Application Layer

The Application Layer serves as the central hub of the engine, acting as a crucial bridge between the low-level Platform Layer and the high-level, game-specific code. Its primary role is to manage the main application lifecycle, including the game loop, state management, and the initialization of various engine subsystems.

This abstraction is essential for simplifying game development. It allows game programmers to focus on creating their game's logic without needing to interact directly with platform-specific details like window creation or OS message pumping. The Application Layer provides a clean, consistent interface for the game to plug into, ensuring that the engine's core remains separate from the game itself.

Central Hub

Connects low-level platform with high-level game code.

Lifecycle Management

Manages game loop, state, and subsystem initialization.

Simplifies Development

Abstracts platform details for game programmers.

Core Concepts: The Game Interface

To connect the engine with the game, the Application Layer defines a clear "contract" or interface that any game must adhere to. This is accomplished through a game struct, which is defined in `game_types.h`.

The `game` Struct

This structure is a collection of configuration data and function pointers. The engine uses this struct to call into the game's code at the appropriate times.

```
/**
 * @brief Represents the basic game state.
 * Called for creation by the application.
 */
typedef struct game {
    // The application configuration.
    application_config app_config;
    // Function pointer to the game's initialize function.
    b8 (*_initialize)(struct game* game_inst);
    // Function pointer to the game's update function.
    b8 (*_update)(struct game* game_inst, f32 delta_time);
    // Function pointer to the game's render function.
    b8 (*_render)(struct game* game_inst, f32 delta_time);
    // Function pointer to handle window resizing.
    void (*_on_resize)(struct game* game_inst, u32 width, u32 height);
    // Game-specific state. Created and managed by the game.
    void* state;
} game;
```

application_config

A struct holding startup settings like window size and title.

Function Pointers

These are the heart of the interface. The engine will call these pointers to run the game's initialization, per-frame updates, rendering logic, and resize handling.

void* state

An opaque pointer for the game's internal state. This allows the game to manage its own memory and state privately.

Internal Application State

The Application Layer also maintains its own private, static state to manage the application's lifecycle. This state is not exposed to the game.

```
typedef struct application_state {
    game* game_inst;
    b8 is_running;
    b8 is_suspended;
    platform_state platform;
    i16 width;
    i16 height;
    f64 last_time;
} application_state;
```

This struct tracks whether the main loop should continue (`is_running`), if the application is paused (`is_suspended`), and holds the instance of the platform state.

The Engine's Entry Point

A key architectural decision is to place the application's main entry point (`main` function) inside the engine library itself. This frees the game programmer from writing boilerplate startup code. The engine's entry point is defined in `entry.h`.

This is made possible by a single, externally-defined function that the game must provide:

```
// Externally-defined function to create a game.
extern b8 create_game(game* out_game);
```

The `extern` keyword tells the engine's compiler that the implementation for `create_game` exists elsewhere—specifically, in the final game executable that links against the engine.

The `main` Function

The engine's `main` function orchestrates the entire startup and shutdown sequence.

```
/**
 * @brief The main entry point of the application.
 */
int main(void) {
    // Request the game instance from the application.
    game game_inst;
    if (!create_game(&game_inst)) {
        KERROR("Could not create game!");
        return -1;
    }
    // Ensure the function pointers exist.
    if (!game_inst.render || !game_inst.update || !game_inst.initialize || !game_inst.on_resize) {
        KERROR("The game's function pointers must be assigned!");
        return -2;
    }
    // Initialize the application layer.
    if (!application_create(&game_inst)) {
        KINFO("Application failed to create!");
        return 1;
    }
    // Begin the game loop.
    if (!application_run()) {
        KINFO("Application did not shutdown gracefully!");
        return 2;
    }
    return 0;
}
```

The lifecycle is simple and robust:

01

Create Game

Calls the external `create_game` function, populating the `game_inst` struct.

02

Validate

Ensures all required function pointers have been assigned by the game.

03

Create Application

Calls `application_create` to initialize subsystems and the platform layer.

04

Run Application

Calls `application_run` to start the main game loop, blocking until the user quits.

Application Lifecycle Functions

The core logic of the Application Layer is split into two main functions: application_create and application_run.

application_create()

This function handles the one-time initialization of the engine and the game.

```
b8 application_create(game* game_inst) {
    if (initialized) {
        KERROR("application_create called more than once.");
        return FALSE;
    }

    app_state.game_inst = game_inst;
    // Initialize subsystems.
    initialize_logging();
    app_state.is_running = TRUE;
    app_state.is_suspended = FALSE;
    // Initialize the platform layer.
    if (!platform_startup(&app_state.platform, ...)) {
        return FALSE;
    }
    // Call the game's initialize function.
    if (!game_inst->initialize(game_inst)) {
        KFATAL("Game failed to initialize.");
        return FALSE;
    }
    initialized = TRUE;
    return TRUE;
}
```

Its responsibilities are:

- → Singleton Check
  Ensures it is only called once.
- → Initialize Subsystems
  Kicks off core systems like the logger.
- → Platform Startup
  Calls `platform_startup` to create a window using the game's configuration.
- → Game Initialization
  Calls the game's `initialize` function pointer, allowing the game to set up its initial state.

application_run()

This function contains the main game loop and runs until the application is signaled to quit.

```
b8 application_run() {
    while (app_state.is_running) {
        // Process OS messages.
        if(!platform_pump_messages(&app_state.platform)) {
            app_state.is_running = FALSE;
        }

        // If the application is not suspended, run the game's frame logic.
        if (!app_state.is_suspended) {
            // Call the game's update routine.
            if (!app_state.game_inst->update(app_state.game_inst, 0.0f)) {
                app_state.is_running = FALSE;
            }
            // Call the game's render routine.
            if (!app_state.game_inst->render(app_state.game_inst, 0.0f)) {
                app_state.is_running = FALSE;
            }
        }
    }
    // Shut down the platform layer.
    platform_shutdown(&app_state.platform);
    return TRUE;
}
```

Each iteration of the loop performs these key actions:

- Pump Messages
  Calls `platform_pump_messages` to process window events and user input. If a quit signal is received, the loop terminates.
- Update
  Calls the game's `update` function pointer, where game logic (e.g., physics, AI) should be handled.
- Render
  Calls the game's `render` function pointer to draw the new frame.

Once the loop exits, `platform_shutdown` is called to clean up all platform resources.

The Game's Responsibility

The final piece of the puzzle is the user-provided code, which typically resides in the "testbed" or game project. This code must fulfill the contract defined by the engine.

Defining the Game (game.h & game.c)

First, the game defines its functions and its internal state struct.

game.h

```
#pragma once

#include <defines.h>
#include <game_types.h>

// The game's internal state struct.
typedef struct game_state {
    f32 delta_time;
} game_state;

// Function prototypes that will be assigned to the game struct.
b8 game_initialize(game* game_inst);
b8 game_update(game* game_inst, f32 delta_time);
b8 game_render(game* game_inst, f32 delta_time);
void game_on_resize(game* game_inst, u32 width, u32 height);
```

game.c

```
#include "game.h"
#include <core/logger.h>

// Implementation of the game's functions.
b8 game_initialize(game* game_inst) {
    KDEBUG("game_initialize() called");
    return TRUE;
}

// other function implementations...
```

Implementing create_game

Finally, the game project must provide the implementation for the create_game function in its own entry file (e.g., entry.c). This is the function that the engine's main will call.

```
#include "game.h"
#include <entry.h>
#include <platform/platform.h>

// Define the function to create a game.
b8 create_game(game* out_game) {
    // 1. Application configuration.
    out_game->app_config.start_pos_x = 100;
    out_game->app_config.start_width = 1280;
    out_game->app_config.name = "Kaffi Engine Testbed";

    // 2. Assign function pointers.
    out_game->update = game_update;
    out_game->render = game_render;
    out_game->initialize = game_initialize;
    out_game->on_resize = game_on_resize;
    // 3. Create the game state.
    out_game->state = platform_allocate(sizeof(game_state), FALSE);
    return TRUE;
}
```

This function is responsible for:

Setting Configuration
Filling out the app_config with desired window settings.

Assigning Pointers
Pointing the engine to the game's own update, render, and initialize functions.

Allocating State
Reserving memory for its internal game_state and assigning the pointer to out_game->state.