TEKAR Documentation

Hi! In this documentation, you can learn to create your own graphical library and game library compatible with our game plateform **TEKAR**.

Create a graphical library

---

1. Getting started

> In order to make your graphical library compatible with our game platform, your graphical library's class **HAVE TO** inherite from our graphical interface:

```
#include <string>
#include "Common.hpp"

namespace arc {
    class IGraphic {
    public:
        virtual void setup() = 0;
        virtual void clear() = 0;
        virtual void display() = 0;
        virtual bool pollEvent() = 0;
        virtual bool isWindowOpen() const = 0;
        virtual int getKeyCode() const = 0;
        virtual void closeWindow() = 0;
        virtual void destroy() = 0;
        virtual int createTexture(const std::string &) = 0;
        virtual void drawBlock(const arc::Block &,
                    const arc::Vec2<size_t> &) = 0;
        virtual void drawText(const std::string &,
            const Vec2<float> &, const Vec2<size_t> &) = 0;
        virtual void setTextAttributes(size_t, size_t) = 0;
    };
```

---

2. Functions' explanations

void setup()

> **setup** is called when the application starts and when switching library.

In this function, you **must**:

> - Initialize your attributes
> - Create the application's window

For instance:

```
void arc::SFML::setup()
{
    _wd = new sf::RenderWindow(sf::VideoMode(width, height), "Ar
    if (!_font.loadFromFile("./fonts/arcade2.ttf"))
        throw arc::ArcadeError("Error when loading font");
}
```

---

void clear()

> **clear** is called in the application rendering loop

In this function, you **must**:

> - Clear the window's rendering

For instance:

```
void arc::Ncurses::clear()
{
    erase();
}
```

---

void display()

> **display** is called in the application rendering loop

In this function, you **must**:

> - Display on screen everthing that has been rendered

For instance:

```
void arc::SFML::display()
{
    _wd->display();
}
```

---

bool pollEvent()

> **pollEvent** is called in the application's main loop

For instance:

```
bool arc::SDL2::pollEvent()
{
    return (SDL_PollEvent(&_event));
}
```

---

bool isWindowOpen() const

> **isWindowOpen** must return **false** if the main window is closed

For instance:

```
bool arc::SFML::isWindowOpen() const
{
    return (_wd->isOpen());
}
```

---

int getKeyCode() const

> **getKeyCode** must return the key pressed at the moment.

> Keys listed in **arc::KeyCode** enumeration must be returned with the value of the enumeration. Other keys must be returned as their *ascii* value

---

void closeWindow()

> **Close the window**

---

void destroy()

> **destroy** must destroy all textures created with *createTexture*

---

int createTexture(const std::string &texture_folder_path)

> **createTexture** load the texture from the texture_folder_path

In this function, you **must**:

> - load the texture from the "*texture_folder_path*/**block.** *extension*"
> - store the texture in a *std::vector*
> - return the texture's index in that same *std::vector* (-1 in case error)

For instance:

```
int arc::SFML::createTexture(const std::string &path)
{
    std::string fpath = path + "/block.png";
    sf::Texture *texture = new sf::Texture;

    if (!texture->loadFromFile(fpath))
        return (-1);
    _blocks.push_back(texture);
    return (_blocks.size() - 1);
}
```

> The file *extension* is of your choice. But make sure that **texture_folder_path** contains that file.

---

void drawBlock(const arc::Block &block, const arc::Vec2 &map_sizes)

> **drawBlock** draw the arc::Block *block* to the screen

For instance:

```
void arc::SDL2::drawBlock(const arc::Block &b,
const arc::Vec2<size_t> &msize) const
{
    if (_blocks.size() <= (size_t)b.textureIdx
        || _blocks[b.textureIdx] == NULL)
        return;
    SDL_Rect rect;
    rect.x = ceil((float)b.pos.x * (float)arc::SDL2::width
        / (msize.x == 0 ? 1.0 : (float)msize.x));
    rect.y = ceil((float)b.pos.y * (float)arc::SDL2::height
        / (msize.y == 0 ? 1.0 : (float)msize.y));
    SDL_QueryTexture(_blocks[b.textureIdx], NULL, NULL, &(rect.w
    rect.w = ceil((float)arc::SDL2::width /
        (msize.x == 0 ? 1.0f : (float)msize.x));
    rect.h = ceil((float)arc::SDL2::height /
        (msize.y == 0 ? 1.0f : (float)msize.y));
    SDL_RenderCopy(_renderer, _blocks[b.textureIdx], NULL, &rect
}
```

> The block's sizes and positions must be changed according to the **map_sizes**

---

void setTextAttributes(size_t font_size, size_t color)

In this function, you **must**:

> - Set the font_size
> - Set the text's color (received as hexadimal format)

Remember:

> These properties will be applied to all the following text until the next call to this function.

For instance:

```
void arc::SDL2::setTextAttributes(size_t font_size, size_t hexco
{
    int r = (hexcolor >> 16) & 0xFF;
    int g = (hexcolor >> 8) & 0xFF;
    int b = (hexcolor) & 0xFF;
    _font_size = font_size;
    _color = {static_cast<Uint8>(r), static_cast<Uint8>(g),
        static_cast<Uint8>(b), 1};
}
```

---

void drawText(const std::string &text, const Vec2 &text_position, const Vec2 &map_sizes)

> Draw the *text* at *text_position* according to the *map_sizes*

For instance:

```
void arc::SDL2::drawText(const std::string &text,
const arc::Vec2<float> &pos, const arc::Vec2<size_t> &msize)
{
    SDL_Surface* s_text = TTF_RenderText_Solid(_font, text.c_str
    SDL_Texture* t_text = SDL_CreateTextureFromSurface(_renderer
    SDL_Rect rect;
    rect.x = pos.x * arc::SDL2::width / (msize.x == 0 ? 1 : msiz
    rect.y = pos.y * arc::SDL2::height / (msize.y == 0 ? 1 : msi
    TTF_SizeText(_font, text.c_str(), &(rect.w), &(rect.h));
    rect.w = rect.w * _font_size / 10.0;
    rect.h = rect.h * _font_size / 10.0;
    SDL_RenderCopy(_renderer, t_text, NULL, &rect);
}
```

KeyCode

```
    namspace arc {
        enum KeyCode {
            LEFT = 128,
            RIGHT,
            UP,
            DOWN,
            ESCAPE,
            ENTER,
```

```
            BACKSPACE
        };
    }
```

Block

```
namespace arc {
    class Block {
    public:
        Block(int textureIdxParam, const arc::Vec2<float> &posPa
            : textureIdx(textureIdxParam), pos(posParam)
            {}
        ~Block() = default;

        void operator=(int textureIdxParam) {
            textureIdx = textureIdxParam;
        }
        void operator=(const arc::Vec2<float> &posParam) {
            pos = posParam;
        }
        void operator=(const arc::Block &blockParam) {
            textureIdx = blockParam.textureIdx;
            pos = blockParam.pos;
        }
    public:
        int textureIdx;
        arc::Vec2<float> pos;
    };
}
```

Create a game library

---

1. Getting started

> In order to make your game library compatible with our game platform, your game library's class **HAVE TO** inherite from our game interface*:

```
#include <memory>
#include <string>
#include "Common.hpp"

namespace arc {
enum GameStatus {
    GAMEOVER,
    PLAYING
};

class IGame {
public:
        virtual ~IGame() = default;

        virtual std::shared_ptr<arc::IGame> getInstance() const

        virtual arc::GameStatus update(int) = 0;

        virtual const std::vector<arc::Vec2<size_t>> &getMapsSiz
        const = 0;

        virtual const arc::blocksLayers &getBlocksLayers() const
        virtual void setBlocks(const std::vector<int> &) = 0;
        virtual const std::vector<std::string> &getTexturesPaths
        const = 0;

        virtual void saveScore(const std::string &) const = 0;
        virtual const std::vector<std::string> &getScores() = 0;
        virtual const std::string &getCurrentScore() = 0;
    };
}
```

---

2. Functions' explanations

void update(int keyCode)

> **update** is called every game loop

In this function, you must:

> - Handle key pressed event
> - Update game's maps

---

const std::vector< arc::Vec2> &getMapsSizes()

> **getMapsSizes** return the maps' sizes (row, col)

---

const arc::blocksLayers &getBlocksLayers() const

> **getBlocksLayers** return an array of maps, each map is an list of arc::Block

---

const std::vector &getTexturesPaths()

> **getTexturesPaths** return an array of paths to the texture's folder

Example:
You want to create an apple's texture, put its .png and .txt (or .3d, ...) in a folder and name them **block**:

> /home/snake/textures/apple/block.png
> /home/snake/textures/apple/block.txt

Then your **getTexturesPaths** must return an array that contains "/home/snake/textures/apple"

---

void setBlocks(const std::vector &textures_idx)

> **setBlocks** is called right after getTexturesPaths, before the game start

In this function, you must

> Save **textures_idx** as an attributes for future uses.

---

const std::vector &getScores()

> **getScores** return all saved scores.

---

void saveScore(const std::string &player_name) const

> **saveScore** save the current score in a file.
