```python
"""
ERES Institute for New Age Cybernetics ~ PlayNAC "KERNEL" (v6.3)
Empirical Realtime Education System × New Age Cybernetic Game Theory

Complete production-ready codebase implementing:
- EarnedPath (EP) Merit-based learning graphs
- GiantERP (GERP) Global resource planning
- Bio-Energetic Proof-of-Work (BEE) EEG-driven mining
- BERC Consensus Bio-Electric Ratings protocol
- MediaProcessor Real-time media transforms
- Mandala-VERTECA HFVN Gesture & voice navigation
- 17x7 Keyword Matrix Game Theory Framework
- Global Actuary Investor Authority (GAIA) integration
"""

import os
import sys
import time
import json
import hashlib
import sqlite3
import logging
import numpy as np
import asyncio
from abc import ABC, abstractmethod
from dataclasses import dataclass, field
from typing import Any, Dict, List, Optional, Tuple, Union
from enum import Enum
from functools import wraps
from collections import defaultdict
import threading
import queue

#
==========================================================================
=====
# CORE CONFIGURATION & MODELS
#
==========================================================================
=====

class ConfigManager:
    """Multi-environment configuration manager with validation"""

    def __init__(self, env_files: List[str] = [".env"]):
        self.env_files = env_files
        self._loaded = False
```

```
        self.config_cache = {}

    def load_env(self) -> None:
        if self._loaded:
            return

        for file in self.env_files:
            if os.path.isfile(file):
                with open(file) as f:
                    for line in f:
                        if line.strip().startswith('#') or '=' not in
line:
                            continue
                        key, val = line.strip().split('=', 1)
                        os.environ.setdefault(key, val)
        self._loaded = True

    def validate(self, required_keys: List[str]) -> None:
        missing = [k for k in required_keys if k not in os.environ]
        if missing:
            raise KeyError(f"Missing required config keys: {missing}")

    def get(self, key: str, default: Any = None) -> Any:
        return os.environ.get(key, default)

@dataclass
class Block:
    """Blockchain block structure"""
    index: int
    timestamp: float
    data: Dict[str, Any]
    previous_hash: str
    nonce: int
    hash: str
    ep_value: float = 0.0
    consensus_links: List[str] = field(default_factory=list)

@dataclass
class MediaTask:
    """Media processing task with EP integration"""
    id: str
    input_frame: Any
    task_type: str
    nonce: int
    timestamp: float
    ep_value: float = 0.0
    complexity_score: float = 0.0
```

```
    gaia_domain: str = ""

@dataclass
class JASLink:
    """Joint Attention Signature consensus link"""
    source: str
    target: str
    weight: float
    timestamp: float
    berc_rating: float = 0.0

@dataclass
class EPNode:
    """EarnedPath node with PERT/CPM integration"""
    node_id: str
    domain_category: str  # Maps to 17x7 keyword matrix
    dependencies: List['EPNode'] = field(default_factory=list)
    state: 'EPState' = None
    result: Any = None
    merit_score: float = 0.0
    gaia_weight: float = 1.0

    def __post_init__(self):
        if self.state is None:
            self.state = EPState.LOCKED

class EPState(Enum):
    LOCKED = 0
    UNLOCKED = 1
    COMPLETED = 2

#
===========================================================================
=====
# KEYWORD MATRIX & GAIA FRAMEWORK
#
===========================================================================
=====

class KeywordMatrix:
    """17 rows x 7 subjects keyword categorization system"""

    MATRIX = {
        1: ["NAC", "EarnedPath", "Resource", "Prime", "Matter",
"Cybernetics", "GiantERP"],
        2: ["Water", "Food", "Shelter", "Work", "Love", "Overall",
"Always"],
```

```
        3: ["Weather", "Which", "Entity", "Thought", "Energy", "Sound",
"Good"],
        4: ["Engage", "Connect", "Collaborate", "Transact", "Mobile",
"Analytics", "Learn"],
        5: ["Self", "Family", "Community", "Nation", "Plane", "World",
"Universe"],
        6: ["Personal", "Public", "Private", "Syntax", "Class", "Method",
"Variable"],
        7: ["Social", "Economic", "Political", "Legal", "Technical",
"Administrative", "History"],
        8: ["Physical", "Data", "Network", "Transport", "Session",
"Presentation", "Application"],
        9: ["Of", "Order", "Wisdom", "Belief", "Power", "Imagination",
"Will"],
        10: ["Principle", "Mind", "Soul", "Spirit", "Life", "Truth",
"Evol"],
        11: ["With", "Who", "What", "Where", "When", "Why", "How"],
        12: ["Help", "Use", "Energy", "Law", "Common", "Risk", "System"],
        13: ["Lust", "Gluttony", "Greed", "Sloth", "Wrath", "Envy",
"Pride"],
        14: ["Grammar", "Rhetoric", "Logic", "Arithmetic", "Geometry",
"Music", "Astronomy"],
        15: ["Chastity", "Temperance", "Charity", "Diligence", "Patience",
"Kindness", "Humility"],
        16: ["Prudence", "Justice", "Remediation", "Courage", "Faith",
"Hope", "Compassion"],
        17: ["Joy", "Anger", "Anxiety", "Pensiveness", "Grief", "Fear",
"Fright"]
    }

    @classmethod
    def categorize_intent(cls, text: str) -> Tuple[int, int]:
        """Map text to matrix coordinates"""
        text_lower = text.lower()
        for row, keywords in cls.MATRIX.items():
            for col, keyword in enumerate(keywords):
                if keyword.lower() in text_lower:
                    return (row, col)
        return (1, 0)  # Default to NAC

    @classmethod
    def get_domain_weight(cls, row: int, col: int) -> float:
        """Calculate GAIA domain weighting"""
        base_weight = 1.0
        # Higher weight for core domains (rows 1-7)
        if row <= 7:
            base_weight *= 1.5
```

```
        # Virtue/Vice balance (rows 13-16)
        if row in [15, 16]:  # Virtues
            base_weight *= 1.2
        elif row == 13:  # Vices
            base_weight *= 0.8
        return base_weight

class GAIAManager:
    """Global Actuary Investor Authority - 23 Principal Industry Domain
Leaders"""

    PRINCIPAL_DOMAINS = [
        "Energy", "Water", "Food", "Shelter", "Healthcare", "Education",
"Transportation",
        "Communication", "Finance", "Manufacturing", "Agriculture",
"Technology", "Defense",
        "Environment", "Mining", "Construction", "Tourism",
"Entertainment", "Research",
        "Governance", "Spirituality", "Arts", "Social"
    ]

    def __init__(self):
        self.domain_leaders = {domain: 0.0 for domain in
self.PRINCIPAL_DOMAINS}
        self.voting_threshold = 0.6
        self.total_votes = 0

    def cast_vote(self, domain: str, weight: float) -> bool:
        """Cast weighted vote for domain ruling"""
        if domain in self.domain_leaders:
            self.domain_leaders[domain] += weight
            self.total_votes += 1
            return True
        return False

    def get_consensus(self) -> Dict[str, float]:
        """Calculate consensus ratings S=E/23(1/x) #C=R*P/M @A=C(ME)²³"""
        if self.total_votes == 0:
            return {}

        consensus = {}
        for domain, votes in self.domain_leaders.items():
            # Apply formula: S=E/23(1/x) where E=votes, x=domain_index
            domain_idx = self.PRINCIPAL_DOMAINS.index(domain) + 1
            consensus[domain] = votes / (23 * (1 / domain_idx))

        return consensus
```

```
#
==========================================================================
=====
# STORAGE & PERSISTENCE
#
==========================================================================
=====

class Storage:
    """SQLite-based persistent storage for blockchain and EP nodes"""

    def __init__(self, path: str = 'playnac.db'):
        self.conn = sqlite3.connect(path, check_same_thread=False)
        self._init_schema()
        self._lock = threading.Lock()

    def _init_schema(self):
        with self._lock:
            c = self.conn.cursor()

            # Blocks table
            c.execute('''
                CREATE TABLE IF NOT EXISTS blocks (
                    idx INTEGER PRIMARY KEY,
                    timestamp REAL,
                    data TEXT,
                    prev_hash TEXT,
                    nonce INTEGER,
                    hash TEXT,
                    ep_value REAL,
                    consensus_links TEXT
                )
            ''')

            # EP Nodes table
            c.execute('''
                CREATE TABLE IF NOT EXISTS ep_nodes (
                    node_id TEXT PRIMARY KEY,
                    domain_category TEXT,
                    state INTEGER,
                    merit_score REAL,
                    gaia_weight REAL,
                    dependencies TEXT,
                    result TEXT
                )
            ''')
```

```
            # JAS Links table
            c.execute('''
                CREATE TABLE IF NOT EXISTS jas_links (
                    id INTEGER PRIMARY KEY AUTOINCREMENT,
                    source TEXT,
                    target TEXT,
                    weight REAL,
                    timestamp REAL,
                    berc_rating REAL
                )
            ''')

            self.conn.commit()

    def save_block(self, block: Block) -> None:
        with self._lock:
            c = self.conn.cursor()
            c.execute('''
                INSERT OR REPLACE INTO blocks
                (idx, timestamp, data, prev_hash, nonce, hash, ep_value,
consensus_links)
                VALUES (?, ?, ?, ?, ?, ?, ?, ?)
            ''', (
                block.index, block.timestamp, json.dumps(block.data),
                block.previous_hash, block.nonce, block.hash,
                block.ep_value, json.dumps(block.consensus_links)
            ))
            self.conn.commit()

    def load_blocks(self) -> List[Dict]:
        with self._lock:
            c = self.conn.cursor()
            c.execute('''
                SELECT idx, timestamp, data, prev_hash, nonce, hash,
ep_value, consensus_links
                FROM blocks ORDER BY idx
            ''')
            rows = c.fetchall()

            blocks = []
            for row in rows:
                blocks.append({
                    'index': row[0],
                    'timestamp': row[1],
                    'data': json.loads(row[2]),
                    'previous_hash': row[3],
```

```
                    'nonce': row[4],
                    'hash': row[5],
                    'ep_value': row[6],
                    'consensus_links': json.loads(row[7]) if row[7] else
[]
                })
            return blocks

    def save_ep_node(self, node: EPNode) -> None:
        with self._lock:
            c = self.conn.cursor()
            c.execute('''
                INSERT OR REPLACE INTO ep_nodes
                (node_id, domain_category, state, merit_score,
gaia_weight, dependencies, result)
                VALUES (?, ?, ?, ?, ?, ?, ?)
            ''', (
                node.node_id, node.domain_category, node.state.value,
                node.merit_score, node.gaia_weight,
                json.dumps([dep.node_id for dep in node.dependencies]),
                json.dumps(node.result) if node.result else None
            ))
            self.conn.commit()

#
===========================================================================
=====
# BIO-ENERGETIC PROOF OF WORK (BEE)
#
===========================================================================
=====

class AuraScanner:
    """EEG/biometric device interface with entropy calculation"""

    def __init__(self):
        self.device_connected = True
        self.sample_rate = 256
        self.calibration_data = None

    def capture(self, duration: float = 1.0) -> np.ndarray:
        """Capture EEG data - stub implementation"""
        # In production, interface with Muse, OpenBCI, etc.
        samples = int(self.sample_rate * duration)
        # Simulate alpha/beta/theta wave patterns
        t = np.linspace(0, duration, samples)
        alpha = 0.3 * np.sin(2 * np.pi * 10 * t)  # 10Hz alpha
```

```
        beta = 0.2 * np.sin(2 * np.pi * 20 * t)    # 20Hz beta
        theta = 0.4 * np.sin(2 * np.pi * 6 * t)    # 6Hz theta
        noise = 0.1 * np.random.randn(samples)
        return alpha + beta + theta + noise

    def calculate_spectral_entropy(self, signal: np.ndarray) -> float:
        """Calculate spectral entropy for EP generation"""
        # FFT and power spectral density
        fft = np.fft.fft(signal)
        psd = np.abs(fft) ** 2
        psd_norm = psd / np.sum(psd)

        # Shannon entropy
        entropy = -np.sum(psd_norm * np.log2(psd_norm + 1e-10))
        return entropy / np.log2(len(psd_norm))  # Normalize to [0,1]

    def is_device_connected(self) -> bool:
        return self.device_connected

class BioPoW:
    """Bio-Energetic Proof of Work engine"""

    def __init__(self, secret_key: str, gerp_factor: float = 0.618):
        self.secret_key = secret_key
        self.scanner = AuraScanner()
        self.gerp_factor = gerp_factor  # Golden ratio factor
        self.difficulty_adjustment = 1.0
        self.target_block_time = 60.0  # seconds

    def generate_ep(self) -> float:
        """Generate Entropic Potential from biometric data"""
        if not self.scanner.is_device_connected():
            # Fallback to synthetic data
            raw_signal = np.random.randn(256)
        else:
            raw_signal = self.scanner.capture()

        entropy = self.scanner.calculate_spectral_entropy(raw_signal)

        # Apply GERP factor and secret key hash
        key_hash = hashlib.sha256(self.secret_key.encode()).hexdigest()
        key_factor = int(key_hash[:8], 16) / (2**32)  # Normalize to [0,1]

        ep = entropy * self.gerp_factor * key_factor
        return max(0.001, min(1.0, ep))  # Clamp to valid range
```

```
    def validate(self, ep_value: float, target: float, tolerance: float =
0.1) -> bool:
        """Validate EP meets difficulty target"""
        adjusted_target = target * self.difficulty_adjustment
        return ep_value >= (adjusted_target - tolerance)

    def adjust_difficulty(self, last_block_times: List[float]) -> None:
        """Adjust mining difficulty based on block times"""
        if len(last_block_times) < 2:
            return

        avg_time = sum(last_block_times) / len(last_block_times)
        ratio = avg_time / self.target_block_time

        # Gradual adjustment
        if ratio > 1.1:  # Too slow
            self.difficulty_adjustment *= 0.9
        elif ratio < 0.9:  # Too fast
            self.difficulty_adjustment *= 1.1

        # Clamp difficulty
        self.difficulty_adjustment = max(0.1, min(10.0,
self.difficulty_adjustment))

#
==========================================================================
=====
# EARNEDPATH SIMULATION ENGINE
#
==========================================================================
=====

class MeritCalculator:
    """Calculate merit scores based on actions and domain weights"""

    def __init__(self):
        self.base_multiplier = 1.0
        self.virtue_bonus = 1.2
        self.vice_penalty = 0.8

    def calculate_merit(self, actions: List[Dict], domain_row: int) ->
float:
        """Calculate merit with virtue/vice modifiers"""
        base_merit = sum(action.get('value', 0) * action.get('weight', 1)
                        for action in actions)

        # Apply virtue/vice modifiers based on keyword matrix
```

```
        if domain_row == 15 or domain_row == 16:  # Virtues
            base_merit *= self.virtue_bonus
        elif domain_row == 13:  # Vices
            base_merit *= self.vice_penalty

        return max(0.0, base_merit)

class SimulationEngine:
    """PERT/CPM-based EarnedPath simulation engine"""

    def __init__(self):
        self.nodes: Dict[str, EPNode] = {}
        self.merit_calculator = MeritCalculator()
        self.simulation_time = 0.0
        self.keyword_matrix = KeywordMatrix()

    def create_node(self, node_id: str, domain_text: str = "",
                    dependencies: List[str] = None) -> EPNode:
        """Create EP node with domain categorization"""
        row, col = self.keyword_matrix.categorize_intent(domain_text)
        domain_category = f"{row}:{col}"

        # Resolve dependencies
        dep_nodes = []
        if dependencies:
            dep_nodes = [self.nodes[dep_id] for dep_id in dependencies
                         if dep_id in self.nodes]

        node = EPNode(
            node_id=node_id,
            domain_category=domain_category,
            dependencies=dep_nodes,
            gaia_weight=self.keyword_matrix.get_domain_weight(row, col)
        )

        self.nodes[node_id] = node
        return node

    def setup_scenario(self, config: Dict[str, Any]) -> None:
        """Build simulation scenario from configuration"""
        scenario_nodes = config.get('nodes', [])

        for node_config in scenario_nodes:
            self.create_node(
                node_config['id'],
                node_config.get('domain', ''),
                node_config.get('dependencies', [])
```

```
            )

    def step(self) -> Dict[str, Any]:
        """Advance simulation one time step"""
        self.simulation_time += 1.0
        unlocked_count = 0
        completed_count = 0

        # Check for nodes that can be unlocked
        for node in self.nodes.values():
            if node.state == EPState.LOCKED:
                if all(dep.state == EPState.COMPLETED for dep in
node.dependencies):
                    node.state = EPState.UNLOCKED
                    unlocked_count += 1

            # Simulate node completion (probabilistic)
            elif node.state == EPState.UNLOCKED:
                completion_prob = 0.3 * node.gaia_weight
                if np.random.random() < completion_prob:
                    # Calculate merit for completion
                    row, col = map(int, node.domain_category.split(':'))
                    actions = [{'value': 1.0, 'weight': node.gaia_weight}]
                    node.merit_score =
self.merit_calculator.calculate_merit(actions, row)
                    node.state = EPState.COMPLETED
                    completed_count += 1

        return {
            'time': self.simulation_time,
            'unlocked': unlocked_count,
            'completed': completed_count,
            'total_nodes': len(self.nodes)
        }

    def get_total_merit(self) -> float:
        """Calculate total accumulated merit"""
        return sum(node.merit_score for node in self.nodes.values()
                  if node.state == EPState.COMPLETED)

#
==========================================================================
=====
# BERC CONSENSUS & JAS LINKS
#
==========================================================================
=====
```

```
class JASConsensus:
    """Joint Attention Signature consensus mechanism"""

    def __init__(self, threshold: float = 0.6):
        self.threshold = threshold
        self.links: List[JASLink] = []
        self.node_registry: Dict[str, float] = {}
        self.rating_validator = RatingValidator()

    def create_link(self, src: MediaTask, tgt: MediaTask, correlation:
float) -> JASLink:
        """Create JAS link between media tasks"""
        # Calculate BERC rating
        berc_rating = self.rating_validator.calculate_berc_rating(src,
tgt)

        link = JASLink(
            source=src.id,
            target=tgt.id,
            weight=correlation,
            timestamp=time.time(),
            berc_rating=berc_rating
        )

        self.links.append(link)
        self._update_node_registry(src.id, correlation)
        self._update_node_registry(tgt.id, correlation)

        return link

    def _update_node_registry(self, node_id: str, weight: float) -> None:
        """Update node reputation in registry"""
        if node_id in self.node_registry:
            self.node_registry[node_id] = (self.node_registry[node_id] +
weight) / 2
        else:
            self.node_registry[node_id] = weight

    def validate(self, task_id: str) -> bool:
        """Validate task using consensus threshold"""
        related_links = [link for link in self.links
                        if link.source == task_id or link.target ==
task_id]

        if not related_links:
            return True  # New nodes pass by default
```

```
        avg_weight = sum(link.weight for link in related_links) /
len(related_links)
        avg_berc = sum(link.berc_rating for link in related_links) /
len(related_links)

        combined_score = (avg_weight + avg_berc) / 2
        return combined_score >= self.threshold

class RatingValidator:
    """Bio-Ecologic Ratings Codex (BERC) validator"""

    def __init__(self):
        self.ecological_factors = {
            'energy_efficiency': 0.3,
            'resource_usage': 0.25,
            'sustainability': 0.25,
            'social_impact': 0.2
        }

    def calculate_berc_rating(self, src: MediaTask, tgt: MediaTask) ->
float:
        """Calculate Bio-Ecologic rating between tasks"""
        # Stub implementation - in production would analyze:
        # - Energy consumption of media processing
        # - Resource efficiency
        # - Environmental impact
        # - Social/community benefit

        base_rating = 0.5

        # Factor in task complexity (lower complexity = higher rating)
        complexity_factor = 1.0 - (src.complexity_score +
tgt.complexity_score) / 2

        # Factor in EP values (higher EP = higher rating)
        ep_factor = (src.ep_value + tgt.ep_value) / 2

        rating = base_rating * complexity_factor * (1 + ep_factor)
        return max(0.0, min(1.0, rating))

#
============================================================================
=====
# MEDIA PROCESSING
```

```
#
================================================================================
=====

class MediaProcessor:
    """Real-time media processing with MD complexity validation"""

    def __init__(self, threshold: float = 0.07):
        self.threshold = threshold
        self.processing_cache = {}

    def calculate_md_complexity(self, frame: np.ndarray) -> float:
        """Calculate Mandala-Dimensional complexity"""
        if frame.ndim == 3:  # Color image
            # Convert to grayscale for analysis
            gray = np.mean(frame, axis=2).astype(np.uint8)
        else:
            gray = frame.astype(np.uint8)

        # Calculate histogram entropy
        hist, _ = np.histogram(gray, bins=256, range=(0, 256))
        hist_norm = hist / np.sum(hist + 1e-10)
        entropy = -np.sum(hist_norm * np.log2(hist_norm + 1e-10))

        return entropy / 8.0  # Normalize to [0,1]

    def validate_md_complexity(self, frame: np.ndarray) -> bool:
        """Validate frame meets MD complexity threshold"""
        complexity = self.calculate_md_complexity(frame)
        return complexity > self.threshold

    def process_media_task(self, task: MediaTask) -> np.ndarray:
        """Process media task with validation"""
        # Validate input complexity
        if not self.validate_md_complexity(task.input_frame):
            raise ValueError(f"MD complexity
{self.calculate_md_complexity(task.input_frame):.3f} below threshold
{self.threshold}")

        # Store complexity score
        task.complexity_score =
self.calculate_md_complexity(task.input_frame)

        # Apply processing based on task type
        if task.task_type == 'style_transfer':
            return self._style_transfer(task.input_frame)
        elif task.task_type == 'enhancement':
```

```
            return self._enhance_frame(task.input_frame)
        else:
            return task.input_frame

    def _style_transfer(self, frame: np.ndarray) -> np.ndarray:
        """Apply artistic style transfer"""
        # Stub - in production would use neural style transfer
        if frame.ndim == 3:
            # Simple color adjustment
            enhanced = frame * 1.2
            enhanced = np.clip(enhanced, 0, 255)
            return enhanced.astype(np.uint8)
        return frame

    def _enhance_frame(self, frame: np.ndarray) -> np.ndarray:
        """Enhance frame quality"""
        # Simple sharpening filter
        if frame.ndim == 3:
            enhanced = frame + 0.3 * (frame - np.roll(frame, 1, axis=0))
            return np.clip(enhanced, 0, 255).astype(np.uint8)
        return frame

#
===========================================================================
=====
# NAVIGATION & VOICE INTERFACES
#
===========================================================================
=====

class IntentParser:
    """Intent parsing with 17x7 keyword matrix integration"""

    def __init__(self):
        self.keyword_matrix = KeywordMatrix()

    def parse(self, text: str) -> Tuple[str, Dict, Tuple[int, int]]:
        """Parse intent and map to keyword matrix"""
        text_lower = text.lower()
        matrix_coords = self.keyword_matrix.categorize_intent(text)

        # Rule-based intent classification
        if any(word in text_lower for word in ['allocate', 'resource',
'distribute']):
            return 'allocate_resource', {'amount':
self._extract_number(text)}, matrix_coords
```

```
        elif any(word in text_lower for word in ['mine', 'block',
'process']):
            return 'mine_block', {}, matrix_coords
        elif any(word in text_lower for word in ['status', 'report',
'show']):
            return 'get_status', {}, matrix_coords
        elif any(word in text_lower for word in ['help', 'assist',
'guide']):
            return 'get_help', {'domain': self._extract_domain(text)},
matrix_coords
        else:
            return 'unknown', {}, matrix_coords

    def _extract_number(self, text: str) -> Optional[float]:
        """Extract numeric value from text"""
        import re
        numbers = re.findall(r'\d+\.?\d*', text)
        return float(numbers[0]) if numbers else None

    def _extract_domain(self, text: str) -> str:
        """Extract domain reference from text"""
        for domain in GAIAManager.PRINCIPAL_DOMAINS:
            if domain.lower() in text.lower():
                return domain
        return "general"

class MandalaTranslator:
    """Mandala-VERTECA gesture translation"""

    SYMBOL_MAP = {
        'thumb_palm': ('🜁', 'home', 'navigate_home'),
        'index_mudra': ('🜂', 'back', 'navigate_back'),
        'middle_press': ('🜃', 'select', 'execute_selection'),
        'ring_swirl': ('🜄', 'menu', 'open_menu'),
        'pinky_wave': ('☿', 'voice', 'activate_voice'),
    }

    def translate(self, gesture: str) -> Tuple[str, str, str]:
        """Translate gesture to symbol, name, and action"""
        return self.SYMBOL_MAP.get(gesture, ('', '', 'unknown'))

    def execute(self, gesture: str, kernel: 'PlayNACKernel') -> str:
        """Execute gesture command on kernel"""
        symbol, name, action = self.translate(gesture)

        if action == 'navigate_home':
            return kernel.get_status()
```

```
        elif action == 'navigate_back':
            return "Navigation: Back"
        elif action == 'execute_selection':
            return "Selection executed"
        elif action == 'open_menu':
            return "Menu opened"
        elif action == 'activate_voice':
            return "Voice mode activated"
        else:
            return f"Unknown gesture: {gesture}"

class GreenBoxEnvironment:
    """Hands-Free Virtual Navigation environment"""

    def __init__(self):
        self.translator = MandalaTranslator()
        self.active = False
        self.current_zone = "home"
        self.gesture_queue = queue.Queue()

    def activate(self) -> None:
        self.active = True

    def deactivate(self) -> None:
        self.active = False

    def on_gesture(self, gesture: str) -> Tuple[str, str, str]:
        """Handle gesture input"""
        if self.active:
            self.gesture_queue.put(gesture)
            return self.translator.translate(gesture)
        return ('', '', '')

    def on_voice(self, text: str, kernel: 'PlayNACKernel') -> str:
        """Handle voice input"""
        if self.active:
```