```
"""
NeuroGraph Cognitive Enhancement Suite — Module 3: Surfacing  v1.1
==================================================================
Changes from v1.0
-----------------
- O(N) full scan replaced with graph.get_active_nodes(threshold) where
  available — the Graph API already exposes this, returning only nodes
  above a voltage threshold.  Falls back to manual scan if the method
  is absent (forward/backward compatibility).
- All magic numbers replaced with CESConfig lookups.
- reset() hook: clears queue and cooldown table when graph is reset.
- Hyperedge snippet uses hyperedge metadata label when present rather
  than always falling back to node text assembly.
- Minor: format_context() returns empty string (not whitespace) so
  callers can use a simple `if context:` check reliably.
"""

from __future__ import annotations

import logging
from dataclasses import dataclass, field
from typing import Any, Dict, List, Optional, Set

from ces_config import CESConfig, make_config

logger = logging.getLogger("neurograph.surfacing")

CONTEXT_HEADER = "<!-- NeuroGraph Surfacing: concepts currently warm in memory --
CONTEXT_FOOTER = "<!-- End surfacing context -->"


# ---------------------------------------------------------------------------
# Data structures
# ---------------------------------------------------------------------------

@dataclass
class SurfacedItem:
    node_id: str
    label: str
    content_snippet: str
    voltage_at_surface: float
    surfaced_at_step: int
    last_seen_step: int
    engaged: bool = False
```

```
    expired: bool = False
    steps_in_queue: int = 0


# ------------------------------------------------------------------------
# SurfacingMonitor
# ------------------------------------------------------------------------

class SurfacingMonitor:
    """
    Post-step activation monitor maintaining a soft surfacing queue.

    Parameters
    ----------
    graph       neuro_foundation.Graph
    vector_db   SimpleVectorDB
    cfg         CESConfig (optional)
    """

    def __init__(
        self,
        graph: Any,
        vector_db: Any,
        cfg: Optional[CESConfig] = None,
    ) -> None:
        self._graph = graph
        self._vector_db = vector_db
        self._cfg = cfg or make_config()

        self._threshold_ratio: float = self._cfg["sm_initial_threshold_ratio"]
        self._queue: List[SurfacedItem] = []
        self._cooldown: Dict[str, int] = {}
        self._current_step: int = 0

        self._total_surfaced: int = 0
        self._total_engaged: int = 0
        self._total_expired: int = 0

    # --------------------------------------------------------------------
    # Public API
    # --------------------------------------------------------------------

    def step(self) -> List[SurfacedItem]:
        """Run one monitoring cycle. Call after graph.step()."""
        self._current_step += 1
        self._expire_stale()
        self._scan_for_candidates()
```

```
        return [item for item in self._queue if not item.expired]

    def format_context(self) -> str:
        """Prompt context block. Returns empty string if queue is empty."""
        active = [item for item in self._queue if not item.expired]
        if not active:
            return ""
        lines = [CONTEXT_HEADER]
        for item in active:
            lines.append(f"- **{item.label}**: {item.content_snippet}")
        lines.append(CONTEXT_FOOTER)
        return "\n".join(lines)

    def signal_engagement(self, fired_node_ids: List[str]) -> None:
        """
        Signal which nodes fired during the most recent LLM turn.
        Drives threshold self-tuning.
        """
        fired_set: Set[str] = set(fired_node_ids)
        for item in self._queue:
            if item.node_id in fired_set and not item.engaged:
                item.engaged = True
                self._total_engaged += 1
                self._adjust_threshold(self._cfg["sm_engage_delta"])

    def reset(self) -> None:
        """Clear queue and cooldowns when the graph is reset mid-session."""
        self._queue.clear()
        self._cooldown.clear()
        self._current_step = 0
        logger.info("SurfacingMonitor reset")

    def stats(self) -> Dict[str, Any]:
        active = [i for i in self._queue if not i.expired]
        return {
            "queue_size": len(active),
            "threshold_ratio": round(self._threshold_ratio, 4),
            "threshold_voltage": round(
                self._threshold_ratio
                * self._graph.config.get("default_threshold", 1.0),
                4,
            ),
            "total_surfaced": self._total_surfaced,
            "total_engaged": self._total_engaged,
            "total_expired": self._total_expired,
            "engagement_rate": round(
                self._total_engaged / max(self._total_surfaced, 1), 3
```

```
        ),
        "current_step": self._current_step,
    }

# ------------------------------------------------------------------
# Internal
# ------------------------------------------------------------------

def _threshold_voltage(self) -> float:
    return (
        self._threshold_ratio
        * self._graph.config.get("default_threshold", 1.0)
    )

def _adjust_threshold(self, delta: float) -> None:
    cfg = self._cfg
    self._threshold_ratio = max(
        cfg["sm_min_threshold_ratio"],
        min(
            cfg["sm_max_threshold_ratio"],
            self._threshold_ratio + delta * cfg["sm_threshold_lr"],
        ),
    )

def _get_candidate_nodes(self, threshold_v: float) -> List[tuple]:
    """
    Return (voltage, node_id) pairs above threshold_v.
    Uses graph.get_active_nodes() if available (O(active) not O(N)),
    falls back to manual scan.
    """
    if hasattr(self._graph, "get_active_nodes"):
        try:
            return [
                (v, nid)
                for nid, v in self._graph.get_active_nodes(threshold=threshol
            ]
        except Exception as exc:
            logger.debug("get_active_nodes failed, falling back: %s", exc)

    # Fallback: full scan
    return [
        (node.voltage, nid)
        for nid, node in self._graph.nodes.items()
        if node.voltage >= threshold_v
    ]

def _expire_stale(self) -> None:
```

```
    threshold_v = self._threshold_voltage()
    cfg = self._cfg
    for item in self._queue:
        if item.expired:
            continue
        item.steps_in_queue += 1
        node = self._graph.nodes.get(item.node_id)
        current_v = node.voltage if node else 0.0
        timed_out = item.steps_in_queue > cfg["sm_expiry_steps"]
        dropped = current_v < threshold_v * 0.5
        if timed_out or dropped:
            item.expired = True
            self._total_expired += 1
            if not item.engaged:
                self._adjust_threshold(cfg["sm_expiry_delta"])
            item.last_seen_step = self._current_step
    self._queue = [
        i for i in self._queue
        if not i.expired or self._current_step - i.last_seen_step < 5
    ]

def _scan_for_candidates(self) -> None:
    cfg = self._cfg
    max_q: int = cfg["sm_max_queue_size"]
    active_count = len([i for i in self._queue if not i.expired])
    if active_count >= max_q:
        return

    threshold_v = self._threshold_voltage()
    queued_ids = {i.node_id for i in self._queue if not i.expired}
    slots = max_q - active_count

    candidates = self._get_candidate_nodes(threshold_v)
    candidates = [
        (v, nid) for v, nid in candidates
        if nid not in queued_ids
        and self._current_step >= self._cooldown.get(nid, 0)
    ]
    candidates.sort(reverse=True)

    for voltage, nid in candidates[:slots]:
        item = self._build_node_item(nid, voltage)
        if item is not None:
            self._queue.append(item)
            self._cooldown[nid] = (
                self._current_step + cfg["sm_requeue_cooldown_steps"]
            )
```

```
            self._total_surfaced += 1
            slots -= 1

    if slots > 0:
        self._scan_hyperedges(threshold_v, queued_ids, slots)

def _build_node_item(
    self, node_id: str, voltage: float
) -> Optional[SurfacedItem]:
    node = self._graph.nodes.get(node_id)
    if node is None:
        return None
    label = (
        (node.metadata or {}).get("label")
        or (node.metadata or {}).get("source", "")
        or node_id[:32]
    )
    snippet = self._fetch_node_snippet(node_id, str(label))
    if not snippet:
        return None
    return SurfacedItem(
        node_id=node_id,
        label=str(label)[:80],
        content_snippet=snippet,
        voltage_at_surface=round(voltage, 4),
        surfaced_at_step=self._current_step,
        last_seen_step=self._current_step,
    )

def _fetch_node_snippet(self, node_id: str, label: str) -> str:
    try:
        if hasattr(self._vector_db, "get_text"):
            text = self._vector_db.get_text(node_id)
            if text:
                return text[:200].strip()
    except Exception:
        pass
    try:
        if label:
            results = self._vector_db.query_similar(
                label,
                k=self._cfg["sm_surface_k"],
                threshold=self._cfg["sm_surface_threshold"],
            )
            if results:
                return results[0].get("content", "")[:200].strip()
    except Exception as exc:
```

```
            logger.debug("Snippet fetch failed for %s: %s", node_id, exc)
        return ""

    def _scan_hyperedges(
        self,
        threshold_v: float,
        queued_ids: Set[str],
        slots: int,
    ) -> None:
        cfg = self._cfg
        min_ratio: float = cfg["sm_he_min_active_ratio"]

        for he_id, he in self._graph.hyperedges.items():
            if slots <= 0:
                break
            if he.refractory_remaining > 0:
                continue
            if he_id in queued_ids:
                continue
            if self._current_step < self._cooldown.get(he_id, 0):
                continue

            active_members = [
                nid for nid in he.member_node_ids
                if nid in self._graph.nodes
                and self._graph.nodes[nid].voltage >= threshold_v
            ]
            ratio = len(active_members) / max(len(he.member_node_ids), 1)
            if ratio < min_ratio:
                continue

            # v1.1: prefer hyperedge metadata label
            label = (he.metadata or {}).get("label", he_id[:32])
            snippet = self._fetch_hyperedge_snippet(he, active_members, str(label
            if not snippet:
                continue

            item = SurfacedItem(
                node_id=he_id,
                label=f"[cluster] {label}"[:80],
                content_snippet=snippet,
                voltage_at_surface=round(ratio, 4),
                surfaced_at_step=self._current_step,
                last_seen_step=self._current_step,
            )
            self._queue.append(item)
            self._cooldown[he_id] = (
```

```
            self._current_step + cfg["sm_requeue_cooldown_steps"]
        )
        self._total_surfaced += 1
        slots -= 1

def _fetch_hyperedge_snippet(
    self, he: Any, active_members: List[str], label: str
) -> str:
    # Try the hyperedge label as a semantic query first
    try:
        if label and not label.startswith("he_"):
            results = self._vector_db.query_similar(
                label,
                k=self._cfg["sm_surface_k"],
                threshold=self._cfg["sm_surface_threshold"],
            )
            if results:
                return results[0].get("content", "")[:200].strip()
    except Exception:
        pass

    # Fallback: assemble from active member text
    parts = []
    for nid in active_members[:3]:
        try:
            if hasattr(self._vector_db, "get_text"):
                text = self._vector_db.get_text(nid)
                if text:
                    parts.append(text[:80].strip())
        except Exception:
            pass
    return " / ".join(p for p in parts if p)[:200]
```