{EPITECH}

ARCADE - DOC

A RETRO PLATFORM

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

ARCADE

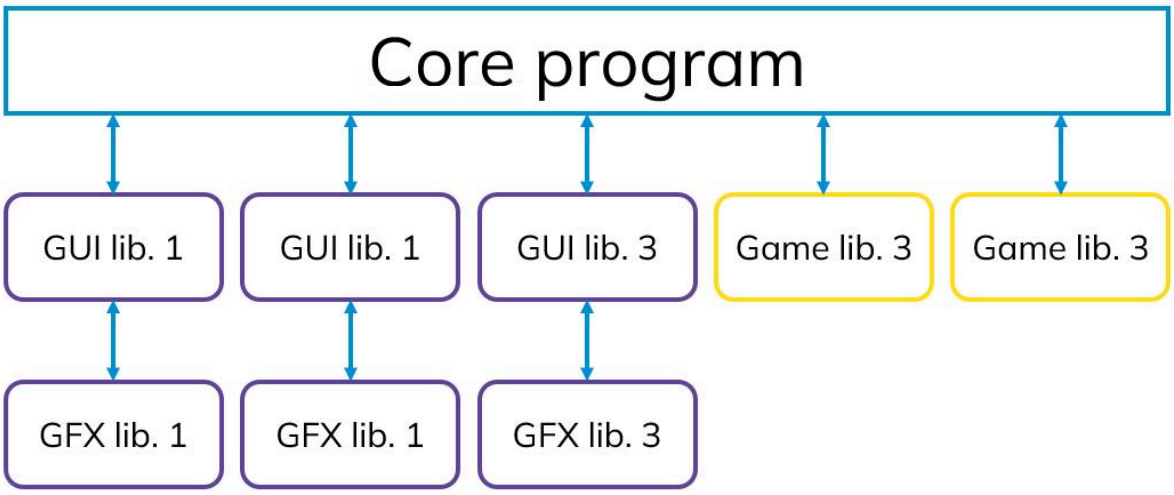
binary name: arcade
language: C++
compilation: via Makefile, including re, clean and fclean

✓ 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>());
}

std::string GetGraphicsName(void)
{
    return ("MyGraphicsLibrary");
}

}
```

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

```
#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;
    virtual std::string GetName(void) const 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
            // You need to calculate the mRatio using the minimum of the desktop
            // width and height minus the quarter of this 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
    }
    // Display the window
}

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`
- `OPENGL`

Adding a new Game

To add a new game, you'll need to create a new module that implements the `IGameModule` 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 Game

1. Create a New Folder for the Game:
   a. Inside your `games/` directory, create a folder for the new game library (e.g., `games/MyGame/`).
2. Implement the Game Module:
   a. Implement the necessary functionality by creating a class that inherits from `Arc::IGameModule`. You'll handle game begin play, end play, event handling, update tick, etc.
3. Define the Game's Entry Point:
   a. You will need to provide a `CreateArcadeObject` function to create the game instance, which is used to initialize the game.

Example Backend using "MyGame"

Let's assume you're adding a backend using a hypothetical game called `MyGame`.

1. Creating the Game's Loader File

In `games/MyGame/Loader.cpp`, we provide the necessary function to create the game object.

games/MyGame/Loader.cpp

```cpp
#include "games/MyGame/MyGame.hpp"
#include <memory>

extern "C"
{

std::unique_ptr<Arc::IGameModule> CreateArcadeObject(void)
{
    return (std::make_unique<Arc::MyGame>());
}

std::string GetGameName(void)
{
    return ("MyGame");
}

}
```

2. The Header for the Game Module

In `games/MyGame/MyGame.hpp`, we define the module that implements the `IGameModule` interface.

games/MyGame/MyGame.hpp

```cpp
#pragma once

#include "Arcade/interfaces/IGameModule.hpp"
#include <memory>

namespace Arc
{

class MyGame : public IGameModule
{
private:
    int mScore; //<! We need to store the current player score for the game
    int mLives; //<! We need to store the current lives of the player

public:
    MyGame(void);
    ~MyGame();

public:
    virtual void BeginPlay(void) override;
    virtual void EndPlay(void) override;
    virtual void Tick(float deltaSeconds) override;
    virtual bool IsGameOver(void) const override;
    virtual int GetScore(void) const override;
    virtual std::string GetName(void) const override;
    virtual std::string GetDescription(void) const override;
    virtual std::string GetSpriteSheet(void) const override;
};

} // namespace Arc
```

3. Implementing the Game Module Logic

In `games/MyGame/MyGame.cpp`, we implement the actual functionality.

games/MyGame/MyGame.cpp

```cpp
#include "games/MyGame/MyGame.hpp"
#include "Arcade/core/API.hpp"
#include <iostream>

namespace Arc
{

MyGame::MyGame(void)
    : mScore(0)
    , mLives(4)
{
    // Initialize the everything you need for the game
}

MyGame::~MyGame()
{
    // Clear everything you've created in the game
}

void MyGame::BeginPlay(void)
{
    // Set the Grid size for the game, for example Pacman is using a 28x31 game grid
    API::PushEvent(API::Event::GRAPHICALS, API::Event::GridSize{28, 31});
}

void MyGame::EndPlay(void)
{
    // Close and reset every behavior of the game
}

bool MyGame::IsGameOver(void) const
{
    // Implement a real is Game Over logic
    // But you can also use the GameOver event that contain the score
    return (false);
}

...
```

games/MyGame/MyGame.cpp

```
...

void MyGraphicsModule::Tick(float deltaSeconds)
{
    // Poll events from Arcade API
    while (auto event = API::PollEvent(API::Event::GAME)) {
        if (auto key = event->GetIf<API::Event::KeyPressed>()) {
            // Do something with the key pressed event
        } else if (auto mouse = event->GetIf<API::Event::MousePressed>()) {
            // Do something with the mouse pressed event
        }
    }

    // Hypotheticaly draw elements on the screen
    API::Draw(
        IGameModule::Asset({0, 1}, "()", Color(255, 0, 0)),
        Vec2i(14, 15)
    );
}

std::string MyGame::GetSpriteSheet(void)
{
    // Return the location of the sprite sheet
    return ("assets/MyGame/sprites.png");
}

} // namespace Arc
```

4. Linking the Game to the Main API

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

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

Conclusion

To add a new game like `MyGame`, you essentially need to:

- Implement a game module class that inherits from `IGameModule`.
- Handle game creation, sprite drawing, input handling, and gamel ogic within that class.
- Provide a `CreateArcadeObject` function to instantiate the game.
- Integrate it with the `API` class to handle events, drawing, and rendering.

This modular approach allows for easy addition of different game into your framework, letting you support various games.

Currently Working Implementations

- `PACMAN (1980 Midway)`
- `SNAKE`
- `NIBBLER`

Coding Conventions

The following coding conventions apply to all code written for the Arcade library. These conventions help maintain consistency and readability across the codebase and make it easier for developers to collaborate.

Header Guards

- Use `#pragma once` for header guards in all header files.
- This prevents the contents of the header file from being included multiple times in the same translation unit.

Dependencies

- Group and order the `#include` directives logically:
  - First, include the relevant project headers (e.g., `Arcade/utils/Color.hpp`).
  - Then, include standard library headers (e.g., `<string>`, `<vector>`, etc.).
- Each header include is separated by a comment block categorizing the inclusion

Namespace Usage

- The primary namespace for the library is `Arc`.
- Nested namespaces for specific game modules should follow the format `Arc::<GameModuleName>`.

Class and Method Documentation

- Each class and method should have brief comments describing its purpose and functionality.
- The format for documentation should include a description of the function's behavior and parameters/returns where applicable.

Indentation and Formatting

- Indentation should be 4 spaces.
- Each function should be preceded by a comment block that describes the purpose of the function and any special considerations.
- Function bodies should be enclosed in braces, and the opening brace should be on the same line as the function signature.

Member Variables

- Member variables should follow a consistent naming convention, typically using camelCase (e.g., `mGameState`, `mInGame`).
- Member variables should be declared in the `private` section of the class, unless access to them is required outside of the class.

Member Initialization

- Use the member initializer list in constructors to initialize member variables.
- Prefer default member initializers where possible for clarity and simplicity.

Constants and Magic Numbers

- Avoid hardcoding "magic numbers" directly into the code. Instead, define constants with meaningful names for clarity.
- For example, when defining screen dimensions or asset sizes, these should be defined as constants.

Function Signatures

- Function names should be descriptive and use CamelCase.
- Always specify the return type and parameter types explicitly.
- For functions that don't modify the class state, use `const` for return types where applicable.

Error Handling

- For functions where failure is possible, ensure proper handling of errors or return values to indicate failure.
- Use `assert` or appropriate logging for debugging and catching errors in development mode.

Memory Management

- Prefer the use of smart pointers (`std::unique_ptr` or `std::shared_ptr`) for automatic memory management.
- Manual memory management (e.g., using `new` and `delete`) should be avoided unless absolutely necessary.

Logic and Control Flow

- Use clear and concise conditionals to avoid deep nesting.
- Prefer early returns for functions where possible to reduce complexity.

File and Folder Structure

- Organize game modules and utilities in separate folders to maintain clarity and separation of concerns.
- Follow a naming convention for files based on the class or module they represent. For example, `Core.hpp` for the `Core` class, `Game.hpp` for the `Game` class, etc.

Return Values

- Always ensure that functions with return values return meaningful results. If a function cannot provide a meaningful value, consider returning a default value or an error code.

Game Menu

The Game Menu is a crucial part of the Arcade library that provides a user interface for selecting games, navigating through them, and launching them. This menu handles interactions, displays game options, and allows players to start or close games. It interacts closely with the libraries that define the games and graphical modules.

Library Discovery

When the `GetLibraries()` method is called, the system scans the `lib/` directory for `.so` files. The discovered game and graphical libraries are then made available to the `MenuGUI`. These libraries are displayed in the menu for the user to select.

- Game Libraries: These libraries implement specific games that can be launched from the menu.
- Graphical Libraries: These libraries implement different graphical backends for rendering the games.

The `SPRITE_MAP` and `SPRITES` provide visual representations of these libraries, allowing the user to see which libraries are available and navigate through them.

Game Poster

The Game Poster feature in the menu provides a visual representation of the selected game. Each game in the library has an associated poster (sprite), which is displayed in the menu screen.

- Posters are retrieved from the `SPRITES` vector.
- The `SPRITE_MAP` helps map game names to specific sprite assets, allowing the correct poster to be displayed for each selected game.

The game posters are pre-defined for a set of supported games. These are stored as assets in the `SPRITES` vector in the `Assets.hpp` file. The poster for each game is represented as a sprite that can be drawn on the screen. Here is the list of supported games and their corresponding pre-defined posters:

- `Tron`
- `Nibbler`
- `Pacman`
- `Centipede`
- `Solarfox`
- `Minesweeper`
- `Galaga`
- `Space Invaders`
- `Sabotage`
- `Street Fighter II`
- `Mortal Kombat`

Pacman (1980 Midway)

The Basis

The premise of Pac-Man is delightfully simple: using a four-way joystick, the player guides Pac-Man—up, down, left, and right—through a maze filled with dots for him to gobble up. Four ghost monsters are also in the maze and chase after our hero, trying to capture and kill him. The goal is to clear the maze of dots while avoiding the deadly ghosts. Each round starts with the ghosts in the "monster pen" at the center of the maze, emerging from it to join in the chase. If Pac-Man is captured by a ghost, a life is lost, the ghosts are returned to their pen, and a new Pac-Man is placed at the starting position before play continues. When the maze is cleared of all dots, the board is reset, and a new round begins. If Pac-Man gets caught by a ghost when he has no extra lives, the game is over.

There are 244 dots in the maze, and Pac-Man must eat them all in order to proceed to the next round. The 240 small dots are worth ten points each, and the four large, flashing dots—best known as energizers—are worth 50 points each. This yields a total of 2,600 points for clearing the maze of dots each round. Players have two ways to increase their score beyond what is earned from eating dots:

The first way to increase your score each round is by turning the tables on your enemies by making them your prey. Whenever Pac-Man eats one of the four energizer dots located in the corners of the maze, the ghosts reverse their direction and, in early levels, turn the same shade of blue for a short period of time before returning to normal. While blue, they are vulnerable to Pac-Man and can be gobbled up for extra points providing they are caught before the time expires. After being eaten, a ghost's eyes will return to the monster pen where it is resurrected, exiting to chase Pac-Man once again. The first ghost captured after an energizer has been eaten is always worth 200 points. Each additional ghost captured from the same energizer will then be worth twice as many points as the one before it—400, 800, and 1,600 points, respectively. If all four ghosts are captured at all four energizers, an additional 12,000 points can be earned on these earlier levels. This should not prove too terribly difficult to achieve for the first few rounds as the ghosts initially remain blue for several seconds. Soon after, however, the ghosts' "blue time" will get reduced to one or two seconds at the most, making it much more problematic to capture all four before time runs out on these boards. By level 19, the ghosts stop turning blue altogether and can no longer be eaten for additional points.

The second way to increase your score each round is by eating the bonus symbols (commonly known as fruit) that appear directly below the monster pen twice each round for additional points. The first bonus fruit appears after 70 dots have been cleared from the maze; the second one appears after 170 dots are cleared. Each fruit is worth anywhere from 100 to 5,000 points, depending on what level the player is currently on. Whenever a fruit appears, the amount of time it stays on the screen before disappearing is always between nine and ten seconds. The exact duration (i.e., 9.3333 seconds, 10.0 seconds, 9.75 seconds, etc.) is variable and does not become predictable with the use of patterns. In other words, executing the same pattern on the same level twice is no guarantee for how long the bonus fruit will stay onscreen each time. This usually goes unnoticed given that the majority of patterns are designed to eat the bonus fruit as quickly as possible after it has been triggered to appear. The symbols used for the last six rounds completed, plus the current round are also shown along the bottom edge of the screen (often called the fruit counter or level counter).

Modus Operandi

Ghosts have three mutually-exclusive modes of behavior they can be in during play: chase, scatter, and frightened. Each mode has a different objective/goal to be carried out:

- `CHASE` - A ghost's objective in chase mode is to find and capture Pac-Man by hunting him down through the maze. Each ghost exhibits unique behavior when chasing Pac-Man, giving them their different personalities: Blinky (red) is very aggressive and hard to shake once he gets behind you, Pinky (pink) tends to get in front of you and cut you off, Inky (light blue) is the least predictable of the bunch, and Clyde (orange) seems to do his own thing and stay out of the way.
- `SCATTER` - In scatter mode, the ghosts give up the chase for a few seconds and head for their respective home corners. It is a welcome but brief rest—soon enough, they will revert to chase mode and be after Pac-Man again.
- `FRIGHTENED` - Ghosts enter frightened mode whenever Pac-Man eats one of the four energizers located in the far corners of the maze. During the early levels, the ghosts will all turn dark blue (meaning they are vulnerable) and aimlessly wander the maze for a few seconds. They will flash moments before returning to their previous mode of behavior.

Reversal Of Fortune

In all three modes of behavior, the ghosts are prohibited from reversing their direction of travel. As such, they can only choose between continuing on their current course or turning off to one side or the other at the next intersection. Thus, once a ghost chooses which way to go at a maze intersection, it has no option but to continue forward on that path until the next intersection is reached. Of course, if you've spent any time playing Pac-Man, you already know the ghosts will reverse direction at certain times. But how can this be if they are expressly prohibited from doing so on their own? The answer is: when changing modes, the system can override the ghosts' normal behavior, forcing them to go the opposite way. Whenever this happens, it is a visual indicator of their behavior changing from one mode to another. Ghosts are forced to reverse direction by the system anytime the mode changes from: chase-to-scatter, chase-to-frightened, scatter-to-chase, and scatter-to-frightened. Ghosts do not reverse direction when changing back from frightened to chase or scatter modes.

When the system forces the ghosts to reverse course, they do not necessarily change direction simultaneously; some ghosts may continue forward for a fraction of a second before turning around. The delay between when the system signals a reversal and when a ghost actually responds depends on how long it takes the ghost to enter the next game tile along its present course after the reversal signal is given (more on tiles in Chapter 3). Once the ghost enters a new tile, it will obey the reversal signal and turn around.

Scatter, Chase, Repeat...

Ghosts alternate between scatter and chase modes during gameplay at predetermined intervals. These mode changes are easy to spot as the ghosts reverse direction when they occur. Scatter modes happen four times per level before the ghosts stay in chase mode indefinitely. Good players will take full advantage of the scatter periods by using the brief moment when the ghosts are not chasing Pac-Man to clear dots from the more dangerous areas of the maze. The scatter/chase timer gets reset whenever a life is lost or a level is completed. At the start of a level or after losing a life, ghosts emerge from the ghost pen already in the first of the four scatter modes.

For the first four levels, the first two scatter periods last for seven seconds each. They change to five seconds each for level five and beyond. The third scatter mode is always set to five seconds. The fourth scatter period lasts for five seconds on level one, but then is only 1/60th of a second for the rest of play. When this occurs, it appears as a simple reversal of direction by the ghosts. The first and second chase periods last for 20 seconds each. The third chase period is 20 seconds on level one but then balloons to 1,033 seconds for levels two through four, and 1,037 seconds for all levels beyond—lasting over 17 minutes! If the ghosts enter frightened mode, the scatter/chase timer is paused. When time runs out, they return to the mode they were in before being frightened and the scatter/chase timer resumes. This information is summarized in the following table (all values are in seconds):

Mode	Level 1	Levels 2–4	Levels 5+
Scatter	7	7	5
Chase	20	20	20
Scatter	7	7	5
Chase	20	20	20
Scatter	5	5	5
Chase	20	1033	1037
Scatter	5	1/60	1/60
Chase	indefinite	indefinite	indefinite

Frightening Behavior

Whenever Pac-Man eats one of the four energizer dots located near the corners of the board, the ghosts reverse direction and, on earlier levels, go into frightened mode for a short period of time. When frightened, the ghosts all turn the same shade of dark blue and move more slowly than normal. They wander aimlessly through the maze and flash white briefly as a warning before returning to their previous mode of behavior. Ghosts use a pseudo-random number generator (PRNG) to pick a way to turn at each intersection when frightened. The PRNG generates an pseudo-random memory address to read the last few bits from. These bits are translated into the direction a frightened ghost must first try. If the selected direction is not blocked by a wall or opposite the ghost's current direction of travel, it is accepted. Otherwise, the code proceeds in a clockwise fashion to the next possible direction and tries again, repeating this test until an acceptable direction is found. The PRNG gets reset with the same intial seed value at the start of each new level and whenever a life is lost. This results in frightened ghosts always choosing the same paths when executing patterns during play. As the levels progress, the time ghosts spend in frightened mode grows shorter until eventually they no longer turn blue at all (they still reverse direction, however). Refer to Table A.1 in the appendices for the frightened time in seconds and number of flashes, per level.

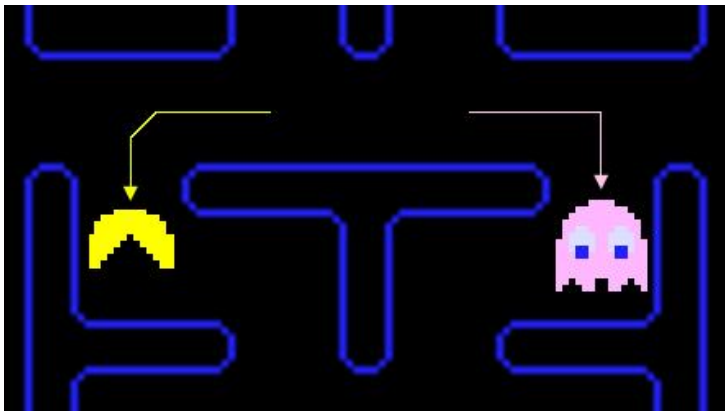
Speed

The game starts with Pac-Man at 80% of his maximum speed. By the fifth level, Pac-Man is moving at full speed and will continue to do so until the 21st level. At that point, he slows back down to 90% and holds this speed for the remainder of the game. Every time Pac-Man eats a regular dot, he stops moving for one frame (1/60th of a second), slowing his progress by roughly ten percent—just enough for a following ghost to overtake him. Eating an energizer dot causes Pac-Man to stop moving for three frames. The normal speed maintained by the ghosts is a little slower than Pac-Man's until the 21st level when they start moving faster than he does. If a ghost enters a side tunnel, however, its speed is cut nearly in half. When frightened, ghosts move at a much slower rate of speed than normal and, for levels one through four, Pac-Man also speeds up. The table below summarizes the speed data for both Pac-Man and the ghosts, per level. This information is also contained in Table A.1 in the appendices.

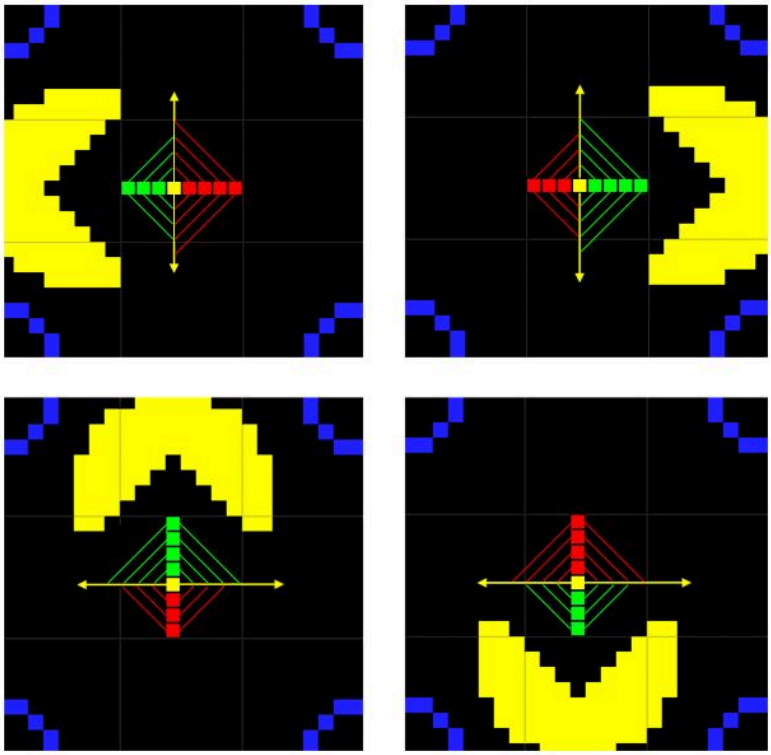
	PAC-MAN SPEED				GHOST SPEED		
LEVEL	NORM	NORM DOTS	FRIGHT	FRIGHT DOTS	NORM	FRIGHT	TUNNEL
1	80%	~71%	90%	~79%	75%	50%	40%
2 – 4	90%	~79%	95%	~83%	85%	55%	45%
5 – 20	100%	~87%	100%	~87%	95%	60%	50%
21+	90%	~79%	_	_	95%	_	50%

Cornering

Pac-Man is able to navigate the turns in the maze faster than his enemies. He does not have to wait until he reaches the middle of a turn to change direction as the ghosts do (see picture below). Instead, he may start turning several pixels before he reaches the center of a turn and for several pixels after passing it. Turns taken one or more pixels before reaching the center are “pre-turns”; turns taken one or more pixels after are “post-turns”. Players learn to consistently move the joystick in the direction Pac-Man should go well before arriving at the center of a turn, ensuring each pre-turn is started as many pixels away from center as possible. This technique is known as cornering and is one of the first skills a new Pac-Man player should master. For every successful pre-turn maneuver, Pac-Man puts a little more distance between himself and any ghosts following close behind. Such a small gain in distance may not seem terribly significant at first, but cornering through a quick series of turns will shake off even the most determined pursuer. It is a vital tool for survival in the higher levels of the game.



Whenever Pac-Man makes a pre-turn or post-turn, his orientation changes, and he starts to move one pixel in his new direction for every pixel traveled in his old direction, effectively doubling his speed as he moves at a 45 degree angle. Once he reaches the centerline of the new direction's path, he starts moving purely in that direction and his speed returns to normal. The greatest distance advantage is thereby gained by making the earliest pre-turn possible. The illustration below shows the layout of pre-turn pixels (shown in green), center point pixels (shown in yellow), and post-turn pixels (shown in red) for each of the four possible directions a turn can be approached. Each example shows Pac-Man entering the same four-way intersection from a different direction. When entering from the left, there are three pre-turn pixels before the center of the turn, and four post-turn pixels. Conversely, entering the same intersection from the right yields four pre-turn pixels and three post-turn ones. Entering from the top as opposed to the bottom exhibits the same property. For any turn that is made later than the earliest possible pre-turn, Pac-Man will be one frame behind where he would be for every pixel of “lateness” in the turn. Basically, it pays to move the joystick well before reaching a turn to maximize your speed.



Turning at the earliest pre-turns possible is also required to successfully execute most any pattern. Patterns are meant to be played with perfect cornering because it removes the human element of uncertainty as to when Pac-Man will turn. Without cornering, it would be nigh-impossible to reproduce the exact timing of every turn as made by the pattern's author, thereby increasing the possibility of unpredictable ghost behavior due to Pac-Man not being in the exact same tile at the exact same time anymore. Typically, the most popular patterns have been those that tend to “hold together” well when small input timing flaws occur (turning three pixels away from center instead of four when approaching a turn from the right is a timing flaw, for example). Other patterns—especially those that bring Pac-Man very close to the ghosts late in the sequence—tend to “fall apart” unless every turn is perfectly cornered. During a long Pac-Man session, even the best players will make occasional timing mistakes during a fast series of turns and have to deal with the possible consequences. As such, one should aim for perfect cornering at all times but remain alert for unexpected ghost behavior from subtle input timing flaws creeping into the pattern.

Home Sweet Home

Commonly referred to as the ghost house or monster pen, this cordoned-off area in the center of the maze is the domain of the four ghosts and off-limits to Pac-Man. Whenever a level is completed or a life is lost, the ghosts are returned to their starting positions in and around the ghost house before play continues—Blinky is always located just above and outside, while the other three are placed inside: Inky on the left, Pinky in the middle, and Clyde on the right. The pink door on top is used by the ghosts to enter or exit the house.
Once a ghost leaves, however, it cannot reenter unless it is first captured by Pac-Man—then the disembodied eyes can return home to be revived. Since Blinky is already on the outside after a level is completed or a life is lost, the only time he can get inside the ghost house is after Pac-Man captures him, and he immediately turns around to leave once revived. That's about all there is to know about Blinky's behavior in terms of the ghost house, but determining when the other three ghosts leave home is an involved process based on several variables and conditions. The rest of this section will deal with them exclusively. Accordingly, any mention of “the ghosts” below refers to Pinky, Inky, and Clyde, but not Blinky.

The first control used to evaluate when the ghosts leave home is a personal counter each ghost retains for tracking the number of dots Pac-Man eats. Each ghost's “dot counter” is reset to zero when a level begins and can only be active when inside the ghost house, but only one ghost's counter can be active at any given time regardless of how many ghosts are inside. The order of preference for choosing which ghost's counter to activate is: Pinky, then Inky, and then Clyde. For every dot Pac-Man eats, the preferred ghost in the house (if any) gets its dot counter increased by one. Each ghost also has a “dot limit” associated with his counter, per level. If the preferred ghost reaches or exceeds his dot limit, it immediately exits the house and its dot counter is deactivated (but not reset). The most-preferred ghost still waiting inside the house (if any) activates its timer at this point and begins counting dots.

Pinky's dot limit is always set to zero, causing him to leave home immediately when every level begins. For the first level, Inky has a limit of 30 dots, and Clyde has a limit of 60. This results in Pinky exiting immediately which, in turn, activates Inky's dot counter. His counter must then reach or exceed 30 dots before he can leave the house. Once Inky starts to leave, Clyde's counter (which is still at zero) is activated and starts counting dots. When his counter reaches or exceeds 60, he may exit. On the second level, Inky's dot limit is changed from 30 to zero, while Clyde's is changed from 60 to 50. Inky will exit the house as soon as the level begins from now on. Starting at level three, all the ghosts have a dot limit of zero for the remainder of the game and will leave the ghost house immediately at the start of every level.

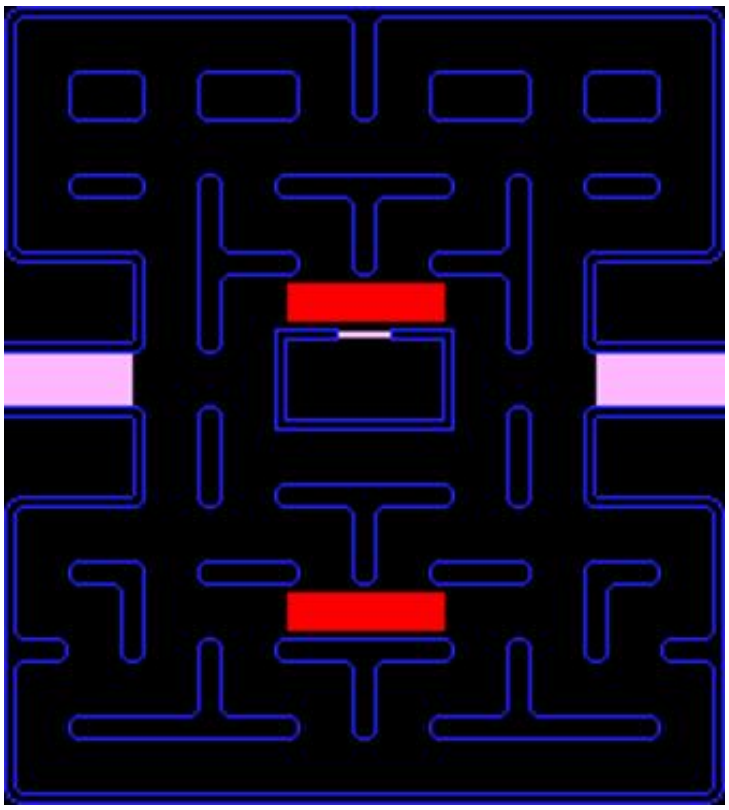
Whenever a life is lost, the system disables (but does not reset) the ghosts' individual dot counters and uses a global dot counter instead. This counter is enabled and reset to zero after a life is lost, counting the number of dots eaten from that point forward. The three ghosts inside the house must wait for this special counter to tell them when to leave. Pinky is released when the counter value is equal to 7 and Inky is released when it equals 17. The only way to deactivate the counter is for Clyde to be inside the house when the counter equals 32; otherwise, it will keep counting dots even after the ghost house is empty. If Clyde is present at the appropriate time, the global counter is reset to zero and deactivated, and the ghosts' personal dot limits are re-enabled and used as before for determining when to leave the house (including Clyde who is still in the house at this time).

If dot counters were the only control, Pac-Man could simply stop eating dots early on and keep the ghosts trapped inside the house forever. Consequently, a separate timer control was implemented to handle this case by tracking the amount of time elapsed since Pac-Man has last eaten a dot. This timer is always running but gets reset to zero each time a dot is eaten. Anytime Pac-Man avoids eating dots long enough for the timer to reach its limit, the most-preferred ghost waiting in the ghost house (if any) is forced to leave immediately, and the timer is reset to zero. The same order of preference described above is used by this control as well. The game begins with an initial timer limit of four seconds, but lowers to it to three seconds starting with level five.

The more astute reader may have already noticed there is subtle flaw in this system resulting in a way to keep Pinky, Inky, and Clyde inside the ghost house for a very long time after eating them. The trick involves having to sacrifice a life in order to reset and enable the global dot counter, and then making sure Clyde exits the house before that counter is equal to 32. This is accomplished by avoiding eating dots and waiting for the timer limit to force Clyde out. Once Clyde is moving for the exit, start eating dots again until at least 32 dots have been consumed since the life was lost. Now head for an energizer and gobble up some ghosts. Blinky will leave the house immediately as usual, but the other three ghosts will remain “stuck” inside as long as Pac-Man continues eating dots with sufficient frequency as not to trigger the control timer. Why does this happen? The key lies in how the global dot counter works—it cannot be deactivated if Clyde is outside the house when the counter has a value of 32. By letting the timer force Clyde out before 32 dots are eaten, the global dot counter will keep counting dots instead of deactivating when it reaches 32. Now when the ghosts are eaten by Pac-Man and return home, they will still be using the global dot counter to determine when to leave. As previously described, however, this counter's logic only checks for three values: 7, 17, and 32, and once those numbers are exceeded, the counter has no way to release the ghosts associated with them. The only control left to release the ghosts is the timer which can be easily avoided by eating a dot every so often to reset it.

The last thing to mention about the ghost house is how to determine whether a ghost will move right or left after exiting the home. Ghosts typically move to the left once they get outside, but if the system changes modes one or more times when a ghost is inside, that ghost will move to the right instead of the left upon leaving the house.

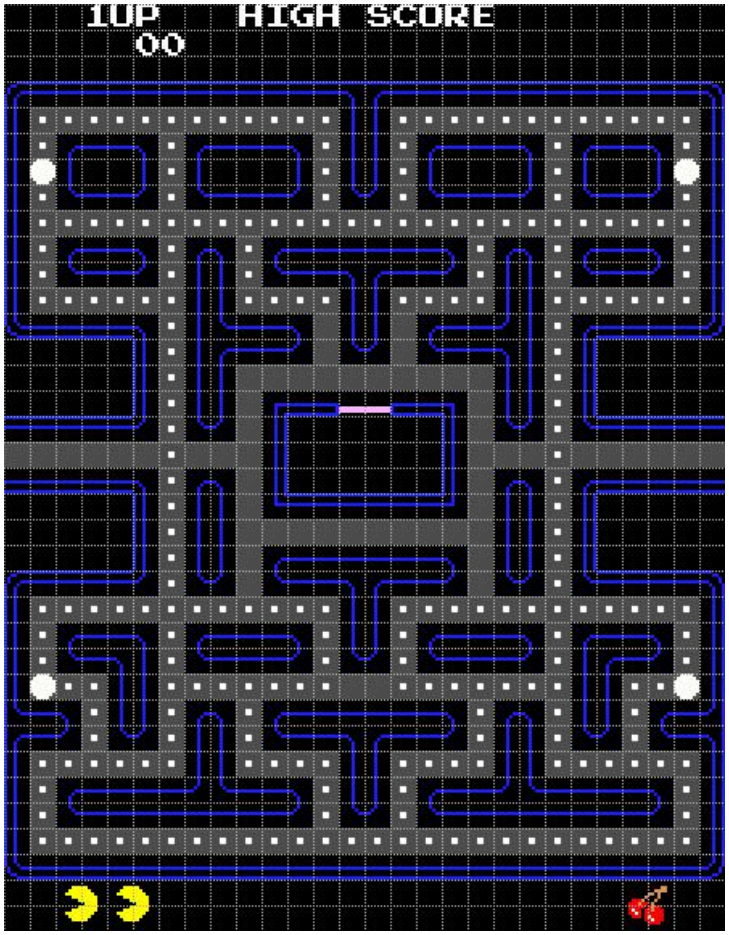
Areas To Exploit



The illustration above highlights four special “zones” in the maze where ghost behavior is limited by certain conditions which can be exploited to the player's advantage. The two red zones represent the areas where ghosts are forbidden to make upward turns. Once a ghost enters either of these two zones, it may only travel from right-to-left or left-to-right until exiting the area. Thus, only Pac-Man has access to these four, upward-facing tunnel entrances. It will serve the player well to remember the ghosts can still access these tunnels from the other end! The red zone restrictions are enforced during both scatter and chase modes, but in frightened mode the red zones are ignored temporarily, allowing the ghosts to turn upwards if they so choose. The pink zones are in the two halves of the connecting side-tunnel. As mentioned previously, any ghost that enters the tunnel will suffer an immediate speed penalty until leaving the zone. This slow-down rule is always enforced and applies to ghosts only—Pac-Man is immune.

Maze Logic

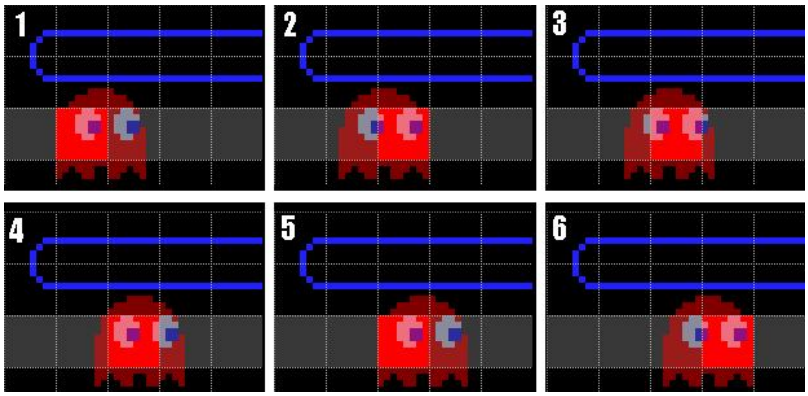
We need to take a look at how ghosts are able to move through the maze in pursuit of a goal. All pathfinding logic described in this chapter is shared by the four ghosts—it is important to understand what they have in common before we get into what makes them different. Before we proceed, let's see how the game tracks the location of Pac-Man and the four ghosts (herein referred to as actors for brevity's sake). The visible game screen should be thought of as a regular grid of tiles, each eight pixels square. The actual pixel dimensions of the screen are 224 x 288, so dividing each value by eight yields a grid that is 28 x 36 tiles in size:



Each tile is either in legal space or dead space. In the picture above, legal space is shown as the gray-colored tiles; all other tiles are considered dead space. Actors only travel between the tiles in legal space. Each dot sits in the center of a tile, meaning they are exactly eight pixels (one tile) apart—this is useful for estimating distances during gameplay.

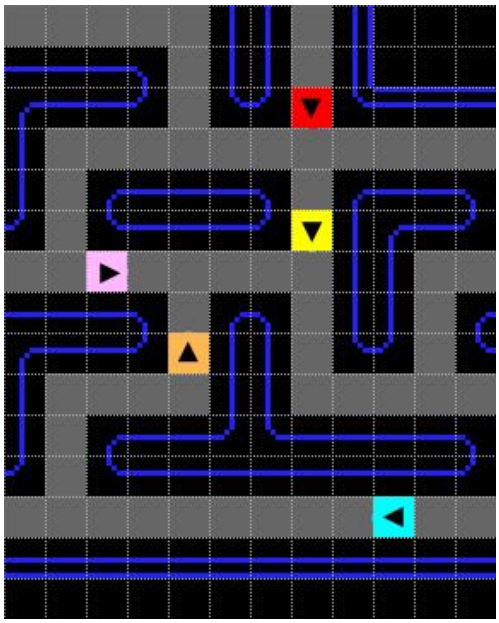
What Tile Am I In?

As the actors move through the maze, the game keeps track of the tile each one occupies. An actor is only associated with a single tile at a time, although its graphic will overlap into the surrounding tiles. The location of the actor's center point is what determines the tile it occupies at any given time. As the actors can move at pixel-level precision, they are often not centered directly on top of the tile they are in. Consider the following example:



The transparent red ghost is moving left-to-right across a row of tiles in legal space. In frame one, its occupied tile (shown in bright red) is near the left side of the picture. It does not matter that some of the ghost's graphic is not in the tile—what matters is that the ghost's center point is in the tile. By frame two, it has moved far enough for its center point to be in the adjacent tile to the right and its occupied tile is updated accordingly. The ghost continues to be associated with the same tile until frame six where its center point has now crossed over into the next one.

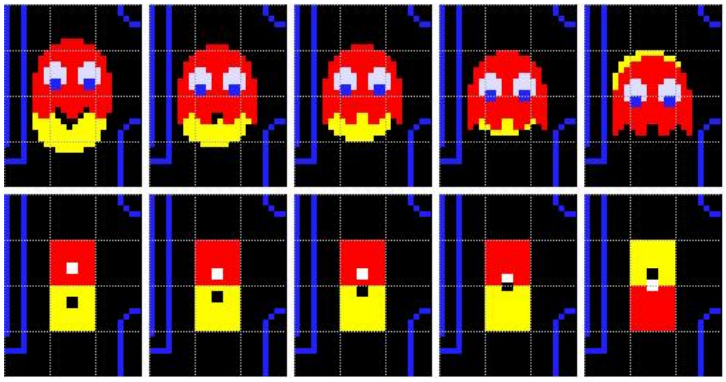
The underlying concept of tiles is essential for understanding the ghosts' pathfinding logic as it only cares about the tile an actor occupies—not its per-pixel location within that tile. To the logic routines, the five actors look very much like the picture below. Each actor is defined by the tile it presently occupies along with its current direction of travel. Distances between actors are also measured in tiles (the pink ghost is five tiles away from Pac-Man horizontally and one tile away vertically, for example).



Just Passing Through

It wasn't too long after the release of Pac-Man when word began to spread of players occasionally passing straight through a ghost unharmed, seemingly at random. This rumor turned out to be completely true as most die-hard Pac-Man players can attest. If you play the game long enough, you will eventually see Pac-Man run into one of the ghosts and come out unscathed on the other side—it doesn't happen very often so enjoy it when it does! Some players have even gone so far as to incorporate this mysterious pass-through oddity into their patterns.

The root cause of this elusive peculiarity lies in the way the game detects collisions between Pac-Man and the four ghosts. Any time Pac-Man occupies the same tile as a ghost, he is considered to have collided with that ghost and a life is lost. It is irrelevant whether the ghost moved into Pac-Man's tile or Pac-Man into the ghost's—the result is the same either way. This logic proves sufficient for handling collisions more than 99% of the time during gameplay, but does not account for one very special case:



The above picture illustrates the conditions necessary to produce this curious behavior. There are five consecutive frames showing Blinky and Pac-Man passing through each other. Below each frame is the same scene represented by the tiles they currently occupy and the per-pixel location of their center points. Pac-Man and Blinky are at just the right position and speed relative to one another to cause them to swap tiles with each other simultaneously. In other words, Pac-Man's center point moves upwards into Blinky's tile in the same 1/60th of a second that Blinky's center point moves downwards into Pac-Man's tile, resulting in them moving past each other without colliding. Note that Pac-Man's origin point is centered on the top edge of his tile in frame four; this is still considered to be inside the bottom tile, but moving up one more pixel will push him over the edge into the next one. Pac-Man and Blinky have now swapped tiles with each other in frame five, and Pac-Man can go on his merry way because he never “collided” (i.e., shared the same tile) with Blinky at all!

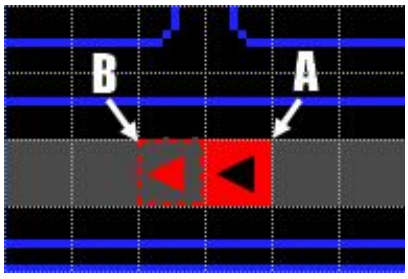
Target Tiles

Whenever a ghost is in chase or scatter mode, it is trying to reach a target tile somewhere on (or off) the screen. A target tile is merely a way to describe the tile a ghost would like to occupy at any given moment. This tile can be fixed in place or change location frequently. Whenever the ghosts scatter to the corners of the maze, for example, each ghost is striving to reach a fixed target tile located somewhere near its home corner. In chase mode, the target tile is usually (but not always) related to Pac-Man's current tile which changes often. Although it may not be obvious at first, the only difference between chase and scatter mode to a ghost is where its target tile is located. The same pathfinding logic applies in either case.

Looking Ahead

Ghosts are always thinking one step into the future as they move through the maze. Whenever a ghost enters a new tile, it looks ahead to the next tile along its current direction of travel and decides which way it will go when it gets there. When it eventually reaches that tile, it will change its direction of travel to whatever it had decided on a tile beforehand. The process is then repeated, looking ahead into the next tile along its new direction of travel and making its next decision on which way to go.

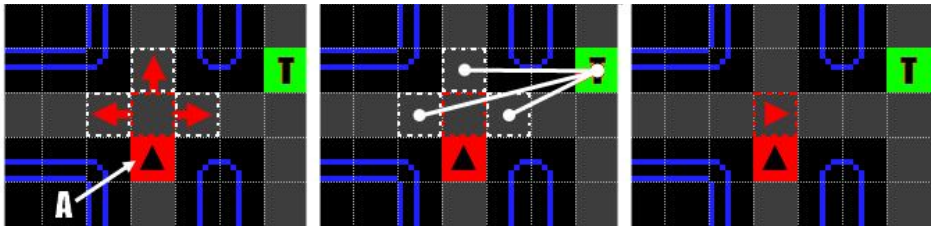
When a ghost looks ahead into the upcoming tile, it must examine the possible exits from that tile to determine a way to proceed. In the picture below, the red ghost has just arrived at tile A and is moving right-to-left. It immediately looks ahead to tile B (the next tile along its direction of travel). Each tile has four potential exits to be considered: right, left, up, and down. In the case of tile B, the up and down exits are blocked by walls and must be discarded as potential candidates. The right exit is also discounted because it would only take the ghost back to tile A again, and ghosts never voluntarily reverse direction. With three of the four possible exits eliminated from tile B, moving left is the only remaining choice.



This example is the most simple to explain as the ghost has but one way it can legally move. As such, we did not have to worry about where its target tile was located. The majority of game tiles in legal space are similar to this one, but things get more interesting when a ghost approaches a tile with more potential exits to choose from.

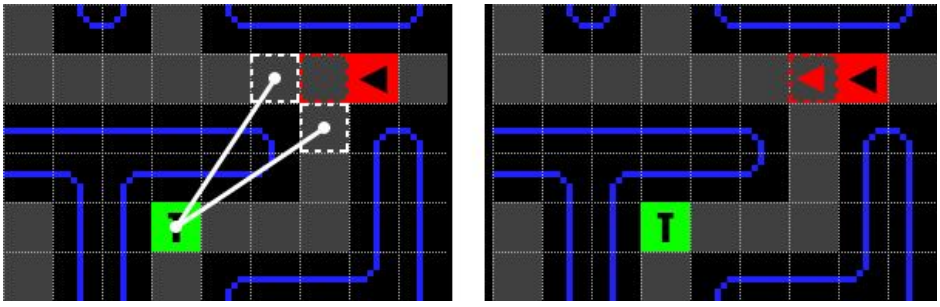
Intersections

When a ghost arrives one tile away from an upcoming intersection, it must choose between several possible directions in which to proceed. Consider the following example:



In the first picture, the red ghost has just reached tile A and is seeking its target (shown as the green tile). It immediately looks ahead to the subsequent tile along its present direction of travel (up). In this case, that tile is a four-way intersection. As this intersection tile has no walls blocking off any of the exits, the ghost can only discard his reverse direction (down), leaving three exits open for travel. It looks one tile beyond the intersection in each of the three remaining directions, collecting “test tiles” (shown as the tiles with dashed, white lines). In the middle picture, the ghost triangulates the distance from each of these test tiles to its target tile. Whichever direction's test tile has the shortest distance to the target becomes the direction the ghost will take upon reaching the intersection tile. In this case, the right test tile has the shortest distance to the target, and the ghost updates its chosen direction for the intersection tile accordingly.

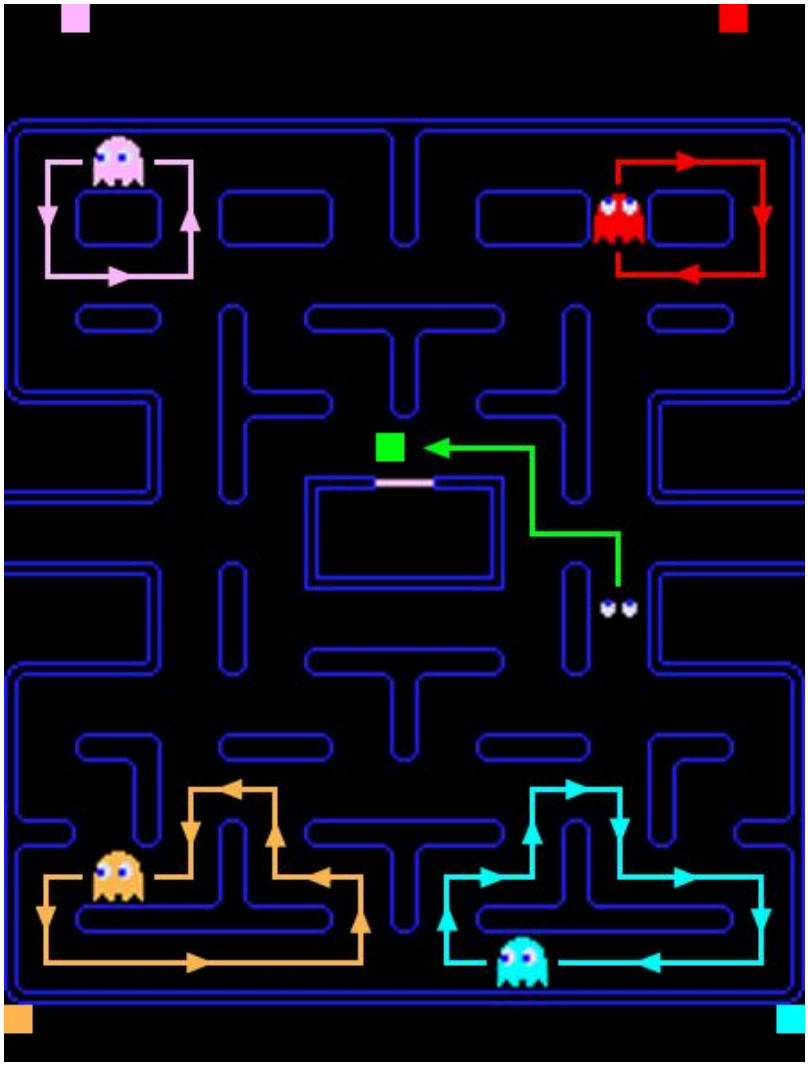
Sometimes a ghost is presented with two or more test tiles that have the same distance to the target tile. In the example below, the red ghost must choose between moving down or left at the upcoming intersection tile. Unfortunately, both test tiles have the same distance to the target (bottom left). To break the tie, the ghost prefers directions in this order: up, left, down, right. Up is the most preferred direction; right is the least. Therefore, the ghost chooses to go left at the intersection because left precedes down in the preference list. Although it may seem obvious to a person that going down was the better choice to reach the target, ghosts are not that smart. They cannot see more than a few tiles ahead and, as a consequence, cannot recognize the disparity between these two options.



Fixed Target Tiles

Each ghost has a fixed target tile it tries to reach while in scatter mode. The picture below shows the physical location of the scatter mode targets used by each ghost (matched to each ghost's color scheme). Notice each target tile is in dead space above or below the actual maze making them impossible for the ghosts to reach. This results in each ghost heading toward the corner of the maze nearest its respective scatter target and then making circles around this area until another mode change occurs. That's all scatter mode really is. The only reason a ghost has a “favorite corner” of the maze at all is due to the location of a fixed target tile it will never reach.

An additional fixed target tile is employed whenever a ghost is eaten by Pac-Man and its disembodied eyes need to return to the ghost house in the center of the maze. This target is located directly above the left side of the “door” to the ghost house and is shown in the picture below as the green tile.

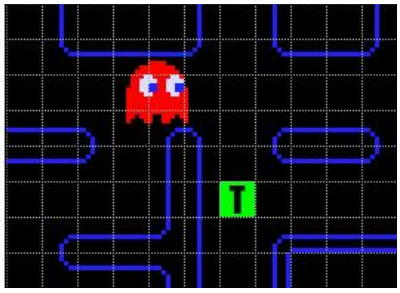


Meet The Ghosts

In the last chapter, we learned how a ghost follows a target tile through the maze. Now we will take a closer look at Blinky, Pinky, Inky, and Clyde to better understand why they behave so differently when in chase mode. They all share the same pathfinding logic for chasing a target tile, so how is it each one behaves differently when following Pac-Man? The answer is delightfully simple: Pac-Man's tile is not always the target. Every ghost has a distinct method for calculating its target tile in chase mode, resulting in their unique personalities. Some of the ghosts use Pac-Man's actual tile as the target; others only use it as an intermediate step to find another tile. Sometimes a ghost is targeting a tile that has absolutely nothing to do with Pac-Man at all! Regardless of where a ghost's target tile is at the time, Pac-Man will still be killed if he gets in that ghost's way.

Rumor has it Toru Iwatani and his team spent months doing nothing but tweaking and refining the ghost A.I. routines before releasing Pac-Man to the world. Their efforts show in the final product: Itawani's team created the illusion of complex pathfinding by using very simple logic and very little code.

Blinky: The red ghost's character is aptly described as that of a shadow and is best-known as “Blinky”. In Japan, his character is represented by the word oikake, which means “to run down or pursue”. Blinky seems to always be the first of the ghosts to track Pac-Man down in the maze. He is by far the most aggressive of the four and will doggedly pursue Pac-Man once behind him.



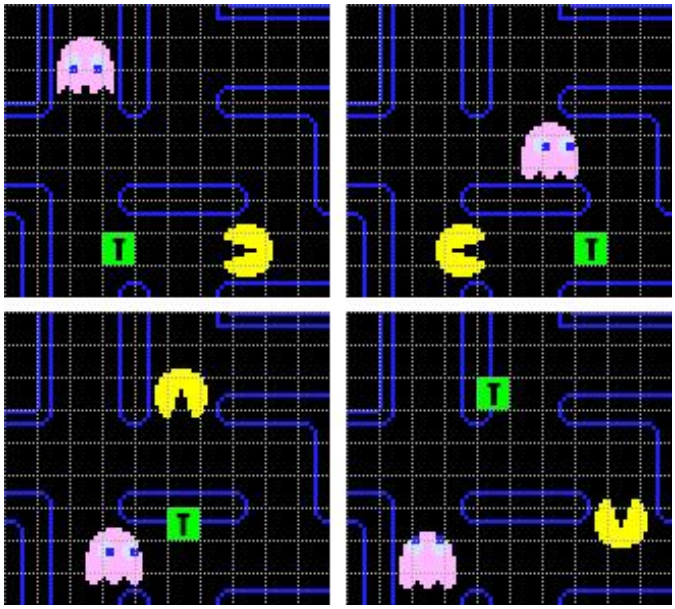
Of all the ghosts' targeting schemes for chase mode, Blinky's is the most simple and direct, using Pac-Man's current tile as his target. In the pictures above, we can see Blinky's target tile is the same as Pac-Man's currently occupied tile. Targeting Pac-Man directly in this way results in a very determined and tenacious ghost who is tough to shake when he's right behind you.

All ghosts move at the same rate of speed when a level begins, but Blinky will increase his rate of speed twice each round based on the number of dots remaining in the maze (if Pac-Man dies this is not necessarily true - more on this in a moment). While in this accelerated state, Blinky is commonly called “Cruise Elroy”, yet no one seems to know where this custom was originated or what it means. On the first level, for example, Blinky becomes Elroy when there are 20 dots remaining in the maze, accelerating to be at least as fast as Pac-Man. More importantly, his scatter mode behavior is also modified at this time to keep targeting Pac-Man's current tile in lieu of his typical fixed target tile for any remaining scatter periods in the level (he will still reverse direction when entering/exiting a scatter period). This results in Elroy continuing to chase Pac-Man while the other three ghosts head for their corners as normal. As if that weren't bad enough, when only 10 dots remain, Elroy speeds up again to the point where he is now moving faster than Pac-Man. As the levels progress, Blinky will turn into Elroy with more dots remaining in the maze than in previous rounds. Refer to Table A.1 in the appendices for dot counts and speeds for both Elroy changes, per level.

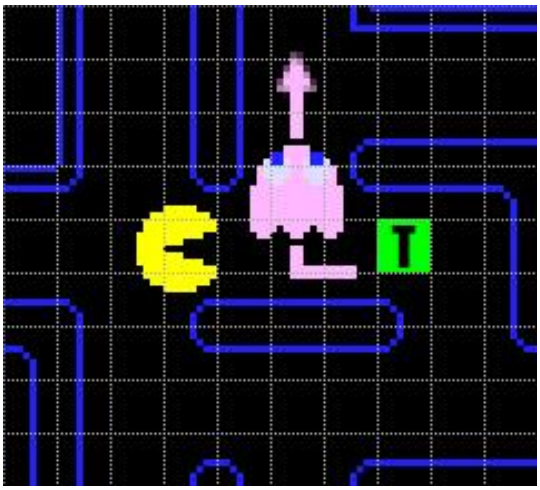
Determining when Blinky turns into Elroy can become more complicated if Pac-Man is killed. The ghosts and Pac-Man are reset to their starting positions whenever a life is lost and, when play continues, Blinky's “Cruise Elroy” abilities are temporarily suspended until the orange ghost (Clyde) stops bouncing up and down inside the ghost house and moves toward the door to exit. Until this happens, Blinky's speed and scatter behavior will remain normal regardless of the number of dots remaining in the maze. Once this temporary restriction is lifted, however, Blinky will resume changing his behavior based on the dot count.

Pinky: Nicknamed “Pinky”, the pink ghost's character is described as one who is speedy. In Japan, he is characterized as machibuse, meaning “to perform an ambush”, perhaps because Pinky always seems to be able to get ahead of you and cut you off when you least expect it. He always moves at the same speed as Inky and Clyde, however, which suggests speedy is a poor translation of the more appropriate machibuse. Pinky and Blinky often seem to be working in concert to box Pac-Man in, leaving him with nowhere to run.

In chase mode, Pinky behaves as he does because he does not target Pac-Man's tile directly. Instead, he selects an offset four tiles away from Pac-Man in the direction Pac-Man is currently moving (with one exception). The pictures below illustrate the four possible offsets Pinky will use to determine his target tile based on Pac-Man's orientation:

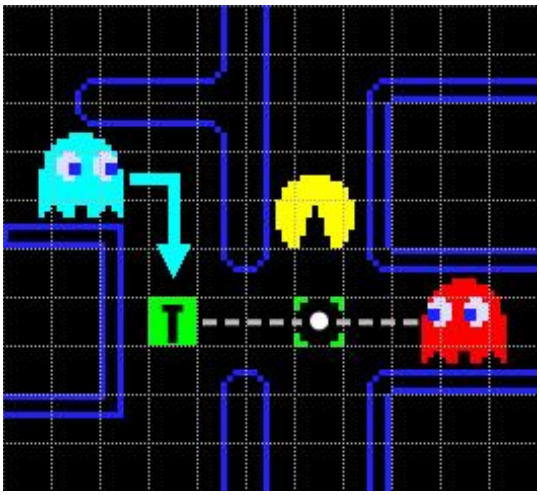


If Pac-Man is moving left, Pinky's target tile will be four game tiles to the left of Pac-Man's current tile. If Pac-Man is moving right, Pinky's tile will be four tiles to the right. If Pac-Man is moving down, Pinky's target is four tiles below. Finally, if Pac-Man is moving up, Pinky's target tile will be four tiles up **and** four tiles to the left. This interesting outcome is due to a subtle error in the logic code that calculates Pinky's offset from Pac-Man. This piece of code works properly for the other three cases but, when Pac-Man is moving upwards, triggers an overflow bug that mistakenly includes a left offset equal in distance to the expected up offset (we will see this same issue in Inky's logic later).

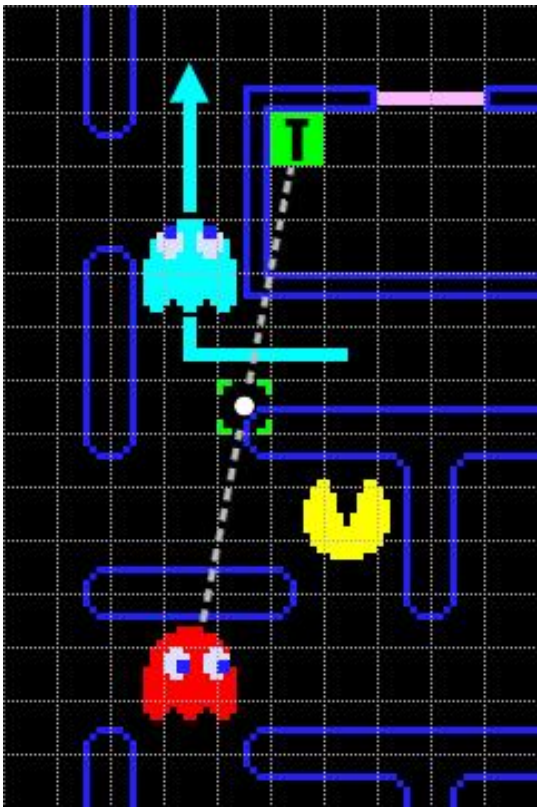


Pinky is the easiest ghost to exert control over thanks to his targeting scheme. By changing direction, you can dictate where Pinky will turn next when he is nearby (see above picture). If you are facing off closely with Pinky, he will turn before he reaches you if he can. This happens due to the fact Pac-Man has come close enough to Pinky for Pinky's target tile to now be behind him. In the picture above, Pinky chooses to turn up at the intersection because moving left would have taken him further away from his target tile. The longest-lived example of this is the technique known as “head faking”. This is where the player shakes the joystick to cause Pac-Man to rapidly change direction back and forth, hopefully causing a ghost to change course in the process. As it turns out, the shaking is not necessary—one well-timed, quick reversal of direction towards Pinky just before he decides what to do at an upcoming intersection is all that is needed to get him off your tail.

Inky: The light-blue ghost is nicknamed “Inky” and his character is described as one who is bashful. In Japan, he is portrayed as kimagure, meaning “a fickle, moody, or uneven temper”. Perhaps not surprisingly, Inky is the least predictable of the ghosts. Sometimes he chases Pac-Man aggressively like Blinky; other times he jumps ahead of Pac-Man as Pinky would. He might even wander off like Clyde on occasion! In fact, Inky may be the most dangerous ghost of all due to his erratic behavior. Bashful is not a very good translation of kimagure, and misleads the player to assume Inky will shy away from Pac-Man when he gets close which is not always the case.



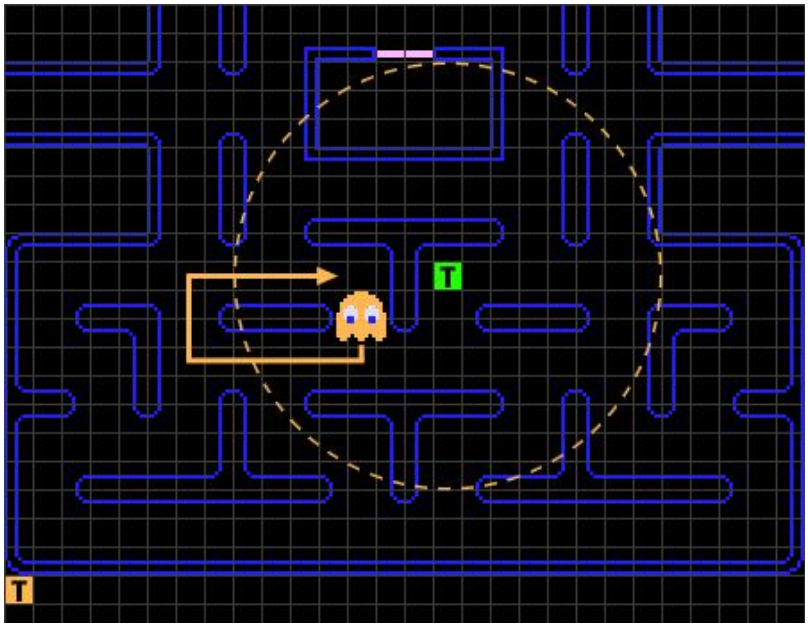
Inky uses the most complex targeting scheme of the four ghosts in chase mode. He needs Pac-Man's current tile/orientation and Blinky's current tile to calculate his final target. To determine Inky's target, we must first establish an intermediate offset two tiles in front of Pac-Man in the direction he is moving (represented by the tile bracketed in green above). Now imagine drawing a vector from the center of the red ghost's current tile to the center of the offset tile, then double the vector length by extending it out just as far again beyond the offset tile. The tile this new, extendend vector points to is Inky's actual target as shown above.



For the same reasons already discussed in Pinky's case, an overflow error occurs with the intermediate offset tile generated for Inky's calculation when Pac-Man is moving up resulting in an offset tile that is two tiles above **and** two tiles to the left (see above picture). The other three orientations (left, right, down) produce the expected result of an offset two tiles in front of Pac-Man in the direction he is moving.

Inky's targeting logic will keep him away from Pac-Man when Blinky is far away from Pac-Man, but as Blinky draws closer, so will Inky's target tile. This explains why Inky's behavior seems more variable as Pac-Man moves away from Blinky. Like Pinky, Inky's course can often be altered by Pac-Man changing direction or “head-faking”. How much or how little effect this will have on Inky's decisions is directly related to where Blinky is at the time.

Clyde: The orange ghost is nicknamed “Clyde” and is characterized as one who is pokey. In Japan, his character is described as otoboke, meaning “pretending ignorance”, and his nickname is “Guzuta”, meaning “one who lags behind”. In reality, Clyde moves at the same speed as Inky and Pinky so his character description is a bit misleading. Clyde is the last ghost to leave the pen and tends to separate himself from the other ghosts by shying away from Pac-Man and doing his own thing when he isn't patrolling his corner of the maze. Although not nearly as dangerous as the other three ghosts, his behavior can seem unpredictable at times and should still be considered a threat.



During chase mode, Clyde's targeting logic changes based on his proximity to Pac-Man (represented by the green target tile above). He first calculates the Euclidean distance between his tile and Pac-Man's tile. If the distance between them is eight tiles or more, Clyde targets Pac-Man directly just as Blinky does. If the distance between them is less than eight tiles, however, Clyde switches his target to the tile he normally uses during scatter mode and heads for his corner until he gets far enough away to start targeting Pac-Man again. In the picture above, Clyde is stuck in an endless loop (as long as Pac-Man stays where he is) thanks to this scheme. While occupying any tile completely outside the dashed perimeter, Clyde's target is Pac-Man. Upon entering the area, Clyde changes his mind and heads for his scatter target instead. Once he exits the perimeter, his target will change back to Pac-Man's current tile again. The end result is Clyde circling around and around until Pac-Man moves elsewhere or a mode change occurs. Clyde is fairly easy to avoid once you understand his targeting scheme. Just remember: he is still dangerous if you manage to get in his way as he runs back to his corner or before he can reach an intersection to turn away from you.

On The Edge Of Forever

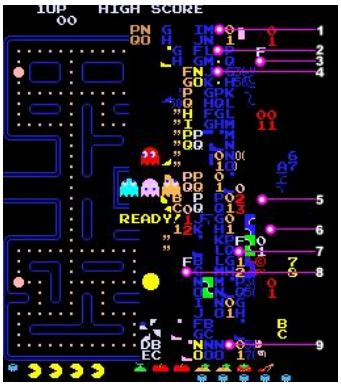
Pac-Man was always meant to be a game with no ending. The developers at Namco mistakenly assumed the game's increasing difficulty was sufficient to prevent anyone from playing indefinitely. Of course, within a few years of Pac-Man's release, players had discovered that every level beyond the 21st was identical. Patterns were quickly created to exploit this fact and, for any player able to get past the first 20 levels, the game now became a test of endurance to see how many points you could rack up before losing focus and making a mistake. High scores soared into the millions and most players agreed the game simply went on forever. Eventually, a few highly-skilled players were able to complete 255 consecutive levels of play (scoring over three million points and taking several hours to accomplish) and found a surprise waiting for them on level 256. It was a surprise no one knew about—not even the developers at Namco.

The 256th level displays the left half of the maze correctly, but the right half is a jumbled mess of randomly colored letters, numbers, and symbols. Notice the bonus counter in the lower-right of the screen is also malfunctioning. The left side of the maze plays normally, but the right side is a different story. Although both the player and the ghosts can navigate through the right half of the screen, the original maze walls no longer apply. Instead, Pac-Man must be guided through a confusing series of open areas, tunnels, one-way intersections, lone walls, and pass-throughs—all invisible to the player—while four ghosts are in hot pursuit.

Why does this broken level happen in the first place?

The culprit is the routine responsible for drawing the bonus symbols along the bottom edge of the screen. Here's what happens: when level 256 is reached, the internal level counter is incremented to 255 (the level counter starts at zero – not one) and the routine for drawing the bonus symbols is called. The routine loads the current level counter value (255) into a CPU register and increments that register by one. Unfortunately, 255 is the largest number that can fit in a single byte which is the size of the Z-80 CPU registers, so when the value is incremented the overflow is discarded leaving a zero in the register instead of the expected value of 256. This zero value leads the routine to believe this is an early level since its value is less than seven. The routine starts drawing bonus symbols using the confused register as a counter. At the end of every drawing loop, the register is decreased by one and then checked to see if it is zero (the signal for the routine to stop drawing symbols). Since the register already has a zero in it to start, the first decrement will roll the value back to 255. It will keep decrementing the register and drawing symbols until the register is reduced to zero again, causing the loop to run a total of 256 times. This means that memory locations outside the bounds of the bonus symbol table are drawn to the screen at increasing locations in video memory. This half-broken level was named the “split screen” by players; developers refer to it as a “kill screen”.

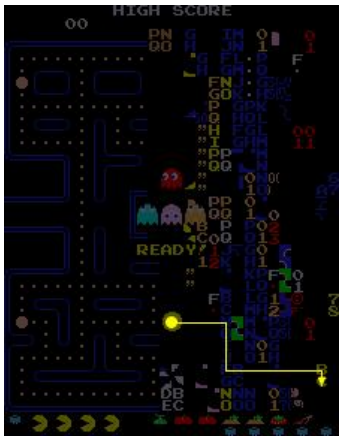
Playing The Level



There are 114 dots on the left half of the screen, nine dots on the right, and one bonus key, totaling 6,310 points. When all of the dots have been cleared, nothing happens. The game does not consider a level to be completed until 244 dots have been eaten, so there is nothing left to do but sacrifice Pac-Man to a hungry ghost. Interestingly, every time a life is lost, the nine dots on the right half of the screen get reset and can be eaten again, resulting in an additional 90 points per extra man. In the best-case scenario (five extra men), 6,760 points is the maximum score possible, but only 168 dots can be harvested—not enough to change levels—so we are stuck. There are no more dots to gobble or energizers to eat. There is no final victory waiting for Pac-Man, only an empty half-maze full of ghosts. The game has an ending after all—just not a very happy or exciting one.

Four of the nine dots on the right half of the screen are invisible, but can be heard when eaten. The picture on the left shows all nine dot locations. Dots 1, 5, 6, and 9 are invisible; the rest can be seen but some are a different color than normal.

Anyone reaching this level quickly realized: to safely map out the right side of the screen something had to be done about the ghosts. After much tinkering, it was discovered that a ghost would get “trapped” on the right edge of the screen if he got too close to it. Once trapped, a ghost can only move up or down but never right or left again. By leading ghosts near the edge of the screen, a skilled player could eventually get the ghosts out of the way and concentrate on exploring the right half of the maze and collecting the dots.



There are many methods for trapping the ghosts. One of the easiest ways to trap the three important ghosts is shown in the picture to the right. The yellow line shows Pac-Man's path from the start of the level to a spot near the bottom-right. The exact instructions are as follows: begin by going right until you reach a blue letter 'N', then go down. Keep going down until you reach a blue letter 'F', then go right. Keep going right until you reach a yellow 'B', then go down again. When executed properly, Pac-Man will hit an invisible wall almost immediately after the last turn is made. Now we wait. The red ghost will get stuck first. The pink ghost follows a few seconds later. The blue ghost will continue to move freely for several moments until the next scatter mode occurs.

At that point, it will try to reach some location near the right edge of the screen and get stuck with the pink and red ghost instead. Now the orange ghost is the only one still on the loose (bottom-right). Clyde is no real threat, however, since he runs to his corner whenever Pac-Man gets close (see Chapter 4), making it relatively easy to clean up all the dots. Be sure to take care around the lower-left corner of the maze—the orange ghost will have nowhere left to run to and will be much more aggressive.

Reference Tables

Table A.1 — Level Specifications (100% speed = 75.75757625 pixels/sec)															
Level	Bonus Symbol	Bonus Points	Pac-Man Speed	Pac-Man Dots Speed	Ghost Speed	Ghost Tunnel Speed	Elroy 1 Dots Left	Elroy 1 Speed	Elroy 2 Dots Left	Elroy 2 Speed	Fright. Pac-Man Speed	Fright Pac-Man Dots Speed	Fright Ghost Speed	Fright. Time (in sec.)	# of Flashes
1	Cherries	100	80%	~71%	75%	40%	20	80%	10	85%	90%	~79%	50%	6	5
2	Strawberry	300	90%	~79%	85%	45%	30	90%	15	95%	95%	~83%	55%	5	5
3	Peach	500	90%	~79%	85%	45%	40	90%	20	95%	95%	~83%	55%	4	5
4	Peach	500	90%	~79%	85%	45%	40	90%	20	95%	95%	~83%	55%	3	5
5	Apple	700	100%	~87%	95%	50%	40	100%	20	105%	100%	~87%	60%	2	5
6	Apple	700	100%	~87%	95%	50%	50	100%	25	105%	100%	~87%	60%	5	5
7	Grapes	1000	100%	~87%	95%	50%	50	100%	25	105%	100%	~87%	60%	2	5
8	Grapes	1000	100%	~87%	95%	50%	50	100%	25	105%	100%	~87%	60%	2	5
9	Galaxian	2000	100%	~87%	95%	50%	60	100%	30	105%	100%	~87%	60%	1	3
10	Galaxian	2000	100%	~87%	95%	50%	60	100%	30	105%	100%	~87%	60%	5	5
11	Bell	3000	100%	~87%	95%	50%	60	100%	30	105%	100%	~87%	60%	2	5
12	Bell	3000	100%	~87%	95%	50%	80	100%	40	105%	100%	~87%	60%	1	3
13	Key	5000	100%	~87%	95%	50%	80	100%	40	105%	100%	~87%	60%	1	3
14	Key	5000	100%	~87%	95%	50%	80	100%	40	105%	100%	~87%	60%	3	5
15	Key	5000	100%	~87%	95%	50%	100	100%	50	105%	100%	~87%	60%	1	3
16	Key	5000	100%	~87%	95%	50%	100	100%	50	105%	100%	~87%	60%	1	3
17	Key	5000	100%	~87%	95%	50%	100	100%	50	105%	–	–	–	–	–
18	Key	5000	100%	~87%	95%	50%	100	100%	50	105%	100%	~87%	60%	1	3
19	Key	5000	100%	~87%	95%	50%	120	100%	60	105%	–	–	–	–	–
20	Key	5000	100%	~87%	95%	50%	120	100%	60	105%	–	–	–	–	–
21+	Key	5000	90%	~79%	95%	50%	120	100%	60	105%	–	–	–	–	–

Hardware

The Random Number Generator (RNG) system in Pac-Man plays a crucial role in determining the behavior of the game's elements, such as the movement patterns of the ghosts. The RNG system used in the original 1980 Pac-Man arcade game was an essential part of the game's mechanics and was implemented in assembly language to work efficiently on hardware with limited resources.
The implementation provided in this document simulates the behavior of that original RNG system using modern C++ in the Arcade library, ensuring compatibility with the game's logic while preserving the retro feel of the original design.

RNG Algorithm

The core RNG algorithm used in Pac-Man is based on a linear congruential generator (LCG). The LCG is a simple and efficient method for generating a sequence of pseudo-random numbers. In the case of the original Pac-Man system, the algorithm was designed to be computationally lightweight to fit the hardware constraints of the arcade machine.

The RNG class in the Arcade library mimics the algorithm used in the original Pac-Man arcade game, which can be described as follows:

- Seed Initialization:
  - The RNG begins with a seed value, which can either be set manually or default to `0`.
  - The value of the seed is then constrained to be between `0` and `8191` (`mSeed % 8192`) to ensure it fits within a predefined range.
- Linear Congruential Generator:
  - The RNG operates by applying the formula:
    - `new seed=(seed×5+1) mod 8192`
  - The seed is repeatedly updated using this formula to generate a new pseudo-random value.
- Random Value Generation:
  - After the seed is updated, the new seed value is used as an index into the ROM array (`ROM_6E[]`), which contains precomputed random values.
  - The value at the computed index is then returned as the random number. This value is typically used to influence game behavior, such as determining which direction a ghost will move or triggering certain events.

Conclusion

The Pac-Man RNG system was originally written in assembly language and utilized a simple but effective linear congruential generator (LCG) to produce pseudo-random numbers. This system, though simplistic by modern standards, was crucial to the dynamic behavior of the game and the challenge it posed to players. The Arcade library preserves the core functionality of this RNG system, enabling developers to recreate the original game experience or to experiment with variations of the algorithm in a more modern C++ environment.

Remotes

For the Arcade Platform, support has been added for various joysticks and remotes, allowing players to enjoy games with different types of controllers. Below is the list of supported remotes:

WII Remotes

The Arcade Platform supports the use of up to four Wii Remotes. This feature allows users to play games with their WiiMotes by connecting them via Bluetooth.

How to Connect a WiiMote:

- Press the `W` key on the Arcade Platform to initiate the pairing process.
- On the Wii Remote, press both the `1` and `2` buttons simultaneously to enter pairing mode.
- The Arcade Platform will detect the WiiMote, and once connected, it will be ready for use.

Xbox Controllers

You can use an Xbox controller or any compatible controller by connecting it through the USB port. Once connected, the Arcade Platform will automatically recognize the controller, allowing you to use it for games.

How to Connect an Xbox Controller:

- Plug your Xbox controller into an available `USB` port on your device.
- The Arcade Platform will automatically recognize the controller and map the necessary buttons.
- The controller is now ready to use.

Libraries Used for Remote Connectivity

To facilitate the connection and functionality of remotes, the Arcade Platform uses the following libraries:

- `udev`: A Linux device manager that helps in detecting and managing the hardware connected to the system, including controllers and remotes.
- `wiiuse`: A library that handles the connection and management of Wii Remotes (WiiMotes) via Bluetooth. It enables the platform to interact with the remotes, detect button presses, and handle motion input.
- `bluez`: A Bluetooth protocol stack for Linux. This library allows the Arcade Platform to manage Bluetooth communication, enabling it to pair and communicate with Bluetooth devices such as Wii Remotes.

By using these libraries, the Arcade Platform is capable of seamlessly integrating multiple types of controllers, enhancing the gaming experience for users.

Audio API Documentation

The **Audio API** allows you to manage and play audio within the Arcade platform. Based on miniaudio, a minimalistic audio library, this API provides an easy way to load, play, stop, and manage audio playback. The system supports basic audio features such as looping, managing multiple audio streams, and caching audio data for efficient reuse.

This API is designed to be simple yet powerful, offering flexibility for various use cases where you need to integrate sound effects or background music into your game.

Key Features

- **Audio Playback:** Play audio from file paths, optionally loop the audio, and control playback using unique IDs.
- **Audio Caching:** Cache audio files to optimize performance and prevent unnecessary reloading of audio data.
- **Stream Management:** Support for multiple audio streams, allowing you to manage several simultaneous audio sources.
- **Automatic Cleanup:** Reference counting ensures audio data is automatically cleaned up when no longer in use.

Audio Stream and Data Handling

Audio Data Structure

The **Audio API** uses two primary structures to manage audio data:

- **cached_audio:** Represents the audio data and decoder for a specific audio file. It includes reference counting to handle loading and unloading efficiently.
- **audio_stream:** Represents a currently playing audio stream, including its playback configuration, status, and associated cached data.

Static Caching Mechanism

Audio files are cached to ensure that the data is loaded once and reused across different playback requests. The `cached_audio` structure holds the decoder and maintains a reference count. When no streams are using a particular audio file, it is released from cache.

Audio Streams

Each playing audio file is associated with an audio_stream that holds the playback state. This includes:

- **ID:** A unique identifier for the audio stream.
- **Finished:** A flag indicating whether the audio stream has finished playing.
- **Stop:** A flag to control when to stop the audio stream.
- **Loop:** A flag specifying whether the audio should loop.

Internal Audio Management

The Audio API uses the miniaudio library to manage audio playback. It integrates directly with the `ma_device` system to handle the audio output, managing data buffers and callbacks. When an audio stream is started, it initializes a device and begins playback using the `AudioThread` function. If the stream is finished or stopped, it cleans up by releasing resources and removing the stream from the active streams map.

Additional Notes

- **Thread Management:** Each audio stream runs in a separate thread to ensure non-blocking audio playback.
- **Error Handling:** If the audio file cannot be decoded or played, no audio will be played, and the system will silently fail.
- **Caching:** When audio data is no longer in use, it will be released from memory to prevent excessive memory consumption.

Conclusion

The **Audio API** provides a simple and efficient way to integrate sound into the Arcade platform. It supports managing audio files through caching and playback control, ensuring smooth audio experiences in your games. Whether you're playing background music or sound effects, this API can easily handle multiple streams and ensure optimal performance.