ChatGPT

FrameChangeCoordinator Class and Methods in Code

The **FrameChangeCoordinator** described in the refactor plan does not yet appear in the main codebase. We searched the repository and found **no** `FrameChangeCoordinator` **class implementation** (e.g. no `ui/controllers/frame_change_coordinator.py`). This suggests the coordinator is not integrated into the current branch. Consequently, none of its methods (`connect`, `disconnect`, `on_frame_changed`, `_update_background`, `_apply_centering`, `_invalidate_caches`, `_update_timeline_widgets`, `_trigger_repaint`) exist in the code. This is a deviation from the plan's Phase 1, which intended to introduce this class [1] [2]. It appears the coordinator is either pending merge or hidden behind a feature flag not present in code. This is a critical discrepancy – the plan's atomic frame-change handler is **absent**, so the app still uses the old multi-handler mechanism.

Coordinator-Invoked Methods Availability

All the methods the coordinator *would* call do exist in the codebase, ensuring the proposed sequence is feasible:

- **Background update**: `view_management_controller.update_background_for_frame(frame)` exists and is currently used to sync the background image on frame changes [3].
- **Centering**: `curve_widget.center_on_frame(frame)` is implemented (delegates to the view camera) [4]. The `CurveViewWidget` also tracks an `auto_center` (centering) mode (`curve_widget.centering_mode`).
- **Cache invalidation**: `curve_widget.invalidate_caches()` is defined [5] and used to clear render caches.
- **Timeline controls update**: `timeline_controller.update_frame_display(frame, update_state=False)` exists [6]. In fact, the TimelineController currently connects the state manager's signal to a lambda calling this method [7].
- **Timeline tabs update**: The TimelineTabs widget has a method for visual frame updates. In code it's named `timeline_tabs._on_state_frame_changed(frame)` – which performs *only UI updates* (highlighting the new frame tab, etc.) [8] [9]. This corresponds to the plan's `_on_frame_changed` method. The public `timeline_tabs.set_current_frame(frame)` method also exists [10], but it **changes state via the StateManager** (calls `self.current_frame = frame`, which updates `state_manager.current_frame`) [11]. The plan explicitly noted that the coordinator should call the visual update method (_on_frame_changed) instead of `set_current_frame` to avoid feedback loops [12]. The code confirms these two distinct methods: one for visual updates only, one for driving state.
- **Final repaint**: `curve_widget.update()` is a standard Qt repaint trigger available on the widget. The coordinator would use this for the "single repaint" step. (No custom method needed – Qt's `update()` is called on the widget.)

Conclusion: All target methods the coordinator would invoke do exist and function as expected in current code (with the minor note that `TimelineTabs._on_state_frame_changed` is the visual update method, named slightly differently than in the plan). The architecture envisioned (background → centering → caches → widget updates → one repaint) is supported by available APIs [13] [14].

TimelineTabs Frame Update vs. State Change

The TimelineTabs implementation aligns with the plan's dual-method approach: it has one method to reflect a new frame in the UI, and another to update the application state. Specifically:

- **Visual update method:** `TimelineTabWidget._on_state_frame_changed(frame)` – connected as an observer to `StateManager.frame_changed` – updates the highlighted tab and frame info **without emitting further signals** [15] [8]. This matches the plan's `_on_frame_changed` (visual-only) behavior. Indeed, the code's docstring labels it "visual updates only" [9].
- **State-changing method:** `TimelineTabWidget.set_current_frame(frame)` – called when a user clicks a frame tab – sets `state_manager.current_frame` via the property setter (thus emitting the frame change signal) [16] [11]. It then emits the deprecated `frame_changed` signal on TimelineTabs for legacy compatibility [17]. The plan mentioned exposing `_on_frame_changed` logic as `set_current_frame` for external use, but ultimately retained both functions [12].

Verification: The coordinator should call the visual update method. Since in code this method is named `_on_state_frame_changed`, the coordinator (once implemented) must invoke that (or alias it) instead of `set_current_frame`. This ensures the coordinator doesn't recursively trigger state changes. Currently, no coordinator exists to do this, but the separation of concerns is correctly implemented in TimelineTabs.

Note: The plan's text uses `_on_frame_changed` naming, but the actual code uses `_on_state_frame_changed` for the same purpose. This is just a naming mismatch – functionally, the method is present and does what's needed. It may be worth renaming or aliasing it for consistency when integrating the coordinator, to avoid any confusion or mis-calling.

Original `frame_changed` Signal Connections (Six Instances)

In the current codebase, **six different handlers** are wired to the `StateManager.frame_changed` signal, exactly as the refactor plan identified [18] [19]. We verified each connection:

1. **MainWindow** – Connected via the SignalConnectionManager:
2. `state_manager.frame_changed → MainWindow.on_state_frame_changed` [20].
3. This handler in MainWindow simply logs the frame change and performs no UI updates [21], acting as a stub observer (the plan notes it's effectively a no-op and could be removed [22] [21]).
4. **ViewManagementController** – Also connected in SignalConnectionManager:
5. `state_manager.frame_changed → ViewManagementController.update_background_for_frame` [3].
6. This ensures the background image is swapped to the new frame's image if a sequence is loaded. Importantly, this handler **does not call** `update()`, it just prepares data (the plan notes this was a patched fix to avoid extra repaints [23]).
7. **TimelineController** – Connects in its own `_connect_signals`:
8. `state_manager.frame_changed → (lambda f: self.update_frame_display(f, update_state=False))` [7].
9. This updates the frame spinbox and slider positions to reflect the new frame, with `update_state=False` to avoid feeding back into state. No repaint is triggered here (the widget adjustments don't inherently repaint).
10. **TimelineTabs** – Connected when TimelineTabs is initialized in MainWindow:

11. `state_manager.frame_changed → TimelineTabWidget._on_state_frame_changed` [15] (called inside `timeline_tabs.set_state_manager(...)`).
12. As discussed, this updates the timeline tab highlights. It explicitly does *not* emit any further frame signals (to prevent loops) [24] .
13. **StateSyncController (CurveViewWidget)** – Connects in the curve view's StateSyncController:
14. `state_manager.frame_changed → StateSyncController._on_state_frame_changed` [25] .
15. This is the handler inside the curve view widget's controller that handles frame changes. Notably, it triggers a repaint and centering: it invalidates caches and calls `widget.update()`, then if centering is enabled, calls `widget.center_on_frame(frame)` [26] . This ordering (paint then center) is part of the race condition.
16. **CurveViewWidget Fallback** – Connects in `CurveViewWidget.set_main_window`:
17. `state_manager.frame_changed → StateSyncController._on_state_frame_changed` (again) [27] .
18. This is essentially a **duplicate connection** to the same StateSync handler as #5. It's done as a "fallback" if the widget was created before the state manager was attached. However, Qt does not prevent connecting the same slot twice, so if the StateSyncController was already connected (#5), this results in **two connections** to `_on_state_frame_changed`. Both will fire, causing duplicate handling (i.e. two `update()` calls) [28] [29] . The code tries a `try/except` on disconnect to avoid errors, but since duplicate connects are allowed (no exception), the guard is ineffective [30] . We confirmed the duplicate connect at `curve_view_widget.py` line 1809 [27] .

All six connections above are present. This means in the current code, a frame change triggers all these in **unspecified order** (Qt's signal order is undefined), exactly as the problem statement in the plan describes [31] [32] . The race condition scenario is real: for example, StateSyncController might run (centering+update) before ViewManagement (background load), causing the "background image lag" jump on the next frame [33] . The plan identified this non-deterministic order and the duplicate update call as key issues [34] – our code review confirms those issues exist.

Duplicate Update Calls in CurveViewWidget

The **CurveViewWidget fallback connection** is indeed causing duplicate `update()` calls on each frame change. As noted, the `StateSyncController._on_state_frame_changed` slot ends up connected twice (once in StateSyncController initialization, once in `set_main_window`). When a frame change signal emits, `_on_state_frame_changed` executes twice, and in each call it invokes `widget.update()` [26] . This results in two paint events per frame change in the worst case. The refactor document explicitly flags this "duplicate connection" and notes *"update() currently called 1-2 times per frame"* as a risk [28] [30] .

Our review confirms the code path: the fallback `connect()` in `CurveViewWidget.set_main_window()` (line 1809) does not check if already connected [27] . Qt allows multiple connections, so no exception is thrown and the `except` block never executes in normal cases [30] . Thus, this duplicate remains unless explicitly handled.

Implication: Without the coordinator, the app sometimes does two repaints on a single frame change. This is inefficient and, combined with ordering issues, can contribute to flicker or state mismatches. It needs to be fixed as per the plan's pre-migration steps. The plan suggested either removing this fallback or adding a guard to ensure it only connects if not already connected [30] . As of now, that fix is not applied – this is an area needing cleanup before or during the coordinator rollout.

Feature Flag for Coordinator and Code Path Parity

The migration strategy mentions a feature flag (e.g. `use_frame_change_coordinator`) to toggle the new coordinator path. We found **no reference in the code to any such flag or toggle** – no configuration setting, environment variable, or conditional branch enables the coordinator logic. This implies the coordinator is not yet wired in, even behind the scenes. The current code exclusively uses the legacy path.

For both code paths (legacy vs. coordinator) to be "functional and equivalent," the plan likely intended: when the flag is **off**, the six independent handlers work as before; when **on**, only the coordinator handles frame changes (and the six direct connections are removed). However, since the coordinator isn't present, the flag logic isn't implemented. **None of the existing connections are gated by a flag** – SignalConnectionManager unconditionally connects all the old handlers [20] [35], and no code disables them at runtime. In other words, the code does *not* yet respect any `use_frame_change_coordinator` flag (it appears to be still in the planning stage).

Consequences: As of now, there is no dual-path to compare – only the old path runs. Once the coordinator is introduced, care must be taken to: (a) add a global flag or config, and (b) conditionally **connect either** the coordinator **or** the old signals, but not both. The plan's Phase 2 outlines this: remove all other `frame_changed` connections and replace with a single coordinator connection [36] [37]. The codebase has not done this yet. We should verify, when implemented, that enabling the coordinator truly disconnects or bypasses *all six* of the old handlers so that we don't end up with both firing. Right now, both code paths are **not present simultaneously**, so "equivalent functionality" is theoretical until the coordinator is in place.

In summary, the feature flag is not yet operational, meaning the new coordinated update path hasn't been tested in the integrated application. Until it is, the old path remains the sole mechanism. Both paths cannot be compared in practice, but on paper they aim to achieve the same end-state each frame. The coordinator should replicate all effects of the six handlers in a controlled sequence (background → centering → caches → UI updates → one repaint). From our analysis of the methods above, once integrated it **will** cover all needed actions. The key is ensuring no legacy handler sneaks in when the flag is on.

Discrepancies, Required Updates, and Rollout Risks

Naming/API Mismatches: One minor discrepancy is the TimelineTabs method name (`_on_state_frame_changed` in code vs `_on_frame_changed` in plan). The plan ultimately decided "no rename needed" [12], so this is just something to document. The coordinator code in the plan calls `timeline_tabs._on_frame_changed(frame)` [38], which would currently fail since the method is named differently. The fix can be trivial (call the existing name or add an alias). Also, the plan suggested possibly making the visual update method public (using `set_current_frame` for external calls) [39], but they explicitly chose to keep both methods distinct and just clarify their usage. No functional issue here, just ensure the coordinator calls the correct function.

Orphaned Legacy Handlers: After introducing the coordinator, certain legacy code should be cleaned up to avoid conflicts:
- The MainWindow's `on_state_frame_changed` (which does nothing) can be removed or left disconnected [22]. It's harmless if it stays unconnected (currently it just logs [21]), but leaving dead code can confuse maintainers.
- More importantly, the **StateSyncController._on_state_frame_changed** logic should be retired once

the coordinator runs. The plan indicates this handler would be removed in Phase 3 [40]. Presently it's the one doing centering and update out-of-order [26]. If the coordinator replaces it, we must ensure StateSyncController no longer connects to `frame_changed` at all, or else it would conflict (e.g. centering twice). In the final design, the coordinator itself handles centering and calls `curve_widget.invalidate_caches()` and one `update()`, making the StateSync handler redundant. Failing to disconnect it would reintroduce the race conditions we're trying to solve. So, as part of feature-flag rollout, **disconnecting or removing StateSync's frame handler is critical**. The same goes for the TimelineController and TimelineTabs connections – they should be skipped or removed when coordinator is active [37]. Currently, these removals are not done. This is a to-do before flipping the switch.
- The CurveViewWidget fallback connection must be addressed (either removed or properly guarded) *before* enabling the coordinator [30]. Otherwise, if somehow the coordinator and fallback both connected, it could be even worse (coordinator calls update once, but a stray StateSync connection might also call update). The plan's Pre-Work item 2 covers fixing this bug upfront [30], but code still shows the old behavior. This is a point of risk during rollout – it should be fixed regardless of coordinator to stop the double-paint issue.

Equivalence of Behavior: Assuming all above are handled, the coordinator path will produce the same end result as the legacy path, just in one tidy package. We double-checked the sequence: the **only difference** in outcome might be that in the legacy path, if centering is enabled, the centering was applied either *before* or *after* the background was set depending on signal order, leading to misalignment. The coordinator will enforce background update first, centering second (which is the correct order) [41] [42]. This is actually an improvement, not a discrepancy – it fixes the race. Otherwise, timeline controls and tabs updates will happen either way, and one final repaint occurs.

One subtle point: the StateSync handler currently calls `invalidate_caches` **and** does an immediate repaint on every frame change [26]. The coordinator plan calls for invalidating caches and delaying the repaint until after widgets update [43] [44]. This should be fine (and better) since no painting happens with stale caches. We just need to ensure no code relies on the *immediate* repaint (e.g. tests expecting two rapid paints). Given the test plan in the document (they plan to add tests to ensure a single repaint and correct order [45] [46]), this should be carefully validated. The equivalence here is logical rather than literal: the coordinator yields **one paint per frame change**, whereas the old path might have one or two (usually one, but two in the race scenario). So in normal conditions it's equivalent, and in race conditions it's an improvement.

Rollout Risks:
- *Partial deployment:* The biggest risk is running the coordinator concurrently with any old connections still active. That would cause unpredictable ordering or duplicate updates (negating the whole purpose). It's imperative that when the feature flag is ON, **exactly one** handler (the coordinator) is connected to `frame_changed` [37]. If even one of the old ones remains, it could reintroduce nondeterminism or double paints. Careful code review and testing (perhaps using the ConnectionVerifier patterns already in code) should be done after implementing Phase 2.
- *Feature toggle logic:* Because there's no flag in place yet, implementing it will require modifying the startup sequence (probably in `SignalConnectionManager`). For example, only connect the coordinator (and not call `set_state_manager` on timeline_tabs, not connect state_sync, etc.) if the flag is true. Conversely, if flag is false, do as now and do not connect the coordinator at all. This conditional logic must be done in a centralized place to avoid fragmentation. Missing any connection in either path could lead to missing functionality (e.g. if the coordinator flag is off but we mistakenly didn't connect one old handler, something might not update). Testing both modes thoroughly is needed.
- *TimelineTabs API clarity:* As noted in the plan, the dual methods in TimelineTabs were a source of confusion. The code now has comments and deprecation warnings to guide developers (the

`frame_changed` signal on TimelineTabs is marked deprecated and they rely on StateManager instead [47] [17]). This is fine, but during the transition, ensure no external code is still connecting to `timeline_tabs.frame_changed`. The SignalConnectionManager explicitly does **not** connect that signal (with a comment explaining why) [47]. This indicates the team already mitigated a potential circular update. As long as everyone uses StateManager (which the coordinator will), we're safe.
- *Logging and debugging:* The coordinator adds logging (e.g. `logger.debug("[FRAME-COORDINATOR] ...")` in the plan code) which is useful [43] [44]. Ensuring these logs remain (and old debug logs like StateSync's are removed or toned down) will help detect if somehow multiple updates occur or the order is wrong during testing.
- *Backwards compatibility:* The plan's Phase 1 was to introduce the coordinator in a backward-compatible way (i.e. add it but not activate). Given it's not in code, when it arrives, it should be added in a way that doesn't change behavior unless enabled. That likely means adding the class, possibly instantiating it in MainWindow (e.g. `self.frame_change_coordinator = FrameChangeCoordinator(self)`), but **not calling** `connect()` unless the flag is true. We should double-check that approach when it's implemented. Also, cleaning up on app exit (calling coordinator.disconnect) is good practice, though not strictly required if app is closing. The plan did provide a `disconnect()` method for completeness [48].

In summary, the current codebase still runs the old multi-handler frame-change flow. All pieces referenced in the refactor plan are present (we verified all six connections and all called methods). However, the new FrameChangeCoordinator logic is missing, and thus the feature flag toggle is not in effect. Before rollout, the team needs to implement the coordinator class, add the gating logic to prefer it over legacy handlers, and remove/disable those legacy connections when the coordinator is active. The primary areas of attention are the TimelineTabs method naming (ensure the coordinator calls the correct function) and the CurveViewWidget double-connection bug (fix that to prevent duplicate paints). Once those are resolved and the old handlers are properly disconnected, the coordinator should eliminate the race condition and ensure **deterministic, single-repaint** frame updates as designed. All equivalent functionality is accounted for; it's mostly a matter of orchestrating the switchover cleanly.

Sources: The analysis is based on the CurveEditor code (SignalConnectionManager, TimelineController, TimelineTabs, StateSyncController, CurveViewWidget) and the FRAME_CHANGE_COORDINATOR_REFACTOR design document for the planned changes [18] [36] [30] [12]. The current code confirms the presence of the six frame_changed handlers [20] [15] [25] [27] and their behaviors, as discussed above.

---

[1] [2] [12] [13] [14] [18] [19] [22] [23] [28] [29] [30] [31] [32] [33] [34] [36] [37] [38] [39] [40] [41] [42] [43] [44] [45] [46] [48] FRAME_CHANGE_COORDINATOR_REFACTOR.md
file://file-W8Rcio5p3YLqNUx1XycsVv

[3] [20] [35] [47] signal_connection_manager.py
https://github.com/semmlerino/CurveEditor/blob/fa40ae0b263017ea86e1ba9df8ce8b56a1c96668/ui/controllers/signal_connection_manager.py

[4] [5] [27] curve_view_widget.py
https://github.com/semmlerino/CurveEditor/blob/fa40ae0b263017ea86e1ba9df8ce8b56a1c96668/ui/curve_view_widget.py

[6] [7] timeline_controller.py
https://github.com/semmlerino/CurveEditor/blob/fa40ae0b263017ea86e1ba9df8ce8b56a1c96668/ui/controllers/timeline_controller.py

[8] [9] [10] [11] [15] [16] [17] [24] timeline_tabs.py
https://github.com/semmlerino/CurveEditor/blob/fa40ae0b263017ea86e1ba9df8ce8b56a1c96668/ui/timeline_tabs.py

21 main_window.py

https://github.com/semmlerino/CurveEditor/blob/fa40ae0b263017ea86e1ba9df8ce8b56a1c96668/ui/main_window.py

25 26 state_sync_controller.py

https://github.com/semmlerino/CurveEditor/blob/fa40ae0b263017ea86e1ba9df8ce8b56a1c96668/ui/controllers/curve_view/state_sync_controller.py