Silesian University of Technology

Computer Programming

Preliminary 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

The general idea

The main idea of the project is to prepare a game similar to *Worms*. The idea is quite big and complicated, but programming most of the mechanisms making the game similar (not exact) to *Worms* should be possible to do in time of two studying blocks. The game would use the libraries *SFML*, *Box2D* and very possibly a few others like *Clipper*.

1 What are Worms?

Worms are turn-based artillery games presented in 2D environment. 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

During gameplay, the map is gradually flooded, causing the worms to have to flee up the map. The terrain is fully destructible by the arsenal of weapons of the worms.

2 More detailed description

The game greets the user with a friendly title screen where they will be able to select the number of worms and most likely name each of them. Each player in turn is given the opportunity to make their move. Worms are selected using a queue. A freshly played turn using a particular worm will place it at the end of the queue. The player will be able to choose from the arsenal available to him and the person whose worms are the only ones to stay alive will win.

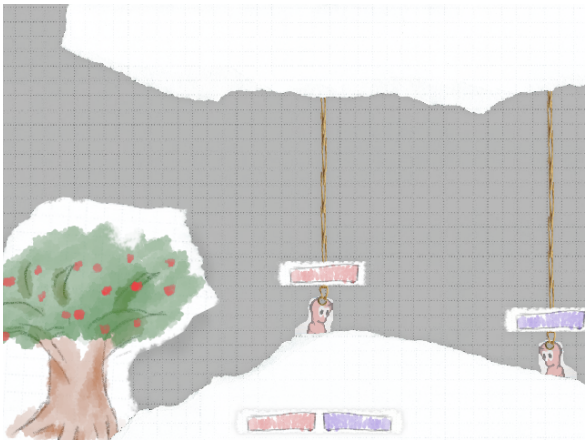
A player can play against another player, although the gameplay takes place locally. It is also not possible to play a game against a bot. Each player has the same amount of time per turn, after which the worm gives up the turn and passes the turn to the next player. If the worm uses one of the weapons in the meantime, the turn ends after another 5 seconds set aside for escape. Some weapons will be an exception here and will not end the turn. Rather, they will be auxiliary tools in the inventory.

At best, if the game can be completed then the available terrain in the game should be destructible. At worst, it may be indestructible terrain built from a drawing (image) that serves as a map (just like in Worms Armageddon), or made up of simple physical blocks contained within the game.

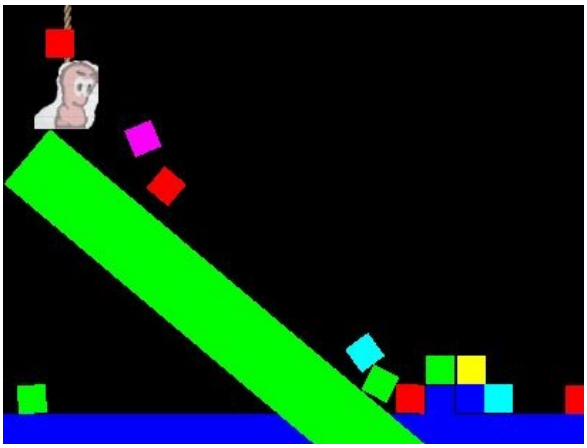
The player should move the worm around the game world using the buttons *W, A, S, D*, jumping using the *spacebar*. Equipment will be available under the *right mouse button* and weapons will be selected by *left-clicking* on the item in the inventory. At any time, the player has the option to pause the game using the *ESC* button. During the pause the game time stops and the player has the option to quit the game. Additionally, it will most likely be possible to look around the game world using the *middle mouse button*, or by moving the mouse closer to the edge of the game window.

3 How my game should look

The style I am most likely to adopt will resemble a theater created from paper. This graphic style is easy to prepare and is quite pleasing to the eye.



Desired graphic style (a sketch)



Screenshot from very early stage of production.

However, it assumes that the game will make it to full completion, where preparing the visual side will mainly be done somewhere at the end of the production process. At this point during the development process, all elements are limited to simple solids.

4 Mechanisms used

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

4.1 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.

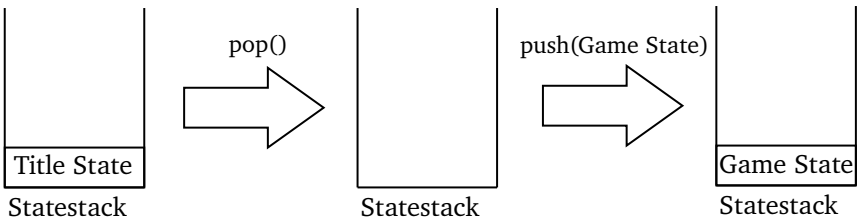


Figure 2: Changing state from one to the other

With Statestack, we 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.

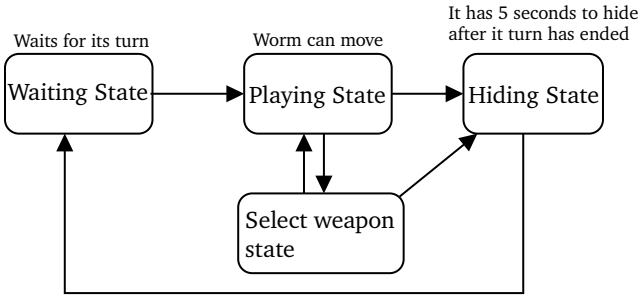


Figure 3: Possible statestack flow of a worm object shown as Finite State Machine

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.

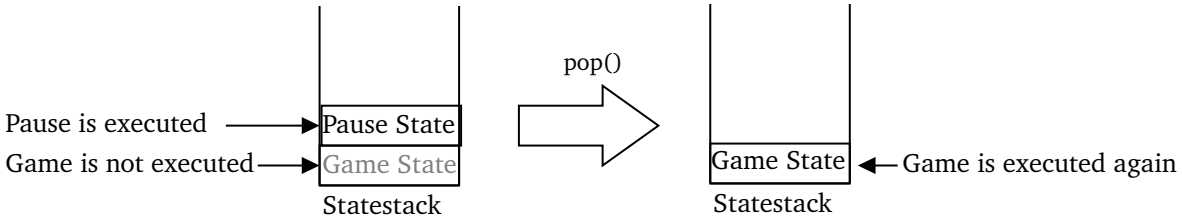


Figure 4: An example of how pause can stop processes running inside the game.

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 will use and that takes away the ability to use *std::stack*.

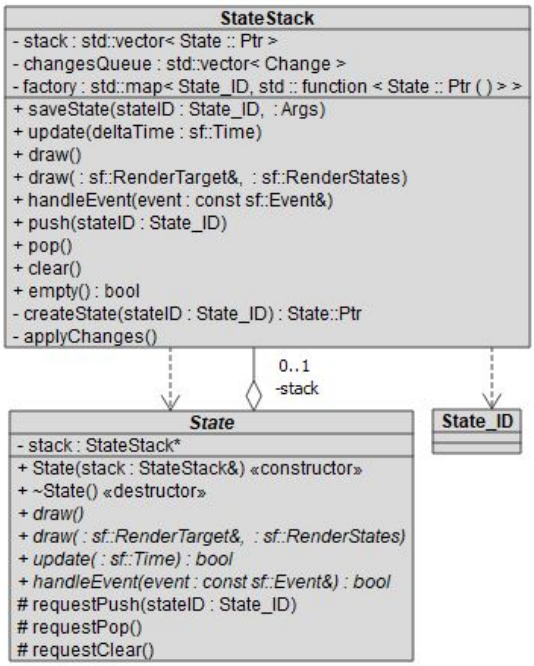


Figure 5: The implementation of the Statestack

The *changeQueue* 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. This allows a state to pop itself from the stack and push another state onto the stack.

4.2 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 6: Tree hierarchy of nodes in Godot Engine

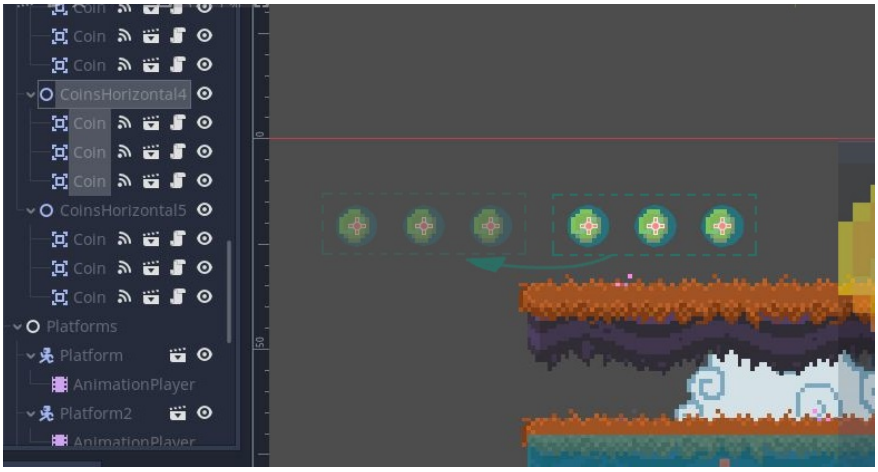


Figure 7: By moving parent node we move all pinned nodes in Godot Engine

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 7. 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.

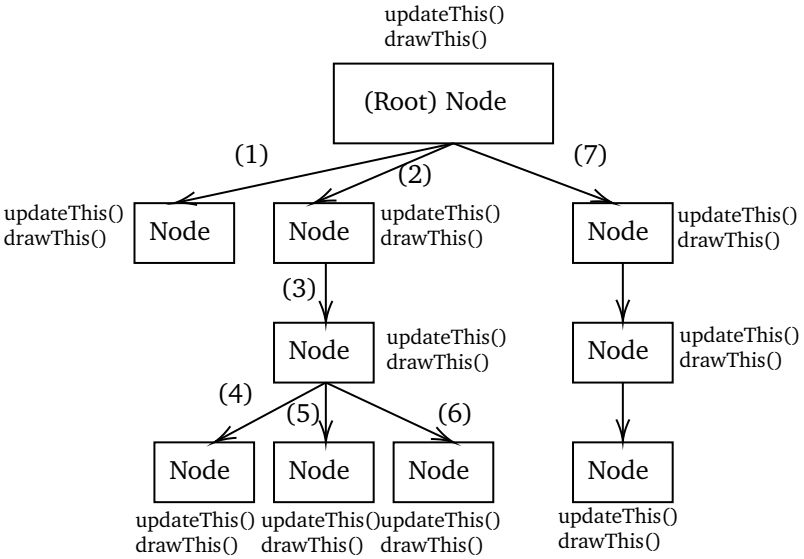


Figure 8: The order and method of displaying and updating individual nodes.

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).

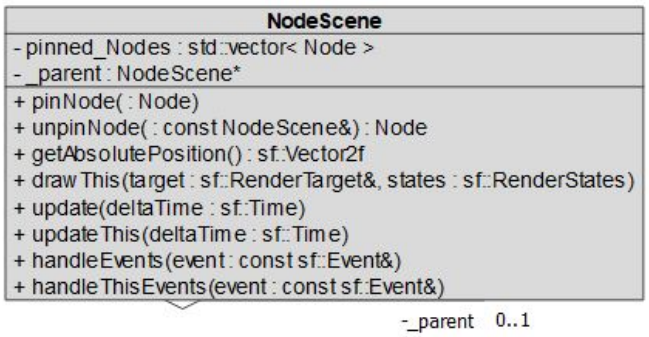


Figure 9: The base class from which other nodes will inherit

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

5 Libraries

5.1 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.

5.2 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.

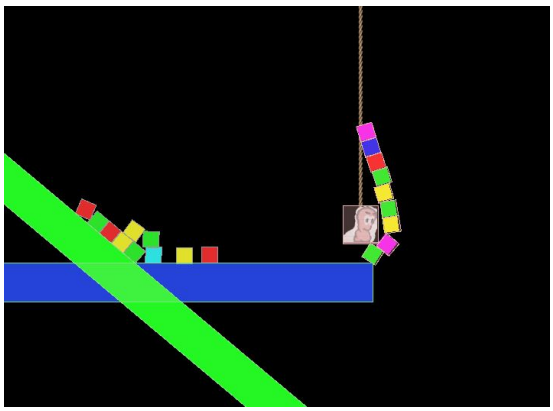


Figure 10: Example use of Box2D inside a very early version of the game.

5.3 Clipper (probably)

The Clipper Library can perform clipping, and offsetting of lines, and polygons. Depending on what stage of production the game reaches before the deadline, there is a chance that *Clipper* will allow for a destructible map. It may not be used, but it is on the list of possible libraries to use. The library is based on *Vatti's clipping algorithm*.

6 Techniques from the thematic classes

The project is large, and it is likely to use information from all classes – except maybe just Class 13 which are regular expressions. It is possible that classes 12 (Threads) may be unused too. 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

7 General scheme of the program operation

Like it was shown in **4.2. 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 **4.1. 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:

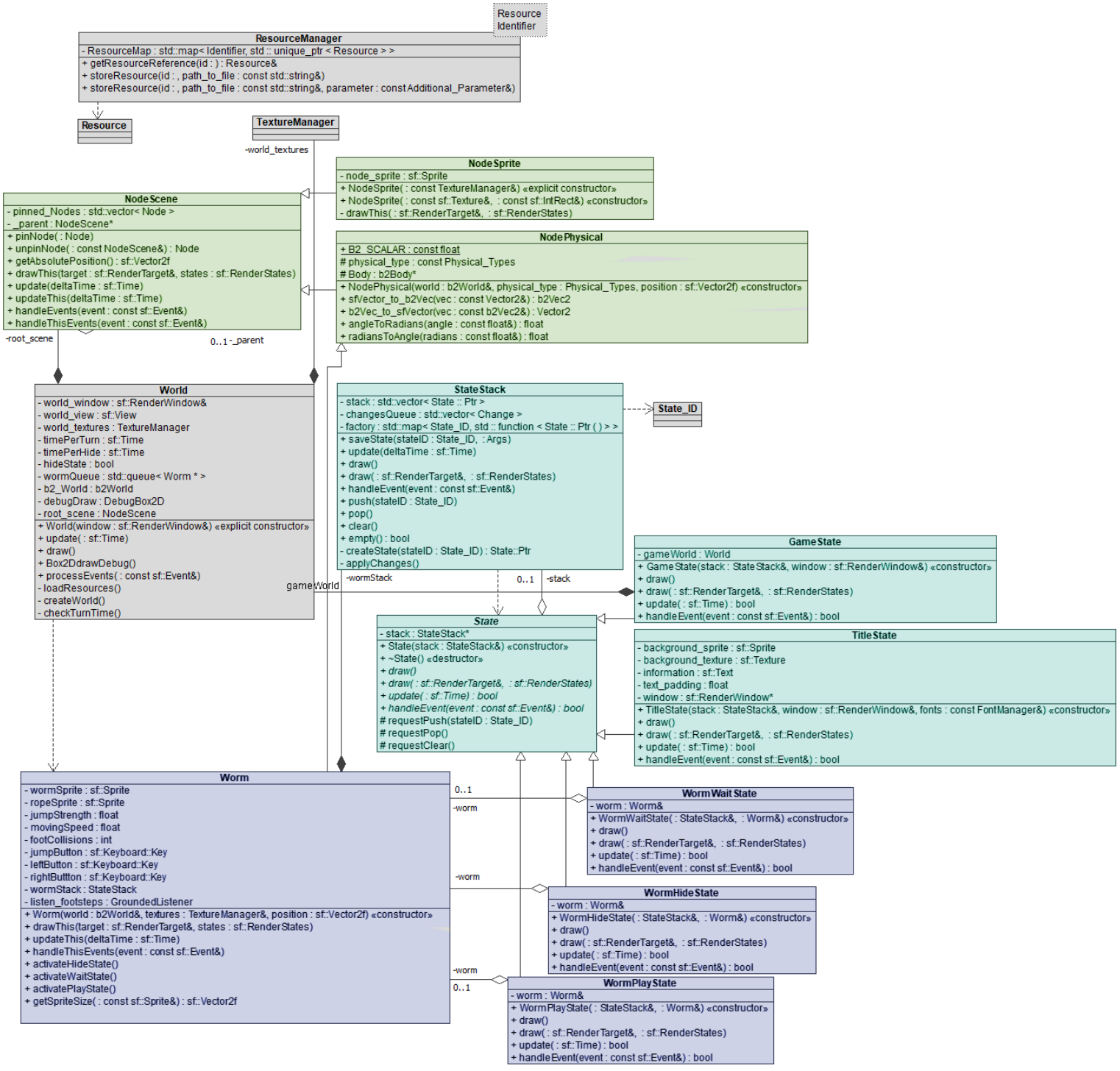
$$distance = speed \cdot time$$

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

8 Class hierarchy



Of course, these are the major classes that are scheduled at this point. Their number will gradually and significantly increase. However, it is safe to say that this is the basis for the game that is being created. From the larger classes it will probably be equipment related things as well as items to be implemented. Their implementation, however, still needs to be tested with at least a few solutions. The code will undergo many refactoring during the work. Its final version may differ from today's status.

8.1 Class hierarchy description

For the purposes of presentation, of course, the entire class hierarchy has been simplified. The states that control the flow of the application will definitely grow up, and will for example include *PauseState*.

There will be more objects inside the game. These will be items, weapons and various objects on the map. Inventory and many useful intermediate classes will also be implemented. Most likely the objects in the map will mainly inherit from *NodePhysical*. The inventory will probably be a separately written class handled inside *WormPlayState*. It will use the list of weapons found in the *Worm* class.

8.1.1 ResourceManager

Keeps resources in one place so they don't have to be loaded from a file each time.

8.1.2 NodeScene, NodeSprite, NodePhysical

From *NodeScene* inherits every other node such as *NodeSprite* (a node that has a graphic in the form of a picture) or *NodePhysical* (a node that supports game physics).
Other example implementations that may appear are:

- NodeText – stores and displays text
- NodePhysicalRectangular – a rectangle with the given color, size and physical properties, which is affected by game physics
- NodePhysicalSprite – graphics (image) that are affected by game physics.
- ... and many more

Not all objects inside the game that inherit from nodes have the prefix "Node...". *Worm* is an example of such a class that does not have it. In the future, many more objects inside the game will inherit from these classes, for example supply crates. These are the basic classes that build objects inside the game on scene.

8.1.3 World

This is the main class that is the game world. It sets up its rules, sets up objects on the scene. It is everything we see during gameplay. It is also a class that often needs to be refactored and divided into smaller parts, function and classes. It grows uncontrollably fast in the production process.

8.1.4 StateStack

It is used twice in the diagram above. In the first case, it controls the flow of the application by allowing user to go from the main menu to the game. In the second case, it allows to set different behaviors for the worm depending on what state it is in. Statestack will most likely find even more uses, although there is a need to find a golden mean here in which it allows to clearly break down the code into smaller parts. If not used properly, it may complicate them.

Statestack in my implementation is a factory that creates individual states only when they should be pushed onto the stack. This gives the possibility to reduce the used memory and computational power. I use Variadic Templates to make it possible.

9 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.