{EPITECH}

ARCADE - DOC

A RETRO PLATFORM

Mallory **SCOTTON**
Hugo **CATHELAIN**
Nathan **FIEVET**

ARCADE

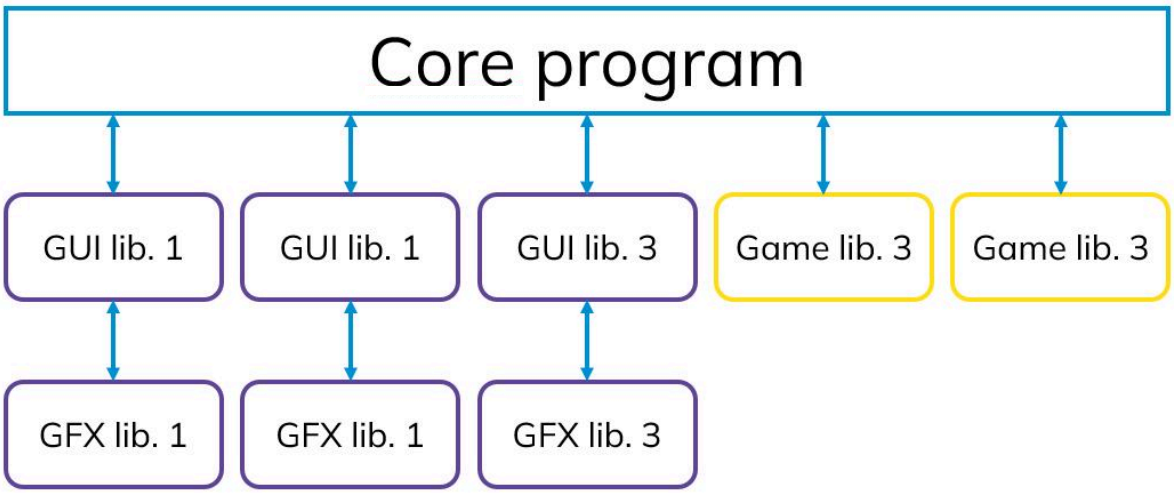
binary name: arcade
language: C++
compilation: via Makefile, including re, clean and fclean

✓ The totality of your source files, except all useless files (binary, temp files, objfiles, ...), must be included in your delivery
✓ All the bonus files (including a potential specific Makefile) should be in a directory named bonus
✓ Error messages have to be written on the error output, and the program should then exit with the 84 error code (0 if there is no error).

Arcade is a **gaming platform** : a program that lets the user choose a game to play and keeps a register of player scores.

To be able to deal with the elements of your gaming plateform at run-time, your graphics libraries and your games must be implemented as **dynamic libraries**, loaded at run-time. Each GUI available for the program must be used as a shared library that will be loaded and used dynamically by the main program. This also applies to your games.

It is **STRICTLY FORBIDDEN** to refer to a graphics library explicitly in your main program. Only your dynamic graphics libraries can do so.



Doc Summary

This documentation provides an overview of the Arcade project, detailing its structure, core, components, and functionality. It covers how the platform handles dynamic library loading, manages memory, and supports game loops, while also explaining how to extend the platform with new games and graphical backends.

Project Structure

This document provides an overview of the directory structure for the Arcade project, descsribing the purpose of each folder and files. The project follows a modular architecture, making it easy to extend with new graphics and game libraries

Root Directory

- `AUTHORS.md`: A file containing the list of authors who contributed to the project.
- `CONTRIBUTING.md`: A guide for developers interested in contributing to the project.
- `LICENSE.md`: Contains the licensing information for the project.
- `README.md`: The main README file providing an overview of the project, setup instructions, and general information.
- `Makefile`: The build system for the project, managing the compilation of the core program, graphics libraries, and game libraries.
- `build/`: Contains the build artifacts, including compiled object files for the Arcade core and libraries.
- `backends/`: Contains the graphical backends of the Arcade platform, implemented as dynamic libraries.
- `games/`: Contains the game libraries, implemented as dynamic libraries.
- `doc/`: Contains project-related documentation files.
- `lib/`: Contains the compiled shared object files (`.so`) for both the graphics libraries and game libraries.

Arcade Directory

- `Main.cpp`: The main entry point of the Arcade application. It initializes and runs the game selection and display the interface.
- `core/`: Contains the core logic of the Arcade platform, including the base classes for the game modules and graphics module.
  - `API.cpp/hpp`: The API used to interact with game and graphical libraries.
  - `Core.cpp/hpp`: The core functionality such as game loops and library management.
  - `Library.hpp/cpp`: The utility class that handle library loading.
- `enums/`: Contains enumerations used across the project.
  - `Inputs.hpp`: Defines input-related enums (e.g., for keyboard controls).
- `errors/`: Contains error-related logic.
  - `DLError.hpp`: A header that defines error handling related to dynamic library loading.
- `interfaces/`: Contains interface definitions for the game and graphical logic.
  - `IGameModule.hpp`: Interface for game modules that define game logic.
  - `IGraphicsModule.hpp`: Interface for graphics modules that manage inputs and rendering.

Enumerations

EKeyboardKey

The `EKeyboardKey` enumeration represents the set of possible keyboard keys that can be mapped for input in an application or game. This set includes all the alphanumeric characters, directional keys, special control keys, numeric keypad keys, and an “UNKNOWN” option for undefined or unrecognized keys.

- A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U, V, W, X, Y, Z: These represent the 26 letters of the English alphabet, each corresponding to a key on a standard keyboard.
- UP, DOWN, LEFT, RIGHT: These represent the arrow keys on a standard keyboard for navigation.
- SPACE, ENTER, ESCAPE, BACKSPACE: These represent common controls keys: spacebar, enter/return key, escape key and the backspace key.
- NUM0, NUM1, NUM2, NUM3, NUM4, NUM5, NUM6, NUM7, NUM8, NUM9: These represent the numeric keys on a standard keyboard, which can include both the top row numbers and the keypad numbers.
- UNKNOWN: This is a fallback value used to represent any key that cannot be classified into one of the above categories or if the key is undefined or unrecognized.

EMouseButton

The `EMouseButton` enumeration defines the three primary mouse buttons. It is used to represent user input from the mouse device during interactions.

- LEFT: Represents the left mouse button, typically used for selecting or clicking on elements.
- RIGHT: Represents the right mouse button, often used for opening context menus or secondary actions.
- MIDDLE: Represents the middle mouse button (or mouse wheel), often used for scrolling or other special input actions.

EChannel

The `EChannel` enumeration is used to categorize different types of event channels within an application or game. Event channels help manage and prioritize different types of events, such as those related to core application functionality, graphics rendering, or game mechanics.

- CORE: Represents events related to the core functionality of the application, such as system-level inputs, application lifecycle events, or configuration changes.
- GRAPHICS: Represents events related to graphics rendering, such as screen updates, visual effects, or scene changes.
- GAME: Represents events related to the gameplay mechanics, such as player input, game state changes, collisions, or game logic processing.

Program Core

Library Loading

The `Library` class is reponsible for dynamically loading and unloading game and graphics libraries within the Arcade platform. These libraries, implemented as shared objects , are loaded at runtime to ensure that the platform can easily support various game and graphical backends.

The `Load` and `Unload` methods in the `Library` class provide an interface for safely managing these dynamic libraries.

Loading a library

To load a library, the `Load` method is called with the path to the shared object file as an argument. This method uses the `dlopen()` function to load the shared object file into memory, with the `RTLD_LAZY` flag, which ensures that the symbols are resolved only when they are used.

If the `dlopen()` function fails to load the library, an exception (`DLError`) is thrown with the error message provided by `dlerror()`, which can be retrieved immediately after attempting to load the library.

Once the library is sucessfully loaded, the method proceeds to find the symbol `CreateArcadeObject` using `dlsym()`. This symbol is expected to point to a function that creates the appropriate object of type `T` (either a graphics or game module). If the symbol is not found, another `DLError` is thrown.

The object is then created by calling this function. If the creation fails, a `DLError` is thrown again. The object is wrapped in a `std::shared_ptr`, ensuring that the library handle is properly cleaned up when the object is no longer in use.

The `shared_ptr` ensures that the dynamically loaded library remains open for the duration of the object's lifecycle, and once the object is destroyed, the library handle is closed via `dlclose()`.

Unloading a library

The `Unload` method is a simpler process. It resets the `std::shared_ptr` to release the object and trigger its destructor. Since the destructor of the `shared_ptr` is responsible for closing the library handle, this operation ensures proper cleanup without requiring any explicit calls to `dlclose()` by the user.

The `reset()` function releases the ownership of the object, which triggers the cleanup code in the shared pointer's custom deleter, ensuring the library is properly unloaded.

Error Handling

Both the `Load` and `Unload` methods handle errors robustly. If any error occurs during the library loading or symbol resolution process, a `DLError` exception is thrown. This allows the calling code to handle such errors appropriately, usually by logging the error and gracefully exiting the application if needed.

The error handling ensures that the Arcade platform can recover from issues such as missing or incompatible libraries, and provides meaningful error messages to the user.

Summary

The `Library` class plays a crucial role in the Arcade platform by abstracting the complexity of dynamically loading and unloading game and graphics libraries. It provides a safe and efficient mechanism for extending the platform with new modules, while also handling errors gracefully to ensure a smooth user experience.

Game Lifecycles

In the Arcade platform, each game is represented by a module that adheres to the `IGameModule` interface. This interface manages the game's lifecycle through several key phases: starting the game, ticking the game logic, and ending the game. These phases are integral to how the game operates within the platform and how its state is managed during the gameplay loop.

The lifecycle is controlled through the following key methods defined in the `IGameModule` interface:

`BeginPlay()`

This method is called when a game is initialized and started. It marks the beginning of the game session and prepares the game for play. This could involve setting up initial game states, loading assets, or initializing gameplay variables. It is invoked once the user selects a game to play.

- **Purpose**: Initializes the game and prepares the environment for gameplay.
- **Usage**: This is typically where the game setup and any necessary pre-game configurations are handled.

`EndPlay()`

When a game ends, the `EndPlay()` method is called to clean up and finalize any game-related processes. This might involve saving the final score, resetting the game state, or preparing the game for reinitialization. It is essential for ensuring that any resources allocated during the game's lifetime are properly cleaned up before the game is finished.

- **Purpose**: Finalizes the game session and releases any resources used.
- **Usage**: Called when the game finishes, either because the player won, lost, or chose to exit the game.

`Tick(float deltaSeconds)`

The `Tick()` method is responsible for updating the game's logic each frame. The `deltaSeconds` parameter, which represents the time elapsed since the last tick, is used to ensure that the game logic runs consistently, independent of the frame rate. This method is called in a loop during gameplay, allowing the game to respond to user inputs, update entities, check game rules, and perform necessary calculations (such as movement or scoring).

- **Purpose**: Runs the core game logic during gameplay, updating objects and responding to events.
- **Usage**: It's called on each frame, passing the elapsed time to ensure smooth, time-based updates.

`IsGameOver() const`

This method checks whether the game is over, allowing the platform to know when to end the current session. The game might end if certain conditions are met, such as the player losing all their lives or completing the final level. It returns `true` if the game is over and `false` if it's still ongoing.

- **Purpose**: Determines if the game has finished, either through player defeat or victory.
- **Usage**: Frequently checked by the platform to decide whether to continue running the game or transition to a different state (such as showing the game over screen).

`GetScore() const`

This method retrieves the current score of the game. The score reflects the player's progress and performance throughout the game and is typically updated after each significant game action, such as scoring points or completing a task.

- **Purpose**: Retrieves the current score of the game.
- **Usage**: This value can be used by the platform to display the player's score and keep track of high scores.

`GetName() const`

The `GetName()` method returns the name of the game. This name is displayed in the Arcade platform's UI and helps identify the game to the player.

- **Purpose**: Provides the name of the game.
- **Usage**: Displayed in the UI to identify the currently playing game.

`GetSpriteSheet() const`

The `GetSpriteSheet()` method returns the path to the sprite sheet or assets required by the game. This typically includes images or characters used in the game's visual elements. The sprite sheet is loaded by the graphical backend and displayed during gameplay.

- **Purpose**: Provides the assets (sprite sheet) for rendering the game's graphics.
- **Usage**: Used by the graphical backend to render the game's visual components.

Summary

The game lifecycle in Arcade is managed through the `IGameModule` interface, which defines the core methods needed for starting, running, and ending a game session. These methods ensure that each game can properly initialize, update, and conclude its gameplay loop, while also providing essential information such as the game's score, name, and assets. The lifecycle is tightly integrated with the platform's event handling and rendering systems, ensuring a smooth and consistent user experience.

The API

The `API` class in the Arcade platform facilitates the handling of events and rendering of game assets.

Emit and receive events

The `API::Event` class encapsulates different types of events that can occur during the game's execution. These events are grouped under various channels, each representing different subsystems of the platform, such as the game logic, graphics, and core operations.

The event system allows for the decoupling of event generation and handling, allowing different modules of the system (such as the game, the graphics engine, and core framework) to communicate in a structured way.

Event Channels

The `API::Event` class categorizes events into three main channels using the enums `EChannel`:

- `GAME`: Events related to the gameplay, such as key presses or game over notifications.
- `GRAPHICS`: Events related to graphical changes, such as changing the resolution or the game's visual state.
- `CORE`: Core system events, such as application closure or grid resizing.

Event Types

The `API::Event` class supports a variety of event types, which encapsulate different kinds of actions or information. The event system is flexible, using C++'s `std::variant` to store any one of these event types. Here are the supported event structures:

- `KeyPressed`: Represents a key press event. It stores the key code (`EKeyboardKey`).

```
struct KeyPressed
{
    EKeyboardKey code;
};
```

- `MousePressed`: Represents a mouse press event, storing the mouse button pressed (`EMouseButton`) and the cursor position (`x`, `y`).

```
struct MousePressed
{
    EMouseButton button;
    int x;
    int y;
};
```

- `GridSize`: Represents a change in the grid's size, storing the new width and height of the grid.

```
struct GridSize
{
    int width;
    int height;
};
```

- `ChangeGraphics`: Represents a request to change the graphics settings, with a delta value that specifies the change amount.

```
struct ChangeGraphics
{
    int delta;
};
```

- `ChangeGame`: Represents a request to change the game or game state, with a delta value indicating the change amount.

```
struct ChangeGame
{
    int delta;
};
```

- `GameOver`: Represents a game over event, containing the final score of the game.

```
struct GameOver
{
    int score;
};
```

- `Closed`: Represents the event of the application closing.

```
struct Closed
{};
```

Each of these event types can be triggered and processed by the `API`, which allows the platform to react accordingly.

Event Handling

The `API::Event` class uses `std::variant` to store any one of the event types defined above. This ensures that an event can carry multiple potential data types while maintaining type safety. The event class provides several methods for interacting with the stored event:

- `Is`: Checks if the event is of a specific type.
- `GetIf`: Retrieves the event data if the event is of the specified type.
- `Visit`: Applies a visitor to the event, allowing different behaviors based on the event type.

Event Queue Management

The `API` class manages a queue of events for each channel, allowing events to be pushed and polled by different parts of the system. The following methods are used to handle events:

- `PollEvent`: Polls the next available event for a specified channel, returning an optional event if one exists.

```
while (auto event = API::PollEvent(API::Event::CORE)) {
    if (event->Is<API::Event::Closed>()) {
        // Do something when the close event is called
    } else if (auto key = event->GetIf<API::Event::KeyPressed>()) {
        // Do something when a key is pressed
    }
}
```

- `PushEvent`: Pushes an event to the event queue of a specified channel.

```
if (!Pacman.IsAlive()) {                        // Check if the player is dead
    API::PushEvent(
        API::Event::CORE,                       // Specify the event channel
        API::Event::GameOver{currentScore}      // Specify the event data
    );
}
```

Event Processing with Visitors

The `Visit` method, which utilizes the Visitor Design Pattern, enables the platform to handle events in a type-safe way. A visitor can be defined to process different types of events, which enhances flexibility and ensures that the event handling code remains clean and maintainable.

For example, a visitor can be used to process a key press event, mouse press event, or game over event, each with specific logic tailored to the event type.

Summary

The `API::Event` class provides a flexible and type-safe way of handling events in the Arcade platform. Events are categorized into channels, such as game, graphics, and core, and can be processed with various methods like `PollEvent` and `PushEvent`. The `API` class facilitates communication between different subsystems of the platform while maintaining modularity and extensibility. By using `std::variant` and `std::visit`, the event system can support a wide range of event types while keeping code maintainable and easy to extend.

Drawing elements

In the `API`, the drawing of elements such as game assets (sprites, characters, etc.) is managed via a dedicated drawing system. This allows the separation of game logic and rendering, providing an efficient way to handle the graphical output.

Asset Representation

The `Asset` class, defined within the `IGameModule`, represents an object that can be drawn. It contains properties like position, size, and the visual representation (characters) that can be rendered to the screen.

Drawing System

The `API` handles the drawing of elements using the following functions and data structures:

- **Queueing Drawables**: The `API` class uses a queue to manage drawable elements. Assets are added to this queue with specific positions.
- **Drawing**: The `Draw` function allows assets to be added to the draw queue with a specified position. This allows the system to render them when the time comes.
- **Processing Drawables**: The `PopDraw` function removes the next drawable element from the queue for processing and rendering.
- **Checking Draw Queue**: The `IsDrawQueueEmpty` function checks if the draw queue is empty, helping to manage whether there are pending drawing tasks.

Explanation of Drawing Functions

- `Draw` **Function**:
  - This function is used to add a drawable element (represented by an `Asset` object) to the draw queue.
  - It accepts an `Asset` (which contains the graphical data) and an (`x`, `y`) position, specifying where the asset should appear on the screen.
  - The asset is pushed into the `mDrawables` queue as a tuple containing the `Asset` and its position.
- `PopDraw` **Function**:
  - This function is used to retrieve and remove the next drawable element from the draw queue.
  - The function returns a tuple consisting of the `Asset`, `x` position, and `y` position, allowing the renderer to handle the asset drawing.
- `IsDrawQueueEmpty` **Function**:
  - This function checks if the draw queue is empty, indicating whether there are any pending drawing operations.
  - It returns `true` if the queue is empty and `false` otherwise, helping to manage whether the system should proceed with rendering.

How the Drawing System Works

- **Asset Queueing**: When the game logic determines that a specific asset needs to be drawn (e.g., a character or sprite), the `Draw` function is called. The asset and its position are queued for rendering.
- **Rendering Loop**: The rendering engine or system would periodically call `PopDraw()` to retrieve the assets to be drawn. This function provides both the asset and its position so that it can be properly placed on the screen.
- **Queue Management**: The `IsDrawQueueEmpty()` function is useful in ensuring that the game engine doesn't waste time processing a draw queue when no assets need to be drawn.

This system provides a decoupled way of handling the drawing process, which is especially useful for managing complex game states and large amounts of graphical data. By pushing and popping elements from a queue, the drawing system becomes highly efficient and flexible for various game scenarios.

This section of the API is crucial for handling all visual rendering logic in the system.

```
Drawing a sprite from a Game
```

```
API::Draw(
    IGameModule::Asset(
        {16, 8},                        // The sprite position in the spriteSheet
        "()",                           // The sprite ascii representation
        Color::Blue,                    // The sprite tint color
        {16, 16}                        // The sprite size (optional, default: 8x8)
    ),
    15, 5                               // The x and y position on the game grid
);
```

Adding a new Graphical Backend

To add a new graphical backend, you'll need to create a new module that implements the `IGraphicsModule` interface and integrates into the existing framework. Here's how you would go about it, with a step-by-step breakdown.

Steps to Add a New Graphical Backend

1. Create a New Folder for the Backend:
   a. Inside your `backends/` directory, create a folder for the new graphical library (e.g., `backends/MyGraphicalLibrary/`).
2. Implement the Backend Module:
   a. Implement the necessary functionality by creating a class that inherits from `Arc::IGraphicsModule`. You'll handle window creation, drawing, event handling, etc.
3. Define the Backend's Entry Point:
   a. You will need to provide a `CreateArcadeObject` function to create the graphical backend instance, which is used to initialize the backend.

Example Backend using "MyGraphicsLibrary"

Let's assume you're adding a backend using a hypothetical graphics library called `MyGraphicsLibrary`.

1. Creating the Backend's Loader File

In `backends/MyGraphicsLibrary/Loader.cpp`, we provide the necessary function to create the backend object.

backends/MyGraphicsLibrary/Loader.cpp

```cpp
#include "backends/MyGraphicsLibrary/MyGraphicsModule.hpp"
#include <memory>

extern "C"
{

std::unique_ptr<Arc::IGraphicsModule> CreateArcadeObject(void)
{
    return (std::make_unique<Arc::MyGraphicsModule>());
}

}
```


2. The Header for the Graphics Module

In `backends/MyGraphicsLibrary/MyGraphicsModule.hpp`, we define the module that implements the `IGraphicsModule` interface.

backends/MyGraphicsLibrary/MyGraphicsModule.hpp

```cpp
#pragma once

#include "Arcade/interfaces/IGraphicsModule.hpp"
#include <MyGraphicsLibrary/Graphics.hpp>  // Hypothetical graphics library
#include <memory>

namespace Arc
{

class MyGraphicsModule : public IGraphicsModule
{
private:
    std::unique_ptr<MyGraphicsLibrary::Window> mWindow;  //<! The window
    std::unique_ptr<MyGraphicsLibrary::Texture> mSpriteSheet;  //<! The sprite sheet
    float mRatio;  //<! Scaling ratio for rendering

public:
    MyGraphicsModule(void);
    ~MyGraphicsModule();

public:
    virtual void Update(void) override;
    virtual void Clear(void) override;
    virtual void Render(void) override;
    virtual void SetTitle(const std::string& title) override;
    virtual void LoadSpriteSheet(const std::string& path) override;
};

} // namespace Arc
```

3. Implementing the Graphics Module Logic

In `backends/MyGraphicsLibrary/MyGraphicsModule.cpp`, we implement the actual functionality.

backends/MyGraphicsLibrary/MyGraphicsModule.cpp

```cpp
#include "backends/MyGraphicsLibrary/MyGraphicsModule.hpp"
#include "Arcade/core/API.hpp"
#include <iostream>

namespace Arc
{

MyGraphicsModule::MyGraphicsModule(void)
    : mRatio(4.f)
{
    // Initialize the window using MyGraphicsLibrary
}

MyGraphicsModule::~MyGraphicsModule()
{
    // Close the window using MyGraphicsLibrary
}

void MyGraphicsModule::Update(void)
{
    // Poll events from Arcade API
    while (auto event = API::PollEvent(API::Event::GRAPHICS)) {
        if (auto gridSize = event->GetIf<API::Event::GridSize>()) {
            // Resize the window based on the grid size
        }
    }

    // Handle MyGraphicsLibrary Events to send
    // KeyPressed events to the CORE channel and MousePressed to the GAME channel
    // (hypothetical)
    API::PushEvent(API::Event::CORE, API::Event::KeyPressed{EKeyboardKey::F});
}

void MyGraphicsModule::Clear(void)
{
    // Clear the windows using MyGraphicsLibrary
}

...
```

backends/MyGraphicsLibrary/MyGraphicsModule.cpp

```
...

void MyGraphicsModule::Render(void)
{
    while (mSpriteSheet && !API::IsDrawQueueEmpty()) {
        auto draw = API::PopDraw();
        auto [asset, x, y, tintColor] = draw;

        // Render the sprite
using MyGraphicsLibrary
    }
    // Display the window
using MyGraphicsLibrary
}

void MyGraphicsModule::SetTitle(const std::string& title)
{
    // Set the title of the window using MygraphicsLibrary
}


void MyGraphicsModule::LoadSpriteSheet(const std::string& path)
{
    // Load the spritesheet texture using MygraphicsLibrary
}

} // namespace Arc
```

4. Linking the Backend to the Main API

You should now be able to integrate this backend with the main `API` framework.

- The `CreateArcadeObject` function allows the `API` to dynamically load and use this new backend, just like the SFML one.
- Ensure that the backend is correctly initialized and unloaded during the program's lifecycle.

Conclusion

To add a new graphical backend like `MyGraphicsLibrary`, you essentially need to:

- Implement a graphics module class that inherits from `IGraphicsModule`.
- Handle window creation, sprite loading, input handling, and rendering within that class.
- Provide a `CreateArcadeObject` function to instantiate the backend.
- Integrate it with the `API` class to handle events, drawing, and rendering.

This modular approach allows for easy addition of different graphical libraries into your framework, letting you support various platforms or graphics APIs.

Currently Working Implementations

- `SFML`
- `NCURSES`
- `SDL2`
- `VULKAN`