```
    var player *Player
    var position rl.Vector2
    var size float32

    // Extract position and size based on player type
    switch p := playerObj.(type) {
if index < 0 || index >= len(am.AIs) {
    return
}

// Free the name for reuse
am.UsedNames[am.AIs[index].Personality.Name] = false

// Remove AI (swap with last element for efficiency)
am.AIs[index] = am.AIs[len(am.AIs)-1]
am.AIs = am.AIs[:len(am.AIs)-1]
am.ActiveAIs--

// Reorganize update groups
am.reorganizeUpdateGroups()
```

}

```
// Draw all AI helicopters
func (am *AIManager) Draw(camera rl.Camera2D) {
// Compute visible range
viewportBounds := rl.Rectangle{
X:      camera.Target.X - constants.ScreenWidth/2/camera.Zoom,
Y:      camera.Target.Y - constants.ScreenHeight/2/camera.Zoom,
Width:  constants.ScreenWidth/camera.Zoom,
Height: constants.ScreenHeight/camera.Zoom,
}
```

```go
    // Add padding for items just offscreen
    padding := float32(100)
    viewportWithPadding := rl.Rectangle{
        X:      viewportBounds.X - padding,
        Y:      viewportBounds.Y - padding,
        Width:  viewportBounds.Width + padding*2,
        Height: viewportBounds.Height + padding*2,
    }

    // Draw only AIs in viewport
    for _, ai := range am.AIs {
        // Get AI bounds
        aiBounds := ai.GetBounds()

        // Check if AI is in view
        if util.CheckRectangleOverlap(aiBounds, viewportWithPadding) {
            ai.Draw()

            // Draw AI name if debug mode is on
            if ai.DebugMode {
                rl.DrawText(
                    ai.Personality.Name,
                    int32(ai.Position.X - 20),
                    int32(ai.Position.Y - ai.Size - 10),
                    10,
                    rl.White,
                )
            }
        }
    }
```

}

// GetAIsByScore returns AIs ordered by score (for leaderboard)
func (am *AIManager) GetAIsByScore() []*entities.AIHelicopter {
// Create a copy to sort
aisCopy := make([]*entities.AIHelicopter, len(am.AIs))
copy(aisCopy, am.AIs)

```go
    // Sort by score (descending)
    util.SortByScore(aisCopy)

    return aisCopy
```

}

// GetBestPerformingAIs returns top N AIs by score
func (am *AIManager) GetBestPerformingAIs(count int) []*entities.AIHelicopter {
sorted := am.GetAIsByScore()

```
if count > len(sorted) {
    count = len(sorted)
}

return sorted[:count]
```

}

// GetAICount returns the number of active AIs
func (am *AIManager) GetAICount() int {
return am.ActiveAIs
}

// GetAverageAISize returns the average size of all AIs
func (am *AIManager) GetAverageAISize() float32 {
if am.ActiveAIs == 0 {
return 0
}

```
totalSize := float32(0)
for _, ai := range am.AIs {
    totalSize += ai.Size
}

return totalSize / float32(am.ActiveAIs)
```

}

7. Dynamic Difficulty Adjustment

Let's add a system that adjusts AI behavior based on player performance:

```go
// systems/difficulty_manager.go
package systems

import (
    "atomblaster/entities"
    "math"
)

// DifficultyLevel represents different game difficulty settings
type DifficultyLevel int

const (
    DifficultyEasy DifficultyLevel = iota
    DifficultyMedium
    DifficultyHard
    DifficultyDynamic // Auto-adjusts based on player performance
)

// DifficultyManager adjusts AI behavior based on player performance
type DifficultyManager struct {
    CurrentLevel      DifficultyLevel
    PlayerPerformance float32 // 0.0 to 1.0, how well player is doing

    // Parameters that change with difficulty
    AIAggressionScale  float32
    AIIntelligenceScale float32
    AIPowerUpPriority  float32
    AIReactionTime     float32

    // Dynamic difficulty adjustment
    DDAEnabled       bool
    LastAdjustmentTime float32
    AdjustmentInterval float32
    PlayerScoreHistory []int
    AIScoreHistory    []int

    // Performance tracking
    PlayerDeathCount  int
    PlayerKillCount   int
    PlayerScoreRate   float32 // Points per minute
```

```go
    LastScore         int
    TimeElapsed       float32
}

// NewDifficultyManager creates a new difficulty management system
func NewDifficultyManager(initialLevel DifficultyLevel) *DifficultyManager {
    dm := &DifficultyManager{
        CurrentLevel:      initialLevel,
        PlayerPerformance: 0.5, // Start at medium
        AIAggressionScale: 1.0,
        AIIntelligenceScale: 1.0,
        AIPowerUpPriority: 1.0,
        AIReactionTime:    1.0,
        DDAEnabled:        initialLevel == DifficultyDynamic,
        LastAdjustmentTime: 0,
        AdjustmentInterval: 30.0, // Check every 30 seconds
        PlayerScoreHistory: make([]int, 0),
        AIScoreHistory:    make([]int, 0),
        PlayerDeathCount:  0,
        PlayerKillCount:   0,
        PlayerScoreRate:   0,
        LastScore:         0,
        TimeElapsed:       0,
    }

    // Apply initial difficulty settings
    dm.ApplyDifficultySettings()

    return dm
}

// ApplyDifficultySettings updates parameters based on current difficulty
func (dm *DifficultyManager) ApplyDifficultySettings() {
    switch dm.CurrentLevel {
    case DifficultyEasy:
        dm.AIAggressionScale = 0.7
        dm.AIIntelligenceScale = 0.7
        dm.AIPowerUpPriority = 0.7
        dm.AIReactionTime = 1.3 // Slower reactions

    case DifficultyMedium:
        dm.AIAggressionScale = 1.0
        dm.AIIntelligenceScale = 1.0
        dm.AIPowerUpPriority = 1.0
        dm.AIReactionTime = 1.0

    case DifficultyHard:
```

```
        dm.AIAggressionScale = 1.3
        dm.AIIntelligenceScale = 1.3
        dm.AIPowerUpPriority = 1.3
        dm.AIReactionTime = 0.7 // Faster reactions

    case DifficultyDynamic:
        // Scale based on player performance
        // 0.0 = very easy, 1.0 = very hard
        dm.AIAggressionScale = 0.7 + dm.PlayerPerformance * 0.6
        dm.AIIntelligenceScale = 0.7 + dm.PlayerPerformance * 0.6
        dm.AIPowerUpPriority = 0.7 + dm.PlayerPerformance * 0.6
        dm.AIReactionTime = 1.3 - dm.PlayerPerformance * 0.6
    }
}

// Update adjusts difficulty based on player performance
func (dm *DifficultyManager) Update(dt float32, player *entities.Player, aiManager
*AIManager) {
    dm.TimeElapsed += dt

    // Track player score rate
    currentScore := player.Score
    scoreDelta := currentScore - dm.LastScore
    dm.LastScore = currentScore

    // Calculate score per minute (sliding window)
    if dm.TimeElapsed > 0 {
        dm.PlayerScoreRate = dm.PlayerScoreRate*0.95 + float32(scoreDelta)/(dt/60)*0.05
    }

    // Check if it's time to adjust difficulty
    if dm.DDAEnabled && dm.TimeElapsed - dm.LastAdjustmentTime >= dm.AdjustmentInterval {
        dm.LastAdjustmentTime = dm.TimeElapsed

        // Store current scores
        dm.PlayerScoreHistory = append(dm.PlayerScoreHistory, player.Score)

        // Get highest AI score
        highestAIScore := 0
        for _, ai := range aiManager.GetBestPerformingAIs(1) {
            highestAIScore = ai.Score
            break
        }
        dm.AIScoreHistory = append(dm.AIScoreHistory, highestAIScore)

        // Calculate performance metrics
        dm.updatePlayerPerformance(player, aiManager)
```

```go
        // Apply new difficulty settings
        dm.ApplyDifficultySettings()

        // Apply settings to all AIs
        dm.updateAIParameters(aiManager)
    }
}

// Update player performance rating
func (dm *DifficultyManager) updatePlayerPerformance(player *entities.Player, aiManager
*AIManager) {
    // Get current metrics
    playerRank := 1

    // Find player's rank among AIs
    ais := aiManager.GetAIsByScore()
    for i, ai := range ais {
        if player.Score < ai.Score {
            playerRank++
        }
    }

    // Calculate performance factors
    rankFactor := float32(1.0 - float64(playerRank-1)/math.Max(1.0, float64(len(ais))))
    sizeFactor := player.Size / dm.GetAverageTopAISize(aiManager)
    scoreFactor := float32(1.0)

    if len(dm.AIScoreHistory) > 0 {
        bestAIScore := dm.AIScoreHistory[len(dm.AIScoreHistory)-1]
        if bestAIScore > 0 {
            scoreFactor = float32(player.Score) / float32(bestAIScore)
            if scoreFactor > 2.0 {
                scoreFactor = 2.0
            }
        }
    }

    // Combine factors
    newPerformance := (rankFactor*0.4 + sizeFactor*0.3 + scoreFactor*0.3)

    // Gradually adjust performance rating (smooth transitions)
    dm.PlayerPerformance = dm.PlayerPerformance*0.7 + newPerformance*0.3

    // Clamp to valid range
    if dm.PlayerPerformance < 0.0 {
        dm.PlayerPerformance = 0.0
```

```go
    } else if dm.PlayerPerformance > 1.0 {
        dm.PlayerPerformance = 1.0
    }
}

// Get average size of top AIs
func (dm *DifficultyManager) GetAverageTopAISize(aiManager *AIManager) float32 {
    topAIs := aiManager.GetBestPerformingAIs(3) // Top 3 AIs

    if len(topAIs) == 0 {
        return 20.0 // Default size
    }

    totalSize := float32(0)
    for _, ai := range topAIs {
        totalSize += ai.Size
    }

    return totalSize / float32(len(topAIs))
}

// Update AI parameters based on current difficulty settings
func (dm *DifficultyManager) updateAIParameters(aiManager *AIManager) {
    for _, ai := range aiManager.AIs {
        // Scale aggression
        baseAggression := ai.Personality.AggressionFactor
        ai.Personality.AggressionFactor = baseAggression * dm.AIAggressionScale

        // Scale reaction time
        baseReactionTime := ai.Personality.ReactionTime
        ai.Personality.ReactionTime = baseReactionTime * dm.AIReactionTime

        // Scale power-up priority
        basePriority := ai.Personality.PowerUpPriority
        ai.Personality.PowerUpPriority = basePriority * dm.AIPowerUpPriority

        // Intelligence adjustment (affects decisions, not a direct multiplier)
        baseIntelligence := ai.Personality.IntelligenceLevel
        if dm.AIIntelligenceScale > 1.1 && baseIntelligence < 5 {
            ai.Personality.IntelligenceLevel += 1
        } else if dm.AIIntelligenceScale < 0.9 && baseIntelligence > 1 {
            ai.Personality.IntelligenceLevel -= 1
        }

        // Cap values
        if ai.Personality.AggressionFactor > 1.0 {
            ai.Personality.AggressionFactor = 1.0
```

```
        }

        if ai.Personality.PowerUpPriority > 1.0 {
            ai.Personality.PowerUpPriority = 1.0
        }
    }
}

// SetDifficulty manually changes difficulty level
func (dm *DifficultyManager) SetDifficulty(level DifficultyLevel) {
    dm.CurrentLevel = level
    dm.DDAEnabled = (level == DifficultyDynamic)
    dm.ApplyDifficultySettings()
}

// GetCurrentDifficultyText returns a string representation of current difficulty
func (dm *DifficultyManager) GetCurrentDifficultyText() string {
    switch dm.CurrentLevel {
    case DifficultyEasy:
        return "Easy"
    case DifficultyMedium:
        return "Medium"
    case DifficultyHard:
        return "Hard"
    case DifficultyDynamic:
        // Return dynamic difficulty with current level indication
        if dm.PlayerPerformance < 0.33 {
            return "Dynamic (Easy)"
        } else if dm.PlayerPerformance < 0.67 {
            return "Dynamic (Medium)"
        } else {
            return "Dynamic (Hard)"
        }
    }

    return "Medium"
}
```

8. Integrating AI into the Main Game

Finally, let's update the game code to integrate our AI systems:

go

```go
// game/game.go (additions for AI)

// Add to Game struct:
type Game struct {
    // ... existing fields

    AIManager         *systems.AIManager
    DifficultyManager *systems.DifficultyManager
}

// Update the New() function:
func New() *Game {
    // ... existing initialization

    // Create AI manager
    aiManager := systems.NewAIManager(rl.Rectangle{
        X:      0,
        Y:      0,
        Width:  constants.WorldWidth,
        Height: constants.WorldHeight,
    })

    // Initialize with some AIs
    aiManager.Initialize(30) // Start with 30 AIs

    // Create difficulty manager (dynamic by default)
    difficultyManager := systems.NewDifficultyManager(systems.DifficultyDynamic)

    game := &Game{
        // ... existing fields
        AIManager:         aiManager,
        DifficultyManager: difficultyManager,
        // ... rest of initialization
    }

    return game
}

// Update the Update method to include AI:
func (g *Game) Update(dt float32) {
    // ... existing update code

    // Reset quadtree
    g.Quadtree.Clear()

    // Add player to quadtree
```

```go
    g.Quadtree.Root.Insert(g.Player, g.Player)

    // Update AI manager
    g.AIManager.Update(dt, g)

    // Add AIs to quadtree
    for _, ai := range g.AIManager.AIs {
        g.Quadtree.Root.Insert(ai, ai)
    }

    // Update difficulty adjustment
    g.DifficultyManager.Update(dt, g.Player, g.AIManager)

    // ... rest of update code

    // Update leaderboard with AI entries
    topAIs := g.AIManager.GetBestPerformingAIs(8)
    for _, ai := range topAIs {
        g.Leaderboard.AddEntry(ai.Personality.Name, ai.Score, ai.Size, false)
    }

    // Make sure player is still in leaderboard
    g.Leaderboard.AddEntry("Player", g.Player.Score, g.Player.Size, true)
}

// Update the Draw method:
func (g *Game) Draw() {
    // ... existing drawing code

    rl.BeginMode2D(g.Camera.GetRLCamera2D())

    // ... world, food, powerups drawing

    // Draw AIs
    g.AIManager.Draw(g.Camera.GetRLCamera2D())

    // Draw player
    g.Player.Draw()

    rl.EndMode2D()

    // ... UI drawing

    // Draw difficulty indicator
    difficultyText := "Difficulty: " + g.DifficultyManager.GetCurrentDifficultyText()
    rl.DrawText(difficultyText, 20, constants.ScreenHeight - 30, 15, rl.White)
```

```go
    // Draw AI count
    aiCountText := fmt.Sprintf("AIs: %d", g.AIManager.GetAICount())
    rl.DrawText(aiCountText, 20, constants.ScreenHeight - 50, 15, rl.White)
}

// Add collision detection with AIs:
func (g *Game) CheckPlayerAICollisions() {
    // Get potential AI collisions from quadtree
    playerBounds := g.Player.GetBounds()
    potentialCollisions := make([]interface{}, 0, 10)
    g.Quadtree.Root.Query(playerBounds, &potentialCollisions)

    for _, potential := range potentialCollisions {
        ai, ok := potential.(*entities.AIHelicopter)
        if !ok {
            continue
        }

        // Skip if shielded
        if g.Player.HasShield || ai.HasShield {
            continue
        }

        // Calculate collision
        dx := g.Player.Position.X - ai.Position.X
        dy := g.Player.Position.Y - ai.Position.Y
        distSq := dx*dx + dy*dy

        // Collision radius is sum of sizes
        minDist := g.Player.Size + ai.Size

        if distSq < minDist*minDist {
            // Collision! Determine who loses
            if g.Player.Size > ai.Size * 1.1 {
                // Player is significantly bigger, AI loses
                g.DestroyAI(ai)
            } else if ai.Size > g.Player.Size * 1.1 {
                // AI is significantly bigger, player loses
                g.PlayerDestroyed()
            } else {
                // Similar sizes, both bounce off
                g.BouncePlayers(g.Player, &ai.Player)
            }
        }
    }
}
```

```go
// Helper methods for AI interactions:
func (g *Game) DestroyAI(ai *entities.AIHelicopter) {
    // Create explosion effect
    g.EffectsSystem.CreateExplosion(ai.Position, ai.Size, ai.Color)

    // Drop food/score based on AI size
    foodToDrop := int(ai.Size)

    for i := 0; i < foodToDrop; i++ {
        angle := rand.Float32() * 2 * math.Pi
        distance := rand.Float32() * ai.Size

        foodPos := rl.Vector2{
            X: ai.Position.X + float32(math.Cos(float64(angle))) * distance,
            Y: ai.Position.Y + float32(math.Sin(float64(angle))) * distance,
        }

        // Create food of appropriate value
        food := entities.NewRandomFood(foodPos)

        // Make sure it's added to the game world
        g.FoodGenerator.FoodEntities = append(g.FoodGenerator.FoodEntities, food)
    }

    // Find and remove the AI
    for i, gameAI := range g.AIManager.AIs {
        if gameAI == ai {
            g.AIManager.RemoveAI(i)
            break
        }
    }

    // Increment player's kill count
    g.Player.Score += 100 // Bonus for killing AI
}

func (g *Game) PlayerDestroyed() {
    // Player is destroyed by an AI

    // Create explosion effect
    g.EffectsSystem.CreateExplosion(g.Player.Position, g.Player.Size, g.Player.Color)

    // Drop food based on player size
    foodToDrop := int(g.Player.Size)

    for i := 0; i < foodToDrop; i++ {
        angle := rand.Float32() * 2 * math.Pi
```

```go
        distance := rand.Float32() * g.Player.Size

        foodPos := rl.Vector2{
            X: g.Player.Position.X + float32(math.Cos(float64(angle))) * distance,
            Y: g.Player.Position.Y + float32(math.Sin(float64(angle))) * distance,
        }

        // Create food
        food := entities.NewRandomFood(foodPos)
        g.FoodGenerator.FoodEntities = append(g.FoodGenerator.FoodEntities, food)
    }

    // Reset player
    g.Player.Position = rl.Vector2{
        X: constants.WorldWidth/2 + (rand.Float32() * 200 - 100),
        Y: constants.WorldHeight/2 + (rand.Float32() * 200 - 100),
    }
    g.Player.Size = g.Player.BaseSize
    g.Player.Velocity = rl.Vector2{X: 0, Y: 0}
    g.Player.Score = g.Player.Score / 2 // Lose half score on death

    // Track death for difficulty adjustment
    g.DifficultyManager.PlayerDeathCount++
}

func (g *Game) BouncePlayers(player1 *entities.Player, player2 *entities.Player) {
    // Calculate impact vector
    dx := player1.Position.X - player2.Position.X
    dy := player1.Position.Y - player2.Position.Y
    dist := float32(math.Sqrt(float64(dx*dx + dy*dy)))

    if dist == 0 {
        // Avoid division by zero
        dx = rand.Float32()*2 - 1
        dy = rand.Float32()*2 - 1
        dist = float32(math.Sqrt(float64(dx*dx + dy*dy)))
    }

    // Normalize direction
    dx /= dist
    dy /= dist

    // Bounce velocity (simplified physics)
    totalMass := player1.Size + player2.Size
    player1MassRatio := player1.Size / totalMass
    player2MassRatio := player2.Size / totalMass
```

```go
    // Apply bounce force
    bounceForce := 300.0 // Force of bounce

    player1.Velocity.X += dx * bounceForce * player2MassRatio
    player1.Velocity.Y += dy * bounceForce * player2MassRatio

    player2.Velocity.X -= dx * bounceForce * player1MassRatio
    player2.Velocity.Y -= dy * bounceForce * player1MassRatio

    // Create small collision effect
    midPoint := rl.Vector2{
        X: (player1.Position.X + player2.Position.X) / 2,
        Y: (player1.Position.Y + player2.Position.Y) / 2,
    }

    g.EffectsSystem.CreateExplosion(midPoint, 10, rl.White)
}
```

9. Performance Optimization for Many AIs

Here's a utility for benchmarking and optimizing AI performance:

go

```go
// util/performance.go
package util

import (
	"time"
	"runtime"
)

// PerformanceStats tracks game performance metrics
type PerformanceStats struct {
	FrameTime       float32
	FPS             float32
	AIUpdateTime    float32
	PhysicsTime     float32
	RenderTime      float32
	MemoryUsage     uint64
	EntityCount     int

	// Tracking for averages
	FrameTimeHistory []float32
	LastGCTime       time.Time
}

// NewPerformanceStats creates a new performance tracker
func NewPerformanceStats() *PerformanceStats {
	return &PerformanceStats{
		FrameTime:        0,
		FPS:              0,
		AIUpdateTime:     0,
		PhysicsTime:      0,
		RenderTime:       0,
		MemoryUsage:      0,
		EntityCount:      0,
		FrameTimeHistory: make([]float32, 0, 60),
		LastGCTime:       time.Now(),
	}
}

// StartTimer begins timing an operation
func StartTimer() time.Time {
	return time.Now()
}

// EndTimer finishes timing and returns elapsed seconds
func EndTimer(start time.Time) float32 {
	return float32(time.Since(start).Seconds())
```

```go
}

// UpdateStats updates performance stats
func (ps *PerformanceStats) UpdateStats(dt float32, entityCount int) {
    // Update FPS
    if dt > 0 {
        ps.FPS = 1.0 / dt
    }

    // Store frame time
    ps.FrameTime = dt
    ps.FrameTimeHistory = append(ps.FrameTimeHistory, dt)

    // Keep history to a reasonable size
    if len(ps.FrameTimeHistory) > 60 {
        ps.FrameTimeHistory = ps.FrameTimeHistory[1:]
    }

    // Track memory usage (every ~1 second)
    if time.Since(ps.LastGCTime).Seconds() > 1.0 {
        // Force GC to get accurate memory usage
        runtime.GC()

        var memStats runtime.MemStats
        runtime.ReadMemStats(&memStats)

        ps.MemoryUsage = memStats.Alloc
        ps.LastGCTime = time.Now()
    }

    // Store entity count
    ps.EntityCount = entityCount
}

// GetAverageFrameTime returns average frame time over history
func (ps *PerformanceStats) GetAverageFrameTime() float32 {
    if len(ps.FrameTimeHistory) == 0 {
        return 0
    }

    total := float32(0)
    for _, time := range ps.FrameTimeHistory {
        total += time
    }

    return total / float32(len(ps.FrameTimeHistory))
}
```

```go
// SuggestOptimizations returns suggestions for performance
func (ps *PerformanceStats) SuggestOptimizations() []string {
    suggestions := make([]string, 0)

    avgFrameTime := ps.GetAverageFrameTime()

    // Check if frame time is too high
    if avgFrameTime > 1.0/30.0 {
        suggestions = append(suggestions, "Frame rate below 30 FPS, reduce entity count")
    }

    // Check if AI updates are taking too much time
    if ps.AIUpdateTime > avgFrameTime * 0.5 {
        suggestions = append(suggestions, "AI updates taking >50% of frame time, reduce AI comp
    }

    // Check if memory usage is high
    if ps.MemoryUsage > 100*1024*1024 { // More than 100MB
        suggestions = append(suggestions, "High memory usage, check for memory leaks")
    }

    // Entity count recommendations
    if ps.EntityCount > 500 && avgFrameTime > 1.0/50.0 {
        suggestions = append(suggestions, "High entity count affecting performance, reduce fooc
    }

    return suggestions
}
```

10. Summary of AI Implementation

Our AI system provides:

1. **Diverse Personalities**: Different AI types (aggressive, cautious, strategic, etc.) with unique behaviors.
2. **State Machine Logic**: AIs intelligently switch between states:
   - Wandering to explore the map
   - Collecting food efficiently
   - Pursuing power-ups
   - Hunting smaller players
   - Evading larger threats
   - Using power-ups strategically
3. **Natural Movement**: Steering behaviors for smooth, realistic movement:

- Seek/pursuit behaviors to chase targets
- Avoidance to prevent collisions
- Wander behaviors for exploration
- Path following for collecting multiple items

4. **Strategic Decision Making**: Higher intelligence AIs make better decisions:
   - Planning efficient collection paths
   - Choosing the most valuable power-ups
   - Selecting vulnerable targets
   - Reacting appropriately to threats
5. **Performance Optimization**:
   - Staggered updates to distribute computation
   - Dynamic entity count adjustment based on performance
   - Spatial partitioning for efficient physics and collision
   - Only processing visible AI entities
6. **Difficulty Adjustment**:
   - System that measures player performance
   - Dynamically adjusts AI aggression, intelligence, etc.
   - Multiple difficulty levels for player preference
7. **Emergent Behavior**:
   - AIs will naturally form food-collection routes
   - Predatory behavior emerges as AIs grow larger
   - Smaller AIs naturally avoid dangerous areas

This implementation creates AI opponents that are engaging, varied, and can adapt to player skill level, while maintaining good performance even with many entities on screen.

{
case *Player:
player = p
position = p.Position
size = p.Size
case *AIHelicopter:
player = &p.Player
position = p.Position
size = p.Size
default:

```
continue
}
```

```
    // Only avoid if the player is larger
    if size <= a.Size {
        continue
    }

    // Calculate vector from obstacle to AI
    awayVector := rl.Vector2{
        X: a.Position.X - position.X,
        Y: a.Position.Y - position.Y,
    }

    // Distance squared to obstacle
    distSq := awayVector.X*awayVector.X + awayVector.Y*awayVector.Y

    // Avoidance radius (based on sizes)
    avoidanceRadiusSq := (a.Size + size + 50) * (a.Size + size + 50)

    // If within avoidance radius, steer away
    if distSq < avoidanceRadiusSq {
        // Normalize away vector
        dist := float32(math.Sqrt(float64(distSq)))
        if dist > 0 {
            awayVector.X /= dist
            awayVector.Y /= dist
        }

        // Strength of avoidance is higher when closer
        avoidanceStrength := 1.0 - (distSq / avoidanceRadiusSq)
        avoidanceStrength *= 2.0 // Scale up for stronger avoidance

        // Apply avoidance to steering, weighted by avoidance strength
        steering.X += awayVector.X * avoidanceStrength
        steering.Y += awayVector.Y * avoidanceStrength

        // Renormalize steering
        steeringLength := float32(math.Sqrt(float64(steering.X*steering.X +
steering.Y*steering.Y)))
        if steeringLength > 0 {
            steering.X /= steeringLength
            steering.Y /= steeringLength
        }
    }
}

return steering
```

```
}
```

5. State-Specific Target Selection

Now let's implement the methods for selecting targets in each AI state:

```go
// Choose a target for wandering
func (a *AIHelicopter) chooseWanderTarget(game *game.Game) {
    // Get current world bounds
    worldBounds := rl.Rectangle{
        X:      0,
        Y:      0,
        Width:  constants.WorldWidth,
        Height: constants.WorldHeight,
    }

    // Choose a random position within the bounds, favoring unexplored areas
    var targetX, targetY float32

    // Higher intelligence means more strategic wandering
    if a.Personality.IntelligenceLevel >= 4 && rand.Float32() < 0.7 {
        // More intelligent AI will wander toward center or food-rich areas

        // Sometimes head toward center
        if rand.Float32() < 0.3 {
            targetX = worldBounds.X + worldBounds.Width * (0.4 + rand.Float32() * 0.2)
            targetY = worldBounds.Y + worldBounds.Height * (0.4 + rand.Float32() * 0.2)
        } else if len(game.FoodGenerator.FoodEntities) > 0 && rand.Float32() < 0.7 {
            // Head toward a general area with food
            // Pick a random existing food to navigate toward
            foodCount := len(game.FoodGenerator.FoodEntities)
            randomFood := game.FoodGenerator.FoodEntities[rand.Intn(foodCount)]

            // Don't go directly to food (that's collecting), but to the general area
            targetX = randomFood.Position.X + (rand.Float32()*2-1) * 300
            targetY = randomFood.Position.Y + (rand.Float32()*2-1) * 300
        } else {
            // Explore a random location
            targetX = worldBounds.X + rand.Float32() * worldBounds.Width
            targetY = worldBounds.Y + rand.Float32() * worldBounds.Height
        }
    } else {
        // Less intelligent AI just wanders randomly
        targetX = worldBounds.X + rand.Float32() * worldBounds.Width
        targetY = worldBounds.Y + rand.Float32() * worldBounds.Height
    }
```

```go
    // Set the target
    a.Target = rl.Vector2{X: targetX, Y: targetY}
    a.TargetEntity = nil
}

// Choose a target for food collection
func (a *AIHelicopter) chooseCollectionTarget() {
    // If we don't know about any food, revert to wandering
    if len(a.KnownFoodPositions) == 0 {
        a.transitionToState(AIStateWandering)
        return
    }

    // Higher intelligence means more strategic food collection
    if a.Personality.IntelligenceLevel >= 3 && len(a.KnownFoodPositions) > 1 {
        // Find the best collection path
        // Start with closest food
        closestIdx := -1
        closestDistSq := float32(math.MaxFloat32)

        for i, foodPos := range a.KnownFoodPositions {
            dx := foodPos.X - a.Position.X
            dy := foodPos.Y - a.Position.Y
            distSq := dx*dx + dy*dy

            if distSq < closestDistSq {
                closestDistSq = distSq
                closestIdx = i
            }
        }

        if closestIdx >= 0 {
            // Set target to closest food
            a.Target = a.KnownFoodPositions[closestIdx]

            // Plan a collection path for future movement
            a.calculateCollectionPath(a.calculateSteering())
        }
    } else {
        // Less intelligent AI just goes for closest food
        closestIdx := -1
        closestDistSq := float32(math.MaxFloat32)

        for i, foodPos := range a.KnownFoodPositions {
            dx := foodPos.X - a.Position.X
            dy := foodPos.Y - a.Position.Y
```

```go
            distSq := dx*dx + dy*dy

            if distSq < closestDistSq {
                closestDistSq = distSq
                closestIdx = i
            }
        }

        if closestIdx >= 0 {
            a.Target = a.KnownFoodPositions[closestIdx]
        }
    }
}

// Choose a target power-up to pursue
func (a *AIHelicopter) choosePowerUpTarget() {
    // If we don't know about any power-ups, revert to wandering
    if len(a.KnownPowerUps) == 0 {
        a.transitionToState(AIStateWandering)
        return
    }

    // Find the best power-up
    bestPowerUpIdx := -1
    bestScore := float32(-1)

    for i, powerUpObj := range a.KnownPowerUps {
        powerUp, ok := powerUpObj.(*PowerUp)
        if !ok {
            continue
        }

        // Calculate distance
        dx := powerUp.Position.X - a.Position.X
        dy := powerUp.Position.Y - a.Position.Y
        distSq := dx*dx + dy*dy
        dist := float32(math.Sqrt(float64(distSq)))

        // Calculate score based on distance and power-up type
        score := 1000.0 / (dist + 1.0) // Base score is higher for closer power-ups

        // Adjust score based on power-up type and personality
        switch powerUp.Type {
        case PowerUpMagnet:
            // Collectors prioritize magnets
            if a.Personality.CollectionEfficiency > 0.6 {
                score *= 1.5
```

```go
            }

        case PowerUpSpeed:
            // Explorers and hunters prioritize speed
            if a.Personality.ExplorationFactor > 0.6 || a.Personality.AggressionFactor > 0.6
{
                score *= 1.5
            }

        case PowerUpShield:
            // Cautious AIs prioritize shields
            if a.Personality.CautionFactor > 0.6 {
                score *= 1.5
            }
            // Also prioritize shields when threatened
            if a.ThreatLevel > 0.4 {
                score *= 1.0 + a.ThreatLevel
            }

        case PowerUpSizeboost:
            // Aggressive AIs prioritize size boosts
            if a.Personality.AggressionFactor > 0.6 {
                score *= 1.5
            }
        }

        // More intelligent AIs make better choices
        if a.Personality.IntelligenceLevel >= 4 {
            // If very close to any power-up, prioritize it
            if dist < 100 {
                score *= 1.5
            }

            // If already have a power-up, less incentive to get same type
            for powerType := range a.ActivePowerUps {
                if powerType == powerUp.Type {
                    score *= 0.5
                }
            }
        }

        if score > bestScore {
            bestScore = score
            bestPowerUpIdx = i
        }
    }
```

```go
    if bestPowerUpIdx >= 0 {
        powerUp, ok := a.KnownPowerUps[bestPowerUpIdx].(*PowerUp)
        if ok {
            a.Target = powerUp.Position
            a.TargetEntity = powerUp
        }
    }
}

// Choose a player to hunt
func (a *AIHelicopter) chooseHuntTarget() {
    // If no players to hunt, revert to wandering
    if !a.canHuntPlayers() {
        a.transitionToState(AIStateWandering)
        return
    }

    // Find the best player to hunt
    bestTargetID := -1
    bestScore := float32(-1)

    for id, playerObj := range a.KnownPlayers {
        var playerPos rl.Vector2
        var playerSize float32

        // Extract position and size based on player type
        switch p := playerObj.(type) {
        case *Player:
            playerPos = p.Position
            playerSize = p.Size
        case *AIHelicopter:
            playerPos = p.Position
            playerSize = p.Size
        default:
            continue
        }

        // Skip if not smaller
        if playerSize >= a.Size * 0.9 {
            continue
        }

        // Calculate distance
        dx := playerPos.X - a.Position.X
        dy := playerPos.Y - a.Position.Y
        distSq := dx*dx + dy*dy
        dist := float32(math.Sqrt(float64(distSq)))
```

```
        // Calculate score based on distance and size difference
        sizeDiff := a.Size - playerSize
        score := sizeDiff * 1000.0 / (dist + 1.0)

        // More intelligent AIs make better hunting decisions
        if a.Personality.IntelligenceLevel >= 4 {
            // Consider target's power-ups if we can detect them (higher intelligence)
            if player, ok := playerObj.(*Player); ok && player.HasShield {
                score *= 0.5 // Less desirable to hunt shielded players
            } else if aiPlayer, ok := playerObj.(*AIHelicopter); ok && aiPlayer.HasShield {
                score *= 0.5
            }

            // Consider number of nearby food - hunting in food-rich areas is more
beneficial
            foodNearTarget := 0
            for _, foodPos := range a.KnownFoodPositions {
                dx := foodPos.X - playerPos.X
                dy := foodPos.Y - playerPos.Y
                foodDistSq := dx*dx + dy*dy

                if foodDistSq < 200*200 {
                    foodNearTarget++
                }
            }

            score *= (1.0 + float32(foodNearTarget) * 0.1)
        }

        if score > bestScore {
            bestScore = score
            bestTargetID = id
        }
    }

    if bestTargetID >= 0 {
        a.TargetEntity = a.KnownPlayers[bestTargetID]

        // Extract position based on player type
        switch p := a.TargetEntity.(type) {
        case *Player:
            a.Target = p.Position
        case *AIHelicopter:
            a.Target = p.Position
        }
    }
```

```go
}

// Choose a direction to evade
func (a *AIHelicopter) chooseEvadeTarget() {
    // Find the biggest threat
    var threatPos rl.Vector2
    var threatSize float32
    maxThreat := float32(0)

    for _, playerObj := range a.KnownPlayers {
        var playerPos rl.Vector2
        var playerSize float32

        // Extract position and size based on player type
        switch p := playerObj.(type) {
        case *Player:
            playerPos = p.Position
            playerSize = p.Size
        case *AIHelicopter:
            playerPos = p.Position
            playerSize = p.Size
        default:
            continue
        }

        // Skip if not bigger
        if playerSize <= a.Size {
            continue
        }

        // Calculate distance
        dx := playerPos.X - a.Position.X
        dy := playerPos.Y - a.Position.Y
        distSq := dx*dx + dy*dy
        dist := float32(math.Sqrt(float64(distSq)))

        // Calculate threat based on size difference and distance
        threat := (playerSize - a.Size) / (dist + 1.0)

        if threat > maxThreat {
            maxThreat = threat
            threatPos = playerPos
            threatSize = playerSize
        }
    }

    // If no threat found, go back to previous state
```

```go
    if maxThreat == 0 {
        a.State = a.PreviousState
        return
    }

    // Calculate evasion direction (away from threat)
    evadeDir := rl.Vector2{
        X: a.Position.X - threatPos.X,
        Y: a.Position.Y - threatPos.Y,
    }

    // Normalize
    evadeDist := float32(math.Sqrt(float64(evadeDir.X*evadeDir.X + evadeDir.Y*evadeDir.Y)))
    if evadeDist > 0 {
        evadeDir.X /= evadeDist
        evadeDir.Y /= evadeDist
    }

    // Calculate target position (away from threat)
    evadeDistance := 300.0 + threatSize * 2.0
    targetX := a.Position.X + evadeDir.X * evadeDistance
    targetY := a.Position.Y + evadeDir.Y * evadeDistance

    // Make sure target is in world bounds
    worldBounds := rl.Rectangle{
        X:      0,
        Y:      0,
        Width:  constants.WorldWidth,
        Height: constants.WorldHeight,
    }

    targetX = util.Clamp(targetX, worldBounds.X + a.Size, worldBounds.X + worldBounds.Width
- a.Size)
    targetY = util.Clamp(targetY, worldBounds.Y + a.Size, worldBounds.Y + worldBounds.Height
- a.Size)

    // Set target
    a.Target = rl.Vector2{X: targetX, Y: targetY}
    a.TargetEntity = nil
}

// Activate and use power-ups effectively
func (a *AIHelicopter) activatePowerUps() {
    // If we have a shield power-up and feel threatened, prioritize it
    if a.HasActivePowerUp(PowerUpShield) && a.ThreatLevel > 0.3 {
        // Find nearby food to collect while shielded
        if len(a.KnownFoodPositions) > 0 {
```

```go
		a.chooseCollectionTarget()
	} else {
		// No food, try approaching a power-up
		if len(a.KnownPowerUps) > 0 {
			a.choosePowerUpTarget()
		} else {
			// Just wander
			a.chooseWanderTarget(nil)
		}
	}
	return
}

// If we have magnet, prioritize food-rich areas
if a.HasActivePowerUp(PowerUpMagnet) {
	if len(a.KnownFoodPositions) > 0 {
		// Find the area with the most food
		bestFoodClusterPos := a.findBestFoodCluster()
		a.Target = bestFoodClusterPos
	} else {
		// No food visible, go exploring
		a.chooseWanderTarget(nil)
	}
	return
}

// If we have speed boost, use it for hunting or power-up collection
if a.HasActivePowerUp(PowerUpSpeed) {
	if a.canHuntPlayers() && a.Personality.AggressionFactor > 0.4 {
		a.chooseHuntTarget()
	} else if len(a.KnownPowerUps) > 0 {
		a.choosePowerUpTarget()
	} else {
		// Use speed to explore
		a.chooseWanderTarget(nil)
	}
	return
}

// If we have size boost active, prioritize food collection
if a.HasActivePowerUp(PowerUpSizeboost) {
	if len(a.KnownFoodPositions) > 0 {
		a.chooseCollectionTarget()
	} else {
		a.chooseWanderTarget(nil)
	}
	return
```

```go
    }

    // Default: just collect food
    a.chooseCollectionTarget()
}

// Find the area with the highest concentration of food
func (a *AIHelicopter) findBestFoodCluster() rl.Vector2 {
    if len(a.KnownFoodPositions) == 0 {
        return a.Position
    }

    // Simple approach: find the food with the most neighbors
    bestIdx := -1
    maxNeighbors := -1

    for i, pos1 := range a.KnownFoodPositions {
        neighbors := 0

        for j, pos2 := range a.KnownFoodPositions {
            if i == j {
                continue
            }

            dx := pos1.X - pos2.X
            dy := pos1.Y - pos2.Y
            distSq := dx*dx + dy*dy

            // Count food within cluster radius
            if distSq < 150*150 {
                neighbors++
            }
        }

        if neighbors > maxNeighbors {
            maxNeighbors = neighbors
            bestIdx = i
        }
    }

    if bestIdx >= 0 {
        return a.KnownFoodPositions[bestIdx]
    }

    // Fallback to closest food
    closestIdx := -1
    closestDistSq := float32(math.MaxFloat32)
```

```go
    for i, pos := range a.KnownFoodPositions {
        dx := pos.X - a.Position.X
        dy := pos.Y - a.Position.Y
        distSq := dx*dx + dy*dy

        if distSq < closestDistSq {
            closestDistSq = distSq
            closestIdx = i
        }
    }

    if closestIdx >= 0 {
        return a.KnownFoodPositions[closestIdx]
    }

    return a.Position
}

// Update target for current state
func (a *AIHelicopter) updateCurrentStateTarget(game *game.Game) {
    switch a.State {
    case AIStateWandering:
        // Check if we've reached the target or been wandering too long
        dx := a.Target.X - a.Position.X
        dy := a.Target.Y - a.Position.Y
        distSq := dx*dx + dy*dy

        if distSq < 50*50 || a.StateTimer > 5.0 {
            a.chooseWanderTarget(game)
        }

    case AIStateCollecting:
        // Update to collect new food or move to next food
        a.chooseCollectionTarget()

    case AIStatePursuingPowerUp:
        // Check if target power-up still exists
        if a.TargetEntity == nil {
            a.choosePowerUpTarget()
            return
        }

        // Check if we're close to target
        dx := a.Target.X - a.Position.X
        dy := a.Target.Y - a.Position.Y
        distSq := dx*dx + dy*dy
```

```
        if distSq < 30*30 {
            // We probably collected it, choose a new target
            a.choosePowerUpTarget()
        }

    case AIStateHunting:
        // Update hunting target
        a.chooseHuntTarget()

    case AIStateEvading:
        // Update evasion direction
        a.chooseEvadeTarget()

    case AIStateUsingPowerUp:
        // Continue using power-up
        a.activatePowerUps()
    }
}
```

6. AI Manager System

Now let's implement a system to create, manage, and optimize multiple AI players:

go

```go
// systems/ai_manager.go
package systems

import (
	"github.com/gen2brain/raylib-go/raylib"
	"atomblaster/constants"
	"atomblaster/entities"
	"atomblaster/util"
	"math/rand"
	"time"
)

// Names for AI helicopters
var aiNames = []string{
	"HeliZapper", "RotorRider", "ChopperChamp", "AirPirate", "SkyRanger",
	"BladeRunner", "ThunderBird", "WhirlyBird", "AirWolf", "FlyingAce",
	"SkyHunter", "VerticalThreat", "PropWash", "RotorRush", "BladeStorm",
	"HoverHero", "ChopperChief", "AeroAce", "HelixHunter", "DownDraft",
	"UpDraft", "SkySlasher", "CopterKing", "PropPunisher", "WindWhirler",
	"StormSeeker", "CloudCutter", "SkyStriker", "BladeBreaker", "HoverHazard"
}

// Personality types for variety
var personalityTypes = []string{
	"aggressive", "cautious", "balanced", "collector", "explorer", "strategic"
}

// AIManager manages all AI helicopters in the game
type AIManager struct {
	AIs               []*entities.AIHelicopter
	MaxAIs            int
	ActiveAIs         int
	MinAISize         float32
	MaxAISize         float32
	SpawnTimer        float32
	SpawnInterval     float32
	TargetAICount     int

	// Performance management
	UpdateGroups      []int // Indices into AIs, grouped for staggered updates
	CurrentGroup      int
	UpdateInterval    float32
	LastUpdateTime    float32

	// Average and total load factors
	AverageLoad       float32
```

```go
    TotalComputeTime  float32
    FramesTracked     int
    LastLoadCheck     time.Time

    // World information
    WorldBounds       rl.Rectangle

    // Name management
    UsedNames         map[string]bool
}

// NewAIManager creates a new manager for AI helicopters
func NewAIManager(worldBounds rl.Rectangle) *AIManager {
    return &AIManager{
        AIs:              make([]*entities.AIHelicopter, 0, 100),
        MaxAIs:           100, // Can go higher if needed
        ActiveAIs:        0,
        MinAISize:        20.0,
        MaxAISize:        50.0,
        SpawnTimer:       0,
        SpawnInterval:    0.5, // Spawn a new AI every 0.5 seconds until target reached
        TargetAICount:    50,  // Target number of AIs to maintain
        UpdateGroups:     make([]int, 5), // 5 update groups
        CurrentGroup:     0,
        UpdateInterval:   0.05, // 50ms between group updates (all AIs update over ~250ms)
        LastUpdateTime:   0,
        AverageLoad:      0,
        TotalComputeTime: 0,
        FramesTracked:    0,
        LastLoadCheck:    time.Now(),
        WorldBounds:      worldBounds,
        UsedNames:        make(map[string]bool),
    }
}

// Initialize with starting AIs
func (am *AIManager) Initialize(initialCount int) {
    // Cap initial count to max
    if initialCount > am.MaxAIs {
        initialCount = am.MaxAIs
    }

    // Create initial AIs
    for i := 0; i < initialCount; i++ {
        am.SpawnAI()
    }
```

```go
    // Set up update groups (divide AIs into groups for staggered updates)
    am.reorganizeUpdateGroups()
}

// SpawnAI creates a new AI helicopter
func (am *AIManager) SpawnAI() *entities.AIHelicopter {
    if am.ActiveAIs >= am.MaxAIs {
        return nil
    }

    // Choose a random position (not too close to world edge)
    padding := float32(100)
    spawnX := am.WorldBounds.X + padding + rand.Float32() * (am.WorldBounds.Width - padding*2)
    spawnY := am.WorldBounds.Y + padding + rand.Float32() * (am.WorldBounds.Height - padding*2)

    // Choose a random personality
    personalityType := personalityTypes[rand.Intn(len(personalityTypes))]

    // Choose a random name that hasn't been used
    name := am.getUnusedName()

    // Create the AI
    ai := entities.NewAIHelicopter(name, rl.Vector2{X: spawnX, Y: spawnY}, personalityType)

    // Set initial size (random between min and max)
    baseSize := am.MinAISize + rand.Float32() * (am.MaxAISize - am.MinAISize)
    ai.Size = baseSize

    // Add to list
    am.AIs = append(am.AIs, ai)
    am.ActiveAIs++

    // Need to reorganize update groups when adding AIs
    am.reorganizeUpdateGroups()

    return ai
}

// Get a name that hasn't been used yet
func (am *AIManager) getUnusedName() string {
    // If all names are used, reset the map
    if len(am.UsedNames) >= len(aiNames) {
        am.UsedNames = make(map[string]bool)
    }

    // Try to find an unused name
    for _, name := range aiNames {
```

```go
        if !am.UsedNames[name] {
            am.UsedNames[name] = true
            return name
        }
    }

    // Fallback: generate a numbered name
    return "Copter" + string(rune(rand.Intn(100)))
}

// Reorganize AIs into update groups
func (am *AIManager) reorganizeUpdateGroups() {
    // Clear current groups
    groupCount := 5 // Number of update groups
    am.UpdateGroups = make([][]int, groupCount)

    // Distribute AIs evenly across groups
    for i := 0; i < len(am.AIs); i++ {
        groupIdx := i % groupCount
        am.UpdateGroups[groupIdx] = append(am.UpdateGroups[groupIdx], i)
    }
}

// Update manages AI system each frame
func (am *AIManager) Update(dt float32, game interface{}) {
    // Handle spawning new AIs if below target
    am.SpawnTimer += dt
    if am.ActiveAIs < am.TargetAICount && am.SpawnTimer >= am.SpawnInterval {
        am.SpawnTimer = 0
        am.SpawnAI()
    }

    // Track load for performance management
    startTime := time.Now()

    // Update AIs in current group only
    am.LastUpdateTime += dt
    if am.LastUpdateTime >= am.UpdateInterval {
        am.LastUpdateTime = 0

        // Update AIs in current group
        if am.CurrentGroup < len(am.UpdateGroups) {
            for _, idx := range am.UpdateGroups[am.CurrentGroup] {
                if idx < len(am.AIs) {
                    am.AIs[idx].Update(dt, game)

                    // Check if AI should be removed (e.g., if too small)
```

```
                if am.AIs[idx].Size < 10 {
                    am.RemoveAI(idx)
                }
            }
        }
    }

    // Move to next group
    am.CurrentGroup = (am.CurrentGroup + 1) % len(am.UpdateGroups)
}

// Track load
am.TotalComputeTime += float32(time.Since(startTime).Seconds())
am.FramesTracked++

// Periodically check load and adjust AI count if needed
if time.Since(am.LastLoadCheck).Seconds() >= 1.0 {
    am.AverageLoad = am.TotalComputeTime / float32(am.FramesTracked)
    am.TotalComputeTime = 0
    am.FramesTracked = 0
    am.LastLoadCheck = time.Now()

    // If average load is too high, reduce target AI count
    if am.AverageLoad > 0.02 { // More than 20ms per frame on AI
        am.TargetAICount = int(float32(am.TargetAICount) * 0.9)
        if am.TargetAICount < 10 {
            am.TargetAICount = 10 // Minimum 10 AIs
        }
    } else if am.AverageLoad < 0.01 && am.TargetAICount < am.MaxAIs {
        // If load is low, increase target AI count
        am.TargetAICount = int(float32(am.TargetAICount) * 1.1)
        if am.TargetAICount > am.MaxAIs {
            am.TargetAICount = am.MaxAIs
        }
    }
}

// Remove excess AIs if above target
if am.ActiveAIs > am.TargetAICount {
    // Remove one AI per frame until we reach target
    am.RemoveAI(len(am.AIs) - 1) // Remove last AI (simplest approach)
}
}

// RemoveAI removes an AI by index
func (am *AIManager) RemoveAI(index int# Advanced AI Helicopter Opponents Implementation
```

1. AI Architecture Overview

Creating believable AI opponents that make the game world feel populated requires a multi-layer

1. Supports different difficulty levels and behaviors
2. Uses state machines for decision-making
3. Implements steering behaviors for natural movement
4. Includes a perception system to simulate awareness
5. Features adaptive strategies based on game conditions
6. Balances CPU usage to support hundreds of AI agents

2. AI Controller Base System

First, let's create the basic AI controller architecture:

```go
// entities/ai_helicopter.go
package entities

import (
	"github.com/gen2brain/raylib-go/raylib"
	"atomblaster/constants"
	"atomblaster/util"
	"math"
	"math/rand"
)

// AIState represents different behavior states of the AI helicopter
type AIState int

const (
	AIStateWandering AIState = iota
	AIStateCollecting
	AIStatePursuingPowerUp
	AIStateHunting
	AIStateEvading
	AIStateUsingPowerUp
)

// AIPersonality defines behavior patterns for different AI types
type AIPersonality struct {
	Name                string
	AggressionFactor    float32 // 0.0 to 1.0, how likely to attack other players
	CautionFactor       float32 // 0.0 to 1.0, how likely to run away when in danger
	CollectionEfficiency float32 // 0.0 to 1.0, how strategically it collects food
	PowerUpPriority     float32 // 0.0 to 1.0, how much it prioritizes power-ups
	ReactionTime        float32 // Delay in seconds before responding to events
```

```go
    IntelligenceLevel   int     // 1-5, affects decision making complexity
    ExplorationFactor   float32 // 0.0 to 1.0, tendency to explore new areas
}

// AIHelicopter extends the basic helicopter with AI-specific fields
type AIHelicopter struct {
    Player              // Embed the player struct for all helicopter functionality

    // AI-specific fields
    State               AIState
    Personality         AIPersonality
    PreviousState       AIState
    StateTimer          float32
    Target              rl.Vector2
    TargetEntity        interface{}
    PerceptionRadius    float32 // How far the AI can "see"
    ReactionTimer       float32 // Tracks reaction time

    // Path following
    PathPoints          []rl.Vector2
    CurrentPathIndex    int
    PathUpdateTimer     float32

    // Decision making
    DecisionTimer       float32
    LastDecisionTime    float32

    // Tactical data
    KnownFoodPositions  []rl.Vector2
    KnownPowerUps       []interface{}
    KnownPlayers        map[int]interface{} // Players the AI is aware of
    ThreatLevel         float32 // 0.0 to 1.0, current perceived danger

    // Memory system
    MemoryTimeout       float32
    LastSeenFood        map[int]float32 // ID to timestamp
    LastSeenPowerUp     map[int]float32
    LastSeenPlayer      map[int]float32

    // Performance management
    ThinkInterval       float32 // How often to make major decisions
    DecisionAgeMax      float32 // Max age of a decision before reconsidering

    // Debug
    DebugMode           bool
}
```

```go
// NewAIHelicopter creates a new AI-controlled helicopter with a given personality
func NewAIHelicopter(name string, position rl.Vector2, personalityType string) *AIHelicopter {
    // Create base helicopter
    player := NewPlayer()
    player.Position = position
    player.Scale = 1.0

    // Set color based on personality
    player.Color = generateColorForPersonality(personalityType)

    // Create the AI personality
    personality := generatePersonality(personalityType)
    personality.Name = name

    // Create AI helicopter
    aiHelicopter := &AIHelicopter{
        Player:            *player,
        State:             AIStateWandering,
        Personality:       personality,
        PreviousState:     AIStateWandering,
        StateTimer:        0,
        Target:            rl.Vector2{X: 0, Y: 0},
        TargetEntity:      nil,
        PerceptionRadius:  400.0, // Base perception radius
        PathPoints:        make([]rl.Vector2, 0),
        CurrentPathIndex:  0,
        PathUpdateTimer:   0,
        DecisionTimer:     0,
        LastDecisionTime:  0,
        KnownFoodPositions: make([]rl.Vector2, 0),
        KnownPowerUps:     make([]interface{}, 0),
        KnownPlayers:      make(map[int]interface{}),
        ThreatLevel:       0,
        MemoryTimeout:     5.0, // 5 seconds memory for seen entities
        LastSeenFood:      make(map[int]float32),
        LastSeenPowerUp:   make(map[int]float32),
        LastSeenPlayer:    make(map[int]float32),
        ThinkInterval:     0.2 + rand.Float32() * 0.3, // 0.2-0.5 seconds between decisions
        DecisionAgeMax:    1.0 + rand.Float32() * 1.0, // 1-2 seconds before reconsidering
        DebugMode:         false,
    }

    // Adjust perception based on intelligence
    aiHelicopter.PerceptionRadius *= (0.7 + float32(personality.IntelligenceLevel) * 0.1)

    return aiHelicopter
}
```

```go
// Generate a repeatable color based on personality type
func generateColorForPersonality(personalityType string) rl.Color {
    switch personalityType {
    case "aggressive":
        return rl.Red
    case "cautious":
        return rl.Blue
    case "balanced":
        return rl.Orange
    case "collector":
        return rl.Green
    case "explorer":
        return rl.Purple
    case "strategic":
        return rl.Yellow
    default:
        // Generate a pseudo-random but consistent color based on name
        hash := util.HashString(personalityType)
        r := uint8((hash) % 200 + 55)
        g := uint8((hash >> 8) % 200 + 55)
        b := uint8((hash >> 16) % 200 + 55)
        return rl.Color{R: r, G: g, B: b, A: 255}
    }
}

// Generate personality traits based on personality type
func generatePersonality(personalityType string) AIPersonality {
    // Base personality with moderate values
    personality := AIPersonality{
        AggressionFactor:     0.5,
        CautionFactor:        0.5,
        CollectionEfficiency: 0.5,
        PowerUpPriority:      0.5,
        ReactionTime:         0.3,
        IntelligenceLevel:    3,
        ExplorationFactor:    0.5,
    }

    // Adjust based on type
    switch personalityType {
    case "aggressive":
        personality.AggressionFactor = 0.8 + rand.Float32() * 0.2
        personality.CautionFactor = 0.2 + rand.Float32() * 0.2
        personality.PowerUpPriority = 0.7 + rand.Float32() * 0.3
        personality.ReactionTime = 0.1 + rand.Float32() * 0.2
        personality.IntelligenceLevel = 3 + rand.Intn(3)
```

```go
    case "cautious":
        personality.AggressionFactor = 0.1 + rand.Float32() * 0.2
        personality.CautionFactor = 0.8 + rand.Float32() * 0.2
        personality.CollectionEfficiency = 0.6 + rand.Float32() * 0.3
        personality.ReactionTime = 0.2 + rand.Float32() * 0.2
        personality.ExplorationFactor = 0.3 + rand.Float32() * 0.3

    case "balanced":
        personality.AggressionFactor = 0.4 + rand.Float32() * 0.3
        personality.CautionFactor = 0.4 + rand.Float32() * 0.3
        personality.CollectionEfficiency = 0.4 + rand.Float32() * 0.3
        personality.PowerUpPriority = 0.4 + rand.Float32() * 0.3
        personality.ReactionTime = 0.2 + rand.Float32() * 0.2
        personality.IntelligenceLevel = 3 + rand.Intn(2)

    case "collector":
        personality.AggressionFactor = 0.2 + rand.Float32() * 0.2
        personality.CollectionEfficiency = 0.8 + rand.Float32() * 0.2
        personality.PowerUpPriority = 0.4 + rand.Float32() * 0.3
        personality.ExplorationFactor = 0.7 + rand.Float32() * 0.3

    case "explorer":
        personality.ExplorationFactor = 0.8 + rand.Float32() * 0.2
        personality.AggressionFactor = 0.3 + rand.Float32() * 0.3
        personality.CautionFactor = 0.3 + rand.Float32() * 0.3

    case "strategic":
        personality.IntelligenceLevel = 4 + rand.Intn(2)
        personality.CautionFactor = 0.5 + rand.Float32() * 0.3
        personality.PowerUpPriority = 0.7 + rand.Float32() * 0.3
        personality.AggressionFactor = 0.5 + rand.Float32() * 0.3
        personality.CollectionEfficiency = 0.6 + rand.Float32() * 0.4
    }

    return personality
}
```

3. AI Update and Decision Making Logic

Now let's implement the core update and decision-making functionality:

go

```go
// AI update method (extends the Player update method)
func (a *AIHelicopter) Update(dt float32, gameState interface{}) {
    // Cast the game state to access needed information
    game := gameState.(*game.Game)

    // Update timers
    a.StateTimer += dt
    a.DecisionTimer += dt
    a.PathUpdateTimer += dt
    a.ReactionTimer += dt

    // Update memory timeouts
    a.updateMemory(dt)

    // Perception - what the AI can see
    if a.ReactionTimer >= a.Personality.ReactionTime {
        a.perceiveEnvironment(game)
        a.ReactionTimer = 0
    }

    // Decision making - only on interval to save CPU
    if a.DecisionTimer >= a.ThinkInterval {
        a.makeDecisions(game)
        a.DecisionTimer = 0
        a.LastDecisionTime = game.GameTime
    }

    // Movement based on current state and target
    a.updateMovement(dt)

    // Call the base Player update (physics, collision, etc.)
    a.Player.Update(dt)
}

// Update the AI's memory of objects in the world
func (a *AIHelicopter) updateMemory(dt float32) {
    // Forget food that hasn't been seen for a while
    for id, lastSeen := range a.LastSeenFood {
        a.LastSeenFood[id] -= dt
        if a.LastSeenFood[id] <= 0 {
            delete(a.LastSeenFood, id)
        }
    }

    // Forget power-ups that haven't been seen for a while
    for id, lastSeen := range a.LastSeenPowerUp {
```

```go
        a.LastSeenPowerUp[id] -= dt
        if a.LastSeenPowerUp[id] <= 0 {
            delete(a.LastSeenPowerUp, id)
        }
    }

    // Forget players that haven't been seen for a while
    for id, lastSeen := range a.LastSeenPlayer {
        a.LastSeenPlayer[id] -= dt
        if a.LastSeenPlayer[id] <= 0 {
            delete(a.LastSeenPlayer, id)
            delete(a.KnownPlayers, id)
        }
    }
}

// Perceive the environment - what the AI can "see"
func (a *AIHelicopter) perceiveEnvironment(game *game.Game) {
    // Get entities near the AI
    nearbyEntities := make([]interface{}, 0)

    // Use the quadtree for efficient spatial querying
    searchRect := rl.Rectangle{
        X:      a.Position.X - a.PerceptionRadius,
        Y:      a.Position.Y - a.PerceptionRadius,
        Width:  a.PerceptionRadius * 2,
        Height: a.PerceptionRadius * 2,
    }
    game.Quadtree.Root.Query(searchRect, &nearbyEntities)

    // Process entities the AI can perceive
    a.KnownFoodPositions = make([]rl.Vector2, 0)
    a.KnownPowerUps = make([]interface{}, 0)

    for _, entity := range nearbyEntities {
        // Check what type of entity this is
        switch e := entity.(type) {
        case *Food:
            foodID := uintptr(unsafe.Pointer(e))
            a.LastSeenFood[int(foodID)] = a.MemoryTimeout
            a.KnownFoodPositions = append(a.KnownFoodPositions, e.Position)

        case *PowerUp:
            powerUpID := uintptr(unsafe.Pointer(e))
            a.LastSeenPowerUp[int(powerUpID)] = a.MemoryTimeout
            a.KnownPowerUps = append(a.KnownPowerUps, e)
```

```go
        case *Player:
            // Don't perceive self
            if e == &a.Player {
                continue
            }

            playerID := uintptr(unsafe.Pointer(e))
            a.LastSeenPlayer[int(playerID)] = a.MemoryTimeout
            a.KnownPlayers[int(playerID)] = e

        case *AIHelicopter:
            // Don't perceive self
            if e == a {
                continue
            }

            aiID := uintptr(unsafe.Pointer(e))
            a.LastSeenPlayer[int(aiID)] = a.MemoryTimeout
            a.KnownPlayers[int(aiID)] = e
        }
    }

    // Calculate threat level based on nearby players
    a.calculateThreatLevel()
}

// Calculate how threatened the AI feels
func (a *AIHelicopter) calculateThreatLevel() {
    a.ThreatLevel = 0

    for _, playerObj := range a.KnownPlayers {
        var playerPos rl.Vector2
        var playerSize float32

        // Extract position and size based on player type
        switch p := playerObj.(type) {
        case *Player:
            playerPos = p.Position
            playerSize = p.Size
        case *AIHelicopter:
            playerPos = p.Position
            playerSize = p.Size
        default:
            continue
        }

        // Calculate distance
```

```go
        dx := playerPos.X - a.Position.X
        dy := playerPos.Y - a.Position.Y
        distSq := dx*dx + dy*dy

        // If player is bigger and close, they're a threat
        if playerSize > a.Size*1.2 { // 20% bigger is threatening
            // Threat increases as distance decreases and size difference increases
            distThreat := math.Max(0, 1.0 - math.Sqrt(float64(distSq))/float64(a.PerceptionRadi
            sizeThreat := math.Min(1.0, float64(playerSize/a.Size - 1.0))

            // Combine factors
            threatFactor := float32(distThreat * sizeThreat * 2.0)

            // Keep track of maximum threat
            if threatFactor > a.ThreatLevel {
                a.ThreatLevel = threatFactor
            }
        }
    }

    // Clamp threat level
    if a.ThreatLevel > 1.0 {
        a.ThreatLevel = 1.0
    }
}

// Main decision making logic
func (a *AIHelicopter) makeDecisions(game *game.Game) {
    // Choose next state based on current situation

    // First, handle immediate threats if cautious enough
    if a.ThreatLevel > 0.5 && rand.Float32() < a.Personality.CautionFactor {
        a.transitionToState(AIStateEvading)
        a.chooseEvadeTarget()
        return
    }

    // Check if current power-up use should continue
    if a.State == AIStateUsingPowerUp {
        // Stay in power-up using state if we still have active power-ups
        if len(a.ActivePowerUps) > 0 {
            // If we've been using power-ups too long, consider other activities
            if a.StateTimer > 3.0 && rand.Float32() < 0.3 {
                // Maybe do something else
                a.chooseNewState(game)
            }
            return
```

```go
        } else {
            // No more power-ups, transition to a new state
            a.chooseNewState(game)
            return
        }
    }

    // Check if we should continue current state
    if a.StateTimer < 1.0 + rand.Float32() * 2.0 {
        // Usually continue current state for 1-3 seconds unless something important happens

        // But check for high-priority opportunities

        // If we're not already pursuing a power-up and one is visible
        if a.State != AIStatePursuingPowerUp && len(a.KnownPowerUps) > 0 &&
            rand.Float32() < a.Personality.PowerUpPriority {
            a.transitionToState(AIStatePursuingPowerUp)
            a.choosePowerUpTarget()
            return
        }

        // If we're aggressive and see a smaller player
        if a.State != AIStateHunting && a.canHuntPlayers() &&
            rand.Float32() < a.Personality.AggressionFactor {
            a.transitionToState(AIStateHunting)
            a.chooseHuntTarget()
            return
        }

        // Otherwise continue in current state
        a.updateCurrentStateTarget(game)
        return
    }

    // Choose a completely new state
    a.chooseNewState(game)
}

// Choose a new state based on AI personality and situation
func (a *AIHelicopter) chooseNewState(game *game.Game) {
    // Create state weights based on personality and current situation
    weights := make(map[AIState]float32)

    // Base weights
    weights[AIStateWandering] = 0.2 * a.Personality.ExplorationFactor
    weights[AIStateCollecting] = 0.3 * a.Personality.CollectionEfficiency
    weights[AIStatePursuingPowerUp] = 0.1 * a.Personality.PowerUpPriority
```

```go
weights[AIStateHunting] = 0.1 * a.Personality.AggressionFactor
weights[AIStateEvading] = 0.1 * a.Personality.CautionFactor
weights[AIStateUsingPowerUp] = 0.1

// Adjust based on current game state

// If we know food locations, increase collection weight
if len(a.KnownFoodPositions) > 0 {
    weights[AIStateCollecting] += 0.3
}

// If we know power-up locations, increase power-up pursuit weight
if len(a.KnownPowerUps) > 0 {
    weights[AIStatePursuingPowerUp] += 0.5
}

// If we can hunt effectively, increase hunting weight
if a.canHuntPlayers() {
    weights[AIStateHunting] += 0.4
}

// If threat level is high, increase evading weight
if a.ThreatLevel > 0.3 {
    weights[AIStateEvading] += a.ThreatLevel * 0.7
}

// If we have active power-ups, increase using weight
if len(a.ActivePowerUps) > 0 {
    weights[AIStateUsingPowerUp] += 0.5
}

// Choose state based on weighted probabilities
totalWeight := float32(0)
for _, weight := range weights {
    totalWeight += weight
}

// Generate random value
rand := rand.Float32() * totalWeight

// Select state based on weights
cumulativeWeight := float32(0)
var selectedState AIState = AIStateWandering // Default

for state, weight := range weights {
    cumulativeWeight += weight
    if rand <= cumulativeWeight {
```

```go
            selectedState = state
            break
        }
    }

    // Transition to the selected state
    a.transitionToState(selectedState)

    // Set up appropriate target for the new state
    switch selectedState {
    case AIStateWandering:
        a.chooseWanderTarget(game)
    case AIStateCollecting:
        a.chooseCollectionTarget()
    case AIStatePursuingPowerUp:
        a.choosePowerUpTarget()
    case AIStateHunting:
        a.chooseHuntTarget()
    case AIStateEvading:
        a.chooseEvadeTarget()
    case AIStateUsingPowerUp:
        a.activatePowerUps()
    }
}

// Transition to a new state
func (a *AIHelicopter) transitionToState(newState AIState) {
    a.PreviousState = a.State
    a.State = newState
    a.StateTimer = 0
}

// Check if the AI is capable of hunting other players
func (a *AIHelicopter) canHuntPlayers() bool {
    // We need to know about other players
    if len(a.KnownPlayers) == 0 {
        return false
    }

    // Look for a suitable target (smaller player)
    for _, playerObj := range a.KnownPlayers {
        var playerSize float32

        switch p := playerObj.(type) {
        case *Player:
            playerSize = p.Size
        case *AIHelicopter:
```

```
            playerSize = p.Size
        default:
            continue
        }

        // If we find at least one smaller player, we can hunt
        if playerSize < a.Size * 0.9 {
            return true
        }
    }

    return false
}
```

4. Advanced Movement and Steering Behaviors

The AI needs to move naturally, not just teleport to targets. Let's implement steering behaviors:

go

```go
// Update movement based on current state and target
func (a *AIHelicopter) updateMovement(dt float32) {
    // Don't move if we don't have a target
    if a.Target.X == 0 && a.Target.Y == 0 {
        return
    }

    // Calculate basic steering toward target
    steering := a.calculateSteering()

    // Apply state-specific movement behaviors
    switch a.State {
    case AIStateWandering:
        // Wander more randomly, don't go straight to target
        wanderJitter := 0.3
        steering = a.applyWander(steering, wanderJitter)

    case AIStateEvading:
        // When evading, move faster
        steering.X *= 1.2
        steering.Y *= 1.2

    case AIStateHunting:
        // When hunting, anticipate target movement if target is a player
        if player, ok := a.TargetEntity.(*Player); ok {
            steering = a.calculatePursuit(player)
        } else if ai, ok := a.TargetEntity.(*AIHelicopter); ok {
            steering = a.calculatePursuit(&ai.Player)
        }

    case AIStateCollecting:
        // When collecting, steer toward best collection paths
        if len(a.KnownFoodPositions) > 3 {
            steering = a.calculateCollectionPath(steering)
        }

    case AIStatePursuingPowerUp:
        // When pursuing power-up, move more directly and faster
        steering.X *= 1.1
        steering.Y *= 1.1
    }

    // Apply obstacle avoidance (avoid other larger players)
    steering = a.avoidObstacles(steering)

    // Set final velocity
```

```go
    a.Velocity.X = steering.X * a.MaxSpeed
    a.Velocity.Y = steering.Y * a.MaxSpeed

    // Calculate rotation to face movement direction
    if steering.X != 0 || steering.Y != 0 {
        a.TargetRotation = float32(math.Atan2(float64(steering.Y), float64(steering.X))) * 180
    }
}

// Calculate basic steering toward target
func (a *AIHelicopter) calculateSteering() rl.Vector2 {
    // Direction to target
    desiredVelocity := rl.Vector2{
        X: a.Target.X - a.Position.X,
        Y: a.Target.Y - a.Position.Y,
    }

    // Normalize
    distance := float32(math.Sqrt(float64(desiredVelocity.X*desiredVelocity.X + desiredVelocity
    if distance > 0 {
        desiredVelocity.X /= distance
        desiredVelocity.Y /= distance
    }

    // If close to target, slow down
    if distance < 50 {
        scale := distance / 50
        desiredVelocity.X *= scale
        desiredVelocity.Y *= scale
    }

    return desiredVelocity
}

// Apply wander behavior - add some randomness to movement
func (a *AIHelicopter) applyWander(steering rl.Vector2, jitter float32) rl.Vector2 {
    // Add random displacement to steering
    steering.X += (rand.Float32()*2 - 1) * jitter
    steering.Y += (rand.Float32()*2 - 1) * jitter

    // Normalize
    length := float32(math.Sqrt(float64(steering.X*steering.X + steering.Y*steering.Y)))
    if length > 0 {
        steering.X /= length
        steering.Y /= length
    }
```

```go
    return steering
}

// Calculate pursuit (anticipate where target will be)
func (a *AIHelicopter) calculatePursuit(target *Player) rl.Vector2 {
    // Vector to target
    toTarget := rl.Vector2{
        X: target.Position.X - a.Position.X,
        Y: target.Position.Y - a.Position.Y,
    }

    // Distance to target
    distance := float32(math.Sqrt(float64(toTarget.X*toTarget.X + toTarget.Y*toTarget.Y)))

    // Time to reach target
    lookAheadTime := distance / a.MaxSpeed

    // Anticipate target's future position
    futurePosition := rl.Vector2{
        X: target.Position.X + target.Velocity.X*lookAheadTime,
        Y: target.Position.Y + target.Velocity.Y*lookAheadTime,
    }

    // Set new target to the anticipated position
    a.Target = futurePosition

    // Return steering toward that position
    return a.calculateSteering()
}

// Calculate path for collecting multiple food items efficiently
func (a *AIHelicopter) calculateCollectionPath(steering rl.Vector2) rl.Vector2 {
    // If we don't have enough known food positions, just use basic steering
    if len(a.KnownFoodPositions) < 3 {
        return steering
    }

    // Find the best collection path (simple greedy algorithm)
    // Start with closest food
    closestIdx := -1
    closestDistSq := float32(math.MaxFloat32)

    for i, foodPos := range a.KnownFoodPositions {
        dx := foodPos.X - a.Position.X
        dy := foodPos.Y - a.Position.Y
        distSq := dx*dx + dy*dy
```

```go
        if distSq < closestDistSq {
            closestDistSq = distSq
            closestIdx = i
        }
    }

    if closestIdx >= 0 {
        // Consider the next 2-3 food items based on current position
        a.PathPoints = make([]rl.Vector2, 0)
        a.PathPoints = append(a.PathPoints, a.KnownFoodPositions[closestIdx])

        // Add next closest food items
        currentPos := a.KnownFoodPositions[closestIdx]
        usedIndices := map[int]bool{closestIdx: true}

        // Add a few more points to the path
        for i := 0; i < 2 && len(usedIndices) < len(a.KnownFoodPositions); i++ {
            nextClosestIdx := -1
            nextClosestDistSq := float32(math.MaxFloat32)

            for j, foodPos := range a.KnownFoodPositions {
                if usedIndices[j] {
                    continue
                }

                dx := foodPos.X - currentPos.X
                dy := foodPos.Y - currentPos.Y
                distSq := dx*dx + dy*dy

                if distSq < nextClosestDistSq {
                    nextClosestDistSq = distSq
                    nextClosestIdx = j
                }
            }

            if nextClosestIdx >= 0 {
                a.PathPoints = append(a.PathPoints, a.KnownFoodPositions[nextClosestIdx])
                usedIndices[nextClosestIdx] = true
                currentPos = a.KnownFoodPositions[nextClosestIdx]
            }
        }

        // Now steer toward the first point in our path
        if len(a.PathPoints) > 0 {
            a.Target = a.PathPoints[0]
            return a.calculateSteering()
        }
```

```go
    }

    return steering
}

// Avoid obstacles (like larger players)
func (a *AIHelicopter) avoidObstacles(steering rl.Vector2) rl.Vector2 {
    // Look for obstacles (larger players)
    for _, playerObj := range a.KnownPlayers {
        var player *Player
        var position rl.Vector2
        var size float32
```