Platform Layer

The Platform Layer is a fundamental component of the Engine, designed to abstract away operating system-specific functionalities. Its primary purpose is to isolate the core engine from the complexities of window management, input handling, file I/O, and other platform-dependent tasks. This allows the engine's code to be written once and run seamlessly across multiple platforms, such as Windows and Linux, with macOS support planned for the future.

By funneling all platform interactions through a single, consistent interface, the engine remains agnostic to the underlying OS it is running on.

Core Concepts: Abstraction and Opaque State

The Interface

The abstraction is achieved through a single header file, `platform.h`, which defines the public-facing interface for the entire platform layer. This file contains function signatures for all platform-related operations but holds no actual implementation code itself.

Opaque State Management

To manage platform-specific data (like a window handle on Windows) without exposing it to the rest of the engine, an opaque pointer technique is used. The `platform_state` struct contains a `void*` pointer, which points to a platform-specific internal state structure defined only within the implementation file (e.g., `platform_win32.c`). This ensures the core engine can manage the platform's state without needing to know any of its internal details.

C

```
/**
 * brief: Holds platform-specific state.
 *
 * This struct uses an opaque void pointer to hide implementation
 * details from the rest of the engine.
 */
typedef struct platform_state {
    void* internal_state;
} platform_state;
```

Platform API: Functions and Usage

The `platform.h` interface exposes a set of functions to handle the application's lifecycle, memory, and other OS interactions.

b8 platform_startup(...)

Initializes the platform, creates a window with a specified size and position, and sets up any necessary internal state.

void platform_shutdown(...)

Cleans up platform resources, including destroying the window.

b8 platform_pump_messages(...)

Processes pending messages from the operating system, such as user input or window events. This must be called continuously within the main application loop.

void* platform_allocate(u64 size, b8 aligned)

Abstracts memory allocation.

void platform_free(void* block, b8 aligned)

Abstracts memory deallocation.

void* platform_*_memory(...)

Provides utility functions for zeroing, copying, and setting blocks of memory.

void platform_console_write(const char* message, u8 colour)

Writes a message to the standard console output stream with a specified color.

void platform_console_write_error(const char* message, u8 colour)

Writes a message to the standard error console output stream with a specified color.

f64 platform_get_absolute_time()

Returns the absolute time in seconds since the system was started, using a high-precision timer.

void platform_sleep(u64 ms)

Pauses the current thread for a specified number of milliseconds.

Windows Implementation: `platform_win32.c`

The Windows-specific implementation of the platform interface resides in `platform_win32.c`. The entire file is wrapped in a preprocessor conditional to ensure it is only compiled when targeting the Windows platform.

C

```
#if KPLATFORM_WINDOWS
// ... Windows-specific implementation ...
#endif // KPLATFORM_WINDOWS
```

Internal State

For Windows, the internal state required includes a handle to the application instance (`HINSTANCE`) and a handle to the main window (`HWND`). This is hidden from the engine via the opaque `void*` pointer in `platform_state`.

C

```
/**
 * brief: Internal state structure for the Win32 platform.
 *
 * Contains handles for the application instance and the main window,
 * which are essential for interacting with the Windows API.
 */
typedef struct internal_state {
    HINSTANCE h_instance;
    HWND hwnd;
} internal_state;
```

Windows Implementation: Windowing & Messages

The implementation of `platform_startup` on Windows involves several steps to create and display a window using the Win32 API.

Window Creation

01

Register Window Class

A `WNDCLASSA` structure is configured with properties like the window procedure callback (`win32_process_message`), icon, and cursor. It is then registered with the OS via `RegisterClassA`.

02

Calculate Window Size

The Win32 API distinguishes between the window's full size (including borders and title bar) and the client area (the drawable space). `AdjustWindowRectEx` is used to calculate the required window size to achieve the desired client area dimensions.

03

Create Window

`CreateWindowExA` is called to create the window handle (`HWND`).

04

Show Window

`ShowWindow` is called to make the window visible on the screen.

Message Pumping

The `platform_pump_messages` function is responsible for keeping the application responsive. It runs a loop that continuously polls for new messages from the OS using `PeekMessageA`. If a message exists, it is translated and dispatched to the appropriate window procedure for handling.

C

```
b8 platform_pump_messages(platform_state* plat_state) {
    MSG message;
    // Process all pending messages in the queue
    while (PeekMessageA(&message, NULL, 0, 0, PM_REMOVE)) {
        TranslateMessage(&message);
        DispatchMessageA(&message); // Dispatches to win32_process_message
    }
    return TRUE;
}
```

Windows Implementation: Message Handling

All messages dispatched by `platform_pump_messages` are sent to a single callback function, known as a "Window Procedure." In this implementation, it is `win32_process_message`.

This function uses a switch statement to handle specific messages that are relevant to the engine.

Key Messages Handled

WM_ERASEBKGND

Handled to prevent the OS from trying to draw the window background, which prevents flickering since the renderer will be drawing over it anyway.

WM_CLOSE

Triggered when the user clicks the 'X' button on the window. An event will be fired for the application to quit gracefully.

WM_DESTROY

Triggered after the window is destroyed. This posts a quit message to terminate the application loop.

WM_SIZE

Occurs when the window is resized. This will be used to notify the renderer that it needs to adjust the viewport.

WM_KEYDOWN / WM_KEYUP

Handle keyboard button presses and releases.

WM_MOUSEMOVE

Tracks mouse movement.

WM_MOUSEWHEEL

Handles mouse wheel scrolling.

WM_*BUTTONDOWN / WM_*BUTTONUP

Handle mouse button presses and releases.

Any messages not explicitly handled by the switch statement are passed to `DefWindowProcA`, which is the default message handler provided by the Windows API.