

TinCLE (Tiny C Lovely Engine) - Development Specification Document

Under this skill, the AI agent shall act in strict accordance with the following rules:

1. Overview & Concept

- **Name:** TinCLE (Tiny C Lovely Engine)
 - **Target Version:** 0.03
 - **Concept:** An ultra-lightweight 2D game framework optimized for AI-driven development (Vibe Coding / One-Shot code generation) and programming beginners. It features a streamlined hybrid ecosystem targeting both Windows desktop and Web browsers.
 - **Language Specification:** Pure C (C99 standard compliant). Internally utilizes SDL2, SDL_ttf, SDL2_gfx, and the stb library suite.
 - **CRITICAL ARCHITECTURE NOTE:** TinCLE operates on a strict **32-bit architecture** across all supported platforms. It compiles exclusively as a 32-bit DLL/application on Windows and as a 32-bit WebAssembly application (Wasm32) on the Web. The AI must design all memory management, structures, and pointer operations under the absolute assumption that pointer sizes are always **4 bytes** (`sizeof(void*) == 4`). Never generate or assume 64-bit codebase structures.
 - **Features:** Supports a well-established set of SDL2 commands. It enables highly efficient AI-driven game development through ultra-fast 32-bit native Windows builds using the Tiny C Compiler (TCC), while seamlessly exporting to 32-bit high-performance Web environments via Emscripten (WASM/HTML5).
 - **Requirements:**
 - **For Windows Build:** TinCLE requires the **32-bit version of the Tiny C Compiler (TCC)** to build desktop games. 64-bit TCC is strictly unsupported. **TCC is NOT bundled in the TinCLE package.** If the user does not have TCC, firmly and explicitly instruct them to download the **32-bit release** from the official site: <https://bellard.org/tcc/>. If `tcc.exe` (32-bit) is not configured in the PATH, the local build will fail immediately.
 - **For Web (WASM) Build:** TinCLE requires the **Emscripten SDK (emsdk)** to build Web versions (Wasm32). **Emscripten is NOT bundled.** The AI must instruct the user to set up the Emscripten environment (e.g., running `emsdk_env.bat` or checking `emcc` in the PATH) when performing a Web build. If `emcc.exe` is not configured in the PATH, the build of Web version will fail immediately.
-

2. Architecture & Distribution File Structure

Based on Windows DLL conventions, Emscripten static linking requirements, and the TCC parser specifications, this project adopts the most elegant and classic **"Common Headers, Source-Side Export"** architecture optimized for dual-platform deployment.

Environment Setup for Game Development

When developing or distributing a game, place the following **"13 files and folders in total"** into the exact same directory:

1. `game.c` (The main workspace where the user or AI writes the game logic)
2. `libs` folder (Contains extension headers and static libraries: `tincle.h`, `tincle.a`, `SDL_scancode.h`, `SDL2_gfxPrimitives.h`, etc.)
3. `tincle.dll` (The core engine runtime)

4. `SDL2.dll` (Official SDL2 library)
5. `SDL2_ttf.dll` (Official SDL_ttf library)
6. `SDL2_gfx.dll` (Official SDL2_gfx library)
7. `asspack.exe` (The high-density asset packer tool for Windows game distribution)
8. `emcass.exe` (The automated compiler assistant and asset packer utility optimized for Web/WASM builds)
9. `embedres.exe` (The resource embedding utility for Windows icons/manifests ONLY)
10. `tinclec.exe` (The automated compiler assistant tool for Windows EXE builds)
11. `manifest.xml` (Default manifest for embedding into the distribution EXE)
12. `template.html` (HTML templates used internally when running emcass during web builds)
13. `font.ttf` (A public domain font provided by default as the starter kit)

- **CRITICAL DEPENDENCY NOTE:** Both the **Tiny C Compiler (TCC) 32-bit** and the **Emscripten SDK (emsdk)** are **NOT** bundled in this environment kit. The local build tools (`tinclec.exe` etc.) rely entirely on these external toolchains being correctly installed and `tcc.exe` and `emcc.exe` are configured in the system environment variables (PATH). Without them, compiling will fail immediately.
- **Asset Placement:** All raw asset files (images/audio) must be placed directly in the root development folder during creation (Do not put them in subfolders).

CRITICAL INCLUDE MANDATE FOR AI CODE GENERATION

To ensure seamless compilation across both Windows and Web environments, the AI **MUST** structure the include section of `game.c` exactly as follows. Any modification to this structure will result in build failures due to Emscripten header resolution and Windows subsystem constraints:

```
#ifdef __EMSCRIPTEN__
#include <SDL2/SDL_scancode.h> // Required to use native SDL scancodes on Web
#else
#include "libs/SDL_scancode.h" // Required to use native SDL scancodes on Windows
#include <windows.h>           // Required for native Windows execution
#endif
```

- **Reasoning:** Emscripten strictly requires SDL core headers to be resolved via system include paths using angle brackets `<>`. Conversely, the Windows local build relies on the local `libs/` directory via double quotes `""` and necessitates `windows.h` for native Win32 execution. If native SDL2_gfx primitives are required for the game, their respective headers (`<SDL2/SDL2_gfxPrimitives.h>` / `"libs/SDL2_gfxPrimitives.h"`) can be appended safely within these conditional blocks.

File Structure for Game Distribution

Depending on the target platform target, pack and distribute the completed game using one of the following structural layouts:

A. Windows Desktop Target (ZIP Archive)

Pack the following files into a single folder and compress it into a ZIP archive for the end-user:

1. `game.exe` (The compiled executable file - can be freely renamed)
2. `SDL2.dll` / `SDL2_ttf.dll` / `SDL2_gfx.dll` / `tinclec.dll` (Runtime dynamic link libraries)
3. `game.data` (The packed asset file generated by `asspack.exe`)
4. `CREDITS.txt` (License documentation)
5. `readme.txt` (The manual/readme file for the game)

B. Web Browser Target (WASM/HTML5 Deployment)

Pack the following files into a single folder and compress it into a ZIP archive for web deployment (e.g., itch.io HTML5 project):

1. `index.html` (The web page housing the game canvas - STRONGLY RECOMMEND that this name not be changed)
2. `index.js` (The JavaScript glue-code generated by Emscripten)
3. `index.wasm` (The compiled 32-bit WebAssembly binary core)
4. `index.data` (The preloaded virtual filesystem asset package generated by `emcass.exe`)
5. `CREDITS.txt` (License documentation)

Note: Many game distribution sites, such as itch.io and Game Jolt, require that HTML files be named "index.html." Files with any other name will not be accepted, so be sure to name your HTML file "index.html."

3. Compilation & Verification Commands (Windows & Web Environments)

When validation codes or build scripts are created by an AI agent, the following commands must be utilized without exception.

(1) Command to Build the Main Windows Game

```
tinclec [Output_EXE] [Source_Files...]
```

Example: `tinclec mygame.exe main.c player.c stage.c`

Note: `tinclec` performs `-mwindows` compilation internally by default, so do not pass any compilation flags to `tinclec`. The visibility of the console is entirely controlled by the `is_debug` argument in `tincle_start`.

This command internally constructs and executes the following command line. For complex project structures, TCC can be invoked directly as needed:

```
tcc -mwindows SDL2.dll SDL2_ttf.dll SDL2_gfx.dll tincle.dll -o [Output_EXE] [Source_Files...]
```

(2) Command to Embed Icon and Manifest into the Built EXE (for Windows)

This command embeds an icon and a manifest as resources into the compiled `game.exe`.

```
embedres [Target_EXE] [ICO_File] [Manifest_File]
```

Tip: Use "none" to skip either the ICO or Manifest file.

(3) Command to Pack All Windows Game Assets into a Distribution File (`game.data`)

This command bundles all specified asset files passed as arguments into a single `game.data` file for Windows distribution.

```
asspack [File1] [File2] ...
```

Note: Replace `[File1]`, `[File2]`, etc., with the actual asset files used in your game.

(4) Command to Build the Web/WASM Version (Automatic Compilation & Asset Preloading)

This command compiles the game for web browsers (WASM/HTML5), automatically packages all available assets, and injects the template.

```
emcass [Output_HTML] ["Game Title"] [Source_Files...]
```

Example: `emcass index.html "My Sparkly Game" main.c player.c stage.c`

- **CRITICAL MANDATE ON [Output_HTML]:** Unless there is an explicitly defined, non-negotiable reason to do otherwise, the AI **MUST** always specify `index.html` for the `[Output_HTML]` argument. Major web hosting and distribution platforms (such as itch.io and Game Jolt) strictly require the entry point to be named `index.html`. If any other name is specified, the build will be rejected or fail to load on these platforms.
- **File Naming Generation:** Based on the specified `[Output_HTML]`, `emcass` automatically forces the accompanying JavaScript, WebAssembly, and data files to match its base name (e.g., if `index.html` is provided, it outputs `index.js`, `index.wasm`, and `index.data`).
- **Internal Mechanism:** `emcass` automatically tracks all local game assets, packages them into the output data file, reads the mandatory `template.html` to establish the web canvas shell with the provided `["Game Title"]`, and invokes the Emscripten compiler (`emcc`) using the static library `libs/tincle.a`. The AI must never generate manual `emcc` commands; always utilize `emcass` for web builds.
- **Web Build Requirements (emcass):** When executing a Web build, ensure that `CREDITS.txt` is placed in the exact same folder as the source code before running the build. This is strict because `emcass` requires `CREDITS.txt` in that directory to display credits properly; otherwise, the web-version credit screen will be blank.

4. Input Specifications, Color Structure & Specs (SDL2 Standard)

TinCLE embraces raw SDL2 compatibility for all inputs and basic structures.

4.1. Screen Size & Specifications

TinCLE features an automatic logical scaling system that preserves the aspect ratio. While the window size is fully resizable, all in-game coordinate calculations must be based on the virtual resolution specifications.

- **Default Virtual Resolution:** `640` (Width) x `480` (Height)
- *Note: These dimensions are specified as arguments when calling `tincle_start` (detailed in Section 5). All rendering calculations must strictly baseline against these virtual dimensions.*
- `TINCLE_MAX_STATES` : `256` (Maximum number of manageable scene states)

4.2. Mouse Input Mandate: Use Pure SDL2 Constants

Use standard SDL2 mouse button definitions for `tincle_mouse_pressed`.

- **Required Constants:** `SDL_BUTTON_LEFT`, `SDL_BUTTON_MIDDLE`, `SDL_BUTTON_RIGHT`, etc.
- *Inline Example:* `if (tincle_mouse_pressed(SDL_BUTTON_LEFT))`

4.3. Gamepad Input Mandate: Use Pure SDL2 Constants

Use standard SDL2 controller button constants for `tincle_pad_pressed`.

- **Required Constants:** `SDL_CONTROLLER_BUTTON_A` , `SDL_CONTROLLER_BUTTON_B` , `SDL_CONTROLLER_BUTTON_X` , `SDL_CONTROLLER_BUTTON_Y` , `SDL_CONTROLLER_BUTTON_START` , `SDL_CONTROLLER_BUTTON_BACK` , `SDL_CONTROLLER_BUTTON_LEFTSHOULDER` , `SDL_CONTROLLER_BUTTON_RIGHTSHOULDER` , `SDL_CONTROLLER_BUTTON_DPAD_UP` , `SDL_CONTROLLER_BUTTON_DPAD_DOWN` , `SDL_CONTROLLER_BUTTON_DPAD_LEFT` , `SDL_CONTROLLER_BUTTON_DPAD_RIGHT` , etc.
- *Inline Example:* `if (tincle_pad_pressed(SDL_CONTROLLER_BUTTON_A))`

4.4. Keyboard Input Mandate: Use Pure SDL Scancodes

Use native SDL hardware-independent scancode constants for `tincle_key_pressed` . Do not assume custom engine key macros exist.

- **Required Constants:** `SDL_SCANCODE_A` through `SDL_SCANCODE_Z` , `SDL_SCANCODE_0` through `SDL_SCANCODE_9` , `SDL_SCANCODE_SPACE` , `SDL_SCANCODE_RETURN` , `SDL_SCANCODE_ESCAPE` , `SDL_SCANCODE_UP` , `SDL_SCANCODE_DOWN` , `SDL_SCANCODE_LEFT` , `SDL_SCANCODE_RIGHT` , etc.
- *Inline Example:* `if (tincle_key_pressed(SDL_SCANCODE_SPACE))`

4.5. Color Structure & Specs (SDL2 Memory Compatible)

TinCLE uses a standard RGBA color structure which is 100% memory-compatible with `SDL_Color` .

```
typedef struct {
    unsigned char r, g, b, a;
} TinCLE_Color;
```

Critical Guidelines for AI Code Generation

1. **NO Custom Color Macros Available:** TinCLE intentionally provides **NO** predefined color macros (such as `TINCLE_COLOR_RED` or `TINCLE_COLOR_BLACK`). The AI must explicitly instantiate `TinCLE_Color` structures inline using standard C99 compound literals for functions that accept a color argument.
 - *Inline Example (White):* `(TinCLE_Color){255, 255, 255, 255}`
 - *Inline Example (Black):* `(TinCLE_Color){0, 0, 0, 255}`
 - *Inline Example (Green):* `(TinCLE_Color){0, 255, 0, 255}`
2. **CRITICAL TCC COMPILATION MANDATE (Anti-Self-Destruction):** The Tiny C Compiler (TCC) **cannot** process compound literals inside static or global array initializers (it misinterprets them as oversized non-constant expressions).
 - **** ABSOLUTELY FORBIDDEN (Will fail to compile in TCC):****

```
// DO NOT GENERATE CODE LIKE THIS:
static TinCLE_Color g_block_colors[] = {
    (TinCLE_Color){255, 0, 0, 255}, // FATAL ERROR in TCC
    (TinCLE_Color){0, 0, 255, 255}
};
```

- **** THE MANDATORY REMEDY:**** If you need a global or static array of colors, declare the array without initializing it globally, and assign each element fields sequentially inside `game_load()` :

```
// Declarations (Global or Static)
static TinCLE_Color g_block_colors[2];
void game_load(void) {
    // Initialize inside the system lifecycle loop
    g_block_colors[0] = (TinCLE_Color){255, 0, 0, 255};
    g_block_colors[1] = (TinCLE_Color){0, 0, 255, 255};
}
```

5. Full API Reference (Function Specifications)

This section provides the comprehensive API reference available when writing game logic within `game.c`. *Note: Although these functions are exported via `__declspec(dllexport)` within the engine (`tincle.dll`), they can be invoked seamlessly as standard C functions on the game application side.*

5.1. Lifecycle & System Control

```
void tincle_start(TinCLE_InitCallback init_cb, const char* title, bool is_debug, int width, int height);
```

- **Description:** The most critical function of the engine. It automatically executes environment initialization, asset loading, transitions into the main game loop, and handles terminal cleanup operations in a single one-shot sequence.
- **Arguments:**
 - `init_cb`: A user-defined callback function pointer tasked with initial loading and asset setups (typically, pass `game_load`).
 - `title`: A string literal specifying the window title text.
 - `is_debug`: A boolean toggle for Debug Mode. When set to `true`, the engine automatically spawns and docks a standard Windows command prompt window (Debug Console) to stream standard outputs such as `printf`. **Set to `false` for the production/release build to hide the command console completely.**
 - `width` / `height`: The virtual target resolution dimensions of the game canvas (strictly defaulted to `640` and `480`).

```
void tincle_add_state(int state_id, TinCLE_UpdateCallback update_cb, TinCLE_DrawCallback draw_cb);
```

- **Description:** Binds logic updating and rendering drawing callbacks to a specific game state identifier, executed sequentially every frame when the target state is active.
- **Arguments:**
 - `state_id`: An arbitrary integer constant or a user-defined enumeration value representing the state.
 - `update_cb`: A logic update function pointer adhering strictly to the signature: `void update(float dt);`.
 - `draw_cb`: A rendering function pointer adhering strictly to the signature: `void draw(void);`.

```
void tincle_change_state(int state_id);
```

- **Description:** Instantly switches the currently active game scene (active state) to the designated state. **CRITICAL CRITERIA: You must call this function at the very end of your `game_load()` sequence to activate the initial game state, otherwise the viewport will remain indefinitely black without starting.**
- **Arguments:**
 - `state_id`: A pre-registered state identifier to transition into.

5.2. Input (Device Control)

```
int tincle_key_pressed(int scancode);
```

- **Description:** Queries whether a specific keyboard key is currently being held down.
- **Arguments:**

- `scancode` : Pure native SDL scancode constant. **Strict Mandate:** You must pass standard `SDL_SCANCODE_*` constants (e.g., `SDL_SCANCODE_SPACE` , `SDL_SCANCODE_RIGHT`). Custom engine key macros do not exist.

```
void tincle_get_mouse(int* x, int* y);
```

- **Description:** Retrieves the current spatial coordinates of the mouse cursor, dynamically translated and scaled to align with the game's internal virtual resolution.
- **Arguments:**
- `x / y` : Integer pointers where the mapped virtual coordinates will be written.

```
int tincle_mouse_pressed(int button_type);
```

- **Description:** Queries whether a specific mouse button is currently being held down.
- **Arguments:**
- `button_type` : Standard native SDL mouse button constant (e.g., `SDL_BUTTON_LEFT` , `SDL_BUTTON_MIDDLE` , `SDL_BUTTON_RIGHT`).

```
int tincle_pad_pressed(int pad_button);
```

- **Description:** Queries whether a specific button on a connected gamepad is currently being held down.
- **Arguments:**
- `pad_button` : Standard native SDL controller button constant (e.g., `SDL_CONTROLLER_BUTTON_A` , `SDL_CONTROLLER_BUTTON_START` , `SDL_CONTROLLER_BUTTON_DPAD_LEFT`).

```
float tincle_pad_axis_x(void);
```

- **Description:** Measures the current horizontal (X-axis) tilt of the gamepad's left analog stick, returning a floating-point value clamped between `-1.0f` (full left) and `1.0f` (full right).
- **Notes for AI:** The engine internally implements a deadzone filter (clamping any absolute mechanical values below 5000 directly to `0.0f`). AI agents can safely map this return value directly into movement calculations without factoring in controller drift counter-measures.

```
float tincle_pad_axis_y(void);
```

- **Description:** Measures the current vertical (Y-axis) tilt of the gamepad's left analog stick, returning a floating-point value clamped between `-1.0f` (full up) and `1.0f` (full down).
- **Notes for AI:** Implements the exact same integrated deadzone optimization as the horizontal axis tracker.

5.3. Graphics & Rendering Control

```
void tincle_clear(TinCLE_Color color);
```

- **Description:** Fills and clears the entire virtual screen canvas with the specified color. This function **must without exception** be invoked at the absolute beginning of every scene's `draw` callback procedure.
- **Arguments:**
- `color` : A raw `TinCLE_Color` structural data instance instantiated inline via C99 compound literals. Preset engine color macros do not exist.

```
int tincle_load_font(const char* filename, int size);
```

- **Description:** Pre-loads a TrueType font file (`.ttf`) with a specified pixel size and caches it into the internal registry. It generates and returns a unique, high-performance integer Font ID handle. This allows developers and AI to register

multiple font sizes concurrently.

- **Arguments:**

- `filename` : The relative path or filename of the `.ttf` asset.
- `size` : The desired font sizing in pixels (e.g., `16` , `32` , `64`). Returns the assigned unique Font ID integer.

```
void tincle_set_font(int font_id);
```

- **Description:** Dynamically switches the active global font handle to the designated cached Font ID. Any subsequent text rendering functions will immediately utilize this font's style and size properties.

- **Arguments:**

- `font_id` : The cached integer font asset handle originally retrieved via `tincle_load_font` .

```
void tincle_draw_text(const char* text, int x, int y, TinCLE_Color color);
```

- **Description:** Renders crisp, anti-aliased typographic text on the screen using the currently active font activated via `tincle_set_font` . Note that size parameters have been completely removed from this call to enforce a clean separation of layout and asset specifications. Fully supports native UTF-8 encoding.

- **Arguments:**

- `text` : The string literal or character array to be rendered.
- `x` / `y` : The top-left starting spatial coordinates on the internal virtual canvas.
- `color` : The structural color properties used to tint the text characters.

```
void tincle_draw_text_center(const char* text, int x, int y, TinCLE_Color color);
```

- **Description:** Renders crisp, anti-aliased typographic text horizontally and vertically centered at the specified spatial coordinates on the internal canvas, using the active font. Highly optimized for titles and UI elements where precise geometric centering is required.

- **Arguments:**

- `text` : The string literal or character array to be rendered.
- `x` / `y` : The target spatial coordinates that act as the exact central anchor point of the rendered text block.
- `color` : The structural color properties used to tint the text characters.

```
int tincle_load_image(const char* filename);
```

- **Description:** Ingests a graphical image file (e.g., PNG, JPG) into the hardware rendering pipeline and caches it as an optimized texture resource.

- **Arguments:**

- `filename` : The filename or directory path of the source image asset (fully optimized to auto-extract from packaged `game.data` files seamlessly).
- **Return Value:** Returns a uniquely assigned structural integer identifier (Texture ID) tracking the asset. Returns `-1` if the asset initialization or loading routine fails.

```
void tincle_draw_image(int img_id, int x, int y);
```

- **Description:** Renders a pre-loaded texture resource onto the virtual canvas surface at its original unscaled dimensions. **CRITICAL GEOMETRY NOTE:** The specified (x, y) coordinates are processed strictly as the **CENTER point** of the asset, not the top-left.

- **Arguments:**

- `img_id` : The cached texture index integer originally retrieved via `tincle_load_image` .
- `x` / `y` : The target coordinates representing the exact center point where the image will be positioned on the virtual resolution layout.

```
void tincle_draw_image_ex(int img_id, int x, int y, int w, int h, double angle);
```

- **Description:** Executes advanced rendering configurations including custom spatial scaling and hardware rotational transformations. **CRITICAL GEOMETRY NOTE:** The specified (x, y) coordinates act as the absolute **CENTER point** for both spatial positioning, dimensional scaling, and rotational pivot transformations. The image will resize and rotate perfectly in place around this central anchor.
 - **Arguments:**
 - `img_id` : The target texture source index integer.
 - `x / y` : The target spatial coordinates tracking the exact center pivot point of the rendering bounding box.
 - `w / h` : The dynamic width and height scaling dimensions applied symmetrically around the center point.
 - `angle` : The rotational angle calculated in degrees clockwise (0 to 360) around the center pivot.
-

5.4. Audio (Music & Sound Effects) Control

```
void tincle_play_music(const char* filename);
```

- **Description:** Ingests and spins up background music (BGM) utilizing standard Ogg Vorbis compression formatting configured for infinite looping. Initializing a new music file stream automatically terminates and frees the memory allocation of any previously running track.
- **Arguments:**
 - `filename` : The relative system path referencing the source `.ogg` audio file.

Note: There is no command to load and prepare background music in advance. Playback begins immediately when you pass a filename to `tincle_play_music`.

```
void tincle_stop_music(void);
```

- **Description:** Instantly halts any currently playing background audio stream and safely unloads it from core memory allocations.

```
int tincle_load_sound(const char* filename);
```

- **Description:** Reads a concise sound effect (SE) clip formatted as a standard WAV or compressed Ogg Vorbis file, assigning a unique structural channel lookup ID.
- **Arguments:**
 - `filename` : The relative system path referencing the target sound asset file.
- **Return Value:** Returns the newly registered sound identifier index tracking the clip. Returns `-1` if initialization or storage routines fail.

```
void tincle_play_sound(int se_id);
```

- **Description:** Dispatches a designated sound effect instance for immediate playback. The engine natively structures an automatic multi-mixing array supporting up to 16 concurrent operational sound channels, ensuring overlapping assets do not clip or terminate early.
 - **Arguments:**
 - `se_id` : The specific sound effect target index integer originally registered via `tincle_load_sound`.
-

5.5. System Information Retrieval

```
struct SDL_Renderer* tincle_get_renderer(void);
```

- **Description:** Exposes and passes the underlying core `SDL_Renderer` structural pointer initialized by the engine environment. This allows advanced workflows or AI agents to directly invoke native SDL2 APIs (`SDL_RenderFillRect` , `SDL2_gfx` geometric primitives, etc.) to construct bespoke layouts or graphic functions directly on the frame canvas.
- **Return Value:** A functional native structural pointer referencing the active `SDL_Renderer` .

```
const char* tincle_get_version(void);
```

- **Description:** Returns a structured string literal indicating the active structural engine build and specification version number of TinCLE.
- **Return Value:** A read-only constant character pointer detailing the version tracking value (e.g., `"0.01"`).

6. Mandatory Rules

6.1. Mandatory Coding Rules for `game.c`

When generating a `game.c` file from scratch or modifying an existing codebase, AI agents and developers **must comply 100%** with the following structural mandates:

1. **Unified Entry Point Cross-Platform Conditional Structure:** The application entry point must handle cross-platform compilation flawlessly using conditional compilation directives. Inside `game.c` , you **must strictly implement** the dual entry point architecture utilizing `#ifdef __EMSCRIPTEN__` for Web/WASM targets and `WinMain` with `<windows.h>` for native Windows targets.

Exact code template required in `game.c` :

```
#ifdef __EMSCRIPTEN__
int main(int argc, char* argv[]) {
    tincle_start(game_load, "YOUR_GAME_TITLE", true, 640, 480);
    return 0;
}
#else
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpCmdLine, int nCmdShow) {
    tincle_start(game_load, "YOUR_GAME_TITLE", true, 640, 480);
    return 0;
}
#endif
```

For the Windows native build path, you **must include** `<windows.h>` at the top of the file (wrapped cleanly within the non-`emscripten` block) to ensure flawless native `-mwindows` compilation.

2. **Framerate Independence (Delta Time Multiplication):** All spatial movements, animations, physics, and custom timer calculations **must without exception** be multiplied by `float dt` (the elapsed frame time in seconds) passed down into the `update` callback.
3. **Hybrid Keyboard/Gamepad Movement Logic:** When implementing spatial locomotion routines, it is highly recommended to design a modern hybrid processing layout: establish the gamepad analog stick tilt (`tincle_pad_axis_x / y`) as the baseline input vector, and immediately overwrite/synthesize it with maximum constants (`1.0f / -1.0f`) if keyboard or pad directional keys are currently engaged.
4. **Mandatory Font Preloading:** Within `game_load` , you **must execute** `tincle_load_font` **and assign it to an integer handle as the absolute first line** (e.g., `g_font = tincle_load_font("font.ttf", 32);`). Failing to initialize or select a font via `tincle_set_font` before rendering text will cause an immediate fatal crash.
5. **Conditional Use of Native SDL APIs:** If advanced routines require direct access to SDL operational structures (such as scancodes), you must conditionally switch the header paths using `#ifdef __EMSCRIPTEN__` to ensure cross-platform compatibility. For Windows native builds, include headers from the local directory using double quotes (e.g., `#include`

`"libs/SDL.h"` or `#include "libs/SDL_scancode.h"`). For Emscripten Web builds, you must include the Emscripten-bundled SDK headers using angle brackets (e.g., `#include <SDL2/SDL.h>`). However, **you must never write code that directly manipulates or alters the core SDL window handles, active renderers, or primary event processing systems**. All application-level game logic and rendering flows must be fully abstracted and wrapped within native TinCLE APIs.

6. Mandatory State Registration inside `game_load` :

- You MUST call `tincle_add_state()` exclusively WITHIN the `game_load()` function.
- DO NOT invoke `tincle_add_state()` inside `main()` or `WinMain()` . It results in a fatal black screen.

6.2. Mandatory AI Coding Constraints & Rules

To ensure 100% compilation on the first attempt without compilation errors, the AI must strictly adhere to these 8 coding rules:

1. **NO SEARCH CITATIONS IN CODE:** Never include text-based source citations, inline footnotes, or reference annotations (such as `[cite: 1]` , `[1]` , or `(Source: ...)`) anywhere inside or outside the code block. These annotations instantly cause syntax errors in C. Output ONLY valid, clean C source code.
2. **Absolute Pure C (C99 Standard):** Do not mix C++ syntax (e.g., no classes, no namespaces).
3. **Double Quotes & Platform-Specific Headers:** Always include engine core headers using `#include "libs/tincle.h"` . When including raw SDL headers, you must cleanly branch your code using `#ifdef __EMSCRIPTEN__` — use `#include "libs/SDL.h"` or `#include "libs/SDL2_gfxPrimitives.h"` or etc. for Windows native and `#include <SDL2/SDL.h>` or `#include <SDL2/SDL2_gfxPrimitives.h>` or etc. for Emscripten Web targets.
4. **Explicit Math Headers:** If you utilize mathematical operations (such as `sinf` , `cosf` , `sqrtf` , `fabsf`), you must explicitly `#include <math.h>` . Do not assume standard headers are pre-included.
5. **Mandatory Font Preloading:** `tincle_load_font()` must be executed in the `game_load()` lifecycle sequence.
6. **Strict State-Separation Model:** Do not write a chaotic single-loop game block. Split game states cleanly (e.g., Title Screen, Gameplay Loop, Game Over Screen) using the built-in State Management API.
7. **Always Use Approved Build Tools (`tinclec` & `emcc`):** When guiding the user on how to build the game, instruct them based on the target platform. Instruct them to execute the bundled `tinclec` tool for native Windows desktop builds, and the `emcc` tool for Web/WASM compilation. Never instruct them to use raw `tcc` or raw `emcc` commands directly.
8. **Never Confuse Asset Utilities:**
 - `asspack.exe` is used to pack all game media assets (images, fonts, sounds) into a secure `game.data` package.
 - `embedres.exe` is used ONLY for embedding binary metadata (like `icon.ico` and application manifest strings) directly into the executable file header. It cannot process game media assets.

7. Creation of Release Version (Distribution Instructions)

When ready to build the final production release for distribution, follow this strict five-step pipeline:

- **Asset Credit Management:** If you recognize, extract, or utilize third-party assets from a provided ZIP file or external sources, you must automatically append the appropriate credit and attribution information to `CREDITS.txt` . Do not leave `CREDITS.txt` as the default template if new third-party assets are introduced.

Step 1: Hide the Debug Console

Locate the `tincle_start` call inside the conditional entry points (`main` for Web or `WinMain` for Windows) in `game.c` . Change the third argument (`is_debug`) from `true` to `false` .

```
// Example: Hiding the runtime debug log or console output
```

```
tincle_start(game_load, "My Ultimate Game", false, 640, 480);  
// Set this to false for production!
```

Step 2: High-Density Asset Packing via `asspack` (for Windows)

To bundle all external textures, audio clips, and font files into an uncrackable, optimized unified package for Windows Desktop deployment, execute the `asspack` command in the terminal. This creates a single consolidated virtual file called `game.data`.

```
> asspack font.ttf player.png enemy.png bgm.ogg shoot.wav
```

Note: Once `game.data` is generated, the Windows engine will automatically look inside this packed binary container. However, if a raw asset file exists locally, it takes precedence over the packed asset. When distributing your game, pack all assets into `game.data` and do not include raw asset files in the release archive.

Step 3: Web/WASM Production Build via `emcass` (for Web)

For web browser deployment, the asset packaging and compilation are unified into a single command using `emcass`. You do not need to use `asspack` manually for Web builds; `emcass` handles both asset preloading and generation of the web deliverables automatically.

```
emcass index.html "My Ultimate Game" main.c player.c stage.c
```

Note: This command scans your local development directory, compiles the code via the Emscripten toolchain, packs all detected assets into `index.data`, and uses `template.html` to output the complete browser-ready package with the mandatory fixed base name `index`. Requirements: When executing a Web build, ensure that `CREDITS.txt` is placed in the exact same folder as the source code before running the build. This is strict because `emcass` requires `CREDITS.txt` in that directory to display credits properly; otherwise, the web-version credit screen will be blank.

Step 4: Final Windows Package Checklist (ZIP Archive)

Distribute the final Windows desktop build by packaging the following **"8 structural items"** inside a ZIP archive. Do not include individual raw source assets, header files, or `.c` files in the player package:

1. `game.exe` (The compiled game binary generated by `tinclec` - can be renamed)
2. `game.data` (The packed resource binary containing all assets generated by `asspack.exe`)
3. `tincle.dll` (The TinCLE core engine runtime)
4. `SDL2.dll` (Official SDL2 library)
5. `SDL2_ttf.dll` (Official SDL_ttf library)
6. `SDL2_gfx.dll` (Official SDL2_gfx library)
7. `CREDITS.txt` (License documentation for the engine and libraries)
8. `readme.txt` (The instruction manual file for the game itself)

Note: Create and prepare a `readme.txt` file containing a description of the game to be distributed.

Step 5: Final Web Package Checklist (ZIP Archive)

Distribute the final Web browser build by packaging the following **"5 structural items"** generated by `emcass` into a single ZIP archive for platform deployment (e.g., itch.io HTML5 project upload):

1. `index.html` (The web page housing the game canvas - MUST NOT be renamed)
2. `index.js` (The JavaScript glue-code generated by Emscripten)

- 3. `index.wasm` (The compiled WebAssembly binary core)
- 4. `index.data` (The virtual filesystem package containing preloaded assets)
- 5. `CREDITS.txt` (License documentation for the engine, Emscripten components, and libraries)

MANDATORY INSTRUCTIONS FOR `readme.txt` CREATION

When the AI agent or developer prepares the `readme.txt` (or platform game page description), the following specifications must be explicitly detailed without exception:

- 1. **Input Shortcuts & Window Controls Disclosure:**
 - **For Windows Desktop Players:** Clearly state that pressing `Alt + Enter` toggles between Fullscreen and Windowed mode, and pressing `Alt + Backspace` immediately resets the game window to its default resolution scale. (Since these core shortcuts are invisible, players have no way of knowing them unless documented here).
 - **For Web Browser Players:** Clearly state that players can toggle Fullscreen mode by clicking the dedicated HTML canvas fullscreen button provided on the game page.
- 2. **Game Creator Copyright Attribution:**
 - The `readme.txt` must clearly display the user's/developer's own copyright notice as the primary game creator (e.g., Copyright (C) 2026 [Your Name / Team Name]).
 - **CRITICAL AI WARNING:** When creating code or release readmes, do NOT just only append the TinCLE framework copyright notice (Copyright (C) 2026 michyo). While the engine license requires credit attribution to the engine author, the game itself belongs to the game creator. The AI must explicitly separate the Game Creator Copyright from the Engine Framework Credits and ensure the game author's identity is clearly defined.

8. Standard Implementation Boilerplate Code for `game.c`

This section provides the comprehensive, bulletproof source code boilerplate template that AI agents must utilize as the structural foundation when spinning up a brand new game project from scratch.

```
#define SDL_MAIN_HANDLED
#include "libs/tincle.h"
#include <stdio.h>
#include <math.h> // Load math.h first, in the safest possible way.
// Automatically switch between headers for the Web version and the Windows version
#ifdef __EMSCRIPTEN__
#include <SDL2/SDL_scancode.h> // Required to use SDL scancodes.
#include <SDL2/SDL2_gfxPrimitives.h>
#else
#include "libs/SDL_scancode.h" // Required to use SDL scancodes.
#include "libs/SDL2_gfxPrimitives.h"
#include <windows.h>
#endif
// [TCC WORKAROUND] Prevent 'M_PI redefined' warning from SDL2_gfx and math.h.
#undef M_PI
#define M_PI 3.14159265358979323846
// [TCC WORKAROUND] Fix 'undefined symbol sinf' error in Windows TCC environments.
#ifndef sinf
#define sinf(x) ((float)sin((double)(x)))
#endif
// =====
// Global System Variables & Asset IDs
// =====
static SDL_Renderer* renderer = NULL;
static float g_player_x = 320.0f;
```

```

static float g_player_y = 240.0f;
static float g_game_time = 0.0f;
// Pre-loaded Asset ID Handles
static int g_img_player = -1;
static int g_font_large = -1;
static int g_font_medium = -1;
// =====
// State: Title Screen (State ID: 0)
// =====
void title_update(float dt) {
    // Use pure native SDL2 scancodes and controller button constants for input queries.
    if (tincle_key_pressed(SDL_SCANCODE_SPACE) || tincle_pad_pressed(SDL_CONTROLLER_BUTTON_START)) {
        tincle_change_state(1); // Transition to Gameplay Loop
    }
}

void title_draw(void) {
    // Clear canvas using inline C99 compound literals for TinCLE_Color structure.
    tincle_clear((TinCLE_Color){0, 0, 0, 255});
    // Activate the desired font asset handle via tincle_set_font prior to string rendering.
    tincle_set_font(g_font_large);
    tincle_draw_text_center("YOUR_GAME_TITLE", 320, 180, (TinCLE_Color){255, 255, 255, 255});
    tincle_set_font(g_font_medium);
    tincle_draw_text_center("PRESS SPACE TO START", 320, 300, (TinCLE_Color){0, 255, 0, 255});
}
// =====
// State: Gameplay Loop (State ID: 1)
// =====
void play_update(float dt) {
    g_game_time += dt;
    // Framerate Independence: Always multiply movement speed by delta time (dt).
    float move_speed = 200.0f;
    if (tincle_key_pressed(SDL_SCANCODE_LEFT) || tincle_pad_pressed(SDL_CONTROLLER_BUTTON_DPAD_LEFT)) g_player_x -= move_speed;
    if (tincle_key_pressed(SDL_SCANCODE_RIGHT) || tincle_pad_pressed(SDL_CONTROLLER_BUTTON_DPAD_RIGHT)) g_player_x += move_speed;
    if (tincle_key_pressed(SDL_SCANCODE_UP) || tincle_pad_pressed(SDL_CONTROLLER_BUTTON_DPAD_UP)) g_player_y -= move_speed;
    if (tincle_key_pressed(SDL_SCANCODE_DOWN) || tincle_pad_pressed(SDL_CONTROLLER_BUTTON_DPAD_DOWN)) g_player_y += move_speed;
    if (tincle_key_pressed(SDL_SCANCODE_ESCAPE) || tincle_pad_pressed(SDL_CONTROLLER_BUTTON_BACK)) {
        tincle_change_state(0); // Return to Title Screen
    }
}

void play_draw(void) {
    tincle_clear((TinCLE_Color){20, 20, 50, 255});
    tincle_set_font(g_font_medium);
    tincle_draw_text("GAME PLAYING", 20, 20, (TinCLE_Color){0, 255, 0, 255});
    // CRITICAL MANDATE: tincle_draw_image handles coordinates as the CENTER point of the asset!
    if (g_img_player != -1) {
        tincle_draw_image(g_img_player, (int)g_player_x, (int)g_player_y);
    } else {
        tincle_draw_text("P", (int)g_player_x, (int)g_player_y, (TinCLE_Color){255, 0, 0, 255});
    }
    // [NATIVE EXTENSION EXAMPLE 1] Render a filled square using native SDL API
    SDL_Rect rect = { 100, 100, 50, 50 };
    SDL_SetRenderDrawColor(renderer, 255, 0, 0, 255);
    SDL_RenderFillRect(renderer, &rect);
    // [NATIVE EXTENSION EXAMPLE 2] Render a floating alpha ellipse using SDL2_gfx primitives
    filledEllipseRGBA(renderer, 400, 300, 50, 25, 255, 0, 0, 128);
    // [NATIVE EXTENSION EXAMPLE 3] Render a floating triangle using SDL2_gfx filledTrigonRGBA
    int tri_top_y = 150 + (int)(sinf(g_game_time * 5.0f) * 15.0f);
    filledTrigonRGBA(renderer,
        400, tri_top_y, // Vertex 1 (Top, moving dynamically)
        360, 200,      // Vertex 2 (Bottom Left)
        440, 200,      // Vertex 3 (Bottom Right)
        0, 255, 255, 200 // Color: Cyan with Alpha
    );
}
// =====

```

```
// System Lifecycle: Initialization & Asset Preloading
// =====
void game_load(void) {
    // Cache the underlying core SDL_Renderer context to enable native graphics extensions
    renderer = tincle_get_renderer();
    // Load font assets explicitly with explicit pixel size arguments.
    g_font_large = tincle_load_font("font.ttf", 64);
    g_font_medium = tincle_load_font("font.ttf", 32);
    // Load image assets and cache their integer ID handles
    g_img_player = tincle_load_image("player.png");
    // Bind operational callback procedures to their respective scene handles
    tincle_add_state(0, title_update, title_draw);
    tincle_add_state(1, play_update, play_draw);
    // Switch to the initial starting scene state
    tincle_change_state(0);
}
// =====
// Main Entry Point (Hybrid Bootstrapper for Windows and WASM)
// =====
#ifdef __EMSCRIPTEN__
// Standard command-line entry point designated for the Web (Emscripten / WASM) runtime engine.
int main(int argc, char* argv[]) {
    tincle_start(game_load, "YOUR_GAME_TITLE", true, 640, 480);
    return 0;
}
#else
// Subsystem-Windows WinMain entry point required for native Windows app execution (TCC, etc.).
// This explicitly suppresses the default standard console window popup on retail execution.
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpCmdLine, int nCmdShow) {
    tincle_start(game_load, "YOUR_GAME_TITLE", true, 640, 480);
    return 0;
}
#endif
```
