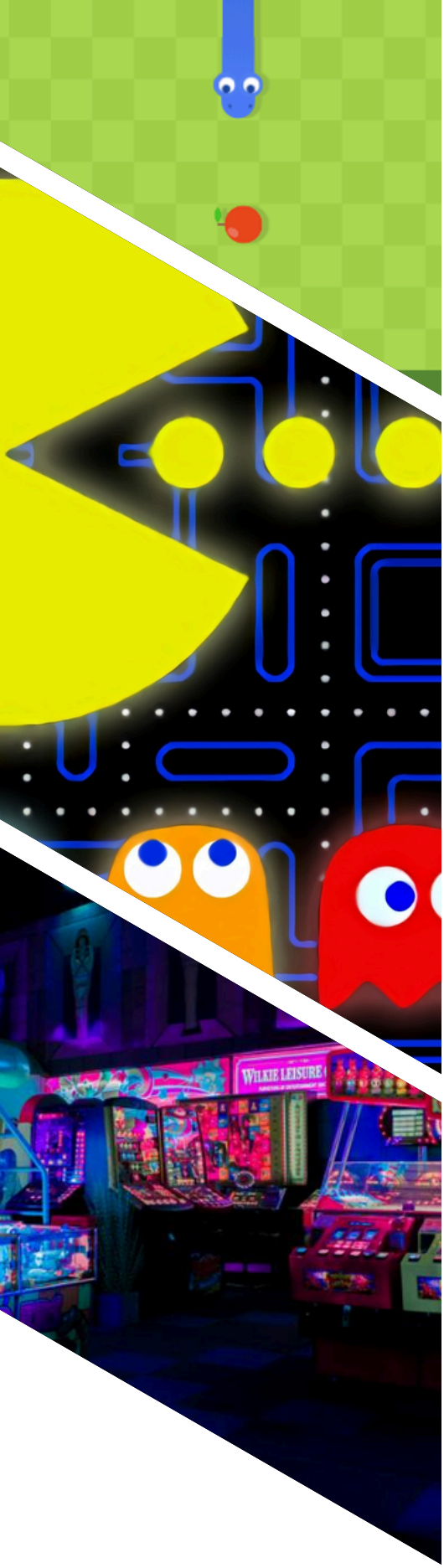


ARCADE ARCHITECTURE DOCUMENTATION

2025

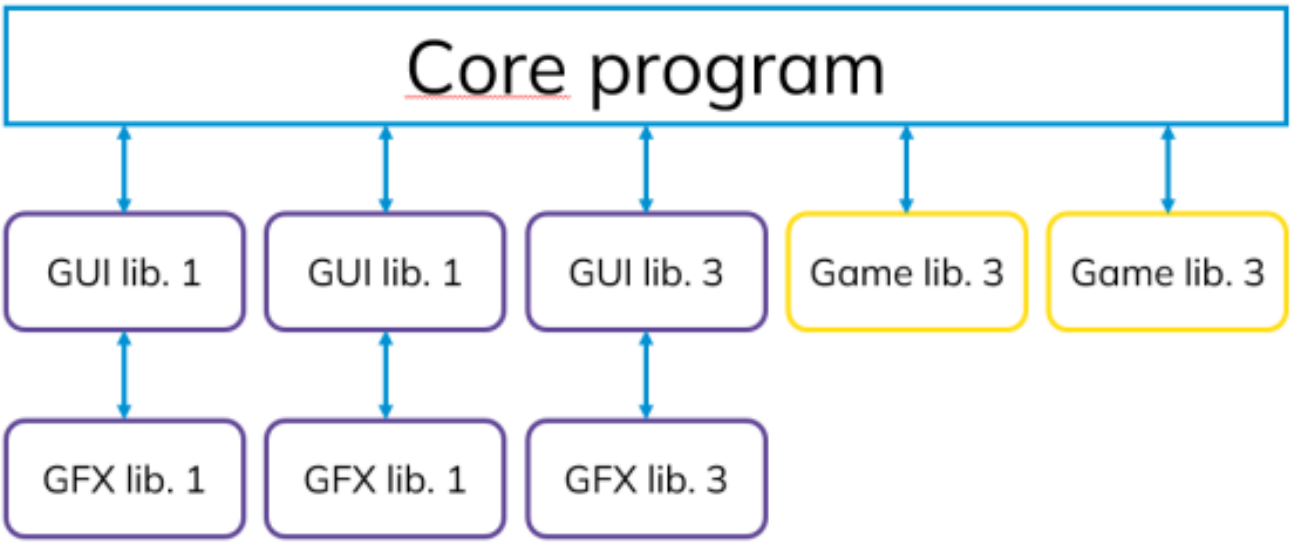
SUMMARY

01.
INTRODUCTION

Welcome to the Arcade project documentation. This guide will walk you through the process of implementing new games and graphic libraries using our interface system.
The Arcade project is designed around a flexible architecture that utilizes shared libraries for both games and graphic rendering. This modular approach allows developers to create new content without modifying the core application.
This documentation will cover:

- How to implement new game modules
- How to implement new graphic library modules

Whether you're adding a classic arcade game or implementing support for a new graphic library, following these guidelines will ensure your contribution integrates seamlessly with the existing Arcade ecosystem.



02.
IMPLEMENT A GAME

Game Module Entry Points

To implement a new game for the Arcade project, you must create a shared library with two specific entry points:

```
extern "C" {
    IGameModule *createGame(void);
    Loader::ModuleType_t getType(void);
}
```

Understanding the Entry Points

- createGame()
  - This function returns an instance of your game class (which must implement the *IGameModule* interface)
  - The core application calls this function to instantiate your game
  - Example: *return new SakeGame();* *where* *SnakeGame* *is your implementation*

- getType()
  - This function identifies the type of module to the core application
  - For games, it should always return *Loader::ModuleType_t::GAME_MODULE*
  - *This helps the core application categorize and manage different types of modules*

Implementation Example

Here's how a typical game module implementation would look:

```
#include "YourGame.hpp"

extern "C" {
    IGameModule *createGame(void) {
        return new YourGame();
    }

    Loader::ModuleType_t getType(void) {
        return Loader::ModuleType_t::GAME_MODULE;
    }
}
```

Export these functions with *extern "C"* to prevent name mangling

When the Arcade core loads your library, it will call these functions to instantiate your game and verify it's a game module.

The IGameModule Interface

Overview

The *IGameModule* interface defines the contract that all game implementations must follow. This interface manages the game's state, entities, scoring, and other essential elements.

Interface Definition

```
typedefstd::vector<std::vector<std::vector<std::shared_ptr<IEntity>>>> grid_t;

typedef enum gameState_e {
    PLAYING,
    WIN,
    LOSE
} gameState_t;

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

        virtual std::size_t getHighScore() const = 0;
        virtual void setHighScore(std::size_t highScore) = 0;

        virtual std::size_t getScore() const = 0;
        virtual void setScore(std::size_t score) = 0;

        virtual grid_t getEntities() const = 0;
        virtual void setEntities(grid_t entities) = 0;

        virtual std::pair<size_t, size_t> getGridSize() const = 0;

        virtual bool getIsStarted() const = 0;
        virtual void setIsStarted(bool isStarted) = 0;

        virtual gameState_t getGameState() const = 0;
        virtual void setGameState(gameState_t gameState) = 0;

        virtual std::vector<std::shared_ptr<IEntity>> getHUD() const = 0;

        virtual size_t getTime() const = 0;
        virtual void setTime(size_t time) = 0;

        virtual void update(std::shared_ptr<IGameModule> gameModule) = 0;

        virtual void changeDifficulty() = 0;
};
```

Method Descriptions

Score Management

- getHighScore / setHighScore
  - Manage the highest score achieved in the game
  - Useful for displaying and saving high scores

- getScore() / setScore()
  - Manage the current game score
  - Should be updated as the player progresses

Entity Management

- getEntities() / setEntities()
  - *getEntities()* returns a 3-dimensional grid of entities (y, x, z)
  - Each position can contain multiple entities at different layers (z-index)
  - Represents the current state of all game objects
  - setEntities() allows setting the entire grid at once

- getGridSize()
  - Returns a pair representing the dimensions of the game grid (width, height)
  - Used by display modules to properly render the game area

Game State

- getIsStarted() / setIsStarted()
  - Manage whether the game has started or not
  - Useful for implementing start HUD and manage events

- getGameState() / setGameState()
  - Indicate whether the game is in progress, won, or lost
  - The game use it to manage events properly

UI and Timing

- getHUD()
  - Returns a vector of entities to be displayed as part of the Heads-Up Display
  - These entities are rendered separately from the main game grid
  - Used for score display, lives, timers, etc.

- getTime() / setTime()
  - *getTime()* returns the elapsed time since the game started
  - *setTime()* sets the game time (for resets or loaded games)

Game Logic

- update()
  - Update is used only for changing the gamestate when needed (time restriction for example)
  - Called by the core application on each frame
  - Receives a shared pointer to itself, allowing for self-reference

- changeDifficulty()
  - Changes the current difficulty level of the game (If possible)
  - Is called when the *NEXTDIFFICULTY* event is triggered

Implementation Recommendations

When implementing the *IGameModule* interface :

- Entity Management :
  - Store your game entities in a 3D grid structure
  - Use the z-dimension for layering (background, game, etc)

- Game State :
  - Set appropriate win/lose conditions to ensure playability

- HUD Elements :
  - Create informative HUD elements for player feedback
  - Include score, time, lives, and other relevant information

- Grid Size :
  - Determine an appropriate grid size for your game
  - Consider the resolution and complexity of your game

The IEntity Interface

Overview

The *IEntity* interface defines the contract for all game entities within the Arcade system. Entities represent any interactive or displayable object within a game, from players and enemies to walls and UI elements.

Interface Definition

```
typedef enum clickType_e {
    LEFT_CLICK,
    RIGHT_CLICK,
    MIDDLE_CLICK
} clickType_t;

class IEntity {
    public:
        ~IEntity() = default;

        virtual void onClick(std::shared_ptr<IGameModule> gameModule,
                             clickType_t type) = 0;
        virtual void moveEntity(std::shared_ptr<IGameModule> gameModule) = 0;
        virtual void moveEntity(std::shared_ptr<IGameModule> gameModule,
                                std::pair<int, int> direction) = 0;
        virtual void onInteract(std::shared_ptr<IGameModule> gameModule) = 0;

        virtual std::pair<size_t, size_t> getPosition() const = 0;
        virtual void setPosition(std::pair<size_t, size_t> position) = 0;
        virtual std::string getSpriteName() const = 0;
        virtual std::size_t getColor() const = 0;
        virtual std::string getText() const = 0;
        virtual bool isMovable() const = 0;
        virtual bool isControlable() const = 0;
        virtual bool hasCollisions() const = 0;
};
```

Method Descriptions

Interaction Methods

- onClick()
  - Called when the entity is clicked by the user
  - Receives the game module and the type of click (left, right, or middle)
  - Implement custom click behavior or leave empty if not needed

- moveEntity() (without direction)
  - Called on entities that are movable but not directly controlled by the player
  - Used for implementing AI movement or automatic movement patterns
  - Receives the game module to allow access to game state and entities

- moveEntity() (with direction)
  - Called on entities that are both movable and controllable
  - Implements movement logic based on the provided direction
  - Direction is represented as a pair of integers (x, y) indicating movement vector

- onInteract()
  - Called when two entities should interact with each other
  - Handles collision responses, pickup effects, or other entity interactions
  - Receives the game module to allow modifying game state and entities

Position and Appearance

- getPosition() / setPosition()
  - Manage the entity's position on the game grid
  - Position is represented as a pair (x, y)

- getSpriteName()
  - Returns the path to the asset that should be used to render this entity
  - Used by graphic libraries that support sprites or images

- getColor()
  - Returns an integer corresponding to a specific color :
    - 0 → Black
    - 1 → White
    - 2 → Red
    - 3 → Green
    - 4 → Blue
  - Used by graphic libraries for colored rendering

- getText()
  - Returns the character or text representation of the entity
  - Used by text-based or simple graphic libraries

Entity Properties

- isMovable()
  - Returns whether the entity can be moved
  - Used to determine if movement logic should be called by the core

- isControlable()
  - Returns wether the entity can be controlled by the player
  - Determines which *moveEntity()* method should be called

- hasCollisions()
  - Returns whether the entity has collision properties
  - Indicates if other entities can pass through this entity or not

Implementation Recommendations

When implementing the *IEntity* interface :

- Base Entity Class :
  - Create a base entity class that implements common functionality
  - Derive specific entity types from this base class

- Movement Logic :
  - For controllable entities, implement robust movement handling in the direction-based *moveEntity()* method
  - For AI entities, implement behavior patterns in the non-directional *moveEntity()* method

- Interaction Handling :
  - Use *onInteract()* to handle entity-to-entity interactions

- Representation :
  - Provide meaningful values for both *getSpriteName()* and *getText()* to ensure compatibility with all graphic libraries
  - Choose appropriate colors for entities to ensure visibility

- Property Consistency :
  - Ensure that the behavior of your entity matches its reported properties

Example Entity Types

- Player Character :
  - isMovable → true
  - isControlable → true
  - hasCollisions → true

- Wall/Obstacle :
  - isMovable → true
  - isControlable → false
  - hasCollisions → true

03.
IMPLEMENT A GRAPHIC DISPLAY

Graphic Module Entry Points

To implement a new graphical display for the Arcade project, you must create a shared library with two specific entry points:

```
extern "C" {
    IWindow *createInstance(void);
    Loader::ModuleType_t getType(void);
}
```

Understanding the Entry Points

- createInstance()
  - This function returns an instance of your graphical window class (which must implement the *IWindow* interface).
  - The core application calls this function to instantiate the graphical display.
  - Example: return new *SFMLWindow();* where SFMLWindow is your implementation.

- getType()
  - This function identifies the type of module to the core application.
  - For graphics, it should always return *Loader::ModuleType_t::GRAPHICAL_MODULE*.
  - This helps the core application categorize and manage different types of modules.

Implementation Example

Here's how a typical graphical module implementation would look:

```
#include "YourWindow.hpp"

extern "C" {
    IWindow *createInstance(void) {
        return new YourWindow();
    }

    Loader::ModuleType_t getType(void) {
        return Loader::ModuleType_t::GRAPHICAL_MODULE;
    }
}
```

Export these functions with *extern "C"* to prevent name mangling.

When the Arcade core loads your library, it will call these functions to instantiate your graphical module and verify its type.

The IWindow Interface

Overview

The *IWindow* interface defines the contract that all graphical display implementations must follow. This interface manages the graphical rendering.

Interface Definition

```
struct color_t {
    int r;
    int g;
    int b;
};

class IWindow {
    public:
        virtual ~IWindow() = default;
        virtual void display() = 0;
        virtual void closeWindow() = 0;
        virtual bool isOpen() = 0;
        virtual void clear() = 0;

        virtual void setMapSize(std::pair<size_t, size_t> size) = 0;
        virtual void resizeWindow(size_t x, size_t y) = 0;

        virtual void drawSprite(std::string asset, int color, std::string text,
            std::pair<size_t, size_t> position) = 0;
        virtual void drawText(std::string text, int color, std::pair<size_t,
            size_t> position) = 0;
        virtual void drawRectangle(int color, std::pair<size_t, size_t>
            position) = 0;
        virtual void drawRectangleMenu(std::pair<size_t, size_t> size,
            std::pair<size_t, size_t> position, color_t color) = 0;
        virtual void drawTextMenu(std::string text, std::pair<size_t, size_t>
            position, color_t color, int charSize) = 0;
        virtual void drawThickRectangle(std::pair<int, int> position,
            std::pair<int, int> size, int thickness) = 0;
        virtual void drawSpriteMenu(std::pair<float, float> size, std::string
            asset, std::pair<int, int> position) = 0;

        virtual bool isMouseOver(std::pair<size_t, size_t> position,
            std::pair<size_t, size_t> size) = 0;
        virtual std::pair<int, int> getWindowSize() = 0;
};
```

Window Management

- display()
  - Renders the current frame onto the screen.
  - Should be called at the end of each frame to update the display.

- closeWindow()
  - Closes the graphical window.
  - Ensures proper cleanup of graphical resources.

- isOpen()
  - Returns whether the window is currently open.
  - Useful for determining when the application should stop rendering.

- clear()
  - Clears the screen before rendering the next frame.
  - Helps avoid graphical artifacts between frames.

Window Size Management

- setMapSize()
  - Sets the logical size of the game map.
  - This is used to scale rendering appropriately.

- resizeWindow()
  - Changes the physical dimensions of the window.
  - Used to adjust resolution or fullscreen settings.

- getWindowSize()
  - Returns the current dimensions of the window.
  - Useful for handling dynamic UI scaling.

Rendering

- drawSprite()
  - Draws a sprite at the given position.
  - asset: The path to the sprite file.
  - color: The color modifier applied to the sprite.
  - text: Optional text overlay on the sprite.
  - position: The grid position of the sprite.
  - If the library can’t draw a sprite it will draw a rectangle instead

- drawText()
  - Renders text on the screen at a specific position.
  - text: The string to be displayed.
  - color: The color of the text.
  - position: The screen coordinates for text placement.

- drawRectangle()
  - Draws a filled rectangle at a given position.
  - color: The fill color of the rectangle.
  - position: The screen coordinates for the rectangle.

Menu Rendering

- drawRectangleMenu()
  - Draws a colored rectangle for menu UI.
  - size: The dimensions of the rectangle.
  - position: The top-left position of the rectangle.
  - color: The fill color of the rectangle.

- drawTextMenu()
  - Renders menu text at a given position.
  - text: The string to be displayed.
  - position: The position of the text in the menu.
  - color: The color of the text.
  - charSize: The font size of the text.

- drawThickRectangle()
  - Draws a rectangle with a specified border thickness.
  - position: The top-left position of the rectangle.
  - size: The width and height of the rectangle.
  - thickness: The border thickness of the rectangle.

- drawSpriteMenu()
  - Draws a sprite specifically for menu UI.
  - size: The dimensions of the sprite.
  - asset: The path to the sprite file.
  - position: The screen position of the sprite.

Interaction

- isMouseOver()
  - Checks if the mouse cursor is hovering over a UI element.
  - position: The top-left position of the UI element.
  - size: The width and height of the UI element.
  - Returns true if the mouse is over the element, false otherwise.
  - If the Library can’t handle it, returns false

Implementation Recommendations

- Window Management
  - Ensure the window remains responsive to system events.
  - Properly handle closing events to avoid crashes or memory leaks.

- UI and Menu Handling
  - Implement distinct styling for menu elements to differentiate them from in-game objects.
  - Use drawTextMenu and drawRectangleMenu for clear, structured UI design.
  - Ensure proper scaling of UI elements across different screen sizes.

The IEvent Interface

Overview

The IEvent interface handles user input and translates events into actions that the core application can process.

Interface Definition

```
class IEvent {
     public:
         typedef enum event_e {
             UP,
             DOWN,
             LEFT,
             RIGHT,
             SPACE,
             ENTER,
             ESCAPE,
             CLOSE,
             NEXTGAME,
             NEXTGRAPHIC,
             REFRESH,
             MOUSECLICK,
             MOUSERIGHTCLICK,
             MOUSELEFTCLICK,
             MENU,
             NOTHING,
             TYPING,
             NEXTDIFFICULTY,
         } event_t;
         virtual ~IEvent() = default;
         virtual void init() = 0;
         virtual event_t pollEvents(std::pair<int, int> gridSize) = 0;
         virtual void cleanup() = 0;
         virtual std::pair<int, int> getMousePos() = 0;
         virtual void setMapSize(std::pair<int, int> size) = 0;

         virtual std::string getUsername() = 0;
         virtual void renderWrittiing() = 0;
};
```

By implementing these interfaces, you ensure that your graphical module is compatible with the Arcade framework. The Arcade core will interact with your graphical module through these interfaces, allowing seamless integration with different rendering backends such as SFML, SDL2, or ncurses.

Method Descriptions

Initialization & Cleanup

- init()
  - Initializes the event system.
  - Must be called before handling user inputs.
  - Sets up internal state for event polling.

- cleanup()
  - Cleans up resources allocated for event handling.
  - Should be called when the event system is no longer needed.

Event Handling

- pollEvents()
  - Polls for user input and returns the corresponding event type.
  - gridSize: The size of the game grid (used for scaling mouse interactions).
  - Returns an event_t value corresponding to the detected user action.

Mouse Input

- getMousePos()
  - Retrieves the current position of the mouse cursor.
  - If map size == {0, 0} → returns the real window coords
  - Else → return the grid ratio coords

Map Size Management

- setMapSize()
  - Sets the dimensions of the game map.
  - Useful for ensuring that input events are correctly mapped to the game's grid.

Text Input Management

- getUsername()
  - Retrieves the username entered by the player.

- renderWrittiing()
  - Renders the text input on screen.
  - This function is used to display the username or other text input in real time.

Implementation Recommendations

- Ensure that all key and mouse events are properly captured and processed.
- Implement pollEvents efficiently to avoid lag or unresponsiveness.
- Store and update the mouse position continuously for smooth interaction.
- Support text input dynamically for features like high-score name entry.
- Optimize event polling for performance, especially in real-time games.