컴포넌트 인스턴스 — 스코프, 생명주기, GC 분석

문서 목적

뷰어 런타임에서 컴포넌트 인스턴스(2D/3D 공통)가 어떤 스코프에서 생존하고, 언제 GC 대상이 되는지를 분석한다.

부록으로 ECO Asset 컴포넌트의 `setInterval` 인터벌 누수 사례를 통해, 인스턴스 속성의 수동 null 처리가 왜 불필요하며 오히려 유해할 수 있는지를 기록한다.

분석일: 2026-02-13

관련 문서

- `Instance_Management.md` — 에디터 관점의 인스턴스 CRUD
- `WScript_Execution_Process.md` — WScript 실행 프로세스
- `THIS_PROPERTY_MEMORY_LEAK.md` — this 참조 해제 후 속성이 살아남는 메모리 누수 시나리오
- `WHY_NULL_UNNECESSARY.md` — 외부 null 처리가 불필요한 이유

---

1. 인스턴스 스코프 체인

1.1 전체 구조

```
window.wemb                                  ← 전역 (앱 수명 동안 영구 생존)
  └── .mainPageComponent                       ← PageComponent (페이지 단위)
      ├── ._masterLayer (TwoLayerCore)         ← 마스터 레이어
      │     ├── ._childs[]                      ← 인스턴스 배열
      │     └── .componentInstanceListMap        ← Map<element, instance>
      │
      ├── ._twoLayer (TwoLayerCore)            ← 2D 레이어
      │     ├── ._childs[]
      │     └── .componentInstanceListMap
      │
      └── ._mainThreeLayer (ThreeLayerCore)     ← 3D 레이어
            ├── ._childs[]
            ├── ._appendElementListMap          ← Map<mesh, instance>
            └── ._instanceContainer             ← Three.js Scene (mesh 참조만)
```

- `window.wemb.mainPageComponent` 는 앱 시작 시 `RegisterGlobalVariableCommand` 에서 할당됨
  - 뷰어: `packages/viewer/src/viewer/controller/ready/RegisterGlobalVariableCommand.ts:58`
  - 에디터: `packages/editor/src/editor/controller/architect/RegisterGlobalVariableCommand.ts:66`
- `mainPageComponent` 자체는 페이지 이동 시 교체되지 않고, 하위 레이어의 자식만 비워짐
- 인스턴스는 **전역 객체가 아니라 레이어의 자식**이며, 레이어 clear 시 참조가 끊어짐

1.2 레이어별 인스턴스 참조 비교

	2D (TwoLayerCore)	3D (ThreeLayerCore)
배열	`_childs[]`	`_childs[]`
역참조 Map	`componentInstanceListMap` Map<DOM element, instance>	`_appendElementListMap` Map<mesh, instance>
Scene 참조	DOM 트리 (element가 부모 노드에 append)	`_instanceContainer` (mesh만, 인스턴스 아님)
상속	`LayerCore` → `WVDisplayObjectContainer`	`LayerCore` → `WVDisplayObjectContainer`

공통점: 두 레이어 모두 `_childs[]` 배열과 역참조 Map 2개가 인스턴스를 잡는 핵심 참조이며, `WVDisplayObjectContainer.removeChildAll()` 로 정리한다.

---

2. 인스턴스 탄생

2.1 2D — TwoLayerCore (WVDisplayObjectContainer.addChildAt 상속)

파일: packages/common/src/womb/core/display/WVDisplayObjectContainer.ts:145-174

```
addChildAt(component, index) {
  this._childs.splice(index, 0, component);              // 참조 1: 배열
  // ... element를 DOM에 append ...
}
```

이후 `onAddedChild()` 에서 역참조 Map에 등록:

파일: packages/common/src/womb/wv/layer/TwoLayerCore.ts:112-114

```
onAddedChild(comInstance) {
  this.componentInstanceListMap.set(comInstance.element, comInstance);  // 참조 2: Map
}
```

2.2 3D — ThreeLayerCore.addChildAt (오버라이드)

파일: packages/common/src/womb/wv/layer/ThreeLayerCore.ts:654-661

```
addChildAt(component, index) {
  // ...
  this._childs.splice(index, 0, component);              // 참조 1: 배열       (L659)
  this._instanceContainer.add(component.appendElement);   // Scene에 mesh 추가  (L660)
  this._appendElementListMap.set(
    component.appendElement, component                    // 참조 2: Map        (L661)
  );
}
```

> *`_instanceContainer` 는 `component.appendElement` (mesh)를 Scene에 추가하는 것이며, 인스턴스 객체 자체를 참조하지 않는다.*

2.3 탄생 시점 참조 요약

참조 대상	2D	3D
부모 배열	`_childs[]`	`_childs[]`
역참조 Map	`componentInstanceListMap`	`_appendElementListMap`
렌더링 트리	DOM parentNode (element 참조)	`_instanceContainer` (mesh 참조)

---

3. WScript에서 this 바인딩

WScript 내부에서 `this` 가 컴포넌트 인스턴스를 가리키는 메커니즘. 이것은 2D/3D 구분 없이 모든 컴포넌트에 공통 적용된다.

3.1 이벤트 디스패치

파일: packages/common/src/wemb/core/events/WVComponentEventDispatcher.ts:89-154

```
dispatchEvent(eventName, vo = {}) {
  let eventVO = {
    script: this._getEventScriptString(eventName),
    eventData: {
      target: this._instance,          // ← 컴포넌트 인스턴스
      instance_id: this._instance.id,
    },
  };
  WScriptEventWatcher.getInstance().$eventBus.$emit('__com_event__', eventVO);
}
```

3.2 스크립트 실행

파일: packages/common/src/wemb/core/events/WScriptEventWatcher.ts:42-46

```
this._$eventBus.$on('__com_event__', (eventVO) => {
  let context = eventVO.eventData.target;              // target = 인스턴스
  let func = new Function('event', eventVO.script);    // 문자열 → 함수
  func.call(context, eventVO.eventData);               // .call(인스턴스) → this 바인딩
});
```

핵심: `func.call(context)` 에 의해 WScript 안의 `this === 컴포넌트 인스턴스` . 이것은 스크립트 실행 시점에만 일시적으로 바인딩되며, 실행 완료 후 `func` 과 `context` 지역변수는 스택에서 빠져 참조가 소멸한다.

---

4. 인스턴스 소멸

4.1 소멸 트리거

페이지 이동 시 각 레이어의 `clear()` → `removeChildAll()` 호출.

2D 레이어:

파일: packages/common/src/wemb/wv/layer/TwoLayerCore.ts:28-32

```
clear() {
  this.removeChildAll();                    // 자식 인스턴스 제거
  this.componentInstanceListMap.clear();     // 역참조 Map 정리
  $(this._element).empty();                  // DOM 정리
}
```

3D 레이어:

파일: packages/common/src/wemb/wv/layer/ThreeLayerCore.ts:135-140

```
// clear()
this.removeChildAll();                  // 자식 인스턴스 제거
this._appendElementListMap.clear();     // 역참조 Map 정리
```

4.2 removeChildAt — 참조 해제 순서

2D (WVDisplayObjectContainer 기본 구현):

파일: packages/common/src/wemb/core/display/WVDisplayObjectContainer.ts:203-224

```
removeChildAt(index) {
  let comInstance = this._childs.splice(index, 1)[0];    // ① 배열에서 제거
  if (comInstance.appendElement && comInstance.appendElement.parentNode) {
    comInstance.appendElement.parentNode.removeChild(     // ② DOM에서 element 제거
      comInstance.appendElement
    );
  }
  this.dispatchEvent(new WVEvent(WVEvent.REMOVE, comInstance));
  comInstance.destroy();                                  // ③ 내부 정리
  return comInstance;                                     // ④ 리턴
}
```

3D (ThreeLayerCore 오버라이드):

파일: packages/common/src/wemb/wv/layer/ThreeLayerCore.ts:690-707

```
removeChildAt(index) {
  let comInstance = this._childs.splice(index, 1)[0];        // ① _childs에서 제거
  this._instanceContainer.remove(comInstance.appendElement);  // ② Scene에서 mesh 제거
  this._appendElementListMap.delete(comInstance.appendElement);// ③ Map에서 제거
  this._componentListUsingRenderer.delete(comInstance.id);    // ④ 렌더링 Map에서 제거
  this.dispatchEvent(new WVEvent(WVEvent.REMOVE, comInstance));
  comInstance.destroy();                                      // ⑤ 내부 정리
  return comInstance;                                         // ⑥ 리턴
}
```

4.3 리턴값 미수신 — 최종 참조 소멸

2D/3D 공통으로 `removeChildAll()` 에서 호출:

파일: packages/common/src/wemb/core/display/WVDisplayObjectContainer.ts:226-237

```
removeChildAll() {
  while (this._childs.length) {
    this.removeChildAt(0);    // 리턴값을 받지 않음 → comInstance 지역변수 소멸
  }
}
```

`removeChildAt` 은 `return comInstance` 를 하지만, `removeChildAll` 에서 리턴값을 아무 변수에도 할당하지 않는다. `comInstance` 지역변수는 `removeChildAt` 스택 프레임이 끝나는 순간 소멸하며, 이것이 인스턴스를 잡고 있던 마지막 참조가 사라지는 지점이다.

4.4 참조 해제 타임라인 (2D/3D 공통)

시점	인스턴스를 참조하는 곳
splice 전	_childs[i], 역참조 Map, comInstance(지역)
splice 후	역참조 Map, comInstance(지역)
Map delete/clear 후	comInstance(지역)

destroy 후	comInstance(지역) — 내부 프로퍼티 정리 완료
return 후 (미수신)	**없음** → GC 대상

---

5. destroy() 체인 상세

`comInstance.destroy()` 호출 시 실행되는 체인. 2D와 3D는 중간 계층이 다르지만, 최상위(`WVDisplayObject.destroy`)와 최하위(`WVComponent._beforeDestroy` / `_onDestroy`)는 공통이다.

5.1 공통 프레임워크 (모든 컴포넌트)

```
WVDisplayObject.destroy()                          ← 진입점
  ├── _beforeDestroy()                               ← WVComponent 오버라이드
  |     ├── BEFORE_DESTROY 이벤트 디스패치 (WScript)
  |     └── _onViewerDestroy()                       ← 컴포넌트별 오버라이드 가능
  |
  ├── _onDestroy() 체인 (자식 → super 순서)
  |     ├── [구체 컴포넌트]._onDestroy()
  |     ├── ... (중간 계층, 아래 참조)
  |     ├── WVComponent._onDestroy()                 → _$WScriptEventBus, _extensionElements,
  |     |                                              _componentEventDispatcher, _styleManager, _page = null
  |     └── WVDisplayObjectContainer._onDestroy()    → removeChildAll() (하위 자식 재귀)
  |
  └── WVDisplayObject.destroy() 후반부:
        → _properties = null
        → _extraProperties = null
        → _updatePropertiesMap = null
        → _dispatcher.destroy() + null
        → _element = null
        → _parent = null
```

5.2 2D 중간 계층

```
[구체 2D 컴포넌트]._onDestroy()
  → WVDOMComponent._onDestroy()       → DOM element 정리
    → WVComponent._onDestroy()
      → WVDisplayObjectContainer._onDestroy()
```

5.3 3D 중간 계층

```
[구체 3D 컴포넌트]._onDestroy()
  → WV3DResourceComponent._onDestroy() → _invalidateSelectItem = null
    → NWV3DComponent._onDestroy()       → _label, _domEvents, 핸들러 6개, _container = null
      |                                   MeshManager.disposeMesh(appendElement)
      → WVComponent._onDestroy()
        → WVDisplayObjectContainer._onDestroy()
```

5.4 핵심 포인트

- `destroy()` 체인은 프레임워크가 **자기 자신이 만든 프로퍼티**만 null 처리한다
- 컴포넌트가 자체적으로 추가한 필드(예: `datasetInfo` , `config` 등)는 이 체인에서 정리되지 않는다
- **그러나 인스턴스 자체가 GC되면 모든 속성이 함께 수거되므로 수동 null 처리는 불필요하다**

---

6. GC 판정 — 인스턴스는 언제 수거되는가

6.1 정상 경로 (외부에 참조를 넘기지 않은 경우)

`removeChildAt` 리턴 후 인스턴스에 도달할 수 있는 루트 경로가 없으므로, **즉시 GC 대상**이 된다. 인스턴스 위의 모든 속성도 함께 수거된다.

6.2 문제 경로 (외부에 this 캡처 클로저를 넘긴 경우)

`setInterval`, `addEventListener` 등으로 `this`를 캡처한 클로저를 브라우저 타이머나 전역 이벤트 시스템에 등록한 경우:

```
루트 (브라우저 타이머 / window 이벤트)
  └── 콜백 클로저
        └── [[Environment]].this ──→ 컴포넌트 인스턴스 전체
                                       └── 모든 속성 (GC 불가)
```

이 경우 `removeChildAt`에서 배열/Map 참조를 끊어도, **클로저가 this를 잡고 있어 인스턴스가 GC되지 않는다.**

해제 조건: `clearInterval` 또는 `removeEventListener`로 클로저 등록을 해제해야 한다.

---

7. 외부에서 인스턴스를 제어할 때의 원칙

7.1 외부 제어는 항상 나쁜가?

아니다. 외부 코드가 인스턴스에 접근하는 것 자체가 문제가 아니라, **접근 방식과 시점**이 문제를 만든다.

코드베이스에서 실제로 사용되는 외부 접근 패턴을 분류하면 다음과 같다:

접근 유형	예시	안전성
읽기	`targetInstance.datasetInfo`	안전
메서드 호출	`targetInstance.showDetail()`	안전
공유 리소스 해제	`unsubscribe(topic, instance)`	안전
생명주기 시점 쓰기	WScript `register`에서 `this.subscriptions = {...}`	안전
Mixin 소유 쓰기	`instance._popup = {...}` (PopupMixin이 생성+정리)	안전
비소유 속성 null	**`instance.datasetInfo = null`**	**위험**

7.2 제어가 필요한 케이스

외부에서 인스턴스에 접근해야 하는 정당한 케이스가 존재한다:

케이스 1: 공유 리소스 해제

인스턴스의 **속성이 아니라** 외부 시스템에 등록된 참조를 해제하는 경우.

```
// GlobalDataPublisher.js
unsubscribe(topic, instance) {
  const subs = subscriberTable.get(topic);   // ← 외부 시스템(subscriberTable) 수정
  for (const sub of subs) {
    if (sub.instance === instance) subs.delete(sub);
  }
  // instance의 속성은 건드리지 않음
}
```

`unsubscribe` 는 `subscriberTable` (공유 리소스)에서 참조를 제거할 뿐, `instance.subscriptions` 를 null로 만들지 않는다. **외부 시스템의 참조를 정리하되, 인스턴스 자체는 건드리지 않는 것**이 핵심이다.

`disposeAllThreeResources` 의 수정도 이 원칙을 따른다:

- `unsubscribe(topic, instance)` — 공유 리소스(subscriberTable) 정리 → 유지
- `dispose3DTree(instance.appendElement)` — 공유 리소스(GPU) 정리 → 유지
- `instance.datasetInfo = null` — 인스턴스 속성 직접 수정 → **제거**

케이스 2: 읽기 접근 (데이터 조회)

페이지가 오케스트레이터로서 컴포넌트의 데이터를 읽어 판단하는 경우.

```
// ECO before_load.js
'@assetClicked': ({ event, targetInstance }) => {
    targetInstance.showDetail();    // ← 메서드 호출 (인스턴스가 제어)
}
```

```
// README Default JS 템플릿 - 페이지가 데이터셋 정보를 읽어 fetch
'@3dObjectClicked': async ({ event, targetInstance }) => {
    const { datasetInfo } = targetInstance;   // ← 읽기만
    if (datasetInfo?.length) {
        for (const { datasetName, param } of datasetInfo) {
            const data = await fetchData(this, datasetName, param);
        }
    }
}
```

읽기는 인스턴스의 상태를 변경하지 않으므로 항상 안전하다.

케이스 3: Mixin에 의한 기능 확장

Mixin이 인스턴스에 속성과 메서드를 **함께** 주입하고, 주입한 것을 **자기가 정리**하는 경우.

```
// PopupMixin.js
PopupMixin.applyShadowPopupMixin = function(instance, options) {
    // === 생성: _popup 상태 + 메서드를 함께 주입 ===
    instance._popup = { host: null, shadowRoot: null, eventCleanups: [] };
    instance.createPopup = function() { /* ... */ };
    instance.showPopup = function() { /* ... */ };
    instance.destroyPopup = function() {
        // === 정리: 자기가 만든 것을 자기가 정리 ===
        fx.each(cleanup => cleanup(), instance._popup.eventCleanups);
        instance._popup.eventCleanups = [];
        if (instance._popup.host) {
            instance._popup.host.remove();
            instance._popup.host = null;
            instance._popup.shadowRoot = null;
        }
    };
};
```

PopupMixin이 `instance._popup` 을 null하는 것은 안전하다. **자기가 만든 속성을 자기가 정리하기 때문**이다. `destroyPopup()` 은 `_popup.host` , `_popup.shadowRoot` 는 null하지만, **다른 코드가 의존하는 속성(datasetInfo 등)은 건드리지 않는다.**

이것이 `disposeAllThreeResources` 가 `instance.datasetInfo = null` 을 한 것과의 차이다:

- PopupMixin: 자기가 만든 `_popup` 을 자기가 정리 → **소유권 일치**
- disposeAllThreeResources: 컴포넌트가 만든 `datasetInfo` 를 외부에서 정리 → **소유권 불일치**

7.3 판단 기준

외부 코드가 인스턴스에 접근할 때 다음을 확인한다:

```
Q1. 읽기인가 쓰기인가?
    → 읽기: 항상 안전

Q2. 쓰기라면, 누가 만든 속성인가?
    → 자기가 만든 속성 (Mixin 소유): 안전 (소유권 일치)
    → 남이 만든 속성: Q3으로

Q3. 남의 속성을 쓸 때, 인스턴스의 생명주기 훅을 통하는가?
    → Yes (메서드 호출, 이벤트 디스패치): 안전 (인스턴스가 제어)
    → No (직접 할당): 위험 (내부 불변식 파괴 가능)
```

7.4 요약 테이블

패턴	구체 예시	원칙	안전성
읽기	`targetInstance.datasetInfo` (페이지 핸들러)	상태 미변경	안전
메서드 호출	`targetInstance.showDetail()`	인스턴스가 제어	안전
공유 리소스 해제	`unsubscribe(topic, instance)`	외부 시스템만 수정, 인스턴스 미수정	안전
Mixin 소유 쓰기	`instance._popup.host = null` (destroyPopup)	생성자 = 정리자	안전
생명주기 시점 쓰기	`this.subscriptions = {...}` (register)	정의된 시점, 의존 없음	안전
비소유 속성 직접 null	**`instance.datasetInfo = null`**	**소유권 불일치, 의존관계 파괴**	**위험**

부록 A: ECO Asset 컴포넌트 인터벌 누수 사례

대상: UPS, CRAC, PDU, SWBD, TempHumiditySensor (모두 WV3DResourceComponent 상속)

A.1 인터벌 등록

파일: packages/packs/src/ECO/components/Asset/UPSComponent/UPSComponent.js:387

```
d._intervalId = setInterval(() => fetchDatasetAndRender.call(this, d), d.refreshInterval);
//                          ^^^^ 화살표 함수 → this(인스턴스) 캡처
```

인터벌 ID는 `this.datasetInfo[]._intervalId` 에 저장된다.

A.2 인터벌 해제 — stopRefresh()

```
function stopRefresh() {
  const datasetInfo = this.datasetInfo ?? [];
  fx.go(
    datasetInfo,
    fx.filter((d) => d._intervalId),
    fx.each((d) => {
      clearInterval(d._intervalId);
```

```
      d._intervalId = null;
    })
  );
}
```

`_onViewerDestroy` 이외에서 `stopRefresh` 가 호출되는 시나리오:

호출 지점	시나리오
`hideDetail()`	사용자가 팝업 닫기 버튼 클릭
`showDetail()`	새 팝업 열기 직전 (기존 인터벌 정리)
`destroyPopup()` 래퍼	팝업 DOM 제거 시 (→ `_onViewerDestroy`에서만 호출)

사용자가 팝업을 닫지 않은 채 페이지를 이동하면, `_onViewerDestroy()` 의 `stopRefresh()` 가 유일한 해제 기회이다.

A.3 Wkit.disposeAllThreeResources — 순서 문제 (수정 전)

파일: packages/static/files/common/libs/solution/Wkit.js

```
// 수정 전 코드
Wkit.disposeAllThreeResources = function (page) {
  fx.go(
    Wkit.makeIterator(page, 'threeLayer'),
    fx.each((instance) => {
      if (instance.subscriptions) { /* ... */ instance.subscriptions = null; }
      instance.customEvents = null;
      instance.datasetInfo = null;       // ← 문제의 코드
      if (instance.appendElement) { Wkit.dispose3DTree(instance.appendElement); }
    })
  );
};
```

이 함수는 WScript의 `beforeDestroy` 이벤트에서 호출된다. (scaffold 템플릿에서는 주석 처리되어 있으나, 기존에 배포된 페이지 스크립트에서는 활성 상태일 수 있음)

A.4 실행 순서

WScript `beforeDestroy` 는 `WVComponent._beforeDestroy()` 에서 `_onViewerDestroy()` **이전에** 디스패치된다:

파일: packages/common/src/wemb/core/component/WVComponent.ts:590-594

```
_beforeDestroy() {
  if (this._exeMode == EXE_MODE.VIEWER && this._componentEventDispatcher) {
    this._componentEventDispatcher.dispatchEvent(WVComponentScriptEvent.BEFORE_DESTROY);
    //  ↑ 여기서 WScript beforeDestroy 실행 → disposeAllThreeResources → datasetInfo = null
    this._onViewerDestroy();
    //  ↑ 그 다음 실행 → stopRefresh() → this.datasetInfo ?? [] → 빈 배열 → 인터벌 해제 실패
  }
}
```

A.5 결과

```
1. BEFORE_DESTROY 디스패치 → disposeAllThreeResources → instance.datasetInfo = null
2. _onViewerDestroy() → stopRefresh() → this.datasetInfo ?? [] → 빈 배열 순회 → 인터벌 해제 안 됨
3. 인터벌 콜백이 this를 잡고 있음 → 인스턴스 GC 불가
4. 5초마다 fetchDatasetAndRender 계속 호출 (좀비 API 콜)
```

메모리 누수 + 좀비 API 호출 동시 발생.

A.6 정상 시나리오 vs 문제 시나리오

정상 (팝업 닫기 → 페이지 이동):

```
사용자가 팝업 닫기 클릭
  → hideDetail() → stopRefresh() → clearInterval 성공 (datasetInfo 건재)
  → 인터벌 해제됨

페이지 이동
  → removeChildAll() → removeChildAt()
    → _childs.splice, Map.delete (외부 참조 끊김)
    → destroy() → 내부 정리
  → 리턴값 미수신 → 인스턴스 GC 대상 → 전체 수거
```

문제 (팝업 열린 채 페이지 이동):

```
페이지 이동
  → WScript BEFORE_DESTROY → disposeAllThreeResources → datasetInfo = null
  → _onViewerDestroy() → stopRefresh() → 빈 배열 순회 → 인터벌 해제 실패
  → removeChildAll() → 외부 참조 끊김
  → 하지만 setInterval 콜백이 this를 잡고 있음
  → 인스턴스 GC 불가 → 메모리 누수 + 좀비 API 호출
```

A.7 근본 원인 — 외부에서 데이터 캡슐을 깨뜨린 사례

만약 컴포넌트의 `_onViewerDestroy` 내부에서 `this.datasetInfo = null` 을 했다면, `stopRefresh()` 를 먼저 호출한 뒤 null 처리하는 순서 제어가 가능했을 것이다.

그러나 `Wkit.disposeAllThreeResources` 는 **외부 함수**로서, 컴포넌트 내부의 정리 의존관계(인터벌 ID가 `datasetInfo` 에 저장되어 있다는 사실)를 알지 못한 채 속성을 직접 null 처리했다.

이것은 인스턴스가 자기 데이터를 정리할 기회를 빼앗는 **캡슐화 위반**이며, 인터벌 해제 실패 → 메모리 누수 → 좀비 API 호출로 이어진 직접적 원인이다.

원칙: 인스턴스의 속성 정리는 인스턴스 자신의 생명주기 훅(`_onViewerDestroy` , `_onDestroy`)에서 수행해야 한다. 외부에서 `instance.xxx = null` 로 속성을 직접 조작하는 것은 내부 정리 로직의 전제 조건을 무너뜨릴 수 있다.

A.8 수정 내용

`Wkit.disposeAllThreeResources` 에서 수동 null 처리를 모두 제거:

```
// 수정 후 코드
Wkit.disposeAllThreeResources = function (page) {
  fx.go(
    Wkit.makeIterator(page, 'threeLayer'),
    fx.each((instance) => {
      // 1. Subscription 정리 (있는 경우)
      if (instance.subscriptions) {
        fx.go(
          Object.keys(instance.subscriptions),
```

```
          fx.each((topic) => unsubscribe(topic, instance))
        );
      }

      // 2. 3D 리소스 정리
      if (instance.appendElement) {
        Wkit.dispose3DTree(instance.appendElement);
      }
    })
  );

  Wkit.clearSceneBackground(scene);
};
```

제거된 코드:

- `instance.subscriptions = null` — unsubscribe 호출은 유지, null 제거
- `instance.customEvents = null`
- `instance.datasetInfo = null`

근거:

- 인스턴스가 GC될 때 속성도 함께 수거되므로 수동 null 처리 불필요
- `datasetInfo = null` 은 `_onViewerDestroy()` 의 `stopRefresh()` 실행을 방해하여 유해
- `subscriptions`, `customEvents` 도 같은 논리로 불필요

관련 문서

- THIS_PROPERTY_MEMORY_LEAK.md — this 참조 해제 후 속성 잔존 시나리오 (클로저, bind, V8 공유 렉시컬 환경)
- WHY_NULL_UNNECESSARY.md — 외부 null 처리가 불필요한 이유 (GC 원칙 기반)
- WKIT_API.md — Wkit API 레퍼런스
- README.md — 아키텍처 가이드