# Design Overview for SoNeat

Name: Minh An Nguyen

Student ID: 104844794

# Summary of Program

# "SoNeat" is an endless runner game featuring Sonic, where players can either control the character or train their own AI using the NEAT algorithm. The game allows players to observe the AI's learning process in real-time as it navigates obstacles. It provides a unique opportunity to see how a genetic algorithm evolves and adapts, making complex AI concepts more accessible.

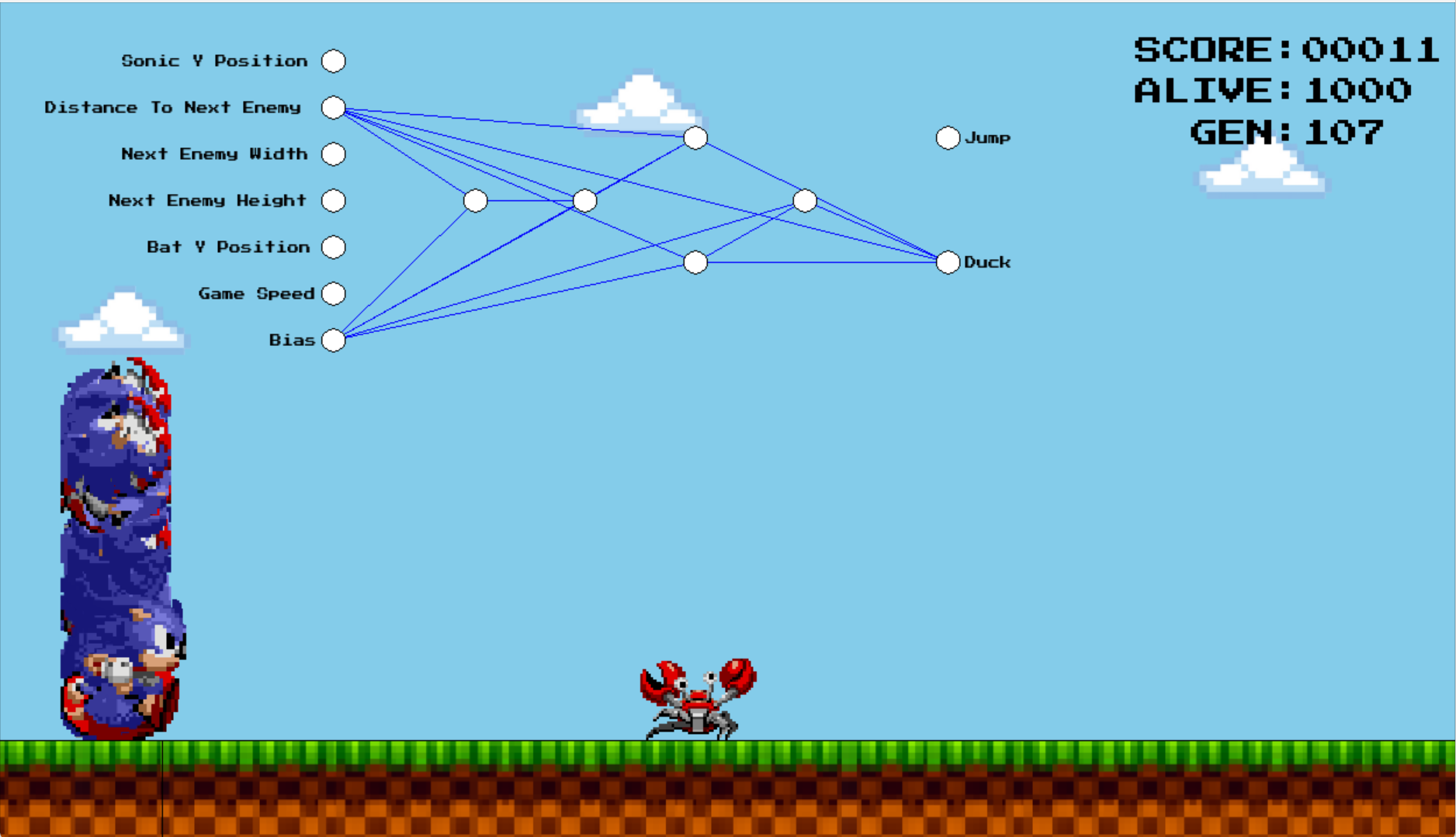
# This game can be a valuable educational tool, demonstrating the learning process of genetic algorithms in an interactive way. By visualizing how the AI improves over time, players can better understand how genetic algorithms work, making it an engaging way to teach these concepts in classrooms or workshops. This hands-on experience can help learners grasp how AI adapts and optimizes solutions, similar to applications in fields like robotics and autonomous vehicles.

# A screen shot of a video game Description automatically generated

A screen shot of a video game

Description automatically generated

# A screen shot of a video game Description automatically generated



# Required Roles

Table 1: Game Logic Classes

|  |  |  |
| --- | --- | --- |
| Class | Type Details | Notes |
| GameObject <<abstract>> | Abstract, Position, Speed, Collision, Update | The GameObject class serves as a base class for all objects with positions, movement, and interactions in the game environment. |
| Sonic | Character, Movement, Handle User input | Represents the player character, managing its actions, interactions in the game. Inherit from GameObject |
| Obstacle <<abstract>> | Defines game barriers, controls interactions and movement. | The Obstacle class is an abstract base inherit from GameObject for specific obstacles, defining shared properties and behaviors for different game obstacles. |
| Ground | Creates continuous ground movement under the player. | Inherits from GameObject; concrete class for visual ground elements. Manages ground scrolling effect. |
| Bat | Flying obstacle in the game. | Inherits from Obstacle; concrete class for specific bat behavior. |
| Cloud | Background scenery element. | Inherits from GameObject; concrete class for background visuals. |
| Crab | An obstacle in the game | Inherits from Obstacle; concrete class for specific Crab behavior. |
| Hog | An obstacle in the game | Inherits from Obstacle; concrete class for specific Hog behavior. |
| ObstacleManager | Manages obstacle creation and updates. Spawns, updates, and removes obstacles. | Concrete class; controls obstacle lifecycle and interactions. |
| ObstacleFactory | Generates obstacle instances dynamically | Static class; centralizes obstacle instantiation logic. |
| EnvironmentManager | Manages background elements (ground, clouds). Updates and renders background scenery. | Concrete class; controls visual background elements' behavior. |

Table 2: UI Classes

|  |  |  |
| --- | --- | --- |
| Class | Type Details | Notes |
| IScreenState | Interface for screen states | Defines methods for managing screen transitions. |
| MainMenuState | Handles menu navigation and state transitions. | Implements IScreenState; concrete class for main menu logic. |
| GameScreenState | Handles game flow, player actions, and updates | Implements IScreenState; concrete class for gameplay management. |

Table 2: AI Classes

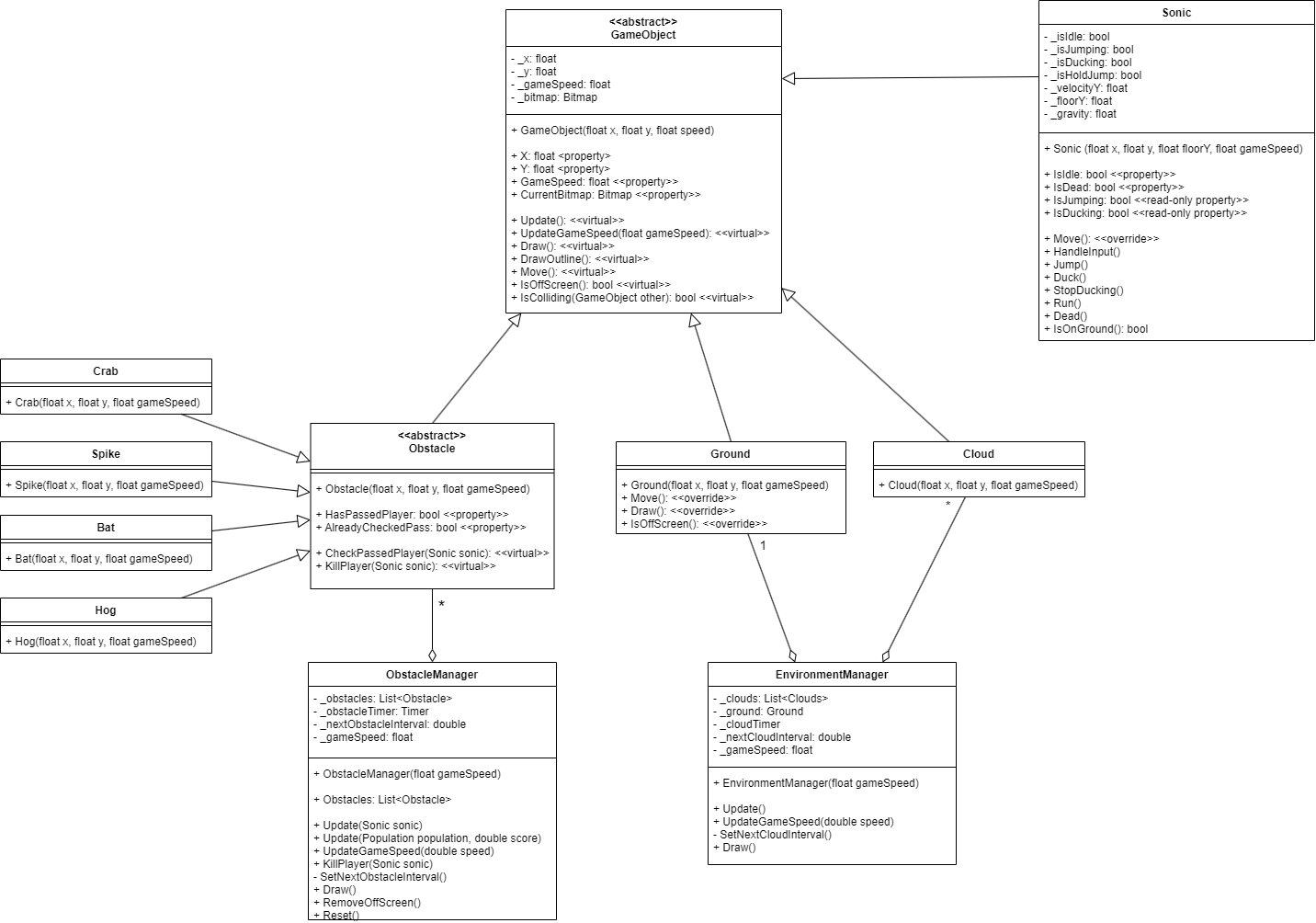
|  |  |  |
| --- | --- | --- |
| Class | Type Details | Notes |
| Node | Represents a neural network node | Concrete class; fundamental unit in the neural network structure. Processes input and generates output through connections. |
| Connection | Links nodes with weighted connections. | Concrete class; essential for network connectivity and weight mutation. Transmits values between nodes with adjustable weights. |
| Genome | Encodes neural network structure | Concrete class; blueprint for creating and evolving neural networks. Manages nodes, connections, and mutations. |
| Agent | Represents an AI entity | Holds a genome and interacts with the environment. |
| Species | Groups similar agents. | Manages species evolution and fitness. Maintains and evolves groups of similar genomes. |
| Neat | Controls population, evolution, and species | Central class for implementing the NEAT evolutionary process. |

Table 4: ObstacleType details

|  |  |
| --- | --- |
| Value | Notes |
| Bat |  |
| Crab |  |
| Hog |  |
| Spike |  |

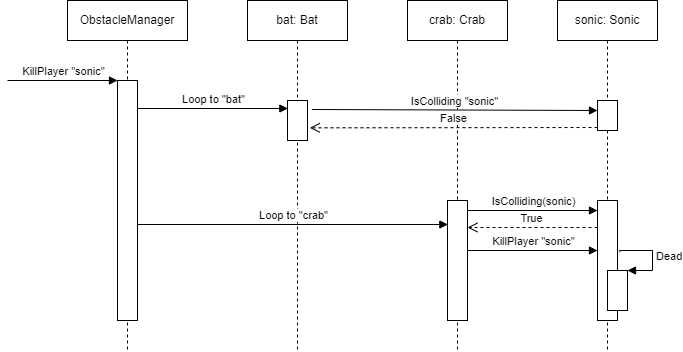
# Class Diagram

At the current process, I have only done the basic diagram for the Game Logic.



# Sequence Diagram

For each update, all the obstacles will execute KillPlayer to check for collision with the player and kill the player if it collides. For example, the list of current obstacles can be a bat and a crab.



# The use of Abstraction

In my program, I use abstraction to manage complexity by creating simplified representations of complex systems. For example, I define an abstract class like Obstacle to encapsulate shared properties and behaviors for all obstacles in my game. Specific obstacles like Bat and Crab inherit from this class, allowing me to implement their unique characteristics without altering the core structure. This approach lets me focus on high-level game mechanics while keeping the code organized and easier to maintain.

# The use of inheritance and Polymorphism

In my game, I use **inheritance** by creating an abstract Obstacle class that serves as a blueprint for various obstacles, defining common properties like position, speed, and collision detection methods. Subclasses such as Bat, Crab, and Spike inherit from Obstacle and provide their own specific implementations, like unique movement patterns. **Polymorphism** allows me to handle these different obstacles uniformly; for example, I can store all types of obstacles in a single list and call the Update method on each one, and the appropriate overridden method is executed based on the actual obstacle type, whether it’s a Bat flying or a Crab moving on the ground.