

ChatGPT 5.2 Accessed December 2025

You said:

i am interested in building a rust game using raylib. I want to be able to run different scenes to separated different modes of the game such as menu, main game, win screen etc. What is the best way to structure this design? I was thinking of having something like a scene trait and having modes and levels implement it. That trait would have things like preprocess, handle input, update, draw, postprocess etc. Is this a good design? What is a good flexible way to implement a good design that lets the behavior of each scene be separated?

ChatGPT said:

Your instinct is solid  — a **scene/state-based architecture** with a trait is *exactly* how most well-structured games handle menus, gameplay, win screens, etc. Raylib works very well with this pattern, and Rust's traits make it clean and safe.

Below I'll walk through:

1. **Why your idea is good**
 2. **A recommended scene trait design**
 3. **How to manage scene transitions cleanly**
 4. **Two common variations (simple vs. scalable)**
 5. **Practical Rust + raylib considerations**
-

1. Is a `Scene` trait a good idea?

Yes. What you're describing is usually called:

- **Scene pattern** or
- **Game State pattern**

Your idea of:

- menu
- gameplay
- win/lose screen

→ each implementing the same interface
→ updated and drawn by a central game loop

is a **proven, flexible design**.

Key benefits:

- Clear separation of behavior
 - Easy to add/remove scenes
 - Avoids giant `match` statements in your main loop
 - Scales well as the project grows
-

2. A clean `Scene` trait (recommended)

You don't need *too many* lifecycle functions at first. Start small and grow later.

Minimal but flexible trait

```
use raylib::prelude::*;

pub enum SceneSwitch {
    None,
    Push(Box<dyn Scene>),
    Replace(Box<dyn Scene>),
    Pop,
    Quit,
}

pub trait Scene {
    fn on_enter(&mut self, _rl: &mut RaylibHandle) {}
    fn on_exit(&mut self, _rl: &mut RaylibHandle) {}

    fn handle_input(&mut self, _rl: &mut RaylibHandle) -> SceneSwitch {
        SceneSwitch::None
    }

    fn update(&mut self, _dt: f32) -> SceneSwitch {
        SceneSwitch::None
    }

    fn draw(&self, d: &mut RaylibDrawHandle);
}
```

Why this works well

- **Scenes control their own transitions**
 - `SceneSwitch` avoids tight coupling between scenes
 - `on_enter` / `on_exit` are cleaner than `preprocess/postprocess`
 - `draw` is immutable (`&self`) → enforces good separation
-

3. Scene manager (the glue)

You'll want a **SceneManager** that owns a stack of scenes.

Why a stack?

- Menu over gameplay
- Pause screen on top of game
- Modal dialogs
- Clean “return to previous state”

Scene manager example

```
pub struct SceneManager {
    scenes: Vec<Box<dyn Scene>>,
    quit: bool,
}

impl SceneManager {
    pub fn new(initial: Box<dyn Scene>, rl: &mut RaylibHandle) -> Self {
        let mut mgr = Self {
            scenes: vec![initial],
            quit: false,
        };
        mgr.scenes.last_mut().unwrap().on_enter(rl);
        mgr
    }

    pub fn update(&mut self, rl: &mut RaylibHandle, dt: f32) {
        if let Some(scene) = self.scenes.last_mut() {
            let switch = scene.handle_input(rl);
            self.apply_switch(switch, rl);
        }

        if let Some(scene) = self.scenes.last_mut() {
            let switch = scene.update(dt);
            self.apply_switch(switch, rl);
        }
    }

    pub fn draw(&self, d: &mut RaylibDrawHandle) {
        if let Some(scene) = self.scenes.last() {
            scene.draw(d);
        }
    }

    fn apply_switch(&mut self, switch: SceneSwitch, rl: &mut RaylibHandle) {
        match switch {
            SceneSwitch::None => {}
            SceneSwitch::Push(scene) => {
                scene.on_enter(rl);
                self.scenes.push(scene);
            }
            SceneSwitch::Replace(scene) => {
                if let Some(mut old) = self.scenes.pop() {
                    old.on_exit(rl);
                }
                self.scenes.push(scene);
            }
        }
    }
}
```

```

        }
        scene.on_enter(rl);
        self.scenes.push(scene);
    }
    SceneSwitch::Pop => {
        if let Some(mut old) = self.scenes.pop() {
            old.on_exit(rl);
        }
    }
    SceneSwitch::Quit => {
        self.quit = true;
    }
}
}

pub fn should_quit(&self) -> bool {
    self.quit || self.scenes.is_empty()
}
}

```

4. Example scene implementation

Menu scene

```

pub struct MenuScene;

impl Scene for MenuScene {
    fn handle_input(&mut self, rl: &mut RaylibHandle) -> SceneSwitch {
        if rl.is_key_pressed(KeyboardKey::KEY_ENTER) {
            return SceneSwitch::Replace(Box::new(GameScene::new()));
        }
        SceneSwitch::None
    }

    fn draw(&self, d: &mut RaylibDrawHandle) {
        d.clear_background(Color::BLACK);
        d.draw_text("Press ENTER to Start", 200, 200, 30, Color::WHITE);
    }
}

```

Gameplay scene

```

pub struct GameScene {
    timer: f32,
}

impl GameScene {
    pub fn new() -> Self {
        Self { timer: 0.0 }
    }
}

impl Scene for GameScene {

```

```

fn update(&mut self, dt: f32) -> SceneSwitch {
    self.timer += dt;
    if self.timer > 10.0 {
        return SceneSwitch::Replace(Box::new(WinScene));
    }
    SceneSwitch::None
}

fn draw(&self, d: &mut RaylibDrawHandle) {
    d.clear_background(Color::DARKBLUE);
    d.draw_text("Playing...", 20, 20, 20, Color::WHITE);
}
}

```

5. Main game loop (raylib-friendly)

```

fn main() {
    let (mut rl, thread) = raylib::init()
        .size(800, 600)
        .title("Rust Raylib Game")
        .build();

    rl.set_target_fps(60);

    let mut scene_manager = SceneManager::new(
        Box::new(MenuScene),
        &mut rl,
    );

    while !rl.window_should_close() && !scene_manager.should_quit() {
        let dt = rl.get_frame_time();

        scene_manager.update(&mut rl, dt);

        let mut d = rl.begin_drawing(&thread);
        scene_manager.draw(&mut d);
    }
}

```

6. Variations & scaling options

Option A – Simple enum-based scenes (small games)

```

enum Scene {
    Menu(MenuScene),
    Game(GameScene),
    Win(WinScene),
}

```

- ✓ Less dynamic dispatch
 - ✗ Becomes messy as scenes grow
 - ✗ Harder to extend modularly
-

Option B – ECS inside a scene (advanced)

- Each **scene owns its own ECS world**
- Scene manages systems and entities
- Scene trait stays the same

This is very common in larger Rust games.

7. Final recommendation

- ✓ Your idea is **correct and idiomatic**
- ✓ Trait-based scenes + manager is the best flexible approach
- ✓ Avoid too many lifecycle hooks early
- ✓ Use a **SceneSwitch enum** instead of direct scene manipulation
- ✓ Stack-based manager gives you pause screens, modals, etc. for free

If you want, I can:

- Refactor this into a **full project layout**
- Show a **pause menu overlay**
- Show how to share assets safely between scenes
- Explain **ownership + lifetimes pitfalls** in Rust game code

Just tell me what level you want to go to next 

You said:

With this design, how can game state be handled that needs to span scenes? For example, what if the player has a score or health needs to transition between scenes?

ChatGPT said:

Great question — this is *the* key design issue once you move beyond toy examples. The short answer is:

**Scenes should not own long-lived game state.
A separate “Game Context / World” should.**

Let's walk through the *clean, flexible ways* to do this in Rust with raylib, and why they work.

Core Principle

Think in **three layers**:

1. **Engine / Loop**
→ owns window, timing, scene manager
2. **Game State (persistent)**
→ player stats, score, inventory, unlocked levels, settings
3. **Scene (temporary behavior)**
→ menu logic, gameplay logic, win screen logic

Scenes *read and mutate* shared state — but **do not own it**.

Option 1 (Recommended): Shared `GameState` passed to scenes

This is the most common and cleanest solution.

Define a persistent game state

```
pub struct GameState {  
    pub score: u32,  
    pub player_health: i32,  
    pub current_level: u32,  
}
```

This struct lives **outside** scenes and persists across transitions.

Update the `Scene` trait to receive it

```
pub trait Scene {  
    fn on_enter(&mut self, _state: &mut GameState) {}  
    fn on_exit(&mut self, _state: &mut GameState) {}  
  
    fn handle_input(  
        &mut self,  
        _rl: &mut RaylibHandle,  
        _state: &mut GameState,  
    ) -> SceneSwitch {  
        SceneSwitch::None  
    }  
}
```

```

fn update(
    &mut self,
    dt: f32,
    _state: &mut GameState,
) -> SceneSwitch {
    SceneSwitch::None
}

fn draw(&self, d: &mut RaylibDrawHandle, _state: &GameState);
}

```

Why this works well

- One authoritative source of truth
 - No global variables
 - Easy to reason about ownership
 - Rust borrow checker stays happy
-

SceneManager now owns GameState

```

pub struct SceneManager {
    scenes: Vec<Box<dyn Scene>>,
    pub state: GameState,
}

```

And passes it into scenes:

```

scene.update(dt, &mut self.state);
scene.draw(&mut d, &self.state);

```

Example: gameplay → win screen

```

impl Scene for GameScene {
    fn update(&mut self, dt: f32, state: &mut GameState) -> SceneSwitch {
        state.score += 1;

        if state.player_health <= 0 {
            return SceneSwitch::Replace(Box::new(WinScene));
        }

        SceneSwitch::None
    }
}

impl Scene for WinScene {
    fn draw(&self, d: &mut RaylibDrawHandle, state: &GameState) {
        d.draw_text(
            &format!("Final Score: {}", state.score),
            200, 200, 30, Color::WHITE,
        )
    }
}

```

```
        ) ;
    }
}
```

- ✓ Score persists
 - ✓ Health persists
 - ✓ Scene code stays isolated
-

Option 2: Scene-local state + shared global state (hybrid)

Use **both**:

- GameState → persistent data
- Scene struct → temporary data

```
pub struct GameScene {
    enemies: Vec<Enemy>,
    level_timer: f32,
}
```

This data **dies when the scene dies**, which is exactly what you want.

Option 3: Resource container (advanced / ECS-like)

Instead of one GameState, use a **resource map**:

```
use std::any::{Any,TypeId};
use std::collections::HashMap;

pub struct Resources {
    map: HashMap<TypeId, Box<dyn Any>>,
}
```

Scenes request what they need:

```
let score = resources.get_mut::<Score>();
```

Pros

- Extremely flexible
- ECS-friendly

Cons

- More complex
- Runtime errors if misused

→ This is great *later*, not early.

Option 4: Scene transitions carry data (not recommended alone)

```
SceneSwitch::ReplaceWithData {
    scene: Box<dyn Scene>,
    score: u32,
}
```

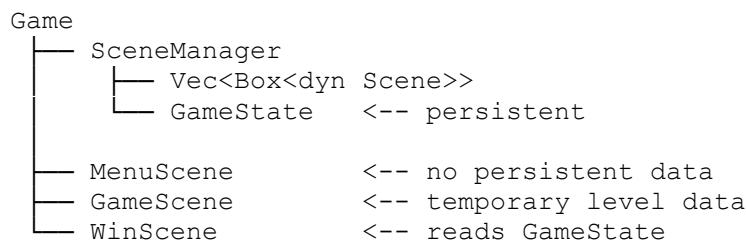
This *works*, but:

- Becomes brittle
- Leads to tightly coupled scenes
- Hard to maintain

Use this **only for very small payloads**, if at all.

Recommended Architecture (TL;DR)

Do this:



Ownership rules

- GameState lives for the whole program
 - Scenes borrow it temporarily
 - Scenes never own cross-scene data
 - Scene transitions don't carry state
-

Bonus: Preventing scene abuse of state

If you want *extra safety*, split state:

```
pub struct GameState {  
    pub player: PlayerState,  
    pub meta: MetaState,  
}
```

Or pass only what a scene needs:

```
fn update(&mut self, ctx: &mut GameplayContext);
```

This prevents menus from accidentally mutating gameplay data.

Final advice

- ✓ You are thinking about the *right problems*
- ✓ This design scales from small games → commercial engines
- ✓ Start with **one GameState struct**
- ✓ Evolve toward resources/ECS only if needed

If you want, I can:

- Show **borrow checker–friendly signatures**
- Show how to **reset state cleanly on New Game**
- Add **save/load support**
- Demonstrate **pause scenes that still update the game**

Just say the word 

ChatGPT can make mistakes. Check important info.

If you want, I can:

- Show **borrow checker–friendly signatures**
- Show how to **reset state cleanly on New Game**
- Add **save/load support**
- Demonstrate **pause scenes that still update the game**

Just say the word 

Yes, show me some borrow-checker friendly signatures. I am concerned that having the scenes pass instances to switchscene may violate borrowing.

Excellent instinct — this is exactly where Rust game architectures usually *break* the first time 

Let's make this **100% borrow-checker friendly**, explain *why* it works, and point out the patterns that *don't*.

The Core Borrowing Problem (in plain words)

The dangerous pattern looks like this:

```
let switch = scene.update(dt, &mut state);  
self.apply_switch(switch);
```

Why this *can* fail:

- `scene` is mutably borrowed
- `state` is mutably borrowed
- `switch` might contain a **new scene**
- `apply_switch` may **remove or replace scene**

Rust doesn't allow:

“Create something that may depend on a borrow, then mutate the owner of that borrow.”

So the design must ensure:

No scene borrows survive past the update call.

The Golden Rule (remember this)

Scene methods return *commands*, never references.

SceneManager executes those commands *after* all borrows end.

That's the entire trick.

Borrow-Checker Friendly Design (Proven Pattern)

1 SceneSwitch owns everything

No references. No lifetimes.

```
pub enum SceneSwitch {
    None,
    Push(Box<dyn Scene>),
    Replace(Box<dyn Scene>),
    Pop,
    Quit,
}
```

This ensures:

- No scene borrows leak
- No self-references
- No lifetime parameters needed

2 Scene trait signatures (safe & clean)

```
pub trait Scene {
    fn on_enter(&mut self, _state: &mut GameState) {}
    fn on_exit(&mut self, _state: &mut GameState) {}

    fn handle_input(
        &mut self,
        _rl: &mut RaylibHandle,
        _state: &mut GameState,
    ) -> SceneSwitch {
        SceneSwitch::None
    }

    fn update(
        &mut self,
        _dt: f32,
        _state: &mut GameState,
    ) -> SceneSwitch {
        SceneSwitch::None
    }

    fn draw(&self, d: &mut RaylibDrawHandle, _state: &GameState);
}
```

Key detail:

- draw takes `&self` and `&GameState`

- update and handle_input take &mut self + &mut GameState
 - No references escape the function
-

3 SceneManager update loop (borrow-safe)

This is the *important* part.

```
pub fn update(&mut self, rl: &mut RaylibHandle, dt: f32) {  
    let switch = {  
        let scene = self.scenes.last_mut().unwrap();  
        scene.handle_input(rl, &mut self.state)  
    };  
    self.apply_switch(switch);  
  
    let switch = {  
        let scene = self.scenes.last_mut().unwrap();  
        scene.update(dt, &mut self.state)  
    };  
    self.apply_switch(switch);  
}
```

Why this works

- The {} blocks **limit borrow scope**
- scene borrow ends before apply_switch
- state borrow ends before apply_switch
- Rust can prove everything is dropped

This is one of the **most important Rust game patterns**.

4 apply_switch is completely isolated

```
fn apply_switch(&mut self, switch: SceneSwitch) {  
    match switch {  
        SceneSwitch::None => {}  
  
        SceneSwitch::Push(mut scene) => {  
            scene.on_enter(&mut self.state);  
            self.scenes.push(scene);  
        }  
  
        SceneSwitch::Replace(mut scene) => {  
            if let Some(mut old) = self.scenes.pop() {  
                old.on_exit(&mut self.state);  
            }  
            scene.on_enter(&mut self.state);  
        }  
    }  
}
```

```

        self.scenes.push(scene);
    }

    SceneSwitch::Pop => {
        if let Some(mut old) = self.scenes.pop() {
            old.on_exit(&mut self.state);
        }
    }

    SceneSwitch::Quit => {}
}
}

```

No borrowing conflicts because:

- `update()` is finished
 - `scene` borrow no longer exists
 - `GameState` borrow is exclusive here
-

5 What *NOT* to do (common mistakes)

✖ Returning references from scenes

```
fn update(&mut self) -> Option<&dyn Scene>;
```

✖ Impossible to make safe.

✖ Storing `&GameState` inside a scene

```
struct GameScene<'a> {
    state: &'a mut GameState, // ✖
}
```

✖ Self-referential borrow trap.

✖ Applying transitions *inside* update

```
scene_manager.replace_scene(new_scene); // ✖
```

✖ Double mutable borrow of `scene_manager`.

6 Alternative: deferred command queue (extra robust)

For more complex games:

```
pub enum Command {  
    Switch(SceneSwitch),  
    PlaySound(SoundId),  
    SaveGame,  
}
```

Scene returns `Vec<Command>`

SceneManager executes commands later.

This makes borrowing **even easier** and is used in big engines.

7 Why this scales

- ✓ No lifetimes in public APIs
- ✓ No Rc/RefCell hacks
- ✓ No unsafe
- ✓ Predictable execution order
- ✓ Easy to test scenes independently

This exact approach is used in:

- Bevy (conceptually)
- Amethyst
- Macroquad games
- Custom Rust engines

Final Mental Model

Scenes describe intent.

SceneManager performs mutations.

Borrows never escape function boundaries.

Once you internalize that, Rust game architecture becomes *much* easier.

If you want next:

- A **pause scene** that overlays without freezing gameplay
- Sharing assets without borrowing issues
- Async loading screens
- ECS integration with this pattern

Just tell me 