ARCADE

A RETRO PLATFORM

Un seul être vous manque et tout est dépeuplé
USEVMETED

by
lucas.dumy@epitech.eu
martin.januario@epitech.eu
sahel.lucas-saoudi@epitech.eu

Before playing:

While on the main menu, you can enter your name using your keyboard. If you hit a highscore while playing, your name will be recorded with your score and time, as long as you don't unplug the machine. You can choose the game to play using the UP and DOWN arrow keys, and validate your choice using the ENTER key. You can switch to another display mode using the keys SHIFT + '3' or SHIFT + '4' and you can exit the program using the ESCAPE key.

How to play:

The games begin paused, meaning you have to unpause them in order to start playing. The controls while playing are as follow:

- arrow keys to move
- ESCAPE to quit the program
- SPACE is the action key
- 'M' brings you back to the main menu
- 'R' allows you to restart the game
- 'P' switches the pause on and off
- SHIFT + '1' or '2' to try another game
- SHIFT + '3' or '4' to try another graphical library

This being said, here are the main informations concerning the two games originally present in USEVMETED Arcade program:

Nibbler:

Despite its name, this game follows the snake game rules. You control the snake standing in the middle of the screen and have to eat pellets in order to grow as much as possible. The default mode movements are snake-related, meaning you can use the RIGHT and LEFT arrow keys to move to the snake's relative right and left. If you want the arrow keys to refer to absolute direction, press the action key while playing (SPACE).

Avoid collisions with walls and try not to eat yourself, not everybody can be an Ouroboros. By the way, the developers could reach 12 200 points, filling the whole map, in 659 seconds. Can you do a better score?

Qix:

In this game, you impersonate a dress, symbol of the modern society. You have to stand against the nature and the natural order of the world and shape it according to your devious way of seeing the modern world. Get rid of the tall grass while avoiding their minions blue bears. Turn as much savage land as possible into beautiful flowers by creating path to claim areas to nature. Be careful not to enter the tall grass nor to meet the bears and prevent the tall grass monster from crossing the path you're creating, otherwise you'd regret it. Be fast, be smart, turn this desolate land into a domesticated meadow! Your score corresponds to 100 times the percentage of the map claimed. Will you be able to beat our developers and their awesome 9 800 points?

How to add a library:

The core of the program can handle new libraries, as long as they follow a specific pattern. Keep in mind that the communication through the core is really limited: from the game to the graphical library, you just send a game map stored as an int ** in an ar::Map and two std::map containing for the first a character which will be seen on the map and the corresponding sprite and for the second the same character and the corresponding color. The graphical library will display the game map received using the data from the std::map corresponding to its capacities. It will also catch the events received from the player and transfer them to the game using an ar::Event. Check the enumeration and structures present in Arcade.hpp, Map.cpp and Map.hpp to know better how the communication works.

How to add a game:

Keep in mind the only thing that will be transferred while the game has started is the game map, an ar::Map. You must inherit from the interface ar::IGame and override all of its methods:

void manageKey(const ar::Event &key)

→ receives an ar::Event corresponding to a key pressed by the user. In this method, the game has to react according to the event (example: go left if the LEFT arrow key is pressed). Must handle the arrow keys, pause, and action key if needed.

const std::map<unsigned char, ar::spriteCoords> &getSprites() const

→ returns a std::map containing as first value the index of a character which can be found in the game map and as second value a structure containing the coordinates of the corresponding sprite on the sprite sheet (starting X position, starting Y position, width and height). Called once for a graphical library handling sprites and stored by it.

const std::string getSpritePath() const

→ returns the path to the sprite sheet containing the game's sprites. Called once for a graphical library handling sprites and stored by it.

const std::map<unsigned char, ar::colorVector> &getColors() const

→ returns a std::map containing as first value the index of a character which can be found in the game map and as second value a structure containing the color to use for the corresponding character (red, green, blue). Called once for a graphical library which doesn't handle sprites and stored by it.

int refreshScore()

→ returns the current score of the game.

int refreshTimer()

→ returns the current time of the game.

bool isGameOver()

→ returns true if the game is over, false otherwise.

ar::Map &getMap()

→ returns the map containing the game map. The ar::Map class contains the actual map stored as an int **, its height and width which can be accessed through the use of getHeight() and getWidth() methods and the location of the player (optional).

void loop()

→ main method, constantly called by the core. Should contain a timer to know whether to update or not the game.

const std::string getGameName() const

→ returns the name of the game.

void setPause()

→ pauses the game, be it already paused or not.

Outside of your class:

extern “C” ar::IGame *createGame()

→ returns a new occurrence of your class.

extern “C” void destroyGame(ar::IGame *game)

→ deletes the game received as argument.

How to add a graphical library:

The graphical library receives an ar::Map to display, as long as the current score and time of the game and returns to the game the key pressed as an ar::Event. It should also handle its own menu.

ar::Event getEvent(int &realEvent)

→ gets the real event thanks to the library capacities and turns it into an ar::Event to send the game. Must handle enough events to handle the following cases: exit, validate, action key, restart, back to menu, pause, movements, change of game and graphical library.

bool canHandleSprites()

→ returns true if the library can handle sprites, false otherwise.

void displayGame(const ar::userInterface &UI, ar::Map &map)

→ displays the game: the game map is stored in the ar::Map class as an int **, the ar::userInterface contains the current username, the current score and the current time. Should use the std::map saved thanks to loadResources in order to know to which display corresponds each character in the game map.

void loadResources(const std::map<unsigned char, ar::colorVector>&colors)

void loadResources(const std::string &filepath,
                const std::map<unsigned char, ar::spriteCoords> &sprites)

→ gets an std::map containing as first value a character which may present in the game map and in second value how to render it: it is either its color if the game library can't handle sprites, or a sprite. If the library can handle sprite, the other argument contains the path to the corresponding sprite sheet.

void initMenu(const std::vector<std::string> &menuChoices,
                const std::string &menuName,
                const std::vector<std::string> &display)

→ Initializes a menu, gets as arguments the list of the games names, the name of the program and the graphical libraries names.

int refreshMenu(const ar::Event &key,
                const std::vector<ar::userInterface> &dataArray)

→ displays the menu set thanks to the initMenu method. Should handle only ar::Event::AR_UP, ar::Event::AR_DOWN, ar::Event::AR_VALIDATE as events, which are used to choose the game which will be launched. dataArray contains the highscore for each game, stored in a structured containing username, score and time. The menu should display it according to the game selected by the player. This method returns the index of the game currently selected, beginning with 0 for the first game.

void refreshUsername(std::string &name, int realKey)

→ updates the string received according to the realKey received, either by adding or removing a character. The string corresponds to the username.

void destroyMenu()

→ destroys all menu resources.

Outside of your class:

extern "C" ar::IDisplay *createDisplay()

→ returns a new occurrence of your class.

extern "C" void destroyDisplay(ar::IDisplay *display)

→ deletes the display received as argument.