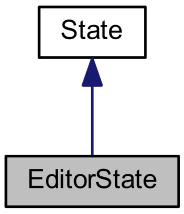
- Worms-Clone/Worms-Clone/Level Editor/Editor.h

1.13 EditorState Class Reference

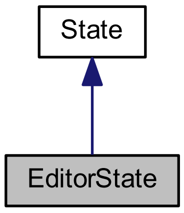
The state in which the player can edit the map inside the game.

```
#include <EditorState.h>
```

Inheritance diagram for EditorState:



Collaboration diagram for EditorState:



Public Member Functions

- **EditorState** (StateStack &stack, sf::RenderWindow &window, const FontManager &fonts)
- void draw () const override

  Draws only this state.
- void draw (sf::RenderTarget &target, sf::RenderStates states) const override

  Draws only this state to the passed target.
- bool update (sf::Time deltaTime) override

  Updates the status/logic of the state.
- bool handleEvent (const sf::Event &event) override

  It takes input (event) from the user and interprets it.

Additional Inherited Members

1.13.1 Detailed Description

The state in which the player can edit the map inside the game.

1.13.2 Member Function Documentation

1.13.2.1 draw()

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

Draws only this state to the passed target.

Parameters

target	where it should be drawn to
states	provides information about rendering process (transform, shader, blend mode)

Implements State.

1.13.2.2 handleEvent()

```
bool EditorState::handleEvent (
            const sf::Event & event )  [override], [virtual]
```

It takes input (event) from the user and interprets it.

Parameters

event	user input

Implements State.

1.13.2.3 update()

```
bool EditorState::update (
            sf::Time deltaTime )  [override], [virtual]
```

Updates the status/logic of the state.

Parameters

deltaTime	the time that has passed since the last frame.

Implements State.

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

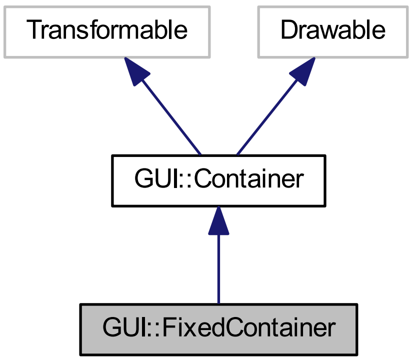
- Worms-Clone/Worms-Clone/States/Application_States/EditorState.h

1.14 GUI::FixedContainer Class Reference

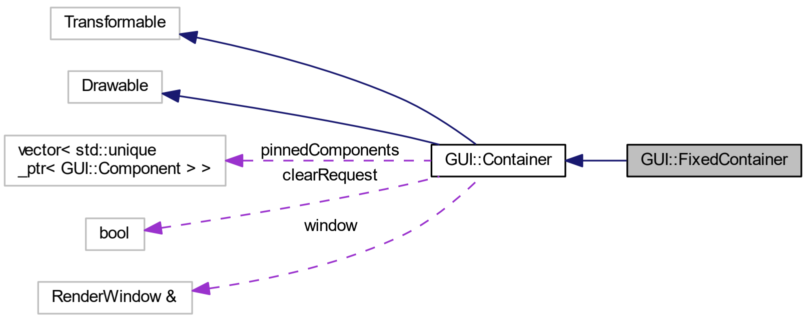
A container in which interface elements are permanently attached to the game window.

```
#include <FixedContainer.h>
```

Inheritance diagram for GUI::FixedContainer:



Collaboration diagram for GUI::FixedContainer:



Public Member Functions

- **FixedContainer** (sf::RenderWindow &window)
- void store (std::unique_ptr< Component > component) override

  Adds a component to the container.
- void update ()

  Updates the logic/status of all components inside the container.
- void draw (sf::RenderTarget &target, sf::RenderStates states) const override

  Draws all components inside the container to the passed target.

Additional Inherited Members

1.14.1 Detailed Description

A container in which interface elements are permanently attached to the game window.

This means that its content continuously follows the screen

1.14.2 Member Function Documentation

1.14.2.1 draw()

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

Draws all components inside the container to the passed target.

Parameters

target	where it should be drawn to
states	provides information about rendering process (transform, shader, blend mode)

1.14.2.2 store()

```
void GUI::FixedContainer::store (
            std::unique_ptr< Component > component )  [override], [virtual]
```

Adds a component to the container.

Parameters

component	Component to add

Reimplemented from GUI::Container.

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

- Worms-Clone/Worms-Clone/GUI/FixedContainer.h

1.15 Game Class Reference

The main game class that controls the entire flow of the application.

```
#include <Game.h>
```

Public Member Functions

- void run ()

  The main loop that controls the operation of the game in the loop.

1.15.1 Detailed Description

The main game class that controls the entire flow of the application.

The whole task of this class is the run() function, which ensures that the game runs. It runs the processes of displaying the game (image), capturing player input and updating the game logic.

1.15.2 Member Function Documentation

1.15.2.1 run()

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

The main loop that controls the operation of the game in the loop.

Updates the game logic, displays the game image and captures the player inputs to the various parts of the program.

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

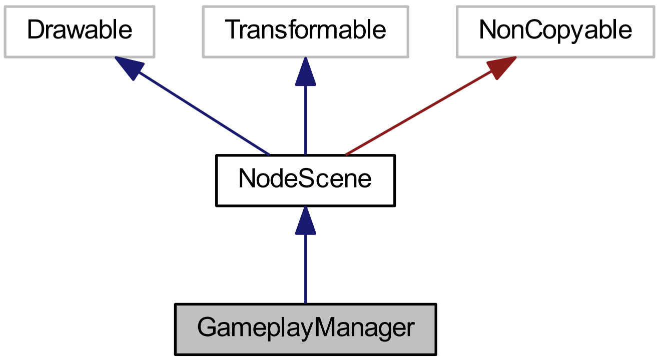
- Worms-Clone/Worms-Clone/Game.h

1.16 GameplayManager Class Reference

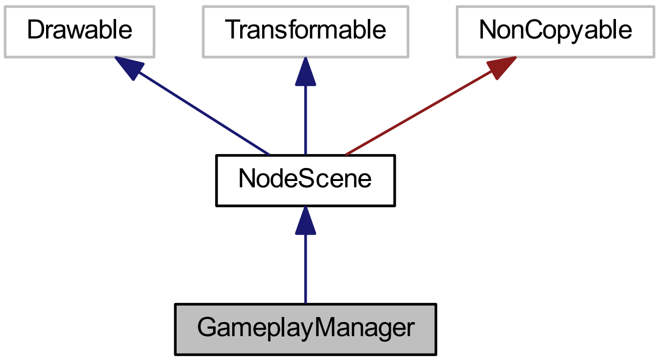
It handles the gameplay rules such as the order of moves – including the worm queue, and even the countdown timer for a given round or state.

```
#include <GameplayManager.h>
```

Inheritance diagram for GameplayManager:



Collaboration diagram for GameplayManager:



Public Member Functions

- **GameplayManager** (b2World &physicalWorld, TextureManager &textures, FontManager &fonts, sf::↩ RenderWindow &window)
- void addWorm (const std::string &name, sf::Color teamColor, sf::Vector2f position)

Creates a worm inside the game.
- void drawThis (sf::RenderTarget &target, sf::RenderStates states) const override

  Draws worms and other pinned nodes.
- void updateThis (sf::Time deltaTime) override

  Updates worms and other pinned nodes.
- void handleThisEvents (const sf::Event &event) override

  Passes on to worms and other pinned nodes the individual inputs made by the player.
- void removeDestroyed () override

  Checks that pinned knots and worms should be removed.
- void setWorldMessage (const std::string &text, sf::Color color=sf::Color::White, sf::Time time=sf::Time::Zero)

  Displays text below the timer for given amount of time.
- void addTime (sf::Time time)

  Adds an additional amount of time to the current state.
- bool anyBullet ()

  It checks if any bullet has been pinned to the scene.
- NodeScene ∗ getRootNode () override

  Returns itself as RootNode.
- const NodeScene ∗ getRootNode () const override

  Returns itself as RootNode.
- bool isGameFinished () const

  Checks whether the game has already finished.

Additional Inherited Members

1.16.1 Detailed Description

It handles the gameplay rules such as the order of moves – including the worm queue, and even the countdown timer for a given round or state.

1.16.2 Member Function Documentation

1.16.2.1 addTime()

```
void GameplayManager::addTime (
            sf::Time time )
```

Adds an additional amount of time to the current state.

Parameters

time	time to add

1.16.2.2 addWorm()

```
void GameplayManager::addWorm (
            const std::string & name,
            sf::Color teamColor,
            sf::Vector2f position )
```

Creates a worm inside the game.

Parameters

name	Name of the worm
teamColor	The team of the worm, and at the same time its colour.
position	Position of the worm on the map

1.16.2.3 anyBullet()

```
bool GameplayManager::anyBullet ( )
```

It checks if any bullet has been pinned to the scene.

Returns

True if there is any bullet pinned to the scene, false otherwise

1.16.2.4 drawThis()

```
void GameplayManager::drawThis (
            sf::RenderTarget & target,
            sf::RenderStates states ) const  [override], [virtual]
```

Draws worms and other pinned nodes.

Parameters

target	where it should be drawn to
states	provides information about rendering process (transform, shader, blend mode)

Reimplemented from NodeScene.

1.16.2.5 getRootNode() [1/2]

```
const NodeScene* GameplayManager::getRootNode ( ) const  [override], [virtual]
```

Returns itself as RootNode.

This ensures that any bullets are assigned to this node.

Returns

Itself (GameplayManager)

Reimplemented from NodeScene.

1.16.2.6 getRootNode() [2/2]

```
NodeScene* GameplayManager::getRootNode ( )  [override], [virtual]
```

Returns itself as RootNode.

This ensures that any bullets are assigned to this node.

Returns

Itself (GameplayManager)

Reimplemented from NodeScene.

1.16.2.7 handleThisEvents()

```
void GameplayManager::handleThisEvents (
             const sf::Event & event )  [override], [virtual]
```

Passes on to worms and other pinned nodes the individual inputs made by the player.

Parameters

event	user input

Reimplemented from NodeScene.

1.16.2.8 isGameFinished()

```
bool GameplayManager::isGameFinished ( ) const
```

Checks whether the game has already finished.

Returns

True if the game has finished, false otherwise

1.16.2.9 setWorldMessage()

```
void GameplayManager::setWorldMessage (
            const std::string & text,
            sf::Color color = sf::Color::White,
            sf::Time time = sf::Time::Zero )
```

Displays text below the timer for given amount of time.

Parameters

text	Text to display
color	Color of the text
time	How much time should it stay on the screen. If not set then it stays infinitely long

1.16.2.10 updateThis()

```
void GameplayManager::updateThis (
            sf::Time deltaTime ) [override], [virtual]
```

Updates worms and other pinned nodes.

Parameters

deltaTime	Time passed since the last frame

Reimplemented from NodeScene.

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

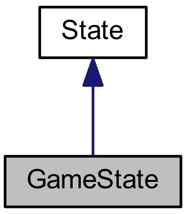
- Worms-Clone/Worms-Clone/Nodes/GameplayManager.h

1.17 GameState Class Reference

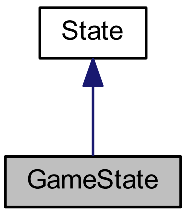
The game state in which the game world is created, all objects are placed and the processes inside the game world are controlled.

```
#include <GameState.h>
```

Inheritance diagram for GameState:



Collaboration diagram for GameState:



Public Member Functions

- **GameState** (StateStack &stack, sf::RenderWindow &window, int &wormAmount, int &numberOfTeams)
- void draw () const override

  Draws only this state.
- void draw (sf::RenderTarget &target, sf::RenderStates states) const override

  Draws only this state to the passed target.
- bool update (sf::Time deltaTime) override

  Updates the status/logic of the state.
- bool handleEvent (const sf::Event &event) override

  It takes input (event) from the user and interprets it.

Additional Inherited Members

1.17.1 Detailed Description

The game state in which the game world is created, all objects are placed and the processes inside the game world are controlled.

1.17.2 Member Function Documentation

1.17.2.1 draw()

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

Draws only this state to the passed target.

Parameters

target	where it should be drawn to
states	provides information about rendering process (transform, shader, blend mode)

Implements State.

1.17.2.2 handleEvent()

```
bool GameState::handleEvent (
             const sf::Event & event )  [override], [virtual]
```

It takes input (event) from the user and interprets it.

Parameters

event	user input

Implements State.

1.17.2.3 update()

```
bool GameState::update (
             sf::Time deltaTime )  [override], [virtual]
```

Updates the status/logic of the state.

Parameters

deltaTime	the time that has passed since the last frame.

Implements State.

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

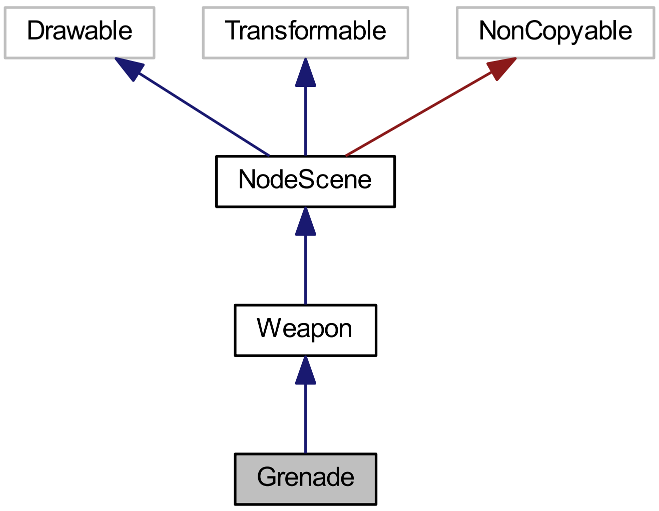
- Worms-Clone/Worms-Clone/States/Application_States/GameState.h

1.18 Grenade Class Reference

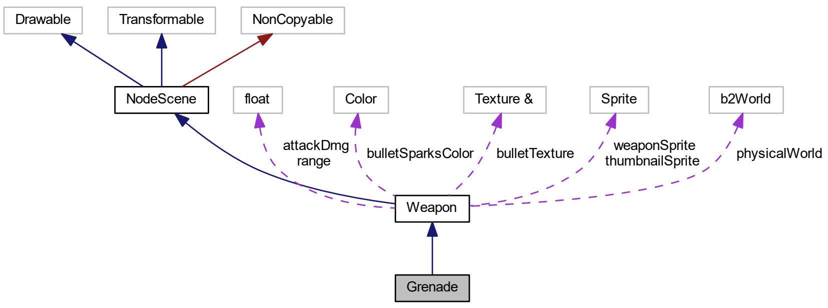
A grenade is a weapon that explodes after a specified.

`#include <Grenade.h>`

Inheritance diagram for Grenade:



Collaboration diagram for Grenade:



Public Member Functions

- **Grenade** (b2World &world, TextureManager &textures, const FontManager &fonts)
- void shoot (NodeScene ∗rootNode, sf::Vector2f position, sf::Vector2f force) override

  The firing function of a weapon that will be executed when the shooting bar is loaded.
- bool isActivation () override

  Is the grenade an activated weapon or a loaded weapon (via the shooting bar).
- bool isRoundEnding () override

  Does using the Grenade end the round.

Additional Inherited Members

1.18.1 Detailed Description

A grenade is a weapon that explodes after a specified.

Deals high damage and has a long range of effect

1.18.2 Member Function Documentation

1.18.2.1 isActivation()

```
bool Grenade::isActivation ( )  [override], [virtual]
```

Is the grenade an activated weapon or a loaded weapon (via the shooting bar).

Returns

True if it is activated weapon, false if it need to be loaded

Implements Weapon.

1.18.2.2 isRoundEnding()

```
bool Grenade::isRoundEnding ( )  [override], [virtual]
```

Does using the Grenade end the round.

Returns

True usage of grenade ends the round, false otherwise

Implements Weapon.

1.18.2.3 shoot()

```
void Grenade::shoot (
             NodeScene * rootNode,
             sf::Vector2f position,
             sf::Vector2f force )  [override], [virtual]
```

The firing function of a weapon that will be executed when the shooting bar is loaded.

Parameters

rootNode	A node that will serve as the parent of a bullet created and added to the game world.
position	The position at which the bullet will form.
force	The force with which the bullet will be fired

Reimplemented from Weapon.

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

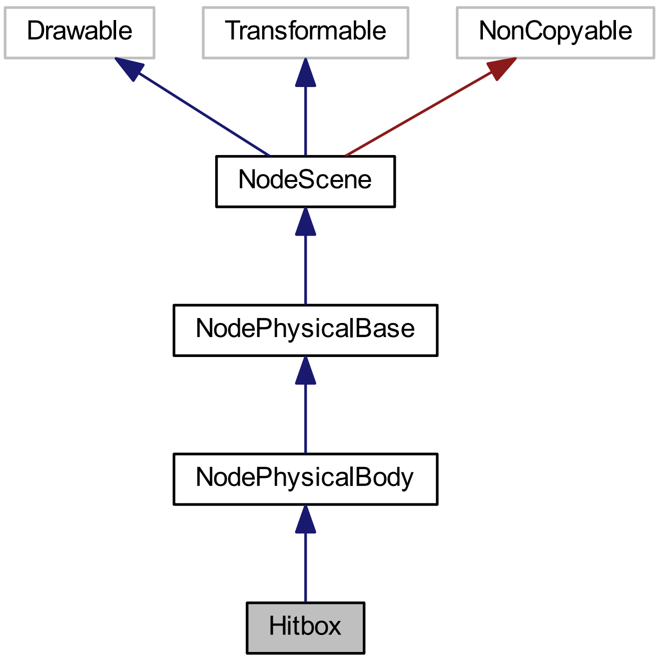
- Worms-Clone/Worms-Clone/Nodes/Physical/Specified/Worm/Weapons/Grenade.h

1.19 Hitbox Class Reference

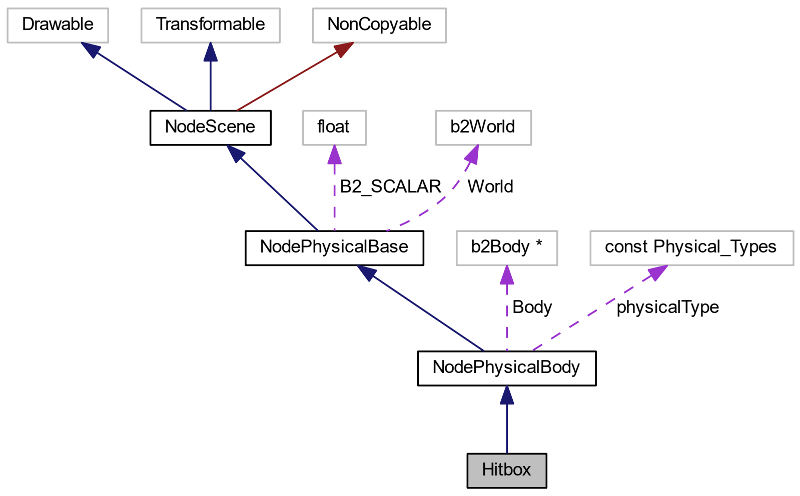
An object that reports a collision within its boundary.

```
#include <Hitbox.h>
```

Inheritance diagram for Hitbox:



Collaboration diagram for Hitbox:



Public Member Functions

- **Hitbox** (b2World &world, sf::Vector2f position, float area_of_range, float max_dmg)
- void updateThis (sf::Time deltaTime) override

  This function removes the object from the map as quickly as possible, so it only takes one iteration.

Friends

- class WorldListener

  Class that handles collisions inside the game.

Additional Inherited Members

1.19.1 Detailed Description

An object that reports a collision within its boundary.

Used to find worm hit by an explosion. Removes itself in the next iteration of the game.

1.19.2 Member Function Documentation

1.19.2.1 updateThis()

```
void Hitbox::updateThis (
            sf::Time deltaTime )  [override], [virtual]
```

This function removes the object from the map as quickly as possible, so it only takes one iteration.

Parameters

deltaTime	Time passed since the last frame

Reimplemented from NodeScene.

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

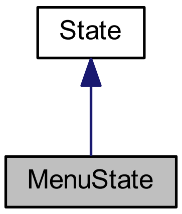
- Worms-Clone/Worms-Clone/Nodes/Physical/Specified/Worm/Weapons/Hitbox.h

1.20 MenuState Class Reference

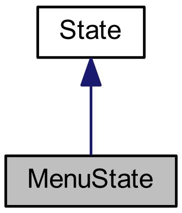
The main menu state, where the player can adjust the game settings and start the game.

```
#include <MenuState.h>
```

Inheritance diagram for MenuState:



Collaboration diagram for MenuState:



Public Member Functions

- **MenuState** (StateStack &stack, const FontManager &fonts, sf::RenderWindow &window, int &wormAmount, int &numberOfTeams)
- void createBackgroundWorld (sf::Vector2f pos)

  Creates a live background for the menu.
- void createGrenades (sf::Vector2f pos)

  Creates a point (spawner) that creates falling grenades.
- void draw () const override

  Draws only this state.
- void draw (sf::RenderTarget &target, sf::RenderStates states) const override

  Draws only this state to the passed target.
- bool update (sf::Time deltaTime) override

  Updates the status/logic of the state.
- bool handleEvent (const sf::Event &event) override

  It takes input (event) from the user and interprets it.

Additional Inherited Members

1.20.1 Detailed Description

The main menu state, where the player can adjust the game settings and start the game.

The menu is quite extensive, as it contains its own physical world.

1.20.2 Member Function Documentation

1.20.2.1 createBackgroundWorld()

```
void MenuState::createBackgroundWorld (
            sf::Vector2f pos )
```

Creates a live background for the menu.

Parameters

pos	The position on which the live background is to be created

1.20.2.2 createGrenades()

```
void MenuState::createGrenades (
            sf::Vector2f pos )
```

Creates a point (spawner) that creates falling grenades.

Parameters

pos	The position on which grenades are to spawn.

1.20.2.3 draw()

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

Draws only this state to the passed target.

Parameters

target	where it should be drawn to
states	provides information about rendering process (transform, shader, blend mode)

Implements State.

1.20.2.4 handleEvent()

```
bool MenuState::handleEvent (
             const sf::Event & event )  [override], [virtual]
```

It takes input (event) from the user and interprets it.

Parameters

event	user input

Implements State.

1.20.2.5 update()

```
bool MenuState::update (
             sf::Time deltaTime )  [override], [virtual]
```

Updates the status/logic of the state.

Parameters

deltaTime	the time that has passed since the last frame.

Implements State.

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

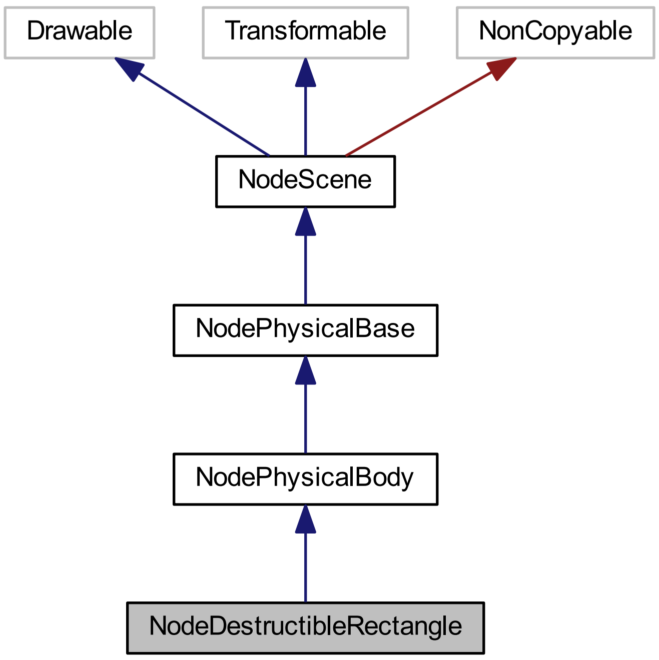
- Worms-Clone/Worms-Clone/States/Application_States/MenuState.h

1.21 NodeDestructibleRectangle Class Reference

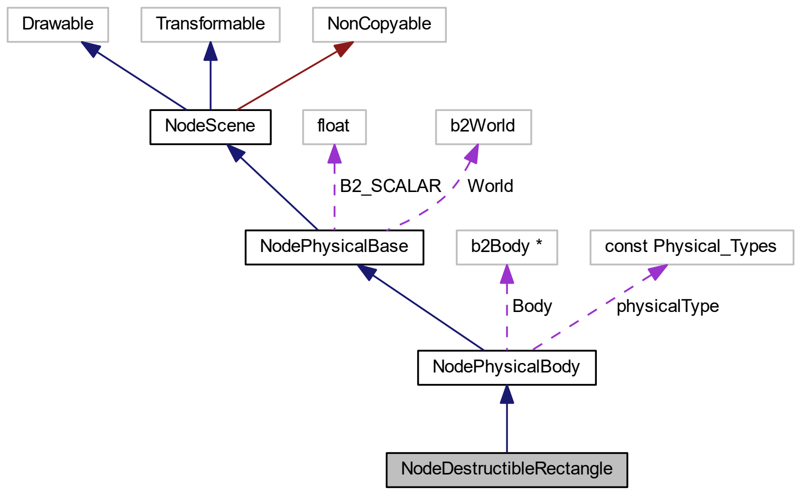
A node that has the properties of a static physical object – it physically interacts with other physical objects and can be damaged inside the game.

```
#include <NodeDestructibleRectangle.h>
```

Inheritance diagram for NodeDestructibleRectangle:



Collaboration diagram for NodeDestructibleRectangle:



Public Member Functions

- **NodeDestructibleRectangle** (b2World &world, sf::Vector2f position, const sf::Vector2f &size)
- void addHole (std::vector< ClipperLib::IntPoint > &figure)

    Calculates the overlapping part of both figures and performs the difference operation (removes the place where the figures overlap).
- void updateThis (sf::Time deltaTime) override

    It updates the current state of the object, synchronises it with its physical form and checks if it needs recalculation after impact.

Additional Inherited Members

1.21.1 Detailed Description

A node that has the properties of a static physical object – it physically interacts with other physical objects and can be damaged inside the game.

This object has a function that calculates the difference with another figure and removes the overlapping part of the object.

1.21.2 Member Function Documentation

1.21.2.1 addHole()

```
void NodeDestructibleRectangle::addHole (
            std::vector< ClipperLib::IntPoint > & figure )
```

Calculates the overlapping part of both figures and performs the difference operation (removes the place where the figures overlap).

Parameters

figure	Points that build a figure which at the intersection removes the current part of the node.

1.21.2.2 updateThis()

```
void NodeDestructibleRectangle::updateThis (
            sf::Time deltaTime )  [override], [virtual]
```

It updates the current state of the object, synchronises it with its physical form and checks if it needs recalculation after impact.

Parameters

deltaTime	Time elapsed since the previous frame

Reimplemented from NodeScene.

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

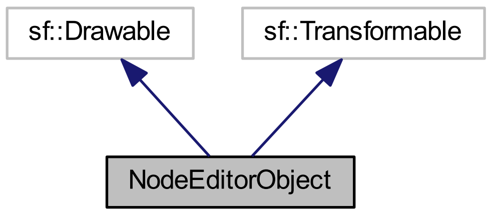
- Worms-Clone/Worms-Clone/Nodes/Physical/NodeDestructibleRectangle.h

1.22 NodeEditorObject Class Reference

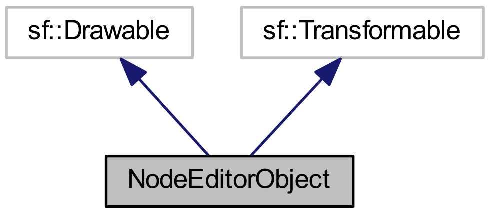
Editor object that allows to rotate and position on the map.

```
#include <NodeEditorObject.h>
```

Inheritance diagram for NodeEditorObject:



Collaboration diagram for NodeEditorObject:



Public Member Functions

- **NodeEditorObject** (const TextureManager &textures, const FontManager &fonts)
- void updateMouse (const sf::Vector2f &mousePosition)

  Updates object states related to mouse position.
- void update (sf::Time deltaTime)

  Updates the status/logic of the node.
- void draw (sf::RenderTarget &target, sf::RenderStates states) const override

  Draws this node to the passed target.
- void handleEvent (const sf::Event &event)

  It takes input (event) from the user and interprets it.
- void setName (const std::string &name)

  Sets the object's name (textual representation).
- void setSize (float width, float height)

  Sets the new object size.
- void setId (unsigned id)

  Sets the object the given identifier (which the object will now represent).
- sf::FloatRect getGlobalBounds () const

  Get the global bounding rectangle of the entity.
- sf::Vector2f getSize () const

  Returns the dimensions of this object.
- unsigned getId ()

  Returns the object identifier that this object represents.
- bool isSelected ()

  A function that checks if the mouse hovers over an object.
- void select ()

  A function that executes once when the user hovers over an object.
- void unselect ()

  The function that executes once when the user leaves the object with the cursor.
- bool isActivated ()

  Checks if objects are activated (user selected / pressed).
- void activate ()

  The function that is executed when user click on the object (check it) – otherwise known as deactivating the object.
- void deactivate ()

  The function that is executed when "unchecking" – otherwise known as deactivating the object.

- void setDestroyed ()

  A function that marks an object as ready for deletion.
- bool isDestroyed () const

  A function that checks if the object is ready to be deleted.

1.22.1 Detailed Description

Editor object that allows to rotate and position on the map.

Contains information needed for the map export to file

1.22.2 Member Function Documentation

1.22.2.1 draw()

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

Draws this node to the passed target.

Parameters

target	where it should be drawn to
states	provides information about rendering process (transform, shader, blend mode)

1.22.2.2 getGlobalBounds()

```
sf::FloatRect NodeEditorObject::getGlobalBounds ( ) const
```

Get the global bounding rectangle of the entity.

Returns

GLobal bounding rectangle of the entity

1.22.2.3 getId()

```
unsigned NodeEditorObject::getId ( )
```

Returns the object identifier that this object represents.

Returns

The object identifier that this object represents

1.22.2.4 getSize()

```
sf::Vector2f NodeEditorObject::getSize ( ) const
```

Returns the dimensions of this object.

Returns

Dimensions of this object

1.22.2.5 handleEvent()

```
void NodeEditorObject::handleEvent (
             const sf::Event & event )
```

It takes input (event) from the user and interprets it.

Parameters

event	user input

1.22.2.6 isActivated()

```
bool NodeEditorObject::isActivated ( )
```

Checks if objects are activated (user selected / pressed).

Returns

True if it is activated, false otherwise

1.22.2.7 isDestroyed()

```
bool NodeEditorObject::isDestroyed ( ) const
```

A function that checks if the object is ready to be deleted.

Returns

True if ready for removal, false otherwise

1.22.2.8 isSelected()

```
bool NodeEditorObject::isSelected ( )
```

A function that checks if the mouse hovers over an object.

Returns

True if user hovers with the mouse over an object, false otherwise

1.22.2.9 setId()

```
void NodeEditorObject::setId (
             unsigned id )
```

Sets the object the given identifier (which the object will now represent).

Parameters

id	Identifier of object to represent.

1.22.2.10 setName()

```
void NodeEditorObject::setName (
             const std::string & name )
```

Sets the object's name (textual representation).

Parameters

name	New name of the object

1.22.2.11 setSize()

```
void NodeEditorObject::setSize (
             float width,
             float height )
```

Sets the new object size.

Parameters

width	New object width
height	New object height

1.22.2.12 update()

```
void NodeEditorObject::update (
            sf::Time deltaTime )
```

Updates the status/logic of the node.

Parameters

deltaTime	the time that has passed since the last frame.

1.22.2.13 updateMouse()

```
void NodeEditorObject::updateMouse (
            const sf::Vector2f & mousePosition )
```

Updates object states related to mouse position.

Parameters

mousePosition	Current mouse coordinates inside the game world

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

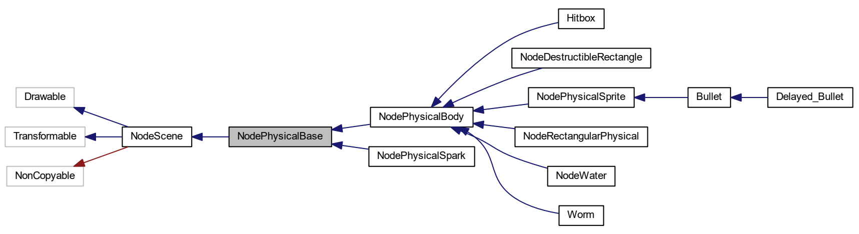
- Worms-Clone/Worms-Clone/Level Editor/NodeEditorObject.h

1.23 NodePhysicalBase Class Reference

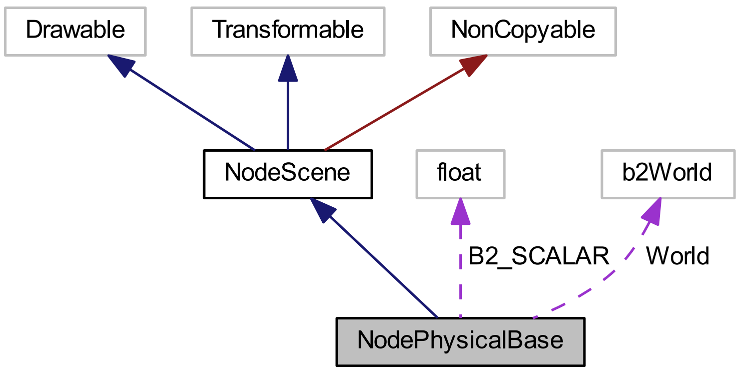
An abstract class of physical object.

```
#include <NodePhysicalBase.h>
```

Inheritance diagram for NodePhysicalBase:



Collaboration diagram for NodePhysicalBase:



Public Member Functions

- **NodePhysicalBase** (b2World &world)
- void update (sf::Time deltaTime) override final

    Updates the object logic of this node and the pinned nodes.
- virtual void **updatePhysics** ()=0

Static Public Member Functions

- template<typename Vector2 >
  static b2Vec2 sfVectorToB2Vec (const Vector2 &vec)

    Converts from pixels to meters.
- template<typename Vector2 >
  static Vector2 b2VecToSfVector (const b2Vec2 &vec)

    Converts from meters to pixels.
- static float angleToRadians (const float &angle)

    Converts from angle to radians.
- static float radiansToAngle (const float &radians)

    Converts from radians to angles.

Static Public Attributes

- static const float B2_SCALAR

  It is used to scale meters to pixels.

Protected Attributes

- b2World ∗ World

  Simulation of the physical world.

Additional Inherited Members

1.23.1 Detailed Description

An abstract class of physical object.

Contains helper functions to convert between angles, radians and units of SFML and BOX2D libraries

1.23.2 Member Function Documentation

1.23.2.1 angleToRadians()

```
static float NodePhysicalBase::angleToRadians (
             const float & angle )  [static]
```

Converts from angle to radians.

Parameters

angle	angle given in degrees

Returns

Angle given in radians

Useful for conversion between Box2D and the SFML

1.23.2.2 b2VecToSfVector()

```
template<typename Vector2 >
Vector2 NodePhysicalBase::b2VecToSfVector (
             const b2Vec2 & vec )  [static]
```

Converts from meters to pixels.

Template Parameters

Vector2	Vector from SFML Library

Parameters

vec	Vector from SFML Library

Returns

Vector from Box2D library

1.23.2.3 radiansToAngle()

```
static float NodePhysicalBase::radiansToAngle (
             const float & radians )  [static]
```

Converts from radians to angles.

Parameters

radians	angle given in radians

Returns

Angle given in degrees

Useful for conversion between Box2D and the SFML

1.23.2.4 sfVectorToB2Vec()

```
template<typename Vector2 >
b2Vec2 NodePhysicalBase::sfVectorToB2Vec (
             const Vector2 & vec )  [static]
```

Converts from pixels to meters.

Template Parameters

Vector2	Vector from SFML Library

Parameters

vec	Vector from SFML Library

Returns

Vector from Box2D library

1.23.2.5 update()

```
void NodePhysicalBase::update (
            sf::Time deltaTime )  [final], [override], [virtual]
```

Updates the object logic of this node and the pinned nodes.

Parameters

deltaTime	the time that has passed since the game was last updated.

In comparison to draw() this function is not derived. It is used to update all pinnedNodes

Reimplemented from NodeScene.

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

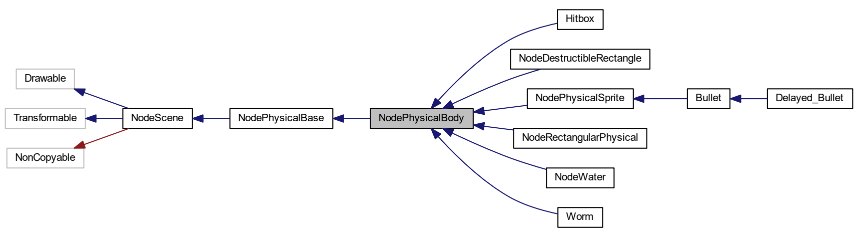
- Worms-Clone/Worms-Clone/Nodes/Physical/NodePhysicalBase.h

1.24 NodePhysicalBody Class Reference

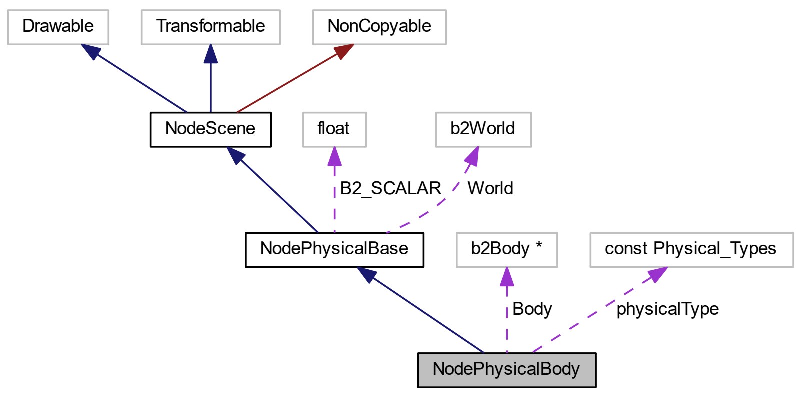
A node containing a physical body inside a simulation of the physical world.

```
#include <NodePhysicalBody.h>
```

Inheritance diagram for NodePhysicalBody:



Collaboration diagram for NodePhysicalBody:



Public Types

- enum Physical_Types { Physical_Types::Kinematic_Type, Physical_Types::Static_Type, Physical_Types::Dynamic_Type }

Public Member Functions

- **NodePhysicalBody** (b2World &world, Physical_Types physical_type, sf::Vector2f position)
- void updatePhysics () override

  Synchronises a graphical object with a simulation inside the physical world.
- void applyForce (sf::Vector2f vector)

  Applies force under the indicated vector.
- void setRotation (float angle)

  Rotates the physical objects.

Protected Attributes

- const Physical_Types physicalType

  Defines the physical type of this object.
- b2Body $*$ Body

  The physical body of an object inside a simulation of the physical world.

Additional Inherited Members

1.24.1 Detailed Description

A node containing a physical body inside a simulation of the physical world.

It can be one of three physical types. It synchronises with the physical simulation on an ongoing basis.

1.24.2 Member Enumeration Documentation

1.24.2.1 Physical_Types

```
enum NodePhysicalBody::Physical_Types  [strong]
```

Enumerator

Kinematic_Type	can move but it will not be affected by other bodies
Static_Type	never moves
Dynamic_Type	body can interact with other bodies

1.24.3 Member Function Documentation

1.24.3.1 applyForce()

```
void NodePhysicalBody::applyForce (
             sf::Vector2f vector )
```

Applies force under the indicated vector.

The force is directed towards the centre of the object.

Parameters

vector	Vector with force to be applied

1.24.3.2 setRotation()

```
void NodePhysicalBody::setRotation (
             float angle )
```

Rotates the physical objects.

Parameters

angle	The angle at which the object should be tilted.

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

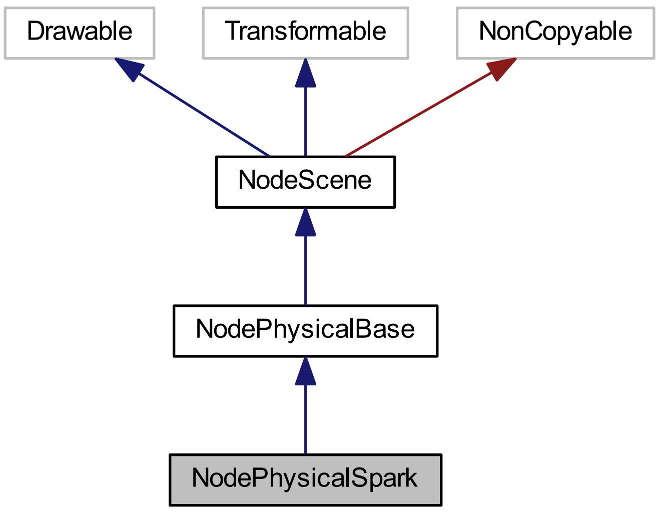
- Worms-Clone/Worms-Clone/Nodes/Physical/NodePhysicalBody.h

1.25 NodePhysicalSpark Class Reference

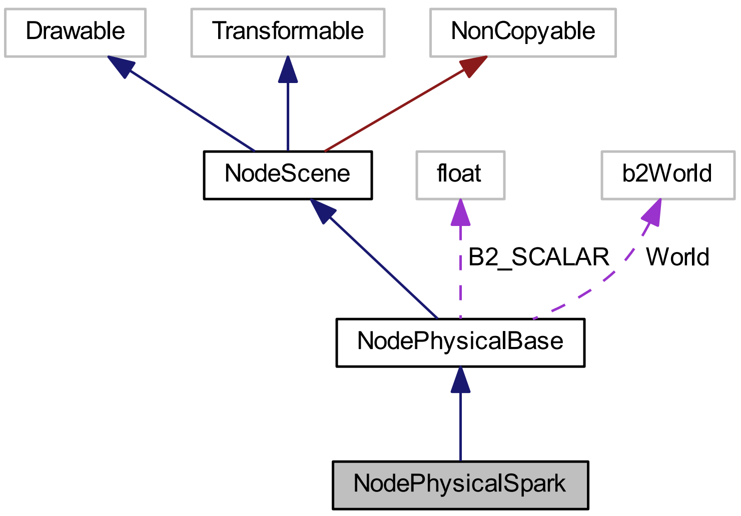
An object that creates tiny objects that escape all around from the place of this object.

```
#include <NodePhysicalSpark.h>
```

Inheritance diagram for NodePhysicalSpark:



Collaboration diagram for NodePhysicalSpark:



Public Member Functions

- **NodePhysicalSpark** (b2World &world, sf::Vector2f position, sf::Color color=sf::Color::White)
- void drawThis (sf::RenderTarget &target, sf::RenderStates states) const override

  Draws all particles to the passed target.
- void updateThis (sf::Time deltaTime) override

  Updates the current state of particles.
- void updatePhysics () override

  It synchronises all created particles with their position inside the physical simulation.

Additional Inherited Members

1.25.1 Detailed Description

An object that creates tiny objects that escape all around from the place of this object.

The object is removed when they are created.

It is used to graphically visualize explosions of various objects.

1.25.2 Member Function Documentation

1.25.2.1 drawThis()

```
void NodePhysicalSpark::drawThis (
            sf::RenderTarget & target,
            sf::RenderStates states ) const  [override], [virtual]
```

Draws all particles to the passed target.

Parameters

target	where it should be drawn to
states	provides information about rendering process (transform, shader, blend mode)

Reimplemented from NodeScene.

1.25.2.2 updateThis()

```
void NodePhysicalSpark::updateThis (
            sf::Time deltaTime )  [override], [virtual]
```

Updates the current state of particles.

Parameters

deltaTime	Time passed since the last frame

It controls where particles should to disappear.

Reimplemented from NodeScene.

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

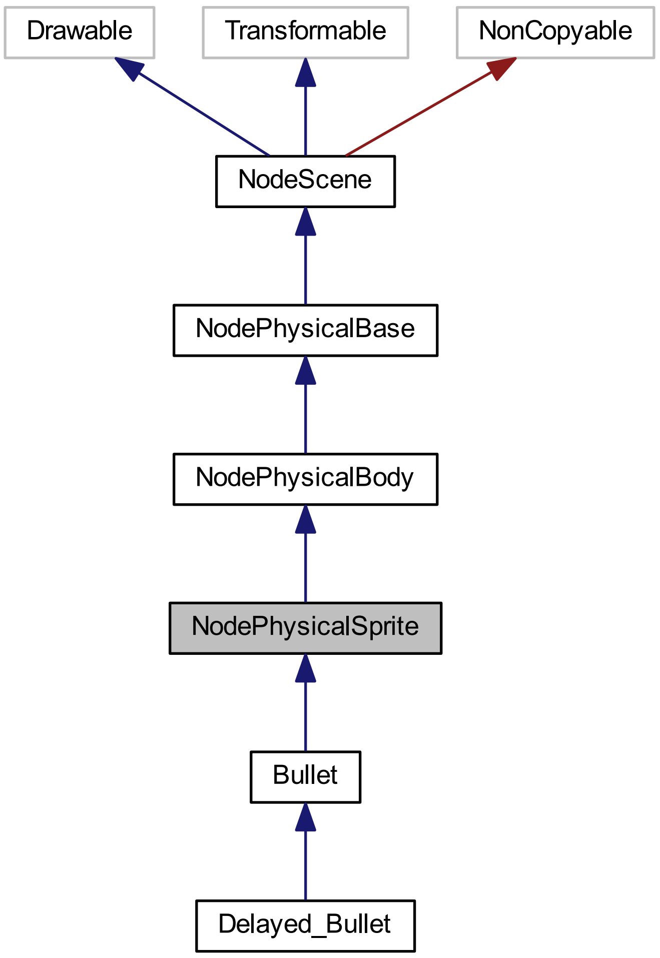
- Worms-Clone/Worms-Clone/Nodes/Physical/NodePhysicalSpark.h

1.26 NodePhysicalSprite Class Reference

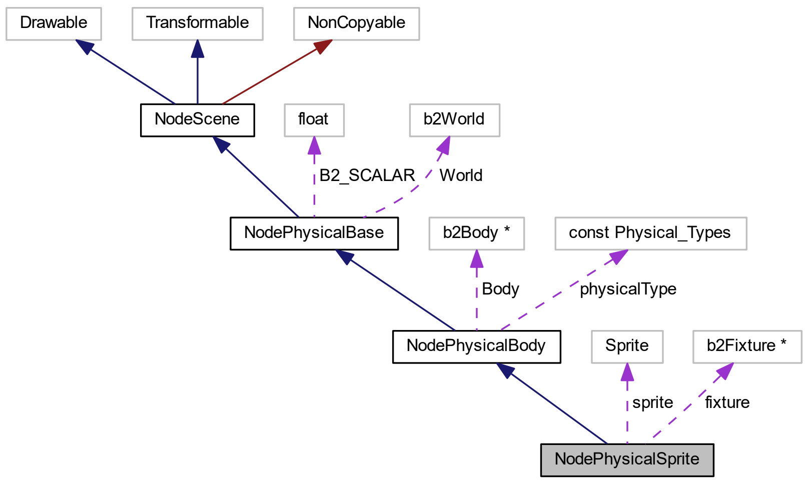
A node that has its own graphical representation and at the same time is a physical object inside the game world.

```
#include <NodePhysicalSprite.h>
```

Inheritance diagram for NodePhysicalSprite:



Collaboration diagram for NodePhysicalSprite:



Public Member Functions

- **NodePhysicalSprite** (b2World &world, Physical_Types physical_type, sf::Texture &texture, sf::Vector2f position)
- **NodePhysicalSprite** (b2World &world, Physical_Types physical_type, sf::Texture &texture, sf::Vector2f position, const sf::IntRect &rect)
- **NodePhysicalSprite** (b2World &world, Physical_Types physical_type, sf::Texture &texture, sf::Vector2f position, const sf::Vector2f &size)
- void drawThis (sf::RenderTarget &target, sf::RenderStates states) const override

  Draws only this node to the passed target.
- void updateThis (sf::Time deltaTime) override

  Updates the current state of the sprite.
- void setSize (const sf::IntRect &)

  Sets the texture size and enhances the body inside the simulation adjusting it to the new dimensions.

Protected Attributes

- sf::Sprite sprite

  Graphical representation of the object.
- b2Fixture ∗ fixture

  Attaches a shape to a body.

Additional Inherited Members

1.26.1 Detailed Description

A node that has its own graphical representation and at the same time is a physical object inside the game world.

1.26.2 Member Function Documentation

1.26.2.1 drawThis()

```
void NodePhysicalSprite::drawThis (
            sf::RenderTarget & target,
            sf::RenderStates states ) const  [override], [virtual]
```

Draws only this node to the passed target.

Parameters

target	where it should be drawn to
states	provides information about rendering process (transform, shader, blend mode)

Reimplemented from NodeScene.

Reimplemented in Delayed_Bullet.

1.26.2.2 updateThis()

```
void NodePhysicalSprite::updateThis (
            sf::Time deltaTime )  [override], [virtual]
```

Updates the current state of the sprite.

Parameters

deltaTime	Time passed since the last frame

It synchronizes the sprite with the physics world

Reimplemented from NodeScene.

Reimplemented in Delayed_Bullet, and Bullet.

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

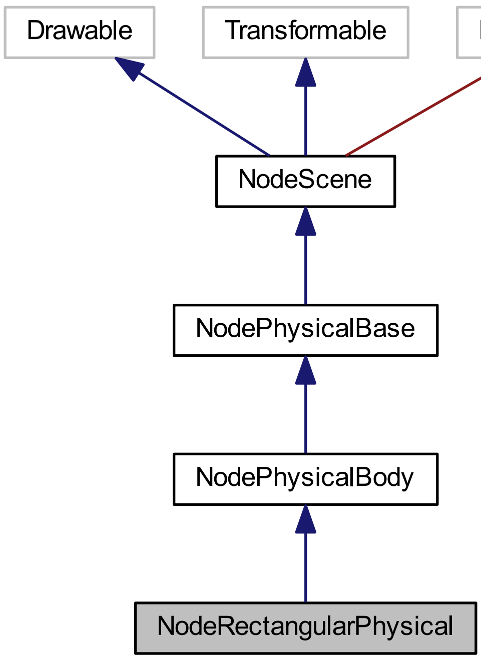
- Worms-Clone/Worms-Clone/Nodes/Physical/NodePhysicalSprite.h

1.27 NodeRectangularPhysical Class Reference

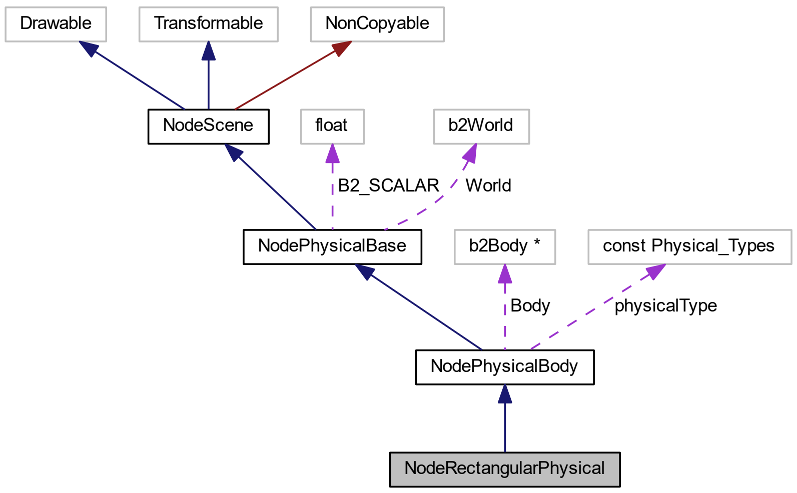
A physical and graphical object representing a rectangle within a game.

```
#include <NodeRectangularPhysical.h>
```

Inheritance diagram for NodeRectangularPhysical:



Collaboration diagram for NodeRectangularPhysical:



Public Member Functions

- **NodeRectangularPhysical** (b2World &world, sf::Vector2f size, sf::Vector2f position, sf::Color color, Physical_Types physical_type=Physical_Types::Kinematic_Type)
- void drawThis (sf::RenderTarget &target, sf::RenderStates states) const override

  Draws only this node to the passed target.

Additional Inherited Members

1.27.1 Detailed Description

A physical and graphical object representing a rectangle within a game.

1.27.2 Member Function Documentation

1.27.2.1 drawThis()

```
void NodeRectangularPhysical::drawThis (
            sf::RenderTarget & target,
            sf::RenderStates states ) const  [override], [virtual]
```

Draws only this node to the passed target.

Parameters

target	where it should be drawn to
states	provides information about rendering process (transform, shader, blend mode)

Reimplemented from NodeScene.

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

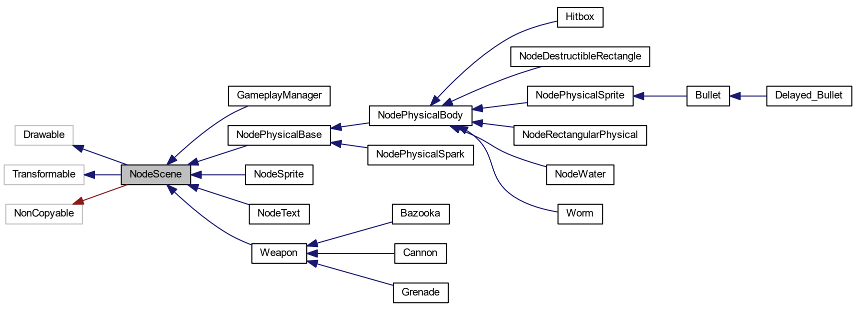
- Worms-Clone/Worms-Clone/Nodes/Physical/NodeRectangularPhysical.h

1.28 NodeScene Class Reference

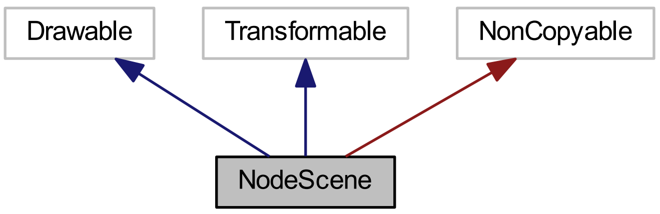
Scene on which objects and other pinned scenes will be drawn.

```
#include <NodeScene.h>
```

Inheritance diagram for NodeScene:



Collaboration diagram for NodeScene:



Public Types

- using **Node** = std::unique_ptr< NodeScene >

Public Member Functions

- void pinNode (Node node)

  Steals ownership, and puts it into the vector of pinned nodes.
- Node unpinNode (const NodeScene &node)

  Removes this NodeScene from pinnedNodes and returns it.
- sf::Vector2f getAbsolutePosition () const

  A function that calculates the absolute position of this node.
- void draw (sf::RenderTarget &target, sf::RenderStates states) const override final

  Draws this node, and all pinned nodes to the passed target.
- virtual void drawThis (sf::RenderTarget &target, sf::RenderStates states) const

  Draws only this node to the passed target.
- virtual void update (sf::Time deltaTime)

  Updates the object logic of this node and the pinned nodes.
- virtual void updateThis (sf::Time deltaTime)

  Updates the logic of this node only.

- virtual void removeDestroyed ()

    Deletes pinned nodes that are marked as deleted.
- void handleEvents (const sf::Event &event)

    It takes input (event) from the user and interprets it, then passes it on to the pinned nodes.
- virtual void handleThisEvents (const sf::Event &event)

    It takes input (event) from the user and interprets it.
- virtual bool isDestroyed ()

    Checks whether the object is to be deleted.
- virtual void setDestroyed ()

    Indicates the object as ready for deletion.

Protected Member Functions

- virtual NodeScene ∗ getRootNode ()

    Recursively finds the root node.
- virtual const NodeScene ∗ getRootNode () const

    Recursively finds the root node.

Friends

- class **GameplayManager**

1.28.1 Detailed Description

Scene on which objects and other pinned scenes will be drawn.

Individual nodes (objects inside the game) can be parents of other nodes. Thus, moving the parent will make the relativistic position of the nodes (children) pinned to it also to change. This makes it very easy to create any hierarchies and scenes within the game. Nodes that are pinned to a parent node at the time of drawing receive **transform** of node they are pinned to, so that they can draw themselves relative to their parent.

We can say that the way the nodes are executed resembles "Depth-first search". It starts at the root node, and it explores as far as possible along each branch before backtracking. This also means that objects higher up in the hierarchy (parents) will be drawn on the screen underneath their pinned nodes (children).

```
                +-----------+
                |(Root) Node|
                +-----+-----+
                      |
+---------+-----+-----------+
|         |                 |
```

+-V–+ +-V–+ +-V–+ (1)|Node| (2)|Node| |Node|(7) +-—+ +-+–+ +-+–+ | | +-V–+ +-V–+ (3)|Node| |Node|(8) +-—-+- +–+-—+ +-+–+ | | | | +-V–+ +-V–+ +–V-+ +-V–+ |Node| |Node| |Node| |Node|(9) +-—+ +-—+ +-—+ +-—+ (4) (5) (6)

It derives from sf::Drawable as it is suppossed to be drawn on the screen. It derives from sf::Transformable which gives all members related with position, rotation and scale It derives from sf::NonCopyable as NodeScene like this should not be copied (it may give many problems in current state) This make copy constructor/assignment to be deleted

1.28.2 Member Function Documentation

1.28.2.1 draw()

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

Draws this node, and all pinned nodes to the passed target.

Parameters

target	where this node, and all pinned nodes should be drawn to.
states	provides information about rendering process (transform, shader, blend mode)

This function is provided inside sf::Drawable. Thanks to this if we pass this object to sf::RenderWindow::draw(), then it will implicitly call this function to draw it!

1.28.2.2 drawThis()

```
virtual void NodeScene::drawThis (
             sf::RenderTarget & target,
             sf::RenderStates states ) const  [virtual]
```

Draws only this node to the passed target.

Parameters

target	where it should be drawn to
states	provides information about rendering process (transform, shader, blend mode)

Reimplemented in Weapon, Worm, GameplayManager, Delayed_Bullet, NodePhysicalSpark, NodePhysicalSprite, and NodeRectangularPhysical.

1.28.2.3 getAbsolutePosition()

```
sf::Vector2f NodeScene::getAbsolutePosition ( ) const
```

A function that calculates the absolute position of this node.

Returns

Returns the absolute position of the node object on the screen.

1.28.2.4 getRootNode() [1/2]

```
virtual NodeScene* NodeScene::getRootNode ( )  [protected], [virtual]
```

Recursively finds the root node.

Returns

the root node

Reimplemented in GameplayManager.

1.28.2.5 getRootNode() [2/2]

```
virtual const NodeScene* NodeScene::getRootNode ( ) const  [protected], [virtual]
```

Recursively finds the root node.

Returns

the root node

Reimplemented in GameplayManager.

1.28.2.6 handleEvents()

```
void NodeScene::handleEvents (
             const sf::Event & event )
```

It takes input (event) from the user and interprets it, then passes it on to the pinned nodes.

Parameters

event	user input

1.28.2.7 handleThisEvents()

```
virtual void NodeScene::handleThisEvents (
             const sf::Event & event )  [virtual]
```

It takes input (event) from the user and interprets it.

Parameters

event	user input

Reimplemented in Worm, and GameplayManager.

1.28.2.8 isDestroyed()

```
virtual bool NodeScene::isDestroyed ( )  [virtual]
```

Checks whether the object is to be deleted.

Returns

True if the object should be deleted, false if not

Reimplemented in Worm, and Bullet.

1.28.2.9 pinNode()

```
void NodeScene::pinNode (
             Node node )
```

Steals ownership, and puts it into the vector of pinned nodes.

Parameters

node	node to steal

After this function, the "node" is a pinned node to this node. The node that uses this function becomes the parent of the "node".

1.28.2.10 removeDestroyed()

```
virtual void NodeScene::removeDestroyed ( )  [virtual]
```

Deletes pinned nodes that are marked as deleted.

Performs operations for lower nodes as well.

Reimplemented in GameplayManager.

1.28.2.11 unpinNode()

```
Node NodeScene::unpinNode (
             const NodeScene & node )
```

Removes this NodeScene from pinnedNodes and returns it.

Parameters

node	node to unpin

Returns

unique_ptr to the unpinned node

Removes this NodeScene from pinned_Nodes and returns it. After this operation, this node is no longer a parent "node".

1.28.2.12 update()

```
virtual void NodeScene::update (
             sf::Time deltaTime )  [virtual]
```

Updates the object logic of this node and the pinned nodes.

Parameters

deltaTime	the time that has passed since the game was last updated.

In comparison to draw() this function is not derived. It is used to update all pinnedNodes

Reimplemented in NodePhysicalBase.

1.28.2.13 updateThis()

```
virtual void NodeScene::updateThis (
             sf::Time deltaTime )  [virtual]
```

Updates the logic of this node only.

Parameters

deltaTime	the time that has passed since the game was last updated

Works analogues to the drawThis(), updates all things related to itself

Reimplemented in Weapon, Worm, GameplayManager, NodeDestructibleRectangle, NodePhysicalSpark, NodePhysicalSprite, Hitbox, Delayed_Bullet, and Bullet.

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

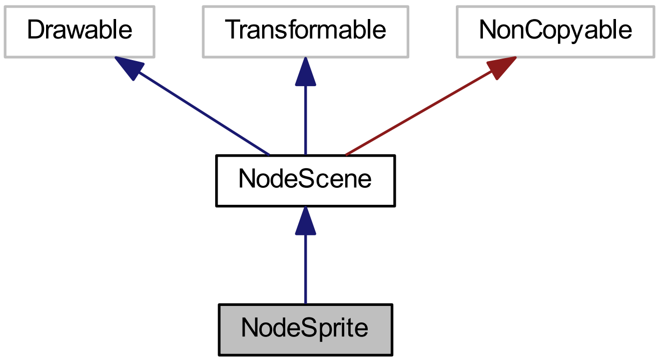
- Worms-Clone/Worms-Clone/Nodes/NodeScene.h

1.29 NodeSprite Class Reference

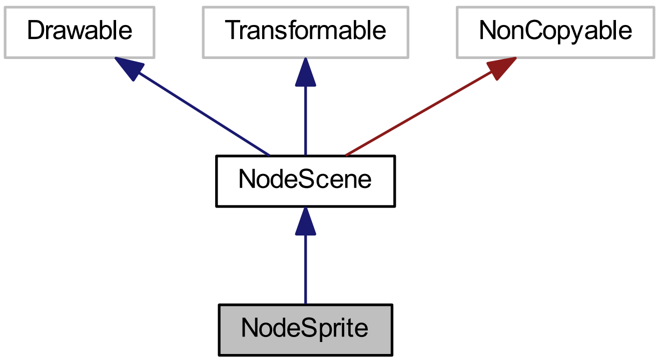
Node containing and displaying a sprite in addition.

`#include <NodeSprite.h>`

Inheritance diagram for NodeSprite:



Collaboration diagram for NodeSprite:



Public Member Functions

- NodeSprite (const sf::Texture &texture)

  Texture to be displayed.
- NodeSprite (const sf::Texture &texture, const sf::IntRect &rect)

  Creates a sprite with given texture and given boundaries.
- sf::Vector2f getSpriteSize () const

  Calculates and returns the dimensions of the sprite.

Additional Inherited Members

1.29.1 Detailed Description

Node containing and displaying a sprite in addition.

1.29.2 Constructor & Destructor Documentation

1.29.2.1 NodeSprite() [1/2]

```
NodeSprite::NodeSprite (
            const sf::Texture & texture )  [explicit]
```

Texture to be displayed.

Parameters

texture	Creates a sprite with given texture

1.29.2.2 NodeSprite() [2/2]

```
NodeSprite::NodeSprite (
            const sf::Texture & texture,
            const sf::IntRect & rect )
```

Creates a sprite with given texture and given boundaries.

Parameters

texture	Texture to be displayed
rect	boundaries to display only given part of the Texture

1.29.3 Member Function Documentation

1.29.3.1 getSpriteSize()

```
sf::Vector2f NodeSprite::getSpriteSize ( ) const
```

Calculates and returns the dimensions of the sprite.

Returns

Dimensions of the sprite

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

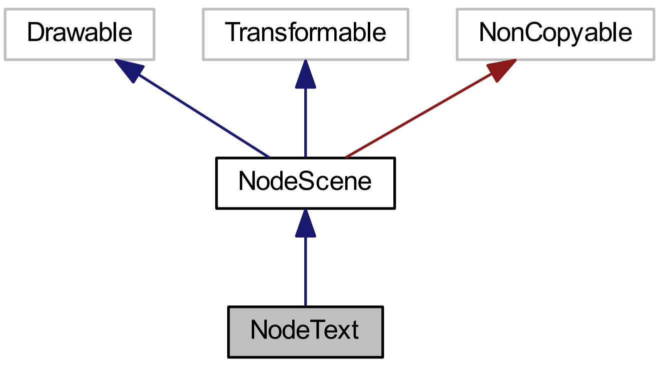
- Worms-Clone/Worms-Clone/Nodes/NodeSprite.h

1.30 NodeText Class Reference

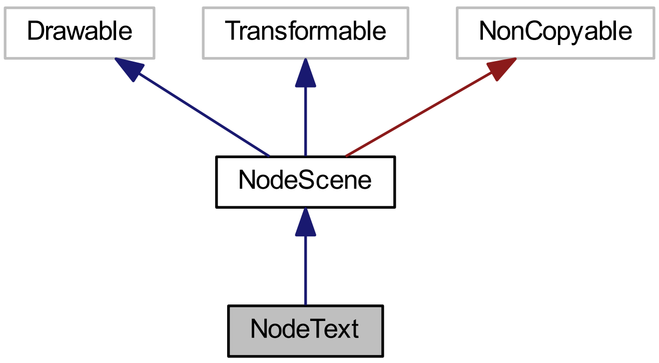
Node that displays text on the screen.

```
#include <NodeText.h>
```

Inheritance diagram for NodeText:



Collaboration diagram for NodeText:



Public Member Functions

- NodeText (const sf::Font &font, unsigned int size, bool outline)

    Creates a node representing text with given parameters.
- NodeText (const sf::Font &font, const std::string &text, unsigned int size, bool outline)

    Creates a node representing text with given parameters.
- void setString (const std::string &text)

    Sets the specified text.
- void setOutline (float thickness, sf::Color color)

    Sets a border for text of a given color.
- void setColor (sf::Color color)

    Sets the text color.
- void setSize (unsigned int size)

    Sets the text size.

Additional Inherited Members

1.30.1 Detailed Description

Node that displays text on the screen.

1.30.2 Constructor & Destructor Documentation

1.30.2.1 NodeText() `[1/2]`

```
NodeText::NodeText (
            const sf::Font & font,
            unsigned int size,
            bool outline )
```

Creates a node representing text with given parameters.

Parameters

font	The font to be used when displaying the text
size	Size in pixels to be used when displaying the text
outline	True if the text should have an outline, false otherwise

1.30.2.2 NodeText() [2/2]

```
NodeText::NodeText (
             const sf::Font & font,
             const std::string & text,
             unsigned int size,
             bool outline )
```

Creates a node representing text with given parameters.

Parameters

text	Text that should be displayed
font	The font to be used when displaying the text
size	Size in pixels to be used when displaying the text
outline	True if the text should have an outline, false otherwise

1.30.3 Member Function Documentation

1.30.3.1 setColor()

```
void NodeText::setColor (
             sf::Color color )
```

Sets the text color.

Parameters

color	Color of the text

1.30.3.2 setOutline()

```
void NodeText::setOutline (
             float thickness,
             sf::Color color )
```

Sets a border for text of a given color.

Parameters

thickness	Thickness of the outline
color	Color of the outline

1.30.3.3 setSize()

```
void NodeText::setSize (
            unsigned int size )
```

Sets the text size.

Parameters

size	Size of the text in pixels

1.30.3.4 setString()

```
void NodeText::setString (
            const std::string & text )
```

Sets the specified text.

Parameters

text	Text to set

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

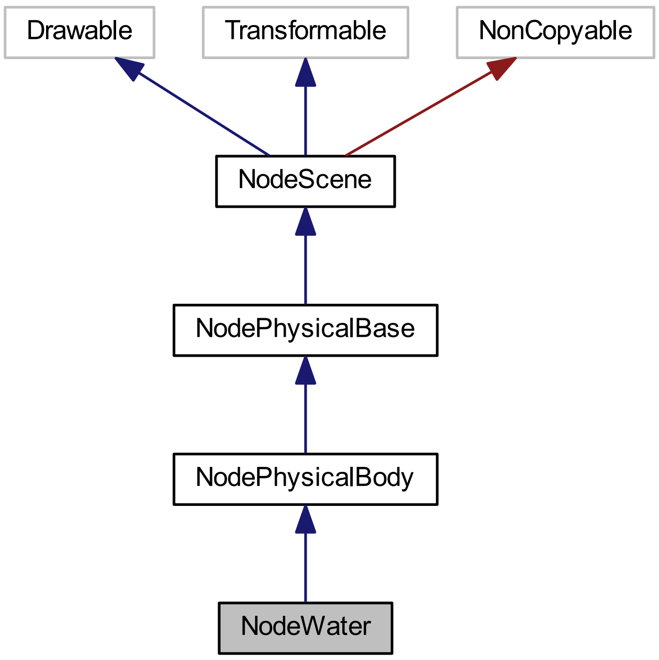
- Worms-Clone/Worms-Clone/Nodes/NodeText.h

1.31 NodeWater Class Reference

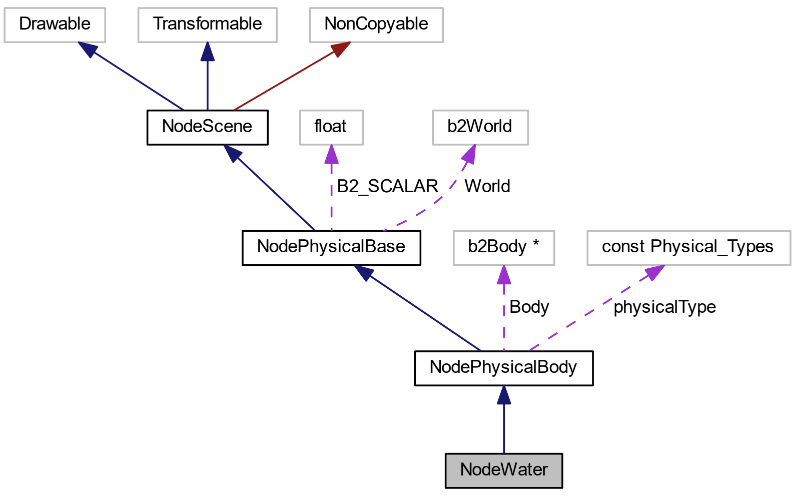
A very simple (a kind of prototype) node that looks like water.

```
#include <NodeWater.h>
```

Inheritance diagram for NodeWater:



Collaboration diagram for NodeWater:



Classes

- struct WaterLayer

A structure showing the layer and its parameters, such as: sprite, moving speed or current texture offset.

Public Member Functions

- **NodeWater** (b2World &world, const sf::Texture &texture)
- void setSize (float width, float height)

  Sets the size o the object.
- void setPosition (float x, float y)

  Sets the position of the water relative to the centre of the object.

Additional Inherited Members

1.31.1 Detailed Description

A very simple (a kind of prototype) node that looks like water.

When it comes into contact with a physical object, it removes it from the world.

1.31.2 Member Function Documentation

1.31.2.1 setPosition()

```
void NodeWater::setPosition (
             float x,
             float y )
```

Sets the position of the water relative to the centre of the object.

Parameters

x	The new position on the x-axis
y	the new position on the y-axis

1.31.2.2 setSize()

```
void NodeWater::setSize (
             float width,
             float height )
```

Sets the size o the object.

Parameters

width	The new width of the object
height	The new height of the object

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

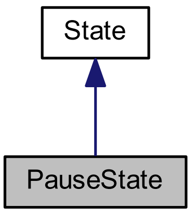
- Worms-Clone/Worms-Clone/Nodes/Physical/NodeWater.h

1.32 PauseState Class Reference

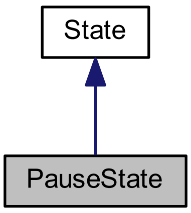
A pause state in which the states underneath this state in the statestack are blocked against state updates, although still can draw onto the screen.

```
#include <PauseState.h>
```

Inheritance diagram for PauseState:



Collaboration diagram for PauseState:



Public Member Functions

- **PauseState** (StateStack &stack, sf::RenderWindow &window, const FontManager &fonts)
- void draw () const override

  Draws only this state.
- void draw (sf::RenderTarget &target, sf::RenderStates states) const override

  Draws only this state to the passed target.
- bool update (sf::Time deltaTime) override

  Updates the status/logic of the state.
- bool handleEvent (const sf::Event &event) override

  It takes input (event) from the user and interprets it.

Additional Inherited Members

1.32.1 Detailed Description

A pause state in which the states underneath this state in the statestack are blocked against state updates, although still can draw onto the screen.

1.32.2 Member Function Documentation

1.32.2.1 draw()

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

Draws only this state to the passed target.

Parameters

target	where it should be drawn to
states	provides information about rendering process (transform, shader, blend mode)

Implements State.

1.32.2.2 handleEvent()

```
bool PauseState::handleEvent (
             const sf::Event & event )  [override], [virtual]
```

It takes input (event) from the user and interprets it.

Parameters

event	user input

Implements State.

1.32.2.3 update()

```
bool PauseState::update (
            sf::Time deltaTime )  [override], [virtual]
```

Updates the status/logic of the state.

Parameters

deltaTime	the time that has passed since the last frame.

Implements State.

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

- Worms-Clone/Worms-Clone/States/Application_States/PauseState.h

1.33 ResourceManager< Resource, Identifier > Class Template Reference

Public Member Functions

- Resource & getResourceReference (Identifier id)

    Checks an individual identifier and returns the resource assigned to it.
- const Resource & getResourceReference (Identifier id) const

    Checks an individual identifier and returns the resource assigned to it.
- Resource **getResourceCopy** (Identifier id) const
- void storeResource (Identifier id, const std::string &path_to_file)

    Assigns a given identifier to a specific resource retrieved from the specified file.
- template<typename Additional_Parameter >
  void storeResource (Identifier id, const std::string &path_to_file, const Additional_Parameter &parameter)

    Assigns a given identifier to a specific resource retrieved from the specified file.

1.33.1 Member Function Documentation

1.33.1.1 getResourceReference() [1/2]

```
template<typename Resource , typename Identifier >
Resource & ResourceManager< Resource, Identifier >::getResourceReference (
            Identifier id )
```

Checks an individual identifier and returns the resource assigned to it.

Parameters

id	Identifier identifying a previously saved resource

Returns

the resource stored for the given identifier.

Returns reference to Resource inside ResourceMap corresponding to given Identifier

1.33.1.2 getResourceReference() [2/2]

```
template<typename Resource , typename Identifier >
const Resource & ResourceManager< Resource, Identifier >::getResourceReference (
             Identifier id ) const
```

Checks an individual identifier and returns the resource assigned to it.

Parameters

id	Identifier identifying a previously saved resource

Returns

the resource stored for the given identifier.

Returns reference to Resource inside ResourceMap corresponding to given Identifier

1.33.1.3 storeResource() [1/2]

```
template<typename Resource , typename Identifier >
void ResourceManager< Resource, Identifier >::storeResource (
             Identifier id,
             const std::string & path_to_file )
```

Assigns a given identifier to a specific resource retrieved from the specified file.

Parameters

id	Identifier to which the resource is to be assigned
path_to_file	File path specifying the resource

Stores the given texture resource inside the ResourceMap

1.33.1.4 storeResource() [2/2]

```
template<typename Resource , typename Identifier >
template<typename Additional_Parameter >
```

```
void ResourceManager< Resource, Identifier >::storeResource (
            Identifier id,
            const std::string & path_to_file,
            const Additional_Parameter & parameter )
```

Assigns a given identifier to a specific resource retrieved from the specified file.

Template Parameters

Additional_Parameter	Type of additional parameter (for example sf::Shared::Type)

Parameters

id	Identifier to which the resource is to be assigned
path_to_file	File path specifying the resource
parameter	Additional argument (fragment shader file path, or sf::IntRect)

Stores the given texture resource inside the ResourceMap

Specialized version of this function to carry one of the methods that sf::Shader define – which is loadFromFile() containing additional Fragment Shader File Path or sf::Shader::Type. It also adds some other features which needs to be described when using loadFromFile().

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

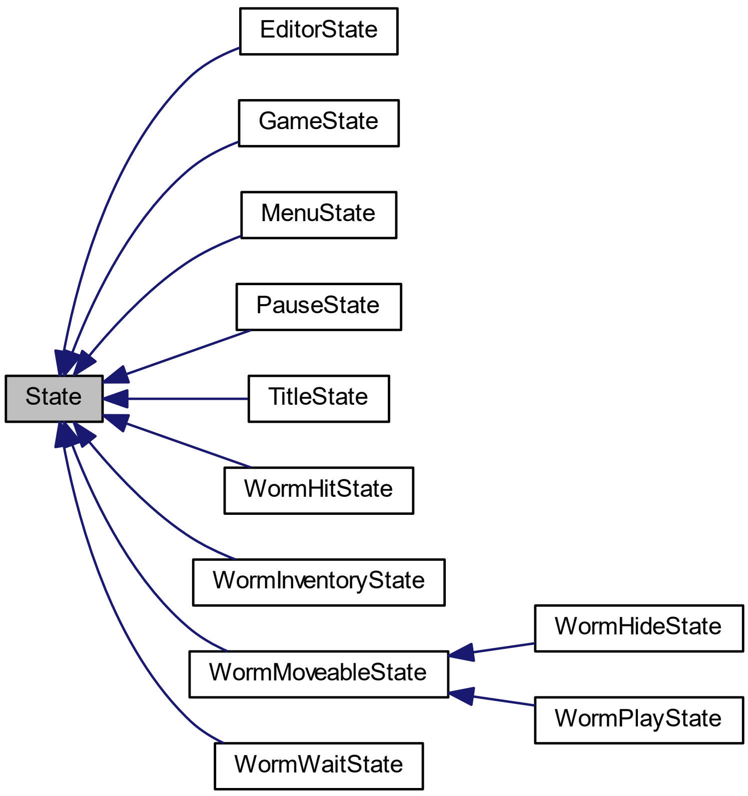
- Worms-Clone/Worms-Clone/Resources/ResourceManager.h

1.34 State Class Reference

The state that is on the stack performs the various functions of updating, drawing and handling user inputs.

```
#include <State.h>
```

Inheritance diagram for State:



Public Types

- using **Ptr** = std::unique_ptr< State >

Public Member Functions

- **State** (StateStack &stack)
- virtual void draw () const =0

  Draws only this state.
- virtual void draw (sf::RenderTarget &target, sf::RenderStates states) const =0

  Draws only this state to the passed target.
- virtual bool update (sf::Time deltaTime)=0

  Updates the logic of this state.
- virtual bool handleEvent (const sf::Event &event)=0

  It takes input (event) from the user and interprets it.

Protected Member Functions

- void requestPush (State_ID stateID)

  The state will be pushed out in the next iteration of the stack.
- void requestPop ()

  The state on the top of the stack will be removed in the next iteration of the stack.
- void requestClear ()

  All states on the stack will be removed in the next iteration of the stack.

1.34.1 Detailed Description

The state that is on the stack performs the various functions of updating, drawing and handling user inputs.

1.34.2 Member Function Documentation

1.34.2.1 draw()

```
virtual void State::draw (
             sf::RenderTarget & target,
             sf::RenderStates states ) const  [pure virtual]
```

Draws only this state to the passed target.

Parameters

target	where it should be drawn to
states	provides information about rendering process (transform, shader, blend mode)

Implemented in WormMoveableState, MenuState, WormPlayState, PauseState, WormHideState, WormHitState, EditorState, TitleState, WormInventoryState, GameState, and WormWaitState.

1.34.2.2 handleEvent()

```
virtual bool State::handleEvent (
             const sf::Event & event )  [pure virtual]
```

It takes input (event) from the user and interprets it.

Parameters

event	user input

Implemented in WormMoveableState, MenuState, WormPlayState, PauseState, WormHideState, WormHitState, EditorState, TitleState, WormInventoryState, GameState, and WormWaitState.

1.34.2.3 requestPush()

```
void State::requestPush (
            State_ID stateID )  [protected]
```

The state will be pushed out in the next iteration of the stack.

Parameters

stateID	Identifier of the state to be pushed

1.34.2.4 update()

```
virtual bool State::update (
            sf::Time deltaTime )  [pure virtual]
```

Updates the logic of this state.

Parameters

deltaTime	the time that has passed since the game was last updated

Implemented in WormMoveableState, MenuState, WormPlayState, PauseState, WormHideState, WormHitState, EditorState, TitleState, WormInventoryState, GameState, and WormWaitState.

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

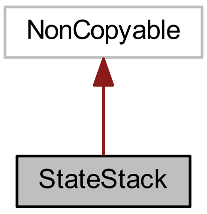
- Worms-Clone/Worms-Clone/States/State.h

1.35 StateStack Class Reference

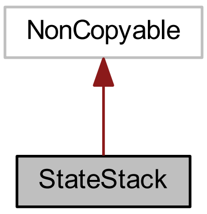
It allows you to create a certain flow of states inside application.

```
#include <StateStack.h>
```

Inheritance diagram for StateStack:



Collaboration diagram for StateStack:



Public Types

- enum Perform { **Push**, **Pop**, **Clear** }

  All actions that may be performed on the StateStack.

Public Member Functions

- template<typename State , typename... Args>
  void saveState (State_ID stateID, Args &&...)

  Creates callable function that generate the State object providing the given args to its constructor.
- void update (sf::Time deltaTime)

  Updates the state game logic on the stack.
- void draw () const

  Draws the states in the stack to the screen.
- void draw (sf::RenderTarget &target, sf::RenderStates state) const

  Draws the states in the stack to the given target with given states.
- void handleEvent (const sf::Event &event)

  It takes input (event) from the user and sends it to the states on the stack.
- void push (State_ID stateID)

Pushes a state with a given identifier onto the stack.
- void pop ()

  Removes the state at the top of the stack.
- void clear ()

  Clears the stack by removing all the states in it.
- bool empty () const

  Checks that there are no states on the stack.
- template<typename State , typename... Args>
  void **saveState** (State_ID stateID, Args &&... args)

1.35.1 Detailed Description

It allows you to create a certain flow of states inside application.

Statestack can control the flow of the game. Change between states not only in the context of the application itself, but also, for example, in the context of a single object.

| | | | | | | |pop() | |push()| |

	+-—>		+-—>	
Title state				Game state

+---------—+ +---------—+ +---------—+ Example of changing state from one to the other

StateStack allows states at the top to cover states below. Thus, for example, a "Game State" can be covered by a "Pause State", which will block it from updating game logic and displaying itself. Of course, the pop() operation will remove the "Pause State" and restore the covered "Game State" perfectly from when it was covered. States, however, have the option of marking individual functions as "transparent", i.e. not blocking the layer below. The transparent layer (state) is the state that returns "true".

1.35.2 Member Function Documentation

1.35.2.1 draw() [1/2]

```
void StateStack::draw ( ) const
```

Draws the states in the stack to the screen.

Draws the states at the top of the stack. If the state is transparent (returns true) then it also draw the state below it. The state below it is also checked for transparency and the process repeats itself.

1.35.2.2 draw() [2/2]

```
void StateStack::draw (
            sf::RenderTarget & target,
            sf::RenderStates state ) const
```

Draws the states in the stack to the given target with given states.

Parameters

target	where drawable object should be drawn to.
state	provides informations about rendering process (transform, shader, blend mode)

Draws the states at the top of the stack. If the state is transparent (returns true) then it also draw the state below it. The state below it is also checked for transparency and the process repeats itself.

1.35.2.3 empty()

```
bool StateStack::empty ( ) const
```

Checks that there are no states on the stack.

Returns

True if the stack is empty, false if there is a state on it.

1.35.2.4 handleEvent()

```
void StateStack::handleEvent (
            const sf::Event & event )
```

It takes input (event) from the user and sends it to the states on the stack.

Parameters

event	user input

It takes input (event) from the user and then passes it to the state on top of the stack. If the state is transparent (returns true) then it also passes the state below it. The state below it is also checked for transparency and the process repeats itself.

All of the states can process user input (event) on their own.

1.35.2.5 pop()

```
void StateStack::pop ( )
```

Removes the state at the top of the stack.

Executing this function on an empty stack leads to unpredictable behaviour.

1.35.2.6 push()

```
void StateStack::push (
            State_ID stateID )
```

Pushes a state with a given identifier onto the stack.

Parameters

stateID	The identifier to which the assigned state will be pushed onto the stack.

Previously, the StateStack must have saved the specified identifier via the saveState function. This function just uses the identifier to create an object of the given state class using a factory.

1.35.2.7 saveState()

```
template<typename State , typename...  Args>
void StateStack::saveState (
            State_ID stateID,
            Args && ...  )
```

Creates callable function that generate the State object providing the given args to its constructor.

Template Parameters

State	The state (class) to be created in a future call.
Args	Arguments to be passed to the constructor of the specified class.

Parameters

stateID	The identifier that will be used to create the class object.

A factory that allows to create any state that Identifier is passed to this function. This allows to not create all states at once and thanks to this avoid unnecesserily big containers holding all those states that might be never used.

By default it of course just pass an information to the factory how to create such an object nothing is yet created

1.35.2.8 update()

```
void StateStack::update (
            sf::Time deltaTime )
```

Updates the state game logic on the stack.

Parameters

deltaTime	the time that has passed since the game was last updated.

Updates the state game logic at the top of the stack. If the state is transparent (returns true) then it also updates the state logic below it. The state below it is also checked for transparency and the process repeats itself.

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

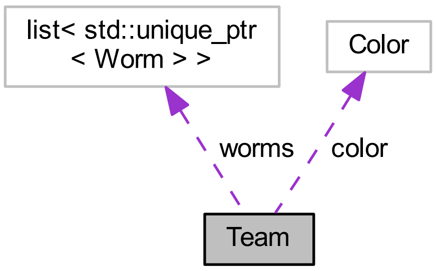
- Worms-Clone/Worms-Clone/States/StateStack.h

1.36 Team Struct Reference

A team made up of their assigned colour and a list of worms.

```
#include <WormQueue.h>
```

Collaboration diagram for Team:



Public Member Functions

- **Team** (sf::Color color, std::unique_ptr< Worm > &worm)

Public Attributes

- sf::Color **color**
- std::list< std::unique_ptr< Worm > > **worms**

Friends

- bool **operator==** (const Team &team1, const Team &team2)

1.36.1 Detailed Description

A team made up of their assigned colour and a list of worms.

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

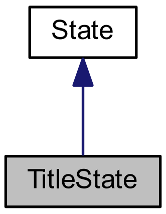
- Worms-Clone/Worms-Clone/Nodes/Physical/Specified/Worm/WormQueue.h

1.37 TitleState Class Reference

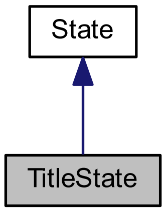
State in which information about game are displayed – before the main menu is displayed.

`#include <TitleState.h>`

Inheritance diagram for TitleState:



Collaboration diagram for TitleState:



Public Member Functions

- **TitleState** (StateStack &stack, sf::RenderWindow &window, const FontManager &fonts)
- void draw () const override

  Draws only this state.
- void draw (sf::RenderTarget &target, sf::RenderStates states) const override

  Draws only this state to the passed target.
- bool update (sf::Time deltaTime) override

  Updates the status/logic of the state.
- bool handleEvent (const sf::Event &event) override

  It takes input (event) from the user and interprets it.

Additional Inherited Members

1.37.1 Detailed Description

State in which information about game are displayed – before the main menu is displayed.

1.37.2 Member Function Documentation

1.37.2.1 draw()

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

Draws only this state to the passed target.

Parameters

target	where it should be drawn to
states	provides information about rendering process (transform, shader, blend mode)

Implements State.

1.37.2.2 handleEvent()

```
bool TitleState::handleEvent (
            const sf::Event & event )  [override], [virtual]
```

It takes input (event) from the user and interprets it.

Parameters

event	user input

Implements State.

1.37.2.3 update()

```
bool TitleState::update (
            sf::Time deltaTime )  [override], [virtual]
```

Updates the status/logic of the state.

Parameters

deltaTime	the time that has passed since the last frame.

Implements State.

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

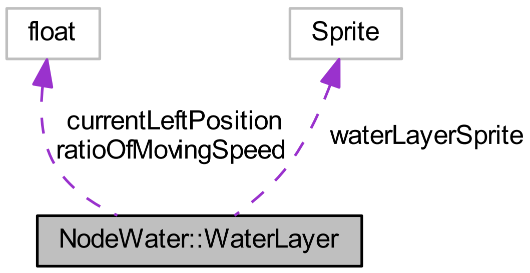
- Worms-Clone/Worms-Clone/States/Application_States/TitleState.h

1.38 NodeWater::WaterLayer Struct Reference

A structure showing the layer and its parameters, such as: sprite, moving speed or current texture offset.

```
#include <NodeWater.h>
```

Collaboration diagram for NodeWater::WaterLayer:



Public Member Functions

- **WaterLayer** (const sf::Texture &texture, float ratioSpeed)

Public Attributes

- sf::Sprite waterLayerSprite

  Sprite being a graphic image of a water wave.
- float ratioOfMovingSpeed

  A factor that determines how fast the waves are moving in this layer.
- float currentLeftPosition = 0.f

  Current texture offset defining the wave displacement.

1.38.1 Detailed Description

A structure showing the layer and its parameters, such as: sprite, moving speed or current texture offset.

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

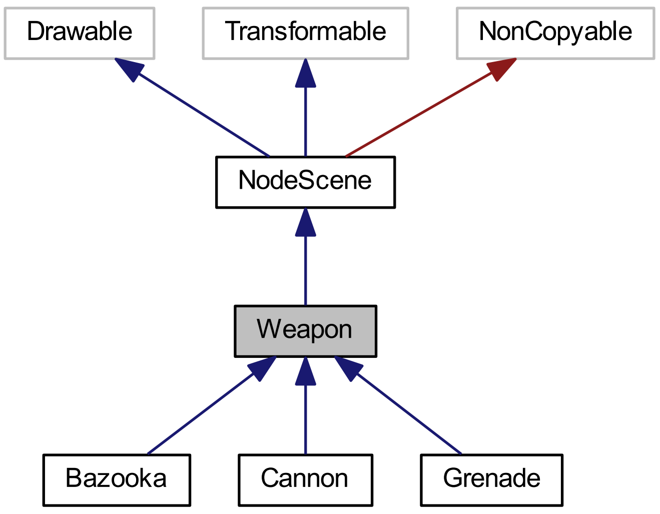
- Worms-Clone/Worms-Clone/Nodes/Physical/NodeWater.h

1.39 Weapon Class Reference

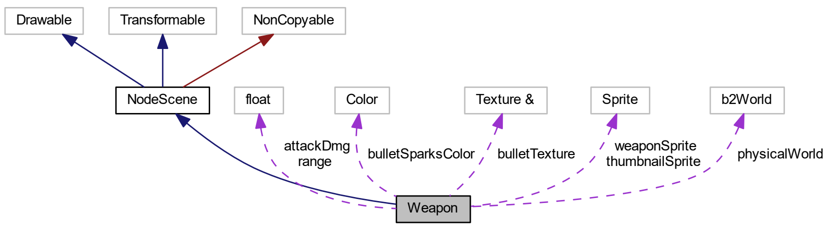
An abstract class of weapon that can be activated or fired with an earlier shooting bar charge.

```
#include <Weapon.h>
```

Inheritance diagram for Weapon:



Collaboration diagram for Weapon:



Public Member Functions

- **Weapon** (b2World &world, sf::Texture &weapon, sf::Texture &thumbnail, sf::Texture &bullet)
- virtual void shoot (NodeScene ∗rootNode, sf::Vector2f position, sf::Vector2f force)

  The firing function of a weapon that will be executed when the shooting bar is loaded.
- virtual void activation (Worm &worm)

  Activates the weapon.
- void setMaxDmg (float dmg)

  Sets the maximum damage this weapon is capable of dealing.
- void setRange (float rng)

  Sets the maximum distance from the explosion point from which it deals damage.
- void setSparkColor (const sf::Color &color)

  Sets the colour of the particles present after the explosion.
- virtual bool isActivation ()=0

  Check if weapon should have shooting bar, or it should just activate with a button.
- virtual bool isRoundEnding ()=0

  Check if usage of the weapon should end the round.
- sf::Sprite & getThumbnailSprite ()

  Returns a sprite which is a graphical representation of the weapon thumbnail in the inventory.
- void rotateWeapon (float angle)

  Adjusts the weapon to a given angle.
- void drawThis (sf::RenderTarget &target, sf::RenderStates states) const override

  Draws only this Weapon to the passed target.
- void updateThis (sf::Time deltaTime) override

  Updates the logic of the weapon.

Protected Attributes

- sf::Sprite weaponSprite

  Sprite which is a visual representation of the weapon in the game.
- sf::Sprite thumbnailSprite

  Sprite which is a visual representation of the weapon in the inventory.
- sf::Texture & bulletTexture

  Sprite which is a visual representation of the bullet in the game.
- sf::Color bulletSparksColor = sf::Color::White

  Colour of the explosion after the projectile explosion.
- float attackDmg = 0.f

  Maximum damage this weapon can deal.
- float range = 0.f

  Weapon striking range.
- b2World & physicalWorld

  Physical simulation of the game world.

Additional Inherited Members

1.39.1 Detailed Description

An abstract class of weapon that can be activated or fired with an earlier shooting bar charge.

1.39.2 Member Function Documentation

1.39.2.1 activation()

```
virtual void Weapon::activation (
            Worm & worm )  [virtual]
```

Activates the weapon.

Used when a weapon does not shoot with usage of shooting bar but can be activated instantly.

Parameters

worm	A worm that uses a particular weapon.

1.39.2.2 drawThis()

```
void Weapon::drawThis (
            sf::RenderTarget & target,
            sf::RenderStates states ) const  [override], [virtual]
```

Draws only this Weapon to the passed target.

Parameters

target	where it should be drawn to
states	provides information about rendering process (transform, shader, blend mode)

Reimplemented from NodeScene.

1.39.2.3 getThumbnailSprite()

```
sf::Sprite& Weapon::getThumbnailSprite ( )
```

Returns a sprite which is a graphical representation of the weapon thumbnail in the inventory.

Returns

A thumbnail that is a graphical representation of a weapon in an inventory.

1.39.2.4 isActivation()

```
virtual bool Weapon::isActivation ( )  [pure virtual]
```

Check if weapon should have shooting bar, or it should just activate with a button.

Returns

True if it is activated with a button, False if it should have been loaded with shooting bar

Implemented in Grenade, Bazooka, and Cannon.

1.39.2.5 isRoundEnding()

```
virtual bool Weapon::isRoundEnding ( )  [pure virtual]
```

Check if usage of the weapon should end the round.

Returns

True usage ends the round, false otherwise

Implemented in Grenade, Bazooka, and Cannon.

1.39.2.6 rotateWeapon()

```
void Weapon::rotateWeapon (
             float angle )
```

Adjusts the weapon to a given angle.

Parameters

angle	Angle at which the weapon should be positioned

1.39.2.7 setMaxDmg()

```
void Weapon::setMaxDmg (
             float dmg )
```

Sets the maximum damage this weapon is capable of dealing.

Parameters

dmg	Maximum possible damage

1.39.2.8 setRange()

```
void Weapon::setRange (
             float rng )
```

Sets the maximum distance from the explosion point from which it deals damage.

Parameters

rng	Maximum possible range at which it still deals damage

1.39.2.9 setSparkColor()

```
void Weapon::setSparkColor (
             const sf::Color & color )
```

Sets the colour of the particles present after the explosion.

Parameters

color	Colour of the particles

1.39.2.10 shoot()

```
virtual void Weapon::shoot (
             NodeScene * rootNode,
             sf::Vector2f position,
             sf::Vector2f force )  [virtual]
```

The firing function of a weapon that will be executed when the shooting bar is loaded.

Parameters

rootNode	A node that will serve as the parent of a bullet created and added to the game world.
position	The position at which the bullet will form.
force	The force with which the bullet will be fired

Reimplemented in Grenade.

1.39.2.11 updateThis()

```
void Weapon::updateThis (
            sf::Time deltaTime )  [override], [virtual]
```

Updates the logic of the weapon.

Parameters

deltaTime	the time that has passed since the game was last updated

Reimplemented from NodeScene.

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

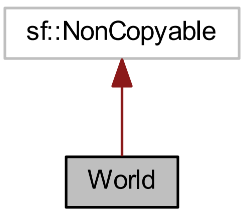
- Worms-Clone/Worms-Clone/Nodes/Physical/Specified/Worm/Weapons/Weapon.h

1.40 World Class Reference

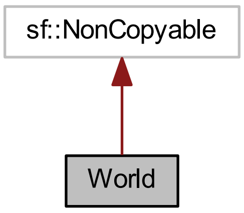
The game world in which all processes inside the game world are controlled.

```
#include <World.h>
```

Inheritance diagram for World:



Collaboration diagram for World:



Public Member Functions

- **World** (sf::RenderWindow &window, int wormAmount, int numberOfTeams)
- void update (sf::Time deltaTime)

  Updates the world logic every iteration.
- void draw () const

  Draws the world every frame.
- void box2DdrawDebug ()

  Draws all collisions (hitboxes) of physical objects to the screen.
- void processEvents (const sf::Event &event)

  It takes input (event) from the user and interprets it.
- bool **isGameFinished** () const

1.40.1 Detailed Description

The game world in which all processes inside the game world are controlled.

These are things like keeping an eye on the order of movements. The turn-based game system. Displaying time on the screen, inserting objects into the world.

1.40.2 Member Function Documentation

1.40.2.1 processEvents()

```
void World::processEvents (
             const sf::Event & event )
```

It takes input (event) from the user and interprets it.

Parameters

event	user input

1.40.2.2 update()

```
void World::update (
             sf::Time deltaTime )
```

Updates the world logic every iteration.

Parameters

deltaTime	the time that has passed since the game was last updated

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

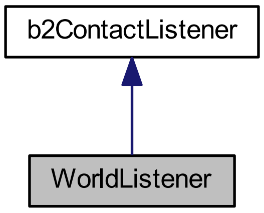
- Worms-Clone/Worms-Clone/World.h

1.41 WorldListener Class Reference

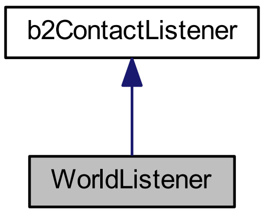
Checks for and handles collisions within the game.

```
#include <WorldListener.h>
```

Inheritance diagram for WorldListener:



Collaboration diagram for WorldListener:



1.41.1 Detailed Description

Checks for and handles collisions within the game.

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

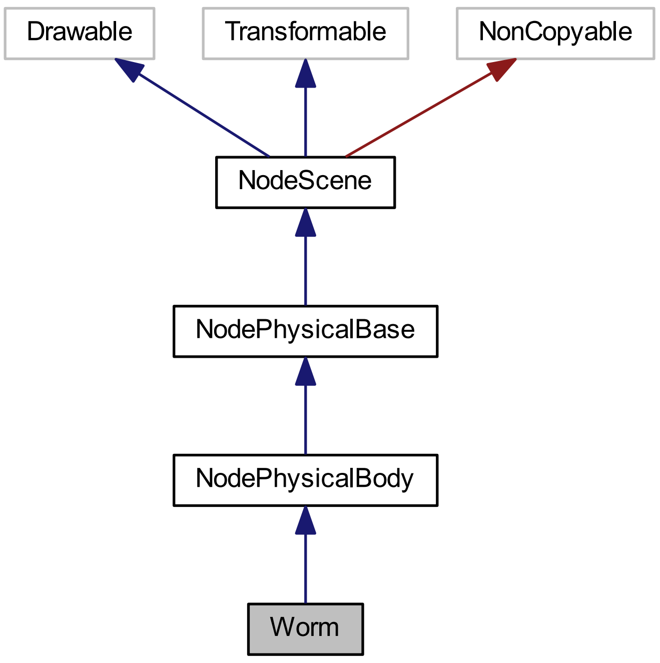
- Worms-Clone/Worms-Clone/WorldListener.h

1.42 Worm Class Reference

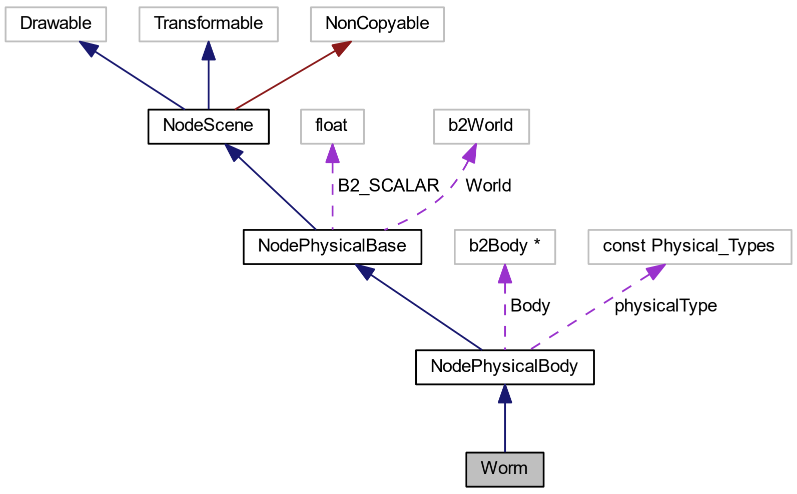
Worm object that can be controlled by the player.

`#include <Worm.h>`

Inheritance diagram for Worm:



Collaboration diagram for Worm:



Public Member Functions

- Worm (b2World &world, TextureManager &textures, const FontManager &fonts, sf::RenderWindow &window, sf::Vector2f position)

Creates a worm.
- void drawThis (sf::RenderTarget &target, sf::RenderStates states) const override

  Draws the worm to the given target.
- void updateThis (sf::Time deltaTime) override

  Updates the logic of the worm.
- void handleThisEvents (const sf::Event &event) override

  It takes input (event) from the user and interprets it.
- void activateState (State_ID state)

  Sets the state the worm is in.
- State_ID getCurrentState () const

  A function that checks what state the worm is currently in.
- bool facingRight ()

  Checks if the worm's face is facing the right side of the screen.
- void setDamage (int dmg)

  Takes away a certain amount of the worm's life points.
- auto setName (const std::string &name) -> void

  Sets a new name for the worm.
- std::string getName ()

  Returns the name of the worm.
- sf::Color getTeam () const

  A function that returns the color of the team the worm is in.
- void setTeam (sf::Color teamColor)

  Sets a new team for the worm.
- bool isDestroyed () override

  Checks whether the worm is to be deleted.
- sf::Vector2f getWormSize () const

  Calculates the size of the worm sprite.

Friends

- class **WorldListener**
- class WormHideState

  Player have ... seconds to hide.
- class WormPlayState

  Player can play his turn.
- class WormWaitState

  Player can't move with this worm.
- class WormHitState

  State after getting hit by other player.
- class **WormMoveableState**
- class WormInventoryState

  State in which player can choose a weapon.

Additional Inherited Members

1.42.1 Detailed Description

Worm object that can be controlled by the player.

1.42.2 Constructor & Destructor Documentation

1.42.2.1 Worm()

```
Worm::Worm (
            b2World & world,
            TextureManager & textures,
            const FontManager & fonts,
            sf::RenderWindow & window,
            sf::Vector2f position )
```

Creates a worm.

Parameters

world	The physical world in which the physical simulation of the worm is located
textures	Texture Holder where the worm texture is located
fonts	Font Holder where the font used to display worm name is located
window	Window into which the worm's inventory should be drawn
position	Position in which the worm should appear
wormQueue	Queue of movements in which the worm is placed

1.42.3 Member Function Documentation

1.42.3.1 activateState()

```
void Worm::activateState (
            State_ID state )
```

Sets the state the worm is in.

Parameters

state	Identifier of the state the worm should be in

1.42.3.2 drawThis()

```
void Worm::drawThis (
            sf::RenderTarget & target,
            sf::RenderStates states ) const  [override], [virtual]
```

Draws the worm to the given target.

Parameters

target	where it should be drawn to
states	provides information about rendering process (transform, shader, blend mode)

Reimplemented from NodeScene.

1.42.3.3 facingRight()

```
bool Worm::facingRight ( )
```

Checks if the worm's face is facing the right side of the screen.

Returns

True if the worm's 'sprite' is facing right, false otherwise

1.42.3.4 getCurrentState()

```
State_ID Worm::getCurrentState ( ) const
```

A function that checks what state the worm is currently in.

Returns

Identifier of the state the worm is in

1.42.3.5 getName()

```
std::string Worm::getName ( )
```

Returns the name of the worm.

Returns

String with name of the worm.

1.42.3.6 getTeam()

```
sf::Color Worm::getTeam ( ) const
```

A function that returns the color of the team the worm is in.

Returns

Color of the team of the worm

1.42.3.7 getWormSize()

```
sf::Vector2f Worm::getWormSize ( ) const
```

Calculates the size of the worm sprite.

Returns

Size of the worm sprite

1.42.3.8 handleThisEvents()

```
void Worm::handleThisEvents (
             const sf::Event & event )  [override], [virtual]
```

It takes input (event) from the user and interprets it.

Parameters

event	user input

Reimplemented from NodeScene.

1.42.3.9 isDestroyed()

```
bool Worm::isDestroyed ( )  [override], [virtual]
```

Checks whether the worm is to be deleted.

Returns

True if the object should be deleted, false if not

Reimplemented from NodeScene.

1.42.3.10 setDamage()

```
void Worm::setDamage (
            int dmg )
```

Takes away a certain amount of the worm's life points.

Parameters

dmg	Number of health points to take away from the worm

1.42.3.11 setName()

```
auto Worm::setName (
            const std::string & name ) -> void
```

Sets a new name for the worm.

Parameters

name	New name of the worm

1.42.3.12 setTeam()

```
void Worm::setTeam (
            sf::Color teamColor )
```

Sets a new team for the worm.

Parameters

teamColor	Color of the new team of the worm

1.42.3.13 updateThis()

```
void Worm::updateThis (
            sf::Time deltaTime )  [override], [virtual]
```

Updates the logic of the worm.

Parameters

deltaTime	the time that has passed since the game was last updated

Reimplemented from NodeScene.

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

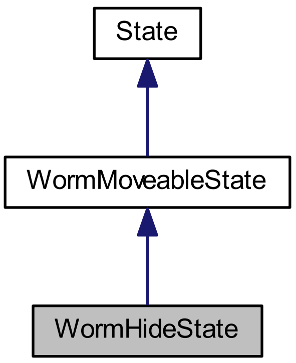
- Worms-Clone/Worms-Clone/Nodes/Physical/Specified/Worm/Worm.h

1.43 WormHideState Class Reference

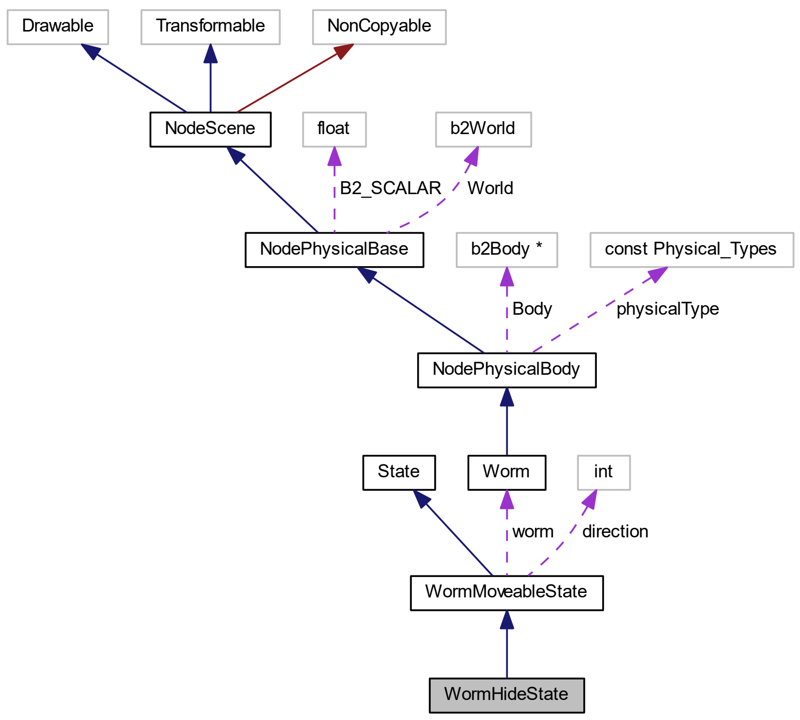
A state in which the worm has a chance to escape and hide.

```
#include <WormHideState.h>
```

Inheritance diagram for WormHideState:



Collaboration diagram for WormHideState:



Public Member Functions

- **WormHideState** (StateStack &, Worm &)
- void draw () const override

  Draws only this state.
- void draw (sf::RenderTarget &target, sf::RenderStates states) const override

  Draws only this state (current state of the worm) to the passed target.
- bool update (sf::Time deltaTime) override

  Updates the status of the worm.
- bool handleEvent (const sf::Event &event) override

  It takes input (event) from the user and interprets it.

Additional Inherited Members

1.43.1 Detailed Description

A state in which the worm has a chance to escape and hide.

It is a rather short time. When this time elapses, the player changes state to "WaitState".

// ==== Attention / Warning!! ==== // For the moment, this time change is done directly through the "Gameplay↩ Manager". It is possible that this will be handled here in the future.

1.43.2 Member Function Documentation

1.43.2.1 draw()

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

Draws only this state (current state of the worm) to the passed target.

Parameters

target	where it should be drawn to
states	provides information about rendering process (transform, shader, blend mode)

Implements WormMoveableState.

1.43.2.2 handleEvent()

```
bool WormHideState::handleEvent (
             const sf::Event & event )  [override], [virtual]
```

It takes input (event) from the user and interprets it.

Parameters

event	user input

Implements WormMoveableState.

1.43.2.3 update()

```
bool WormHideState::update (
             sf::Time deltaTime )  [override], [virtual]
```

Updates the status of the worm.

Parameters

deltaTime	the time that has passed since the last frame.

Implements WormMoveableState.

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

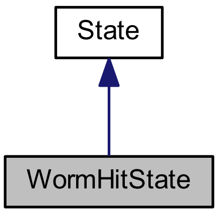
- Worms-Clone/Worms-Clone/Nodes/Physical/Specified/Worm/WormHideState.h

1.44 WormHitState Class Reference

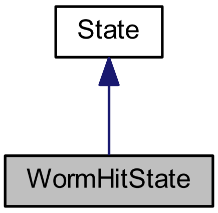
A state in which a worm has been hit by something.

`#include <WormHitState.h>`

Inheritance diagram for WormHitState:



Collaboration diagram for WormHitState:



Public Member Functions

- **WormHitState** (StateStack &, Worm &, TextureManager &textures)
- void draw () const override

  Draws only this state.
- void draw (sf::RenderTarget &target, sf::RenderStates states) const override

  Draws only this state (current state of the worm) to the passed target.
- bool update (sf::Time deltaTime) override

  Updates the status of the worm.
- bool handleEvent (const sf::Event &event) override

  It takes input (event) from the user and interprets it.

Additional Inherited Members

1.44.1 Detailed Description

A state in which a worm has been hit by something.

This state is responsible for changing the texture that shows the player that this worm has been hit. The character is then also unlocked and can physically be moved. It will only be locked again on contact with the ground and at a sufficiently low speed. A few seconds later, the state will also change back to "WaitState", among other things.

1.44.2 Member Function Documentation

1.44.2.1 draw()

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

Draws only this state (current state of the worm) to the passed target.

Parameters

target	where it should be drawn to
states	provides information about rendering process (transform, shader, blend mode)

Implements State.

1.44.2.2 handleEvent()

```
bool WormHitState::handleEvent (
             const sf::Event & event )  [override], [virtual]
```

It takes input (event) from the user and interprets it.

Parameters

event	user input

Implements State.

1.44.2.3 update()

```
bool WormHitState::update (
            sf::Time deltaTime )  [override], [virtual]
```

Updates the status of the worm.

Parameters

deltaTime	the time that has passed since the last frame.

Implements State.

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

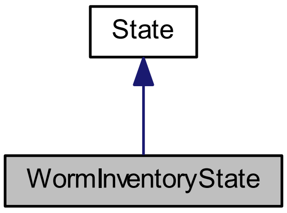
- Worms-Clone/Worms-Clone/Nodes/Physical/Specified/Worm/WormHitState.h

1.45 WormInventoryState Class Reference

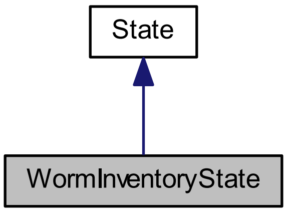
A condition that displays the player's inventory and allows them to change to another weapon.

`#include <WormInventoryState.h>`

Inheritance diagram for WormInventoryState:



Collaboration diagram for WormInventoryState:



Public Member Functions

- **WormInventoryState** (StateStack &, Worm &, TextureManager &textures, const FontManager &fonts, sf::↩ RenderWindow &window)
- void draw () const override

  Draws only this state.
- void draw (sf::RenderTarget &target, sf::RenderStates states) const override

  Draws only this state (current state of the worm) to the passed target.
- bool update (sf::Time deltaTime) override

  Updates the status of the worm.
- bool handleEvent (const sf::Event &event) override

  It takes input (event) from the user and interprets it.

Additional Inherited Members

1.45.1 Detailed Description

A condition that displays the player's inventory and allows them to change to another weapon.

It also displays the number of weapons the player has.

1.45.2 Member Function Documentation

1.45.2.1 draw()

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

Draws only this state (current state of the worm) to the passed target.

Parameters

target	where it should be drawn to
states	provides information about rendering process (transform, shader, blend mode)

Implements State.

1.45.2.2 handleEvent()

```
bool WormInventoryState::handleEvent (
             const sf::Event & event )  [override], [virtual]
```

It takes input (event) from the user and interprets it.

Parameters

event	user input

Implements State.

1.45.2.3 update()

```
bool WormInventoryState::update (
            sf::Time deltaTime )  [override], [virtual]
```

Updates the status of the worm.

Parameters

deltaTime	the time that has passed since the last frame.

Implements State.

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

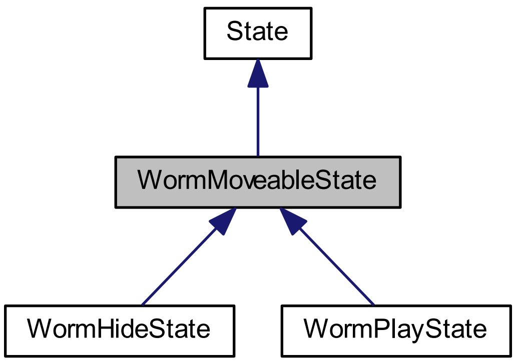
- Worms-Clone/Worms-Clone/Nodes/Physical/Specified/Worm/WormInventoryState.h

1.46 WormMoveableState Class Reference

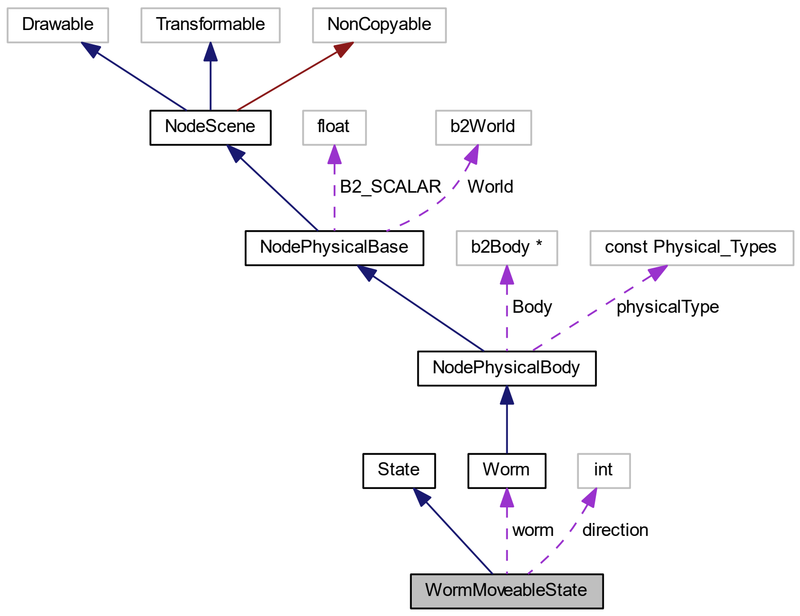
An abstract class that contains functions related to the movement of the worm.

`#include <WormMoveableState.h>`

Inheritance diagram for WormMoveableState:



Collaboration diagram for WormMoveableState:



Public Member Functions

- void draw () const override=0

    Draws only this state.
- void draw (sf::RenderTarget &target, sf::RenderStates states) const override=0

    Draws only this state (current state of the worm) to the passed target.
- bool update (sf::Time deltaTime) override=0

    Updates the status of the worm.
- bool handleEvent (const sf::Event &event) override=0

    It takes input (event) from the user and interprets it.
- bool facingRight ()

    Checks whether the worm's face turns to the right.

Protected Member Functions

- **WormMoveableState** (StateStack &, Worm &)
- void handleMovement (const sf::Event &event)

    Interprets player input in terms of worm movement.
- void updateMovement (sf::Time deltatime)

    Updates the state of the worm in terms of worm movement.

Protected Attributes

- int direction

    Determines the direction in which the worm is looking.
- Worm & **worm**

Additional Inherited Members

1.46.1 Detailed Description

An abstract class that contains functions related to the movement of the worm.

Then they inherit from it all the states that need to move inside the game – for example, HideState or PlayState

1.46.2 Member Function Documentation

1.46.2.1 draw()

```
void WormMoveableState::draw (
            sf::RenderTarget & target,
            sf::RenderStates states ) const  [override], [pure virtual]
```

Draws only this state (current state of the worm) to the passed target.

Parameters

target	where it should be drawn to
states	provides information about rendering process (transform, shader, blend mode)

Implements State.

Implemented in WormPlayState, and WormHideState.

1.46.2.2 facingRight()

```
bool WormMoveableState::facingRight ( )
```

Checks whether the worm's face turns to the right.

Returns

True if the worm looks to the right of the screen, false otherwise

1.46.2.3 handleEvent()

```
bool WormMoveableState::handleEvent (
            const sf::Event & event )  [override], [pure virtual]
```

It takes input (event) from the user and interprets it.

Parameters

event	user input

Implements State.

Implemented in WormPlayState, and WormHideState.

1.46.2.4 handleMovement()

```
void WormMoveableState::handleMovement (
             const sf::Event & event )  [protected]
```

Interprets player input in terms of worm movement.

Parameters

event	input of the player

A function that is responsible for taking player input and interpreting it to create the correct movement of the worm around the map.

1.46.2.5 update()

```
bool WormMoveableState::update (
             sf::Time deltaTime )  [override], [pure virtual]
```

Updates the status of the worm.

Parameters

deltaTime	the time that has passed since the last frame.

Implements State.

Implemented in WormPlayState, and WormHideState.

1.46.2.6 updateMovement()

```
void WormMoveableState::updateMovement (
             sf::Time deltatime )  [protected]
```

Updates the state of the worm in terms of worm movement.

Parameters

deltatime	Time elapsed since the previous frame.

1.46.3 Member Data Documentation

1.46.3.1 direction

```
int WormMoveableState::direction  [protected]
```

Determines the direction in which the worm is looking.

1 if looks right, -1 if looks left Thanks to this I can easily set some vectors so they're pointing the proper direction

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

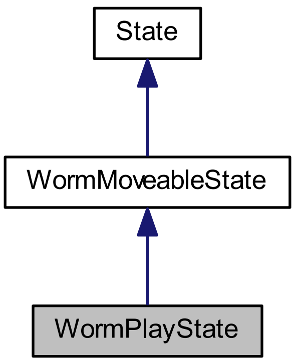
- Worms-Clone/Worms-Clone/Nodes/Physical/Specified/Worm/WormMoveableState.h

1.47 WormPlayState Class Reference

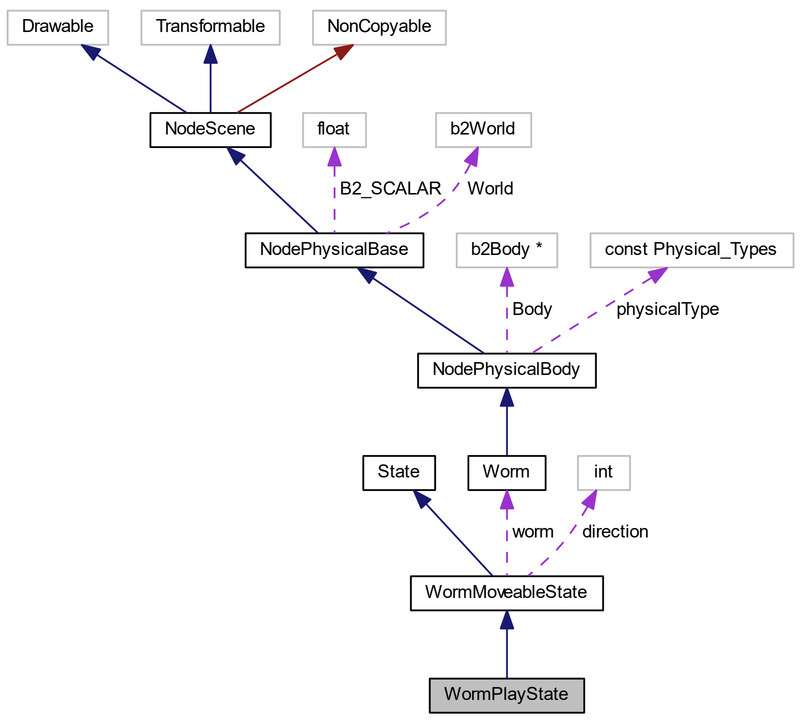
A worm state in which the worm is controlled by the player.

```
#include <WormPlayState.h>
```

Inheritance diagram for WormPlayState:



Collaboration diagram for WormPlayState:



Public Member Functions

- **WormPlayState** (StateStack &, Worm &)
- void draw () const override

  Draws only this state.
- void draw (sf::RenderTarget &target, sf::RenderStates states) const override

  Draws only this state (current state of the worm) to the passed target.
- bool update (sf::Time deltaTime) override

  Updates the status of the worm.
- bool handleEvent (const sf::Event &event) override

  It takes input (event) from the user and interprets it.
- void shoot ()

  A function that shoots a bullet.

Additional Inherited Members

1.47.1 Detailed Description

A worm state in which the worm is controlled by the player.

This state allows the worm to move around the game world. Change states (for example, open equipment) or shoot.

1.47.2 Member Function Documentation

1.47.2.1 draw()

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

Draws only this state (current state of the worm) to the passed target.

Parameters

target	where it should be drawn to
states	provides information about rendering process (transform, shader, blend mode)

Implements WormMoveableState.

1.47.2.2 handleEvent()

```
bool WormPlayState::handleEvent (
            const sf::Event & event )  [override], [virtual]
```

It takes input (event) from the user and interprets it.

Parameters

event	user input

Implements WormMoveableState.

1.47.2.3 shoot()

```
void WormPlayState::shoot ( )
```

A function that shoots a bullet.

Depending on the weapon setting, it can change the worm's state to "HideState".

1.47.2.4 update()

```
bool WormPlayState::update (
            sf::Time deltaTime )  [override], [virtual]
```

Updates the status of the worm.

Parameters

deltaTime	the time that has passed since the last frame.

Implements WormMoveableState.

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

- Worms-Clone/Worms-Clone/Nodes/Physical/Specified/Worm/WormPlayState.h

1.48 WormQueue Class Reference

A queue of the teams consisting of the worms.

```
#include <WormQueue.h>
```

Public Member Functions

- void addWorm (std::unique_ptr< Worm > &worm)

  Add worm to the queue.
- Worm & getNextWorm ()

  Returns the next worm that should play its turn.
- bool isEmpty () const

  Checks that there is no worm in the queue.
- void update (sf::Time deltaTime)

  Updates the status of worms in the queue.
- void draw (sf::RenderTarget &target, sf::RenderStates states) const

  Draws all the worms to the given target.
- void handleEvents (const sf::Event &event)

  It takes input (event) from the user and passes to the worms.
- Worm & front ()

  Returns the currently played worm.
- void removeDestroyed ()

  Removes all worms marked for removal.
- int aliveTeams ()

  Calculates the number of teams currently alive (still playing).

1.48.1 Detailed Description

A queue of the teams consisting of the worms.

```
+--------+currentTeam
|
v
```

+----------------------—+ +----------------------—+

Team		Team
sf::Color::Red		sf::Color::Blue

+-----------------------—+ +-----------------------—+ |Worm, Worm1, Worm2, Worm3| |Worm, Worm1, Worm2, Worm3| +-----------------------—+ +-----------------------—+ ^ |

- front()

1.48.2 Member Function Documentation

1.48.2.1 addWorm()

```
void WormQueue::addWorm (
             std::unique_ptr< Worm > & worm )
```

Add worm to the queue.

Parameters

worm	Worm to add

1.48.2.2 aliveTeams()

```
int WormQueue::aliveTeams ( )
```

Calculates the number of teams currently alive (still playing).

Returns

Number of teams currently alive

1.48.2.3 draw()

```
void WormQueue::draw (
             sf::RenderTarget & target,
             sf::RenderStates states ) const
```

Draws all the worms to the given target.

Parameters

target	where it should be drawn to
states	provides information about rendering process (transform, shader, blend mode)

1.48.2.4 front()

```
Worm& WormQueue::front ( )
```

Returns the currently played worm.

Returns

Reference to the worm currently being played

1.48.2.5 getNextWorm()

```
Worm& WormQueue::getNextWorm ( )
```

Returns the next worm that should play its turn.

Returns

A reference to a worm that should play

```
               An example of getNextWorm()

             +---------+currentTeam
             |
             v
```

+-----------------------—+ +-----------------------—+

Team		Team
sf::Color::Red		sf::Color::Blue

+-----------------------—+ +-----------------------—+ |Worm, Worm1, Worm2, Worm3| |Worm, Worm1, Worm2, Worm3| +-----------------------—+ +-----------------------—+

```
                      currentTeam+---------+
                                           |
                                           v
```

+-----------------------—+ +-----------------------—+

Team		Team
sf::Color::Red		sf::Color::Blue

+------------------------—+ +-------------------------—+ |<–Worm1, Worm2, Worm3 | |Worm, Worm1, Worm2, Worm3| +-+---- ----------------—^-+ +–+----------------------—+ | | | +-----—>Worm--------—+ v Return Move worm to the end this

1.48.2.6 handleEvents()

```
void WormQueue::handleEvents (
             const sf::Event & event )
```

It takes input (event) from the user and passes to the worms.

Parameters

event	user input

1.48.2.7 isEmpty()

```
bool WormQueue::isEmpty ( ) const
```

Checks that there is no worm in the queue.

Returns

True if there is no worm in the queue, false otherwise

1.48.2.8 removeDestroyed()

```
void WormQueue::removeDestroyed ( )
```

Removes all worms marked for removal.

If their teams are empty, deletes them too

1.48.2.9 update()

```
void WormQueue::update (
             sf::Time deltaTime )
```

Updates the status of worms in the queue.

Parameters

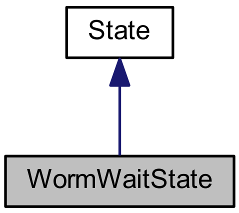
deltaTime	Time elapsed since last frame

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

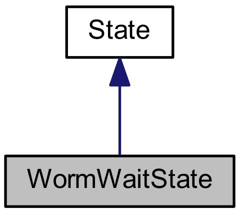
- Worms-Clone/Worms-Clone/Nodes/Physical/Specified/Worm/WormQueue.h

1.49 WormWaitState Class Reference

Inheritance diagram for WormWaitState:



Collaboration diagram for WormWaitState:



Public Member Functions

- **WormWaitState** (StateStack &, Worm &)
- void draw () const override

  Draws only this state.
- void draw (sf::RenderTarget &target, sf::RenderStates states) const override

  Draws only this state (current state of the worm) to the passed target.
- bool update (sf::Time deltaTime) override

  Updates the status of the worm.
- bool handleEvent (const sf::Event &event) override

  It takes input (event) from the user and interprets it.

Additional Inherited Members

1.49.1 Member Function Documentation

1.49.1.1 draw()

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

Draws only this state (current state of the worm) to the passed target.

Parameters

target	where it should be drawn to
states	provides information about rendering process (transform, shader, blend mode)

Implements State.

1.49.1.2 handleEvent()

```
bool WormWaitState::handleEvent (
            const sf::Event & event )  [override], [virtual]
```

It takes input (event) from the user and interprets it.

Parameters

event	user input

Implements State.

1.49.1.3 update()

```
bool WormWaitState::update (
            sf::Time deltaTime )  [override], [virtual]
```

Updates the status of the worm.

Parameters

deltaTime	the time that has passed since the last frame.

Implements State.

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

- Worms-Clone/Worms-Clone/Nodes/Physical/Specified/Worm/WormWaitState.h

Index

activateState
    Worm, 111
activation
    Weapon, 102
addHole
    NodeDestructibleRectangle, 49
addTime
    GameplayManager, 35
addWorm
    GameplayManager, 35
    WormQueue, 130
aliveTeams
    WormQueue, 130
angleToRadians
    NodePhysicalBase, 57
anyBullet
    GameplayManager, 36
applyForce
    NodePhysicalBody, 61

b2ContactListener, 1
b2Draw, 1
b2VecToSfVector
    NodePhysicalBase, 57
back
    GUI::Container, 21
Bazooka, 2
    isActivation, 3
    isRoundEnding, 3
Bullet, 4
    isDestroyed, 6
    setSparkColor, 6
    updateThis, 6

Cannon, 13
    isActivation, 15
    isRoundEnding, 15
Collision, 15
createBackgroundWorld
    MenuState, 46
createGrenades
    MenuState, 46

DebugBox2D, 23
Delayed_Bullet, 24
    drawThis, 26
    updateThis, 27
direction
    WormMoveableState, 126
draw
    EditorState, 30
    GameState, 40
    GUI::Button, 9
    GUI::Container, 21
    GUI::FixedContainer, 32
    MenuState, 47
    NodeEditorObject, 52
    NodeScene, 72
    PauseState, 85
    State, 90
    StateStack, 93
    TitleState, 98
    WormHideState, 117
    WormHitState, 119
    WormInventoryState, 121
    WormMoveableState, 124
    WormPlayState, 128
    WormQueue, 130
    WormWaitState, 134
drawThis
    Delayed_Bullet, 26
    GameplayManager, 36
    NodePhysicalSpark, 63
    NodePhysicalSprite, 67
    NodeRectangularPhysical, 69
    NodeScene, 72
    Weapon, 102
    Worm, 111
Dynamic_Type
    NodePhysicalBody, 61

Editor, 27
    handleEvent, 28
    update, 28
EditorState, 28
    draw, 30
    handleEvent, 30
    update, 30
empty
    StateStack, 94

facingRight
    Worm, 112
    WormMoveableState, 124
front
    GUI::Container, 22
    WormQueue, 131

Game, 33
    run, 33

GameplayManager, 34
  addTime, 35
  addWorm, 35
  anyBullet, 36
  drawThis, 36
  getRootNode, 36, 37
  handleThisEvents, 37
  isGameFinished, 37
  setWorldMessage, 38
  updateThis, 38
GameState, 38
  draw, 40
  handleEvent, 40
  update, 40
getAbsolutePosition
  NodeScene, 72
getCurrentState
  Worm, 112
getGlobalBounds
  GUI::Button, 9
  GUI::Component, 17
  NodeEditorObject, 52
getId
  NodeEditorObject, 52
getLocalBounds
  GUI::Button, 9
  GUI::Component, 18
getName
  Worm, 112
getNextWorm
  WormQueue, 131
getResourceReference
  ResourceManager< Resource, Identifier >, 86, 87
getRootNode
  GameplayManager, 36, 37
  NodeScene, 72, 73
getSize
  NodeEditorObject, 52
getSpriteSize
  NodeSprite, 78
getTeam
  Worm, 112
getThumbnailSprite
  Weapon, 102
getWormSize
  Worm, 113
Grenade, 41
  isActivation, 42
  isRoundEnding, 42
  shoot, 42
GUI::Button, 7
  draw, 9
  getGlobalBounds, 9
  getLocalBounds, 9
  handleEvents, 9
  matchSizeToText, 11
  setActiveFunction, 11
  setDeactiveFunction, 11
  setPositionBelow, 11
  setSize, 12
  setText, 12
  update, 13
GUI::Component, 16
  getGlobalBounds, 17
  getLocalBounds, 18
  handleEvents, 18
  isActive, 18
  isSelected, 19
  setPositionBelow, 19
  update, 19
GUI::Container, 20
  back, 21
  draw, 21
  front, 22
  handleEvents, 22
  isEmpty, 22
  store, 22
GUI::FixedContainer, 31
  draw, 32
  store, 32

handleEvent
  Editor, 28
  EditorState, 30
  GameState, 40
  MenuState, 47
  NodeEditorObject, 53
  PauseState, 85
  State, 90
  StateStack, 94
  TitleState, 98
  WormHideState, 117
  WormHitState, 119
  WormInventoryState, 121
  WormMoveableState, 124
  WormPlayState, 128
  WormWaitState, 134
handleEvents
  GUI::Button, 9
  GUI::Component, 18
  GUI::Container, 22
  NodeScene, 73
  WormQueue, 132
handleMovement
  WormMoveableState, 125
handleThisEvents
  GameplayManager, 37
  NodeScene, 73
  Worm, 113
Hitbox, 43
  updateThis, 44

isActivated
  NodeEditorObject, 53
isActivation
  Bazooka, 3
  Cannon, 15

Grenade, 42
Weapon, 102
isActive
GUI::Component, 18
isDestroyed
Bullet, 6
NodeEditorObject, 53
NodeScene, 74
Worm, 113
isEmpty
GUI::Container, 22
WormQueue, 132
isGameFinished
GameplayManager, 37
isRoundEnding
Bazooka, 3
Cannon, 15
Grenade, 42
Weapon, 103
isSelected
GUI::Component, 19
NodeEditorObject, 53

Kinematic_Type
NodePhysicalBody, 61

matchSizeToText
GUI::Button, 11
MenuState, 45
createBackgroundWorld, 46
createGrenades, 46
draw, 47
handleEvent, 47
update, 47

NodeDestructibleRectangle, 48
addHole, 49
updateThis, 50
NodeEditorObject, 50
draw, 52
getGlobalBounds, 52
getId, 52
getSize, 52
handleEvent, 53
isActivated, 53
isDestroyed, 53
isSelected, 53
setId, 54
setName, 54
setSize, 54
update, 55
updateMouse, 55
NodePhysicalBase, 55
angleToRadians, 57
b2VecToSfVector, 57
radiansToAngle, 58
sfVectorToB2Vec, 58
update, 59
NodePhysicalBody, 59
applyForce, 61
Dynamic_Type, 61
Kinematic_Type, 61
Physical_Types, 61
setRotation, 61
Static_Type, 61
NodePhysicalSpark, 62
drawThis, 63
updateThis, 64
NodePhysicalSprite, 64
drawThis, 67
updateThis, 67
NodeRectangularPhysical, 67
drawThis, 69
NodeScene, 69
draw, 72
drawThis, 72
getAbsolutePosition, 72
getRootNode, 72, 73
handleEvents, 73
handleThisEvents, 73
isDestroyed, 74
pinNode, 74
removeDestroyed, 74
unpinNode, 74
update, 75
updateThis, 75
NodeSprite, 76
getSpriteSize, 78
NodeSprite, 77
NodeText, 78
NodeText, 79, 80
setColor, 80
setOutline, 80
setSize, 81
setString, 81
NodeWater, 81
setPosition, 83
setSize, 83
NodeWater::WaterLayer, 99

PauseState, 84
draw, 85
handleEvent, 85
update, 86
Physical_Types
NodePhysicalBody, 61
pinNode
NodeScene, 74
pop
StateStack, 94
processEvents
World, 106
push
StateStack, 94

radiansToAngle
NodePhysicalBase, 58
removeDestroyed

NodeScene, 74
WormQueue, 132
requestPush
State, 91
ResourceManager< Resource, Identifier >, 86
getResourceReference, 86, 87
storeResource, 87
rotateWeapon
Weapon, 103
run
Game, 33

saveState
StateStack, 95
setActiveFunction
GUI::Button, 11
setColor
NodeText, 80
setDamage
Worm, 113
setDeactiveFunction
GUI::Button, 11
setId
NodeEditorObject, 54
setMaxDmg
Weapon, 103
setName
NodeEditorObject, 54
Worm, 114
setOutline
NodeText, 80
setPosition
NodeWater, 83
setPositionBelow
GUI::Button, 11
GUI::Component, 19
setRange
Weapon, 104
setRotation
NodePhysicalBody, 61
setSize
GUI::Button, 12
NodeEditorObject, 54
NodeText, 81
NodeWater, 83
setSparkColor
Bullet, 6
Weapon, 104
setString
NodeText, 81
setTeam
Worm, 114
setText
GUI::Button, 12
setWorldMessage
GameplayManager, 38
sfVectorToB2Vec
NodePhysicalBase, 58
shoot
Grenade, 42
Weapon, 104
WormPlayState, 128
State, 88
draw, 90
handleEvent, 90
requestPush, 91
update, 91
StateStack, 91
draw, 93
empty, 94
handleEvent, 94
pop, 94
push, 94
saveState, 95
update, 95
Static_Type
NodePhysicalBody, 61
store
GUI::Container, 22
GUI::FixedContainer, 32
storeResource
ResourceManager< Resource, Identifier >, 87

Team, 96
TitleState, 97
draw, 98
handleEvent, 98
update, 98

unpinNode
NodeScene, 74
update
Editor, 28
EditorState, 30
GameState, 40
GUI::Button, 13
GUI::Component, 19
MenuState, 47
NodeEditorObject, 55
NodePhysicalBase, 59
NodeScene, 75
PauseState, 86
State, 91
StateStack, 95
TitleState, 98
World, 107
WormHideState, 117
WormHitState, 119
WormInventoryState, 122
WormMoveableState, 125
WormPlayState, 128
WormQueue, 132
WormWaitState, 134
updateMouse
NodeEditorObject, 55
updateMovement
WormMoveableState, 125
updateThis

Bullet, 6
Delayed_Bullet, 27
GameplayManager, 38
Hitbox, 44
NodeDestructibleRectangle, 50
NodePhysicalSpark, 64
NodePhysicalSprite, 67
NodeScene, 75
Weapon, 105
Worm, 114

Weapon, 100
activation, 102
drawThis, 102
getThumbnailSprite, 102
isActivation, 102
isRoundEnding, 103
rotateWeapon, 103
setMaxDmg, 103
setRange, 104
setSparkColor, 104
shoot, 104
updateThis, 105
World, 105
processEvents, 106
update, 107
WorldListener, 107
Worm, 108
activateState, 111
drawThis, 111
facingRight, 112
getCurrentState, 112
getName, 112
getTeam, 112
getWormSize, 113
handleThisEvents, 113
isDestroyed, 113
setDamage, 113
setName, 114
setTeam, 114
updateThis, 114
Worm, 111
WormHideState, 115
draw, 117
handleEvent, 117
update, 117
WormHitState, 118
draw, 119
handleEvent, 119
update, 119
WormInventoryState, 120
draw, 121
handleEvent, 121
update, 122
WormMoveableState, 122
direction, 126
draw, 124
facingRight, 124
handleEvent, 124
handleMovement, 125
update, 125
updateMovement, 125
WormPlayState, 126
draw, 128
handleEvent, 128
shoot, 128
update, 128
WormQueue, 129
addWorm, 130
aliveTeams, 130
draw, 130
front, 131
getNextWorm, 131
handleEvents, 132
isEmpty, 132
removeDestroyed, 132
update, 132
WormWaitState, 133
draw, 134
handleEvent, 134
update, 134