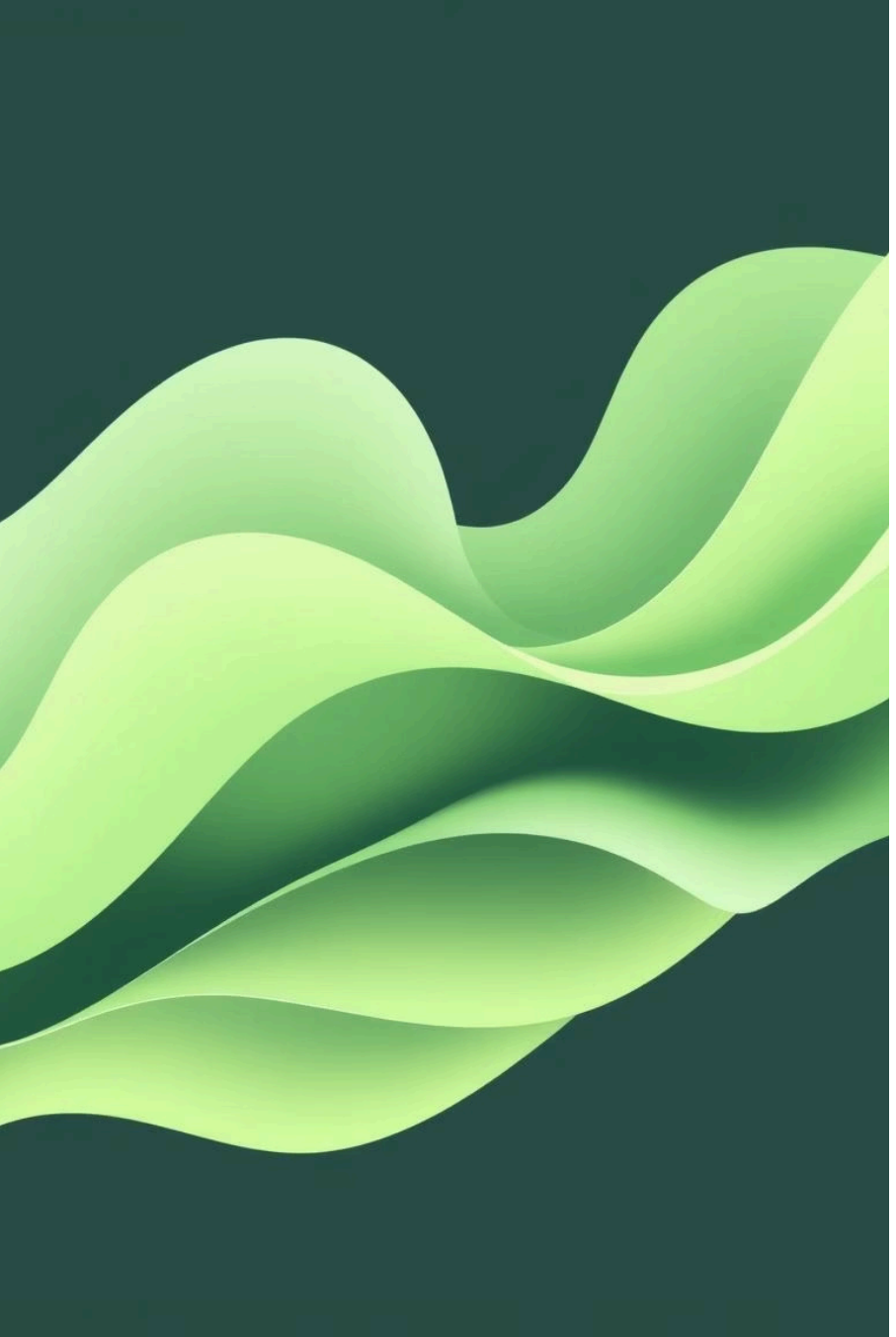

Assertion System

Assertions are a critical debugging tool that validates assumptions made in the code at runtime. The Engine provides a simple yet powerful assertion system that halts execution immediately when a condition (expression) evaluates to false.

This allows developers to catch logic errors and invalid program states early in the development process. Like the logging system, assertions can be completely disabled for release builds to ensure they have no performance impact on the final product.

Features and Configuration

The entire assertion system can be enabled or disabled globally with a single preprocessor definition in asserts.h.

To enable assertions, ensure the `KASSERTIONS_ENABLED` macro is defined. To disable all assertions, simply comment out or remove this line.

```
// Disable assertions by commenting out the below line
#define KASSERTIONS_ENABLED
```

When disabled, all assertion macros compile to nothing, effectively removing them from the code and guaranteeing zero runtime overhead.

API and Usage

The assertion system provides three macros for different use cases.

- **KASSERT(expression)**
  - The most common assertion. It takes a boolean expression. If the expression is false, it logs a fatal error with the expression, file, and line number, and then halts the application.
- **KASSERT_MSG(expression, message)**
  - Identical to KASSERT, but also accepts a custom string message. This message is included in the fatal log output, providing additional context about the failure.
- **KASSERT_DEBUG(expression)**
  - A special assertion that is only active in debug builds (_DEBUG is defined). In release builds, this macro compiles to nothing, making it ideal for expensive checks that are only needed during development.

```
// Halts if 1 does not equal 1 (it won't)
KASSERT(1 == 1);

// Halts and provides a custom message if a pointer is null
KASSERT_MSG(ptr != 0, "Pointer 'ptr' should not be null here.");
```

Implementation Details

The assertion system relies on two key implementation details: a platform-agnostic debug break and an assertion failure reporting function.

Debug Break Macro

To halt execution, a `debugBreak()` macro is defined. It abstracts the compiler-specific intrinsic function for triggering a breakpoint:

- **MSVC (_MSC_VER)**: Uses `__debugbreak()`.
- **GCC/Clang**: Uses `__builtin_trap()`.

```
#if _MSC_VER
    #include <intrin.h>
    #define debugBreak() __debugbreak()
#else
    #define debugBreak() __builtin_trap()
#endif
```

Assertion Failure Reporting

When an assertion fails, it calls `report_assertion_failure()`. This function is implemented in `logger.c` to leverage the existing logging system. It formats a detailed message containing the failed expression, the custom message (if any), the source file, and the line number, and then logs it as a `FATAL` error before `debugBreak()` is called.

```
void report_assertion_failure(const char* expression, const char* message, const char* file, i32 line)
{
    log_output(LOG_LEVEL_FATAL, "Assertion Failure: %s, message: '%s', in file: %s, line: %d\n", expression, message, file, line);
}
```