A.L.P-a.C.A

1.0.0

Generated by Doxygen 1.8.13

Contents

Chapter 1

About

This reference defines and describes the interfaces, concept, symbols and procedures making up the Arcade Library Plug-able Common API (henceforth referred to as A.L.P-a.C.A). A.L.P-a.C.A is intended, in the context of the academic `Arcade EPITECH project`, as a common API allowing the manipulation of various games and graphic libraries with the core program of this project. Any game library or graphic handler library following this API is therefore compatible with any core implementing it (save for platform incompatibilities). This API's concepts are dependent on the use of the `dl library` (or any system capable of faithfuly reproducing its behavior) in the implementation of the core.

Libraries

A.L.P-a.C.A libraires implementation should follow the A.L.P-a.C.A Libraries documentation.

Entity File Format

Game libraries entities shall be defined by the A.L.P-a.C.A Entity File Format.

Ressources Acquisition

Libraries ressources locations shall be defined by the RAFH

Doc completion

- [x] Libraries documentation
- [x] Entity File Format documentation
- [x] Ressources hierarchy

Chapter 2

Libraries

A.L.P-a.C.A defines two types of usable library, each with a handler class interface :

- `Game Libraries`
- `Graphic Libraries`

Game libraries must implement the `IGame` interface as handler type, while Graphic libraries must implement both the `IGraphic` and `IDisplayable` interfaces, with `IGraphic` as their handler type.

Library symbols

To be loadable by a core, both types of library must define a C-style function with the symbol `CreateHandler` taking no argument and returning a pointer to a dynamically constructed (aka `new` constructed) instance of their implementation of their respective handler type. The signature of the function for a library with a handler type interface `InterfaceType` implemented as `ImplementationType` should therefore be either :

```
{c++}
InterfaceType   *CreateHandler();
```

or

```
{c++}
ImplementationType  *CreateHandler();
```

To ensure symbol preservation against `c++` symbol mangling it is highly encouraged to place the definition of this factory function in an `extern "C"` clause such as

```
{c++}
extern "C" {
    InterfaceType   *CreateHandler()
    {
        return new ImplementationType;
    }
}
```

Game Handler

A game handler must be a complete implementation in a Game Library of the `IGame` interface, and provide **game logic for handling keyboard presses** through Keycode as `int32_t` type values, as well as **frame ticking game logic** and **status event game logic**, to create a playable game on the core program.

Graphic Handler

A Graphic handler must be a complete implementation in a Graphic Library of the `IGraphic` interface, and provide a display for the user on which to draw a game loaded in the core program through a Game handler. The nature of the display is **implementation defined** but must provide **Keyboard Input handling**, **Visual on-screen display**, and either **Text writing on display**, **Texture drawing on display** or both, through its implementation and the implementation of the `IDisplayable` interface associated with the Graphical library.

Displayable Entity

The entities used to draw the frames of the game on the handler's display must be instances of the Graphic library's implementation of the `IDisplayable` interface. This implementation must provide the ability for entities to have a list of several named states in which they can be placed. These states must be defined by the parsing of a file associated with the desired entity, under the A.L.P-a.C.A custom '.entity' file format. This file must be provided in the Graphical library's own ressource folder in the RAFH.

Chapter 3

Entity File Format

The A.L.P-a.C.A custom Entity File Format (.entity extension) describes a displayable entity for the purpose of the `Arcade EPITECH project`, all its possible states and how to display them depending on display's capabilities

White space & Comments

All lines **starting** with the caracter '#' in an entity file shall be ignored. All empty lines shall be ignored.

Sprite

The first usable line must contain either the path to a loadable sprite texture file or the string "undefined". This path must be relative from the execution folder of the core, and designate a file available in the appropriate RAFH location.

States

All following lines define all states of the graphic entity under the following format:

```
<state name>:<upleft coordinates>:<downright coordinates>:<color>:<back color>:<ASCII character>
```

Where:

- `<state name>` is the name of the state,
- `<upleft coordinates>` are the coordinates of the upleft included corner pixel of the rectangle in the texture file containing the image of the state, if the sprite texture is "undefined", this is to be ignored
- `<downright coordinates>` are the coordinates of the downright included corner pixel of the rectangle in the texture file containing the image of the state, if the sprite texture is "undefined", this is to be ignored
- `<color>` is the color of the character for ASCII display of the state
- `<back color>` is the color of the background for ASCII display of the state
- `<ASCII character>` is the character of the state for ASCII display

Coordinates

Coordinates in a `.entity` file should follow the format `[<x>,<y>]`

Where:

- `<x>` is the position in pixel, on the horizontal axis of the texture file, of the coordinate
- `<y>` is the position in pixel, on the vertical axis of the texture file, of the coordinate

Colors

Colors in a `.entity` file should follow the 32bit hexadecimal integer format with color order RRGGBBAA (ex for yellow : FFFF00FF)

Exemple

An entity file should roughly look like :

```
# Some comment
<path/to/sprite/texture.imageformat>

<1st state name>:[<x>,<y>]:[<x>,<y>]:<color>:<back color>:<ASCII character>
<2nd state name>:[<x>,<y>]:[<x>,<y>]:<color>:<back color>:<ASCII character>
<3rd state name>:[<x>,<y>]:[<x>,<y>]:<color>:<back color>:<ASCII character>
...
<Nth state name>:[<x>,<y>]:[<x>,<y>]:<color>:<back color>:<ASCII character>
```

Chapter 4

Ressource Acquisition File Hierarchy

To ensure compatibility of all libraries, ressources acquisition of Entity files, Spritesheet textures and others must follow the *same pattern*, which means all ressources lookups must follow a *common path structure hierarchy*. As such, A.L.P-a.C.A defines the following **Ressource Acquisition File Hierarchy**:

- <root>/
  - libs/
    - <lib>/
  - games/
    - <lib>/

Where:

- <root> is the folder from which the core is being executed
- <lib> is a folder named after an A.L.P-a.C.A compliant library, and contain that library's own ressource hierarchy

Each library's own ressource hierarchy's model is left to its own discretion.

Exemple

```
arcade/
    libs/
        somelib/
            fonts/
                someFont.ttf
    games/
        someGame/
            entities/
                anEntity.entity
                anotherEntity.entity
            textures/
                anEntitysTexture.png
                anotherEntitysTexture.png
```

Chapter 5

Class Index

5.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

Chapter 6

Class Documentation

6.1 IDisplayable Class Reference

Displayable entity interface.

`#include <IDisplayable.hpp>`

Public Member Functions

- virtual IDisplayable & operator++ ()=0

  Increment the Displayable entity's state.
- virtual IDisplayable & operator-- ()=0

  Decrement the Displayable entity's state.
- virtual void setState (const std::string &stateName)=0

  Puts the Displayable entity in the state associated with stateName.
- virtual void setState (std::size_t stateId)=0

  Put the Displayable entity in the stateId'th state.
- virtual const std::string & getState () const =0

  State name accessor.

6.1.1 Detailed Description

Displayable entity interface.

Defines an entity object that can be displayed. Such entity can have several states, those states are defined in a .entity file following the definition of the format given by the A.L.P-a.C.A reference. An implementation of this API is responsible for providing a corresponding construction mecanism.

6.1.2 Member Function Documentation

6.1.2.1 getState()

```
virtual const std::string& IDisplayable::getState ( ) const  [pure virtual]
```

State name accessor.

Access the current state and returns its name

Returns

Name of the current state

6.1.2.2 operator++()

```
virtual IDisplayable& IDisplayable::operator++ ( )  [pure virtual]
```

Increment the Displayable entity's state.

Puts the Displayable entity in its next state. States are ordered as declared in the corresponding .entity file. Wraps around to the first state if used at the end of the state list.

Returns

A reference to itself after state incrementation

6.1.2.3 operator--()

```
virtual IDisplayable& IDisplayable::operator-- ( )  [pure virtual]
```

Decrement the Displayable entity's state.

Puts the Displayable entity in its previous state. States are ordered as declared in the corresponding .entity file. Wraps around to the last state if used at the beggining of the state list.

Returns

A reference to itself after state decrementation

6.1.2.4 setState() [1/2]

```
virtual void IDisplayable::setState (
            const std::string & stateName )  [pure virtual]
```

Puts the Displayable entity in the state associated with stateName.

Parameters

stateName	Name of the desired state for the Displayable entity

6.1.2.5 setState() [2/2]

```
virtual void IDisplayable::setState (
            std::size_t stateId )  [pure virtual]
```

Put the Displayable entity in the stateId'th state.

States are ordered as declared in the corresponding .entity file.

Parameters

stateId	Index of the desired state for the Displayable entity

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

- IDisplayable.hpp

6.2 IGame Class Reference

Loadable Game handler interface.

`#include <IGame.hpp>`

Public Types

- enum KeyCode : int32_t {
  arrowUp = 0x415b1b, arrowDown = 0x425b1b, arrowRight = 0x435b1b, arrowLeft = 0x445b1b,
  home = 0x485b1b, end = 0x465b1b, pageUp = 0x355b1b, pageDown = 0x365b1b }
  Special Keycode values.

Public Member Functions

- virtual bool update (std::chrono::nanoseconds deltaT, std::chrono::seconds upTime)=0
  Runs 1 frame of the game.
- virtual void handleKey (int32_t key)=0
  React to the key pressed.
- virtual void setGraphic (IGraphic &handler)=0
  Gives a graphic handler to the game.
- virtual void onEnable ()=0
  Event called when the game is enabled.
- virtual void onDisable ()=0
  Event called when the game is disabled.

6.2.1 Detailed Description

Loadable Game handler interface.

Defines a game object that can be loaded and used for the purpose of the Arcade EPITECH project

6.2.2 Member Enumeration Documentation

6.2.2.1 KeyCode

```
enum IGame::KeyCode :  int32_t
```

Special Keycode values.

Enum type of keycodes for special keys. These keycodes are more than 1 char wide in data size. As these values are also int32_t this list is not exhaustive as a list of valid keycode for a game implementation keybinds. Valid keycodes also include ASCII values for the corresponding characters

Enumerator

arrowUp	Keycode for the Up arrow key.
arrowDown	Keycode for the Down arrow key.
arrowRight	Keycode for the Right arrow key.
arrowLeft	Keycode for the Left arrow key.
home	Keycode for the Home key.
end	Keycode for the End key.
pageUp	Keycode for the Page up key.
pageDown	Keycode for the Page down key.

6.2.3 Member Function Documentation

6.2.3.1 handleKey()

```
virtual void IGame::handleKey (
            int32_t key )  [pure virtual]
```

React to the key pressed.

Handle the reaction of the game to the input of a keycode

Parameters

key	Keycode of the pressed key

6.2.3.2 onDisable()

```
virtual void IGame::onDisable ( )  [pure virtual]
```

Event called when the game is disabled.

Called when the game is disabled by the core, such as when the core changes to another game or before deleting the game. Behavior is implementation defined

6.2.3.3 onEnable()

```
virtual void IGame::onEnable ( )  [pure virtual]
```

Event called when the game is enabled.

Called when the game is enabled by the core, such as when the core creates the game. Behavior is implementation defined

6.2.3.4 setGraphic()

```
virtual void IGame::setGraphic (
            IGraphic & handler )  [pure virtual]
```

Gives a graphic handler to the game.

Select handler as the new graphic handler for the game object. The graphic object referenced by handler is required to exist until another call to setGraphic is finished

Parameters

handler	Reference to a graphic handler object

6.2.3.5 update()

```
virtual bool IGame::update (
            std::chrono::nanoseconds deltaT,
            std::chrono::seconds upTime )  [pure virtual]
```

Runs 1 frame of the game.

An implementation of this API must put the general in loop game code in this function

Parameters

deltaT	is The duration between last frame and this one,
upTime	is The duration from the start of the game up to now

Returns

The status of the game, true if active, false if not and if the game object should be destroyed

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

- IGame.hpp

6.3 IGraphic Class Reference

Loadable Graphic handler interface.

`#include <IGraphic.hpp>`

Public Member Functions

- virtual void setEntity (float x, float y, IDisplayable &entity)=0

  Draws entity on the screen defined by the handler.
- virtual void write (int x, int y, const std::string &text)=0

  Writes text on the screen defined by the handler.
- virtual void setSize (int x, int y)=0

  Defines board size.
- virtual bool update ()=0

  Update the frame on the display.
- virtual void clear ()=0

  Clear the virtual board defined by the handler.
- virtual IDisplayable * createDisplayable (const std::string &path)=0

  Displayable entity factory.
- virtual bool hasInput ()=0

  Checks for input in the display of the graphic handler.
- virtual int32_t getInput ()=0

  Input getter.

6.3.1 Detailed Description

Loadable Graphic handler interface.

Defines a graphic handler object that can be loaded and used for the purpose of the Arcade EPITECH project. Implementation of this API is responsible for defining a display for the user, using a virtual board of cells whose origin is the upleft corner of the display, with the X axis increasing to the right and the Y axis increasing downward

6.3.2 Member Function Documentation

6.3.2.1 clear()

```
virtual void IGraphic::clear ( )  [pure virtual]
```

Clear the virtual board defined by the handler.

Erases every entities and texts drawn on the virtual board defined by the handler

6.3.2.2 createDisplayable()

```
virtual IDisplayable* IGraphic::createDisplayable (
            const std::string & path )  [pure virtual]
```

Displayable entity factory.

Creates a displayable entity that the handler can draw with setEntity form the .entity file given as argument

Parameters

path	Path to the .entity file to be used for entity construction

Returns

Constructed Displayable entity

Warning

The returned object needs to be deleted before the handlerthat gave it is destroyed as the destruction of said object will call the implementation side destructor which may not be defined anymore after handler destruction and result in undefined behavior

6.3.2.3 getInput()

```
virtual int32_t IGraphic::getInput ( )  [pure virtual]
```

Input getter.

Get the current input of the display defined by the graphic handler. As fetching method is implementation dependant, calling this method without a positive return of hasInput result in undefined behavior

Returns

The keycode of the current input in the display

6.3.2.4 hasInput()

```
virtual bool IGraphic::hasInput ( )  [pure virtual]
```

Checks for input in the display of the graphic handler.

Returns

true if some input is ready, false otherwise

6.3.2.5 setEntity()

```
virtual void IGraphic::setEntity (
            float x,
            float y,
            IDisplayable & entity )  [pure virtual]
```

Draws entity on the screen defined by the handler.

Draws an entity at a position defined by the given coordinates x & y Units of these coordinate correspond to the dimensions of a cell on the display of the handler

Parameters

x	Position on the X axis of where to draw the entity
y	Position on the Y axis of where to draw the entity
entity	Displayable entity to be drawn

6.3.2.6 setSize()

```
virtual void IGraphic::setSize (
            int x,
            int y )  [pure virtual]
```

Defines board size.

Set the virtual board of the handler to the size given in arguments, with units being cells of the board

Parameters

x	Width of the new virtual board
y	Height of the new virtual board

6.3.2.7 update()

```
virtual bool IGraphic::update ( )  [pure virtual]
```

Update the frame on the display.

Updating of the frame draw visually on the display for the user every entities & texts drawn on the board

Returns

The status of the display, true if its still open, false if a user or else closed it and the application should close

6.3.2.8 write()

```
virtual void IGraphic::write (
            int x,
            int y,
            const std::string & text )  [pure virtual]
```

Writes text on the screen defined by the handler.

Writes text at a position defined by the given coordinates x & y Units of these coordinate correspond to the dimensions of a cell on the display of the handler

Parameters

x	Position on the X axis of where to write the text
y	Position on the Y axis of where to write the text
text	Text to be written

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

- IGraphic.hpp

Index