Prompt - Build a multi agent framework for building a single player game as per user description

Response -

To build a multi-agent framework for a single-player game, we can create a system where each agent represents a different aspect or character in the game. Here's a plan for the framework:

1. **Game Master Agent**: This agent will manage the overall game state, rules, and interactions between other agents.
2. **Player Agent**: This agent will represent the player in the game, handling player actions and decisions.
3. **NPC Agents**: These agents will represent non-player characters (NPCs) in the game, each with their own behaviors and interactions.
4. **Environment Agent**: This agent will manage the game environment, including the map, objects, and events.

### Technical Plan

1. **Game Master Agent**:
   - Manages the game state and rules.
   - Handles interactions between the Player Agent and NPC Agents.
   - Updates the game state based on player actions and NPC behaviors.

2. **Player Agent**:
   - Represents the player in the game.
   - Handles player input and decisions.
   - Communicates with the Game Master Agent to update the game state.

3. **NPC Agents**:
   - Each NPC Agent will have its own behavior and decision-making logic.
   - Communicates with the Game Master Agent to update the game state based on their actions.

4. **Environment Agent**:
   - Manages the game environment, including the map and objects.

   - Handles events and updates the game state accordingly.

### Implementation Steps

1. **Create the Game Master Agent**:
   - Define the game state and rules.
   - Implement logic to handle interactions between the Player Agent and NPC Agents.

2. **Create the Player Agent**:
   - Implement logic to handle player input and decisions.
   - Define communication with the Game Master Agent.

3. **Create NPC Agents**:
   - Implement behavior and decision-making logic for each NPC.
   - Define communication with the Game Master Agent.

4. **Create the Environment Agent**:
   - Implement logic to manage the game environment.
   - Handle events and update the game state.

Let me know if you confirm this plan, and I will proceed with generating the agents.

To proceed with building the multi-agent framework for a single-player game, I will generate the initial code for the agents based on the plan outlined earlier. Here's the summary of the agents and their roles:

1. **Game Master Agent**: Manages the game state, rules, and interactions.
2. **Player Agent**: Represents the player and handles player actions.
3. **NPC Agents**: Represent non-player characters with their own behaviors.
4. **Environment Agent**: Manages the game environment, including the map and objects.

I will now generate the initial code for these agents. Please confirm if you agree with this plan.

```
def handle_player_input():
    """Handle player input and update the game state."""
    last_message = env.get_last_message()
    if last_message["role"] == "user":
        env.add_reply(f"Player action: {last_message['content']}")

def handle_npc_behavior(npc_id: str):
    """Handle NPC behavior and update the game state.

    npc_id: The ID of the NPC.
    """
    env.add_reply(f"NPC {npc_id} is taking action.")

def update_environment():
    """Update the game environment and handle events."""
    env.add_reply("Environment updated.")

def manage_game_state():
    """Manage the game state, rules, and interactions."""
    env.add_reply("Game state managed.")

tool_registry = env.get_tool_registry(new=True)
tool_registry.register_tool(handle_player_input)
tool_registry.register_tool(handle_npc_behavior)
tool_registry.register_tool(update_environment)
tool_registry.register_tool(manage_game_state)

prompt = {"role": "system", "content": "You are a game master
managing the game state, player, NPCs, and environment."}
env.completions_and_run_tools([prompt] + env.list_messages(),
tools=tool_registry.get_all_tool_definitions())```
```

Chat with it - Agent uploaded successfully.
https://app.near.ai/agents/kirikiri.near/multi_agent_game_framework/gen-20250315001907

Example-

Match-3 Puzzle Game

For a match-3 game, you would:

- Create grid generation with diverse game pieces
- Implement swapping mechanics with validation rules
- Develop matching detection and clearing animations
- Build scoring system and level progression logic
- Add special power-ups and combo mechanics

https://app.near.ai/agents/kirikiri.near/agents-builder/latest/run?threadId=thread_fa2a8c18a13e4c44a849a18a

https://app.near.ai/agents/kirikiri.near/multi_agent_game_framework/gen-20250315003908

asset-generator.py

```
from nearai.agents.environment import Environment

def run(env: Environment):

    # Your agent code here

    prompt = {"role": "system", "content": "You are an Asset Generator Agent specializing in creating visual elements for games. Your purpose is to interpret user requests for game assets, generate appropriate images, and upload them to Cloudinary for storage and access."}

    result = env.completion([prompt] + env.list_messages())

    env.add_reply(result)

    env.request_user_input()

run(env)
```

Claude Response -

I'd be happy to help you create a comprehensive plan for a multi-agent framework to build video games according to users' wishes. Based on the code you shared, it seems you've already started developing some of the core agent classes. I'll design a complete system around this foundation.

Product Requirements Document: Game Development Multi-Agent Framework

1. Overview

A collaborative multi-agent system that automates the video game development process according to user specifications. The system will coordinate specialized AI agents that handle different aspects of game development, from conceptualization to implementation, allowing users to create games with minimal technical expertise.

2. Agents Required

Building on your existing `BaseAgent`, `GameDesignerAgent`, and `NarrativeDesignerAgent` classes, here's a complete set of agents needed:

1. **Project Manager Agent** - Coordinates all other agents, maintains project state, handles resource allocation
2. **Game Designer Agent** - Creates game mechanics, rules, systems, and overall gameplay experiences
3. **Narrative Designer Agent** - Develops storylines, dialogue, and narrative progression
4. **Visual Artist Agent** - Generates concept art, character designs, environments, and UI elements
5. **Level Designer Agent** - Creates game levels, maps, and spatial layouts

6. **Sound Designer Agent** - Produces sound effects, music, and audio implementation
7. **Code Generator Agent** - Writes, tests, and optimizes game code across required platforms
8. **QA Tester Agent** - Tests gameplay, finds bugs, and ensures quality standards
9. **User Interface Agent** - Handles communication between the user and the system
10. **Asset Manager Agent** - Organizes, catalogs, and optimizes game assets

3. Task Flow

4. User Journey

1. **Onboarding**
   - User accesses the platform
   - System explains capabilities and limitations
   - User creates new project or loads existing one
2. **Requirements Gathering**
   - User defines game genre, platforms, art style, etc.
   - User provides narrative theme, gameplay elements
   - User sets constraints (time, complexity, resources)
3. **Initial Design Phase**
   - System generates initial design documents
   - User reviews designs (GDD, narrative outline, concept art)
   - User provides feedback and requests revisions
4. **Development Phase**
   - System shows development progress
   - User can view intermediate outputs (level mock-ups, character models)
   - User can test early prototypes and provide feedback

5. **Testing Phase**
    - System shows test results and identified issues
    - User can play test builds and report issues
    - User approves fixes and improvements
6. **Finalization**
    - System packages final game for distribution
    - User receives complete project files
    - User can request post-launch updates or improvements

5. Task Breakdown Between Agents

6. Technical Architecture

7. Tools and Technologies

Core Infrastructure

1. **Language & Framework**
    - Python for agent logic and orchestration
    - FastAPI/Django for web services
    - Redis for message queuing and task distribution
2. **Data Storage**
    - MongoDB for project data, configurations, and agent states
    - S3/Azure Blob for asset storage
    - Git for version control
3. **AI Models**
    - LLMs (GPT-4 or equivalent) for design, narrative, and code generation
    - Stable Diffusion or similar for image generation
    - AudioLM for sound generation

  - Code-specific models for game code generation

Game Development Tools Integration

1. **Game Engines**
   - Unity API integration
   - Unreal Engine API integration
   - Godot integration
   - Custom HTML5/WebGL output
2. **Asset Creation Tools**
   - Blender for 3D modeling (via Python API)
   - Adobe Creative Cloud integration
   - Substance for texturing
3. **Build and Deployment**
   - Docker for containerization
   - CI/CD pipelines (GitHub Actions, Jenkins)
   - Platform-specific build tools (Android SDK, iOS, Steam)

8. Implementation Plan

9. Communication and Messaging System

The multi-agent system requires a robust messaging system to facilitate communication between agents. Building upon your existing `send_message` and `receive_message` methods, here's a more detailed specification:

Message Structure

```
{

  "message_id": "unique_id",

  "timestamp": "ISO-8601 timestamp",

  "from_agent": "sender_agent_type",
  "to_agent": "recipient_agent_type",
  "type": "message_type",
  "priority": 1-5,
  "content": {
    "context": {},
    "data": {},
    "request_id": "related_request_id"
  },
  "metadata": {
    "conversation_id": "thread_id",
    "project_id": "project_identifier",
    "requires_response": true/false
  }
}
```

Message Types

- **task_assignment**: Assign new tasks to agents
- **task_update**: Updates on task progress
- **request_input**: Request information from another agent
- **provide_input**: Provide requested information
- **feedback**: User or agent feedback on artifacts
- **notification**: System events and updates
- **error**: Error reports and issues
- **artifact_update**: Information about new/updated artifacts

Message Queue Implementation

- Redis pub/sub for real-time communication

- Persistent storage of message history in MongoDB
- Priority-based message processing
- Delivery confirmation mechanisms
- Message retry logic for enhanced reliability

10. Risk Analysis and Mitigation

11. Evaluation Metrics

To measure the success of the multi-agent game development system, the following metrics should be tracked:

1. **Game Completion Rate**
    - Percentage of initiated projects that reach completion
    - Time to completion relative to project complexity
2. **User Satisfaction**
    - NPS scores for the overall system
    - Satisfaction ratings for individual game components
    - Feedback analysis for agent performance
3. **Agent Performance**
    - Task completion time
    - Quality scores for artifacts
    - Number of revision cycles needed
    - Resource utilization efficiency
4. **System Performance**
    - End-to-end processing times
    - Error rates and recovery success
    - System uptime and reliability
    - Scalability under varying project loads
5. **Output Quality**
    - Technical quality of generated games

    - Originality and creativity scores
    - Playability and fun factor evaluations
    - Cross-platform compatibility success

12. Future Extensions

1. **Advanced Capabilities**
    - Multiplayer game support
    - Advanced physics and AI behavior systems
    - VR/AR game development
    - Procedural content generation
2. **Ecosystem Expansion**
    - Asset marketplace integration
    - Community feature sharing
    - Template library for common game types
    - Plugin system for custom agent capabilities
3. **User Collaboration**
    - Multi-user collaborative projects
    - Team workflow support
    - Role-based access controls
    - Real-time collaborative editing
4. **Analytics and Optimization**
    - Player behavior prediction
    - Game balancing recommendations
    - Marketing materials generation
    - Monetization strategy optimization

This comprehensive plan provides a roadmap for developing your multi-agent game development framework. The system builds upon your existing agent architecture while expanding it to cover the entire game development process, from concept to completion.

I'll implement the core code structure for your multi-agent game development framework. I'll focus on creating the essential components that will allow the system to function as described in the PRD.

```
import os

import json

import uuid

import datetime

from typing import Dict, List, Any, Optional

from abc import ABC


from base_agent import BaseAgent


class ProjectManagerAgent(BaseAgent):

    """

    Agent responsible for coordinating all other agents and managing the overall project.

    Acts as the central hub for communication between agents.

    """



    def __init__(self):

        super().__init__("project_manager")

        self.project_data = {}

        self.agents = {}
        self.tasks = {}
        self.artifacts = {}
        self.feedback = {}
        self.status = "initializing"
        self.message_history = []

    def initialize_project(self, project_name: str, requirements:
Dict[str, Any]) -> str:
        """Initialize a new game development project with user
requirements."""
        project_id = str(uuid.uuid4())

        self.project_data = {
            "project_id": project_id,
            "project_name": project_name,
            "creation_date": datetime.datetime.now().isoformat(),
            "requirements": requirements,
            "status": "initialized",
            "game_elements":
self._extract_game_elements(requirements),
            "current_phase": "design",
            "milestones": []
        }
        self.status = "active"

        # Create initial milestones
        self._create_project_milestones()

        return project_id

    def _extract_game_elements(self, requirements: Dict[str, Any]) -> Dict[str, Any]:
        """Extract core game elements from user requirements."""
        # In a real implementation, this would use an LLM to parse requirements
        game_elements = {
            "genre": requirements.get("genre", "undefined"),
            "platforms": requirements.get("platforms", ["PC"]),
            "target_audience": requirements.get("target_audience", "General"),
            "art_style": requirements.get("art_style", "Modern"),
            "core_mechanics": requirements.get("core_mechanics", []),
            "narrative_theme": requirements.get("narrative_theme", "Adventure")
        }
        return game_elements


    def _create_project_milestones(self) -> None:
        """Create the standard project milestones based on the game type."""
        milestones = [
            {
                "id": str(uuid.uuid4()),
                "name": "Initial Design",
                "phase": "design",
                "status": "pending",
                "tasks": [],
                "dependencies": [],
                "completion_percentage": 0
            },
            {
                "id": str(uuid.uuid4()),
                "name": "Content Creation",
                "phase": "development",
                "status": "pending",
                "tasks": [],
                "dependencies": ["Initial Design"],
                "completion_percentage": 0
            },
            {
                "id": str(uuid.uuid4()),
                "name": "Prototype",
                "phase": "development",
                "status": "pending",
                "tasks": [],
                "dependencies": ["Content Creation"],
                "completion_percentage": 0
            },
            {
                "id": str(uuid.uuid4()),
                "name": "Testing",
                "phase": "testing",
                "status": "pending",
                "tasks": [],
                "dependencies": ["Prototype"],
                "completion_percentage": 0
            },
            {
                "id": str(uuid.uuid4()),
                "name": "Final Build",
                "phase": "release",
                "status": "pending",
                "tasks": [],
                "dependencies": ["Testing"],
                "completion_percentage": 0
            }
        ]

        self.project_data["milestones"] = milestones

    def register_agent(self, agent: BaseAgent) -> None:
        """Register an agent with the project manager."""
        self.agents[agent.agent_type] = agent
        agent.project_manager = self

    def create_and_assign_initial_tasks(self) -> None:
        """Create and assign initial tasks to registered agents."""
        # Create game design document task
        if "game_designer" in self.agents:
            self._create_and_assign_task(
                agent_type="game_designer",
                task_name="Create game design document",
                description="Create a comprehensive game design document based on project requirements",
                priority=1,
                dependencies=[],
                milestone_id=self.project_data["milestones"][0]["id"]
            )

        # Create narrative design document task
        if "narrative_designer" in self.agents:
            self._create_and_assign_task(
                agent_type="narrative_designer",
                task_name="Create narrative design document",
                description="Create a narrative design document including story, characters, and dialogue",
                priority=1,
                dependencies=[],
                milestone_id=self.project_data["milestones"][0]["id"]
            )
        # Create concept art task
        if "visual_artist" in self.agents:
            self._create_and_assign_task(
                agent_type="visual_artist",
                task_name="Create initial concept art",
                description="Create concept art for main characters,
environments, and key visual elements",
                priority=1,
                dependencies=[],
                milestone_id=self.project_data["milestones"][0]["id"]
            )

    def _create_and_assign_task(self, agent_type: str, task_name:
str, description: str,
                               priority: int, dependencies: List[str],
milestone_id: str) -> str:
        """Create a new task and assign it to the specified agent."""
        task_id = str(uuid.uuid4())

        task = {
            "task_id": task_id,
            "name": task_name,
            "description": description,

            "status": "assigned",

            "assigned_agent": agent_type,

            "priority": priority,

            "creation_date": datetime.datetime.now().isoformat(),

            "due_date": None,  # Would calculate based on project
timeline

            "dependencies": dependencies,

            "milestone_id": milestone_id

        }



        # Store task in tasks dictionary

        self.tasks[task_id] = task



        # Add task to milestone

        for milestone in self.project_data["milestones"]:

            if milestone["id"] == milestone_id:

                milestone["tasks"].append(task_id)



        # Assign task to agent

        if agent_type in self.agents:

            self.agents[agent_type].assign_task(task)
            # Send a message to notify the agent
            self.send_message(
                from_agent="project_manager",
                to_agent=agent_type,
                message_type="task_assignment",
                content={"task_id": task_id, "task_details": task}
            )

        return task_id

    def update_task_status(self, task_id: str, status: str, result:
Optional[Dict[str, Any]] = None) -> None:
        """Update the status of a task and process any results."""
        if task_id in self.tasks:
            self.tasks[task_id]["status"] = status

            if status == "completed" and result:
                self.tasks[task_id]["result"] = result
                self.tasks[task_id]["completion_date"] =
datetime.datetime.now().isoformat()
```

```
                # Update milestone completion percentage
                self._update_milestone_progress(self.tasks[task_id]["milestone_id"])

                # Process any artifacts that were created
                if "artifact_id" in result:
                    self._process_artifact(result["artifact_id"])

                # Check if this completion enables any dependent tasks
                self._check_and_assign_dependent_tasks(task_id)

    def _update_milestone_progress(self, milestone_id: str) -> None:
        """Update the completion percentage of a milestone based on completed tasks."""
        for milestone in self.project_data["milestones"]:
            if milestone["id"] == milestone_id:
                if not milestone["tasks"]:
                    milestone["completion_percentage"] = 0
                    continue

                completed_tasks = sum(1 for task_id in milestone["tasks"]
                                if task_id in self.tasks and self.tasks[task_id]
["status"] == "completed")
            milestone["completion_percentage"] =
int((completed_tasks / len(milestone["tasks"])) * 100)


            if milestone["completion_percentage"] == 100:
                milestone["status"] = "completed"
                # Check if next milestone can be started
                self._advance_project_phase()


    def _advance_project_phase(self) -> None:
        """Advance the project to the next phase if current phase is
complete."""
        phases = ["design", "development", "testing", "release"]
        current_phase = self.project_data["current_phase"]


        # Check if all milestones in the current phase are complete
        current_phase_complete = all(
            milestone["status"] == "completed"
            for milestone in self.project_data["milestones"]
            if milestone["phase"] == current_phase
        )
        if current_phase_complete and current_phase in phases:

            next_phase_index = phases.index(current_phase) + 1

            if next_phase_index < len(phases):

                self.project_data["current_phase"] =
phases[next_phase_index]

                # Create tasks for the next phase

                self._create_tasks_for_current_phase()


    def _create_tasks_for_current_phase(self) -> None:

        """Create tasks appropriate for the current project phase."""

        current_phase = self.project_data["current_phase"]


        if current_phase == "development":

            # Create development tasks

            if "level_designer" in self.agents:

                milestone_id = next((m["id"] for m in
self.project_data["milestones"]

                                    if m["name"] == "Content Creation"), None)

                if milestone_id:

                    self._create_and_assign_task(

                        agent_type="level_designer",

                        task_name="Create game levels",
                        description="Design and implement game levels
based on the GDD",
                        priority=1,
                        dependencies=[],  # Would reference design doc
task IDs
                        milestone_id=milestone_id
                    )


            if "code_generator" in self.agents:
                milestone_id = next((m["id"] for m in
self.project_data["milestones"]
                                    if m["name"] == "Prototype"), None)
                if milestone_id:
                    self._create_and_assign_task(
                        agent_type="code_generator",
                        task_name="Implement core mechanics",
                        description="Implement the core game mechanics
as defined in the GDD",
                        priority=1,
                        dependencies=[],  # Would reference design doc
task IDs
                        milestone_id=milestone_id
                    )
            # Similar tasks for sound_designer, etc.


        elif current_phase == "testing":
            # Create testing tasks
            if "qa_tester" in self.agents:
                milestone_id = next((m["id"] for m in
self.project_data["milestones"]
                                    if m["name"] == "Testing"), None)
                if milestone_id:
                    self._create_and_assign_task(
                        agent_type="qa_tester",
                        task_name="Perform gameplay testing",
                        description="Test the game for bugs, balance issues,
and player experience",
                        priority=1,
                        dependencies=[],  # Would reference prototype task
IDs
                        milestone_id=milestone_id
                    )


    def _check_and_assign_dependent_tasks(self,
completed_task_id: str) -> None:
        """Check if any tasks are now unblocked by the completion of
this task."""
```

```
        # In a real implementation, would check task dependencies
and assign new tasks
        pass

    def register_artifact(self, artifact_type: str, artifact_data: Any,
metadata: Dict[str, Any]) -> str:
        """Register a new artifact with the system."""
        artifact_id = str(uuid.uuid4())

        artifact = {
            "artifact_id": artifact_id,
            "artifact_type": artifact_type,
            "creation_date": datetime.datetime.now().isoformat(),
            "last_modified": datetime.datetime.now().isoformat(),
            "metadata": metadata,
            "version": 1,
            "data": artifact_data
        }

        self.artifacts[artifact_id] = artifact
        return artifact_id
```

```
def _process_artifact(self, artifact_id: str) -> None:
    """Process a newly created or updated artifact."""
    if artifact_id not in self.artifacts:
        return

    artifact = self.artifacts[artifact_id]
    artifact_type = artifact["artifact_type"]

    # Handle artifact based on type
    if artifact_type == "design_document":
        # Notify narrative designer about the design document
        if "narrative_designer" in self.agents:
            self.send_message(
                from_agent="project_manager",
                to_agent="narrative_designer",
                message_type="artifact_update",
                content={
                    "artifact_id": artifact_id,
                    "artifact_type": artifact_type,
                    "action": "review"
                }
            )

    elif artifact_type == "narrative_document":
        # Notify level designer about the narrative document
        if "level_designer" in self.agents:
            self.send_message(
                from_agent="project_manager",
                to_agent="level_designer",
                message_type="artifact_update",
                content={
                    "artifact_id": artifact_id,
                    "artifact_type": artifact_type,
                    "action": "review"
                }
            )

def handle_message(self, message: Dict[str, Any]) -> None:
    """Handle incoming messages based on message type."""
    message_type = message.get("type", "")
    content = message.get("content", {})
    # Log the message for history
    self.message_history.append(message)

    if message_type == "status_update":
        # An agent is providing a status update
        if "task_id" in content:
            task_id = content["task_id"]
            status = content.get("status", "")
            result = content.get("result", {})

            self.update_task_status(task_id, status, result)

    elif message_type == "request_resource":
        # An agent is requesting a resource (e.g., an artifact)
        resource_type = content.get("resource_type", "")
        resource_id = content.get("resource_id", "")

        if resource_type == "artifact" and resource_id in self.artifacts:
            # Send the artifact to the requesting agent
            self.send_message(
                from_agent="project_manager",
        to_agent=message["from_agent"],
        message_type="resource_response",
        content={
            "resource_type": resource_type,
            "resource_id": resource_id,
            "data": self.artifacts[resource_id],
            "request_id": content.get("request_id", "")
        }
    )

elif message_type == "user_feedback":
# User has provided feedback
feedback_id = str(uuid.uuid4())

self.feedback[feedback_id] = {
    "feedback_id": feedback_id,
    "timestamp": datetime.datetime.now().isoformat(),
    "content": content
}

# Distribute feedback to relevant agents
```

```
            self._distribute_feedback(feedback_id)

    def _distribute_feedback(self, feedback_id: str) -> None:
        """Distribute user feedback to the appropriate agents."""
        if feedback_id not in self.feedback:
            return

        feedback = self.feedback[feedback_id]
        content = feedback["content"]

        # Check for design feedback
        if "design" in content:
            if "game_designer" in self.agents:
                self._create_and_assign_task(
                    agent_type="game_designer",
                    task_name="Process design feedback",
                    description="Review and incorporate user feedback on game design",
                    priority=2,
                    dependencies=[],
                    milestone_id=self.project_data["milestones"][0]["id"]
                )
        # Check for narrative feedback
        if "narrative" in content:
            if "narrative_designer" in self.agents:
                self._create_and_assign_task(
                    agent_type="narrative_designer",
                    task_name="Process narrative feedback",
                    description="Review and incorporate user feedback on
narrative elements",
                    priority=2,
                    dependencies=[],
                    milestone_id=self.project_data["milestones"][0]["id"]
                )

        # Similar handling for other types of feedback


    def send_message(self, from_agent: str, to_agent: str,
message_type: str, content: Dict) -> str:
        """Send a message to another agent."""
        message_id = str(uuid.uuid4())


        message = {
            "message_id": message_id,
            "timestamp": datetime.datetime.now().isoformat(),
            "from_agent": from_agent,
            "to_agent": to_agent,
            "type": message_type,
            "content": content
        }

        # Log the message
        self.message_history.append(message)

        # Deliver to recipient agent
        if to_agent in self.agents:
            self.agents[to_agent].receive_message(message)

        return message_id

    def get_project_status(self) -> Dict[str, Any]:
        """Get the current status of the project."""
        # Calculate overall project completion
        total_tasks = len(self.tasks)
        completed_tasks = sum(1 for task in self.tasks.values() if task["status"] == "completed")

        completion_percentage = 0
        if total_tasks > 0:
            completion_percentage = int((completed_tasks / total_tasks) * 100)

        return {
            "project_id": self.project_data.get("project_id", ""),
            "project_name": self.project_data.get("project_name", ""),
            "current_phase": self.project_data.get("current_phase", ""),
            "completion_percentage": completion_percentage,
            "milestones": [
                {
                    "name": milestone["name"],
                    "status": milestone["status"],
                    "completion_percentage": milestone["completion_percentage"]
                }
                for milestone in self.project_data.get("milestones", [])
            ],
            "recent_activities": self._get_recent_activities(5)
        }

    def _get_recent_activities(self, count: int) -> List[Dict[str, Any]]:
        """Get the most recent project activities."""
        # In a real implementation, would derive from task updates and messages
        return [
            {
                "timestamp": message["timestamp"],
                "type": message["type"],
                "from_agent": message["from_agent"],
                "to_agent": message["to_agent"],
                "summary": self._generate_activity_summary(message)
            }
            for message in sorted(self.message_history, key=lambda m: m["timestamp"], reverse=True)[:count]
        ]

    def _generate_activity_summary(self, message: Dict[str, Any]) -> str:
        """Generate a human-readable summary of an activity."""
        message_type = message.get("type", "")
        if message_type == "task_assignment":
            task_name = message.get("content", {}).get("task_details", {}).get("name", "Unknown task")
            return f"Task '{task_name}' assigned to {message['to_agent']}"

        elif message_type == "status_update":
            task_id = message.get("content", {}).get("task_id", "")
            status = message.get("content", {}).get("status", "")

            if task_id in self.tasks:
                task_name = self.tasks[task_id]["name"]
                return f"Task '{task_name}' status updated to '{status}'"

            return f"Task status updated to '{status}'"

        # Handle other message types similarly

        return f"Message of type '{message_type}' from {message['from_agent']} to {message['to_agent']}"

    def get_artifact(self, artifact_id: str) -> Optional[Dict[str, Any]]:
        """Retrieve an artifact by ID."""
        return self.artifacts.get(artifact_id)

    def save_project_state(self, filepath: str) -> None:
        """Save the current project state to a file."""
        project_state = {
            "project_data": self.project_data,
            "tasks": self.tasks,
            "artifacts": self.artifacts,
            "feedback": self.feedback,
            "message_history": self.message_history
        }

        with open(filepath, 'w') as f:
            json.dump(project_state, f, indent=2)

    def load_project_state(self, filepath: str) -> None:
        """Load project state from a file."""
        if not os.path.exists(filepath):
            return

        with open(filepath, 'r') as f:
            project_state = json.load(f)

        self.project_data = project_state.get("project_data", {})
        self.tasks = project_state.get("tasks", {})
        self.artifacts = project_state.get("artifacts", {})
        self.feedback = project_state.get("feedback", {})
        self.message_history = project_state.get("message_history", [])
        self.status = "active"

import uuid
import datetime
from typing import Dict, List, Any, Optional
from abc import ABC, abstractmethod

class BaseAgent(ABC):
    """Base class for all specialized agents in the game development framework."""

    def __init__(self, agent_type: str, project_manager=None):
        self.agent_id = str(uuid.uuid4())
        self.agent_type = agent_type
        self.project_manager = project_manager
        self.message_queue = []

        self.current_tasks = []

        self.completed_tasks = []

        self.knowledge_base = {}

        self.artifacts = {}


    def receive_message(self, message: Dict[str, Any]) -> None:

        """Receive a message from another agent."""

        self.message_queue.append(message)

        self.process_messages()


    def send_message(self, to_agent: str, message_type: str, content: Dict) -> str:

        """Send a message to another agent through the project manager."""

        if self.project_manager:

            return self.project_manager.send_message(

                from_agent=self.agent_type,

                to_agent=to_agent,

                message_type=message_type,

                content=content

            )
        return ""


    def process_messages(self) -> None:
        """Process all queued messages."""
        for message in self.message_queue[:]:
            self.handle_message(message)
            self.message_queue.remove(message)


    @abstractmethod
    def handle_message(self, message: Dict[str, Any]) -> None:
        """Handle a specific message based on its type."""
        pass


    def assign_task(self, task: Dict[str, Any]) -> None:
        """Assign a new task to this agent."""
        self.current_tasks.append(task)


    def complete_task(self, task_id: str, result: Dict[str, Any]) -> None:
        """Mark a task as completed and submit results."""
        # Find the task in current tasks
        task = next((t for t in self.current_tasks if t["task_id"] ==
task_id), None)
```

```
        if task:
            # Remove from current tasks
            self.current_tasks = [t for t in self.current_tasks if
t["task_id"] != task_id]

            # Add to completed tasks
            task["result"] = result
            task["completion_date"] =
datetime.datetime.now().isoformat()
            self.completed_tasks.append(task)

            # Update project manager
            if self.project_manager:
                self.project_manager.update_task_status(task_id,
"completed", result)

                # Send a status update message
                self.send_message(
                    to_agent="project_manager",
                    message_type="status_update",
                    content={
                        "task_id": task_id,
                        "status": "completed",
                        "result": result
                    }
                )

    def register_artifact(self, artifact_type: str, artifact_data: Any, metadata: Dict) -> str:
        """Register a new artifact with the project manager."""
        if self.project_manager:
            metadata["creator_agent"] = self.agent_type
            artifact_id = self.project_manager.register_artifact(artifact_type, artifact_data, metadata)
            self.artifacts[artifact_id] = artifact_data
            return artifact_id
        return ""

    @abstractmethod
    def process_task(self, task: Dict[str, Any]) -> Dict[str, Any]:
        """Process a specific task and return the result."""
        pass

    def run_tasks(self) -> None:
        """Process all current tasks."""
        for task in self.current_tasks[:]:
            result = self.process_task(task)
            self.complete_task(task["task_id"], result)


    def request_artifact(self, artifact_id: str) -> Optional[Dict[str, Any]]:
        """Request an artifact from the project manager."""
        if not self.project_manager:
            return None


        request_id = str(uuid.uuid4())


        # Send a request for the artifact
        self.send_message(
            to_agent="project_manager",
            message_type="request_resource",
            content={
                "resource_type": "artifact",
                "resource_id": artifact_id,
                "request_id": request_id
            }
        )

        # In a real async implementation, would wait for response
        # For now, directly request from project manager
        return self.project_manager.get_artifact(artifact_id)

    def request_input_from_agent(self, to_agent: str, topic: str,
context: Dict[str, Any]) -> Optional[Dict[str, Any]]:
        """Request input from another agent on a specific topic."""
        if not self.project_manager:
            return None

        request_id = str(uuid.uuid4())

        # Send request for input
        self.send_message(
            to_agent=to_agent,
            message_type=f"request_{to_agent.split('_')[0]}_input",
            content={
                "topic": topic,
                "context": context,
                "request_id": request_id
            }
        )

        # In a real async implementation, would wait for response
        # For this prototype, would need to be handled in the specific agent implementation
        return None

    def load_knowledge_base(self, knowledge_path: Optional[str] = None) -> None:
        """Load knowledge base for this agent type."""
        # In a real implementation, would load from a file or database
        # For now, each agent subclass should override this with domain-specific knowledge
        pass
```

Visual Artist.py

```
import uuid
import datetime
import json
from typing import Dict, List, Any, Optional
import base64
import io
import os

from base_agent import BaseAgent

class VisualArtistAgent(BaseAgent):
    """Agent responsible for creating visual assets for the game."""


    def __init__(self, project_manager=None, use_real_ai_models=False, model_config=None):
        super().__init__("visual_artist", project_manager)
        self.art_styles = {}
        self.color_palettes = {}
        self.concept_art = {}
        self.use_real_ai_models = use_real_ai_models
        self.model_config = model_config or {}
        self.load_knowledge_base()


    def load_knowledge_base(self) -> None:
        """Load art knowledge base."""
        # This would load from an actual knowledge base file/database
        self.knowledge_base = {
            "art_styles": {
```

```
        "pixel_art": {
            "description": "Pixelated graphics often used in retro-style games",
            "characteristic_elements": ["limited color palette", "visible pixels", "simplified shapes"],
            "suitable_genres": ["platformer", "roguelike", "retro"]
        },
        "cartoon": {
            "description": "Animated, exaggerated style with bold outlines",
            "characteristic_elements": ["bold outlines", "exaggerated proportions", "bright colors"],
            "suitable_genres": ["casual", "adventure", "family"]
        },
        "realistic": {
            "description": "Photorealistic style attempting to simulate real-world visuals",
            "characteristic_elements": ["detailed textures", "realistic lighting", "accurate proportions"],
            "suitable_genres": ["simulation", "sports", "first-person shooter"]
        },
        "low_poly": {
            "description": "3D style using a small number of polygons, giving a geometric look",
            "characteristic_elements": ["visible polygons", "flat colors or simple textures", "geometric shapes"],

            "suitable_genres": ["indie", "puzzle", "action"]

        }

    },

    "color_palettes": {

        "fantasy": ["#1a2980", "#26d0ce", "#9733ee", "#2ecc71", "#f1c40f"],

        "horror": ["#1f1f1f", "#761f1f", "#3b1f76", "#541f1f", "#0a0a0a"],

        "sci_fi": ["#0a192f", "#3498db", "#2ecc71", "#f39c12", "#8e44ad"],

        "retro": ["#f9d5bb", "#f66767", "#d35656", "#3cbbb1", "#2d3999"]

    },

    "character_design_principles": {

        "silhouette": "Ensure character has recognizable shape even in outline form",

        "color_identity": "Use consistent, meaningful colors for character recognition",

        "personality": "Visual elements should reflect character personality",

        "proportion": "Proportions can be manipulated to emphasize character traits",

        "detail_level": "Balance detail with readability for target platform"
        }
    }

def handle_message(self, message: Dict[str, Any]) -> None:
    """Handle incoming messages."""
    message_type = message.get("type", "")
    content = message.get("content", {})

    if message_type == "request_visual_input":
        # Another agent is requesting visual design input
        visual_input =
self.generate_visual_input(content.get("topic"),
content.get("context"))
        self.send_message(
            to_agent=message["from_agent"],
            message_type="visual_input_response",
            content={"visual_input": visual_input, "request_id":
content.get("request_id")}
        )
    elif message_type == "feedback_visual":
        # Process feedback on visual designs
        self.incorporate_feedback(content.get("feedback"),
content.get("artifact_id"))
```

```
        elif message_type == "artifact_update":
            # Another agent has created/updated an artifact we should know about
            self.review_artifact(content.get("artifact_id"), content.get("artifact_type"))

    def generate_visual_input(self, topic: str, context: Dict) -> Dict[str, Any]:
        """Generate visual design input on a specific topic."""
        # This would use an AI image generation model for visual suggestions
        # For this prototype, we'll return placeholder data
        if topic == "art_style":
            genre = context.get("genre", "")
            suitable_styles = [
                style for style, data in self.knowledge_base["art_styles"].items()
                if genre in data["suitable_genres"]
            ]

            return {
                "suggested_styles": suitable_styles,
                "rationale": f"These styles are well-suited for the {genre} genre.",
                "examples": [f"{style}_example.jpg" for style in suitable_styles]
            }
        elif topic == "color_palette":
            theme = context.get("theme", "")
            if theme in self.knowledge_base["color_palettes"]:
                return {
                    "suggested_palette": self.knowledge_base["color_palettes"][theme],
                    "rationale": f"This palette conveys the {theme} mood effectively."
                }

        return {"message": "Visual input not available for this topic"}

    def incorporate_feedback(self, feedback: str, artifact_id: str) -> None:
        """Incorporate feedback into visual designs."""
        # This would use an LLM to interpret feedback and modify designs
        # For now, just log the feedback
        print(f"Visual Artist incorporating feedback for artifact {artifact_id}: {feedback}")
        # In a real implementation, would retrieve the artifact, modify it, and update it
        if artifact_id in self.artifacts:
            # Make modifications based on feedback
            # Re-register the updated artifact
            self.register_artifact(
                artifact_type="visual_design_update",
                artifact_data=self.artifacts[artifact_id],  # In reality, would be modified
                metadata={
                    "original_artifact_id": artifact_id,
                    "feedback_applied": feedback,
                    "description": "Updated based on feedback"
                }
            )

    def review_artifact(self, artifact_id: str, artifact_type: str) -> None:
        """Review an artifact created by another agent."""
        if not self.project_manager:
            return

        artifact = self.request_artifact(artifact_id)
        if not artifact:
            return

        # Take action based on artifact type
        if artifact_type == "design_document":
            # Extract visual design requirements from GDD
            design_data = artifact.get("data", {})
            art_style = design_data.get("art_style", "")

            # Use design document to inform visual development
            self.concept_art["game_style"] = art_style

    def process_task(self, task: Dict[str, Any]) -> Dict[str, Any]:
        """Process a visual art task."""
        task_name = task.get("name", "")

        if task_name == "Create initial concept art":
            return self.create_concept_art(task)
        elif task_name == "Create character designs":
            return self.create_character_designs(task)
        elif task_name == "Create environment art":
            return self.create_environment_art(task)
        elif task_name == "Create UI design":
            return self.create_ui_design(task)

        return {"status": "failed", "message": "Unknown task type"}

    def create_concept_art(self, task: Dict[str, Any]) -> Dict[str, Any]:
        """Create initial concept art for the game."""
        # In a real implementation, this would use an image generation model
        # For prototype purposes, create placeholder data

        # Get game elements from project manager
        game_elements = {}
        if self.project_manager and hasattr(self.project_manager, "project_data"):
            game_elements = self.project_manager.project_data.get("game_elements", {})

        genre = game_elements.get("genre", "undefined")
        art_style = game_elements.get("art_style", "cartoon")

        # Determine appropriate style and palette
        suitable_styles = []
        for style, data in self.knowledge_base["art_styles"].items():
            if genre in data["suitable_genres"]:
                suitable_styles.append(style)

        selected_style = art_style if art_style in suitable_styles and suitable_styles else (
            suitable_styles[0] if suitable_styles else "cartoon"
        )

        theme = game_elements.get("narrative_theme", "fantasy")
        palette = self.knowledge_base["color_palettes"].get(
            theme, self.knowledge_base["color_palettes"]["fantasy"]
        )

        # Generate concept art descriptions (in a real system, would generate actual images)
        concept_art_collection = {
            "main_character": self._generate_concept_description("character", selected_style, genre, palette),
            "environment": self._generate_concept_description("environment", selected_style, genre, palette),
            "key_item": self._generate_concept_description("item", selected_style, genre, palette),
            "ui_theme": self._generate_concept_description("ui", selected_style, genre, palette)
        }

        # Register the concept art as an artifact
        artifact_id = self.register_artifact(
            artifact_type="concept_art",
            artifact_data=concept_art_collection,
            metadata={
                "description": "Initial Concept Art Collection",
                "art_style": selected_style,
                "color_palette": palette
            }
        )

        return {
            "status": "completed",
            "artifact_id": artifact_id,
            "concept_art": concept_art_collection
        }
```

```
    def _generate_concept_description(self, concept_type: str, style: str, genre: str, palette: List[str]) -> Dict[str, Any]:
        """Generate a description for concept art (placeholder for image generation)."""
        style_data = self.knowledge_base["art_styles"].get(style, {})
        characteristics = style_data.get("characteristic_elements", [])

        # In a real implementation, this description would be used to generate an image
        # For the prototype, it serves as a placeholder for the image
        description = f"{style} style {concept_type} for a {genre} game, featuring "

        if concept_type == "character":
            description += "a protagonist with "
            if genre == "platformer":
                description += "agile appearance and determined expression. "
            elif genre == "rpg":
                description += "heroic presence and customizable features. "
            else:
                description += "distinctive silhouette and personality. "
        elif concept_type == "environment":
            description += "a key game location with "
            if genre == "platformer":
                description += "platforms, obstacles, and collectibles. "
            elif genre == "rpg":
                description += "atmospheric lighting and interactive elements. "
            else:
                description += "unique landmarks and navigation paths. "
        elif concept_type == "item":
            description += "an important game item that "
            if genre == "platformer":
                description += "enhances the character's movement abilities. "
            elif genre == "rpg":
                description += "provides stat boosts or special powers. "
            else:
                description += "serves a key gameplay function. "
        elif concept_type == "ui":
            description += "a user interface design that "
            if genre == "platformer":
                description += "shows score, lives, and collectibles clearly. "
            elif genre == "rpg":
                description += "displays character stats, inventory, and quest info. "
            else:
                description += "provides essential player feedback seamlessly. "

        # Add style characteristics
        description += f"Art features include {', '.join(characteristics)}. "

        # Add color information
        description += f"Primary colors: {', '.join(palette[:3])}."

        # If using real image generation models
        image_data = None
        if self.use_real_ai_models:
            image_data = self._generate_image(description)

        return {
            "description": description,
            "style": style,
            "color_palette": palette,
            "image_prompt": description,
            "image_data": image_data  # Would be base64 encoded in a real implementation
        }

    def _generate_image(self, prompt: str) -> Optional[str]:
        """Generate an image using AI image generation (placeholder)."""
        # In a real implementation, would call an image generation API
        # For now, return None or a placeholder
        return None

    def create_character_designs(self, task: Dict[str, Any]) -> Dict[str, Any]:
        """Create detailed character designs based on concept art."""
        # Similar to create_concept_art but focused on characters
        # Would reference concept art artifact for consistency

        # For prototype, return placeholder data
        return {
            "status": "completed",
            "message": "Character designs created",
            "character_designs": {
                "protagonist": {
                    "description": "Main character design document",
                    "image": None  # Would be image data in real implementation
                },
                "antagonist": {
                    "description": "Antagonist character design document",
                    "image": None
                }
            }
        }

    def create_environment_art(self, task: Dict[str, Any]) -> Dict[str, Any]:
        """Create environment art for the game levels."""
        # Would create environment art based on level designs and concept art

        # For prototype, return placeholder
        return {
            "status": "completed",
            "message": "Environment art created",
            "environment_art": {
```

```
            "level_1": {
                "description": "First level environment design",
                "image": None
            },
            "level_2": {
                "description": "Second level environment design",
                "image": None
            }
        }
    }

def create_ui_design(self, task: Dict[str, Any]) -> Dict[str, Any]:
    """Create user interface designs for the game."""
    # Would create UI mockups based on game requirements

    # For prototype, return placeholder
    return {
        "status": "completed",
        "message": "UI design created",
        "ui_designs": {
            "main_menu": {
                "description": "Main menu interface design",
                "image": None
            },
            "hud": {
                "description": "In-game HUD design",
                "image": None
            },
            "inventory": {
                "description": "Inventory screen design",
                "image": None
            }
        }
    }
```

CodeGenerator.py

```
import os
import uuid
import datetime
from typing import Dict, List, Any, Optional
import json

from base_agent import BaseAgent
```

```
class CodeGeneratorAgent(BaseAgent):
    """Agent responsible for generating game code and technical implementations."""


    def __init__(self, project_manager=None):
        super().__init__("code_generator", project_manager)
        self.code_templates = {}
        self.supported_engines = ["unity", "unreal", "godot", "html5"]
        self.supported_languages = ["c#", "python", "c++", "javascript"]
        self.current_engine = None
        self.current_language = None
        self.code_modules = {}
        self.load_knowledge_base()


    def load_knowledge_base(self) -> None:
        """Load code generation knowledge base."""
        # This would load from an actual knowledge base file/database
        self.knowledge_base = {
            "design_patterns": {
                "entity_component_system": {
                    "description": "Composition over inheritance pattern for game objects",
                    "use_cases": ["complex entity behaviors", "modular game objects"],
                    "code_examples": {}  # Would contain example implementations
                },
                "state_machine": {
                    "description": "Pattern for managing object states and transitions",
                    "use_cases": ["character behavior", "game flow", "AI"],
                    "code_examples": {}
                },
                "singleton": {
                    "description": "Global access point for a single instance",
                    "use_cases": ["game managers", "resource managers"],
                    "code_examples": {}
                },
                "observer": {
                    "description": "One-to-many dependency for event notifications",
                    "use_cases": ["UI updates", "achievement systems", "event handling"],
                    "code_examples": {}
                }
```

```
            },
            "game_mechanics": {
                "movement": {
                    "2d_platformer": {
                        "parameters": ["speed", "jump height", "gravity"],
                        "code_examples": {}
                    },
                    "top_down": {
                        "parameters": ["speed", "rotation", "acceleration"],
                        "code_examples": {}
                    }
                },
                "combat": {
                    "melee": {
                        "parameters": ["damage", "range", "cooldown"],
                        "code_examples": {}
                    },
                    "ranged": {
                        "parameters": ["damage", "projectile speed", "rate of fire"],
                        "code_examples": {}
                    }
```

```
        }
    },
    "engine_specifics": {
        "unity": {
            "project_structure": {},
            "common_apis": {},
            "best_practices": []
        },
        "unreal": {
            "project_structure": {},
            "common_apis": {},
            "best_practices": []
        },
        "godot": {
            "project_structure": {},
            "common_apis": {},
            "best_practices": []
        }
    }
}
```

```
        # Load code templates (in a real system, these would be
detailed templates)

        self.code_templates = {

            "unity": {

                "player_controller": "// Unity C# Player Controller
Template\nusing UnityEngine;\n\npublic class PlayerController :
MonoBehaviour {\n    // Movement parameters\n    public float
moveSpeed = 5f;\n    \n    void Update() {\n        // Movement logic
here\n    }\n}",

                "game_manager": "// Unity C# Game Manager
Template\nusing UnityEngine;\n\npublic class GameManager :
MonoBehaviour {\n    // Singleton instance\n    public static
GameManager Instance { get; private set; }\n    \n    void Awake()
{\n        // Singleton pattern implementation\n        if (Instance ==
null) {\n            Instance = this;\n
DontDestroyOnLoad(gameObject);\n        } else {\n
Destroy(gameObject);\n        }\n    }\n}"

            },

            "godot": {

                "player_controller": "# Godot GDScript Player Controller
Template\nextends KinematicBody2D\n\n# Movement
parameters\nexport var speed = 200\n\nfunc
_physics_process(delta):\n    # Movement logic here\n    pass",

                "game_manager": "# Godot GDScript Game Manager
Template\nextends Node\n\n# Singleton pattern in Godot\n\nfunc
_ready():\n    # Initialize game systems\n    pass"

            },

            "html5": {

                "game_loop": "// HTML5 JavaScript Game Loop
Template\nconst canvas =
document.getElementById('gameCanvas');\nconst ctx = canvas.getContext('2d');\n\nlet lastTime = 0;\n\nfunction gameLoop(timestamp) {\n    const deltaTime = timestamp - lastTime;\n    lastTime = timestamp;\n    \n    update(deltaTime);\n    render();\n    \n    requestAnimationFrame(gameLoop);\n}\n\nfunction update(deltaTime) {\n    // Update game state\n}\n\nfunction render() {\n    // Render game objects\n}\n\nrequestAnimationFrame(gameLoop);"
            }
        }


    def handle_message(self, message: Dict[str, Any]) -> None:
        """Handle incoming messages."""
        message_type = message.get("type", "")
        content = message.get("content", {})

        if message_type == "request_code_input":
            # Another agent is requesting code input
            code_input = self.generate_code_input(content.get("topic"), content.get("context"))
            self.send_message(
                to_agent=message["from_agent"],
                message_type="code_input_response",
                content={"code_input": code_input, "request_id": content.get("request_id")}
            )
        elif message_type == "feedback_code":
            # Process feedback on code implementation
            self.incorporate_feedback(content.get("feedback"), content.get("artifact_id"))
        elif message_type == "artifact_update":
            # Another agent has created/updated an artifact we should know about
            self.review_artifact(content.get("artifact_id"), content.get("artifact_type"))


    def generate_code_input(self, topic: str, context: Dict) -> Dict[str, Any]:
        """Generate code input on a specific topic."""
        # This would use an LLM or other AI to generate code suggestions
        # For this prototype, we'll return placeholder data
        if topic == "design_pattern":
            pattern_name = context.get("pattern", "")
            if pattern_name in self.knowledge_base["design_patterns"]:
                pattern = self.knowledge_base["design_patterns"][pattern_name]
                return {
                    "pattern_name": pattern_name,
                    "description": pattern["description"],
                    "suitable_for": pattern["use_cases"],
                    "code_example": "// Example code would go here"  # In real implementation, would have actual code
                }
        elif topic == "game_mechanic":
            mechanic = context.get("mechanic", "")
            mechanic_type = context.get("type", "")

            if (mechanic in self.knowledge_base["game_mechanics"] and
                mechanic_type in self.knowledge_base["game_mechanics"][mechanic]):
                mechanic_data = self.knowledge_base["game_mechanics"][mechanic][mechanic_type]
                return {
                    "mechanic": f"{mechanic_type} {mechanic}",
                    "parameters": mechanic_data["parameters"],
                    "code_example": "// Example code would go here"  # In real implementation, would have actual code
                }

        return {"message": "Code input not available for this topic"}
```

```
    def incorporate_feedback(self, feedback: str, artifact_id: str) ->
None:
        """Incorporate feedback into code implementation."""
        # This would use an LLM to interpret feedback and modify
code
        # For now, just log the feedback
        print(f"Code Generator incorporating feedback for artifact
{artifact_id}: {feedback}")

        # In a real implementation, would retrieve the code artifact,
modify it based on feedback, and update it
        if artifact_id in self.artifacts:
            # Make modifications based on feedback
            # Re-register the updated artifact
            self.register_artifact(
                artifact_type="code_update",
                artifact_data=self.artifacts[artifact_id],  # In reality, would
be modified
                metadata={
                    "original_artifact_id": artifact_id,
                    "feedback_applied": feedback,
                    "description": "Updated based on feedback"
                }
            )
```

```
def review_artifact(self, artifact_id: str, artifact_type: str) -> None:
    """Review an artifact created by another agent."""
    if not self.project_manager:
        return

    artifact = self.request_artifact(artifact_id)
    if not artifact:
        return

    # Take action based on artifact type
    if artifact_type == "design_document":
        # Extract implementation requirements from GDD
        design_data = artifact.get("data", {})
        platforms = design_data.get("target_platforms", [])

        # Determine appropriate engine and language
        self.determine_engine_and_language(platforms)

        # Store key gameplay mechanics for implementation
        if "core_mechanics" in design_data:
                self.code_modules["core_mechanics"] = design_data["core_mechanics"]

        elif artifact_type == "level_design":
            # Store level data for implementation
            level_data = artifact.get("data", {})
            self.code_modules["levels"] = level_data

    def determine_engine_and_language(self, platforms: List[str]) -> None:
        """Determine the most appropriate game engine and language based on platforms."""
        # Simple logic for engine selection based on platforms
        if "mobile" in platforms and "PC" in platforms:
            self.current_engine = "unity"
            self.current_language = "c#"
        elif "PC" in platforms and "console" in platforms:
            self.current_engine = "unreal"
            self.current_language = "c++"
        elif "web" in platforms:
            self.current_engine = "html5"
            self.current_language = "javascript"
        else:
```

```
        # Default to Unity as a general-purpose engine
        self.current_engine = "unity"
        self.current_language = "c#"

    def process_task(self, task: Dict[str, Any]) -> Dict[str, Any]:
        """Process a code generation task."""
        task_name = task.get("name", "")

        if task_name == "Initialize project structure":
            return self.initialize_project_structure(task)
        elif task_name == "Implement core mechanics":
            return self.implement_core_mechanics(task)
        elif task_name == "Implement game systems":
            return self.implement_game_systems(task)
        elif task_name == "Create build configuration":
            return self.create_build_configuration(task)

        return {"status": "failed", "message": "Unknown task type"}

    def initialize_project_structure(self, task: Dict[str, Any]) -> Dict[str,
Any]:
        """Initialize the code project structure for the selected engine."""
        # In a real implementation, this would create actual project files
        # For prototype purposes, create a project structure description


        if not self.current_engine:
            # Get game elements and determine engine if not already set
            game_elements = {}
            if self.project_manager and hasattr(self.project_manager, "project_data"):
                game_elements = self.project_manager.project_data.get("game_elements", {})


            platforms = game_elements.get("platforms", ["PC"])
            self.determine_engine_and_language(platforms)


        # Generate project structure based on engine
        project_structure = self._generate_project_structure()


        # Register the project structure as an artifact
        artifact_id = self.register_artifact(
            artifact_type="project_structure",
            artifact_data=project_structure,
            metadata={
                "description": f"Initial {self.current_engine.capitalize()} Project Structure",
                "engine": self.current_engine,
                "language": self.current_language
            }
        )

        return {
            "status": "completed",
            "artifact_id": artifact_id,
            "project_structure": project_structure
        }

    def _generate_project_structure(self) -> Dict[str, Any]:
        """Generate project structure for the current engine."""
        if self.current_engine == "unity":
            return {
                "engine": "Unity",
                "language": "C#",
                "folders": {
```

```
        "Assets": {
            "Scripts": {
                "Core": {},
                "Player": {},
                "UI": {},
                "Managers": {},
                "Gameplay": {}
            },
            "Prefabs": {},
            "Scenes": {},
            "Materials": {},
            "Textures": {},
            "Audio": {}
        },
        "ProjectSettings": {},
        "Packages": {}
    },
    "configuration_files": [
        "ProjectSettings.asset",
        "EditorSettings.asset",
        "InputManager.asset"
        ]
    }
elif self.current_engine == "unreal":
    return {
        "engine": "Unreal Engine",
        "language": "C++/Blueprint",
        "folders": {
            "Content": {
                "Blueprints": {},
                "Maps": {},
                "Materials": {},
                "Meshes": {},
                "Textures": {},
                "UI": {},
                "Audio": {}
            },
            "Source": {
                "GameName": {
                    "Public": {},
                    "Private": {}
                }
            },
            "Config": {}
        },
        "configuration_files": [
            "DefaultEngine.ini",
            "DefaultGame.ini",
            "DefaultInput.ini"
        ]
    }
elif self.current_engine == "godot":
    return {
        "engine": "Godot",
        "language": "GDScript",
        "folders": {
            "scenes": {},
            "scripts": {},
            "assets": {
                "textures": {},
                "models": {},
                "audio": {}
            },
                    "ui": {}
                },
                "configuration_files": [
                    "project.godot"
                ]
            }
        elif self.current_engine == "html5":
            return {
                "engine": "HTML5/JavaScript",
                "language": "JavaScript",
                "folders": {
                    "src": {
                        "js": {},
                        "css": {}
                    },
                    "assets": {
                        "images": {},
                        "audio": {}
                    },
                    "index.html": {}
                },
                "configuration_files": [
                    "package.json",
                    "webpack.config.js"
                ]
            }
        return {"message": "Engine not supported"}

    def implement_core_mechanics(self, task: Dict[str, Any]) -> Dict[str, Any]:
        """Implement the core game mechanics."""
        # In a real implementation, this would generate actual code files
        # For prototype purposes, generate code descriptions and samples

        # Get mechanics from GDD or project data
        mechanics = []
        if "core_mechanics" in self.code_modules:
            mechanics = self.code_modules["core_mechanics"]
        elif self.project_manager and hasattr(self.project_manager, "project_data"):
            game_elements = self.project_manager.project_data.get("game_elements", {})
            mechanics = game_elements.get("core_mechanics", [])
        # If no mechanics found, use placeholder
        if not mechanics:
            mechanics = ["movement", "collision", "interaction"]

        # Generate code for each mechanic
        code_implementations = {}
        for mechanic in mechanics:
            code_implementations[mechanic] = self._generate_mechanic_code(mechanic)

        # Register the code implementations as an artifact
        artifact_id = self.register_artifact(
            artifact_type="code_implementation",
            artifact_data=code_implementations,
            metadata={
                "description": "Core Game Mechanics Implementation",
                "engine": self.current_engine,
                "language": self.current_language,
                "mechanics": mechanics
            }
        )
        return {
            "status": "completed",
            "artifact_id": artifact_id,
            "code_implementations": code_implementations
        }

    def _generate_mechanic_code(self, mechanic: str) -> Dict[str,
Any]:
        """Generate code for a specific game mechanic."""
        # This would use an LLM to generate appropriate code
        # For prototype, use templates or placeholder text

        code_sample = ""
        description = ""

        if self.current_engine == "unity":
            if mechanic == "movement":
                code_sample = """
using UnityEngine;

public class PlayerMovement : MonoBehaviour
```

```
{
    public float moveSpeed = 5f;
    public float jumpForce = 5f;

    private Rigidbody2D rb;
    private bool isGrounded;

    void Start()
    {
        rb = GetComponent<Rigidbody2D>();
    }

    void Update()
    {
        float moveX = Input.GetAxis("Horizontal");
        rb.velocity = new Vector2(moveX * moveSpeed, rb.velocity.y);

        if (Input.GetButtonDown("Jump") && isGrounded)
        {
            rb.AddForce(new Vector2(0f, jumpForce),
ForceMode2D.Impulse);
            isGrounded = false;
        }
    }

    void OnCollisionEnter2D(Collision2D collision)
    {
        if (collision.gameObject.CompareTag("Ground"))
        {
            isGrounded = true;
        }
    }
}
"""
                description = "Unity C# implementation of 2D character movement with jumping mechanics"
            elif mechanic == "collision":
                code_sample = """
using UnityEngine;

public class CollisionHandler : MonoBehaviour
{
    public int playerHealth = 100;
    public AudioClip hitSound;
```

```
private AudioSource audioSource;

void Start()
{
    audioSource = GetComponent<AudioSource>();
}

void OnCollisionEnter2D(Collision2D collision)
{
    if (collision.gameObject.CompareTag("Enemy"))
    {
        TakeDamage(10);
    }
    else if (collision.gameObject.CompareTag("Hazard"))
    {
        TakeDamage(25);
    }
}

void TakeDamage(int damage)
```

```
{
    playerHealth -= damage;


    if (hitSound && audioSource)
    {
        audioSource.PlayOneShot(hitSound);
    }


    if (playerHealth <= 0)
    {
        Die();
    }
}


void Die()
{
    // Death logic
    Debug.Log("Player died");


    // Restart level
```

```
        // SceneManager.LoadScene(SceneManager.GetActiveScene().buildIndex);
    }
}
"""
                description = "Unity C# implementation of collision detection and response system"
            elif mechanic == "interaction":
                code_sample = """
using UnityEngine;

public class InteractionSystem : MonoBehaviour
{
    public float interactionDistance = 2f;
    public LayerMask interactableLayer;
    public KeyCode interactKey = KeyCode.E;

    private Camera mainCamera;

    void Start()
    {
        mainCamera = Camera.main;
```

```
}

void Update()
{
    if (Input.GetKeyDown(interactKey))
    {
        TryInteract();
    }
}

void TryInteract()
{
    RaycastHit2D hit = Physics2D.Raycast(
        transform.position,
        transform.right,
        interactionDistance,
        interactableLayer
    );

    if (hit.collider != null)
    {
            IInteractable interactable =
hit.collider.GetComponent<IInteractable>();

            if (interactable != null)
            {
                interactable.Interact(this.gameObject);
            }
        }
    }
}

// Interface for interactable objects
public interface IInteractable
{
    void Interact(GameObject interactor);
}
"""
                description = "Unity C# implementation of a flexible
interaction system using interfaces"

        elif self.current_engine == "godot":
            if mechanic == "movement":
                code_sample = """
```

```
extends KinematicBody2D

export var speed = 300
export var jump_force = 600
export var gravity = 1200

var velocity = Vector2.ZERO
var is_jumping = false

func _physics_process(delta):
    # Apply gravity
    velocity.y += gravity * delta


    # Get input
    var direction = Input.get_action_strength("ui_right") -
Input.get_action_strength("ui_left")
    velocity.x = direction * speed


    # Jump
    if Input.is_action_just_pressed("ui_select") and is_on_floor():
        velocity.y = -jump_force
        is_jumping = true


    # Apply movement
    velocity = move_and_slide(velocity, Vector2.UP)


    # Reset jump state
    if is_on_floor() and is_jumping:
        is_jumping = false
"""
            description = "Godot GDScript implementation of 2D character movement with jumping mechanics"


        # Add more implementations for other engines/mechanics


        return {
            "code": code_sample,
            "description": description,
            "engine": self.current_engine,
            "language": self.current_language,
            "mechanic": mechanic
        }


    def implement_game_systems(self, task: Dict[str, Any]) -> Dict[str, Any]:
        """Implement game systems like saving, inventory, dialogue, etc."""
        # In a real implementation, would generate actual code files
        # For prototype purposes, generate code descriptions

        # Determine which systems to implement based on the game design
        systems = ["save_system", "game_state", "audio_manager"]

        # Generate code for each system
        system_implementations = {}
        for system in systems:
            system_implementations[system] = self._generate_system_code(system)

        # Register the systems as an artifact
        artifact_id = self.register_artifact(
            artifact_type="game_systems",
            artifact_data=system_implementations,
            metadata={
                "description": "Game Systems Implementation",
                "engine": self.current_engine,
                "language": self.current_language,
                "systems": systems
            }
        )

        return {
            "status": "completed",
            "artifact_id": artifact_id,
            "system_implementations": system_implementations
        }

    def _generate_system_code(self, system: str) -> Dict[str, Any]:
        """Generate code for a specific game system."""
        # This would use an LLM to generate appropriate code
        # For prototype, use templates or placeholder text

        code_sample = ""
        description = ""

        if self.current_engine == "unity":
            if system == "save_system":
                code_sample = """
using UnityEngine;
using System.IO;
using System.Runtime.Serialization.Formatters.Binary;

public class SaveSystem : MonoBehaviour
{
    public static SaveSystem Instance { get; private set; }


    void Awake()
    {
        if (Instance == null)
        {
            Instance = this;
            DontDestroyOnLoad(gameObject);
        }
        else
        {
            Destroy(gameObject);
        }
    }


    public void SaveGame(GameData data, string saveName = "save")
    {
        BinaryFormatter formatter = new BinaryFormatter();
        string path = Application.persistentDataPath + "/" + saveName + ".sav";
        FileStream stream = new FileStream(path, FileMode.Create);

        formatter.Serialize(stream, data);
        stream.Close();

        Debug.Log("Game saved to: " + path);
    }

    public GameData LoadGame(string saveName = "save")
    {
        string path = Application.persistentDataPath + "/" + saveName + ".sav";

        if (File.Exists(path))
        {
            BinaryFormatter formatter = new BinaryFormatter();
            FileStream stream = new FileStream(path, FileMode.Open);
            GameData data = formatter.Deserialize(stream) as GameData;
            stream.Close();

            return data;
        }
        else
        {
            Debug.LogWarning("Save file not found at: " + path);
            return null;
        }
    }
}

[System.Serializable]
public class GameData
{
    public int level;
    public int score;
    public float[] playerPosition;
    public int playerHealth;
    // Add more game state variables as needed
```

```
}
"""
                description = "Unity C# implementation of a save/load system using binary serialization"

        # Add more implementations for other engines/systems

        return {
            "code": code_sample,
            "description": description,
            "engine": self.current_engine,
            "language": self.current_language,
            "system": system
        }

    def create_build_configuration(self, task: Dict[str, Any]) -> Dict[str, Any]:
        """Create build configuration for the target platforms."""
        # In a real implementation, would generate build scripts and configs
        # For prototype purposes, generate build setup descriptions

        # Get target platforms
        platforms = []
        if self.project_manager and hasattr(self.project_manager, "project_data"):
            game_elements = self.project_manager.project_data.get("game_elements", {})
            platforms = game_elements.get("platforms", ["PC"])

        # If no platforms found, use placeholder
        if not platforms:
            platforms = ["PC", "web"]

        # Generate build configuration for each platform
        build_configs = {}
        for platform in platforms:
            build_configs[platform] = self._generate_build_config(platform)

        # Register the build configurations as an artifact
        artifact_id = self.register_artifact(
            artifact_type="build_configuration",
            artifact_data=build_configs,
            metadata={
                "description": "Game Build Configurations",
                "engine": self.current_engine,
                "platforms": platforms
            }
        )

        return {
            "status": "completed",
            "artifact_id": artifact_id,
            "build_configurations": build_configs
        }

    def _generate_build_config(self, platform: str) -> Dict[str, Any]:
        """Generate build configuration for a specific platform."""
        # This would use templates or LLM to generate appropriate build config
        # For prototype, use placeholder text

        config = {}

        if self.current_engine == "unity":
            if platform.lower() == "pc":
                config = {
```

```
            "target_platform": "Windows/Mac/Linux Standalone",
            "architecture": "x86_64",
            "graphics_apis": ["OpenGL", "Vulkan", "DirectX"],
            "build_options": {
                "development_build": False,
                "compress": True,
                "include_debug_symbols": False
            },
            "build_script": "// Build script would go here"
        }
    elif platform.lower() == "mobile":
        config = {
            "target_platform": "Android/iOS",
            "architecture": "ARM64",
            "graphics_apis": ["OpenGL ES", "Metal"],
            "build_options": {
                "development_build": False,
                "compress": True,
                "include_debug_symbols": False
            },
            "build_script": "// Build script would go here"
        }
    elif platform.lower() == "web":
        config = {
            "target_platform": "WebGL",
            "architecture": "WebAssembly",
            "graphics_apis": ["WebGL"],
            "build_options": {
                "development_build": False,
                "compress": True,
                "include_debug_symbols": False
            },
            "build_script": "// Build script would go here"
        }

# Add more implementations for other engines/platforms


return {
    "configuration": config,
    "platform": platform,
    "engine": self.current_engine
}
```

UserInterface.py

```
import uuid
import datetime
from typing import Dict, List, Any, Optional
import json

from base_agent import BaseAgent

class UserInterfaceAgent(BaseAgent):
    """
    Agent responsible for handling communication between the user and the multi-agent system.
    Translates user requirements and feedback into tasks for other agents.
    """


    def __init__(self, project_manager=None):
        super().__init__("user_interface", project_manager)
        self.conversation_history = []
        self.user_requirements = {}
        self.feedback_pending = {}
        self.current_session_id = None
        self.load_knowledge_base()
    def load_knowledge_base(self) -> None:
        """Load UI agent knowledge base."""
        # This would load from an actual knowledge base file/database
        self.knowledge_base = {
            "requirement_templates": {
                "genre": {
                    "prompt": "What genre of game would you like to create?",
                    "examples": ["platformer", "rpg", "puzzle", "shooter", "strategy"],
                    "follow_up_questions": {
                        "platformer": ["How many levels?", "What kind of art style?"],
                        "rpg": ["Turn-based or action?", "What kind of setting?"]
                    }
                },
                "platforms": {
                    "prompt": "Which platforms should the game support?",
                    "examples": ["PC", "mobile", "web", "console"],
                    "follow_up_questions": {
                        "mobile": ["Android, iOS, or both?", "Touch controls or virtual joystick?"],
                        "PC": ["Minimum system requirements?", "Controller support?"]
                    }
                },
                "art_style": {
                    "prompt": "What art style would you prefer?",
                    "examples": ["pixel art", "cartoon", "realistic", "low poly"],
                    "follow_up_questions": {
                        "pixel art": ["Modern or retro?", "Color palette?"],
                        "realistic": ["Stylized or photorealistic?", "Art references?"]
                    }
                }
            },
            "feedback_templates": {
                "design": {
                    "prompt": "What do you think of the game design document?",
                    "follow_up_questions": ["Are the mechanics as you expected?", "Is there anything missing?"]
                },
                "narrative": {
                    "prompt": "How do you feel about the story and characters?",
                    "follow_up_questions": ["Does the story fit your vision?", "Any character changes needed?"]
                },
                "visuals": {
                    "prompt": "What are your thoughts on the visual designs?",
                    "follow_up_questions": ["Is the art style what you expected?", "Any specific visual elements you'd like to change?"]
                }
            }
        }

    def start_new_session(self, user_id: str) -> str:
        """Start a new user session."""
        session_id = str(uuid.uuid4())
        self.current_session_id = session_id

        session_data = {
            "session_id": session_id,
            "user_id": user_id,
            "start_time": datetime.datetime.now().isoformat(),
            "last_activity": datetime.datetime.now().isoformat(),
            "status": "active"
        }

        # In a real system, would store this in a database

        return session_id

    def handle_user_input(self, user_input: str, context: Optional[Dict[str, Any]] = None) -> Dict[str, Any]:
        """
        Process user input and determine appropriate responses or actions.
        This is the main entry point for user interaction.
        """
        if not context:
            context = {}

        # Add to conversation history
        self.conversation_history.append({
            "timestamp": datetime.datetime.now().isoformat(),
            "speaker": "user",
            "content": user_input
        })


        # Determine input type and context to decide how to process
        if not self.user_requirements and "project_id" not in context:
            # New project, start requirements gathering
            return self.start_requirements_gathering(user_input)
        elif "feedback_requested" in context:
            # User is providing feedback on an artifact
            return self.process_user_feedback(user_input,
context.get("artifact_id"), context.get("artifact_type"))
        elif "awaiting_clarification" in context:
            # User is clarifying a previous requirement
            return self.process_requirement_clarification(user_input,
context.get("requirement_key"))
        elif "command" in context:
            # User is giving a specific command
            return self.process_user_command(user_input,
context.get("command"))
        else:
            # General conversation or project-related input
            return self.process_general_input(user_input)
```

```python
    def start_requirements_gathering(self, initial_input: str) -> Dict[str, Any]:
        """Start the process of gathering user requirements for a new game project."""
        # This would use an LLM to understand the initial description and structure it
        # For prototype purposes, extract some basic information and prompt for more

        # Initialize requirements structure
        self.user_requirements = {
            "initial_description": initial_input,
            "genre": None,
            "platforms": [],
            "art_style": None,
            "target_audience": None,
            "core_mechanics": [],
            "narrative_theme": None
        }

        # Extract info from initial input (in a real system, would use NLP/LLM)
        # Simple keyword matching for prototype
        genres = ["platformer", "rpg", "puzzle", "shooter", "strategy"]
```

```
for genre in genres:
    if genre in initial_input.lower():
        self.user_requirements["genre"] = genre
        break

platforms = ["pc", "mobile", "web", "console"]
for platform in platforms:
    if platform in initial_input.lower():
        self.user_requirements["platforms"].append(platform)

art_styles = ["pixel art", "cartoon", "realistic", "low poly"]
for style in art_styles:
    if style in initial_input.lower():
        self.user_requirements["art_style"] = style
        break

# Determine next question to ask based on what's missing
next_question = self.determine_next_requirement_question()

return {
    "response_type": "requirements_gathering",
            "message": f"I'll help you create your game! {next_question['question']}",
            "next_requirement": next_question["key"],
            "current_requirements": self.user_requirements
        }


    def determine_next_requirement_question(self) -> Dict[str, Any]:
        """Determine the next question to ask in the requirements gathering process."""
        # Check which key requirements are missing and formulate a question
        if not self.user_requirements.get("genre"):
            return {
                "key": "genre",
                "question": "What genre of game would you like to create?",
                "examples": self.knowledge_base["requirement_templates"]["genre"]["examples"]
            }
        elif not self.user_requirements.get("platforms"):
            return {
                "key": "platforms",
                "question": "Which platforms should your game support?",
                "examples":
self.knowledge_base["requirement_templates"]["platforms"]
["examples"]

            }

        elif not self.user_requirements.get("art_style"):

            return {

                "key": "art_style",

                "question": "What art style would you prefer for your
game?",

                "examples":
self.knowledge_base["requirement_templates"]["art_style"]
["examples"]

            }

        elif not self.user_requirements.get("target_audience"):

            return {

                "key": "target_audience",

                "question": "Who is your target audience for this game?",

                "examples": ["children", "teens", "adults", "casual
gamers", "hardcore gamers"]

            }

        elif not self.user_requirements.get("core_mechanics") or
len(self.user_requirements.get("core_mechanics", [])) == 0:

            return {

                "key": "core_mechanics",
                "question": "What are the core gameplay mechanics you want in your game?",
                "examples": ["jumping", "shooting", "puzzle solving", "resource management"]
            }
        elif not self.user_requirements.get("narrative_theme"):
            return {
                "key": "narrative_theme",
                "question": "What theme or setting would you like for your game's narrative?",
                "examples": ["fantasy", "sci-fi", "historical", "modern", "abstract"]
            }
        else:
            # All basic requirements gathered
            return {
                "key": "additional_details",
                "question": "Is there anything else you'd like to add to your game requirements?",
                "examples": []
            }

    def process_requirement_clarification(self, user_input: str, requirement_key: str) -> Dict[str, Any]:
        """Process user clarification for a specific requirement."""
        # Update the requirement based on user input
        if requirement_key in self.user_requirements:
            if isinstance(self.user_requirements[requirement_key], list):
                # For list-type requirements, add to list
                items = [item.strip() for item in user_input.split(',')]
                self.user_requirements[requirement_key].extend(items)
            else:
                # For single value requirements, replace
                self.user_requirements[requirement_key] = user_input
        elif requirement_key == "additional_details":
            # Store any additional details
            self.user_requirements["additional_details"] = user_input

            # Requirements gathering complete, initialize project
            if self.project_manager:
                project_name = f"{self.user_requirements.get('genre',
'New')} Game"
                project_id =
self.project_manager.initialize_project(project_name,
self.user_requirements)

                # Register agents if they haven't been registered yet
                # In a real implementation, would check if agents are registered

                # Create initial tasks
                self.project_manager.create_and_assign_initial_tasks()

                return {
                    "response_type": "project_initialized",
                    "message": f"Great! I've created your game project. The initial design phase has begun. I'll update you as we make progress.",
                    "project_id": project_id,
                    "next_steps": "Initial design documents are being created. You'll be able to review them soon."
                }

        # Determine next question or action
        next_question = self.determine_next_requirement_question()

        return {
            "response_type": "requirements_gathering",
            "message": f"Thanks for that information! {next_question['question']}",
            "next_requirement": next_question["key"],
            "current_requirements": self.user_requirements
        }

    def process_user_feedback(self, user_input: str, artifact_id: str,
artifact_type: str) -> Dict[str, Any]:
        """Process user feedback on a specific artifact."""
        # Send feedback to the project manager
        if self.project_manager:
            feedback_content = {}

            # Map artifact type to feedback category
            if artifact_type == "design_document":
                feedback_content["design"] = user_input
            elif artifact_type == "narrative_document":
                feedback_content["narrative"] = user_input
            elif artifact_type == "concept_art":
                feedback_content["visuals"] = user_input
            else:
                feedback_content["general"] = user_input

            # Send feedback message
            self.send_message(
                to_agent="project_manager",
                message_type="user_feedback",
                content=feedback_content
            )

            # Store pending feedback for follow-up
            self.feedback_pending[artifact_id] = {
                "timestamp": datetime.datetime.now().isoformat(),
                "artifact_id": artifact_id,
                "artifact_type": artifact_type,
                "feedback": user_input
            }

            return {
                "response_type": "feedback_received",
                "message": "Thank you for your feedback! The team will incorporate your suggestions into the next iteration.",
                "next_steps": "I'll let you know when the updated version is ready for review."
            }

        return {
            "response_type": "error",
            "message": "Unable to process feedback at this time.",
            "error": "Project manager not available"
        }

    def process_user_command(self, user_input: str, command: str) -> Dict[str, Any]:
        """Process a specific user command."""
        if command == "status":
            # Get project status
            if self.project_manager:
                status = self.project_manager.get_project_status()

                return {
                    "response_type": "status_report",
                    "message": f"Here's the current status of your game project:",
                    "status": status
                }
            else:
                return {
                    "response_type": "error",
                    "message": "Unable to get project status at this time.",
                    "error": "Project manager not available"
                }
        elif command == "help":
            # Provide help information
            return {
                "response_type": "help",
                "message": "Here are some things you can do:",
                "commands": [
                    {"name": "status", "description": "Check the current status of your game project"},
                    {"name": "review [artifact]", "description": "Review a specific artifact"},
                    {"name": "modify [aspect]", "description": "Request changes to a specific aspect of the game"}
                ]
            }

        return {
            "response_type": "unknown_command",
            "message": f"I'm not sure how to handle the command '{command}'. Type 'help' for available commands."
        }
```

```
    def process_general_input(self, user_input: str) -> Dict[str, Any]:
        """Process general user input when not in a specific context."""
        # This would use an LLM to understand the user's intent
        # For prototype purposes, do simple keyword matching

        input_lower = user_input.lower()

        if "status" in input_lower or "progress" in input_lower:
            return self.process_user_command(user_input, "status")
        elif "help" in input_lower:
            return self.process_user_command(user_input, "help")
        elif any(word in input_lower for word in ["review", "check",
"see"]):
            # User wants to review something
            artifact_types = ["design", "narrative", "art", "code"]
            for artifact_type in artifact_types:
                if artifact_type in input_lower:
                    return self.handle_review_request(artifact_type)
        elif any(word in input_lower for word in ["change", "modify",
"update"]):
            # User wants to modify something
            aspects = ["design", "story", "visuals", "mechanics"]
            for aspect in aspects:
                if aspect in input_lower:
                    return self.handle_modification_request(aspect)

        # Default response for input we can't specifically categorize
        return {
            "response_type": "general",
            "message": "I'm here to help you with your game project. You can ask for status updates, review specific aspects, or request modifications. What would you like to do?"
        }

    def handle_review_request(self, artifact_type: str) -> Dict[str, Any]:
        """Handle a user request to review a specific artifact type."""
        # This would retrieve the latest artifact of the requested type
        # For prototype purposes, return a placeholder response

        if not self.project_manager:
            return {
                "response_type": "error",
                "message": "Unable to retrieve artifacts at this time.",
                "error": "Project manager not available"
            }

        # In a real implementation, would query project manager for artifacts
        return {
            "response_type": "review_request",
            "message": f"I'll show you the latest {artifact_type} artifacts as soon as they're ready.",
            "artifact_type": artifact_type,
            "status": "pending"  # In reality, would check if artifacts exist
        }

    def handle_modification_request(self, aspect: str) -> Dict[str, Any]:
        """Handle a user request to modify a specific aspect of the game."""
        # Set up context for gathering modification details

        return {
            "response_type": "modification_request",
            "message": f"I'd be happy to help modify the {aspect} of your game. Please describe what changes you'd like to make.",
            "aspect": aspect,
            "awaiting_details": True
        }

    def handle_message(self, message: Dict[str, Any]) -> None:
        """Handle incoming messages from other agents."""
        message_type = message.get("type", "")
        content = message.get("content", {})

        if message_type == "artifact_ready":
            # An artifact is ready for user review
            artifact_id = content.get("artifact_id")
            artifact_type = content.get("artifact_type")

            # In a real system, would notify the user through the interface
            # For prototype, add to conversation history as system message
            self.conversation_history.append({
                "timestamp": datetime.datetime.now().isoformat(),
                "speaker": "system",
                "content": f"A new {artifact_type} is ready for your review."
            })
        elif message_type == "status_update":
            # Project status update
            status = content.get("status")

            # In a real system, would update the UI
            # For prototype, add to conversation history
            self.conversation_history.append({
                "timestamp": datetime.datetime.now().isoformat(),
                "speaker": "system",
                "content": f"Project status update: {status}"
            })

    def format_response(self, response_data: Dict[str, Any]) -> str:
        """Format a structured response data object into a natural
language response."""
        # This would use an LLM to generate natural, conversational
responses
        # For prototype, use simple templating


        response_type = response_data.get("response_type",
"general")
        message = response_data.get("message", "")
        # Add additional content based on response type
        if response_type == "status_report":
            status = response_data.get("status", {})
            completion = status.get("completion_percentage", 0)
            phase = status.get("current_phase", "")

            message += f"\n\nYour project is {completion}% complete and currently in the {phase} phase."

            # Add milestone information
            milestones = status.get("milestones", [])
            if milestones:
                message += "\n\nMilestones:"
                for milestone in milestones:
                    message += f"\n- {milestone['name']}: {milestone['completion_percentage']}% complete"

        elif response_type == "requirements_gathering":
            # Add examples if available
            next_requirement = response_data.get("next_requirement", "")
            examples = []
            if next_requirement in
self.knowledge_base.get("requirement_templates", {}):
                template =
self.knowledge_base["requirement_templates"][next_requirement]
                examples = template.get("examples", [])

            if examples:
                message += f"\n\nFor example: {', '.join(examples)}"

        # Add to conversation history
        self.conversation_history.append({
            "timestamp": datetime.datetime.now().isoformat(),
            "speaker": "assistant",
            "content": message
        })

        return message

    def get_conversation_history(self, max_entries: Optional[int] =
None) -> List[Dict[str, Any]]:
        """Get recent conversation history."""
        if max_entries:
            return self.conversation_history[-max_entries:]
        return self.conversation_history

    def process_task(self, task: Dict[str, Any]) -> Dict[str, Any]:
        """Process a task assigned to the UI agent."""
        task_name = task.get("name", "")

        if task_name == "Present artifact for review":
            return self.present_artifact_for_review(task)
        elif task_name == "Collect feedback":
            return self.collect_feedback(task)

        return {"status": "failed", "message": "Unknown task type"}

    def present_artifact_for_review(self, task: Dict[str, Any]) ->
Dict[str, Any]:
        """Present an artifact to the user for review."""
        artifact_id = task.get("artifact_id")
        artifact_type = task.get("artifact_type")

        if not artifact_id or not self.project_manager:
            return {
                "status": "failed",
                "message": "Missing artifact information or project manager"
            }

        # Get the artifact
        artifact = self.request_artifact(artifact_id)
        if not artifact:
            return {
                "status": "failed",
                "message": "Failed to retrieve artifact"
            }

        # Format presentation of the artifact (in a real system, would create UI elements)
        presentation = self._format_artifact_presentation(artifact)

        # In a real system, would update the UI with the presentation
        # For prototype, add to conversation history
        self.conversation_history.append({
            "timestamp": datetime.datetime.now().isoformat(),
            "speaker": "system",
            "content": f"Here is the {artifact_type} for your review:
\n\n{presentation}"
        })

        return {
            "status": "completed",
            "message": "Artifact presented for review",
            "artifact_id": artifact_id,
            "artifact_type": artifact_type,
            "feedback_requested": True
        }

    def _format_artifact_presentation(self, artifact: Dict[str, Any]) ->
str:
        """Format an artifact for presentation to the user."""
        artifact_type = artifact.get("artifact_type", "")
        artifact_data = artifact.get("data", {})

        if artifact_type == "design_document":
            # Format game design document
            game_title = artifact_data.get("game_title", "Game")
            genre = artifact_data.get("genre", "Unknown")
            platforms = ", ".join(artifact_data.get("target_platforms",
["Unknown"]))

            presentation = f"== GAME DESIGN DOCUMENT ==\n\n"
            presentation += f"Title: {game_title}\n"
            presentation += f"Genre: {genre}\n"
            presentation += f"Platforms: {platforms}\n\n"

            # Add core mechanics
            presentation += "Core Mechanics:\n"
            for mechanic, details in artifact_data.get("core_mechanics",
{}).items():
                presentation += f"- {mechanic}\n"

            # Add more sections as needed

            return presentation

        elif artifact_type == "narrative_document":
            # Format narrative document
            title = artifact_data.get("title", "Story")
            theme = artifact_data.get("theme", "Unknown")
            presentation = f"== NARRATIVE DESIGN ==\n\n"
            presentation += f"Title: {title}\n"
            presentation += f"Theme: {theme}\n\n"

            # Add character descriptions
            if "characters" in artifact_data:
                presentation += "Characters:\n"
                for character, details in artifact_data.get("characters",
{}).items():
                    presentation += f"- {character}:
{details.get('description', '')}\n"

            # Add plot summary
            if "plot_summary" in artifact_data:
                presentation += f"\nPlot Summary:
\n{artifact_data['plot_summary']}\n"

            return presentation

        elif artifact_type == "concept_art":
            # Format concept art presentation
            presentation = f"== CONCEPT ART ==\n\n"
            for item, details in artifact_data.items():

                presentation += f"{item.replace('_', ' ').title()}:\n"

                presentation += f"{details.get('description', '')}\n\n"


            return presentation


        # Default formatting for unknown artifact types

        return f"== {artifact_type.upper()}
==\n\n{json.dumps(artifact_data, indent=2)}"


    def collect_feedback(self, task: Dict[str, Any]) -> Dict[str, Any]:

        """Collect user feedback on an artifact."""

        artifact_id = task.get("artifact_id")

        artifact_type = task.get("artifact_type")


        # In a real system, would prompt the user for feedback through
UI

        # For prototype, add to conversation history

        self.conversation_history.append({

            "timestamp": datetime.datetime.now().isoformat(),

            "speaker": "system",
            "content": f"What do you think of the {artifact_type}? Please provide your feedback."
        })

        return {
            "status": "completed",
            "message": "Feedback request sent to user",
            "artifact_id": artifact_id,
            "artifact_type": artifact_type,
            "awaiting_feedback": True
        }
```

System Orchestrator

```
import os
import json
import argparse
import logging
import time
from typing import Dict, List, Any, Optional
import uuid
import datetime

# Import all agent classes
```

```
from base_agent import BaseAgent
from project_manager_agent import ProjectManagerAgent
from game_designer_agent import GameDesignerAgent
from narrative_designer_agent import NarrativeDesignerAgent
from visual_artist_agent import VisualArtistAgent
from code_generator_agent import CodeGeneratorAgent
from user_interface_agent import UserInterfaceAgent

# Add imports for other agents when implemented
# from level_designer_agent import LevelDesignerAgent
# from sound_designer_agent import SoundDesignerAgent
# from qa_tester_agent import QATesterAgent
# from asset_manager_agent import AssetManagerAgent

class GameDevAgentSystem:
    """
    Main orchestrator for the multi-agent game development system.
    Handles agent initialization, coordination, and system operations.
    """


    def __init__(self, config_path: str = None, log_level: str = "INFO"):
        """Initialize the system with configuration."""
        # Setup logging
        self._setup_logging(log_level)

        # Load configuration
        self.config = self._load_config(config_path)

        # Initialize components
        self.project_manager = None
        self.agents = {}
        self.active_projects = {}
        self.user_sessions = {}

        # Initialize system
        self._initialize_system()

    def _setup_logging(self, log_level: str) -> None:
        """Setup logging for the system."""
        log_levels = {
            "DEBUG": logging.DEBUG,
            "INFO": logging.INFO,
            "WARNING": logging.WARNING,
            "ERROR": logging.ERROR,
            "CRITICAL": logging.CRITICAL
        }

        level = log_levels.get(log_level.upper(), logging.INFO)

        logging.basicConfig(
            level=level,
            format='%(asctime)s - %(name)s - %(levelname)s - %(message)s',
            handlers=[
                logging.FileHandler("game_dev_system.log"),
                logging.StreamHandler()
            ]
        )

        self.logger = logging.getLogger("GameDevSystem")
        self.logger.info("Logging initialized")

    def _load_config(self, config_path: Optional[str]) -> Dict[str, Any]:
        """Load system configuration from file."""
        default_config = {
            "system": {
```

```
            "save_directory": "./projects",
            "checkpoint_interval": 300,  # seconds
            "max_projects": 10
        },
        "agents": {
            "use_real_ai_models": False,
            "llm_config": {
                "model": "gpt-4",
                "temperature": 0.7,
                "max_tokens": 2000
            },
            "image_model_config": {
                "model": "stable-diffusion",
                "width": 1024,
                "height": 1024
            }
        }
    }

    if not config_path or not os.path.exists(config_path):
        self.logger.warning(f"Config file not found, using default 
configuration")
```

```
    return default_config

try:
    with open(config_path, 'r') as f:
        config = json.load(f)

    # Merge with defaults for any missing values
    merged_config = default_config.copy()
    for section, values in config.items():
        if section in merged_config:
            merged_config[section].update(values)
        else:
            merged_config[section] = values

    self.logger.info(f"Configuration loaded from {config_path}")
    return merged_config

except Exception as e:
    self.logger.error(f"Error loading config: {str(e)}")
    return default_config
```

```
def _initialize_system(self) -> None:
    """Initialize the system and create agent instances."""
    self.logger.info("Initializing system")

    # Create save directory if it doesn't exist
    save_dir = self.config["system"]["save_directory"]
    if not os.path.exists(save_dir):
        os.makedirs(save_dir)

    # Initialize project manager first
    self.project_manager = ProjectManagerAgent()

    # Initialize agents
    self._initialize_agents()

    self.logger.info("System initialization complete")

def _initialize_agents(self) -> None:
    """Initialize all agent instances."""
    self.logger.info("Initializing agents")
```

```
        # Create agent instances and register with project manager
        agents_config = self.config["agents"]
        use_real_ai = agents_config.get("use_real_ai_models", False)

        # Game Designer Agent
        self.agents["game_designer"] = GameDesignerAgent(self.project_manager)
        self.project_manager.register_agent(self.agents["game_designer"])

        # Narrative Designer Agent
        self.agents["narrative_designer"] = NarrativeDesignerAgent(self.project_manager)
        self.project_manager.register_agent(self.agents["narrative_designer"])

        # Visual Artist Agent
        visual_artist = VisualArtistAgent(
            self.project_manager,
            use_real_ai_models=use_real_ai,
            model_config=agents_config.get("image_model_config")
        )
        self.agents["visual_artist"] = visual_artist
        self.project_manager.register_agent(visual_artist)

        # Code Generator Agent
        self.agents["code_generator"] = CodeGeneratorAgent(self.project_manager)
        self.project_manager.register_agent(self.agents["code_generator"])

        # User Interface Agent
        self.agents["user_interface"] = UserInterfaceAgent(self.project_manager)
        self.project_manager.register_agent(self.agents["user_interface"])

        # Add more agents as they are implemented
        # self.agents["level_designer"] = LevelDesignerAgent(self.project_manager)
        # self.project_manager.register_agent(self.agents["level_designer"])

        # self.agents["sound_designer"] = SoundDesignerAgent(self.project_manager)
        # self.project_manager.register_agent(self.agents["sound_designer"])
        # self.agents["qa_tester"] =
QATesterAgent(self.project_manager)
        # self.project_manager.register_agent(self.agents["qa_tester"])

        # self.agents["asset_manager"] =
AssetManagerAgent(self.project_manager)
        #
self.project_manager.register_agent(self.agents["asset_manager"])

        self.logger.info(f"Initialized {len(self.agents)} agents")

    def create_project(self, project_name: str, requirements: Dict[str,
Any], user_id: str) -> str:
        """Create a new game development project."""
        self.logger.info(f"Creating new project: {project_name}")

        # Check if we're at the maximum number of active projects
        if len(self.active_projects) >= self.config["system"]
["max_projects"]:
            oldest_project_id = min(self.active_projects, key=lambda p:
self.active_projects[p]["creation_time"])
            self._archive_project(oldest_project_id)

        # Initialize the project
        project_id =
self.project_manager.initialize_project(project_name, requirements)

        # Store project in active projects
        self.active_projects[project_id] = {
            "project_id": project_id,
            "project_name": project_name,
            "user_id": user_id,
            "creation_time": datetime.datetime.now(),
            "last_activity": datetime.datetime.now()
        }

        # Create initial tasks
        self.project_manager.create_and_assign_initial_tasks()

        # Save initial project state
        self._save_project_state(project_id)

        self.logger.info(f"Project created with ID: {project_id}")
        return project_id

    def load_project(self, project_id: str) -> bool:
        """Load a project from storage."""
        project_path = os.path.join(self.config["system"]["save_directory"], f"{project_id}.json")

        if not os.path.exists(project_path):
            self.logger.error(f"Project file not found: {project_path}")
            return False

        try:
            self.logger.info(f"Loading project: {project_id}")
            self.project_manager.load_project_state(project_path)

            # Add to active projects
            if project_id not in self.active_projects:
                self.active_projects[project_id] = {
                    "project_id": project_id,
                    "project_name": self.project_manager.project_data.get("project_name", "Unknown"),
                    "user_id": "unknown",  # Would be stored in the project data
                    "creation_time": datetime.datetime.now(),
                    "last_activity": datetime.datetime.now()
                }
            return True

        except Exception as e:
            self.logger.error(f"Error loading project: {str(e)}")
            return False

    def _save_project_state(self, project_id: str) -> None:
        """Save the current state of a project."""
        if not self.project_manager:
            self.logger.warning("Cannot save project: Project manager not initialized")
            return

        save_path = os.path.join(self.config["system"]["save_directory"], f"{project_id}.json")

        try:
            self.project_manager.save_project_state(save_path)
            self.logger.info(f"Project state saved to: {save_path}")
        except Exception as e:
            self.logger.error(f"Error saving project state: {str(e)}")
    def _archive_project(self, project_id: str) -> None:
        """Archive a project to free up active slots."""
        if project_id in self.active_projects:
            # Save final state
            self._save_project_state(project_id)

            # Move to archive directory
            archive_dir = os.path.join(self.config["system"]
["save_directory"], "archive")
            if not os.path.exists(archive_dir):
                os.makedirs(archive_dir)

            source = os.path.join(self.config["system"]["save_directory"],
f"{project_id}.json")
            destination = os.path.join(archive_dir, f"{project_id}.json")

            try:
                import shutil
                shutil.move(source, destination)
                self.logger.info(f"Project {project_id} archived")
            except Exception as e:
```

```
            self.logger.error(f"Error archiving project: {str(e)}")


        # Remove from active projects

        del self.active_projects[project_id]


    def process_user_input(self, user_input: str, user_id: str,
session_id: Optional[str] = None, context: Optional[Dict[str, Any]] =
None) -> Dict[str, Any]:

        """

        Process user input and route to the appropriate agent.

        This is the main entry point for user interaction.

        """

        # Ensure user has a session

        if not session_id or session_id not in self.user_sessions:

            # Create new session

            ui_agent = self.agents.get("user_interface")

            if ui_agent:

                session_id = ui_agent.start_new_session(user_id)

                self.user_sessions[session_id] = {

                    "user_id": user_id,

                    "session_id": session_id,

                    "start_time": datetime.datetime.now(),
                    "last_activity": datetime.datetime.now()
                }
            else:
                self.logger.error("User interface agent not available")
                return {"error": "System not initialized properly"}

        # Update session activity time
        if session_id in self.user_sessions:
            self.user_sessions[session_id]["last_activity"] = datetime.datetime.now()

        # Process input through user interface agent
        ui_agent = self.agents.get("user_interface")
        if not ui_agent:
            self.logger.error("User interface agent not available")
            return {"error": "System not initialized properly"}

        # Let UI agent handle the input
        response_data = ui_agent.handle_user_input(user_input, context)

        # Check if this is creating a new project
        if response_data.get("response_type") == "project_initialized" and "project_id" in response_data:
            project_id = response_data["project_id"]
            # Update session with project
            if session_id in self.user_sessions:
                self.user_sessions[session_id]["project_id"] = project_id

        # Format the response for the user
        response_text = ui_agent.format_response(response_data)

        return {
            "session_id": session_id,
            "response": response_text,
            "response_data": response_data
        }

    def run_agent_cycle(self) -> None:
        """Run a single processing cycle for all agents."""
        self.logger.debug("Running agent processing cycle")

        for agent_name, agent in self.agents.items():
            # Process any pending messages
            agent.process_messages()

            # Run current tasks
            agent.run_tasks()

    def start_background_processing(self, interval: Optional[float] = None) -> None:
        """Start background processing loop."""
        if not interval:
            interval = 1.0  # Default to 1 second between cycles

        self.logger.info(f"Starting background processing with interval: {interval}s")

        try:
            while True:
                self.run_agent_cycle()

                # Checkpoint projects periodically
                self._check_for_project_checkpoints()

                time.sleep(interval)
        except KeyboardInterrupt:
            self.logger.info("Background processing stopped by user")
        except Exception as e:
            self.logger.error(f"Error in background processing: {str(e)}")

    def _check_for_project_checkpoints(self) -> None:
        """Check if any projects need to be checkpointed."""
        checkpoint_interval = self.config["system"]
["checkpoint_interval"]
        current_time = time.time()

        for project_id, project_info in self.active_projects.items():
            last_activity = project_info.get("last_checkpoint_time", 0)

            if current_time - last_activity > checkpoint_interval:
                self._save_project_state(project_id)
                self.active_projects[project_id]["last_checkpoint_time"] =
current_time

    def get_project_status(self, project_id: str) -> Optional[Dict[str,
Any]]:
        """Get the current status of a project."""
        if project_id not in self.active_projects:
            # Try to load it first
            if not self.load_project(project_id):
                self.logger.warning(f"Project not found: {project_id}")
                return None

        return self.project_manager.get_project_status()

    def shutdown(self) -> None:
        """Shutdown the system, saving all current states."""
        self.logger.info("Shutting down system")

        # Save all active projects
        for project_id in self.active_projects:
            self._save_project_state(project_id)

        self.logger.info("System shutdown complete")
def main():
    """Main entry point when run as a script."""
    parser = argparse.ArgumentParser(description="Game Development Multi-Agent System")
    parser.add_argument("--config", help="Path to configuration file")
    parser.add_argument("--log-level", default="INFO", help="Logging level")
    args = parser.parse_args()


    system = GameDevAgentSystem(config_path=args.config, log_level=args.log_level)


    # Start background processing
    system.start_background_processing()


if __name__ == "__main__":
    main()
```

Flask API code:

```
from flask import Flask, request, jsonify, send_from_directory
import os
import uuid
import threading
import json
from typing import Dict, List, Any, Optional

# Import the system orchestrator
from system_orchestrator import GameDevAgentSystem
```

```
app = Flask(__name__)

# Initialize the game dev system

system = GameDevAgentSystem()

# Start background processing in a separate thread

processing_thread =
threading.Thread(target=system.start_background_processing,
args=(0.5,))

processing_thread.daemon = True

processing_thread.start()

# Store user sessions

user_sessions = {}

@app.route('/api/health', methods=['GET'])

def health_check():

    """Health check endpoint."""

    return jsonify({"status": "ok", "version": "1.0.0"})

@app.route('/api/session', methods=['POST'])

def create_session():

    """Create a new user session."""

    data = request.json or {}

    user_id = data.get('user_id', str(uuid.uuid4()))
    # Process through system to create session
    ui_agent = system.agents.get("user_interface")
    if ui_agent:
        session_id = ui_agent.start_new_session(user_id)
        user_sessions[session_id] = {
            "user_id": user_id,
            "session_id": session_id
        }
        return jsonify({"session_id": session_id, "user_id": user_id})

    return jsonify({"error": "Failed to create session"}), 500
@app.route('/api/chat', methods=['POST'])
def process_message():
    """Process a user message."""
    data = request.json or {}

    user_input = data.get('message', '')
    session_id = data.get('session_id', '')
    context = data.get('context', {})
    if not session_id:
        return jsonify({"error": "No session ID provided"}), 400

    if session_id not in user_sessions:
        return jsonify({"error": "Invalid session ID"}), 403

    user_id = user_sessions[session_id].get("user_id", "unknown")

    # Process the message through the system
    result = system.process_user_input(user_input, user_id, session_id, context)

    return jsonify(result)
@app.route('/api/projects/<project_id>/status', methods=['GET'])
def project_status(project_id):
    """Get the status of a project."""
    session_id = request.args.get('session_id', '')

    if not session_id or session_id not in user_sessions:
        return jsonify({"error": "Invalid session"}), 403
    status = system.get_project_status(project_id)

    if not status:
        return jsonify({"error": "Project not found"}), 404

    return jsonify({"project_id": project_id, "status": status})
@app.route('/api/projects', methods=['POST'])
def create_project():
    """Create a new project."""
    data = request.json or {}
    session_id = data.get('session_id', '')

    if not session_id or session_id not in user_sessions:
        return jsonify({"error": "Invalid session"}), 403

    project_name = data.get('project_name', 'New Game')
    requirements = data.get('requirements', {})
    user_id = user_sessions[session_id].get("user_id", "unknown")

    project_id = system.create_project(project_name, requirements,
user_id)
```

```
    return jsonify({
        "project_id": project_id,
        "project_name": project_name,
        "status": "initialized"
    })

@app.route('/api/projects/<project_id>/artifacts', methods=['GET'])
def list_artifacts(project_id):
    """List artifacts for a project."""
    session_id = request.args.get('session_id', '')

    if not session_id or session_id not in user_sessions:
        return jsonify({"error": "Invalid session"}), 403

    # Check if project exists
    status = system.get_project_status(project_id)
    if not status:
        return jsonify({"error": "Project not found"}), 404

    # Get artifacts from project manager
    artifacts = []
    if system.project_manager and hasattr(system.project_manager, "artifacts"):
        for artifact_id, artifact in system.project_manager.artifacts.items():
            artifacts.append({
                "artifact_id": artifact_id,
                "artifact_type": artifact.get("artifact_type", "unknown"),
                "creation_date": artifact.get("creation_date", ""),
                "metadata": artifact.get("metadata", {})
            })

    return jsonify({
        "project_id": project_id,
        "artifacts": artifacts
    })

@app.route('/api/projects/<project_id>/artifacts/<artifact_id>', methods=['GET'])
def get_artifact(project_id, artifact_id):
    """Get a specific artifact."""
    session_id = request.args.get('session_id', '')

    if not session_id or session_id not in user_sessions:
        return jsonify({"error": "Invalid session"}), 403


    # Get artifact from project manager

    if system.project_manager and hasattr(system.project_manager,
"artifacts"):

        artifact = system.project_manager.artifacts.get(artifact_id)


        if artifact:

            return jsonify({

                "project_id": project_id,

                "artifact_id": artifact_id,

                "artifact": artifact

            })


    return jsonify({"error": "Artifact not found"}), 404

@app.route('/api/projects/<project_id>/feedback',
methods=['POST'])

def submit_feedback(project_id):

    """Submit feedback on a project or artifact."""

    data = request.json or {}

    session_id = data.get('session_id', '')
```

```
if not session_id or session_id not in user_sessions:
    return jsonify({"error": "Invalid session"}), 403

feedback = data.get('feedback', '')
artifact_id = data.get('artifact_id', '')
feedback_type = data.get('type', 'general')

if not feedback:
    return jsonify({"error": "No feedback provided"}), 400

# Create feedback content
feedback_content = {
    feedback_type: feedback
}

if artifact_id:
    feedback_content["artifact_id"] = artifact_id

# Send feedback through UI agent
ui_agent = system.agents.get("user_interface")
if ui_agent:
        ui_agent.send_message(
            to_agent="project_manager",
            message_type="user_feedback",
            content=feedback_content
        )
        return jsonify({"status": "Feedback submitted", "project_id": project_id})

    return jsonify({"error": "Failed to submit feedback"}), 500

@app.route('/static/<path:path>')
def serve_static(path):
    """Serve static files."""
    return send_from_directory('static', path)

@app.route('/')
def index():
    """Serve the main application page."""
    return send_from_directory('static', 'index.html')

if __name__ == '__main__':
    # Ensure static directory exists
    if not os.path.exists('static'):
        os.makedirs('static')
        # Create a simple index.html if it doesn't exist
        if not os.path.exists('static/index.html'):
            with open('static/index.html', 'w') as f:
                f.write("""
                <!DOCTYPE html>
                <html>
                <head>
                    <title>Game Development AI Agent System</title>
                    <meta charset="UTF-8">
                    <meta name="viewport" content="width=device-width, initial-scale=1.0">
                    <style>
                        body { font-family: Arial, sans-serif; margin: 0; padding: 20px; }
                        h1 { color: #333; }
                        .container { max-width: 800px; margin: 0 auto; }
                        .chat-container { border: 1px solid #ccc; border-radius: 5px; padding: 10px; height: 400px; overflow-y: auto; margin-bottom: 10px; }
                        .input-container { display: flex; }
                        #user-input { flex-grow: 1; padding: 10px; margin-right: 10px; }
                        button { padding: 10px 20px; background: #4CAF50; color: white; border: none; border-radius: 5px; cursor: pointer; }
                    </style>
                </head>
                <body>
                    <div class="container">
                        <h1>Game Development AI Agent System</h1>
                        <div class="chat-container" id="chat-container"></div>
                        <div class="input-container">
                            <input type="text" id="user-input" placeholder="Describe your game idea...">
                            <button id="send-btn">Send</button>
                        </div>
                    </div>

                    <script>
                        let sessionId = '';

                        // Initialize session
                        async function initSession() {
                            try {
                                const response = await fetch('/api/session', {
                                    method: 'POST',
                                    headers: {
                                        'Content-Type': 'application/json'
                                    },
                                    body: JSON.stringify({
                                        session_id: sessionId,
                                        message: message
                                    })
                                });

                                const data = await response.json();
                                addMessage('assistant', data.response);
                            } catch (error) {
                                console.error('Error sending message:', error);
                                addMessage('system', 'Error communicating
            with the server.');
                            }
                        }

                        // Add a message to the chat
                        function addMessage(sender, message) {
```

```
            const chatContainer =
document.getElementById('chat-container');

            const messageElement =
document.createElement('div');

            messageElement.style.marginBottom = '10px';

            messageElement.style.padding = '10px';

            messageElement.style.borderRadius = '5px';



            if (sender === 'user') {

              messageElement.style.backgroundColor =
'#e6f7ff';

              messageElement.style.textAlign = 'right';

              messageElement.innerHTML = `<strong>You:</
strong> ${message}`;

            } else if (sender === 'assistant') {

              messageElement.style.backgroundColor =
'#f0f0f0';

              messageElement.innerHTML =
`<strong>Assistant:</strong> ${message}`;

            } else {

              messageElement.style.backgroundColor =
'#f8f8f8';

              messageElement.style.color = '#666';

              messageElement.style.fontStyle = 'italic';

              messageElement.innerHTML = message;
            }

            chatContainer.appendChild(messageElement);
            chatContainer.scrollTop = chatContainer.scrollHeight;
        }

        // Event listeners
        document.getElementById('send-btn').addEventListener('click', () => {
            const userInput = document.getElementById('user-input');
            const message = userInput.value.trim();

            if (message) {
                sendMessage(message);
                userInput.value = '';
            }
        });

        document.getElementById('user-input').addEventListener('keypress', (e) => {
            if (e.key === 'Enter') {
```

```
                                const userInput = document.getElementById('user-input');
                                const message = userInput.value.trim();

                                if (message) {
                                    sendMessage(message);
                                    userInput.value = '';
                                }
                            }
                        });

                        // Initialize on page load
                        window.onload = initSession;
                    </script>
                </body>
                </html>
                """)

    # Start the Flask server
    app.run(host='0.0.0.0', port=5000, debug=True)
({})
                                });
                                const data = await response.json();
                                sessionId = data.session_id;

                                // Add welcome message
                                addMessage('system', 'Welcome to the Game Development AI Agent System! Describe your game idea to get started.');
                            } catch (error) {
                                console.error('Error initializing session:', error);
                                addMessage('system', 'Error connecting to the server. Please refresh the page.');
                            }
                        }

                        // Send a message
                        async function sendMessage(message) {
                            if (!sessionId) {
                                addMessage('system', 'Session not initialized. Please refresh the page.');
                                return;
                            }
```

```
addMessage('user', message);

try {
    const response = await fetch('/api/chat', {
        method: 'POST',
        headers: {
            'Content-Type': 'application/json'
        },
        body: JSON.stringify
```

Sample Usage Script

```
#!/usr/bin/env python3
"""
Sample script demonstrating how to use the Game Development Multi-Agent System.
This provides a command-line interface to interact with the system.
"""

import argparse
import json
import os
import sys
import time
```

```
from typing import Dict, Any, Optional

# Import the system orchestrator
from system_orchestrator import GameDevAgentSystem

def print_colored(text: str, color: str = "default") -> None:
    """Print colored text in the terminal."""
    colors = {
        "red": "\033[91m",
        "green": "\033[92m",
        "yellow": "\033[93m",
        "blue": "\033[94m",
        "magenta": "\033[95m",
        "cyan": "\033[96m",
        "default": "\033[0m",
    }


    color_code = colors.get(color.lower(), colors["default"])
    end_color = colors["default"]


    print(f"{color_code}{text}{end_color}")

def print_banner() -> None:
    """Print a welcome banner for the application."""
    banner = """
    ╔══════════════════════════════════════════════════════════════╗
    ║                                                              ║
    ║   Game Development Multi-Agent System                        ║
    ║   Create video games using AI agents                         ║
    ║                                                              ║
    ╚══════════════════════════════════════════════════════════════╝
    """
    print_colored(banner, "cyan")

def get_user_input(prompt: str) -> str:
    """Get input from the user with a colored prompt."""
    print_colored(f"\n{prompt}", "yellow")
    return input("> ")

def display_response(response: Dict[str, Any]) -> None:
    """Display a formatted system response."""
    if "error" in response:
        print_colored(f"\nERROR: {response['error']}", "red")
        return
    if "response" in response:
        print_colored("\nAI Assistant:", "green")
        print(response["response"])
    else:
        print_colored("\nSystem Response:", "blue")
        print(json.dumps(response, indent=2))

def interactive_mode(system: GameDevAgentSystem) -> None:
    """Run the system in interactive command-line mode."""
    print_banner()
    print_colored("\nWelcome to the Game Development Multi-Agent System!", "cyan")
    print("This system will help you create a video game using AI agents.")
    print("Type 'quit' or 'exit' to end the session.")


    user_id = f"cli_user_{int(time.time())}"
    session_id = None
    context = {}


    # Create a new session
    ui_agent = system.agents.get("user_interface")
    if ui_agent:
        session_id = ui_agent.start_new_session(user_id)
        print_colored(f"\nSession created. ID: {session_id}", "blue")
    else:
        print_colored("\nERROR: Failed to initialize the system.", "red")
        return

    # Main interaction loop
    while True:
        user_input = get_user_input("What kind of game would you like to create?")

        if user_input.lower() in ["quit", "exit"]:
            print_colored("\nThank you for using the Game Development Multi-Agent System!", "cyan")
            break

        # Process user input
        response = system.process_user_input(user_input, user_id, session_id, context)
        display_response(response)

        # Update context based on response data
        if "response_data" in response:
            response_data = response["response_data"]

            # Update context
            if "next_requirement" in response_data:
                context["awaiting_clarification"] = True
                context["requirement_key"] = response_data["next_requirement"]
            elif "feedback_requested" in response_data and response_data["feedback_requested"]:
                context["feedback_requested"] = True
                context["artifact_id"] = response_data.get("artifact_id")
                context["artifact_type"] = response_data.get("artifact_type")
            else:
                # Clear context for general input
                context = {}

        # Run an agent cycle to process any tasks
        system.run_agent_cycle()

def main() -> None:
    """Main entry point for the sample script."""
    parser = argparse.ArgumentParser(description="Game Development Multi-Agent System CLI")
    parser.add_argument("--config", help="Path to configuration file")
    parser.add_argument("--log-level", default="INFO", help="Logging level")
    args = parser.parse_args()

    # Initialize the system
    system = GameDevAgentSystem(config_path=args.config, log_level=args.log_level)

    try:
        # Run in interactive mode
        interactive_mode(system)
    except KeyboardInterrupt:
        print_colored("\nExiting...", "yellow")
    finally:
        # Clean shutdown
        system.shutdown()

if __name__ == "__main__":
    main()
```

Guidelines:

```
# requirements.txt

# Core dependencies for the Game Development Multi-Agent Framework


# System Dependencies

Flask==2.3.3

Werkzeug==2.3.7

gunicorn==21.2.0

pydantic==2.1.1

python-dotenv==1.0.0


# AI and NLP dependencies

# Using placeholders - in a real implementation you would use specific libraries

# openai==0.28.0

# llama-index==0.8.4

# langchain==0.0.267

# anthropic==0.5.2

# torch==2.0.1

# transformers==4.32.1

# diffusers==0.21.0

# accelerate==0.22.0
# scipy==1.11.2
# pillow==10.0.0

# Development dependencies
pytest==7.4.0
pytest-cov==4.1.0
flake8==6.1.0
black==23.7.0
isort==5.12.0
mypy==1.5.1

# Additional utilities
tqdm==4.66.1
numpy==1.25.2
pandas==2.1.0
requests==2.31.0
aiohttp==3.8.5
asyncio==3.4.3
websockets==11.0.3

# Installation Instructions
```

# Installation Guide for Game Development Multi-Agent Framework

## Prerequisites

- Python 3.8 or higher

- pip (Python package installer)

- Virtual environment tool (optional but recommended)

## Setup

### 1. Clone the repository

```bash

git clone https://github.com/yourusername/game-dev-agent-framework.git

cd game-dev-agent-framework

```

### 2. Create and activate a virtual environment (optional but recommended)

```bash

# Using venv

python -m venv venv

# Activate on Windows

venv\Scripts\activate

# Activate on macOS/Linux

source venv/bin/activate
```

### 3. Install dependencies

```bash
pip install -r requirements.txt
```

### 4. Configure environment variables

Create a `.env` file in the root directory with the following content:

```
# API Keys for AI services (if using real AI models)
OPENAI_API_KEY=your_openai_key
ANTHROPIC_API_KEY=your_anthropic_key

# System Configuration
SAVE_DIRECTORY=./projects
MAX_PROJECTS=10
CHECKPOINT_INTERVAL=300
```

### 5. Run the system

#### Option 1: Web Interface

```bash
```

python main_flask_api.py

~~~

Then open your browser and navigate to `http://localhost:5000`

Option 2: Command Line Interface

```bash

python sample_usage.py

~~~

## Project Structure

```

game-dev-agent-framework/

├── base_agent.py   # Base agent class

├── project_manager_agent.py   # Project manager agent

├── game_designer_agent.py   # Game designer agent

├── narrative_designer_agent.py   # Narrative designer agent

├── visual_artist_agent.py   # Visual artist agent

├── code_generator_agent.py   # Code generator agent

├── user_interface_agent.py   # User interface agent

├── system_orchestrator.py   # System orchestrator

├── main_flask_api.py   # Flask API server

├── sample_usage.py   # Command-line interface
```

```
├── requirements.txt            # Dependencies
├── static/                     # Static web files
│   └── index.html              # Main web interface
├── projects/                   # Project save directory
└── tests/                      # Test files
```

## Adding New Agents

To add a new specialized agent:

1. Create a new file `your_agent_name.py`

2. Extend the `BaseAgent` class

3. Implement required methods (`handle_message`, `process_task`)

4. Add agent initialization to `system_orchestrator.py`

Example:

```python
from base_agent import BaseAgent

class YourAgentName(BaseAgent):
    def __init__(self, project_manager=None):
        super().__init__("your_agent_type", project_manager)
        # Initialize agent-specific properties
```

```
def load_knowledge_base(self) -> None:
    """Load agent-specific knowledge."""
    self.knowledge_base = {
        # Your agent's knowledge
    }

def handle_message(self, message: Dict[str, Any]) -> None:
    """Handle incoming messages."""
    message_type = message.get("type", "")
    content = message.get("content", {})

    # Handle different message types

def process_task(self, task: Dict[str, Any]) -> Dict[str, Any]:
    """Process agent-specific tasks."""
    task_name = task.get("name", "")

    # Process different task types

    return {"status": "completed", "result": {}}
```

```

## Customizing AI Models

If you want to use real AI models, update the configuration and implement the appropriate API calls in the agent classes.

Requirements.txt

```
# requirements.txt

# Core dependencies for the Game Development Multi-Agent Framework

# System Dependencies

Flask==2.3.3

Werkzeug==2.3.7

gunicorn==21.2.0

pydantic==2.1.1

python-dotenv==1.0.0

# AI and NLP dependencies

# Using placeholders - in a real implementation you would use specific libraries

# openai==0.28.0

# llama-index==0.8.4

# langchain==0.0.267

# anthropic==0.5.2
```
```

```
# torch==2.0.1
# transformers==4.32.1
# diffusers==0.21.0
# accelerate==0.22.0
# scipy==1.11.2
# pillow==10.0.0

# Development dependencies
pytest==7.4.0
pytest-cov==4.1.0
flake8==6.1.0
black==23.7.0
isort==5.12.0
mypy==1.5.1

# Additional utilities
tqdm==4.66.1
numpy==1.25.2
pandas==2.1.0
requests==2.31.0
aiohttp==3.8.5
asyncio==3.4.3
websockets==11.0.3
```

# Installation Instructions

# Installation Guide for Game Development Multi-Agent Framework

## Prerequisites

- Python 3.8 or higher

- pip (Python package installer)

- Virtual environment tool (optional but recommended)

## Setup

### 1. Clone the repository

```bash

git clone https://github.com/yourusername/game-dev-agent-framework.git

cd game-dev-agent-framework

```

### 2. Create and activate a virtual environment (optional but recommended)

```bash

# Using venv

python -m venv venv
```

```
# Activate on Windows
venv\Scripts\activate

# Activate on macOS/Linux
source venv/bin/activate
```

### 3. Install dependencies

```bash
pip install -r requirements.txt
```

### 4. Configure environment variables

Create a `.env` file in the root directory with the following content:

```
# API Keys for AI services (if using real AI models)
OPENAI_API_KEY=your_openai_key
ANTHROPIC_API_KEY=your_anthropic_key

# System Configuration
SAVE_DIRECTORY=./projects
MAX_PROJECTS=10
CHECKPOINT_INTERVAL=300
```

### 5. Run the system

#### Option 1: Web Interface

```bash
python main_flask_api.py
```

Then open your browser and navigate to `http://localhost:5000`

#### Option 2: Command Line Interface

```bash
python sample_usage.py
```

## Project Structure

```
game-dev-agent-framework/
├── base_agent.py                  # Base agent class
├── project_manager_agent.py       # Project manager agent
├── game_designer_agent.py         # Game designer agent
├── narrative_designer_agent.py    # Narrative designer agent
├── visual_artist_agent.py         # Visual artist agent
├── code_generator_agent.py        # Code generator agent
├── user_interface_agent.py        # User interface agent
├── system_orchestrator.py         # System orchestrator
├── main_flask_api.py              # Flask API server
├── sample_usage.py                # Command-line interface
├── requirements.txt               # Dependencies
├── static/                        # Static web files
│   └── index.html                 # Main web interface
├── projects/                      # Project save directory
└── tests/                         # Test files
```

## Adding New Agents

To add a new specialized agent:

1. Create a new file `your_agent_name.py`
2. Extend the `BaseAgent` class
3. Implement required methods (`handle_message`, `process_task`)
4. Add agent initialization to `system_orchestrator.py`

Example:

```python
from base_agent import BaseAgent
```

```
class YourAgentName(BaseAgent):
    def __init__(self, project_manager=None):
        super().__init__("your_agent_type", project_manager)
        # Initialize agent-specific properties

    def load_knowledge_base(self) -> None:
        """Load agent-specific knowledge."""
        self.knowledge_base = {
            # Your agent's knowledge
        }

    def handle_message(self, message: Dict[str, Any]) -> None:
        """Handle incoming messages."""
        message_type = message.get("type", "")
        content = message.get("content", {})

        # Handle different message types

    def process_task(self, task: Dict[str, Any]) -> Dict[str, Any]:
        """Process agent-specific tasks."""
        task_name = task.get("name", "")
        # Process different task types


        return {"status": "completed", "result": {}}
```

## Customizing AI Models

If you want to use real AI models, update the configuration and implement the appropriate API calls in the agent classes.

Let me complete the Flask API server code for your game development multi-agent system:

Now I've completed the implementation of your multi-agent framework for video game development. Let me summarize what I've created:

System Components

1. **Base Agent (base_agent.py)**:
    - Abstract base class that defines the common functionality for all agents
    - Handles messaging, task management, and artifact registration
2. **Project Manager Agent (project_manager_agent.py)**:
    - Central coordination agent that manages the project lifecycle
    - Handles task distribution, artifact storage, and project state

3. **Game Designer Agent (game_designer_agent.py)**:
    - Creates game design documents and core gameplay mechanics
    - Manages the overall game design vision
4. **Visual Artist Agent (visual_artist_agent.py)**:
    - Generates concept art and visual design elements
    - Handles art style selection and visual feedback
5. **Code Generator Agent (code_generator_agent.py)**:
    - Implements game mechanics as actual code
    - Handles technical architecture and build configurations
6. **User Interface Agent (user_interface_agent.py)**:
    - Manages communication between users and the system
    - Translates user requirements and feedback
7. **System Orchestrator (system_orchestrator.py)**:
    - Initializes and coordinates all agent components
    - Manages system configuration and background processing
8. **Flask API Server (main_flask_api.py)**:
    - Provides a web interface for interacting with the system
    - Handles HTTP requests and user sessions
9. **Command-line Interface (sample_usage.py)**:
    - Alternative interface for terminal-based interaction
    - Demonstrates system usage patterns

How It Works

1. **User Interaction**:
    - Users describe their game concept to the system
    - The User Interface Agent processes these inputs and creates structured requirements
2. **Initial Design**:
    - The Project Manager creates tasks for the Game Designer and other agents
    - The Game Designer creates a Game Design Document (GDD)
    - The Visual Artist creates concept art based on the requirements
    - The Narrative Designer (not fully implemented) would create the story elements
3. **Development**:
    - Based on the approved design documents, development tasks are assigned
    - The Code Generator implements game mechanics in the chosen engine
    - Level Designer would create game environments (placeholder implementation)
4. **Iteration and Feedback**:
    - Users can provide feedback on any artifact
    - Agents revise their outputs based on feedback
    - The system tracks progress and task dependencies
5. **Final Output**:
    - The system produces a complete game project including code, assets, and documentation
    - All artifacts are maintained in a structured format for deployment

How to Extend

The system is designed to be modular and extensible:

1. **Adding New Agents**: Create a new class that extends BaseAgent and implement the required methods
2. **Integration with Real AI Models**: Replace placeholder implementations with actual AI model calls
3. **Supporting More Game Engines**: Extend the Code Generator Agent with additional engine templates
4. **Adding New Asset Types**: Implement additional artifact types and corresponding agent capabilities

This implementation provides a solid foundation that you can build upon by integrating actual AI models (such as GPT-4 for text generation and Stable Diffusion for image generation), and by implementing additional specialized agents like the Level Designer, QA Tester, and Sound Designer.

System Architecture-

```
flowchart TB
    User[User] --> MasterAgent[Master Agent]


    subgraph "Core System"
        MasterAgent --> ProjectManager[Project Manager Agent]
        ProjectManager --> TaskScheduler[Task Scheduler]
        ProjectManager --> ArtifactRegistry[Artifact Registry]
        ProjectManager --> MessageBroker[Message Broker]
        MessageBroker <--> Redis[(Redis Queue)]
    end
    subgraph "Design Agents"
        ProjectManager --> GameDesigner[Game Designer Agent]
        ProjectManager --> NarrativeDesigner[Narrative Designer Agent]
        ProjectManager --> VisualArtist[Visual Artist Agent]
        ProjectManager --> LevelDesigner[Level Designer Agent]
        ProjectManager --> SoundDesigner[Sound Designer Agent]
    end

    subgraph "Implementation Agents"
        ProjectManager --> CodeGenerator[Code Generator Agent]
        ProjectManager --> AssetManager[Asset Manager Agent]
        ProjectManager --> QATester[QA Tester Agent]
        ProjectManager --> BuildManager[Build Manager Agent]
    end

    subgraph "Storage Layer"
        ProjectManager --> Database[(MongoDB)]
        AssetManager --> ObjectStorage[(S3/Object Storage)]
    end
    subgraph "Game Engine"
        BuildManager --> WebBuildPipeline[Web Build Pipeline]
        WebBuildPipeline --> DeploymentServer[Deployment Server]
    end

    DeploymentServer --> Browser[Browser Playable Game]
    Browser --> User
```

Master Agent Implementation

```
#!/usr/bin/env python3
"""
Master Agent - The coordinator of the entire game development process.
Acts as the primary interface between the user and the agent system.
"""

import os
import json
import uuid
import logging
import datetime
import time
from typing import Dict, List, Any, Optional, Tuple
import threading
import queue

# Import for AI model integration
import openai
from dotenv import load_dotenv

# Import system components
from project_manager_agent import ProjectManagerAgent
from game_designer_agent import GameDesignerAgent
from narrative_designer_agent import NarrativeDesignerAgent
from visual_artist_agent import VisualArtistAgent
from level_designer_agent import LevelDesignerAgent
from sound_designer_agent import SoundDesignerAgent
from code_generator_agent import CodeGeneratorAgent
from asset_manager_agent import AssetManagerAgent
from qa_tester_agent import QATesterAgent
from build_manager_agent import BuildManagerAgent

# Load environment variables
load_dotenv()
```

```
# Configure OpenAI API (or other AI service)
openai.api_key = os.getenv("OPENAI_API_KEY")

class MasterAgent:
    """
    Master Agent that coordinates the entire game development process.
    Acts as the primary interface for the user, processing natural language
    requirements and orchestrating the work of specialized agents.
    """

    def __init__(self, config_path: Optional[str] = None):
        """Initialize the Master Agent with configuration."""
        self.agent_id = str(uuid.uuid4())
        self.agent_type = "master_agent"

        # Setup logging
        logging.basicConfig(
            level=logging.INFO,
            format='%(asctime)s - %(name)s - %(levelname)s - %(message)s',
            handlers=[
                logging.FileHandler("master_agent.log"),
                logging.StreamHandler()
            ]
        )
        self.logger = logging.getLogger("MasterAgent")

        # Load configuration
        self.config = self._load_config(config_path)

        # Initialize state
        self.active_projects = {}
        self.current_project_id = None
        self.conversations = {}

        # Message queues for async processing
        self.message_queue = queue.Queue()
        self.response_queue = queue.Queue()

        # Initialize Project Manager and other agents
        self.project_manager = ProjectManagerAgent()
        self.initialize_agents()
```

```
        # Start background processing thread
        self.running = True
        self.process_thread =
threading.Thread(target=self._background_processing)
        self.process_thread.daemon = True
        self.process_thread.start()

        self.logger.info("Master Agent initialized")

    def _load_config(self, config_path: Optional[str]) -> Dict[str, Any]:
        """Load configuration from file or use defaults."""
        default_config = {
            "ai_model": "gpt-4",
            "temperature": 0.7,
            "project_directory": "./projects",
            "web_deployment_directory": "./web_games",
            "web_server_port": 8080,
            "checkpoint_interval": 300,  # seconds
            "debug_mode": False
        }
        if not config_path or not os.path.exists(config_path):
            self.logger.info("Using default configuration")
            return default_config


        try:
            with open(config_path, 'r') as f:
                config = json.load(f)


            # Merge with defaults
            for key, value in default_config.items():
                if key not in config:
                    config[key] = value


            self.logger.info(f"Loaded configuration from {config_path}")
            return config
        except Exception as e:
            self.logger.error(f"Error loading config: {e}")
            return default_config


    def initialize_agents(self) -> None:
        """Initialize all specialized agents and connect them to the
project manager."""
```

```
        self.logger.info("Initializing specialized agents")


        # Create agent instances

        self.agents = {

            "game_designer":
GameDesignerAgent(self.project_manager),

            "narrative_designer":
NarrativeDesignerAgent(self.project_manager),

            "visual_artist": VisualArtistAgent(self.project_manager),

            "level_designer":
LevelDesignerAgent(self.project_manager),

            "sound_designer":
SoundDesignerAgent(self.project_manager),

            "code_generator":
CodeGeneratorAgent(self.project_manager),

            "asset_manager":
AssetManagerAgent(self.project_manager),

            "qa_tester": QATesterAgent(self.project_manager),

            "build_manager": BuildManagerAgent(self.project_manager)

        }


        # Register agents with the project manager

        for agent_type, agent in self.agents.items():

            self.project_manager.register_agent(agent)
        self.logger.info(f"Initialized {len(self.agents)} specialized agents")

    def process_user_input(self, user_input: str, user_id: str,
                           session_id: Optional[str] = None) -> Dict[str, Any]:
        """
        Process user input and coordinate the appropriate response or action.
        This is the main entry point for user interaction.
        """
        # Generate or retrieve session ID
        if not session_id:
            session_id = str(uuid.uuid4())
            self.conversations[session_id] = {
                "user_id": user_id,
                "messages": [],
                "created_at": datetime.datetime.now().isoformat(),
                "project_id": None
            }

        # Log the user input
        self.logger.info(f"Received user input in session {session_id[:8]}: {user_input[:50]}...")
        self.conversations[session_id]["messages"].append({
            "role": "user",
            "content": user_input,
            "timestamp": datetime.datetime.now().isoformat()
        })

        # Analyze the input using AI to determine intent
        intent, entities = self._analyze_user_intent(user_input, session_id)

        # Process based on the detected intent
        if intent == "create_new_game":
            return self._handle_new_game_request(user_input, session_id, entities)
        elif intent == "modify_existing_game":
            return self._handle_modification_request(user_input, session_id, entities)
        elif intent == "request_status":
            return self._handle_status_request(session_id)
        elif intent == "provide_feedback":
            return self._handle_feedback(user_input, session_id, entities)
        elif intent == "play_game":
            return self._handle_play_request(session_id)
        else:
            # General conversation or unclear intent
            return self._handle_general_conversation(user_input, session_id)


    def _analyze_user_intent(self, user_input: str, session_id: str) -> Tuple[str, Dict[str, Any]]:
        """
        Analyze user input to determine intent and extract relevant entities.
        Uses AI to understand natural language requirements.
        """
        try:
            # Get conversation history for context
            conversation_history = self._get_conversation_history(session_id)


            # Prepare prompt for the AI
            prompt = [
                {"role": "system", "content": "You are an assistant that analyzes user input to determine their intent related to game development. Identify the main intent and extract relevant entities."},
                *conversation_history,
                {"role": "user", "content": user_input}
            ]

            # Call OpenAI API
            response = openai.ChatCompletion.create(
                model=self.config["ai_model"],
                messages=prompt,
                temperature=0.3,  # Lower temperature for more focused
    analysis
                max_tokens=500,
                response_format={"type": "json_object"}
            )

            # Parse the response
            analysis = json.loads(response.choices[0].message.content)

            # Extract intent and entities
            intent = analysis.get("intent", "general_conversation")
            entities = analysis.get("entities", {})
```

```
            self.logger.debug(f"Detected intent: {intent}, entities: {entities}")
            return intent, entities

        except Exception as e:
            self.logger.error(f"Error analyzing user intent: {e}")
            return "general_conversation", {}

    def _get_conversation_history(self, session_id: str, max_messages: int = 10) -> List[Dict[str, str]]:
        """Get recent conversation history for context."""
        if session_id not in self.conversations:
            return []

        # Format recent messages for the AI
        messages = self.conversations[session_id]["messages"][-max_messages:]
        formatted_messages = [
            {"role": msg["role"], "content": msg["content"]}
            for msg in messages
        ]

        return formatted_messages
    def _handle_new_game_request(self, user_input: str, session_id:
str,
                                entities: Dict[str, Any]) -> Dict[str, Any]:
        """Handle a request to create a new game."""
        self.logger.info(f"Handling new game request in session
{session_id[:8]}")

        # Extract requirements from user input using AI
        requirements = self._extract_game_requirements(user_input)

        # Create a new project
        project_name = requirements.get("game_title", "New Game")
        project_id =
self.project_manager.initialize_project(project_name, requirements)

        # Store project ID in session
        self.conversations[session_id]["project_id"] = project_id
        self.current_project_id = project_id

        # Create initial tasks based on requirements
        self.project_manager.create_and_assign_initial_tasks()
        # Generate response to user
        response = self._generate_response(
            "I'll help you create a new game based on your requirements. "
            f"I've created a new project called '{project_name}'. "
            "I'll coordinate our team to design and implement your game. "
            "This includes game design, art, narrative, programming, and testing. "
            "I'll keep you updated on our progress and ask for your feedback at key milestones.",
            session_id
        )

        return {
            "response": response,
            "session_id": session_id,
            "project_id": project_id,
            "status": "project_created"
        }

    def _extract_game_requirements(self, user_input: str) -> Dict[str, Any]:
        """
        Extract structured game requirements from natural language input.
        Uses AI to parse and structure the requirements.
        """
        try:
            # Prepare prompt for the AI
            prompt = [
                {"role": "system", "content": (
                    "Extract structured game requirements from the user's input. "
                    "Include fields like: game_title, genre, platforms, art_style, "
                    "target_audience, core_mechanics, narrative_theme, and any other relevant details."
                )},
                {"role": "user", "content": user_input}
            ]

            # Call OpenAI API
            response = openai.ChatCompletion.create(
                model=self.config["ai_model"],
                messages=prompt,
                temperature=0.3,
                max_tokens=1000,
                response_format={"type": "json_object"}
            )

            # Parse the response
            requirements =
json.loads(response.choices[0].message.content)

            # Ensure required fields exist
            required_fields = ["game_title", "genre", "platforms",
"art_style"]
            for field in required_fields:
                if field not in requirements:
                    if field == "game_title":
                        requirements[field] = "Untitled Game"
                    elif field == "genre":
                        requirements[field] = "casual"
                    elif field == "platforms":
                        requirements[field] = ["web"]
                    elif field == "art_style":
                        requirements[field] = "cartoon"
            self.logger.info(f"Extracted requirements: {requirements}")
            return requirements

        except Exception as e:
            self.logger.error(f"Error extracting game requirements: {e}")
            # Return basic requirements if extraction fails
            return {
                "game_title": "Untitled Game",
                "genre": "casual",
                "platforms": ["web"],
                "art_style": "cartoon",
                "target_audience": "general",
                "core_mechanics": ["simple gameplay"],
                "narrative_theme": "adventure"
            }

    def _handle_modification_request(self, user_input: str,
session_id: str,
                                    entities: Dict[str, Any]) -> Dict[str, Any]:
        """Handle a request to modify an existing game."""
        # Check if there's an active project
        project_id = self.conversations[session_id].get("project_id")
        if not project_id:
            response = self._generate_response(
                "I don't see an active game project. Would you like to create a new game?",
                session_id
            )
            return {"response": response, "session_id": session_id, "status": "no_active_project"}

        # Extract the aspect to modify and the requested changes
        aspect = entities.get("aspect", "general")

        # Create modification task based on the aspect
        if aspect == "gameplay":
            task_agent = "game_designer"
            task_name = "Modify gameplay mechanics"
        elif aspect == "narrative" or aspect == "story":
            task_agent = "narrative_designer"
            task_name = "Modify narrative elements"
        elif aspect == "visuals" or aspect == "art":
            task_agent = "visual_artist"
            task_name = "Modify visual elements"
        elif aspect == "level" or aspect == "map":

            task_agent = "level_designer"

            task_name = "Modify level design"

        elif aspect == "audio" or aspect == "sound":

            task_agent = "sound_designer"

            task_name = "Modify audio elements"

        else:

            task_agent = "game_designer"

            task_name = "General modification"


        # Create the modification task

        task_id = self.project_manager._create_and_assign_task(

            agent_type=task_agent,

            task_name=task_name,

            description=f"Modify {aspect} based on user request:
{user_input}",

            priority=2,

            dependencies=[],

            milestone_id=self.project_manager.project_data["milestones
"][0]["id"]

        )
        # Generate response to user

        response = self._generate_response(

            f"I'll help you modify the {aspect} of your game. "

            f"I've assigned the task to our {task_agent.replace('_', ' ')}. "

            "I'll let you know when the changes are ready for your
review.",

            session_id

        )



        return {

            "response": response,

            "session_id": session_id,

            "project_id": project_id,

            "status": "modification_requested",

            "task_id": task_id

        }



    def _handle_status_request(self, session_id: str) -> Dict[str, Any]:

        """Handle a request for project status."""

        # Check if there's an active project

        project_id = self.conversations[session_id].get("project_id")

        if not project_id:
            response = self._generate_response(
                "I don't see an active game project. Would you like to create a new game?",
                session_id
            )
            return {"response": response, "session_id": session_id, "status": "no_active_project"}

        # Get project status from project manager
        project_status = self.project_manager.get_project_status()

        # Format status information for the user
        status_info = (
            f"Project: {project_status.get('project_name', 'Unknown')}\n"
            f"Current phase: {project_status.get('current_phase', 'Unknown')}\n"
            f"Overall completion: {project_status.get('completion_percentage', 0)}%\n\n"
            "Milestone status:\n"
        )

        for milestone in project_status.get("milestones", []):
            status_info += f"- {milestone['name']}: {milestone['completion_percentage']}% complete\n"
        status_info += "\nRecent activities:\n"
        for activity in project_status.get("recent_activities", [])[:3]:
            status_info += f"- {activity['summary']}\n"

        # Generate response to user
        response = self._generate_response(
            f"Here's the current status of your game project:
\n\n{status_info}",
            session_id
        )

        return {
            "response": response,
            "session_id": session_id,
            "project_id": project_id,
            "status": "status_provided",
            "project_status": project_status
        }

    def _handle_feedback(self, user_input: str, session_id: str,
                        entities: Dict[str, Any]) -> Dict[str, Any]:
```

```
    """Handle user feedback on the game or specific aspects."""

    # Check if there's an active project

    project_id = self.conversations[session_id].get("project_id")

    if not project_id:

        response = self._generate_response(

            "I don't see an active game project. Would you like to create a new game?",

            session_id

        )

        return {"response": response, "session_id": session_id, "status": "no_active_project"}


    # Extract the aspect the feedback is about

    aspect = entities.get("aspect", "general")


    # Create a feedback entry

    feedback_id = str(uuid.uuid4())

    feedback_content = {aspect: user_input}


    if "feedback" not in self.project_manager.project_data:

        self.project_manager.project_data["feedback"] = {}
    self.project_manager.project_data["feedback"][feedback_id] = {
        "feedback_id": feedback_id,
        "timestamp": datetime.datetime.now().isoformat(),
        "content": feedback_content
    }

    # Distribute feedback to relevant agents
    self.project_manager._distribute_feedback(feedback_id)

    # Generate response to user
    response = self._generate_response(
        f"Thank you for your feedback about the {aspect} of your game. "
        "I've shared it with the team, and we'll incorporate your suggestions. "
        "I'll let you know when the updates are ready for your review.",
        session_id
    )

    return {
        "response": response,
        "session_id": session_id,
        "project_id": project_id,
        "status": "feedback_received",
        "feedback_id": feedback_id
    }

def _handle_play_request(self, session_id: str) -> Dict[str, Any]:
    """Handle a request to play the current game."""
    # Check if there's an active project
    project_id = self.conversations[session_id].get("project_id")
    if not project_id:
        response = self._generate_response(
            "I don't see an active game project. Would you like to create a new game?",
            session_id
        )
        return {"response": response, "session_id": session_id, "status": "no_active_project"}

    # Check if the game has been built and is playable
    if "build_manager" in self.agents:
        build_status = self.agents["build_manager"].get_latest_build_status(project_id)
        if build_status.get("status") == "completed":
            game_url = build_status.get("play_url")

            if game_url:
                response = self._generate_response(
                    f"Your game is ready to play! You can play it at: {game_url}",
                    session_id
                )

                return {
                    "response": response,
                    "session_id": session_id,
                    "project_id": project_id,
                    "status": "game_playable",
                    "play_url": game_url
                }

        # Game isn't built yet, check status and potentially trigger a build
        project_status = self.project_manager.get_project_status()
        completion = project_status.get("completion_percentage", 0)
        if completion >= 70:  # If game is mostly complete
            # Trigger a build
            build_task_id = self.project_manager._create_and_assign_task(
                agent_type="build_manager",
                task_name="Build web playable version",
                description="Create a web-playable build of the current game version",
                priority=1,
                dependencies=[],
                milestone_id=self.project_manager.project_data["milestones"][-1]["id"]
            )

            response = self._generate_response(
                "Your game is almost ready, but needs to be built for the web first. "
                "I've started the build process. This should take a few minutes. "
                "I'll let you know when it's ready to play!",
                session_id
            )
            return {
                "response": response,
                "session_id": session_id,
                "project_id": project_id,
                "status": "building_game",
                "task_id": build_task_id
            }
        else:
            response = self._generate_response(
                f"Your game is still in development (about {completion}% complete). "
                "It's not ready to play yet. Would you like to see the current status "
                "or provide feedback on any specific aspects?",
                session_id
            )

            return {
                "response": response,
                "session_id": session_id,
                "project_id": project_id,
                "status": "game_not_ready",
                "completion": completion
            }

        # Build manager not available
        response = self._generate_response(
            "I'm sorry, but I can't build a playable version right now. "
            "The build system is currently unavailable.",
            session_id
        )

        return {
            "response": response,
            "session_id": session_id,
            "project_id": project_id,
            "status": "build_system_unavailable"
        }

    def _handle_general_conversation(self, user_input: str,
session_id: str) -> Dict[str, Any]:
        """Handle general conversation or unclear intent."""
        # Generate a response using AI
        response = self._generate_response(user_input, session_id)
        return {
            "response": response,
            "session_id": session_id,
            "status": "general_conversation"
        }

    def _generate_response(self, content: str, session_id: str) -> str:
        """
        Generate a natural language response, optionally using AI.
        Stores the response in the conversation history.
        """
        # For complex responses, use AI to generate natural language
        if len(content) > 200 or '\\n' in content:
            try:
                # Get conversation history for context
                conversation_history =
self._get_conversation_history(session_id)

                # Prepare prompt for the AI
                prompt = [
                    {"role": "system", "content": "You are a helpful game development coordinator. Respond in a friendly, professional manner."},
                    *conversation_history,
                    {"role": "assistant", "content": content}  # What we want to say
                ]

                # Call OpenAI API
                response = openai.ChatCompletion.create(
                    model=self.config["ai_model"],
                    messages=prompt,
                    temperature=0.7,
                    max_tokens=1000
                )

                # Use the AI-generated response
                content = response.choices[0].message.content

            except Exception as e:
                self.logger.error(f"Error generating AI response: {e}")
                # Fall back to the original content if AI fails
                pass
        # Store the response in conversation history
        if session_id in self.conversations:
            self.conversations[session_id]["messages"].append({
                "role": "assistant",
                "content": content,
                "timestamp": datetime.datetime.now().isoformat()
            })

        return content

    def _background_processing(self) -> None:
        """Background thread for processing agent tasks and messages."""
        self.logger.info("Starting background processing thread")

        while self.running:
            try:
                # Process any pending messages
                while not self.message_queue.empty():
                    message = self.message_queue.get()
                    self._process_message(message)
                # Run agent cycles
                for agent_name, agent in self.agents.items():
                    # Process messages and tasks
                    agent.process_messages()
                    agent.run_tasks()

                # Process project manager
                if self.project_manager:
                    self.project_manager.process_messages()
                    self.project_manager.run_tasks()

                # Check for completed builds that need notification
                self._check_for_completed_builds()

                # Sleep to avoid hammering the CPU
                time.sleep(0.1)

            except Exception as e:
                self.logger.error(f"Error in background processing: {e}")
                time.sleep(1)  # Sleep longer after an error
```

```
def _process_message(self, message: Dict[str, Any]) -> None:
    """Process an incoming message in the background thread."""
    message_type = message.get("type", "")
    content = message.get("content", {})

    if message_type == "build_completed":
        # A game build has completed
        project_id = content.get("project_id")
        session_id = content.get("session_id")
        play_url = content.get("play_url")

        if session_id and session_id in self.conversations:
            # Notify the user that their game is ready to play
            response = self._generate_response(
                f"Great news! Your game is now ready to play. "
                f"You can play it at: {play_url}",
                session_id
            )

            # Add to response queue for the web server to pick up
```

```
                self.response_queue.put({
                    "session_id": session_id,
                    "response": response,
                    "project_id": project_id,
                    "status": "game_playable",
                    "play_url": play_url
                })

    def _check_for_completed_builds(self) -> None:
        """Check for newly completed builds that require user
notification."""
        if "build_manager" not in self.agents:
            return

        build_manager = self.agents["build_manager"]

        # Check for completed builds that haven't been notified
        for project_id, project_info in self.active_projects.items():
            build_status =
build_manager.get_latest_build_status(project_id)

            if (build_status.get("status") == "completed" and
not build_status.get("notified", False)):

# Find the session associated with this project
session_id = None
for sid, session in self.conversations.items():
    if session.get("project_id") == project_id:
        session_id = sid
        break

if session_id:
    # Mark as notified
    build_status["notified"] = True

    # Send a build_completed message to be processed
    self.message_queue.put({
        "type": "build_completed",
        "content": {
            "project_id": project_id,
            "session_id": session_id,
            "play_url": build_status.get("play_url")
        }
```

```
                    })

    def shutdown(self) -> None:
        """Gracefully shut down the Master Agent and all components."""
        self.logger.info("Shutting down Master Agent")

        # Stop background processing
        self.running = False
        if self.process_thread.is_alive():
            self.process_thread.join(timeout=5)

        # Save projects and clean up
        if self.project_manager:
            for project_id in list(self.active_projects.keys()):
                project_path = os.path.join(
                    self.config["project_directory"],
                    f"{project_id}.json"
                )
                self.project_manager.save_project_state(project_path)

        self.logger.info("Master Agent shutdown complete")
```

```
# Example usage
if __name__ == "__main__":
    import argparse


    parser = argparse.ArgumentParser(description="Master Agent for Game Development")
    parser.add_argument("--config", help="Path to configuration file")
    args = parser.parse_args()


    master_agent = MasterAgent(config_path=args.config)


    try:
        while True:
            user_input = input("You: ")
            if user_input.lower() in ["exit", "quit"]:
                break


            response = master_agent.process_user_input(user_input, "cli_user")
            print(f"Master Agent: {response['response']}")


    except KeyboardInterrupt:
        print("\nShutting down...")
    finally:
        master_agent.shutdown()
```

Level Designer Agent Implementation

```
#!/usr/bin/env python3
"""
Level Designer Agent - Responsible for creating game levels and environments.
"""

import os
import uuid
import datetime
import json
from typing import Dict, List, Any, Optional

from base_agent import BaseAgent

class LevelDesignerAgent(BaseAgent):
    """
    Agent responsible for designing game levels and environments.
    Creates level layouts, encounter design, objective placement, and spatial elements.
    """

    def __init__(self, project_manager=None):
        super().__init__("level_designer", project_manager)
        self.level_templates = {}
        self.level_designs = {}
        self.current_level = None
        self.level_metadata = {}
        self.load_knowledge_base()

    def load_knowledge_base(self) -> None:
        """Load level design knowledge base."""
        # This would load from an actual knowledge base file/database
        self.knowledge_base = {
            "level_patterns": {
                "linear": {
                    "description": "A straightforward path from start to finish",
                    "suitable_genres": ["platformer", "action", "adventure"],
                    "key_elements": ["clear path", "progressive difficulty", "pacing elements"]
                },
                "hub_and_spoke": {
                    "description": "Central area with branches leading to objectives",
                    "suitable_genres": ["adventure", "rpg", "metroidvania"],
                    "key_elements": ["central hub", "distinct branches", "unlockable paths"]
                },
                "open_world": {
                    "description": "Large explorable area with scattered objectives",
                    "suitable_genres": ["rpg", "sandbox", "adventure"],
                    "key_elements": ["points of interest", "discovery mechanics", "navigation aids"]
                },
                "arena": {
                    "description": "Enclosed space focused on combat or challenges",
                    "suitable_genres": ["shooter", "fighting", "action"],
                    "key_elements": ["balanced layout", "cover positions", "resource placement"]
                },
                "branching": {
                    "description": "Multiple paths with different challenges or stories",
                    "suitable_genres": ["adventure", "rpg", "puzzle"],
                    "key_elements": ["choice points", "converging paths", "varying difficulty"]

                }

            },

            "encounter_types": {

                "combat": {

                    "variations": ["ambush", "arena fight", "boss battle", "swarm", "duel"],

                    "key_elements": ["enemy placement", "cover", "escape routes", "rewards"]

                },

                "puzzle": {

                    "variations": ["environmental", "logic", "physics", "pattern matching", "timed"],

                    "key_elements": ["clear feedback", "learning curve", "hints", "satisfaction"]

                },

                "exploration": {

                    "variations": ["hidden areas", "scenic vistas", "resource caches", "lore discovery"],

                    "key_elements": ["visual cues", "rewards", "player curiosity", "memorable landmarks"]

                },

                "narrative": {
                    "variations": ["dialogue", "cutscene trigger",
    "environmental storytelling", "log/journal entry"],
                    "key_elements": ["context appropriate", "pacing",
    "emotional impact", "player agency"]
                },
                "platforming": {
                    "variations": ["precision jumping", "moving platforms",
    "hazard avoidance", "timed sequences"],
                    "key_elements": ["fair challenge", "skill progression",
    "recovery options", "rhythm"]
                }
            },
            "environmental_themes": {
                "natural": ["forest", "mountains", "desert", "ocean",
    "caves"],
                "urban": ["city", "ruins", "industrial", "residential",
    "commercial"],
                "fantasy": ["magical forest", "ancient temple", "elemental
    plane", "floating islands"],
                "sci_fi": ["spaceship", "alien planet", "cyberpunk city",
    "research facility"],
                "historical": ["castle", "ancient civilization", "battlefield",
    "archaeological site"]
            },
            "objective_types": {
                "reach_destination": {
                    "description": "Player must reach a specific location",

                    "elements": ["clear signposting", "progressive challenge", "milestone rewards"]

                },

                "defeat_enemies": {

                    "description": "Player must defeat specific or all enemies",

                    "elements": ["enemy variety", "pacing", "challenge curve", "encounter spaces"]

                },

                "collect_items": {

                    "description": "Player must find and collect specific items",

                    "elements": ["distribution pattern", "discovery mechanics", "collection feedback"]

                },

                "solve_puzzle": {

                    "description": "Player must solve one or more puzzles",

                    "elements": ["instruction clarity", "feedback", "scaffolded difficulty"]

                },

                "escort_protect": {

                    "description": "Player must protect or escort an NPC or object",
                    "elements": ["defensive positions", "threat pacing", "ally pathing", "player agency"]
                },
                "timed_challenge": {
                    "description": "Player must complete an objective within time limit",
                    "elements": ["clear timing", "shortcuts", "fail recovery", "time extension pickups"]
                }
            }
        }

        # Load level templates based on genre (in a real system, these would be more detailed)
        self.level_templates = {
            "platformer": {
                "tutorial": {
                    "layout": "linear",
                    "length": "short",
                    "encounters": ["basic_movement", "simple_enemy", "first_collectible"],
                    "obstacles": ["simple_gap", "low_platform", "breakable_object"]
                },
                "standard": {
                    "layout": "linear_with_secrets",
                    "length": "medium",
                    "encounters": ["enemy_group", "mini_boss",
"collectible_challenge"],
                    "obstacles": ["moving_platform", "hazard_zone",
"height_challenge"]
                },
                "boss": {
                    "layout": "arena",
                    "length": "short",
                    "encounters": ["boss_introduction", "boss_fight",
"reward_collection"],
                    "obstacles": ["boss_specific_hazards", "healing_spots",
"phase_transitions"]
                }
            },
            "rpg": {
                "town": {
                    "layout": "hub",
                    "length": "medium",
                    "encounters": ["npc_dialogue", "shop_interaction",
"quest_giver"],
                    "points_of_interest": ["inn", "marketplace", "guild_hall", "town_square"]
                },
                "dungeon": {
                    "layout": "branching",
                    "length": "long",
                    "encounters": ["enemy_groups", "trap_rooms", "mini_boss", "treasure_room"],
                    "obstacles": ["locked_door", "puzzle_room", "collapsing_passage", "dark_area"]
                },
                "wilderness": {
                    "layout": "open",
                    "length": "long",
                    "encounters": ["random_enemies", "traveler_npc", "resource_node", "hidden_cave"],
                    "points_of_interest": ["vista", "campsite", "ruins", "special_tree_or_rock"]
                }
            },
            "shooter": {
                "corridor": {
                    "layout": "linear",
                    "length": "medium",
                    "encounters": ["ambush", "sniper_spot", "close_quarters", "explosive_hazards"],
                    "cover_elements": ["crates", "pillars", "doorways", "vehicle"]
                },
                "arena": {
                    "layout": "circular",
                    "length": "medium",
                    "encounters": ["waves", "boss_fight", "turret_section"],
                    "cover_elements": ["central_structure", "perimeter_walls", "height_variations"]
                },
                "compound": {
                    "layout": "hub_and_spoke",
                    "length": "long",
                    "encounters": ["patrol_routes", "stealth_section", "defense_point", "vehicle_section"],
                    "points_of_interest": ["command_center", "barracks", "motor_pool", "communications"]
                }
            }
        }

    def handle_message(self, message: Dict[str, Any]) -> None:
        """Handle incoming messages."""
        message_type = message.get("type", "")
        content = message.get("content", {})

        if message_type == "request_level_input":
            # Another agent is requesting level design input
            level_input = self.generate_level_input(content.get("topic"), content.get("context"))
            self.send_message(
                to_agent=message["from_agent"],
                message_type="level_input_response",
                content={"level_input": level_input, "request_id": content.get("request_id")}
            )
        elif message_type == "feedback_level":
            # Process feedback on level designs
            self.incorporate_feedback(content.get("feedback"), content.get("artifact_id"))
        elif message_type == "artifact_update":
            # Another agent has created/updated an artifact we should know about
            self.review_artifact(content.get("artifact_id"), content.get("artifact_type"))
```

```
    def generate_level_input(self, topic: str, context: Dict) -> Dict[str, Any]:
        """Generate level design input on a specific topic."""
        # This would use an LLM or other AI to generate level design suggestions
        # For this prototype, we'll return placeholder data
        if topic == "level_structure":
            genre = context.get("genre", "")
            suitable_patterns = [
                pattern for pattern, data in self.knowledge_base["level_patterns"].items()
                if genre in data["suitable_genres"]
            ]

            return {
                "suggested_patterns": suitable_patterns,
                "rationale": f"These level structures are well-suited for the {genre} genre.",
                "key_elements": [
                    self.knowledge_base["level_patterns"][pattern]["key_elements"]
                    for pattern in suitable_patterns
                ]
            }
        elif topic == "encounter_design":
            genre = context.get("genre", "")
            level_type = context.get("level_type", "")


            # Determine appropriate encounter types for the genre and level
            if genre == "platformer":
                encounter_types = ["platforming", "combat", "puzzle"]
            elif genre == "rpg":
                encounter_types = ["combat", "narrative", "exploration"]
            elif genre == "shooter":
                encounter_types = ["combat", "exploration", "timed_challenge"]
            else:
                encounter_types = list(self.knowledge_base["encounter_types"].keys())


            return {
                "suggested_encounters": encounter_types,
                "variations": [
                    self.knowledge_base["encounter_types"][encounter]["variations"]
                    for encounter in encounter_types
                ],
                "key_elements": [
                    self.knowledge_base["encounter_types"][encounter]["key_elements"]
                    for encounter in encounter_types
                ]
            }

        return {"message": "Level design input not available for this topic"}


    def incorporate_feedback(self, feedback: str, artifact_id: str) -> None:
        """Incorporate feedback into level designs."""
        # This would use an LLM to interpret feedback and modify designs
        # For now, just log the feedback
        print(f"Level Designer incorporating feedback for artifact {artifact_id}: {feedback}")

        # In a real implementation, would retrieve the level design, modify it, and update it
        if artifact_id in self.artifacts:
            # Make modifications based on feedback
            # Re-register the updated artifact
            self.register_artifact(
                artifact_type="level_design_update",
                artifact_data=self.artifacts[artifact_id],  # In reality, would be modified
                metadata={
                    "original_artifact_id": artifact_id,
                    "feedback_applied": feedback,
                    "description": "Updated based on feedback"
                }
            )

    def review_artifact(self, artifact_id: str, artifact_type: str) -> None:
        """Review an artifact created by another agent."""
        if not self.project_manager:
            return

        artifact = self.request_artifact(artifact_id)
        if not artifact:
            return

        # Take action based on artifact type
        if artifact_type == "design_document":
            # Extract level-relevant information from game design document
            design_data = artifact.get("data", {})

            # Store relevant design info for level creation
            if "genre" in design_data:
                self.level_metadata["genre"] = design_data["genre"]

            if "level_design" in design_data:
                self.level_metadata["level_structure"] = design_data["level_design"]

            if "core_mechanics" in design_data:
                self.level_metadata["core_mechanics"] = design_data["core_mechanics"]

        elif artifact_type == "narrative_document":
            # Extract narrative elements relevant to level design
            narrative_data = artifact.get("data", {})

            # Store narrative elements that influence level design
            if "locations" in narrative_data:
                self.level_metadata["narrative_locations"] =
narrative_data["locations"]

            if "key_events" in narrative_data:
                self.level_metadata["key_events"] =
narrative_data["key_events"]

    def process_task(self, task: Dict[str, Any]) -> Dict[str, Any]:
        """Process a level design task."""
        task_name = task.get("name", "")

        if task_name == "Create game levels":
            return self.create_game_levels(task)
        elif task_name == "Design specific level":
            return self.design_specific_level(task)
        elif task_name == "Modify level design":
            return self.modify_level_design(task)
        elif task_name == "Create level progression":
            return self.create_level_progression(task)

        return {"status": "failed", "message": "Unknown task type"}
    def create_game_levels(self, task: Dict[str, Any]) -> Dict[str, Any]:
        """Create a collection of game levels based on the game design."""
        # In a real implementation, this would create detailed level designs
        # For prototype purposes, create placeholder data

        # Get game metadata from task or project data
        game_genre = self.level_metadata.get("genre", "")
        if not game_genre and self.project_manager:
            game_elements = self.project_manager.project_data.get("game_elements", {})
            game_genre = game_elements.get("genre", "platformer")

        # Determine how many and what types of levels to create
        if game_genre == "platformer":
            level_types = ["tutorial", "standard", "standard", "standard", "boss"]
        elif game_genre == "rpg":
            level_types = ["town", "wilderness", "dungeon", "town", "dungeon", "wilderness", "dungeon"]
        elif game_genre == "shooter":
            level_types = ["corridor", "arena", "compound", "corridor", "arena"]
        else:
            # Default level progression
            level_types = ["tutorial", "standard", "standard", "boss"]


        # Create level designs
        levels = {}
        for i, level_type in enumerate(level_types):
            level_id = f"level_{i+1}"


            # Get template for this level type and genre
            template = self.level_templates.get(game_genre, {}).get(level_type, {})


            # Create level based on template
            level_design = self._create_level_from_template(level_id, level_type, template, game_genre)


            # Store the level design
            levels[level_id] = level_design


        # Create a level progression
        level_progression = {
            "start_level": "level_1",
            "level_order": list(levels.keys()),
            "branching_paths": {},  # Would contain optional or alternate
paths
            "level_dependencies": {},  # Would contain unlock conditions
            "difficulty_curve": "standard"  # or "challenging", "relaxed",
etc.
        }

        # Package everything together
        level_package = {
            "levels": levels,
            "progression": level_progression,
            "metadata": {
                "genre": game_genre,
                "level_count": len(levels),
                "estimated_playtime": f"{len(levels) * 15}-{len(levels) * 25}
minutes"
            }
        }

        # Register the level designs as an artifact
        artifact_id = self.register_artifact(
            artifact_type="level_designs",
            artifact_data=level_package,
            metadata={
                "description": f"Game Level Designs ({len(levels)} levels)",
                "genre": game_genre,
                "level_types": level_types
            }
        )

        return {
            "status": "completed",
            "artifact_id": artifact_id,
            "level_package": level_package
        }

    def _create_level_from_template(self, level_id: str, level_type: str, template: Dict[str, Any], genre: str) -> Dict[str, Any]:
        """Create a detailed level design based on a template."""
        # Generate a unique level name
        level_names = {
            "platformer": {
```

```
            "tutorial": ["First Steps", "Learning the Ropes", "Basics Training"],
            "standard": ["Wilderness Trail", "Ancient Ruins", "Mystic Caverns", "Cloudy Heights"],
            "boss": ["Final Challenge", "Boss Encounter", "Ultimate Test"]
        },
        "rpg": {
            "town": ["Willowbrook", "Hammerfall", "Riverdale", "Port Harbor"],
            "dungeon": ["Dark Cave", "Ancient Tomb", "Forgotten Tower", "Dragon's Lair"],
            "wilderness": ["Enchanted Forest", "Desert Sands", "Frozen Peaks", "Misty Swamp"]
        },
        "shooter": {
            "corridor": ["Narrow Escape", "Breach and Clear", "Infiltration"],
            "arena": ["Last Stand", "Proving Ground", "Battlezone"],
            "compound": ["Enemy HQ", "Research Facility", "Military Base"]
        }
    }

    # Pick a name based on genre and level type
```

```
        name_options = level_names.get(genre, {}).get(level_type, ["Level"])
        import random
        level_name = random.choice(name_options)

        # Layout is the basic structure of the level
        layout = template.get("layout", "linear")

        # Get appropriate environmental theme
        if genre == "platformer":
            themes = self.knowledge_base["environmental_themes"]["natural"] + self.knowledge_base["environmental_themes"]["fantasy"]
        elif genre == "rpg":
            themes = self.knowledge_base["environmental_themes"]["fantasy"] + self.knowledge_base["environmental_themes"]["historical"]
        elif genre == "shooter":
            themes = self.knowledge_base["environmental_themes"]["urban"] + self.knowledge_base["environmental_themes"]["sci_fi"]
        else:
            themes = [theme for theme_list in self.knowledge_base["environmental_themes"].values() for theme in theme_list]
        # Pick a random theme
        theme = random.choice(themes)

        # Determine objective type
        if level_type == "tutorial":
            objective_type = "reach_destination"
        elif level_type == "boss":
            objective_type = "defeat_enemies"
        else:
            objective_types = 
list(self.knowledge_base["objective_types"].keys())
            objective_type = random.choice(objective_types)

        # Get objective details
        objective_details = self.knowledge_base["objective_types"]
[objective_type]

        # Create structured level data
        level_data = {
            "id": level_id,
            "name": level_name,
            "type": level_type,
            "layout": {
                "structure": layout,
                "theme": theme,
                "size": template.get("length", "medium"),
                "rooms": self._generate_room_layout(layout,
template.get("length", "medium")),
                "connections": {}  # Would contain how rooms connect
            },
            "objectives": {
                "primary": {
                    "type": objective_type,
                    "description": objective_details["description"],
                    "elements": objective_details["elements"],
                    "location": "end_of_level"  # or specific room id
                },
                "secondary": []  # Would contain optional objectives
            },
            "encounters":
self._generate_encounters(template.get("encounters", []), genre),
            "environment": {
                "theme": theme,
                "mood": "neutral",  # or "tense", "peaceful", "mysterious",
etc.
```

```
                "lighting": "natural",  # or "dark", "moody", "bright", etc.
                "weather": "clear",  # or "rainy", "foggy", "snowy", etc.
                "time_of_day": "day"  # or "night", "dusk", "dawn", etc.
            },
            "gameplay_elements": {
                "obstacles": template.get("obstacles", []),
                "collectibles": self._generate_collectibles(genre, level_type),
                "interactive_objects": self._generate_interactive_objects(genre)
            },
            "technical_requirements": {
                "estimated_assets": self._estimate_asset_requirements(template, theme),
                "special_mechanics": [],  # Would list any level-specific mechanics
                "performance_considerations": []  # Would list any specific optimizations needed
            }
        }

        return level_data
```

```
    def _generate_room_layout(self, structure: str, size: str) -> List[Dict[str, Any]]:
        """Generate a list of rooms for the level based on structure and size."""
        # Determine number of rooms based on size
        if size == "short":
            room_count = 3
        elif size == "medium":
            room_count = 5
        elif size == "long":
            room_count = 8
        else:
            room_count = 5

        # Generate rooms
        rooms = []
        for i in range(room_count):
            # Determine room type based on position in level
            if i == 0:
                room_type = "entrance"
            elif i == room_count - 1:
                room_type = "exit"
            else:
                room_types = ["standard", "challenge", "reward", "rest"]
                import random
                room_type = random.choice(room_types)

            # Create room data
            room = {
                "id": f"room_{i+1}",
                "type": room_type,
                "size": "medium",  # or "small", "large"
                "description": f"A {room_type} room with standard
elements"
            }

            rooms.append(room)

        return rooms

    def _generate_encounters(self, encounter_types: List[str], genre:
str) -> List[Dict[str, Any]]:
        """Generate encounter definitions for the level."""
        encounters = []
        for encounter_type in encounter_types:
            # Create a basic encounter definition
            encounter = {
                "id": f"encounter_{len(encounters)+1}",
                "type": encounter_type,
                "difficulty": "medium",  # or "easy", "hard"
                "location": f"room_{len(encounters)+1}",  # Placeholder
                "description": f"A {encounter_type} encounter"
            }

            encounters.append(encounter)

        return encounters

    def _generate_collectibles(self, genre: str, level_type: str) -> List[Dict[str, Any]]:
        """Generate collectible items for the level."""
        collectibles = []

        # Determine collectible types based on genre
        if genre == "platformer":
```

```
        collectible_types = ["coin", "gem", "powerup", "extra_life"]
    elif genre == "rpg":
        collectible_types = ["gold", "potion", "equipment", "quest_item"]
    elif genre == "shooter":
        collectible_types = ["ammo", "health", "armor", "weapon"]
    else:
        collectible_types = ["generic_collectible"]

    # Generate a few collectibles
    import random
    count = random.randint(3, 8)

    for i in range(count):
        collectible_type = random.choice(collectible_types)

        collectible = {
            "id": f"collectible_{i+1}",
            "type": collectible_type,
            "value": "medium",  # or "low", "high"
            "placement": "exploration"  # or "reward", "hidden", "path"
        }
```

```
            collectibles.append(collectible)

        return collectibles

    def _generate_interactive_objects(self, genre: str) -> List[Dict[str, Any]]:
        """Generate interactive objects for the level."""
        interactive_objects = []

        # Determine object types based on genre
        if genre == "platformer":
            object_types = ["switch", "moving_platform", "door", "bounce_pad"]
        elif genre == "rpg":
            object_types = ["chest", "npc", "crafting_station", "door"]
        elif genre == "shooter":
            object_types = ["explosive_barrel", "cover", "terminal", "turret"]
        else:
            object_types = ["generic_interactive"]

        # Generate a few objects
        import random
        count = random.randint(2, 5)

        for i in range(count):
            object_type = random.choice(object_types)

            interactive_object = {
                "id": f"object_{i+1}",
                "type": object_type,
                "interaction": "simple",  # or "complex", "timed", "sequential"
                "purpose": "gameplay"  # or "narrative", "reward", "hint"
            }

            interactive_objects.append(interactive_object)

        return interactive_objects

    def _estimate_asset_requirements(self, template: Dict[str, Any], theme: str) -> Dict[str, int]:
        """Estimate the assets required for this level."""
        # This would be more sophisticated in a real implementation
        return {
            "environment_models": 15,
            "character_models": 5,
            "animations": 10,
            "textures": 20,
            "sound_effects": 10,
            "music_tracks": 1
        }

    def design_specific_level(self, task: Dict[str, Any]) -> Dict[str,
Any]:
        """Design a specific level based on provided requirements."""
        # Extract level requirements from task
        level_id = task.get("level_id", f"level_{uuid.uuid4()}")
        level_type = task.get("level_type", "standard")
        requirements = task.get("requirements", {})

        # Get the appropriate template
        genre = requirements.get("genre", "platformer")
        template = self.level_templates.get(genre, {}).get(level_type,
{})
```

```
        # Create the level design
        level_design = self._create_level_from_template(level_id, level_type, template, genre)

        # Apply specific requirements
        if "theme" in requirements:
            level_design["environment"]["theme"] = requirements["theme"]

        if "objective" in requirements:
            level_design["objectives"]["primary"]["type"] = requirements["objective"]

        # Register the level design as an artifact
        artifact_id = self.register_artifact(
            artifact_type="level_design",
            artifact_data=level_design,
            metadata={
                "description": f"Level Design: {level_design['name']}",
                "level_id": level_id,
                "level_type": level_type,
                "genre": genre
            }
        )

        return {
            "status": "completed",
            "artifact_id": artifact_id,
            "level_design": level_design
        }

    def modify_level_design(self, task: Dict[str, Any]) -> Dict[str, Any]:
        """Modify an existing level design based on feedback."""
        # Extract modification requirements
        level_id = task.get("level_id")
        modifications = task.get("modifications", {})

        if not level_id:
            return {"status": "failed", "message": "No level ID provided"}

        # Find the level design
        level_design = None
        artifact_id = None
        # Search through artifacts for the level
        for aid, artifact in self.artifacts.items():
            if isinstance(artifact, dict) and "id" in artifact and artifact["id"] == level_id:
                level_design = artifact
                artifact_id = aid
                break

        if not level_design:
            return {"status": "failed", "message": f"Level design not found: {level_id}"}

        # Apply modifications
        modified_design = level_design.copy()

        if "layout" in modifications:
            modified_design["layout"]["structure"] = modifications["layout"]

        if "theme" in modifications:
            modified_design["environment"]["theme"] = modifications["theme"]
        if "difficulty" in modifications:
            # Apply difficulty changes to encounters
            for encounter in modified_design["encounters"]:
                encounter["difficulty"] = modifications["difficulty"]

        if "objectives" in modifications:
            modified_design["objectives"]["primary"]["type"] =
modifications["objectives"]

        # Register the modified design as a new artifact
        new_artifact_id = self.register_artifact(
            artifact_type="level_design",
            artifact_data=modified_design,
            metadata={
                "description": f"Modified Level Design:
{modified_design['name']}",
                "original_artifact_id": artifact_id,
                "level_id": level_id,
                "modifications": list(modifications.keys())
            }
        )
        return {
            "status": "completed",
            "artifact_id": new_artifact_id,
            "level_design": modified_design,
            "modifications": list(modifications.keys())
        }

    def create_level_progression(self, task: Dict[str, Any]) -> Dict[str,
Any]:
        """Create a progression system for multiple levels."""
        # Extract requirements
        level_ids = task.get("level_ids", [])
        progression_type = task.get("progression_type", "linear")

        if not level_ids:
            return {"status": "failed", "message": "No level IDs provided"}

        # Create progression structure
        progression = {
            "type": progression_type,
            "start_level": level_ids[0],
            "level_order": level_ids
```

```
}

# Add branching paths for non-linear progression
if progression_type == "branching":
    progression["branches"] = {}
    # Would define optional or alternative paths

# Add level dependencies (unlock conditions)
progression["level_dependencies"] = {}
for i in range(1, len(level_ids)):
    progression["level_dependencies"][level_ids[i]] = {
        "required_levels": [level_ids[i-1]],
        "required_items": [],
        "required_achievements": []
    }

# Register the progression as an artifact
artifact_id = self.register_artifact(
    artifact_type="level_progression",
    artifact_data=progression,
    metadata={
```

```
                "description": "Level Progression System",
                "level_count": len(level_ids),
                "progression_type": progression_type
            }
        )

        return {
            "status": "completed",
            "artifact_id": artifact_id,
            "progression": progression
        }
```

Build Manager Agent -

```
// For a 4x4 puzzle, if the empty tile is on an even row from the bottom and
    // inversions is odd, or if the empty tile is on an odd row from the bottom and
    // inversions is even, the puzzle is unsolvable. In that case, swap the first two elements.
    const emptyTileRow = Math.floor(array.indexOf(15) / 4);
    const emptyTileRowFromBottom = 3 - emptyTileRow;

    if ((emptyTileRowFromBottom % 2 === 0 && inversions % 2 === 1) ||
        (emptyTileRowFromBottom % 2 === 1 && inversions % 2 === 0)) {
        [array[0], array[1]] = [array[1], array[0]];
    }
}

function selectTile(tile, scene) {
    if (gameComplete) return;

    const tileX = tile.currentIndex % 4;
    const tileY = Math.floor(tile.currentIndex / 4);

    // Check adjacent tiles to find the empty space
    const adjacentPositions = [
        { x: tileX, y: tileY - 1 }, // above
        { x: tileX, y: tileY + 1 }, // below
        { x: tileX - 1, y: tileY }, // left
        { x: tileX + 1, y: tileY }  // right
    ];

    let emptySpaceFound = false;
    for (const pos of adjacentPositions) {
        if (pos.x >= 0 && pos.x < 4 && pos.y >= 0 && pos.y < 4) {
            const adjacentIndex = pos.y * 4 + pos.x;
            const tileAtPosition = tiles.find(t => t.currentIndex === adjacentIndex);

            if (!tileAtPosition) {
                // This is the empty space, move the tile here
                const newX = 240 + pos.x * 80;
                const newY = 150 + pos.y * 80;

                scene.tweens.add({
                    targets: tile,
                    x: newX,
                    y: newY,
                    duration: 200,
                    ease: 'Power2',
                    onComplete: function() {
                        // Update tile position
                        const oldIndex = tile.currentIndex;
                        tile.currentIndex = adjacentIndex;
                        // Update moves counter
                        moves++;
                        movesText.setText(`Moves: ${moves}`);

                        // Check if puzzle is solved
                        checkPuzzleComplete(scene);
                    }
                });

                emptySpaceFound = true;
                break;
            }
        }
    }
}

function checkPuzzleComplete(scene) {
    // Check if all tiles are in correct position
    const complete = tiles.every(tile => {
        const correctX = tile.originalIndex % 4;
        const correctY = Math.floor(tile.originalIndex / 4);
        const correctIndex = correctY * 4 + correctX;
        return tile.currentIndex === correctIndex;
    });

    if (complete) {
        gameComplete = true;

        // Show completion message
        scene.add.text(400, 300, 'Puzzle Complete!', {
            fontSize: '40px',
            fill: '#fff',
            backgroundColor: '#000',
            padding: { x: 20, y: 10 }
        }).setOrigin(0.5).setDepth(10);

        // Show final moves
        scene.add.text(400, 350, `Completed in ${moves} moves`, {
            fontSize: '30px',
            fill: '#fff',
            backgroundColor: '#000',
            padding: { x: 20, y: 10 }
        }).setOrigin(0.5).setDepth(10);
    }
}
"""

    def _create_three_js_game(self, project_name: str) -> str:
        """Create a simple Three.js 3D game script."""
        return f"""// Game variables
let scene, camera, renderer;
let player, obstacles = [];
let score = 0;
let gameOver = false;
let moveSpeed = 0.1;
let level = 1;

// Initialize the game
function init() {{
    // Create scene
    scene = new THREE.Scene();
    scene.background = new THREE.Color(0x87CEEB);

    // Create camera
    camera = new THREE.PerspectiveCamera(75, window.innerWidth / window.innerHeight, 0.1, 1000);
    camera.position.z = 5;
    camera.position.y = 2;

    // Create renderer
    renderer = new THREE.WebGLRenderer({{ antialias: true }});
    renderer.setSize(window.innerWidth, window.innerHeight);
    document.getElementById('game').appendChild(renderer.domElement);

    // Add lights
    const ambientLight = new THREE.AmbientLight(0xffffff, 0.5);
    scene.add(ambientLight);

    const directionalLight = new THREE.DirectionalLight(0xffffff, 0.8);
    directionalLight.position.set(0, 10, 5);
    scene.add(directionalLight);

    // Create ground
    const groundGeometry = new THREE.PlaneGeometry(20, 100);
    const groundMaterial = new THREE.MeshStandardMaterial({{ color: 0x22CC22 }});
    const ground = new THREE.Mesh(groundGeometry,
groundMaterial);
    ground.rotation.x = -Math.PI / 2;
    ground.position.y = -1;
    ground.position.z = -25;
    scene.add(ground);

    // Create player
    const playerGeometry = new THREE.BoxGeometry(1, 1, 1);
    const playerMaterial = new
THREE.MeshStandardMaterial({{ color: 0x0000FF }});
    player = new THREE.Mesh(playerGeometry, playerMaterial);
    player.position.y = 0;
    scene.add(player);

    // Create HUD
    createHUD();

    // Create obstacles
    createObstacles();

    // Add event listeners
    window.addEventListener('keydown', handleKeyDown);
    window.addEventListener('resize', onWindowResize);


    // Start animation loop
    animate();


    // Show title
    showTitle('{project_name}');
}}

function createHUD() {{
    // Score display
    const scoreElement = document.createElement('div');
    scoreElement.id = 'score';
    scoreElement.style.position = 'absolute';
    scoreElement.style.top = '10px';
    scoreElement.style.left = '10px';
    scoreElement.style.color = 'white';
    scoreElement.style.fontSize = '24px';
    scoreElement.innerHTML = 'Score: 0';
    document.body.appendChild(scoreElement);
    // Level display
    const levelElement = document.createElement('div');
    levelElement.id = 'level';
    levelElement.style.position = 'absolute';
    levelElement.style.top = '10px';
    levelElement.style.right = '10px';
    levelElement.style.color = 'white';
    levelElement.style.fontSize = '24px';
    levelElement.innerHTML = 'Level: 1';
    document.body.appendChild(levelElement);
}}

function createObstacles() {{
    // Clear old obstacles
    for (let obstacle of obstacles) {{
        scene.remove(obstacle);
    }}
    obstacles = [];

    // Create new obstacles
    const obstacleCount = 5 + level * 2;
    const obstacleGeometry = new THREE.BoxGeometry(1.5, 1, 1);
    const obstacleMaterial = new THREE.MeshStandardMaterial({{ color: 0xFF0000 }});

    for (let i = 0; i < obstacleCount; i++) {{
        const obstacle = new THREE.Mesh(obstacleGeometry, obstacleMaterial);
        obstacle.position.z = -10 - i * 5;
        obstacle.position.x = Math.random() * 16 - 8; // Random position across width
        obstacle.rotation.y = Math.random() * Math.PI;
        scene.add(obstacle);
        obstacles.push(obstacle);
    }}
}}

function handleKeyDown(event) {{
    if (gameOver) return;

    // Player movement
    if (event.key === 'ArrowLeft' || event.key === 'a') {{
        player.position.x -= moveSpeed * 10;
    }} else if (event.key === 'ArrowRight' || event.key === 'd') {{
        player.position.x += moveSpeed * 10;
    }}

    // Clamp player position to road
    player.position.x = Math.max(-8, Math.min(8, player.position.x));
}}
function onWindowResize() {{
    camera.aspect = window.innerWidth / window.innerHeight;
    camera.updateProjectionMatrix();
    renderer.setSize(window.innerWidth, window.innerHeight);
}}
function showTitle(title) {{
    const titleElement = document.createElement('div');
    titleElement.style.position = 'absolute';
    titleElement.style.top = '100px';
    titleElement.style.width = '100%';
    titleElement.style.textAlign = 'center';
    titleElement.style.color = 'white';
    titleElement.style.fontSize = '48px';
    titleElement.style.fontWeight = 'bold';
    titleElement.style.textShadow = '2px 2px 4px #000000';
    titleElement.innerHTML = title;
    document.body.appendChild(titleElement);


    // Remove after 3 seconds
    setTimeout(() => {{
        document.body.removeChild(titleElement);
    }}, 3000);
}}

function showGameOver() {{
    const gameOverElement = document.createElement('div');
    gameOverElement.style.position = 'absolute';
    gameOverElement.style.top = '200px';
    gameOverElement.style.width = '100%';
    gameOverElement.style.textAlign = 'center';
    gameOverElement.style.color = 'white';
    gameOverElement.style.fontSize = '72px';
    gameOverElement.style.fontWeight = 'bold';
    gameOverElement.style.textShadow = '2px 2px 4px #000000';
    gameOverElement.innerHTML = 'GAME OVER';
    document.body.appendChild(gameOverElement);


    // Play again button
    const playAgainElement = document.createElement('div');
    playAgainElement.style.position = 'absolute';
    playAgainElement.style.top = '300px';
    playAgainElement.style.width = '100%';
    playAgainElement.style.textAlign = 'center';
    playAgainElement.style.color = 'white';
    playAgainElement.style.fontSize = '36px';
    playAgainElement.style.cursor = 'pointer';
    playAgainElement.innerHTML = 'Play Again';
    playAgainElement.onclick = resetGame;
    document.body.appendChild(playAgainElement);
}}

function resetGame() {{
    // Remove game over elements
    const gameOverElement = document.querySelector('div');
    if (gameOverElement)
document.body.removeChild(gameOverElement);
    const playAgainElement = document.querySelector('div');
    if (playAgainElement)
document.body.removeChild(playAgainElement);


    // Reset game state
    score = 0;

    level = 1;

    gameOver = false;

    moveSpeed = 0.1;

    player.position.set(0, 0, 0);



    // Update HUD

    document.getElementById('score').innerHTML = 'Score: 0';

    document.getElementById('level').innerHTML = 'Level: 1';



    // Create new obstacles

    createObstacles();

}}

function animate() {{

    if (!gameOver) {{

        requestAnimationFrame(animate);



        // Move obstacles

        for (let obstacle of obstacles) {{

            obstacle.position.z += moveSpeed;

            obstacle.rotation.y += 0.01;
        // If obstacle passed player
        if (obstacle.position.z > 5) {{
            // Reset obstacle position
            obstacle.position.z = -60;
            obstacle.position.x = Math.random() * 16 - 8;

            // Increase score
            score += 10;
            document.getElementById('score').innerHTML = `Score: ${score}`;

            // Level up every 100 points
            if (score > 0 && score % 100 === 0) {{
                level++;
                document.getElementById('level').innerHTML = `Level: ${level}`;
                moveSpeed += 0.02;
                showTitle(`Level ${level}`);
            }}
        }}
```

```
            // Check collision
            if (Math.abs(obstacle.position.x - player.position.x) < 1 &&
                Math.abs(obstacle.position.z - player.position.z) < 1) {{
                gameOver = true;
                showGameOver();
            }}
        }}

        // Render scene
        renderer.render(scene, camera);
    }}
}}
// Start the game
window.onload = init;
"""

    def _create_pixel_art_game_js(self, project_name: str) -> str:
        """Create a simple pixel art style game script."""
        return f"""// Pixel Art Game Engine
const TILE_SIZE = 16;
const CANVAS_WIDTH = 256;
const CANVAS_HEIGHT = 240;
const GRAVITY = 0.5;

// Game state
let canvas, ctx;
let player = {{
    x: 80,
    y: 200,
    width: TILE_SIZE,
    height: TILE_SIZE * 2,
    vx: 0,
    vy: 0,
    jumping: false,
    facing: 'right',
    frame: 0,
    frameTime: 0
}};

let platforms = [];
let collectibles = [];
let score = 0;
let gameOver = false;
let keys = {{}};
```

```
// Game assets (would be images in real implementation)
const assets = {{
    loaded: false,
    player: null,
    tiles: null,
    collectible: null,
    background: null
}};

// Initialize the game
function init() {{
    // Create canvas
    canvas = document.createElement('canvas');
    canvas.width = CANVAS_WIDTH;
    canvas.height = CANVAS_HEIGHT;
    canvas.style.width = '800px';
    canvas.style.height = '600px';
    canvas.style.imageRendering = 'pixelated';
    ctx = canvas.getContext('2d');
    document.getElementById('game').appendChild(canvas);

    // Load assets (in a real game, would load actual images)
    loadAssets();


    // Set up level
    createLevel();


    // Set up event listeners
    window.addEventListener('keydown', e => keys[e.key] = true);
    window.addEventListener('keyup', e => keys[e.key] = false);


    // Start game loop
    requestAnimationFrame(gameLoop);
}}

function loadAssets() {{
    // In a real game, would load images
    // For prototype, create placeholder colored rectangles


    // Create placeholder assets
    assets.player = {{ color: '#3498db' }};  // Blue
    assets.tiles = {{ color: '#27ae60' }};   // Green
    assets.collectible = {{ color: '#f1c40f' }}; // Yellow
    assets.background = {{ color: '#34495e' }}; // Dark blue
    assets.loaded = true;
}}

function createLevel() {{
    // Create ground
    platforms.push({{
        x: 0,
        y: 224,
        width: CANVAS_WIDTH,
        height: TILE_SIZE
    }});

    // Create platforms
    platforms.push({{
        x: 50,
        y: 180,
        width: TILE_SIZE * 4,
        height: TILE_SIZE
    }});

    platforms.push({{
```

```
    x: 150,

    y: 140,

    width: TILE_SIZE * 5,

    height: TILE_SIZE

}});


platforms.push({{

    x: 30,

    y: 100,

    width: TILE_SIZE * 3,

    height: TILE_SIZE

}});


// Create collectibles

for (let i = 0; i < 10; i++) {{

    collectibles.push({{

        x: 30 + i * 20,

        y: 80 - (i % 3) * 40,

        width: TILE_SIZE / 2,

        height: TILE_SIZE / 2,

        collected: false
        }});
    }}
}}

function update(deltaTime) {{
    if (gameOver) return;


    // Player movement
    const speed = 2;
    player.vx = 0;


    if (keys['ArrowLeft'] || keys['a']) {{
        player.vx = -speed;
        player.facing = 'left';
    }}
    if (keys['ArrowRight'] || keys['d']) {{
        player.vx = speed;
        player.facing = 'right';
    }}


    // Jumping
    if ((keys['ArrowUp'] || keys[' '] || keys['w']) && !player.jumping) {{
        player.vy = -10;
        player.jumping = true;
    }}

    // Apply gravity
    player.vy += GRAVITY;

    // Update position
    player.x += player.vx;
    player.y += player.vy;

    // Check platform collisions
    player.jumping = true;
    for (const platform of platforms) {{
        if (player.x < platform.x + platform.width &&
            player.x + player.width > platform.x &&
            player.y + player.height > platform.y &&
            player.y < platform.y + platform.height) {{

            // Landing on top of platform
            if (player.vy > 0 && player.y + player.height < platform.y + platform.height / 2) {{
                player.y = platform.y - player.height;
                player.vy = 0;
                player.jumping = false;
            }}
            // Hitting platform from below
            else if (player.vy < 0 && player.y > platform.y +
platform.height / 2) {{
                player.y = platform.y + platform.height;
                player.vy = 0;
            }}
            // Side collision
            else if (player.vx > 0) {{
                player.x = platform.x - player.width;
            }} else if (player.vx < 0) {{
                player.x = platform.x + platform.width;
            }}
        }}
    }}

    // Check collectible collisions
    for (const collectible of collectibles) {{
        if (!collectible.collected &&
            player.x < collectible.x + collectible.width &&
            player.x + player.width > collectible.x &&
            player.y < collectible.y + collectible.height &&
            player.y + player.height > collectible.y) {{

            collectible.collected = true;
            score += 10;

            // Check if all collectibles are collected
            if (collectibles.every(c => c.collected)) {{
                gameOver = true;
                setTimeout(() => {{
                    alert('You win! Score: ' + score);
                    resetGame();
                }}, 500);
            }}
        }}
    }}

    // Screen boundaries
    if (player.x < 0) player.x = 0;
```

```
    if (player.x + player.width > CANVAS_WIDTH) player.x =
CANVAS_WIDTH - player.width;


    // Check if player fell off the bottom

    if (player.y > CANVAS_HEIGHT) {{

        gameOver = true;

        setTimeout(() => {{

            alert('Game Over! Score: ' + score);

            resetGame();

        }}, 500);

    }}


    // Animation

    player.frameTime += deltaTime;

    if (player.frameTime > 100) {{  // Change frame every 100ms

        player.frameTime = 0;

        player.frame = (player.frame + 1) % 4;  // 4 frames of animation

    }}

}}

function render() {{

    // Clear canvas
    ctx.fillStyle = assets.background.color;
    ctx.fillRect(0, 0, CANVAS_WIDTH, CANVAS_HEIGHT);

    // Draw platforms
    ctx.fillStyle = assets.tiles.color;
    for (const platform of platforms) {{
        ctx.fillRect(platform.x, platform.y, platform.width, platform.height);
    }}

    // Draw collectibles
    ctx.fillStyle = assets.collectible.color;
    for (const collectible of collectibles) {{
        if (!collectible.collected) {{
            ctx.beginPath();
            ctx.arc(
                collectible.x + collectible.width / 2,
                collectible.y + collectible.height / 2,
                collectible.width / 2,
                0, Math.PI * 2
            );
            ctx.fill();
        }}
    }}

    // Draw player
    ctx.fillStyle = assets.player.color;
    ctx.fillRect(player.x, player.y, player.width, player.height);

    // Draw score
    ctx.fillStyle = '#FFFFFF';
    ctx.font = 'bold 16px monospace';
    ctx.fillText('SCORE: ' + score, 10, 20);
    ctx.fillText('{project_name}', CANVAS_WIDTH - 100, 20);
}}
function resetGame() {{
    player.x = 80;
    player.y = 200;
    player.vx = 0;
    player.vy = 0;
    player.jumping = false;

    score = 0;
    gameOver = false;


    collectibles.forEach(c => c.collected = false);
}}

function gameLoop(timestamp) {{
    static lastTime = 0;
    const deltaTime = timestamp - lastTime;
    lastTime = timestamp;


    update(deltaTime);
    render();


    requestAnimationFrame(gameLoop);
}}

// Start the game when page loads
window.onload = init;
"""


    def _create_placeholder_assets(self, build_dir: str, template_type: str) -> None:
        """Create placeholder assets for the game."""
        # In a real implementation, would copy actual assets
        # For prototype, create minimal SVG placeholders

        # Create placeholder images based on template type
        if template_type == "phaser_platformer":
            assets = {
                "sky.png": self._create_placeholder_svg(800, 600, "#87CEEB"),
                "platform.png": self._create_placeholder_svg(400, 32, "#8B4513"),
                "coin.png": self._create_placeholder_svg(32, 32, "#FFD700", "circle"),
                "player.png": self._create_placeholder_svg(288, 48, "#FF0000")  # 9x48 sprite sheet
            }
        elif template_type == "phaser_puzzle":
            assets = {
                "puzzle_bg.png": self._create_placeholder_svg(800, 600, "#2C3E50"),
                "puzzle_tiles.png": self._create_placeholder_svg(1920, 128, "#3498DB")  # 15x128 sprite sheet
            }
        elif template_type == "three_3d_game":
            # No image assets needed for this template as it uses 3D primitives
            assets = {}
        elif template_type == "pixel_art_game":
            # No separate assets needed for this template as it uses canvas drawing
            assets = {}

        # Write assets to files
        for filename, svg_content in assets.items():
            asset_path = os.path.join(build_dir, "assets/images", filename)
            with open(asset_path, "w") as f:
                f.write(svg_content)

    def _create_placeholder_svg(self, width: int, height: int, color: str, shape: str = "rect") -> str:
        """Create a placeholder SVG image."""
        if shape == "rect":
            svg = f"""<svg xmlns="http://www.w3.org/2000/svg" width="{width}" height="{height}" viewBox="0 0 {width} {height}">
    <rect width="{width}" height="{height}" fill="{color}" />
</svg>"""
        elif shape == "circle":
            radius = min(width, height) / 2
            cx, cy = width / 2, height / 2
            svg = f"""<svg xmlns="http://www.w3.org/2000/svg" width="{width}" height="{height}" viewBox="0 0 {width} {height}">
    <circle cx="{cx}" cy="{cy}" r="{radius}" fill="{color}" />
</svg>"""
        else:
            # Default to rectangle
            svg = f"""<svg xmlns="http://www.w3.org/2000/svg" width="{width}" height="{height}" viewBox="0 0 {width} {height}">
    <rect width="{width}" height="{height}" fill="{color}" />
</svg>"""

        return svg

    def _list_build_files(self, build_dir: str) -> List[str]:
        """List all files in the build directory."""
        file_list = []
        for root, dirs, files in os.walk(build_dir):
            for file in files:
                file_path = os.path.join(root, file)
                relative_path = os.path.relpath(file_path, build_dir)
                file_list.append(relative_path)
        return file_list

    def create_build_configuration(self, task: Dict[str, Any]) -> Dict[str, Any]:
        """Create a build configuration for the game."""
        # Extract requirements from task
        project_id = task.get("project_id")
        platforms = task.get("platforms", ["web"])

        if not project_id and self.project_manager:
            project_id = self.project_manager.project_data.get("project_id")

            if hasattr(self.project_manager, "project_data"):
                game_elements = self.project_manager.project_data.get("game_elements", {})
                platforms = game_elements.get("platforms", platforms)

        if not project_id:
            return {"status": "failed", "message": "No project ID available"}

        # Create configuration for web platform
        web_config = {
            "target_platform": "web",
            "game_engine": "phaser",  # Default engine
            "build_options": {
                "minify": True,
                "sourcemaps": False,
                "bundle_assets": True,
                "optimize_images": True
            },
            "deployment": {
                "platform": "local_server",  # or "github_pages", "netlify", etc.
                "options": {
                    "port": 8000,
                    "path": f"/games/{project_id}"
                }
            },
            "custom_settings": {}
        }

        # Determine appropriate game engine based on game genre
        if self.project_manager and hasattr(self.project_manager,
"project_data"):

            game_elements =
self.project_manager.project_data.get("game_elements", {})

            genre = game_elements.get("genre", "")


            if genre in ["fps", "3d", "shooter"]:

                web_config["game_engine"] = "three.js"

            elif genre in ["retro", "pixel"]:

                web_config["game_engine"] = "pixel_game_engine"

            elif genre in ["puzzle", "casual"]:

                web_config["game_engine"] = "phaser"

                web_config["custom_settings"]["physics"] = "arcade"


        # Store the configuration

        if project_id not in self.build_configs:

            self.build_configs[project_id] = {}


        self.build_configs[project_id]["web"] = web_config


        # Register the build configuration as an artifact

        artifact_id = self.register_artifact(
            artifact_type="build_configuration",
            artifact_data={
                "web": web_config,
                "project_id": project_id
            },
            metadata={
                "description": "Game Build Configuration",
                "project_id": project_id,
                "platforms": platforms
            }
        )

        return {
            "status": "completed",
            "artifact_id": artifact_id,
            "build_configs": {
                "web": web_config
            }
        }

    def deploy_game(self, task: Dict[str, Any]) -> Dict[str, Any]:
        """Deploy the game to a web server."""

        # Extract requirements from task

        project_id = task.get("project_id")

        build_id = task.get("build_id")

        deployment_platform = task.get("deployment_platform",
"local_server")



        if not project_id and self.project_manager:

            project_id =
self.project_manager.project_data.get("project_id")



        if not project_id:

            return {"status": "failed", "message": "No project ID
available"}



        # If no build ID provided, use the most recent build

        if not build_id and project_id in self.current_builds:

            build_id = self.current_builds[project_id]



        if not build_id:

            return {"status": "failed", "message": "No build ID available"}



        # Get the build directory
        build_dir = self._get_build_directory(project_id, build_id)

        if not os.path.exists(build_dir):
            return {"status": "failed", "message": f"Build directory not found: {build_dir}"}

        try:
            # Get deployment options
            if project_id in self.build_configs and "web" in self.build_configs[project_id]:
                deployment_config = self.build_configs[project_id]["web"]["deployment"]
            else:
                # Default deployment config
                deployment_config = {
                    "platform": deployment_platform,
                    "options": {
                        "port": 8000,
                        "path": f"/games/{project_id}"
                    }
                }

            # Perform deployment based on platform
            if deployment_platform == "local_server":
                # For prototype, just return the play URL
                play_url = self._get_play_url(project_id, build_id)

            # Store deployment URL
            self.deployment_urls[project_id] = play_url

            # Register the deployment as an artifact
            artifact_id = self.register_artifact(
                artifact_type="deployment",
                artifact_data={
                    "project_id": project_id,
                    "build_id": build_id,
                    "platform": deployment_platform,
                    "play_url": play_url,
                    "deployment_time":
datetime.datetime.now().isoformat()
                },
                metadata={
                    "description": f"Game Deployment
({deployment_platform})",
                    "project_id": project_id,
                    "build_id": build_id,
                    "platform": deployment_platform
                }
            )

            return {
                "status": "completed",
                "artifact_id": artifact_id,
                "play_url": play_url,
                "platform": deployment_platform
            }

        except Exception as e:
            return {
                "status": "failed",
                "message": f"Deployment failed: {str(e)}"
            }

def optimize_assets(self, task: Dict[str, Any]) -> Dict[str, Any]:
        """Optimize game assets for web deployment."""
        # In a real implementation, would compress and optimize
assets
```

```
        # For prototype purposes, just log what would be done


        project_id = task.get("project_id")

        build_id = task.get("build_id")


        if not project_id and self.project_manager:

            project_id =
self.project_manager.project_data.get("project_id")


        if not project_id:

            return {"status": "failed", "message": "No project ID
available"}


        # If no build ID provided, use the most recent build

        if not build_id and project_id in self.current_builds:

            build_id = self.current_builds[project_id]


        if not build_id:

            return {"status": "failed", "message": "No build ID available"}


        # Get the build directory

        build_dir = self._get_build_directory(project_id, build_id)
        if not os.path.exists(build_dir):
            return {"status": "failed", "message": f"Build directory not
found: {build_dir}"}

        # Log optimizations that would be performed
        optimizations = {
            "images": {
                "type": "compression",
                "files_processed": 0,
                "space_saved": "0 bytes"
            },
            "code": {
                "type": "minification",
                "files_processed": 0,
                "space_saved": "0 bytes"
            },
            "audio": {
                "type": "compression",
                "files_processed": 0,
                "space_saved": "0 bytes"
            }
```

```
}

# Count files that would be optimized
for root, dirs, files in os.walk(build_dir):
    for file in files:
        file_path = os.path.join(root, file)
        if file.endswith(('.png', '.jpg', '.jpeg', '.svg')):
            optimizations["images"]["files_processed"] += 1
        elif file.endswith(('.js')):
            optimizations["code"]["files_processed"] += 1
        elif file.endswith(('.mp3', '.ogg', '.wav')):
            optimizations["audio"]["files_processed"] += 1

# Register the optimization as an artifact
artifact_id = self.register_artifact(
    artifact_type="asset_optimization",
    artifact_data={
        "project_id": project_id,
        "build_id": build_id,
        "optimizations": optimizations,
        "optimization_time": datetime.datetime.now().isoformat()
```

```
            },
            metadata={
                "description": "Asset Optimization Report",
                "project_id": project_id,
                "build_id": build_id
            }
        )

        return {
            "status": "completed",
            "artifact_id": artifact_id,
            "optimizations": optimizations
        }

    def get_latest_build_status(self, project_id: str) -> Dict[str, Any]:
        """Get the status of the latest build for a project."""
        if not project_id:
            return {"status": "unknown", "message": "No project ID provided"}

        if project_id in self.current_builds:
            build_id = self.current_builds[project_id]
            if build_id in self.build_history:
                return self.build_history[build_id]

        return {"status": "not_found", "message": "No builds found for this project"}

    def get_play_url(self, project_id: str) -> Optional[str]:
        """Get the URL where the game can be played."""
        if project_id in self.deployment_urls:
            return self.deployment_urls[project_id]

        if project_id in self.current_builds:
            build_id = self.current_builds[project_id]
            if build_id in self.build_history:
                return self.build_history[build_id].get("play_url")

        return Noned, build_id)
            elif deployment_platform == "github_pages":
                # In a real implementation, would push to GitHub Pages
                play_url = f"https://username.github.io/game-dev-agent/games/{project_id}"
            elif deployment_platform == "netlify":
                # In a real implementation, would deploy to Netlify
                play_url = f"https://{project_id}.netlify.app"
            elif deployment_platform == "vercel":
                # In a real implementation, would deploy to Vercel
                play_url = f"https://{project_id}.vercel.app"
            else:
                # Default to local server
                play_url = self._get_play_url(project_i#!/usr/bin/env python3
"""
Build Manager Agent - Responsible for packaging and deploying the game as a browser-playable version.
"""

import os
import uuid
import datetime
import json
import shutil
import subprocess
import threading
from typing import Dict, List, Any, Optional

from base_agent import BaseAgent
```

```
class BuildManagerAgent(BaseAgent):
    """
    Agent responsible for packaging and deploying the game in a browser-playable format.
    Handles build configuration, asset bundling, and web deployment.
    """

    def __init__(self, project_manager=None):
        super().__init__("build_manager", project_manager)
        self.build_configs = {}
        self.build_history = {}
        self.current_builds = {}
        self.deployment_urls = {}
        self.load_knowledge_base()

    def load_knowledge_base(self) -> None:
        """Load build management knowledge base."""
        # This would load from an actual knowledge base file/database
        self.knowledge_base = {
            "web_build_templates": {
                "phaser": {
```

```
            "description": "HTML5 game framework for Canvas and WebGL",
            "suitable_for": ["2D", "casual", "platformer", "puzzle"],
            "files": [
                "index.html",
                "js/phaser.min.js",
                "js/game.js",
                "css/style.css",
                "assets/sprites.png",
                "assets/audio.mp3"
            ],
            "config_options": {
                "render_mode": ["Canvas", "WebGL", "Auto"],
                "scale_mode": ["Fit", "Resize", "None"],
                "pixel_art": [True, False]
            }
        },
        "three_js": {
            "description": "JavaScript 3D library for WebGL",
            "suitable_for": ["3D", "simulation", "first-person", "immersive"],
            "files": [
                "index.html",
                "js/three.min.js",
                "js/game.js",
                "css/style.css",
                "assets/models",
                "assets/textures",
                "assets/audio"
            ],
            "config_options": {
                "render_mode": ["WebGL", "WebGL2"],
                "shadows": [True, False],
                "anti_aliasing": [True, False],
                "physics_engine": ["None", "Cannon", "Ammo"]
            }
        },
        "pixel_game_engine": {
            "description": "Simple HTML5 game engine for retro-style games",
            "suitable_for": ["pixel art", "retro", "2D", "simple"],
            "files": [
                "index.html",
                "js/engine.js",
                "js/game.js",
                "css/style.css",
                "assets/sprites.png",
                "assets/audio.mp3"
            ],
            "config_options": {
                "tile_size": [8, 16, 32],
                "palette": ["default", "gameboy", "cpc", "ega"],
                "fixed_resolution": [True, False]
            }
        }
    },
    "deployment_platforms": {
        "github_pages": {
            "description": "Free static site hosting from GitHub",
            "setup_steps": [
                "Create gh-pages branch",
                "Push build files to branch",
                "Configure GitHub repository settings"
            ],
            "url_pattern": "https://{username}.github.io/{repository}"
        },
        "netlify": {
          "description": "Continuous deployment platform with free tier",
          "setup_steps": [
            "Create Netlify account",
            "Connect to repository",
            "Configure build settings",
            "Deploy site"
          ],
          "url_pattern": "https://{project-name}.netlify.app"
        },
        "vercel": {
          "description": "Platform for static sites and Serverless Functions",
          "setup_steps": [
            "Create Vercel account",
            "Connect to repository",
            "Configure project settings",
            "Deploy"
          ],
          "url_pattern": "https://{project-name}.vercel.app"
```

```
            },
            "local_server": {
                "description": "Deploy to local web server for testing",
                "setup_steps": [
                    "Copy build files to server directory",
                    "Configure server settings",
                    "Start server"
                ],
                "url_pattern": "http://localhost:{port}"
            }
        },
        "asset_optimizations": {
            "image": {
                "sprite_sheet": "Combine multiple images into a single sprite sheet",
                "compression": "Reduce file size with optimized compression",
                "rescale": "Reduce dimensions to appropriate resolution",
                "format_conversion": "Convert to web-optimized formats like WebP"
            },
            "audio": {
                "compression": "Use appropriate audio compression",
                "format_conversion": "Convert to web-optimized
formats like MP3 or OGG",
                "streaming": "Set up audio streaming for large files"
            },
            "code": {
                "minification": "Remove whitespace and shorten
variable names",
                "bundling": "Combine multiple JS files",
                "tree_shaking": "Remove unused code",
                "lazy_loading": "Load code only when needed"
            }
        },
        "web_templates": {
            "phaser_platformer": "phaser_platformer_template/",
            "phaser_puzzle": "phaser_puzzle_template/",
            "three_3d_game": "three_3d_game_template/",
            "pixel_art_game": "pixel_art_template/"
        }
    }

def handle_message(self, message: Dict[str, Any]) -> None:
    """Handle incoming messages."""
    message_type = message.get("type", "")
    content = message.get("content", {})

    if message_type == "request_build":
        # A request to build the game
        self._handle_build_request(content)
    elif message_type == "build_status_request":
        # A request for build status
        self._handle_build_status_request(content)
    elif message_type == "artifact_update":
        # Another agent has created/updated an artifact we should know about
        self.review_artifact(content.get("artifact_id"), content.get("artifact_type"))

def _handle_build_request(self, content: Dict[str, Any]) -> None:
    """Handle a request to build the game."""
    project_id = content.get("project_id")
    build_type = content.get("build_type", "web")
    requester = content.get("from_agent", "unknown")
    if not project_id:
        self.send_message(
            to_agent=requester,
            message_type="build_response",
            content={
                "status": "failed",
                "message": "No project ID provided",
                "request_id": content.get("request_id")
            }
        )
        return

    # Start build process in a separate thread
    build_thread = threading.Thread(
        target=self._process_build_request,
        args=(project_id, build_type, requester,
content.get("request_id"))
    )
    build_thread.daemon = True
    build_thread.start()

    # Respond immediately that build is in progress
    self.send_message(
        to_agent=requester,
        message_type="build_response",
        content={
            "status": "in_progress",
            "message": f"Build of type '{build_type}' started for project {project_id}",
            "request_id": content.get("request_id")
        }
    )

def _process_build_request(self, project_id: str, build_type: str, requester: str, request_id: str) -> None:
    """Process a build request in a separate thread."""
    try:
        # Create a build
        build_result = self.create_web_build({"project_id": project_id, "build_type": build_type})

        # Send response when complete
        if build_result.get("status") == "completed":
            self.send_message(
                to_agent=requester,
                message_type="build_response",
                content={
                    "status": "completed",
                    "message": "Build completed successfully",
                    "build_id": build_result.get("build_id"),
                    "play_url": build_result.get("play_url"),
                    "request_id": request_id
                }
            )
        else:
            self.send_message(
                to_agent=requester,
                message_type="build_response",
                content={
                    "status": "failed",
                    "message": build_result.get("message", "Build failed"),
                    "request_id": request_id
                }
            )
    except Exception as e:
        # Send error response
        self.send_message(
            to_agent=requester,
            message_type="build_response",
            content={
                "status": "failed",
                "message": f"Build error: {str(e)}",
                "request_id": request_id
            }
        )

def _handle_build_status_request(self, content: Dict[str, Any]) -> None:
    """Handle a request for build status."""
    project_id = content.get("project_id")
    build_id = content.get("build_id")
    requester = content.get("from_agent", "unknown")

    if not project_id and not build_id:
        self.send_message(
            to_agent=requester,
            message_type="build_status_response",
            content={
```

```
            "status": "failed",
            "message": "No project ID or build ID provided",
            "request_id": content.get("request_id")
        }
    )
    return

# Get build status
if build_id and build_id in self.build_history:
    status = self.build_history[build_id]
else:
    # Get latest build for the project
    status = self.get_latest_build_status(project_id)

self.send_message(
    to_agent=requester,
    message_type="build_status_response",
    content={
        "status": status.get("status", "unknown"),
        "build_id": status.get("build_id"),
        "play_url": status.get("play_url"),
                "build_time": status.get("build_time"),
                "request_id": content.get("request_id")
            }
        )

    def review_artifact(self, artifact_id: str, artifact_type: str) -> None:
        """Review an artifact created by another agent."""
        if not self.project_manager:
            return

        artifact = self.request_artifact(artifact_id)
        if not artifact:
            return

        # Take action based on artifact type
        if artifact_type == "code_implementation":
            # Store code implementations for build
            data = artifact.get("data", {})
            project_id =
self.project_manager.project_data.get("project_id")

            if project_id:
```

```
        if project_id not in self.build_configs:
            self.build_configs[project_id] = {}

        self.build_configs[project_id]["code"] = data

    def process_task(self, task: Dict[str, Any]) -> Dict[str, Any]:
        """Process a build task."""
        task_name = task.get("name", "")

        if task_name == "Build web playable version":
            return self.create_web_build(task)
        elif task_name == "Create build configuration":
            return self.create_build_configuration(task)
        elif task_name == "Deploy game":
            return self.deploy_game(task)
        elif task_name == "Optimize assets":
            return self.optimize_assets(task)

        return {"status": "failed", "message": "Unknown task type"}

    def create_web_build(self, task: Dict[str, Any]) -> Dict[str, Any]:
        """Create a web-playable build of the game."""
        # In a real implementation, this would compile and package the game
        # For prototype purposes, create a simple web template

        project_id = task.get("project_id")
        if not project_id and self.project_manager:
            project_id = self.project_manager.project_data.get("project_id")

        if not project_id:
            return {"status": "failed", "message": "No project ID available"}

        # Generate a build ID
        build_id = str(uuid.uuid4())
        timestamp = datetime.datetime.now().isoformat()

        # Get project info
        project_name = "Untitled Game"
        game_genre = "casual"
        template_type = "phaser_platformer"
        if self.project_manager:
            if hasattr(self.project_manager, "project_data"):
                project_name = self.project_manager.project_data.get("project_name", project_name)
                game_elements = self.project_manager.project_data.get("game_elements", {})
                game_genre = game_elements.get("genre", game_genre)

        # Determine appropriate template based on genre
        if game_genre == "platformer":
            template_type = "phaser_platformer"
        elif game_genre == "puzzle":
            template_type = "phaser_puzzle"
        elif game_genre == "shooter" or game_genre == "fps":
            template_type = "three_3d_game"
        elif game_genre == "retro" or game_genre == "pixel":
            template_type = "pixel_art_game"

        # Create the build directory
        build_dir = self._get_build_directory(project_id, build_id)
        os.makedirs(build_dir, exist_ok=True)
        try:
            # Use template to create web game
            self._create_game_from_template(template_type, build_dir,
project_name, project_id)


            # Record build status
            play_url = self._get_play_url(project_id, build_id)


            build_status = {
                "build_id": build_id,
                "project_id": project_id,
                "status": "completed",
                "build_time": timestamp,
                "play_url": play_url,
                "template_used": template_type,
                "notified": False
            }


            self.build_history[build_id] = build_status
            self.current_builds[project_id] = build_id


            # Register the build as an artifact
```

```
artifact_id = self.register_artifact(
    artifact_type="web_build",
    artifact_data={
        "build_id": build_id,
        "project_id": project_id,
        "template_used": template_type,
        "play_url": play_url,
        "files": self._list_build_files(build_dir)
    },
    metadata={
        "description": f"Web Playable Build ({template_type})",
        "project_id": project_id,
        "build_id": build_id,
        "template": template_type
    }
)

return {
    "status": "completed",
    "build_id": build_id,
    "play_url": play_url,
            "artifact_id": artifact_id
        }

    except Exception as e:
        # Record failure
        build_status = {
            "build_id": build_id,
            "project_id": project_id,
            "status": "failed",
            "build_time": timestamp,
            "error": str(e)
        }

        self.build_history[build_id] = build_status

        return {
            "status": "failed",
            "message": f"Build failed: {str(e)}",
            "build_id": build_id
        }
    def _get_build_directory(self, project_id: str, build_id: str) -> str:

        """Get the directory path for a build."""

        # In a real system, this would use configured paths

        base_dir =
os.environ.get("WEB_DEPLOYMENT_DIRECTORY", "./
web_games")

        project_dir = os.path.join(base_dir, project_id)

        build_dir = os.path.join(project_dir, build_id)

        return build_dir



    def _get_play_url(self, project_id: str, build_id: str) -> str:

        """Get the URL where the game can be played."""

        # In a real system, this would use the actual deployment URL

        base_url = os.environ.get("GAME_SERVER_URL", "http://
localhost:8000")

        play_url = f"{base_url}/games/{project_id}/{build_id}/"

        return play_url



    def _create_game_from_template(self, template_type: str,
build_dir: str, project_name: str, project_id: str) -> None:

        """Create a playable web game using a template."""

        # In a real system, this would copy a template and inject
game-specific code

        # For prototype, we'll create some minimal files
        # Create index.html
        index_html = f"""<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>{project_name}</title>
    <link rel="stylesheet" href="css/style.css">
</head>
<body>
    <div id="game-container">
        <div id="game"></div>
    </div>

    <!-- Scripts -->
    <script src="js/phaser.min.js"></script>
    <script src="js/game.js"></script>
</body>
</html>
"""
```

```
}

        # Create CSS
        css = """body {
    margin: 0;
    padding: 0;
    background-color: #333;
    display: flex;
    justify-content: center;
    align-items: center;
    height: 100vh;
    font-family: Arial, sans-serif;
}

#game-container {
    width: 800px;
    height: 600px;
    box-shadow: 0 0 10px rgba(0,0,0,0.5);
}

canvas {
    display: block;
}
"""
```

```
        # Create game.js based on template type
        if template_type == "phaser_platformer":
            game_js = self._create_phaser_platformer_js(project_name)
        elif template_type == "phaser_puzzle":
            game_js = self._create_phaser_puzzle_js(project_name)
        elif template_type == "three_3d_game":
            game_js = self._create_three_js_game(project_name)
        elif template_type == "pixel_art_game":
            game_js = self._create_pixel_art_game_js(project_name)
        else:
            game_js =
self._create_phaser_platformer_js(project_name)  # Default

        # Create directories
        os.makedirs(os.path.join(build_dir, "css"), exist_ok=True)
        os.makedirs(os.path.join(build_dir, "js"), exist_ok=True)
        os.makedirs(os.path.join(build_dir, "assets"), exist_ok=True)
        os.makedirs(os.path.join(build_dir, "assets/images"),
exist_ok=True)
        os.makedirs(os.path.join(build_dir, "assets/audio"),
exist_ok=True)
```

```
        # Write files
        with open(os.path.join(build_dir, "index.html"), "w") as f:
            f.write(index_html)

        with open(os.path.join(build_dir, "css/style.css"), "w") as f:
            f.write(css)

        with open(os.path.join(build_dir, "js/game.js"), "w") as f:
            f.write(game_js)

        # Download or copy Phaser library
        phaser_path = os.path.join(build_dir, "js/phaser.min.js")
        if not os.path.exists(phaser_path):
            # In a real system, would download or copy from local resources
            # For prototype, create a minimal placeholder
            with open(phaser_path, "w") as f:
                f.write("// Phaser library would be here in a real system\n")

        # Create placeholder assets
        self._create_placeholder_assets(build_dir, template_type)
```

```
    def _create_phaser_platformer_js(self, project_name: str) -> str:
        """Create a simple Phaser platformer game script."""
        return f"""// Game configuration
const config = {{
    type: Phaser.AUTO,
    width: 800,
    height: 600,
    physics: {{
        default: 'arcade',
        arcade: {{
            gravity: {{ y: 300 }},
            debug: false
        }}
    }},
    scene: {{
        preload: preload,
        create: create,
        update: update
    }}
}};
```

```
// Initialize the game
const game = new Phaser.Game(config);

// Game variables
let player;
let platforms;
let coins;
let score = 0;
let scoreText;
let gameOver = false;

function preload() {{
    // Load assets
    this.load.image('sky', 'assets/images/sky.png');
    this.load.image('ground', 'assets/images/platform.png');
    this.load.image('coin', 'assets/images/coin.png');
    this.load.spritesheet('player', 'assets/images/player.png',
{{ frameWidth: 32, frameHeight: 48 }});
}}

function create() {{
    // Create the sky background
    this.add.image(400, 300, 'sky');
    // Create platforms group
    platforms = this.physics.add.staticGroup();

    // Create the ground
    platforms.create(400, 568, 'ground').setScale(2).refreshBody();

    // Create some platforms
    platforms.create(600, 400, 'ground');
    platforms.create(50, 250, 'ground');
    platforms.create(750, 220, 'ground');

    // Create player
    player = this.physics.add.sprite(100, 450, 'player');
    player.setBounce(0.2);
    player.setCollideWorldBounds(true);

    // Player animations
    this.anims.create({{
        key: 'left',
        frames: this.anims.generateFrameNumbers('player', {{ start: 0, end: 3 }}),
        frameRate: 10,
        repeat: -1
    }});
    this.anims.create({{
        key: 'turn',
        frames: [{{ key: 'player', frame: 4 }}],
        frameRate: 20
    }});

    this.anims.create({{
        key: 'right',
        frames: this.anims.generateFrameNumbers('player', {{ start: 5, end: 8 }}),
        frameRate: 10,
        repeat: -1
    }});

    // Add collision between player and platforms
    this.physics.add.collider(player, platforms);

    // Create coins
    coins = this.physics.add.group({{
        key: 'coin',
        repeat: 11,
        setXY: {{ x: 12, y: 0, stepX: 70 }}
    }});
    coins.children.iterate(function (child) {{
        child.setBounceY(Phaser.Math.FloatBetween(0.4, 0.8));
    }});

    this.physics.add.collider(coins, platforms);
    this.physics.add.overlap(player, coins, collectCoin, null, this);

    // Score text
    scoreText = this.add.text(16, 16, 'Score: 0', {{ fontSize: '32px', fill: '#000' }});

    // Create cursor keys for input
    cursors = this.input.keyboard.createCursorKeys();
}}

function update() {{
    if (gameOver) {{
        return;
    }}

    // Player movement
    if (cursors.left.isDown) {{
        player.setVelocityX(-160);
        player.anims.play('left', true);
    }} else if (cursors.right.isDown) {{
        player.setVelocityX(160);
        player.anims.play('right', true);
    }} else {{
        player.setVelocityX(0);
        player.anims.play('turn');
    }}

    // Player jump
    if (cursors.up.isDown && player.body.touching.down) {{
        player.setVelocityY(-330);
    }}
}}

function collectCoin(player, coin) {{
    coin.disableBody(true, true);


    // Update score
    score += 10;
    scoreText.setText('Score: ' + score);


    // Check if all coins are collected
    if (coins.countActive(true) === 0) {{
        // Respawn all coins
        coins.children.iterate(function (child) {{
            child.enableBody(true, child.x, 0, true, true);
        }});

        // Create a new platform
        const x = (player.x < 400) ? Phaser.Math.Between(400, 800) :
Phaser.Math.Between(0, 400);
        const platform = platforms.create(x, 150, 'ground');
        platform.setScale(0.5).refreshBody();
        platform.setImmovable(true);
    }}
}}
"""

    def _create_phaser_puzzle_js(self, project_name: str) -> str:
        """Create a simple Phaser puzzle game script."""
        return f"""// Game configuration
const config = {{
    type: Phaser.AUTO,
    width: 800,
    height: 600,
    scene: {{
        preload: preload,
        create: create,
        update: update
    }}
}};

// Initialize the game
const game = new Phaser.Game(config);

// Game variables
let tiles = [];
let selectedTile = null;
let moves = 0;
let movesText;
let gameComplete = false;

function preload() {{
    // Load assets
    this.load.image('background', 'assets/images/puzzle_bg.png');
    this.load.spritesheet('tiles', 'assets/images/puzzle_tiles.png',
{{ frameWidth: 128, frameHeight: 128 }});
}}
```

```
function create() {{
    // Add background
    this.add.image(400, 300, 'background');

    // Title
    this.add.text(400, 50, '{project_name}', {{
        fontSize: '32px',
        fill: '#fff',
        fontStyle: 'bold'
    }}).setOrigin(0.5);

    // Create puzzle grid
    createPuzzle(this);

    // Moves counter
    movesText = this.add.text(400, 550, 'Moves: 0', {{
        fontSize: '24px',
        fill: '#fff'
    }}).setOrigin(0.5);

    // Instructions
    this.add.text(400, 500, 'Click tiles to swap and solve the puzzle',
{{
        fontSize: '18px',
        fill: '#ccc'
    }}).setOrigin(0.5);
}}

function update() {{
    // Game update logic
}}

function createPuzzle(scene) {{
    // Create a 4x4 grid of tiles
    let positions = [];
    for (let i = 0; i < 16; i++) {{
        positions.push(i);
    }}

    // Shuffle positions (ensure solvable)
    shuffleArray(positions);

    // Create tiles
    let xOffset = 240;
    let yOffset = 150;
    let tileSize = 80;

    for (let i = 0; i < 16; i++) {{
        let tileIndex = positions[i];
        if (tileIndex < 15) {{ // Skip the last tile to create the empty
space
            const x = xOffset + (i % 4) * tileSize;
            const y = yOffset + Math.floor(i / 4) * tileSize;

            let tile = scene.add.sprite(x, y, 'tiles', tileIndex);
            tile.displayWidth = tileSize;
            tile.displayHeight = tileSize;
            tile.originalIndex = tileIndex;
            tile.currentIndex = i;

            // Make tile interactive
            tile.setInteractive();
            tile.on('pointerdown', function() {{
                selectTile(this, scene);
            }});
            tiles.push(tile);
        }}
    }}
}}

function shuffleArray(array) {{
    // Fisher-Yates shuffle
    for (let i = array.length - 1; i > 0; i--) {{
        const j = Math.floor(Math.random() * (i + 1));
        [array[i], array[j]] = [array[j], array[i]];
    }}

    // Ensure the puzzle is solvable
    let inversions = 0;
    for (let i = 0; i < array.length; i++) {{
        for (let j = i + 1; j < array.length; j++) {{
            if (array[i] > array[j] && array[i] !== 15 && array[j] !== 15) {{
                inversions++;
            }}
        }}
    }}
    // For a 4x4 puzzle, if the empty tile is on an even row from the bottom and
    // inversions is odd, or if the empty tile is on an odd row from the bottom and
```

Web Server for playable games

```
#!/usr/bin/env python3
"""
Simple HTTP server for hosting the browser-playable games created by the agent system.
This server handles game deployment and provides access to playable games.
"""

import os
import json
import http.server
import socketserver
import urllib.parse
from typing import Dict, Any, Optional
import argparse

# Configuration
DEFAULT_PORT = 8000
DEFAULT_GAMES_DIR = "./web_games"
```

```
class
GameServerHandler(http.server.SimpleHTTPRequestHandler):
    """Custom HTTP request handler for the game server."""


    # Override the default directory
    def __init__(self, *args, **kwargs):
        self.games_directory = kwargs.pop('games_directory',
DEFAULT_GAMES_DIR)
        super().__init__(*args, directory=self.games_directory,
**kwargs)


    def do_GET(self):
        """Handle GET requests."""
        # Parse the URL path
        parsed_path = urllib.parse.urlparse(self.path)
        path = parsed_path.path


        # Root path - show the game list
        if path == '/' or path == '/index.html':
            self.send_response(200)
            self.send_header('Content-type', 'text/html')
            self.end_headers()
            self.wfile.write(self.generate_game_list_html().encode())
            return

        # Game listing as JSON
        elif path == '/api/games':
            self.send_response(200)
            self.send_header('Content-type', 'application/json')
            self.end_headers()
            self.wfile.write(json.dumps(self.get_available_games()).encode())
            return

        # Specific game API
        elif path.startswith('/api/games/'):
            parts = path.split('/')
            if len(parts) >= 4:
                project_id = parts[3]
                game_info = self.get_game_info(project_id)

                if game_info:
                    self.send_response(200)
                    self.send_header('Content-type', 'application/json')
                    self.end_headers()
                    self.wfile.write(json.dumps(game_info).encode())
                else:
                    self.send_error(404, f"Game {project_id} not found")
                return

        # Handle static files and game files
        try:
            super().do_GET()
        except FileNotFoundError:
            self.send_error(404, f"File not found")

    def get_available_games(self) -> Dict[str, Any]:
        """Get a list of all available games."""
        games = {}

        # Get all project folders
        try:
            project_dirs = [d for d in os.listdir(self.games_directory)
                          if os.path.isdir(os.path.join(self.games_directory, d))]
            for project_id in project_dirs:
                game_info = self.get_game_info(project_id)
                if game_info:
                    games[project_id] = game_info
        except FileNotFoundError:
            # Games directory doesn't exist yet
            pass

        return games

    def get_game_info(self, project_id: str) -> Optional[Dict[str, Any]]:
        """Get information about a specific game."""
        project_dir = os.path.join(self.games_directory, project_id)

        if not os.path.isdir(project_dir):
            return None

        # Get the latest build (assume subdirectories are builds)
        build_dirs = [d for d in os.listdir(project_dir)
                    if os.path.isdir(os.path.join(project_dir, d))]
        if not build_dirs:
            return None

        # Sort by creation time, newest first
        build_dirs.sort(key=lambda d: os.path.getctime(os.path.join(project_dir, d)), reverse=True)
        latest_build = build_dirs[0]

        # Check if the build has an index.html
        index_path = os.path.join(project_dir, latest_build, 'index.html')
        if not os.path.exists(index_path):
            return None

        # Check for game.json if it exists (would contain metadata)
        game_json_path = os.path.join(project_dir, latest_build, 'game.json')
        metadata = {}

        if os.path.exists(game_json_path):
            try:
                with open(game_json_path, 'r') as f:
                    metadata = json.load(f)
            except:
                pass

        # Get basic information
        title = metadata.get('title', f"Game {project_id}")
        description = metadata.get('description', "A game created by the AI agent system")

        # Create paths
        play_url = f"/games/{project_id}/{latest_build}/"
        thumbnail_url = f"/games/{project_id}/{latest_build}/thumbnail.png"

        # Check if thumbnail exists, if not, use a default
        thumbnail_path = os.path.join(project_dir, latest_build, 'thumbnail.png')
        if not os.path.exists(thumbnail_path):
            thumbnail_url = "/static/default-thumbnail.png"

        return {
            "project_id": project_id,
            "build_id": latest_build,
            "title": title,
            "description": description,

            "play_url": play_url,

            "thumbnail_url": thumbnail_url,

            "creation_time": os.path.getctime(os.path.join(project_dir,
latest_build)),

            "metadata": metadata

        }


    def generate_game_list_html(self) -> str:

        """Generate HTML for the game list page."""

        games = self.get_available_games()


        html = """<!DOCTYPE html>

<html lang="en">

<head>

    <meta charset="UTF-8">

    <meta name="viewport" content="width=device-width, initial-
scale=1.0">

    <title>AI-Generated Games</title>

    <style>

        body {

            font-family: Arial, sans-serif;
            max-width: 1200px;
            margin: 0 auto;
            padding: 20px;
            background-color: #f5f5f5;
        }
        h1 {
            color: #333;
            text-align: center;
            margin-bottom: 30px;
        }
        .games-container {
            display: grid;
            grid-template-columns: repeat(auto-fill, minmax(300px, 1fr));
            gap: 20px;
        }
        .game-card {
            background: white;
            border-radius: 8px;
            overflow: hidden;
            box-shadow: 0 4px 8px rgba(0,0,0,0.1);
            transition: transform 0.3s ease;
```

```
}
.game-card:hover {
   transform: translateY(-5px);
}
.game-thumbnail {
   width: 100%;
   height: 180px;
   object-fit: cover;
}
.game-info {
   padding: 15px;
}
.game-title {
   font-size: 18px;
   font-weight: bold;
   margin: 0 0 10px 0;
}
.game-description {
   color: #666;
   font-size: 14px;
   margin-bottom: 15px;
```

```
    height: 60px;
    overflow: hidden;
}
.play-button {
    display: block;
    background: #4CAF50;
    color: white;
    text-align: center;
    padding: 10px;
    text-decoration: none;
    border-radius: 4px;
    font-weight: bold;
}
.no-games {
    text-align: center;
    grid-column: 1 / -1;
    padding: 50px;
    background: white;
    border-radius: 8px;
}
footer {
```

```
            margin-top: 40px;
            text-align: center;
            color: #666;
        }
    </style>
</head>
<body>
    <h1>AI-Generated Games</h1>

    <div class="games-container">
"""


        if not games:
            html += """
        <div class="no-games">
            <h2>No games available yet</h2>
            <p>Games created by the AI agent system will appear here once they're ready.</p>
        </div>
"""
        else:
            for project_id, game in games.items():
                html += f"""
        <div class="game-card">
            <img src="{game['thumbnail_url']}" alt="{game['title']}" class="game-thumbnail">
            <div class="game-info">
                <h2 class="game-title">{game['title']}</h2>
                <p class="game-description">{game['description']}</p>
                <a href="{game['play_url']}" class="play-button">Play Game</a>
            </div>
        </div>
"""

        html += """
    </div>

    <footer>
        <p>Games created by the AI Game Development Agent System</p>
    </footer>
</body>
</html>
"""
```

```
        return html

def create_static_directory(games_dir: str) -> None:
    """Create the static directory and add default assets."""
    static_dir = os.path.join(games_dir, "static")
    os.makedirs(static_dir, exist_ok=True)

    # Create a default thumbnail
    default_thumbnail_path = os.path.join(static_dir, "default-thumbnail.png")

    if not os.path.exists(default_thumbnail_path):
        # Create a simple SVG as a default thumbnail
        svg = """<svg xmlns="http://www.w3.org/2000/svg" width="300" height="180" viewBox="0 0 300 180">
  <rect width="300" height="180" fill="#2C3E50" />
  <text x="150" y="90" font-family="Arial" font-size="24" fill="white" text-anchor="middle">Game Preview</text>
  <rect x="120" y="110" width="60" height="20" fill="#3498DB" />
  <circle cx="90" cy="120" r="15" fill="#E74C3C" />
  <circle cx="210" cy="120" r="15" fill="#2ECC71" />
</svg>"""
```

```
        with open(default_thumbnail_path, 'w') as f:
            f.write(svg)

def main():
    """Main entry point for the game server."""
    parser = argparse.ArgumentParser(description='Simple HTTP server for AI-generated games')
    parser.add_argument('--port', type=int, default=DEFAULT_PORT, help='Port to run the server on')
    parser.add_argument('--games-dir', type=str, default=DEFAULT_GAMES_DIR, help='Directory containing the games')
    args = parser.parse_args()

    # Create games directory if it doesn't exist
    os.makedirs(args.games_dir, exist_ok=True)

    # Create static directory for assets
    create_static_directory(args.games_dir)

    # Custom handler with games directory
    handler = lambda *args, **kwargs: GameServerHandler(*args, games_directory=args.games_dir, **kwargs)
    # Create and start the server

    with socketserver.TCPServer(("", args.port), handler) as httpd:

        print(f"Game server running at http://localhost:{args.port}")

        print(f"Games directory: {args.games_dir}")

        httpd.serve_forever()

if __name__ == "__main__":

    main()
```

Asset Manager Agent Implementaion

```
#!/usr/bin/env python3

"""

Asset Manager Agent - Responsible for organizing, cataloging, and optimizing game assets.

"""

import os

import uuid

import datetime

import json

import shutil

from typing import Dict, List, Any, Optional
from base_agent import BaseAgent

class AssetManagerAgent(BaseAgent):
    """
    Agent responsible for organizing, cataloging, and optimizing game assets.
    Manages sprites, textures, audio, and other game assets.
    """


    def __init__(self, project_manager=None):
        super().__init__("asset_manager", project_manager)
        self.asset_catalog = {}
        self.asset_storage = {}
        self.asset_dependencies = {}
        self.load_knowledge_base()


    def load_knowledge_base(self) -> None:
        """Load asset management knowledge base."""
        # This would load from an actual knowledge base file/database
        self.knowledge_base = {
            "asset_types": {
                "sprite": {
```

```
          "description": "2D graphics for characters, objects, and UI elements",
          "file_formats": [".png", ".jpg", ".svg"],
          "optimization_techniques": [
            "sprite_sheet",
            "texture_atlas",
            "compression",
            "rescale"
          ]
        },
        "texture": {
          "description": "Image files applied to 3D models",
          "file_formats": [".png", ".jpg", ".tga"],
          "optimization_techniques": [
            "compression",
            "mipmapping",
            "texture_atlasing"
          ]
        },
        "audio": {
          "description": "Sound effects and music",
          "file_formats": [".mp3", ".wav", ".ogg"],
            "optimization_techniques": [
                "compression",
                "streaming",
                "format_conversion"
            ]
        },
        "font": {
            "description": "Typefaces for text rendering",
            "file_formats": [".ttf", ".otf", ".woff"],
            "optimization_techniques": [
                "subsetting",
                "bitmap_fonts"
            ]
        },
        "model": {
            "description": "3D models for characters and environments",
            "file_formats": [".gltf", ".obj", ".fbx"],
            "optimization_techniques": [
                "mesh_simplification",
                "level_of_detail",
                "texture_compression"
        ]
      },
      "data": {
        "description": "Game configuration and data files",
        "file_formats": [".json", ".xml", ".csv"],
        "optimization_techniques": [
          "compression",
          "minification"
        ]
      }
    },
    "asset_pipelines": {
      "sprite_processing": [
        "crop_and_clean",
        "create_sprite_sheet",
        "optimize_for_web"
      ],
      "audio_processing": [
        "normalize_volume",
        "compress_to_appropriate_format",
        "create_variations"
```

```
            ],
            "texture_processing": [
                "resize_to_power_of_two",
                "generate_mipmaps",
                "compress_for_target_platform"
            ]
        },
        "naming_conventions": {
            "sprite": "{category}_{name}_{state}_{frame}.{ext}",
            "texture": "{category}_{material}_{resolution}.{ext}",
            "audio": "{category}_{name}_{variation}.{ext}",
            "font": "{family}_{weight}_{style}.{ext}",
            "model": "{category}_{name}_{version}.{ext}",
            "data": "{category}_{name}_{version}.{ext}"
        }
    }

def handle_message(self, message: Dict[str, Any]) -> None:
    """Handle incoming messages."""
    message_type = message.get("type", "")
    content = message.get("content", {})
```

```
        if message_type == "request_asset":
            # A request for an asset from another agent
            self._handle_asset_request(message["from_agent"],
content)
        elif message_type == "register_asset":
            # A request to register a new asset
            self._handle_asset_registration(message["from_agent"],
content)
        elif message_type == "asset_status_request":
            # A request for asset status
            self._handle_asset_status_request(message["from_agent"],
content)
        elif message_type == "artifact_update":
            # Another agent has created/updated an artifact we should
know about
            self.review_artifact(content.get("artifact_id"),
content.get("artifact_type"))


    def _handle_asset_request(self, requester: str, content: Dict[str,
Any]) -> None:
        """Handle a request for an asset."""
        asset_id = content.get("asset_id")
        asset_type = content.get("asset_type")
        asset_name = content.get("asset_name")
        request_id = content.get("request_id", "")

        # Find the requested asset
        asset_data = None

        if asset_id and asset_id in self.asset_catalog:
            asset_data = self.asset_catalog[asset_id]
        elif asset_name and asset_type:
            # Search by name and type
            for aid, asset in self.asset_catalog.items():
                if asset.get("name") == asset_name and asset.get("type") == asset_type:
                    asset_data = asset
                    asset_id = aid
                    break

        if asset_data:
            # Send the asset data to the requester
            self.send_message(
                to_agent=requester,
                message_type="asset_response",
                content={
                    "asset_id": asset_id,
                    "asset_data": asset_data,
                    "status": "found",
                    "request_id": request_id
                }
            )
        else:
            # Asset not found
            self.send_message(
                to_agent=requester,
                message_type="asset_response",
                content={
                    "status": "not_found",
                    "message": f"Asset not found with the provided
parameters",
                    "request_id": request_id
                }
            )


    def _handle_asset_registration(self, requester: str, content:
Dict[str, Any]) -> None:
        """Handle a request to register a new asset."""
        asset_type = content.get("asset_type")
        asset_name = content.get("asset_name")
        asset_data = content.get("asset_data")
        metadata = content.get("metadata", {})
        request_id = content.get("request_id", "")

        if not asset_type or not asset_name or not asset_data:
            # Missing required fields
            self.send_message(
                to_agent=requester,
                message_type="asset_registration_response",
                content={
                    "status": "failed",
                    "message": "Missing required fields: asset_type,
asset_name, or asset_data",
                    "request_id": request_id
                }
            )
            return

        # Generate an asset ID
```

```
asset_id = str(uuid.uuid4())

# Add to catalog
self.asset_catalog[asset_id] = {
    "type": asset_type,
    "name": asset_name,
    "data": asset_data,
    "metadata": metadata,
    "registration_time": datetime.datetime.now().isoformat(),
    "registered_by": requester
}

# Send confirmation
self.send_message(
    to_agent=requester,
    message_type="asset_registration_response",
    content={
        "status": "success",
        "asset_id": asset_id,
        "message": f"Asset registered with ID: {asset_id}",
        "request_id": request_id
```

```
            }
        )

    def _handle_asset_status_request(self, requester: str, content:
Dict[str, Any]) -> None:
        """Handle a request for asset status."""
        project_id = content.get("project_id")
        asset_types = content.get("asset_types", [])
        request_id = content.get("request_id", "")

        # Filter assets by project and types
        filtered_assets = {}

        for asset_id, asset in self.asset_catalog.items():
            if project_id and asset.get("metadata", {}).get("project_id") !=
project_id:
                continue

            if asset_types and asset.get("type") not in asset_types:
                continue

            filtered_assets[asset_id] = {
```

```
            "type": asset.get("type"),
            "name": asset.get("name"),
            "metadata": asset.get("metadata"),
            "registration_time": asset.get("registration_time")
        }

    # Send status response
    self.send_message(
        to_agent=requester,
        message_type="asset_status_response",
        content={
            "asset_count": len(filtered_assets),
            "assets": filtered_assets,
            "request_id": request_id
        }
    )

def review_artifact(self, artifact_id: str, artifact_type: str) -> None:
    """Review an artifact created by another agent."""
    if not self.project_manager:
        return
    artifact = self.request_artifact(artifact_id)
    if not artifact:
        return

    # Take action based on artifact type
    if artifact_type == "concept_art":
        # Extract assets from concept art
        self._process_concept_art_artifact(artifact)
    elif artifact_type == "level_designs":
        # Extract asset requirements from level designs
        self._process_level_designs_artifact(artifact)

def _process_concept_art_artifact(self, artifact: Dict[str, Any]) -> None:
    """Process concept art artifact to extract potential game assets."""
    data = artifact.get("data", {})
    project_id = self.project_manager.project_data.get("project_id")

    for key, concept in data.items():
        # For each concept art piece, create asset entry
        asset_name = f"concept_{key}"
        asset_id = str(uuid.uuid4())

        # Store in catalog
        self.asset_catalog[asset_id] = {
            "type": "concept",
            "name": asset_name,
            "data": concept.get("description", ""),
            "metadata": {
                "project_id": project_id,
                "description": concept.get("description", ""),
                "style": concept.get("style", ""),
                "color_palette": concept.get("color_palette", [])
            },
            "registration_time": datetime.datetime.now().isoformat(),
            "registered_by": "asset_manager"
        }

def _process_level_designs_artifact(self, artifact: Dict[str, Any]) -> None:
    """Process level designs artifact to extract asset requirements."""
    data = artifact.get("data", {})
    project_id =
self.project_manager.project_data.get("project_id")

    levels = data.get("levels", {})
    asset_requirements = {}

    for level_id, level in levels.items():
        # Extract asset requirements from level design
        environment = level.get("environment", {})
        theme = environment.get("theme", "unknown")

        if theme not in asset_requirements:
            asset_requirements[theme] = {
                "environment": 0,
                "props": 0,
                "characters": 0,
                "audio": 0
            }

        # Increment counts based on level needs
        asset_requirements[theme]["environment"] += 1
        # Count props from gameplay elements
        gameplay_elements = level.get("gameplay_elements", {})
        props = gameplay_elements.get("obstacles", []) +
gameplay_elements.get("interactive_objects", [])
        asset_requirements[theme]["props"] += len(props)

        # Count character needs from encounters
        encounters = level.get("encounters", [])
        asset_requirements[theme]["characters"] += len(encounters)

        # Assume each level needs at least 2 audio assets
        asset_requirements[theme]["audio"] += 2

    # Store asset requirements
    self.asset_dependencies[project_id] = asset_requirements

def process_task(self, task: Dict[str, Any]) -> Dict[str, Any]:
    """Process an asset management task."""
    task_name = task.get("name", "")

    if task_name == "Catalog project assets":
        return self.catalog_project_assets(task)
    elif task_name == "Generate asset placeholders":
        return self.generate_asset_placeholders(task)
    elif task_name == "Organize asset structure":
        return self.organize_asset_structure(task)
    elif task_name == "Optimize assets for web":
        return self.optimize_assets_for_web(task)

    return {"status": "failed", "message": "Unknown task type"}

def catalog_project_assets
```