Creating a Pacman Game

This document describes how to create a game using the *PacMan Like Project* template, and the rest of the QGAMES libraries.

# Content

[Content 2](#_Toc98086246)

[Illustrations 3](#_Toc98086247)

[Introduction 4](#_Toc98086248)

[Installing the template in Visual Studio 2017 4](#_Toc98086249)

[Tools needed to extend the template 4](#_Toc98086250)

[Installing QGAMES libraries first 4](#_Toc98086251)

[Creating basic Pacman like games 4](#_Toc98086252)

[Reading Conf.xml 5](#_Toc98086253)

[Basics of the game 7](#_Toc98086254)

[The PACMAN::DataGame class 8](#_Toc98086255)

[The PACMAN::DataGame::LevelDefinition class 8](#_Toc98086256)

[The PACMAN::DataGame::LevelDefinition::FruitCondition class 9](#_Toc98086257)

[The PACMAN::DataGame::LevelDefinition::ScatterChase class 10](#_Toc98086258)

[The PACMAN::DataGame::LevelDefinition::LeaveHomeCondition class 10](#_Toc98086259)

[The PACMAN::DataGame::LevelDefinition::ElroyCondition class 11](#_Toc98086260)

[Adding a new map to the game 13](#_Toc98086261)

[The classical maps 13](#_Toc98086262)

[A map in detail though Tile’s oyes 14](#_Toc98086263)

[Defining your own maps 18](#_Toc98086264)

[Adding levels to the game 22](#_Toc98086265)

[The main method 22](#_Toc98086266)

[The menu 23](#_Toc98086267)

[Defining new levels 24](#_Toc98086268)

[Creating new scenes 26](#_Toc98086269)

[What the standard one does 26](#_Toc98086270)

[Defining a new type of scene 29](#_Toc98086271)

[Creating new monsters 33](#_Toc98086272)

# Illustrations

[1: Conf.xml 5](#_Toc98086209)

[2: Where to find the attributes of the scale in your configuration (windows 10) 6](#_Toc98086210)

[3: The Menu 7](#_Toc98086211)

[4: The constructor of LevelDefinition 8](#_Toc98086212)

[5: Constructor of FruitCondition 9](#_Toc98086213)

[6: The default gifts (usually known as fruits) used in pacman 10](#_Toc98086214)

[7: Constructor of ScatterChaseCycle 10](#_Toc98086215)

[8: Constructor of LeavHomeCondition 11](#_Toc98086216)

[9: Constructor of ElroyCondition 11](#_Toc98086217)

[10: Defining a level 11](#_Toc98086218)

[11: Defining the maps 13](#_Toc98086219)

[12: Using Tiled to modify our mazes 13](#_Toc98086220)

[13: Defining the worlds 14](#_Toc98086221)

[14: Tiles used to locate 15](#_Toc98086222)

[15: Zones 17](#_Toc98086223)

[16: To draw the path of the maze 17](#_Toc98086224)

[17: The tiles used to draw the external aspect of the maze 18](#_Toc98086225)

[18: Standard implementation of maps.xml 18](#_Toc98086226)

[19: A piece of the implementation of maps.xml in PacManII 19](#_Toc98086227)

[20: Ms. Pacman Type 1 19](#_Toc98086228)

[21: Standard implementation of worlds.xml 20](#_Toc98086229)

[22: An example of th implementation of worlds.xml could look like after adding maps 20](#_Toc98086230)

[23: Adding a new level, but including the new type of map 21](#_Toc98086231)

[24: main method of a very basic pacman game 22](#_Toc98086232)

[25: A piece of the standard menu definition 23](#_Toc98086233)

[26: PacManII main method 24](#_Toc98086234)

[27: Defining in the menu new types of levels. PacManII example 25](#_Toc98086235)

[28: The class BasicScene 26](#_Toc98086236)

[29: Implementation of the class BasicScene 27](#_Toc98086237)

[30: The definition of the class StandardScene 27](#_Toc98086238)

[31: Implementation of the class StandardScene 29](#_Toc98086239)

[32: A new class with a new monster. Example of PacManII 29](#_Toc98086240)

[33: Implementation of a new class with a new type of monster 30](#_Toc98086241)

[34: Redefining the world builder 31](#_Toc98086242)

[35: Implementing the method createSceneObject 31](#_Toc98086243)

[36: Redefining the class Gam 32](#_Toc98086244)

[37: Redefining the fil worlds.xml 32](#_Toc98086245)

# Introduction

## Installing the template **in** Visual Studio 2017

Copy the .zip file into the directory *./Visual Studio 2017/My Exported Templates* usually located in *./Users/XXX/Documents* in the same disk where *Windows OS* is installed.

To a better understanding of how to extend the library or the template and basic games themselves, it is needed a deeper understanding of how QGAMES library has been built and works. Please read the book “How to make a 2D / 3D engine for complex games”.

### Tools needed to extend the template

To extend the template there are a number of tools that can (should maybe) be used to facilitate the job. We suggest downloading and installing them before moving forward:

* ***Tiled*** (<https://www.mapeditor.org/>): Probably the most important one. It is needed to build the maps being part of the different scenes and worlds of the game.
* ***TexturePackcageGUI*** (<https://www.codeandweb.com/texturepacker>): Used to create sprite sheets starting from the individual elements.
* ***SpriteFontBuilder (SFB)*** (<https://www.johnwordsworth.com/projects/sprite-font-builder/>): Used to create fonts. It is extremely useful indeed.
* ***GIMP*** (<http://www.gimp.org.es/>): To build the o modify the individual element of both sprites and backgrounds. The sprites built usually are integrated in a sprite sheet, using the **TexturePackageGUI**.

### Installing QGAMES libraries first

Before creating any new game, it is needed to install QGAMES library.

Download it from <https://sourceforge.net/projects/qgames-library/>, and follow the instructions explained in README.rtf file to lave it ready to operate.

Remember that QGAMES libraries do not include other required dependencies. Install those ones before moving forward: SDL, OGRE, … See how to install fully QGAME library in its *Read.rtf* file carefully. The are needed to compile the games.

### Creating basic Pacman like games

Add a new project. Select the option *Add a new project*. Then, under the folder *Visual C++* should appear the type *Pacman Like Project* (represented with the icon of our lovely character). Introduce a name for your game and select the folder under which you want to locate it (we suggest *games* or *samples*). Click *Ok*.

Visual Studio will expand all elements included in the template, creating your project. By the way it is fully executable.

By default, the template will locate *.obj* elements in the folder *../../obj* (taking the game folder as the origin) and the executables in the folder *../../exe*. The template will look for the libraries it needs under the folder *../../lib*.

If you have located the base of your game in a folder different that *games* or *examples*, please check that the compilation process will find what it is needed correctly.

The template does not directly include the needed runtime libraries (*SDL*, *TinyXML*, *ZIP*). The new executable will try to find them under the new game directory. By default, the template will expect an SDL executable. So, copy the libraries (*.dll* files) from any another example in the folder of the new game[[1]](#footnote-1), or from the directory *commonlibs* located at the installation dir.

### Reading Conf.xml

Let us take a look to the *conf.xml* file (in the game’s directory). It will look like such as:

<?xml version="1.0"?>

<Properties>

<!-- General Properties -->

<!-- The name of the game -->

<Property id="GAMENAME" value="Pacman"/>

<!-- The icon of the game -->

<Property id="GAMEICON" value="50000"/> <!-- Represents the logo of pacman -->

<!-- Where the temporal files are stored in release version,

and also the configuration saved -->

<Property id="DATADIR" value="C:/Temp"/>

<!-- When windows is in Full 4K and the scale letters

are in other proportion than 1 (take care of this prm) -->

<Property id="XSCALE" value="2.5"/>

<Property id="YSCALE" value="2.5"/>

<!-- Related with the visible screen -->

<Property id="SCREENWIDTH" value="900"/>

<Property id="SCREENHEIGHT" value="600"/>

<Property id="SCREENXPOSITION" value="10"/>

<Property id="SCREENYPOSITION" value="10"/>

<!-- The number of lives -->

<Property id="LIVES" value="3"/>

<!-- Whether to show the FPS -->

<Property id="FPS" value="YES"/>

<!-- The property with the value of the initial level -->

<Property id="LEVEL" value="1"/>

<!-- To draw the target of every artist. Important to follow the game! -->

<Property id="DRAWTARGET" value="YES"/>

<!-- The password to avoid being destroyed -->

<Property id="PASSWORD" value="PACMANPWD\_WRONG"/>

</Properties>

1: Conf.xml

Many of the attributes defined can be changed to accomplish later your own needs.

All of them are quite self-explained except maybe a couple of them:

* ***XSCALE* & *YSCALE***.

SDL libraries (version 2.0) have a bug.

They don’t manage windows’ scale properly. So, it is needed to set it externally (using software). Those variables define that scale. Their values should be equivalent to the values set at *Configuration -> System -> Display -> Scale & Distribution:*

Una captura de pantalla de una computadora

Descripción generada automáticamente

2: Where to find the attributes of the scale in your configuration (windows 10)

Compare what is define here with what appears in the *Conf.xml*. If there is any difference, align the two values, and finally restart the game. Now, it should work if it did not do before properly.

* ***PASSWORD***, a value of PACMANPWD will make you never die. Any other value, as the one it has by default, is the normal behaviour.

Let us start to understand the different elements of the game and how to modify them.

# Basics of the game

A game consists of different levels.

The levels are indicated (in this image shown, the option selected is *classic*) in the menu of the game. With that option many things can be configured: The main character to be used, the types and number of mazes where the play is going to happen, the difficulty of each and even, the type of monsters to appear and when. Ah! and the behaviour of the fruit.

To, to create your own version of the game, this is the first place to actuate. However, it would be the last to be explained!

Texto

Descripción generada automáticamente con confianza baja

3: The Menu

Each type of game (Classical,…) is associated with a *PACMAN::DataGame* instance. That *DataGame* class can even be extended to include your own needs. The class is defined in the file *../../include/PacManLike/pcdataga.hpp*. Look to it for better understanding.

## The PACMAN::DataGame class

The class *PACMAN::DataGame* requires only two parameters to be built:

A list with all levels (instances of the class *PACMAN::DataGame::LevelDefinition*), and the number of points after which the player is gifted with an additional life.

### The PACMAN::DataGame::LevelDefinition class

Every *QGAMES::DataGame::LevelDefinition* (constructor) is made ap of the following 19 attributes[[2]](#footnote-2):

PACMAN::DataGame::LevelDefinition::LevelDefinition

(

int wT,

int sT,

int mT,

int iM,

int pB,

int pPB,

double mSC,

const FruitConditions & fC,

const ScatterChaseCycles & sC,

const LeaveHomeConditions & lHC,

double pS,

double pED,

double pWF,

double pWEFD,

double gS,

double gWF,

double gWE,

double gWT,

const ElroyConditions & eC

)

4: The constructor of LevelDefinition

* **wT**: The number of the world of the level.
* **mT**: The number of the scene of the level.
* **sT**: The number of the map corresponding to the level.

World, scenes and maps can be reused across different levels (changing maybe other attributes).

* **iM**: A number to indicate whether after the level an intermission screen is required. -1 will mean no, and another number will mean the type of intermission screen to show. By default, 0 is the standard one, that is: Pacman being chased by monsters and after leaving the screen by the right side, the other way around.
* **pB**: The points given when a normal ball is eaten.
* **pBB**: The points given when a power ball is eaten.
* **mSC**: Seconds the threaten status last after eating a ball.
* **fC**: The fruit conditions for the level (later analysed).
* **sC**: The scatter/chase monster cycles. In a level the monsters go repeatedly through two situations: Either to run away to their homes (something that has to be defined per each) or to go for the main character. What to do so, is defined in this parameter.
* **lHC**: The conditions to leave home (per monster, and later analysed).
* **pS**: The normal speed of pacman (in relative way from 0 to 1 float number).
* **pED**: The speed of pacman when is eating balls.
* **pWF**: The speed of pacman when it is frighting monsters (after e.g. eaten a ball).
* **pWEF**: The speed of pacman when is eating power balls.
* **gS**: The normal speed of the ghost.
* **gWF**: The speed of a ghost when is being threatened by pacman.
* **gWE**: The speed of an enemy when it is exiting home.
* **gWT**: The sped of the monster when it is crossing a tunnel.
* **eC**: The very famous *elroy* conditions applied usually only a *Blinky*. The *elroy* conditions define speed conditions when there are just a few balls pending to be eaten.

### The PACMAN::DataGame::LevelDefinition::FruitCondition class

The Fruit conditions is a list of objects *PACMAN::DataLevel::LevelDefinition::FruitCondition*, made up of the following attributes:

FruitCondition

(

int bS,

int bP,

int nB,

double sD,

double spd,

int pId

)

5: Constructor of FruitCondition

* **bS**: The number symbol to be used. According with the following picture (read from left to tight and from top to bottom), and located in the directory *./forms*

Imagen que contiene luz

Descripción generada automáticamente

6: The default gifts (usually known as fruits) used in pacman

* **bP**: The points given when eaten.
* **nB**: The number of balls to be eaten by the pacman before it appears.
* **sD**: The number of seconds for the fruit to disappear after appearing.
* **spd**: The speed of the movement across the maze. 0 will imply it is stopped at it initial position.
* **pID**: The initial position on the maze. It has to me marked in the definition of the maze or being –1 meaning random.

In a level a list of them can be defined. The fruits are sequential. That means the next level in the list no appear until the previous one has disappear.

### The PACMAN::DataGame::LevelDefinition::ScatterChase class

The scatter / chase cycles is a list of objects of the type: *QGAMES::DataGame::LevelDefinition::ScatterChaseCycle*. Each of these elements need the following attributes to be built. At least one of them should exist in the definition of the level.

ScatterChaseCycle

(

QGAMES::bdata sS,

QGAMES::bdata sC

)

7: Constructor of ScatterChaseCycle

* **sS**: The seconds looking for home.
* **sC**: The seconds chasing pacman.

### The PACMAN::DataGame::LevelDefinition::LeaveHomeCondition class

The leaving – home conditions is a list of conditions for each monster to leave home. It belongs to the class *PACMAN::DataGame::LevelDefinition::LeaveHomeCondition*. There should be defined at least one per level. The attributes needed to define it are:

LeaveHomeCondition

(

double sL,

const std::vector <int>& dL

)

8: Constructor of LeavHomeCondition

* **sL**: Number of seconds maximum to leave. When no ball is eaten, this is the number of seconds to wait. Any time a ball is eaten, the counter is started back.
* **dL**: The list of the dots eaten by pacman before each monster leaves home.

### The PACMAN::DataGame::LevelDefinition::ElroyCondition class

And the elroy conditions, is an object of the type *PACMAN::DataGame::LevelDefinition::ElroyCondition*. The elroy conditions usually activates when only a couple of balls left. It is not for all the monsters. As the number of remaining balls gown down, the speed uses to go up. In the default game, only Blinky has it.

ElroyCondition

(

const std::vector <bool>& cS,

const std::vector <int> nL,

const std::vector <double>& mS

)

9: Constructor of ElroyCondition

* **cS**: Defne whether to apply or not the elroy condition.
* **nL**: The number of balls pending in the maze before activating the elroy condition.
* **mS**: The speed for the monster when the elroy condition is activated.

So, to create a level, could be done with the following instructions, e.g:

PACMAN::DataGame::LevelDefinition::ScatterChaseCycles c1 =

{ { 7.0f, 20.0f }, { 7.0f, 20.0f }, { 5.0f, 20.0f }, { 5.0f, 999999.0f } };

PACMAN::DataGame::LevelDefinition::LeaveHomeConditions lH1 =

{ { 4.0f, { 0, 0, 21, 42 } }, { 4.0f, { 0, 5, 12, 22 } } };

PACMAN::DataGame::LevelDefinition::ElroyConditions eC1 =

{ { { true /\*\* No more monsters. \*/, false }, { 10, 0 }, { 0.8f, 0.0f } },

{ { true, false }, { 5, 0 }, { 0.85f, 0.0f } } };

PACMAN::PACMAN::DataGame::LevelDefinition level (

{ \_\_PACMAN\_BASICWORLD\_\_,

\_\_PACMAN\_BASICBLOCKSCENES\_\_,

\_\_PACMAN\_BASICBLUEMAP\_\_,

-1, // Meaning none...

15, 75, 8.0,

{ { 4, 200, 40, 10.0, 0.0, 0 } }, // Cherries

c1, lH1,

0.80, 0.71, 0.90, 0.79,

0.75, 0.50, 0.40, 0.40,

eC1 }),

10: Defining a level

Using the C++ initialization list possibilities.

The class *PACMAN::DataGame* includes a method (static) called: ***classicCoreLevels ()***, that returns a list with the first 21 levels standard in a classic Pacman (including their configuration).

This class also defined another method: ***classicDataGame (int nL = \_\_PACMAN\_MAXNUMBERLEVELS\_\_)***, that returns a full DataGame object with as many levels as required (always more than 21). The first 21 levels are the core classical ones. The rest are all a copy of this last one (in regards to the attributes), changing the color of the maze.

# Adding a new map to the game

## The classical maps

The maps used for any are defined in the file: pcmaps.xml located in the root dir of the test game you created. Its aspect is as follows:

<?xml version="1.0"?>

<Maps>

<!-- Includes -->

<Include file="admaps.xml"/>

<!-- The maps with different colors -->

<Map id="50000" type="1" file="maps/normalmap\_blue.tmx"/>

<Map id="50100" type="1" file="maps/normalmap\_green.tmx"/>

<Map id="50200" type="1" file="maps/normalmap\_yellow.tmx"/>

<Map id="50300" type="1" file="maps/normalmap\_red.tmx"/>

<!-- Simple very classic map -->

<Map id="51000" type="1" file="maps/map.tmx"/>

<!-- A simple blue map with a couple of balls to eat and

just for testing purposes -->

<SetOfMaps baseId="52000" number="2" type="1" file="maps/maptest.tmx"/>

</Maps>

11: Defining the maps

The file referred in that file, are in the subdir maps. They all can be open with Tiled tool. You will notice that the maps id-ed as 5000, 50100, 50200 and 50300 represent the different versions of the very classical maze in different colours: blue, green, yellow and red.

We will explain how to create your owns later. Don’t worry!.

Interfaz de usuario gráfica

Descripción generada automáticamente

12: Using Tiled to modify our mazes

The worlds and scenes are defined in pcworlds.xml. A set of that file is:

<?xml version="1.0"?>

<Worlds>

<!-- Includes -->

<Include file="adworlds.xml"/>

<World id="50000">

<Scenes>

<Scene id="50000">

<Maps>

<Map id="50000" file="maps/normalmap\_blue.tmx"/>

<Map id="50100" file="maps/normalmap\_green.tmx"/>

<Map id="50200" file="maps/normalmap\_yellow.tmx"/>

<Map id="50300" file="maps/normalmap\_red.tmx"/>

</Maps>

<Entities>

…

</Entities>

</Scene>

…

</Scenes>

</World>

</Worlds>

13: Defining the worlds

Observe that the world 5000 and the scene defines all the other maps shown above.

The scene 5000 is the classical one. That is with 4 monsters and fruit!.

This what the macros *\_\_PACMAN\_BASICWORLD\_\_* (5000) and *\_\_PACMAN\_BASICBLOCKSCENES\_\_* (5000) contains.

*\_\_PACMAN\_BASICBLUEMAP\_\_*, and the equivalent for other colours represents 50000 (*\_\_PACMAN\_BASICBLUEMAP\_\_*), 50100 (*\_\_PACMAN\_BASICGREENMAP\_\_*), 50200 (*\_\_PACMAN\_BASICYELLOWMAP\_\_*) and 50300 (*\_\_PACMAN\_BASICREDMAP\_\_*).

Different types of scenes, with different behaviours and even monsters could be defined, but leave it for the advanced section of this document.

## A map in detail though Tile’s oyes

Open any of the four classical mazes using tiled.

First, you will notice the map is a little smaller (1 tile per side) than the net available. This is for two reasons: To be able to define target positions for the monsters mainly (when the are under scatter mode e.g.) out of the maze. Positions that they could never reach and therefore they will be around of them. And, to allow an artist can exit (visually speaking) though a side of a tunnel and back from the other side.

The engine of the pacman library is prepare for that circumstance, so keep it in your own designs.

The map is made up different layers.

* **ArtistsLocations**:

Layer to define important locations for the artists playing.

Each important location will be marked with an specific tile of the type *location* (see right – above):

Interfaz de usuario gráfica, Aplicación, Tabla

Descripción generada automáticamente

14: Tiles used to locate

Specifically, those important locations are:

* + Scatter monster positions:

Where the different monsters must go under scatter mode.

Every Monster has a number to identify him: *Blinky* = 0, *Pinky* = 1, *Inky* = 2, *Clyde* = 3.

The tiles used to indicate that scatter positions are the ones in the top of the sprite sheet shown above and in the sequential position that the number of each monster points.

Maybe you have already noticed, it could be possible to define 10 different types of monsters!

* + Monster initial positions:

Where the different monsters start any play from.

The tiles used to indicate that scatter positions are the ones in the second line of the sprite sheet shown above and in the sequential position that the number of each monster points.

You will see that the position could be located out the monsters’ home, as it is Blinky’s case.

* + Pacman initial position:

Where the pacman starts from.

It is indicated with the 1 tile in the 3 line of the sprite sheet above.

* + Things initial position:

Pacman framework manages no fruits but things instead. One type of thing is a fruit.

The initial position of a thing is marked with the tiles of the 4th line in the sprite sheet above. By default, for the normal fruit only the first tile of that row is needed.

* **MazeLocations**:

Locations that point important places for all the artists moving in the maze.

All maze important positions are marked with tiles of the 7th line of the positions sprite sheet.

Those locations are:

* + The location of the enter of monster’s home:

Very important to know to avoid e.g. that pacman can enter there r to know when monsters are finally totally free to move across the maze.

That position is marked with the first tile.

* + The location where the monsters must go back after being eaten:

Marked with the third tile in of the row 7th.

* + Tunnel connection:

The second tile of the 7th line is used for so and also the 4th onwards.

There must always be a par of them. To indicate the connection! This point is analysed at construction time and no map is allowed to be built not following this rule.

* **MazeZones**:

There are also important zones to be highlighted in the any maze. Those zones imply (or could) different behaviours when a monster or pacman (e.g.) pass thought them.

To mark those zones, it will be used the sprite sheet declare under the pattern zones, this aspect:



15: Zones

The zones considered by default are:

* + The yellow one:

To indicate monsters’ home. In that zone, the monsters move slowly.

* + The blue one:

To indicate a tunnel zone. In that zone the monsters move again slowly unless there are about to be eaten.

* + The white one:

To indicate an special zone of the maze where once a monster enters in it, it has to move from right to left or the other way around, but it will not be possible him to leave the route in other directions.

* **Directions**:

Mabe the core layer of the map definition. It defines the possibilities to move. It has to be dran using the sprite sheet of the pattern *mazedirs*:



16: To draw the path of the maze

* **Maze**:

This is the graphical external graphical representation of the maze. This is the only layer shown. The rest are only for the game to control how to manage the maze, but none of them will be shown at all.

Here you could use your own graphics.

That graphics must be added under the pattern corners.

Th default ones are including in the sprite sheet having this aspect:

Diagrama

Descripción generada automáticamente

17: The tiles used to draw the external aspect of the maze

* **Background**:

Just the background. We can also play with it!

Look carefully how the standard maps have been defined to understand the meaning of each patter, layer, sprite sheet used, and so.

## Defining your own maps

You already have the tools to start.

You will find an empty map (*empty.tmx*) in the *.\maps* directory. Copy it with your name and start what you imagination leaves you to do! This is here to simply the initial steps of defining any maze.

Take care. All the layers defined in the previous section should always exists and also the patterns containing the sprite sheets used for different reasons. Otherwise, a crack at running time could arise.

Once you have it, you have to introduce it in the list of maps to be considerer.

You could modify the file .\pcmaps.xml to do so. But it is a little bit dangerous as it could modify the normal behaviour of the standard game. Open *./maps.xml* instead:

<?xml version="1.0"?>

<Maps>

<!-- Includes -->

<Include file="pcmaps.xml"/>

</Maps>

18: Standard implementation of maps.xml

It is prepared to extend the system.

By default, it only includes the *.\pcmaps.xml* already reviewed. To ways can be chosen at this point depending on the question: Do we want that the new map replaces other previous or is it a real new one to be considered in different levels that the standard ones?

If the wish is the first, the definition of the new map should be added before the include instruction, and after in the other situation.

An example from PacManII:

<?xml version="1.0"?>

<Maps>

<!-- Includes -->

<Include file="pcmaps.xml"/>

<!-- Miss Pacman 1 -->

<Map id="60000" type="1" file="maps/misspacman1map\_blue.tmx"/>

<Map id="60200" type="1" file="maps/misspacman1map\_red.tmx"/>

<Map id="60400" type="1" file="maps/misspacman1map\_green.tmx"/>

<Map id="60600" type="1" file="maps/misspacman1map\_yellow.tmx"/>

…

</Maps>

19: A piece of the implementation of maps.xml in PacManII

Where new maps are added to support the different Ms. Pacman mazes in different colours. As they are news (the have a different id) and they don’t expect to replace any other that already existed, they are added after the *include* statement.

One of these maps is, under Tiled:

You can notice that the sprite sheet used to draw the maze is different that the standard one. It is possible. But the patter used is the same, corners” and the name of the layer when everything has been drawn is the same as well, Maze.

Interfaz de usuario gráfica, Diagrama

Descripción generada automáticamente

20: Ms. Pacman Type 1

Now these maps should be included in scenes.

In this case imaging the scene we want to use for our game is the same that the standard one, that is, the one having fruits and 4 monsters. We only want to change the possibility of having different type of maps to play with.

Taking a look to the file: *.\world.xml*

<?xml version="1.0"?>

<Worlds>

<!-- Includes -->

<Include file="pcworlds.xml"/>

</Worlds>

21: Standard implementation of worlds.xml

We must wonder the same question than above.

In this case, as we want to have the same scene than the standard one but including some additional maps, we should come up with a file like:

<?xml version="1.0"?>

<Worlds>

<World id="50000">

<Scenes>

<Scene id="50000">

<Maps>

<Map id="50000" file="maps/normalmap\_blue.tmx"/>

<Map id="50100" file="maps/normalmap\_green.tmx"/>

<Map id="50200" file="maps/normalmap\_yellow.tmx"/>

<Map id="50300" file="maps/normalmap\_red.tmx"/>

<Map id="60000" file="maps/misspacman1map\_blue.tmx"/>

<Map id="60200" file="maps/misspacman1map\_red.tmx"/>

<Map id="60400" file="maps/misspacman1map\_green.tmx"/>

<Map id="60600" file="maps/misspacman1map\_yellow.tmx"/>

</Maps>

<Entities>

…

</Entities>

</Scene>

…

</Scenes>

</World>

<!-- Includes -->

<Include file="pcworlds.xml"/>

</Worlds>

22: An example of th implementation of worlds.xml could look like after adding maps

Where it has the same aspect that in ./pcworlds.xml but including the Ms. Pacman mazes.

Now the definition of a level like as follows will be possible, where the number of map to be used has been changed.

PACMAN::DataGame::LevelDefinition::ScatterChaseCycles c1 =

{ { 7.0f, 20.0f }, { 7.0f, 20.0f }, { 5.0f, 20.0f }, { 5.0f, 999999.0f } };

PACMAN::DataGame::LevelDefinition::LeaveHomeConditions lH1 =

{ { 4.0f, { 0, 0, 21, 42 } }, { 4.0f, { 0, 5, 12, 22 } } };

PACMAN::DataGame::LevelDefinition::ElroyConditions eC1 =

{ { { true /\*\* No more monsters. \*/, false }, { 10, 0 }, { 0.8f, 0.0f } },

{ { true, false }, { 5, 0 }, { 0.85f, 0.0f } } };

PACMAN::PACMAN::DataGame::LevelDefinition level (

{ \_\_PACMAN\_BASICWORLD\_\_,

\_\_PACMAN\_BASICBLOCKSCENES\_\_,

60000, // For Ms. Pacman type!

-1, // Meaning none...

15, 75, 8.0,

{ { 4, 200, 40, 10.0, 0.0, 0 } }, // Cherries

c1, lH1,

0.80, 0.71, 0.90, 0.79,

0.75, 0.50, 0.40, 0.40,

eC1 }),

23: Adding a new level, but including the new type of map

# Adding levels to the game

## The main method

Now let’s look at the main standard method:

/\*\*

\* @file

\* File: main.cpp \n

\* Framework: Commty Game Library (CGL) \n

\* Author: Ignacio Cea Forniés (Community Networks) \n

\* Creation Date: 21/08/2021 \n

\* Description: Main programm for PacManII game

\* Versions: 1.0 Initial

\*/

#include "stdafx.h"

#include <PacManLike/pcinclude.hpp>

using namespace PACMAN;

#ifndef \_CONSOLE

#include <SDL.h>

#ifdef \_\_cplusplus

#define C\_LINKAGE "C"

#else

#define C\_LINKAGE

#endif /\* \_\_cplusplus \*/

#if \_MSC\_VER >= 1900

extern C\_LINKAGE FILE \_\_iob\_func[3] = { \*stdin,\*stdout,\*stderr };

#endif

extern C\_LINKAGE int main(int argc, char \*argv[])

#else

int \_tmain (int argc, char \*argv [])

#endif /\* \_CONSOLE \*/

{

#ifdef NDEBUG

// In the release version, the resources are in a zip file

// This instruction reads them and store in temporaly files when needed!

// The resource reader is deleted at the end, and all the temporal file with them!

QGAMES::InZipResourceReader rR

(QGAMES::ConfigurationFile (std::string (\_\_GAME\_CONFIGURATIONFILE\_\_)).

property (std::string (\_\_GAME\_DATADIRPROPERTYNAME\_\_)));

#endif

// The more difficult the game is more number of levels it has...

Game game ({

DataGame::classicDataGame (25), //Easy

DataGame::classicDataGame (30), // Difficult

DataGame::classicDataGame (35) // Hard

});

game.setLinkDrawToFrameRate (true);

game.setPreLoad (false);

game.exec ();

return (0);

}

24: main method of a very basic pacman game

The class Game is created in the lasts lines.

The constructor of *PACMAN::Game* is:

Game (const DataGames& dt);

It receives a list of *PACMAN::DataGame* instances, each of which will be assigned to the different levels shown in the menu. In our standard example the method ***classDataGame()*** defined within the class *QGAMES::DataGame* is used to instantiate three versions of the classical game but varying the number of levels maximum among 25, 30 or 35.

Remember that the level from 21 onwards is always a copy of the last one but changing the colour of the maze.

## The menu

If more were added, there must be possible to select also them from the menu.

The menu is defined in the file: ./adguisystem.xml. Part of this file is:

<?xml version="1.0"?>

<GUISystems>

<!-- Standard main menu used in selection game states -->

<GUISystem id="10000" main="10001">

<!-- The main widget has to be a composite (type 0)

to allow the reference to other widgets -->

<Widget id="10001" type="0" position="10,10,0">

<RefWidget id="10002"/>

<RefWidget id="10003"/>

<RefWidget id="10004"/>

<RefWidget id="10005"/>

<RefWidget id="10006"/>

<RefWidget id="10007"/>

<RefWidget id="10008"/>

<RefWidget id="10009"/>

<RefWidget id="10010"/>

</Widget>

…

<!-- Option 6: To select the difficulty of the game -->

<Widget id="10007" type="3" position="0,300,0">

<Attributes>

<Attribute id="Form" value="60013"/>

<Attribute id="BaseChr" value="32"/>

<Attribute id="AlphaOn" value="255"/>

<Attribute id="AlphaOff" value="255"/>

<Attribute id="PixelsAmong" value="0"/>

<Attribute id="Direction" value="HORIZONTALRIGHT"/>

<Attribute id="StringValues"

value="Easy,Medium,Difficult "/>

<Attribute id="Text" value="Classic"/>

</Attributes>

</Widget>

…

25: A piece of the standard menu definition

The option 6 contains the explanation of each of the three levels of difficulty.

So, if a new list of levels is required, this file has to be modified. Add that modification in the file: ./guisystem.xml, before the include. In this case, just to change a single option is needed to redefine the whole file.

## Defining new levels

Let’s a look about how PacManII does.

The main method is now in PacManII:

/\*\*

\* @file

\* File: main.cpp \n

\* Framework: PacManII \n

\* Author: Ignacio Cea Forniés (Community Networks) \n

\* Creation Date: 21/08/2021 \n

\* Description: Main programm for PacManII game

\* Versions: 1.0 Initial

\*/

#include "stdafx.h"

#include "Game.hpp"

#include "DataGame.hpp"

using namespace PacManII;

#ifndef \_CONSOLE

#include <SDL.h>

#ifdef \_\_cplusplus

#define C\_LINKAGE "C"

#else

#define C\_LINKAGE

#endif /\* \_\_cplusplus \*/

#if \_MSC\_VER >= 1900

extern C\_LINKAGE FILE \_\_iob\_func[3] = { \*stdin,\*stdout,\*stderr };

#endif

extern C\_LINKAGE int main(int argc, char \*argv[])

#else

int \_tmain (int argc, char \*argv [])

#endif /\* \_CONSOLE \*/

{

#ifdef NDEBUG

// In the release version, the resources are in a zip file

// This instruction reads them and store in temporaly files when needed!

// The resource reader is deleted at the end, and all the temporal file with them!

QGAMES::InZipResourceReader rR

(QGAMES::ConfigurationFile (std::string (\_\_GAME\_CONFIGURATIONFILE\_\_)).

property (std::string (\_\_GAME\_DATADIRPROPERTYNAME\_\_)));

#endif

// 5 types of game...

Game game ({

PACMAN::DataGame::classicDataGame (),

DataGame::missDataGame (),

DataGame::mineDataGame (),

DataGame::hardDataGame (),

DataGame::trainingDataGame ()

});

game.setLinkDrawToFrameRate (true);

game.setPreLoad (false);

game.exec ();

return (0);

}

26: PacManII main method

And the options defined in ./guisystem.cml is:

<?xml version="1.0"?>

<GUISystems>

<!-- Standard main menu used in selection game states -->

<GUISystem id="10000" main="10001">

<!-- The main widget has to be a composite (type 0)

to allow the reference to other widgets -->

<Widget id="10001" type="0" position="10,10,0">

<RefWidget id="10002"/>

<RefWidget id="10003"/>

<RefWidget id="10004"/>

<RefWidget id="10005"/>

<RefWidget id="10006"/>

<RefWidget id="10007"/>

<RefWidget id="10008"/>

<RefWidget id="10009"/>

<RefWidget id="10010"/>

</Widget>

…

<!-- Option 6: To select the difficulty of the game -->

<Widget id="10007" type="3" position="0,300,0">

<Attributes>

<Attribute id="Form" value="60013"/>

<Attribute id="BaseChr" value="32"/>

<Attribute id="AlphaOn" value="255"/>

<Attribute id="AlphaOff" value="255"/>

<Attribute id="PixelsAmong" value="0"/>

<Attribute id="Direction" value="HORIZONTALRIGHT"/>

<Attribute id="StringValues"

value="Classic,Miss Pacman,Mine,Hard,Training"/>

<Attribute id="Text" value="Classic"/>

</Attributes>

</Widget>

…

27: Defining in the menu new types of levels. PacManII example

The options are: Classic, Miss Pacman, Mine, Hard and Training. All aligned with the definition at main method level.

# Creating new scenes

## What the standard one does

Now it is time to create a new scene.

First, let’s look what the standard one does. The very basic scene in any Pacman Game is *PACMAN::BasicScene*. This class inherits from *PACMAN::StandardScene*.

The *PACMAN::BasicScene* define at construction level *QGAMES::ActionBlocks* to manage but the movement of the 4 classical monsters and also the fruit that appears from time to time in the place defined in the maze for that.

class BasicScene : public StandardScene

{

public:

BasicScene (int c,

const QGAMES::Maps& m,

const QGAMES::Scene::Connections& cn =

QGAMES::Scene::Connections (),

const QGAMES::SceneProperties& p = QGAMES::SceneProperties (),

const QGAMES::EntitiesPerLayer& ePL =

QGAMES::EntitiesPerLayer ());

virtual void initialize () override;

};

28: The class BasicScene

The implementation of both methods is:

// ---

PACMAN::BasicScene::BasicScene (int c, const QGAMES::Maps& m, const QGAMES::Scene::Connections& cn,

const QGAMES::SceneProperties& p, const QGAMES::EntitiesPerLayer& ePL)

: StandardScene (c, m, cn, p, ePL)

{

addActionBlock (new PACMAN::MonsterSceneActionBlock

(0, PACMAN::MonsterSceneActionBlock::Properties (true,

\_\_PACMAN\_BLINKYBASEENTITYID\_\_, PACMAN::Blinky::\_NUMBER, 200)), false);

addActionBlock (new PACMAN::MonsterSceneActionBlock

(1, PACMAN::MonsterSceneActionBlock::Properties (true,

\_\_PACMAN\_PINKYBASEENTITYID\_\_, PACMAN::Pinky::\_NUMBER, 200)), false);

addActionBlock (new PACMAN::MonsterSceneActionBlock

(2, PACMAN::MonsterSceneActionBlock::Properties (true,

\_\_PACMAN\_INKYBASEENTITYID\_\_, PACMAN::Inky::\_NUMBER, 200)), false);

addActionBlock (new PACMAN::MonsterSceneActionBlock

(3, PACMAN::MonsterSceneActionBlock::Properties (true,

\_\_PACMAN\_CLYDEBASEENTITYID\_\_, PACMAN::Clyde::\_NUMBER, 200)), false);

addActionBlock (new PACMAN::ThingSceneActionBlock (4,

PACMAN::ThingSceneActionBlock::Properties ()), false);

}

// ---

void PACMAN::BasicScene::initialize ()

{

PACMAN::Game\* g = dynamic\_cast <PACMAN::Game\*> (game ());

assert (g != nullptr);

std::vector <int> bS, bP, bE, pId;

std::vector <QGAMES::bdata> sD, spd;

for (auto i : g -> levelDefinition (g -> level ()).fruitConditions ())

{

bS.push\_back (i.bonusSymbolId ());

bP.push\_back (i.bonusPoints ());

bE.push\_back (i.numberBallsEatenToAppear ());

sD.push\_back (\_\_BD i.secondsBonusToDisappear ());

spd.push\_back (\_\_BD i.speed ());

pId.push\_back (i.positionId ());

}

dynamic\_cast <PACMAN::ThingSceneActionBlock\*> (actionBlock (4)) ->

setProperties (PACMAN::ThingSceneActionBlock::Properties

(\_\_PACMAN\_FRUITBASEENTITYID\_\_, bS, bP, bE, sD, spd, pId));

PACMAN::StandardScene::initialize ();

}

29: Implementation of the class BasicScene

That is very simple to understand.

As you can see the constructor creates the 5 action blocks (id-ed from 0 to 4) needed to manage each monster and the fruit, while the ***initialize()*** method adjust the behaviour of the fruit attending to the configuration data of the level.

That class inherits, as commented before, from *PACMAN::StandardScene*:

class StandardScene : public Scene

{

public:

StandardScene (int c, const QGAMES::Maps& m,

const QGAMES::Scene::Connections& cn = QGAMES::Scene::Connections (),

const QGAMES::SceneProperties& p = QGAMES::SceneProperties (),

const QGAMES::EntitiesPerLayer& ePL = QGAMES::EntitiesPerLayer ())

: Scene (c, m, cn, p, ePL),

\_pacman (nullptr),

\_numberMonstersEaten (0)

{ }

virtual void initialize () override;

virtual void updatePositions () override;

virtual void finalize () override;

virtual void processEvent (const QGAMES::Event& evnt) override;

private:

PacMan\* \_pacman;

// Implementation

int \_numberMonstersEaten;

};

30: The definition of the class StandardScene

And the implementation of that methods is:

// ---

void PACMAN::StandardScene::initialize ()

{

addCharacter (\_pacman = dynamic\_cast <PACMAN::PacMan\*> (game () ->

character (\_\_PACMAN\_PACMANBASEENTITYID\_\_)));

\_pacman -> setMap (activeMap ());

PACMAN::Scene::initialize ();

\_pacman -> setCurrentState (\_\_PACMAN\_PACMANSTATESTANDLOOKINGRIGHT\_\_);

\_pacman -> setVisible (true);

PACMAN::Map\* aM = dynamic\_cast <PACMAN::Map\*> (activeMap ());

assert (aM != nullptr);

\_pacman -> setPositionFromMazePosition

(aM -> pacmanInitialPosition (0 /\*\* There can't be more. \*/));

\_numberMonstersEaten = 0;

}

// ---

void PACMAN::StandardScene::updatePositions ()

{

PACMAN::Scene::updatePositions ();

// Regarding the status of the map...

if (\_percentageCleaned == \_\_BD 1)

notify (QGAMES::Event (\_\_PACMAN\_PACMANREACHEDGOAL\_\_, \_pacman));

// To chase or not...(depends on what monsters do)

if ((onOffSwitch (\_SWITCHMONSTERSBEINGTHREATEN) -> isOn () && !\_

pacman -> isChasing ()) ||

(!onOffSwitch (\_SWITCHMONSTERSBEINGTHREATEN) -> isOn () &&

\_pacman -> isChasing ()))

\_pacman -> toChase (onOffSwitch (\_SWITCHMONSTERSBEINGTHREATEN) -> isOn ());

if (!onOffSwitch (\_SWITCHMONSTERSBEINGTHREATEN) -> isOn ())

\_numberMonstersEaten = 0;

}

// ---

void PACMAN::StandardScene::finalize ()

{

PACMAN::Scene::finalize ();

\_pacman -> setMap (nullptr);

removeCharacter (\_pacman);

\_pacman = nullptr;

}

// ---

void PACMAN::StandardScene::processEvent (const QGAMES::Event& evnt)

{

if (world () == nullptr ||

(world () != nullptr && world () -> activeScene () != this))

return;

if (evnt.code () == \_\_PACMAN\_BALLEATEN\_\_)

{

if (static\_cast <PACMAN::TilePath\*> (evnt.data ()) -> hasPower ())

\_numberMonstersEaten = 0;

}

else

if (evnt.code () == \_\_PACMAN\_PACMANEATMONSTER\_\_)

{

PACMAN::Monster\* mtr = static\_cast <PACMAN::Monster\*> (evnt.data ());

freezeMonstersAndPacmanForAWhile (true);

int nPts = mtr -> points () << (\_numberMonstersEaten++);

showMessage (new PACMAN::Scene::MessageToShow

(std::to\_string (nPts), \_\_PACMAN\_NEWTOUR24YELLOWLETTERS\_\_,

\_pacman -> position (),

(int) (game () -> framesPerSecond () \* 1 /\*\* One second. \*/)));

\_pacman -> setScore (\_pacman -> score () + nPts);

if (\_numberMonstersEaten == numberMonsters ())

\_pacman -> setScore (\_pacman -> score () + 12000); // Additional!!

}

PACMAN::Scene::processEvent (evnt);

}

31: Implementation of the class StandardScene

Where:

***initialize()***: Sets pacman in the position defined in the map for so (position 0).

***updatePositions ()***: Controls whether there is no more balls to eat. In that case a notification is thrown. That notification Will be translated later into either a change in the level or the end of the game (in case this level was the last). Additionaly controls whether the pacman is in chase actitude.

***processEvent ()***: Control mainly whether pacman has eaten a monster. This is done to control the assignation of points. In a standard scene, the points finally assigned to pacman are not only the ones defined at Monster level (look at configuration chapter), but any sequential Monster eaten multiply that value by two with an additional of 12.000 points if all are eaten.

So, we need a new type of Scene only in the case any of those behaviours shown must bee changed.

## Defining a new type of scene

Imagine another we want to add another monster to the scene.

This is done, e.g, in *PacManII*. Some of the scenes includes a new monster called *Wormy* (we will be back later). That monster appears after some time moving in the maze. The class is defined as:

using namespace PacManII;

class BasicSceneWithAppearingMonsters final : public PACMAN::BasicScene

{

public:

BasicSceneWithAppearingMonsters

(int c, const QGAMES::Maps& m,

const QGAMES::Scene::Connections& cn = QGAMES::Scene::Connections (),

const QGAMES::SceneProperties& p = QGAMES::SceneProperties (),

const QGAMES::EntitiesPerLayer& ePL = QGAMES::EntitiesPerLayer ())

: PACMAN::BasicScene (c, m, cn, p, ePL),

\_additionalMonstersActionBlocks ()

{ }

virtual void initialize () override;

virtual void finalize () override;

private:

// Implementation

std::vector <QGAMES::SceneActionBlock\*> \_additionalMonstersActionBlocks;

};

32: A new class with a new monster. Example of PacManII

And its methods:

// ---

void PacManII::BasicSceneWithAppearingMonsters::initialize ()

{

PacManII::Game\* g = dynamic\_cast <PacManII::Game\*> (game ());

assert (g != nullptr);

const PacManII::DataGame::LevelDefinition\* nLDef =

dynamic\_cast <const PacManII::DataGame::LevelDefinition\*>

(&g -> levelDefinition (g -> level ()));

if (nLDef == nullptr ||

(nLDef != nullptr && nLDef -> additionalMonsters ().empty ()))

return;

\_additionalMonstersActionBlocks = { };

for (int i = 0;

i < (int) nLDef -> additionalMonsters ().size (); i++)

{

PacManII::DataGame::LevelDefinition::AdditionalMonster aM =

nLDef -> additionalMonsters ()[i];

PacManII::ElementToAppearSceneActionBlock\* mABlkCtrl =

new PacManII::ElementToAppearSceneActionBlock (100 + (i << 1),

PacManII::ElementToAppearSceneActionBlock::Properties

(aM.monsterId (), QGAMES::MazeModel::\_noPosition,

aM.numberBallsEatenToAppear (),

aM.maxSecondsToAppear (), 101 + (i << 1)));

PacManII::WormyMonsterSceneActionBlock\* mABlk =

new PacManII::WormyMonsterSceneActionBlock (101 + (i << 1),

PacManII::WormyMonsterSceneActionBlock::Properties

(PACMAN::MonsterSceneActionBlock::Properties

(false, aM.monsterId (),

aM.monsterNumber (), aM.points ()),

aM.trailLength ()));

\_additionalMonstersActionBlocks.push\_back (mABlk);

addActionBlock (mABlk, false); // Not initialized now...

\_additionalMonstersActionBlocks.push\_back (mABlkCtrl);

addActionBlock (mABlkCtrl, false); // Not initialized now...

}

PACMAN::BasicScene::initialize ();

}

// ---

void PacManII::BasicSceneWithAppearingMonsters::finalize ()

{

for (auto i : \_additionalMonstersActionBlocks)

removeActionBlock (i);

\_additionalMonstersActionBlocks = { };

PACMAN::BasicScene::finalize ();

}

33: Implementation of a new class with a new type of monster

Where two additional blocks to control each additional monster defined in the configuration data are added.

Let’s notice that the type of configuration data used in *PacManII* is also different that the standard one (*PacManII::DataGame::LevelDefinition* instead *PACMAN::DataGame::LevelDefinition*).

In this case to modify the builder is also need, as this new type of scene has to be taken into account:

using namespace PacManII;

class WorldBuilder final : public PACMAN::WorldBuilder

{

public:

WorldBuilder () = delete;

WorldBuilder (const std::string& eDef, QGAMES::MapBuilder\* mB)

: PACMAN::WorldBuilder (eDef, mB)

{ }

WorldBuilder (const WorldBuilder&) = delete;

WorldBuilder& operator = (const WorldBuilder&) = delete;

protected:

virtual QGAMES::Scene\* createSceneObject (int ns, const QGAMES::Maps& m,

const QGAMES::Scene::Connections& cn,

const QGAMES::SceneProperties& p,

const QGAMES::EntitiesPerLayer& ePL) override;

virtual QGAMES::SceneActionBlock\* createSceneActionBlockObject (int nAB,

const QGAMES::SceneActionBlockProperties& prps) override;

};

34: Redefining the world builder

And specifically, the method ***createSceneObject()*** is, now:

// ---

QGAMES::Scene\* PacManII::WorldBuilder::createSceneObject

(int ns, const QGAMES::Maps& m,

const QGAMES::Scene::Connections& cn, const QGAMES::SceneProperties& p,

const QGAMES::EntitiesPerLayer& ePL)

{

QGAMES::Scene\* result = nullptr;

if (ns == \_\_PACMANII\_BLOCKSCENES1\_\_ ||

ns == \_\_PACMANII\_BLOCKSCENES3MS1\_\_ ||

ns == \_\_PACMANII\_BLOCKSCENES3MS2\_\_ ||

ns == \_\_PACMANII\_BLOCKSCENES3MS3\_\_ ||

ns == \_\_PACMANII\_BLOCKSCENES3MS4\_\_)

result = new PacManII::BasicSceneWithAppearingMonsters (ns, m, cn, p, ePL);

else

if (ns == \_\_PACMANII\_BLOCKSCENES2MS1\_\_ ||

ns == \_\_PACMANII\_BLOCKSCENES2MS2\_\_ ||

ns == \_\_PACMANII\_BLOCKSCENES2MS3\_\_ ||

ns == \_\_PACMANII\_BLOCKSCENES2MS4\_\_)

result = new PACMAN::BasicScene (ns, m, cn, p, ePL);

else

result = PACMAN::WorldBuilder::createSceneObject (ns, m, cn, p, ePL);

return (result);

}

35: Implementing the method createSceneObject

Where the new type of scene is considering.

If a new world builder is needed, then a new game class is also necessary:

using namespace PacManII;

class Game : public PACMAN::Game

{

public:

Game (const PACMAN::DataGames& dt)

: PACMAN::Game (dt)

{ }

protected:

virtual QGAMES::EntityBuilder\* createEntityBuilder () override

{ return (new EntityBuilder

(parameter (\_\_GAME\_PROPERTYENTITIESFILE\_\_),

formBuilder (), movementBuilder ())); }

virtual QGAMES::MovementBuilder\* createMovementBuilder () override

{ return (new MovementBuilder

(parameter (\_\_GAME\_PROPERTYMOVEMENTSFILE\_\_))); }

virtual QGAMES::WorldBuilder\* createWorldBuilder () override

{ return (new WorldBuilder

(parameter (\_\_GAME\_PROPERTYWORLDSFILE\_\_),

mapBuilder ())); }

};

36: Redefining the class Gam

Where the last instruction is accountable to create the new World Builder.

But also, that new type of scene has to be reflected in the configuration. Let’s look to the worlds.xml file defined for PacManII:

<World id="60000">

<Scenes>

<Scene id="60000"> <!-- This type of scene adds Wormy -->

<Maps>

<Map id="50000" file="maps/normalmap\_blue.tmx"/>

<Map id="50100" file="maps/normalmap\_green.tmx"/>

<Map id="50200" file="maps/normalmap\_yellow.tmx"/>

<Map id="50300" file="maps/normalmap\_red.tmx"/>

</Maps>

<Entities>

…

</Entities>

</Scene>

</Scenes>

</World>

37: Redefining the fil worlds.xml

Now, the circle to include a new type of scene is complete.

# Creating new monsters

## Th standard monsters

With th framework is possible to create a new type of monster.

Blinky, Pinky, Inky y Clyde inherits from the class *PACMAN::StandardMonster*:

class StandardMonster : public Monster

{

public:

StandardMonster (int cId, int mN, const QGAMES::Forms& f = QGAMES::Forms (),

const QGAMES::Entity::Data& d = QGAMES::Entity::Data ())

: Monster (cId, mN, f, d)

{ }

virtual QGAMES::Vector preferredDirectionAtStartingToMove () const override

{ return (QGAMES::Vector (\_\_BD -1, \_\_BD 0, \_\_BD 0)); }

virtual void setReferenceArtists (const std::vector <Artist\*>& r) override;

virtual void updatePositions () override;

protected:

virtual QGAMES::MazeModel::PositionInMaze targetMazePosition () const override;

virtual void setStateToStandLookingTo (const QGAMES::Vector& d) override;

virtual void setStateToMoveTo (const QGAMES::Vector& d) override;

// Implementation

virtual void adaptSpeed () override;

};

And this other inherits also from *PACMAN::Monster*:

class Monster : public Artist

{

public:

// The different status a monster cab ne in...

enum class Status

{ \_NOTDEFINED = 0, \_ATHOME = 1, \_EXITINGHOME = 2,

\_CHASING = 3, \_RUNNINGWAY = 4, \_TOBEEATEN = 5, \_BEINGEATEN = 6 };

Monster (int cId, int mN, const QGAMES::Forms& f = QGAMES::Forms (),

const QGAMES::Entity::Data& d = QGAMES::Entity::Data ());

virtual bool isEnemy (const PacmanElement\* elmnt) const override;

int monsterNumber () const

{ return (\_monsterNumber); }

virtual QGAMES::Vector preferredDirectionAtStartingToMove () const = 0;

int points () const

{ return (\_points); }

void setPoints (int p)

{ \_points = p; }

bool isAtHome () const;

bool isInATunnelHall () const;

QGAMES::Vector directionToStartMovement () const;

virtual void toStand () override;

virtual void toMove (const QGAMES::Vector& d) override;

void toChase (bool f);

void toChaseDeferred (bool o);

void toBeThreaten (bool f);

void toBlink (bool b, int bp);

virtual bool hasElroyPossibility () const

{ return (false); }

virtual void setElroyCondition (int nC)

{ \_elroyCondition = -1; }

int elroyCondition () const

{ return (\_elroyCondition); }

virtual bool isAlive () const override

{ return (\_status != Status::\_BEINGEATEN); }

virtual bool isStanding () const override

{ return (\_status == Status::\_ATHOME); }

virtual bool isMoving () const override

{ return (!isStanding () && \_status != Status::\_NOTDEFINED); }

bool isDangerous () const

{ return (isAlive () &&

(\_status == Status::\_CHASING ||

\_status == Status::\_RUNNINGWAY ||

\_status == Status::\_EXITINGHOME)); }

bool isChasing () const

{ return (\_status == Status::\_CHASING); }

bool isRunningAway () const

{ return (\_status == Status::\_RUNNINGWAY); }

bool canReverseMovement () const;

Status status () const

{ return (\_status); }

virtual void initialize () override;

virtual void updatePositions () override;

virtual void finalize () override;

virtual void processEvent (const QGAMES::Event& evnt) override;

virtual void whenCollisionWith (QGAMES::Entity\* e) override;

protected:

\_\_DECLARECOUNTERS\_\_ (Counters);

virtual QGAMES::Counters\* createCounters () override

{ return (new Counters); }

\_\_DECLAREONOFFSWITCHES\_\_ (OnOffSwitches)

virtual QGAMES::OnOffSwitches\* createOnOffSwitches () override

{ return (new OnOffSwitches); }

void setStatus (const Status& st);

Status lastStatus () const

{ return (\_lastStatus); }

virtual void whatToDoWhenStopStatusIsRequested

(const QGAMES::Vector& d) override;

virtual void whatToDoWhenMovementStatusIsRequested

(const QGAMES::Vector& d) override;

virtual void whatToDoOnCurrentPosition () override

{ }

// Implementation

virtual void adaptSpeed () = 0;

virtual QGAMES::MazeModel::PathInMaze& recalculatePathInMaze

(const QGAMES::Vector& mD = QGAMES::Vector::\_noPoint) override;

virtual QGAMES::MazeModel::PositionInMaze runAwayMazePosition () const;

protected:

int \_monsterNumber;

int \_points;

Status \_status;

Status \_lastStatus;

int \_elroyCondition;

static const int \_COUNTERBLINKSITUATION = 0;

static const int \_SWITCHBLINKACTIVE = 0;

static const int \_SWITCHBLINKSITUATION = 1;

};

Any monster passes through 6 different states:

* **\_NOTDEFINED**: Almost never used.
* \_**ATHOME**: When the monster is at home. The home can be someplace in or out the monster’s jail. It is really the place where the monster starts its movement.
* **\_EXITINGHOME**: When it is in jail, this state is to reflect the movement between in and the entry of the jail.
* \_**CHASING**: When it is looking for pacman, or the element he has defined to pursuit.
* **\_RUNNINGWAY**: When it is running from pacman (or the element defined so) heading to the scatter place.
* \_**TOBEEATEN**: When it can be eaten by pacman.
* **\_BEINGEATEN**: When it has being eaten an it is going back to monster’s jail, to start back the movement.

How a monster passes through all those different states is managed by the class *PACMAN::MonsterSceneActionBlock*. So to move a monster in a scene a *MonsterSceneActionBlock* object within the scene has to be defined either.

The behavioural of any standard monster is contained mainly in a simple method defined in *PACMAN::StandardMonster* class:

virtual QGAMES::MazeModel::PositionInMaze targetMazePosition () const override;

This method determines where the target position maze of the monster depending on the status it is in.

e.g. for Blinky:

// ---

QGAMES::MazeModel::PositionInMaze PACMAN::Blinky::targetMazePosition () const

{

QGAMES::MazeModel::PositionInMaze result = QGAMES::MazeModel::\_noPosition;

switch (status ())

{

case PACMAN::Monster::Status::\_NOTDEFINED:

result = currentMazePosition ();

break;

case PACMAN::Monster::Status::\_CHASING:

if (\_referenceArtists [0] != nullptr)

{

if ((result = \_referenceArtists [0] ->

currentMazePosition ()) == currentMazePosition ())

result = nextXGridPosition (1);

}

else

result = runAwayMazePosition ();

break;

default:

result = PACMAN::StandardMonster::targetMazePosition ();

}

return (result);

}

Where you can notice that the target position when Blinky is in **\_CHASING** status is the same that the position of **\_referenceArtists [0]** (if defined), that is defined as pacman for a *PACMAN::StandardScene*. Notice also that if Blinky is already where **\_refereceArtists [0]** is, the next position in the maze is choosen instead. This is to avoid the monster to stop (it could happen e.g. when pacman can not be died).

The target position for the rest of states is common for all standard monsters, and it is defined in the *PACMAN::StandardMonster* class:

// ---

QGAMES::MazeModel::PositionInMaze PACMAN::StandardMonster::targetMazePosition () const

{

QGAMES::MazeModel::PositionInMaze result = QGAMES::MazeModel::\_noPosition;

if (pMap () == nullptr)

return (result);

switch (status ())

{

case PACMAN::Monster::Status::\_NOTDEFINED:

case PACMAN::Monster::Status::\_CHASING:

assert (false);

case PACMAN::Monster::Status::\_ATHOME:

result = currentMazePosition ();

break;

case PACMAN::Monster::Status::\_EXITINGHOME:

result = pMap () -> monsterExitingHomePosition ();

break;

case PACMAN::Monster::Status::\_RUNNINGWAY:

case PACMAN::Monster::Status::\_TOBEEATEN:

result = runAwayMazePosition ();

break;

case PACMAN::Monster::Status::\_BEINGEATEN:

result = pMap () -> monsterReturningPositionAfterDieing ();

break;

}

return (result);

}

When you can see that, e.g., the target position when it is **\_ATHOME** is the same place it is already in. The *MonsterAceneActionBlock* class will know that situation and will push the monster to the next status.

Try to link the name of the methods with the tiles used in the definition of a map (in the layer *ArtistLocations*).

The aspect of any monster is defined in configuration files. The file defining the standard monsters is: *./pcentities.xml*. .g the piece defining Blinky:

<!-- Blinky -->

<SetOfEntities fromId="50020" number="10">

<Forms>

<Form id="50020"/>

</Forms>

<MovementsBase>

<!-- Moving through the maze -->

<Movement id="550020"/>

<!-- At home -->

<Movement id="580020"/>

</MovementsBase>

<Animations>

<!-- Looking to the right -->

<Animation id="0" type="0">

<Attributes>

<Attribute id="FORM" value="50020"/>

<Attribute id="INITIALASPECT" value="0"/>

<Attribute id="FINALASPECT" value="2"/>

<Attribute id="FRACTIONTOCHANGE" value="6"/>

<Attribute id="LOOP" value="1"/>

<!-- Not linked with anyone -->

</Attributes>

</Animation>

<!-- Looking to the left -->

<Animation id="1" type="0">

<Attributes>

<Attribute id="FORM" value="50020"/>

<Attribute id="INITIALASPECT" value="28"/>

<Attribute id="FINALASPECT" value="30"/>

<Attribute id="FRACTIONTOCHANGE" value="6"/>

<Attribute id="LOOP" value="1"/>

<!-- Not linked with anyone -->

</Attributes>

</Animation>

<!-- Looking up -->

<Animation id="2" type="0">

<Attributes>

<Attribute id="FORM" value="50020"/>

<Attribute id="INITIALASPECT" value="42"/>

<Attribute id="FINALASPECT" value="44"/>

<Attribute id="FRACTIONTOCHANGE" value="6"/>

<Attribute id="LOOP" value="1"/>

<!-- Not linked with anyone -->

</Attributes>

</Animation>

<!-- Looking down -->

<Animation id="3" type="0">

<Attributes>

<Attribute id="FORM" value="50020"/>

<Attribute id="INITIALASPECT" value="14"/>

<Attribute id="FINALASPECT" value="16"/>

<Attribute id="FRACTIONTOCHANGE" value="6"/>

<Attribute id="LOOP" value="1"/>

<!-- Not linked with anyone -->

</Attributes>

</Animation>

<!-- Now can be eaten -->

<Animation id="4" type="0">

<Attributes>

<Attribute id="FORM" value="50020"/>

<Attribute id="INITIALASPECT" value="12"/>

<Attribute id="FINALASPECT" value="14"/>

<Attribute id="FRACTIONTOCHANGE" value="6"/>

<Attribute id="LOOP" value="1"/>

<!-- Not linked with anyone -->

</Attributes>

</Animation>

<!-- Going back home -->

<Animation id="5" type="0">

<Attributes>

<Attribute id="FORM" value="50020"/>

<Attribute id="INITIALASPECT" value="40"/>

<Attribute id="FINALASPECT" value="42"/>

<Attribute id="FRACTIONTOCHANGE" value="6"/>

<Attribute id="LOOP" value="1"/>

<!-- Not linked with anyone -->

</Attributes>

</Animation>

<!-- Now can be eaten, but in blink status -->

<Animation id="6" type="0">

<Attributes>

<Attribute id="FORM" value="50020"/>

<Attribute id="INITIALASPECT" value="26"/>

<Attribute id="FINALASPECT" value="28"/>

<Attribute id="FRACTIONTOCHANGE" value="6"/>

<Attribute id="LOOP" value="1"/>

<!-- Not linked with anyone -->

</Attributes>

</Animation>

</Animations>

<States>

<!-- (Starting/Chasing/Going Home/) Pacman while looking right -->

<State id="0" animation="0" movementBase="550020"/>

<!-- (Starting/Chasing/Going Home/) Pacman while looking left -->

<State id="1" animation="1" movementBase="550020"/>

<!-- (Starting/Chasing/Going Home/) Pacman while looking up -->

<State id="2" animation="2" movementBase="550020"/>

<!-- (Starting/Chasing/Going Home/) Pacman while looking down -->

<State id="3" animation="3" movementBase="550020"/>

<!-- Running away from Pacman (they can be eaten) -->

<State id="4" animation="4" movementBase="550020"/>

<!-- After being eaten going back home -->

<State id="5" animation="5" movementBase="550020"/>

<!-- Moving at home looking right -->

<State id="6" animation="0" movementBase="580020"/>

<!-- Moving at home looking left -->

<State id="7" animation="1" movementBase="580020"/>

<!-- Moving at home looking up -->

<State id="8" animation="2" movementBase="580020"/>

<!-- Moving at home looking down -->

<State id="9" animation="3" movementBase="580020"/>

<!-- Running away from Pacman (they can be eaten), but blink -->

<State id="10" animation="6" movementBase="550020"/>

</States>

</SetOfEntities>

With many references also to things defined in the file: ./pcmovements.xml and the file: ./pcforms.xml.

One important thing to highlight here is that there are up to 10 Blinkies. And the some could be said for the other classical monsters. So it would be even possible to set up and scene with 4 Blinkies chasing you! Imagine!

You see that the possibilities are almost infinite.

Simple modifications

A couple of simple modification could be possible without programming at all:

To change standard monster’s aspect.

Monster’s aspect is defined in the file: chompersprites.png:

Patrón de fondo

Descripción generada automáticamente

Replacing the different aspects in the same positions and maintaining the same size for everything, is the simplest modification. E.g. Could it be the way to proceed to create Pacman Junior?

To change the standard monsters:

To do so, it would be necessary to change this sprite sheet. But doing so, don’t forget in changing also ./pcforms.xml (the sizes of the new sprite could be different) and also ./pcentities.xml, to adequate the new form, aspects, locations, etc. to the animations needed.

But the two modifications possible alters the standard implementation of the game. So. It is better to everything using other sprite sheets and modifying no the standard configuration file, but including the new forms, aspects, entities, etc. in the files ./forms.xml, ./entities.xml, always before the include statement.

1. If you select to leave the *commonlibs* libraries are where they are, then you should set the variable PATH for the system to call them when running an application. [↑](#footnote-ref-1)
2. Remember that any action QGAMES game is consists of: different *worlds* (*QGAMES::World*). Each *world* is made up of one or many *scenes* (*QGAMES::Scene*) and each *scene* can has one or many maps where the action finally takes place. In the case of *PacMan*, the maps are tiled. [↑](#footnote-ref-2)