```
        // Score system
        Score:            0,
        TotalFoodCollected: 0,

        // PowerUp default values
        ActivePowerUps:   make(map[entities.PowerUpType]*PowerUpEffect),
        MagnetRange:      0, // No magnetic attraction by default
        SpeedMultiplier:  1.0, // Normal speed
        HasShield:        false,
        SizeBoostFactor:  1.0, // Normal size growth
    }

    // ... rest of initialization

    return player

}

// Add these methods to Player:

// ApplyPowerUp activates a power-up effect on the player
func (p *Player) ApplyPowerUp(powerUp *entities.PowerUp) {
// Create or update power-up effect
effect := &PowerUpEffect{
Type:          powerUp.Type,
RemainingTime: powerUp.Duration,
TotalDuration: powerUp.Duration,
}
```

```
    // Store in active power-ups map
    p.ActivePowerUps[powerUp.Type] = effect

    // Apply immediate effect based on type
    switch powerUp.Type {
    case entities.PowerUpMagnet:
        p.MagnetRange = 150.0 // Set attraction range

    case entities.PowerUpSpeed:
        p.SpeedMultiplier = 1.5 // 50% speed boost

    case entities.PowerUpShield:
        p.HasShield = true

    case entities.PowerUpSizeboost:
        p.SizeBoostFactor = 2.0 // Double size growth from food
    }

}

// UpdatePowerUps should be called every frame to update power-up effects
func (p *Player) UpdatePowerUps(dt float32) {
// Update all active power-up durations
for powerType, effect := range p.ActivePowerUps {
effect.RemainingTime -= dt
```

```
        // Remove expired power-ups
        if effect.RemainingTime <= 0 {
            delete(p.ActivePowerUps, powerType)

            // Reset effect based on type
            switch powerType {
            case entities.PowerUpMagnet:
                p.MagnetRange = 0

            case entities.PowerUpSpeed:
                p.SpeedMultiplier = 1.0

            case entities.PowerUpShield:
                p.HasShield = false

            case entities.PowerUpSizeboost:
                p.SizeBoostFactor = 1.0
            }
        }
    }

}

// Add to Update method:
func (p *Player) Update(dt float32) {
// ... existing update code
```

```
    // Update power-up effects
    p.UpdatePowerUps(dt)

    // Apply speed multiplier to velocity (after calculating base velocity)
    if p.SpeedMultiplier != 1.0 {
        p.Velocity.X *= p.SpeedMultiplier
        p.Velocity.Y *= p.SpeedMultiplier

        // Remember to reapply max speed limit after multiplier
        speed := float32(math.Sqrt(float64(p.Velocity.X*p.Velocity.X +
    p.Velocity.Y*p.Velocity.Y)))
        maxAdjustedSpeed := p.MaxSpeed * p.SpeedMultiplier
        if speed > maxAdjustedSpeed {
            p.Velocity.X = (p.Velocity.X / speed) * maxAdjustedSpeed
            p.Velocity.Y = (p.Velocity.Y / speed) * maxAdjustedSpeed
        }
    }

    // ... remaining update code

}

// Add to Draw method:
func (p *Player) Draw() {
// ... existing drawing code
```

```
// Draw active power-up effects

// Draw shield if active
if p.HasShield {
    shieldSize := p.Size * 1.3
    shieldColor := rl.SkyBlue

    // Make shield pulse
    effect, exists := p.ActivePowerUps[entities.PowerUpShield]
    if exists {
        // Calculate pulse based on remaining time
        pulseFreq := 2.0 + (1.0 - effect.RemainingTime/effect.TotalDuration) * 5.0
        pulseAmount := math.Sin(float64(p.GameTime) * pulseFreq)

        // Adjust alpha based on pulse and remaining time
        alpha := 128 + int(70*pulseAmount)
        if effect.RemainingTime < 1.0 {
            alpha = int(float32(alpha) * effect.RemainingTime)
        }

        shieldColor.A = uint8(alpha)
    }

    rl.DrawCircleLinesEx(p.Position, shieldSize, 2.0, shieldColor)
}

// Draw magnet field if active
if p.MagnetRange > 0 {
    effect, exists := p.ActivePowerUps[entities.PowerUpMagnet]
    if exists {
        magnetColor := rl.Purple

        // Make field pulse and fade based on remaining time
        alpha := 40 + int(15*math.Sin(float64(p.GameTime)*3.0))
        if effect.RemainingTime < 1.0 {
            alpha = int(float32(alpha) * effect.RemainingTime)
        }

        magnetColor.A = uint8(alpha)

        // Draw magnetic field as a circle
        rl.DrawCircleV(p.Position, p.MagnetRange, magnetColor)
    }
}

// Visual indicator for speed boost
```

```
if p.SpeedMultiplier > 1.0 {
    effect, exists := p.ActivePowerUps[entities.PowerUpSpeed]
    if exists {
        // Draw speed lines behind helicopter
        speedLineCount := 8
        speedLineLength := p.Size * 1.5

        for i := 0; i < speedLineCount; i++ {
            angle := float32(i) * (2 * math.Pi
    distance := rand.Float32() * size * 0.8

    // Calculate initial position (slightly randomized from center)
    posX := position.X + float32(math.Cos(float64(angle))) * distance * 0.2
    posY := position.Y + float32(math.Sin(float64(angle))) * distance * 0.2

    // Calculate velocity (outward from center)
    speed := 50.0 + rand.Float32() * 100.0 * (size / 30.0) // Scale with explosion size
    velX := float32(math.Cos(float64(angle))) * speed
    velY := float32(math.Sin(float64(angle))) * speed

    // Randomize color slightly
    r := int(color.R) + rand.Intn(40) - 20
    g := int(color.G) + rand.Intn(40) - 20
    b := int(color.B) + rand.Intn(40) - 20

    if r < 0 {
        r = 0
    } else if r > 255 {
        r = 255
    }

    if g < 0 {
        g = 0
    } else if g > 255 {
        g = 255
    }

    if b < 0 {
        b = 0
    } else if b > 255 {
        b = 255
    }

    // Randomize lifetime
    lifetime := 0.5 + rand.Float32() * 0.5 // 0.5-1.0 seconds

    particles[i] = ExplosionParticle{
```

```
            Position:    rl.Vector2{X: posX, Y: posY},
            Velocity:    rl.Vector2{X: velX, Y: velY},
            Color:       rl.Color{R: uint8(r), G: uint8(g), B: uint8(b), A: 255},
            Size:        1.0 + rand.Float32() * 3.0 * (size / 30.0), // Scale with explosion
    size
            Lifetime:    lifetime,
            MaxLifetime: lifetime,
            Active:      true,
        }
    }

    return &Explosion{
        Position:      position,
        Particles:     particles,
        TimeExisting:  0,
        Duration:      1.0, // 1 second total explosion duration
        ShockwaveSize: 0,
        MaxShockwave:  size * 1.5,
        Color:         color,
        Active:        true,
    }

}
```

```
func (e *Explosion) Update(dt float32) {
if !e.Active {
return
}
```

```
// Update explosion lifetime
e.TimeExisting += dt

// Check if explosion is complete
if e.TimeExisting >= e.Duration {
    e.Active = false
    return
}

// Update shockwave
progressRatio := e.TimeExisting / e.Duration

// Shockwave grows quickly then fades
if progressRatio < 0.3 {
    // Grow phase (0-30% of duration)
    e.ShockwaveSize = e.MaxShockwave * (progressRatio / 0.3)
} else {
    // Fade phase (30-100% of duration)
    e.ShockwaveSize = e.MaxShockwave
}

// Update particles
for i := range e.Particles {
    if e.Particles[i].Active {
        // Update position
        e.Particles[i].Position.X += e.Particles[i].Velocity.X * dt
        e.Particles[i].Position.Y += e.Particles[i].Velocity.Y * dt

        // Apply drag to slow particles
        e.Particles[i].Velocity.X *= 0.95
        e.Particles[i].Velocity.Y *= 0.95

        // Update lifetime
        e.Particles[i].Lifetime -= dt

        // Deactivate expired particles
        if e.Particles[i].Lifetime <= 0 {
            e.Particles[i].Active = false
        } else {
            // Update alpha based on remaining life
            lifeRatio := e.Particles[i].Lifetime / e.Particles[i].MaxLifetime
            e.Particles[i].Color.A = uint8(255 * lifeRatio)
        }
    }
}
```

```
}

func (e *Explosion) Draw() {
if !e.Active {
return
}

  // Draw shockwave
  shockwaveAlpha := uint8(255 * (1.0 - e.TimeExisting/e.Duration))
  shockwaveColor := rl.Color{
      R: e.Color.R,
      G: e.Color.G,
      B: e.Color.B,
      A: shockwaveAlpha,
  }

  rl.DrawCircleLinesEx(e.Position, e.ShockwaveSize, 3.0, shockwaveColor)

  // Draw central flash (bright at start, fades quickly)
  if e.TimeExisting < e.Duration * 0.3 {
      flashAlpha := uint8(255 * (1.0 - e.TimeExisting/(e.Duration*0.3)))
      flashColor := rl.White
      flashColor.A = flashAlpha

      // Draw central circle with size proportional to shockwave
      rl.DrawCircleV(e.Position, e.ShockwaveSize * 0.3, flashColor)
  }

  // Draw particles
  for i := range e.Particles {
      if e.Particles[i].Active {
          rl.DrawCircleV(e.Particles[i].Position, e.Particles[i].Size, e.Particles[i].Color)
      }
  }

}
```

```go
// systems/effects_system.go
package systems

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

// EffectsSystem manages visual effects like explosions and text popups
type EffectsSystem struct {
    Explosions []*entities.Explosion

    // Could also add other visual effects like:
    // TextPopups []TextPopup
    // FoodTrails []FoodTrail
}

func NewEffectsSystem() *EffectsSystem {
    return &EffectsSystem{
        Explosions: make([]*entities.Explosion, 0, 20),
    }
}

func (es *EffectsSystem) Update(dt float32) {
    // Update explosions
    for i := len(es.Explosions) - 1; i >= 0; i-- {
        es.Explosions[i].Update(dt)

        // Remove inactive explosions
        if !es.Explosions[i].Active {
            // Remove using swap and pop (order doesn't matter)
            es.Explosions[i] = es.Explosions[len(es.Explosions)-1]
            es.Explosions = es.Explosions[:len(es.Explosions)-1]
        }
    }

    // Update other effect types here
}

func (es *EffectsSystem) Draw() {
    // Draw explosions
    for _, explosion := range es.Explosions {
        explosion.Draw()
```

```
    }

    // Draw other effect types here
}

// CreateExplosion adds a new explosion effect
func (es *EffectsSystem) CreateExplosion(position rl.Vector2, size float32, color rl.Color)
{
    explosion := entities.NewExplosion(position, size, color)
    es.Explosions = append(es.Explosions, explosion)
}

// CreateFoodPickupEffect creates visual effect when food is collected
func (es *EffectsSystem) CreateFoodPickupEffect(position rl.Vector2, color rl.Color) {
    // Create a small particle burst
    smallExplosion := entities.NewExplosion(position, 10.0, color)
    es.Explosions = append(es.Explosions, smallExplosion)
}

// CreatePowerUpPickupEffect creates a bright effect when a power-up is collected
func (es *EffectsSystem) CreatePowerUpPickupEffect(position rl.Vector2, color rl.Color) {
    // Create a larger, flashier explosion for power-ups
    explosion := entities.NewExplosion(position, 25.0, color)
    es.Explosions = append(es.Explosions, explosion)
}
```

5. Leaderboard System Implementation

go

```go
// ui/leaderboard.go
package ui

import (
    "github.com/gen2brain/raylib-go/raylib"
    "sort"
    "fmt"
)

// Player information for the leaderboard
type LeaderboardEntry struct {
    Name  string
    Score int
    Size  float32
    IsPlayer bool
}

type Leaderboard struct {
    Entries       []LeaderboardEntry
    Position      rl.Vector2
    Size          rl.Vector2
    BorderColor   rl.Color
    BackgroundColor rl.Color
    TextColor     rl.Color
    HighlightColor rl.Color
    Visible       bool
    MaxEntries    int
}

func NewLeaderboard() *Leaderboard {
    return &Leaderboard{
        Entries:       make([]LeaderboardEntry, 0, 10),
        Position:      rl.Vector2{X: 20, Y: 100},
        Size:          rl.Vector2{X: 200, Y: 300},
        BorderColor:   rl.Gray,
        BackgroundColor: rl.Color{R: 0, G: 0, B: 0, A: 180},
        TextColor:     rl.White,
        HighlightColor: rl.Yellow,
        Visible:       true,
        MaxEntries:    8,
    }
}

// AddEntry adds or updates a player entry
func (lb *Leaderboard) AddEntry(name string, score int, size float32, isPlayer bool) {
    // Check if player already exists
```

```
    for i, entry := range lb.Entries {
        if entry.Name == name {
            // Update existing entry
            lb.Entries[i].Score = score
            lb.Entries[i].Size = size
            lb.Entries[i].IsPlayer = isPlayer
            return
        }
    }

    // Add new entry
    lb.Entries = append(lb.Entries, LeaderboardEntry{
        Name:     name,
        Score:    score,
        Size:     size,
        IsPlayer: isPlayer,
    })
}

// Sort entries by score
func (lb *Leaderboard) SortEntries() {
    sort.Slice(lb.Entries, func(i, j int) bool {
        return lb.Entries[i].Score > lb.Entries[j].Score
    })
}

func (lb *Leaderboard) Draw() {
    if !lb.Visible {
        return
    }

    // Sort entries before drawing
    lb.SortEntries()

    // Draw background
    rl.DrawRectangleV(lb.Position, lb.Size, lb.BackgroundColor)
    rl.DrawRectangleLinesEx(
        rl.Rectangle{
            X:      lb.Position.X,
            Y:      lb.Position.Y,
            Width:  lb.Size.X,
            Height: lb.Size.Y,
        },
        2,
        lb.BorderColor,
    )
```

```
// Draw title
rl.DrawText("LEADERBOARD", int32(lb.Position.X + 10), int32(lb.Position.Y + 10), 20, lb.Tex

// Draw header separators
rl.DrawLine(
    int32(lb.Position.X),
    int32(lb.Position.Y + 40),
    int32(lb.Position.X + lb.Size.X),
    int32(lb.Position.Y + 40),
    lb.BorderColor,
)

// Draw column headers
rl.DrawText("RANK", int32(lb.Position.X + 10), int32(lb.Position.Y + 45), 16, lb.TextColor)
rl.DrawText("NAME", int32(lb.Position.X + 60), int32(lb.Position.Y + 45), 16, lb.TextColor)
rl.DrawText("SCORE", int32(lb.Position.X + 150), int32(lb.Position.Y + 45), 16, lb.TextColo

// Draw separator
rl.DrawLine(
    int32(lb.Position.X),
    int32(lb.Position.Y + 65),
    int32(lb.Position.X + lb.Size.X),
    int32(lb.Position.Y + 65),
    lb.BorderColor,
)

// Draw entries
entryCount := len(lb.Entries)
if entryCount > lb.MaxEntries {
    entryCount = lb.MaxEntries
}

for i := 0; i < entryCount; i++ {
    entry := lb.Entries[i]
    yPos := lb.Position.Y + 75 + float32(i*25)

    // Choose color based on whether this is the player
    textColor := lb.TextColor
    if entry.IsPlayer {
        textColor = lb.HighlightColor
    }

    // Draw rank
    rl.DrawText(fmt.Sprintf("#%d", i+1), int32(lb.Position.X + 10), int32(yPos), 16, textCo

    // Draw name (truncate if too long)
    name := entry.Name
```

```
            if len(name) > 10 {
                name = name[:7] + "..."
            }
            rl.DrawText(name, int32(lb.Position.X + 60), int32(yPos), 16, textColor)

            // Draw score
            rl.DrawText(fmt.Sprintf("%d", entry.Score), int32(lb.Position.X + 150), int32(yPos), 16
        }
    }
```

6. Integration with Main Game

To integrate all these systems, we need to add them to the main game structure and initialize them properly. Here's how to update the game/game.go file:

go

```go
// game/game.go (updates)

// In the Game struct, add:
type Game struct {
    // ... existing fields

    // Systems
    FoodGenerator   *systems.FoodGenerator
    PowerUpManager  *systems.PowerUpManager
    EffectsSystem   *systems.EffectsSystem

    // UI elements
    Leaderboard     *ui.Leaderboard

    // Game state tracking
    GameTime        float32

    // AI players for leaderboard
    AIPlayers       []string
    AIScores        []int
    AIUpdateTimer   float32
}

// In the New() function, initialize the new systems:
func New() *Game {
    // ... existing initialization

    // Create food generator with world bounds
    foodGenerator := systems.NewFoodGenerator(rl.Rectangle{
        X:      0,
        Y:      0,
        Width:  constants.WorldWidth,
        Height: constants.WorldHeight,
    })

    // Initialize with starting food
    foodGenerator.Initialize(300)

    // Create power-up manager
    powerUpManager := systems.NewPowerUpManager(rl.Rectangle{
        X:      0,
        Y:      0,
        Width:  constants.WorldWidth,
        Height: constants.WorldHeight,
    })
```

```
// Create effects system
effectsSystem := systems.NewEffectsSystem()

// Create leaderboard
leaderboard := ui.NewLeaderboard()

// Generate some AI player names for the leaderboard
aiNames := []string{
    "HeliZapper", "RotorRider", "ChopperChamp",
    "AirPirate", "SkyRanger", "BladeRunner",
    "ThunderBird", "WhirlyBird", "AirWolf",
    "FlyingAce", "SkyHunter", "VerticalThreat"
}

// Randomly select 8 AI names
rand.Shuffle(len(aiNames), func(i, j int) {
    aiNames[i], aiNames[j] = aiNames[j], aiNames[i]
})

selectedAINames := aiNames[:8]
aiScores := make([]int, 8)

// Initialize AI scores with random values
for i := range aiScores {
    aiScores[i] = 500 + rand.Intn(2000)
}

// Add player and AI entries to leaderboard
leaderboard.AddEntry("Player", 0, 20.0, true)
for i, name := range selectedAINames {
    leaderboard.AddEntry(name, aiScores[i], 20.0 + float32(aiScores[i])/100, false)
}

game := &Game{
    Player:         entities.NewPlayer(),
    Camera:         NewCamera(player),
    Quadtree:       util.NewQuadtree(...),
    FoodGenerator:  foodGenerator,
    PowerUpManager: powerUpManager,
    EffectsSystem:  effectsSystem,
    Leaderboard:    leaderboard,
    Minimap:        ui.NewMinimap(),
    GameTime:       0,
    AIPlayers:      selectedAINames,
    AIScores:       aiScores,
    AIUpdateTimer:  0,
}
```

```
    return game
}

// Update the Update method to include the new systems:
func (g *Game) Update(dt float32) {
    // Update game time
    g.GameTime += dt
    g.Player.GameTime = g.GameTime

    // Reset quadtree
    g.Quadtree.Clear()

    // Add player to quadtree
    g.Quadtree.Root.Insert(g.Player, g.Player)

    // Update food system
    g.FoodGenerator.Update(dt, g.Quadtree)
    for _, food := range g.FoodGenerator.FoodEntities {
        g.Quadtree.Root.Insert(food, food)
    }

    // Update power-up manager
    g.PowerUpManager.Update(dt, g.Quadtree)
    for _, powerUp := range g.PowerUpManager.PowerUps {
        g.Quadtree.Root.Insert(powerUp, powerUp)
    }

    // Update player
    g.Player.Update(dt)

    // Apply magnetic attraction to food (if power-up is active)
    if g.Player.MagnetRange > 0 {
        for _, food := range g.FoodGenerator.FoodEntities {
            force := g.Player.AttractionForce(food.Position)
            food.Position.X += force.X * dt
            food.Position.Y += force.Y * dt
        }
    }

    // Check for collisions
    g.CheckPlayerFoodCollisions()
    g.CheckPlayerPowerUpCollisions()

    // Update visual effects
    g.EffectsSystem.Update(dt)
```

```
    // Update camera
    g.Camera.Update(dt)

    // Update AI players (simulate their progress)
    g.AIUpdateTimer += dt
    if g.AIUpdateTimer >= 1.0 { // Update AI scores every second
        g.AIUpdateTimer = 0

        // Update AI scores
        for i := range g.AIScores {
            // Some AIs gain points, some might lose
            scoreChange := rand.Intn(50) - 10
            g.AIScores[i] += scoreChange

            // Ensure minimum score
            if g.AIScores[i] < 500 {
                g.AIScores[i] = 500 + rand.Intn(100)
            }

            // Update leaderboard for this AI
            g.Leaderboard.AddEntry(
                g.AIPlayers[i],
                g.AIScores[i],
                20.0 + float32(g.AIScores[i])/100,
                false,
            )
        }
    }

    // Update player entry in leaderboard
    g.Leaderboard.AddEntry("Player", g.Player.Score, g.Player.Size, true)
}

// Update the Draw method to include the new systems:
func (g *Game) Draw() {
    rl.BeginDrawing()
    rl.ClearBackground(rl.Black)

    // Begin 2D mode with camera
    rl.BeginMode2D(g.Camera.GetRLCamera2D())

    // Draw world background
    // ... (existing grid drawing code)

    // Draw food entities
    for _, food := range g.FoodGenerator.FoodEntities {
        // Use culling logic to only draw food near viewport
```

```
        viewportBounds := rl.Rectangle{
            X:      g.Camera.Position.X,
            Y:      g.Camera.Position.Y,
            Width:  constants.ScreenWidth / g.Camera.Zoom,
            Height: constants.ScreenHeight / g.Camera.Zoom,
        }

        // Add padding for items just offscreen
        padding := 100.0
        viewportWithPadding := rl.Rectangle{
            X:      viewportBounds.X - padding,
            Y:      viewportBounds.Y - padding,
            Width:  viewportBounds.Width + padding*2,
            Height: viewportBounds.Height + padding*2,
        }

        if util.CheckRectangleOverlap(food.GetBounds(), viewportWithPadding) {
            food.Draw()
        }
    }

    // Draw power-ups
    for _, powerUp := range g.PowerUpManager.PowerUps {
        // Use same culling logic as food
        viewportBounds := rl.Rectangle{
            X:      g.Camera.Position.X,
            Y:      g.Camera.Position.Y,
            Width:  constants.ScreenWidth / g.Camera.Zoom,
            Height: constants.ScreenHeight / g.Camera.Zoom,
        }

        padding := 100.0
        viewportWithPadding := rl.Rectangle{
            X:      viewportBounds.X - padding,
            Y:      viewportBounds.Y - padding,
            Width:  viewportBounds.Width + padding*2,
            Height: viewportBounds.Height + padding*2,
        }

        if util.CheckRectangleOverlap(powerUp.GetBounds(), viewportWithPadding) {
            powerUp.Draw()
        }
    }

    // Draw visual effects
    g.EffectsSystem.Draw()
```

```
    // Draw player
    g.Player.Draw()

    rl.EndMode2D()

    // Draw UI elements

    // Draw score
    rl.DrawText(fmt.Sprintf("Score: %d", g.Player.Score), 20, 20, 20, rl.White)

    // Draw size indicator
    rl.DrawText(fmt.Sprintf("Size: %.1f", g.Player.Size), 20, 50, 20, rl.White)

    // Draw active power-ups UI
    g.DrawPowerUpUI()

    // Draw leaderboard
    g.Leaderboard.Draw()

    // Draw minimap
    g.Minimap.Draw(g)

    rl.EndDrawing()
}
```

This completes the implementation plan for Phase 2, which includes:

1. An enhanced food system with different food types and values
2. A comprehensive power-up system with four distinct power-ups
3. Visual effects for explosions and feedback
4. A leaderboard system with AI players
5. Growth mechanics for the player helicopter

By implementing these features, the game will have the core mechanics that make .io games addictive: collecting items to grow, competing on a leaderboard, and using power-ups for temporary advantages.
/ float32(speedLineCount))
offsetX := -math.Cos(float64(angle)) * float64(speedLineLength)
offsetY := -math.Sin(float64(angle)) * float64(speedLineLength)

```
            startPos := rl.Vector2{
                X: p.Position.X + float32(offsetX)*0.2,
                Y: p.Position.Y + float32(offsetY)*0.2,
            }

            endPos := rl.Vector2{
                X: p.Position.X + float32(offsetX),
                Y: p.Position.Y + float32(offsetY),
            }

            // Flash the lines based on remaining time
            alpha := 180 + int(75*math.Sin(float64(p.GameTime)*8.0))
            if effect.RemainingTime < 1.0 {
                alpha = int(float32(alpha) * effect.RemainingTime)
            }

            lineColor := rl.Red
            lineColor.A = uint8(alpha)

            rl.DrawLineEx(startPos, endPos, 2.0, lineColor)
        }
    }
}

// Visual indicator for size boost
if p.SizeBoostFactor > 1.0 {
    effect, exists := p.ActivePowerUps[entities.PowerUpSizeboost]
    if exists {
        // Draw growing circles around helicopter
        pulseFreq := 2.0
        maxRings := 3

        for i := 0; i < maxRings; i++ {
            // Stagger the ring animations
            ringPhase := float32(i) * 0.33
            ringTime := math.Mod(float64(p.GameTime*pulseFreq+ringPhase), 1.0)

            // Ring grows from player size to max range then disappears
            ringSize := p.Size * (1.0 + float32(ringTime)*2.0)

            // Fade out as ring expands
            alpha := 255 - uint8(ringTime*255)

            // Further reduce alpha when power-up is about to expire
            if effect.RemainingTime < 1.0 {
                alpha = uint8(float32(alpha) * effect.RemainingTime)
```

```
                }

                ringColor := rl.Orange
                ringColor.A = alpha

                rl.DrawCircleLinesEx(p.Position, ringSize, 2.0, ringColor)
            }
        }
    }

}

// GrowFromFood handles size growth when consuming food
func (p *Player) GrowFromFood(food *entities.Food) {
// Apply food value to score
p.Score += food.Value
p.TotalFoodCollected++
```

```
    // Calculate size growth (with boost factor if power-up is active)
    growthAmount := float32(food.Value) * p.GrowthMultiplier * p.SizeBoostFactor

    // Apply growth with diminishing returns for higher sizes
    if p.Size < 30 {
        // Full growth at small sizes
        p.Size += growthAmount
    } else if p.Size < 60 {
        // 75% efficiency at medium sizes
        p.Size += growthAmount * 0.75
    } else {
        // 50% efficiency at large sizes
        p.Size += growthAmount * 0.5
    }

    // Cap at maximum size
    if p.Size > p.MaxSize {
        p.Size = p.MaxSize
    }

    // As player grows, slightly increase acceleration but reduce max rotation speed
    p.Acceleration = 500.0 + (p.Size - p.BaseSize) * 2.0
    p.RotationSpeed = 5.0 - (p.Size - p.BaseSize) / 40.0

    if p.RotationSpeed < 1.0 {
        p.RotationSpeed = 1.0 // Minimum rotation speed
    }

    // Adjust rotor sizes based on helicopter size
    p.MainRotor.Size = 30.0 * (p.Size / p.BaseSize)
    p.TailRotor.Size = 15.0 * (p.Size / p.BaseSize)

}

// Helper methods for power-up and food attraction

// GetActivePowerUpTimeRemaining returns the time remaining for a specific power-up
func (p *Player) GetActivePowerUpTimeRemaining(powerType entities.PowerUpType) float32 {
effect, exists := p.ActivePowerUps[powerType]
if exists {
return effect.RemainingTime
}
return 0
}
```

```
// GetPowerUpEffectProgress returns a value 0-1 indicating remaining power-up duration
func (p *Player) GetPowerUpEffectProgress(powerType entities.PowerUpType) float32 {
effect, exists := p.ActivePowerUps[powerType]
if exists && effect.TotalDuration > 0 {
return effect.RemainingTime / effect.TotalDuration
}
return 0
}

// HasActivePowerUp checks if a specific power-up is currently active
func (p *Player) HasActivePowerUp(powerType entities.PowerUpType) bool {
_, exists := p.ActivePowerUps[powerType]
return exists
}

// AttractionForce calculates magnetic attraction force to a point
func (p *Player) AttractionForce(targetPos rl.Vector2) rl.Vector2 {
// If magnet power-up isn't active, no attraction
if p.MagnetRange <= 0 {
return rl.Vector2{}
}
```

```
    // Calculate direction and distance
    dx := targetPos.X - p.Position.X
    dy := targetPos.Y - p.Position.Y
    distSq := dx*dx + dy*dy

    // If outside magnet range, no attraction
    if distSq > p.MagnetRange*p.MagnetRange {
        return rl.Vector2{}
    }

    // Calculate force based on distance (stronger closer to player)
    dist := float32(math.Sqrt(float64(distSq)))
    forceMagnitude := 1.0 - (dist / p.MagnetRange) // 1.0 at center, 0.0 at edge
    forceMagnitude = forceMagnitude * forceMagnitude * 100.0 // Square for non-linear falloff

    // Calculate normalized direction
    var force rl.Vector2
    if dist > 0 {
        force.X = dx / dist * forceMagnitude
        force.Y = dy / dist * forceMagnitude
    }

    return force

}
```

3.2 Game System Integration

```go
// game/game.go (modifications)

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

    PowerUpManager *systems.PowerUpManager
    FoodGenerator  *systems.FoodGenerator
    Effects        *systems.EffectsSystem // Optional visual effects system

    // Game state tracking
    GameTime       float32
}

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

    // Create food generator
    foodGenerator := systems.NewFoodGenerator(rl.Rectangle{
        X:      0,
        Y:      0,
        Width:  constants.WorldWidth,
        Height: constants.WorldHeight,
    })

    // Initialize with some food
    foodGenerator.Initialize(300) // Start with 300 food items

    // Create power-up manager
    powerUpManager := systems.NewPowerUpManager(rl.Rectangle{
        X:      0,
        Y:      0,
        Width:  constants.WorldWidth,
        Height: constants.WorldHeight,
    })

    // ... rest of initialization

    return &Game{
        // ... other fields
        FoodGenerator:  foodGenerator,
```

```
        PowerUpManager: powerUpManager,
        GameTime:       0,
    }
}

// Modify Update method:
func (g *Game) Update(dt float32) {
    // Update game time
    g.GameTime += dt
    g.Player.GameTime = g.GameTime // Share game time with player for animations

    // Reset quadtree each frame
    g.Quadtree.Clear()

    // Add player to quadtree
    g.Quadtree.Root.Insert(g.Player, g.Player)

    // Update and add food entities to quadtree
    g.FoodGenerator.Update(dt, g.Quadtree)
    for _, food := range g.FoodGenerator.FoodEntities {
        g.Quadtree.Root.Insert(food, food)
    }

    // Update and add power-ups to quadtree
    g.PowerUpManager.Update(dt, g.Quadtree)
    for _, powerUp := range g.PowerUpManager.PowerUps {
        g.Quadtree.Root.Insert(powerUp, powerUp)
    }

    // Update player
    g.Player.Update(dt)

    // Apply magnetic attraction to food (if power-up is active)
    if g.Player.MagnetRange > 0 {
        for _, food := range g.FoodGenerator.FoodEntities {
            // Calculate attraction force
            force := g.Player.AttractionForce(food.Position)

            // Apply force to food position
            food.Position.X += force.X * dt
            food.Position.Y += force.Y * dt
        }
    }

    // Check for player-food collisions
    g.CheckPlayerFoodCollisions()
```

```
    // Check for player-powerup collisions
    g.CheckPlayerPowerUpCollisions()

    // Update camera
    g.Camera.Update(dt)
}

// Add collision detection methods:

func (g *Game) CheckPlayerFoodCollisions() {
    // Get potential food collisions from quadtree
    playerBounds := g.Player.GetBounds()
    potentialCollisions := make([]interface{}, 0, 20)
    g.Quadtree.Root.Query(playerBounds, &potentialCollisions)

    // Check actual collisions
    for _, potential := range potentialCollisions {
        if food, ok := potential.(*entities.Food); ok {
            // Simple circle collision
            playerPos := g.Player.Position
            foodPos := food.Position

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

            // Compare against squared distance
            minDist := g.Player.Size + food.Size

            if distSq < minDist*minDist {
                // Collision! Player eats the food
                g.FoodGenerator.RemoveFood(food)

                // Apply growth effect
                g.Player.GrowFromFood(food)

                // Play sound effect (if implemented)
                // g.audio.PlaySound("pickup.wav")

                // Create pickup visual effect
                // g.Effects.CreateFoodPickupEffect(food.Position, food.Color)
            }
        }
    }
}

func (g *Game) CheckPlayerPowerUpCollisions() {
```

```
    // Get potential power-up collisions from quadtree
    playerBounds := g.Player.GetBounds()
    potentialCollisions := make([]interface{}, 0, 10)
    g.Quadtree.Root.Query(playerBounds, &potentialCollisions)

    // Check actual collisions
    for _, potential := range potentialCollisions {
        if powerUp, ok := potential.(*entities.PowerUp); ok {
            // Simple circle collision
            playerPos := g.Player.Position
            powerUpPos := powerUp.Position

            dx := playerPos.X - powerUpPos.X
            dy := playerPos.Y - powerUpPos.Y
            distSq := dx*dx + dy*dy

            // Compare against squared distance
            minDist := g.Player.Size + powerUp.Size

            if distSq < minDist*minDist && powerUp.Active {
                // Collision! Player collects the power-up
                g.PowerUpManager.RemovePowerUp(powerUp)

                // Apply power-up effect
                g.Player.ApplyPowerUp(powerUp)

                // Play power-up sound
                // g.audio.PlaySound("powerup.wav")

                // Create power-up visual effect
                // g.Effects.CreatePowerUpPickupEffect(powerUp.Position, powerUp.Color)
            }
        }
    }
}

// Modify Draw method to include power-up rendering:
func (g *Game) Draw() {
    rl.BeginDrawing()
    rl.ClearBackground(rl.Black)

    // Begin 2D camera mode
    rl.BeginMode2D(g.Camera.GetRLCamera2D())

    // ... existing drawing code

    // Draw food entities
```

```
for _, food := range g.FoodGenerator.FoodEntities {
    // Only draw if within or near camera view (similar to your previous culling code)
    viewportBounds := rl.Rectangle{
        X:      g.Camera.Position.X,
        Y:      g.Camera.Position.Y,
        Width:  constants.ScreenWidth,
        Height: constants.ScreenHeight,
    }

    // Add padding to viewport for items just offscreen
    padding := 100.0
    viewportWithPadding := rl.Rectangle{
        X:      viewportBounds.X - padding,
        Y:      viewportBounds.Y - padding,
        Width:  viewportBounds.Width + padding*2,
        Height: viewportBounds.Height + padding*2,
    }

    if util.CheckRectangleOverlap(food.GetBounds(), viewportWithPadding) {
        food.Draw()
    }
}

// Draw power-ups
for _, powerUp := range g.PowerUpManager.PowerUps {
    // Apply the same viewport culling as for food
    viewportBounds := rl.Rectangle{
        X:      g.Camera.Position.X,
        Y:      g.Camera.Position.Y,
        Width:  constants.ScreenWidth,
        Height: constants.ScreenHeight,
    }

    padding := 100.0
    viewportWithPadding := rl.Rectangle{
        X:      viewportBounds.X - padding,
        Y:      viewportBounds.Y - padding,
        Width:  viewportBounds.Width + padding*2,
        Height: viewportBounds.Height + padding*2,
    }

    if util.CheckRectangleOverlap(powerUp.GetBounds(), viewportWithPadding) {
        powerUp.Draw()
    }
}

// Draw player (already includes power-up effects from our earlier modifications)
```

```
    g.Player.Draw()

    rl.EndMode2D()

    // Draw UI elements

    // Draw score
    rl.DrawText(TextFormat("Score: %d", g.Player.Score), 20, 20, 20, rl.White)

    // Draw size indicator
    rl.DrawText(TextFormat("Size: %.1f", g.Player.Size), 20, 50, 20, rl.White)

    // Draw active power-ups UI
    g.DrawPowerUpUI()

    // Draw minimap
    g.Minimap.Draw(g)

    rl.EndDrawing()
}

// Add PowerUp UI rendering:
func (g *Game) DrawPowerUpUI() {
    // Draw active power-up indicators at the bottom of the screen
    uiY := constants.ScreenHeight - 70
    iconSize := float32(50)
    padding := float32(10)
    startX := constants.ScreenWidth/2 - (iconSize*4 + padding*3)/2

    // Background for power-up area
    rl.DrawRectangle(
        int32(startX - padding),
        int32(uiY - padding),
        int32(iconSize*4 + padding*5),
        int32(iconSize + padding*2),
        rl.Color{R: 0, G: 0, B: 0, A: 128},
    )

    // Draw each power-up slot
    powerUpTypes := []entities.PowerUpType{
        entities.PowerUpMagnet,
        entities.PowerUpSpeed,
        entities.PowerUpShield,
        entities.PowerUpSizeboost,
    }

    for i, powerType := range powerUpTypes {
```

```
slotX := startX + float32(i)*(iconSize+padding)
slotRect := rl.Rectangle{
    X:      slotX,
    Y:      uiY,
    Width:  iconSize,
    Height: iconSize,
}

// Draw slot background
rl.DrawRectangleRec(slotRect, rl.Color{R: 20, G: 20, B: 20, A: 200})

// Check if this power-up is active
if g.Player.HasActivePowerUp(powerType) {
    // Get progress (0-1) of power-up duration
    progress := g.Player.GetPowerUpEffectProgress(powerType)

    // Draw power-up icon (simplified version of the in-world icon)
    var iconColor rl.Color

    switch powerType {
    case entities.PowerUpMagnet:
        iconColor = rl.Purple
    case entities.PowerUpSpeed:
        iconColor = rl.Red
    case entities.PowerUpShield:
        iconColor = rl.SkyBlue
    case entities.PowerUpSizeboost:
        iconColor = rl.Orange
    }

    // Draw icon
    iconCenter := rl.Vector2{
        X: slotRect.X + slotRect.Width/2,
        Y: slotRect.Y + slotRect.Height/2,
    }

    iconSize := slotRect.Width * 0.6

    // Draw simple icon based on power-up type
    switch powerType {
    case entities.PowerUpMagnet:
        rl.DrawCircleLinesEx(iconCenter, iconSize/2, 2, iconColor)
        rl.DrawRectangle(
            int32(iconCenter.X - iconSize/8),
            int32(iconCenter.Y - iconSize/3),
            int32(iconSize/4),
            int32(iconSize/1.5),
```

```
            iconColor,
        )

    case entities.PowerUpSpeed:
        // Draw lightning bolt
        rl.DrawTriangle(
            rl.Vector2{X: iconCenter.X - iconSize/3, Y: iconCenter.Y - iconSize/2},
            rl.Vector2{X: iconCenter.X + iconSize/4, Y: iconCenter.Y},
            rl.Vector2{X: iconCenter.X - iconSize/6, Y: iconCenter.Y},
            iconColor,
        )
        rl.DrawTriangle(
            rl.Vector2{X: iconCenter.X - iconSize/6, Y: iconCenter.Y},
            rl.Vector2{X: iconCenter.X + iconSize/3, Y: iconCenter.Y + iconSize/2},
            rl.Vector2{X: iconCenter.X + iconSize/6, Y: iconCenter.Y},
            iconColor,
        )

    case entities.PowerUpShield:
        rl.DrawCircleLinesEx(iconCenter, iconSize/2, 2, iconColor)
        rl.DrawCircleLinesEx(iconCenter, iconSize/3, 2, iconColor)

    case entities.PowerUpSizeboost:
        // Draw expand arrows
        arrowSize := iconSize / 3

        // Draw four arrows pointing outward
        for j := 0; j < 4; j++ {
            angle := float32(j) * 90.0
            radAngle := angle * math.Pi / 180.0

            // Arrow end (outer point)
            endX := iconCenter.X + float32(math.Cos(float64(radAngle))) * arrowSize
            endY := iconCenter.Y + float32(math.Sin(float64(radAngle))) * arrowSize

            // Draw arrow line
            rl.DrawLineEx(
                iconCenter,
                rl.Vector2{X: endX, Y: endY},
                2.0,
                iconColor,
            )

            // Draw arrow head
            headSize := arrowSize * 0.3
            headAngle1 := radAngle + math.Pi*0.75
            headAngle2 := radAngle - math.Pi*0.75
```

```
            head1X := endX + float32(math.Cos(float64(headAngle1))) * headSize
            head1Y := endY + float32(math.Sin(float64(headAngle1))) * headSize

            head2X := endX + float32(math.Cos(float64(headAngle2))) * headSize
            head2Y := endY + float32(math.Sin(float64(headAngle2))) * headSize

            rl.DrawLineEx(
                rl.Vector2{X: endX, Y: endY},
                rl.Vector2{X: head1X, Y: head1Y},
                2.0,
                iconColor,
            )

            rl.DrawLineEx(
                rl.Vector2{X: endX, Y: endY},
                rl.Vector2{X: head2X, Y: head2Y},
                2.0,
                iconColor,
            )
        }
    }

    // Draw progress bar below icon
    progressBarHeight := 5.0
    progressBarWidth := slotRect.Width - 4

    // Background of progress bar
    rl.DrawRectangle(
        int32(slotRect.X + 2),
        int32(slotRect.Y + slotRect.Height - progressBarHeight - 2),
        int32(progressBarWidth),
        int32(progressBarHeight),
        rl.Color{R: 50, G: 50, B: 50, A: 200},
    )

    // Fill of progress bar
    rl.DrawRectangle(
        int32(slotRect.X + 2),
        int32(slotRect.Y + slotRect.Height - progressBarHeight - 2),
        int32(progressBarWidth * progress),
        int32(progressBarHeight),
        iconColor,
    )

    // Flash when about to expire
    if progress < 0.2 {
```

```
                // Add a pulse effect to draw attention
                if int(g.GameTime*10)%2 == 0 {
                    rl.DrawRectangleLinesEx(slotRect, 2, iconColor)
                }
            }
        } else {
            // Draw empty slot with icon outline
            var iconColor rl.Color

            switch powerType {
            case entities.PowerUpMagnet:
                iconColor = rl.Color{R: 100, G: 50, B: 150, A: 100}
            case entities.PowerUpSpeed:
                iconColor = rl.Color{R: 150, G: 50, B: 50, A: 100}
            case entities.PowerUpShield:
                iconColor = rl.Color{R: 50, G: 150, B: 200, A: 100}
            case entities.PowerUpSizeboost:
                iconColor = rl.Color{R: 150, G: 100, B: 50, A: 100}
            }

            // Draw faded icon
            iconCenter := rl.Vector2{
                X: slotRect.X + slotRect.Width/2,
                Y: slotRect.Y + slotRect.Height/2,
            }

            rl.DrawCircleLinesEx(iconCenter, slotRect.Width/4, 1, iconColor)
        }

        // Draw slot border
        rl.DrawRectangleLinesEx(slotRect, 1, rl.Color{R: 100, G: 100, B: 100, A: 255})
    }
}
```

4. Optional Explosion Effects for Polished Visuals

go

```go
// entities/explosion.go
package entities

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

type ExplosionParticle struct {
    Position    rl.Vector2
    Velocity    rl.Vector2
    Color       rl.Color
    Size        float32
    Lifetime    float32
    MaxLifetime float32
    Active      bool
}

type Explosion struct {
    Position      rl.Vector2
    Particles     []ExplosionParticle
    TimeExisting  float32
    Duration      float32
    ShockwaveSize float32
    MaxShockwave  float32
    Color         rl.Color
    Active        bool
}

func NewExplosion(position rl.Vector2, size float32, color rl.Color) *Explosion {
    particleCount := int(20 + size/2)
    if particleCount > 60 {
        particleCount = 60 // Cap particles for performance
    }

    // Create explosion particles
    particles := make([]ExplosionParticle, particleCount)

    for i := range particles {
        // Random angle and distance from center
        angle := rand.Float32() * 2 * math# Phase 2 Implementation Plan: Growth & Powerup Syste

## 1. Enhanced Food System

### 1.1 Food Types and Variety
```

```go
// entities/food.go
package entities

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

// FoodType enum for different food variants
type FoodType int

const (
    FoodTypeBasic FoodType = iota
    FoodTypePremium
    FoodTypeRare
)

type Food struct {
    Position   rl.Vector2
    Velocity   rl.Vector2 // For moving food
    Color      rl.Color
    Size       float32
    Value      int
    Type       FoodType
    PulseTime  float32
    Rotation   float32
    HasPhysics bool // Whether food moves or stays stationary
}

func NewRandomFood(position rl.Vector2) *Food {
    // Determine food type based on rarity
    foodTypeRoll := rand.Float32()
    var foodType FoodType
    var color rl.Color
    var size float32
    var value int
    var hasPhysics bool

    switch {
    case foodTypeRoll < 0.02: // 2% chance for rare food
        foodType = FoodTypeRare
        color = rl.Gold
        size = 8.0 + rand.Float32() * 2.0
        value = 25 + rand.Intn(15)
```

```
        hasPhysics = true // Rare food moves around

    case foodTypeRoll < 0.15: // 13% chance for premium food
        foodType = FoodTypePremium
        color = rl.Blue
        size = 5.0 + rand.Float32() * 3.0
        value = 10 + rand.Intn(8)
        hasPhysics = rand.Float32() < 0.3 // 30% chance to have physics

    default: // 85% common food
        foodType = FoodTypeBasic
        color = rl.Green
        size = 3.0 + rand.Float32() * 2.0
        value = 2 + rand.Intn(5)
        hasPhysics = false
    }

    // Set velocity for moving food
    var velocity rl.Vector2
    if hasPhysics {
        angle := rand.Float32() * 2 * math.Pi
        speed := 20.0 + rand.Float32() * 30.0
        velocity = rl.Vector2{
            X: float32(math.Cos(float64(angle))) * speed,
            Y: float32(math.Sin(float64(angle))) * speed,
        }
    }

    return &Food{
        Position:   position,
        Velocity:   velocity,
        Color:      color,
        Size:       size,
        Value:      value,
        Type:       foodType,
        PulseTime:  rand.Float32() * 2 * math.Pi, // Random start phase
        Rotation:   rand.Float32() * 360,
        HasPhysics: hasPhysics,
    }
}

func (f *Food) Update(dt float32, worldBounds rl.Rectangle) {
    // Animation update
    f.PulseTime += dt * 2
    if f.PulseTime > 2*math.Pi {
        f.PulseTime -= 2 * math.Pi
    }
```

```
    // Rotation update
    rotationSpeed := 30.0
    if f.Type == FoodTypePremium {
        rotationSpeed = 60.0
    } else if f.Type == FoodTypeRare {
        rotationSpeed = 90.0
    }

    f.Rotation += dt * rotationSpeed
    if f.Rotation > 360 {
        f.Rotation -= 360
    }

    // Physics update (for moving food)
    if f.HasPhysics {
        f.Position.X += f.Velocity.X * dt
        f.Position.Y += f.Velocity.Y * dt

        // Bounce off world boundaries
        if f.Position.X < worldBounds.X + f.Size {
            f.Position.X = worldBounds.X + f.Size
            f.Velocity.X = -f.Velocity.X
        } else if f.Position.X > worldBounds.X + worldBounds.Width - f.Size {
            f.Position.X = worldBounds.X + worldBounds.Width - f.Size
            f.Velocity.X = -f.Velocity.X
        }

        if f.Position.Y < worldBounds.Y + f.Size {
            f.Position.Y = worldBounds.Y + f.Size
            f.Velocity.Y = -f.Velocity.Y
        } else if f.Position.Y > worldBounds.Y + worldBounds.Height - f.Size {
            f.Position.Y = worldBounds.Y + worldBounds.Height - f.Size
            f.Velocity.Y = -f.Velocity.Y
        }

        // Apply slight drag to eventually slow down
        f.Velocity.X *= 0.99
        f.Velocity.Y *= 0.99
    }
}

func (f *Food) Draw() {
    // Get pulse factor (between 0.85 and 1.15)
    pulseFactor := 0.85 + 0.15*float32(math.Sin(float64(f.PulseTime)))

    // Draw food with pulsing size
```

```
drawSize := f.Size * pulseFactor

// Number of sides varies by food type
sides := 5 // Pentagon for basic
if f.Type == FoodTypePremium {
    sides = 6 // Hexagon for premium
} else if f.Type == FoodTypeRare {
    sides = 8 // Octagon for rare
}

// Draw a polygon with rotation
centerX := f.Position.X
centerY := f.Position.Y

// Calculate all vertex positions
vertices := make([]rl.Vector2, sides)
for i := 0; i < sides; i++ {
    angle := f.Rotation*math.Pi/180 + float32(i)*2*math.Pi/float32(sides)
    vertices[i] = rl.Vector2{
        X: centerX + drawSize * float32(math.Cos(float64(angle))),
        Y: centerY + drawSize * float32(math.Sin(float64(angle))),
    }
}

// Draw polygon outline
for i := 0; i < sides; i++ {
    j := (i + 1) % sides
    lineThickness := 1.5
    if f.Type == FoodTypePremium {
        lineThickness = 2.0
    } else if f.Type == FoodTypeRare {
        lineThickness = 2.5
    }

    rl.DrawLineEx(vertices[i], vertices[j], lineThickness, f.Color)
}

// Draw center based on food type
innerSize := drawSize * 0.3
if f.Type == FoodTypeBasic {
    rl.DrawCircleV(f.Position, innerSize, f.Color)
} else if f.Type == FoodTypePremium {
    // Draw a star shape
    innerVertices := make([]rl.Vector2, sides)
    for i := 0; i < sides; i++ {
        angle := f.Rotation*math.Pi/180 + float32(i)*2*math.Pi/float32(sides) + math.Pi/flc
        innerVertices[i] = rl.Vector2{
```

```
                X: centerX + innerSize * float32(math.Cos(float64(angle))),
                Y: centerY + innerSize * float32(math.Sin(float64(angle))),
            }
        }

        for i := 0; i < sides; i++ {
            rl.DrawLineEx(f.Position, vertices[i], 1.0, f.Color)
        }
    } else if f.Type == FoodTypeRare {
        // Draw a glowing center with inner circle
        rl.DrawCircleV(f.Position, innerSize, f.Color)

        // Add a pulsing glow effect
        glowAlpha := uint8(128 + 127*math.Sin(float64(f.PulseTime*1.5)))
        glowColor := rl.Color{R: f.Color.R, G: f.Color.G, B: f.Color.B, A: glowAlpha}
        rl.DrawCircleV(f.Position, innerSize*1.8, glowColor)
    }
}

func (f *Food) GetBounds() rl.Rectangle {
    return rl.Rectangle{
        X:      f.Position.X - f.Size,
        Y:      f.Position.Y - f.Size,
        Width:  f.Size * 2,
        Height: f.Size * 2,
    }
}
```

1.2 Food Generator System

go

```go
// systems/food_generator.go
package systems

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

type FoodGenerator struct {
    FoodEntities []*entities.Food
    FoodPool     *util.Pool

    // Food generation parameters
    MaxFood           int
    SpawnTimer        float32
    BaseSpawnRate     float32 // Base food items per second
    SpawnRateVariance float32 // Random variance in spawn rate
    CurrentSpawnRate  float32 // Current adjusted spawn rate

    // World parameters
    WorldBounds rl.Rectangle

    // Food cluster generation
    ClusterTimer      float32
    ClusterInterval   float32
    ClusterChance     float32
}

func NewFoodGenerator(worldBounds rl.Rectangle) *FoodGenerator {
    maxFood := 1000 // Support up to 1000 food items

    // Create pool for food entities
    foodPool := util.NewPool(
        func() interface{} {
            return &entities.Food{}
        },
        maxFood / 2, // Pre-allocate half of max capacity
    )

    return &FoodGenerator{
        FoodEntities:     make([]*entities.Food, 0, maxFood),
        FoodPool:         foodPool,
        MaxFood:          maxFood,
```

```
        SpawnTimer:       0,
        BaseSpawnRate:    8.0,  // 8 food items per second
        SpawnRateVariance: 3.0, // +/- 3 variance
        CurrentSpawnRate: 8.0,  // Initial rate
        WorldBounds:      worldBounds,
        ClusterTimer:     0,
        ClusterInterval:  5.0,  // Check for cluster generation every 5 seconds
        ClusterChance:    0.3,  // 30% chance to spawn a cluster
    }
}

// Initialize with some starting food
func (fg *FoodGenerator) Initialize(amount int) {
    for i := 0; i < amount; i++ {
        randomX := fg.WorldBounds.X + rand.Float32() * fg.WorldBounds.Width
        randomY := fg.WorldBounds.Y + rand.Float32() * fg.WorldBounds.Height

        food := entities.NewRandomFood(rl.Vector2{X: randomX, Y: randomY})
        fg.FoodEntities = append(fg.FoodEntities, food)
    }
}

func (fg *FoodGenerator) Update(dt float32, quadtree *util.Quadtree) {
    // Update spawn timer
    fg.SpawnTimer += dt

    // Vary spawn rate slightly to create natural feeling
    fg.CurrentSpawnRate = fg.BaseSpawnRate + (rand.Float32()*2-1) * fg.SpawnRateVariance

    // Check if it's time to spawn new food
    spawnInterval := 1.0 / fg.CurrentSpawnRate

    for fg.SpawnTimer >= spawnInterval && len(fg.FoodEntities) < fg.MaxFood {
        fg.SpawnTimer -= spawnInterval

        // Create new food at random position
        randomX := fg.WorldBounds.X + rand.Float32() * fg.WorldBounds.Width
        randomY := fg.WorldBounds.Y + rand.Float32() * fg.WorldBounds.Height

        food := entities.NewRandomFood(rl.Vector2{X: randomX, Y: randomY})
        fg.FoodEntities = append(fg.FoodEntities, food)

        // Add to quadtree for collision detection
        quadtree.Root.Insert(food, food)
    }

    // Handle cluster generation
```

```
    fg.ClusterTimer += dt
    if fg.ClusterTimer >= fg.ClusterInterval {
        fg.ClusterTimer = 0

        if rand.Float32() < fg.ClusterChance && len(fg.FoodEntities) < fg.MaxFood - 20 {
            fg.GenerateCluster()
        }
    }

    // Update all food entities
    for _, food := range fg.FoodEntities {
        food.Update(dt, fg.WorldBounds)
    }
}

// Generate a cluster of food in one area
func (fg *FoodGenerator) GenerateCluster() {
    // Choose a random point for the cluster center
    centerX := fg.WorldBounds.X + rand.Float32() * fg.WorldBounds.Width
    centerY := fg.WorldBounds.Y + rand.Float32() * fg.WorldBounds.Height
    center := rl.Vector2{X: centerX, Y: centerY}

    // Determine cluster parameters
    clusterSize := 10 + rand.Intn(15) // 10-25 food items
    clusterRadius := 50.0 + rand.Float32() * 100.0 // 50-150 radius

    // Higher chance of premium/rare food in clusters
    premiumChance := 0.3  // 30% chance of premium
    rareChance := 0.08    // 8% chance of rare

    // Generate food in the cluster
    for i := 0; i < clusterSize; i++ {
        // Make sure we don't exceed max food
        if len(fg.FoodEntities) >= fg.MaxFood {
            break
        }

        // Random position within the cluster radius
        angle := rand.Float32() * 2 * math.Pi
        distance := rand.Float32() * clusterRadius

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

        // Create food with higher chance of premium/rare
```

```
var food *entities.Food

typeRoll := rand.Float32()
if typeRoll < rareChance {
    // Create a rare food
    food = &entities.Food{
        Position:   position,
        Color:      rl.Gold,
        Size:       8.0 + rand.Float32() * 2.0,
        Value:      25 + rand.Intn(15),
        Type:       entities.FoodTypeRare,
        PulseTime:  rand.Float32() * 2 * math.Pi,
        Rotation:   rand.Float32() * 360,
        HasPhysics: true,
    }

    // Set velocity for rare food
    angle := rand.Float32() * 2 * math.Pi
    speed := 20.0 + rand.Float32() * 30.0
    food.Velocity = rl.Vector2{
        X: float32(math.Cos(float64(angle))) * speed,
        Y: float32(math.Sin(float64(angle))) * speed,
    }
} else if typeRoll < rareChance + premiumChance {
    // Create a premium food
    food = &entities.Food{
        Position:   position,
        Color:      rl.Blue,
        Size:       5.0 + rand.Float32() * 3.0,
        Value:      10 + rand.Intn(8),
        Type:       entities.FoodTypePremium,
        PulseTime:  rand.Float32() * 2 * math.Pi,
        Rotation:   rand.Float32() * 360,
        HasPhysics: rand.Float32() < 0.3,
    }

    if food.HasPhysics {
        angle := rand.Float32() * 2 * math.Pi
        speed := 15.0 + rand.Float32() * 20.0
        food.Velocity = rl.Vector2{
            X: float32(math.Cos(float64(angle))) * speed,
            Y: float32(math.Sin(float64(angle))) * speed,
        }
    }
} else {
    // Create a basic food
    food = &entities.Food{
```

```
                Position:   position,
                Color:      rl.Green,
                Size:       3.0 + rand.Float32() * 2.0,
                Value:      2 + rand.Intn(5),
                Type:       entities.FoodTypeBasic,
                PulseTime:  rand.Float32() * 2 * math.Pi,
                Rotation:   rand.Float32() * 360,
                HasPhysics: false,
            }
        }

        fg.FoodEntities = append(fg.FoodEntities, food)
    }
}

func (fg *FoodGenerator) RemoveFood(food *entities.Food) {
    // Find and remove food from list
    for i, f := range fg.FoodEntities {
        if f == food {
            // Return to pool (for potential reuse)
            fg.FoodPool.Return(food)

            // Remove from list (order not important, so use swap-and-pop)
            lastIdx := len(fg.FoodEntities) - 1
            fg.FoodEntities[i] = fg.FoodEntities[lastIdx]
            fg.FoodEntities = fg.FoodEntities[:lastIdx]
            break
        }
    }
}
```

2. Power-up System Implementation

2.1 Core Power-up Structure

go

```go
// entities/powerup.go
package entities

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

// PowerUpType enum to identify different power-up types
type PowerUpType int

const (
    PowerUpMagnet PowerUpType = iota
    PowerUpSpeed
    PowerUpShield
    PowerUpSizeboost
)

type PowerUp struct {
    Position    rl.Vector2
    Velocity    rl.Vector2
    Type        PowerUpType
    Color       rl.Color
    Size        float32
    Rotation    float32
    Duration    float32 // How long the power-up lasts when collected (in seconds)
    PulseTime   float32
    Lifetime    float32 // How long the power-up exists in the world before disappearing
    MaxLifetime float32 // Maximum lifetime
    Active      bool
}

func NewPowerUp(position rl.Vector2, powerType PowerUpType) *PowerUp {
    var color rl.Color
    var duration float32
    var size float32 = 12.0 // Base size for all power-ups

    // Configure based on type
    switch powerType {
    case PowerUpMagnet:
        color = rl.Purple
        duration = 10.0 // 10 seconds of magnetic attraction

    case PowerUpSpeed:
        color = rl.Red
```

```
        duration = 8.0 // 8 seconds of speed boost

    case PowerUpShield:
        color = rl.SkyBlue
        duration = 5.0 // 5 seconds of shield

    case PowerUpSizeboost:
        color = rl.Orange
        duration = 15.0 // 15 seconds of increased size gain
    }

    // Random initial velocity
    angle := rand.Float32() * 2 * math.Pi
    speed := 10.0 + rand.Float32() * 20.0

    return &PowerUp{
        Position:    position,
        Velocity:    rl.Vector2{
                        X: float32(math.Cos(float64(angle))) * speed,
                        Y: float32(math.Sin(float64(angle))) * speed,
                    },
        Type:        powerType,
        Color:       color,
        Size:        size,
        Rotation:    rand.Float32() * 360,
        Duration:    duration,
        PulseTime:   rand.Float32() * 2 * math.Pi,
        Lifetime:    30.0 + rand.Float32() * 30.0, // 30-60 seconds before disappearing
        MaxLifetime: 30.0 + rand.Float32() * 30.0,
        Active:      true,
    }
}

func (p *PowerUp) Update(dt float32, worldBounds rl.Rectangle) {
    // Update position based on velocity
    p.Position.X += p.Velocity.X * dt
    p.Position.Y += p.Velocity.Y * dt

    // Bounce off world boundaries
    if p.Position.X < worldBounds.X + p.Size {
        p.Position.X = worldBounds.X + p.Size
        p.Velocity.X = -p.Velocity.X * 0.8 // Slight damping on bounce
    } else if p.Position.X > worldBounds.X + worldBounds.Width - p.Size {
        p.Position.X = worldBounds.X + worldBounds.Width - p.Size
        p.Velocity.X = -p.Velocity.X * 0.8
    }
```

```
    if p.Position.Y < worldBounds.Y + p.Size {
        p.Position.Y = worldBounds.Y + p.Size
        p.Velocity.Y = -p.Velocity.Y * 0.8
    } else if p.Position.Y > worldBounds.Y + worldBounds.Height - p.Size {
        p.Position.Y = worldBounds.Y + worldBounds.Height - p.Size
        p.Velocity.Y = -p.Velocity.Y * 0.8
    }

    // Apply drag
    p.Velocity.X *= 0.98
    p.Velocity.Y *= 0.98

    // Rotate
    p.Rotation += dt * 45 // 45 degrees per second
    if p.Rotation > 360 {
        p.Rotation -= 360
    }

    // Update pulse animation
    p.PulseTime += dt * 3
    if p.PulseTime > 2*math.Pi {
        p.PulseTime -= 2 * math.Pi
    }

    // Update lifetime
    p.Lifetime -= dt
    if p.Lifetime <= 0 {
        p.Active = false
    }
}

func (p *PowerUp) Draw() {
    if !p.Active {
        return
    }

    // Calculate pulse effect (pulsate more rapidly as expiration approaches)
    lifetimeRatio := p.Lifetime / p.MaxLifetime
    pulseSpeed := 1.0 + (1.0 - lifetimeRatio) * 2.0 // Pulse faster when near expiration
    pulseAmount := 0.2 + (1.0 - lifetimeRatio) * 0.3 // Pulse more dramatically when near expir

    pulseFactor := 1.0 - pulseAmount + pulseAmount * float32(math.Sin(float64(p.PulseTime) * pu

    // Draw with pulsing size
    drawSize := p.Size * pulseFactor

    // Make color pulse alpha when nearing expiration
```

```
    drawColor := p.Color
    if p.Lifetime < 5.0 { // Last 5 seconds
        alphaFactor := math.Sin(float64(p.PulseTime) * 2.0)
        minAlpha := uint8(100 + 155 * lifetimeRatio) // Fade from 255 to 100 as time runs out

        drawColor.A = uint8(float64(minAlpha) + alphaFactor*float64(255-minAlpha))
    }

    // Draw based on power-up type
    switch p.Type {
    case PowerUpMagnet:
        drawMagnetPowerUp(p.Position, drawSize, drawColor, p.Rotation)
    case PowerUpSpeed:
        drawSpeedPowerUp(p.Position, drawSize, drawColor, p.Rotation)
    case PowerUpShield:
        drawShieldPowerUp(p.Position, drawSize, drawColor, p.Rotation)
    case PowerUpSizeboost:
        drawSizeBoostPowerUp(p.Position, drawSize, drawColor, p.Rotation)
    }
}

func drawMagnetPowerUp(pos rl.Vector2, size float32, color rl.Color, rotation float32) {
    // Draw outer circle
    rl.DrawCircleLinesEx(pos, size, 2.0, color)

    // Draw magnet symbol
    radAngle := rotation * math.Pi / 180.0

    // North pole
    northX := pos.X + float32(math.Cos(float64(radAngle))) * size * 0.6
    northY := pos.Y + float32(math.Sin(float64(radAngle))) * size * 0.6

    // South pole
    southX := pos.X - float32(math.Cos(float64(radAngle))) * size * 0.6
    southY := pos.Y - float32(math.Sin(float64(radAngle))) * size * 0.6

    // Draw poles
    rl.DrawRectanglePro(
        rl.Rectangle{X: northX - size*0.3, Y: northY - size*0.3, Width: size*0.6, Height: size*
        rl.Vector2{X: size*0.3, Y: size*0.3},
        rotation,
        color,
    )

    rl.DrawRectanglePro(
        rl.Rectangle{X: southX - size*0.3, Y: southY - size*0.3, Width: size*0.6, Height: size*
        rl.Vector2{X: size*0.3, Y: size*0.3},
```

```
        rotation,
        color,
    )
}

func drawSpeedPowerUp(pos rl.Vector2, size float32, color rl.Color, rotation float32) {
    // Draw outer circle
    rl.DrawCircleLinesEx(pos, size, 2.0, color)

    // Draw lightning bolt symbol
    vertices := []rl.Vector2{
        {X: pos.X - size*0.4, Y: pos.Y - size*0.6}, // Top left
        {X: pos.X + size*0.1, Y: pos.Y - size*0.1}, // Middle right
        {X: pos.X - size*0.1, Y: pos.Y - size*0.1}, // Middle left
        {X: pos.X + size*0.4, Y: pos.Y + size*0.6}, // Bottom right
        {X: pos.X,            Y: pos.Y + size*0.1}, // Bottom middle
        {X: pos.X - size*0.2, Y: pos.Y + size*0.1}, // Bottom left
    }

    // Rotate vertices around center
    radAngle := rotation * math.Pi / 180.0
    sinRot := float32(math.Sin(float64(radAngle)))
    cosRot := float32(math.Cos(float64(radAngle)))

    for i := range vertices {
        // Translate to origin
        x := vertices[i].X - pos.X
        y := vertices[i].Y - pos.Y

        // Rotate
        rotX := x*cosRot - y*sinRot
        rotY := x*sinRot + y*cosRot

        // Translate back
        vertices[i].X = rotX + pos.X
        vertices[i].Y = rotY + pos.Y
    }

    // Draw lightning bolt
    for i := 0; i < len(vertices); i++ {
        rl.DrawLineEx(vertices[i], vertices[(i+1)%len(vertices)], 2.0, color)
    }
}

func drawShieldPowerUp(pos rl.Vector2, size float32, color rl.Color, rotation float32) {
    // Draw outer circle
    rl.DrawCircleLinesEx(pos, size, 2.0, color)
```

```
    // Draw shield symbol (a rounded rectangle)
    shieldWidth := size * 0.8
    shieldHeight := size * 1.1

    shieldRect := rl.Rectangle{
        X:      pos.X - shieldWidth/2,
        Y:      pos.Y - shieldHeight/2,
        Width:  shieldWidth,
        Height: shieldHeight,
    }

    rl.DrawRectangleRounded(shieldRect, 0.5, 6, color)

    // Draw inner shield details (smaller rounded rectangle)
    innerShieldRect := rl.Rectangle{
        X:      pos.X - shieldWidth*0.4,
        Y:      pos.Y - shieldHeight*0.4,
        Width:  shieldWidth*0.8,
        Height: shieldHeight*0.8,
    }

    rl.DrawRectangleRoundedLines(innerShieldRect, 0.5, 6, 1.0, color)
}

func drawSizeBoostPowerUp(pos rl.Vector2, size float32, color rl.Color, rotation float32) {
    // Draw outer circle
    rl.DrawCircleLinesEx(pos, size, 2.0, color)

    // Draw expand symbol (arrows pointing outward)
    arrowSize := size * 0.6

    // Draw four arrows pointing outward
    for i := 0; i < 4; i++ {
        angle := float32(i) * 90.0 + rotation
        radAngle := angle * math.Pi / 180.0

        // Arrow start (inner point)
        startX := pos.X + float32(math.Cos(float64(radAngle))) * size * 0.2
        startY := pos.Y + float32(math.Sin(float64(radAngle))) * size * 0.2

        // Arrow end (outer point)
        endX := pos.X + float32(math.Cos(float64(radAngle))) * size * 0.8
        endY := pos.Y + float32(math.Sin(float64(radAngle))) * size * 0.8

        // Draw arrow line
        rl.DrawLineEx(
```

```
            rl.Vector2{X: startX, Y: startY},
            rl.Vector2{X: endX, Y: endY},
            2.0,
            color,
        )

        // Draw arrow head
        headSize := size * 0.2
        headAngle1 := radAngle + math.Pi*0.85 // Head angle offset
        headAngle2 := radAngle - math.Pi*0.85

        head1X := endX + float32(math.Cos(float64(headAngle1))) * headSize
        head1Y := endY + float32(math.Sin(float64(headAngle1))) * headSize

        head2X := endX + float32(math.Cos(float64(headAngle2))) * headSize
        head2Y := endY + float32(math.Sin(float64(headAngle2))) * headSize

        rl.DrawLineEx(
            rl.Vector2{X: endX, Y: endY},
            rl.Vector2{X: head1X, Y: head1Y},
            2.0,
            color,
        )

        rl.DrawLineEx(
            rl.Vector2{X: endX, Y: endY},
            rl.Vector2{X: head2X, Y: head2Y},
            2.0,
            color,
        )
    }
}

func (p *PowerUp) GetBounds() rl.Rectangle {
    return rl.Rectangle{
        X:      p.Position.X - p.Size,
        Y:      p.Position.Y - p.Size,
        Width:  p.Size * 2,
        Height: p.Size * 2,
    }
}
```

2.2 PowerUp Manager System

go

```go
// systems/powerup_manager.go
package systems

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

type PowerUpManager struct {
    PowerUps       []*entities.PowerUp
    PowerUpPool    *util.Pool

    // PowerUp generation settings
    SpawnTimer          float32
    BaseSpawnInterval   float32 // Average seconds between spawns
    SpawnIntervalJitter float32 // Random jitter added to spawn interval
    MaxPowerUps         int     // Maximum number of power-ups in world

    // World bounds
    WorldBounds rl.Rectangle
}

func NewPowerUpManager(worldBounds rl.Rectangle) *PowerUpManager {
    maxPowerUps := 10 // Maximum 10 power-ups at once

    powerUpPool := util.NewPool(
        func() interface{} {
            return &entities.PowerUp{}
        },
        maxPowerUps,
    )

    return &PowerUpManager{
        PowerUps:           make([]*entities.PowerUp, 0, maxPowerUps),
        PowerUpPool:        powerUpPool,
        SpawnTimer:         0,
        BaseSpawnInterval:  20.0, // Spawn roughly every 20 seconds
        SpawnIntervalJitter: 10.0, // +/- 10 seconds
        MaxPowerUps:        maxPowerUps,
        WorldBounds:        worldBounds,
    }
}
```

```go
func (pm *PowerUpManager) Update(dt float32, quadtree *util.Quadtree) {
    // Update spawn timer
    pm.SpawnTimer += dt

    // Calculate when next power-up should spawn
    nextSpawnTime := pm.BaseSpawnInterval + (rand.Float32()*2-1) * pm.SpawnIntervalJitter

    // Check if it's time to spawn and we have room
    if pm.SpawnTimer >= nextSpawnTime && len(pm.PowerUps) < pm.MaxPowerUps {
        pm.SpawnTimer = 0

        // Choose a random position that's at least 10% away from world edges
        padding := pm.WorldBounds.Width * 0.1
        randomX := pm.WorldBounds.X + padding + rand.Float32() * (pm.WorldBounds.Width - paddir
        randomY := pm.WorldBounds.Y + padding + rand.Float32() * (pm.WorldBounds.Height - paddi

        // Choose a random power-up type
        powerUpType := entities.PowerUpType(rand.Intn(4)) // 4 types defined in enum

        // Create and add the power-up
        powerUp := entities.NewPowerUp(rl.Vector2{X: randomX, Y: randomY}, powerUpType)
        pm.PowerUps = append(pm.PowerUps, powerUp)

        // Add to quadtree
        quadtree.Root.Insert(powerUp, powerUp)
    }

    // Update existing power-ups and remove inactive ones
    for i := len(pm.PowerUps) - 1; i >= 0; i-- {
        pm.PowerUps[i].Update(dt, pm.WorldBounds)

        // Remove expired power-ups
        if !pm.PowerUps[i].Active {
            // Return to pool
            pm.PowerUpPool.Return(pm.PowerUps[i])

            // Remove from list
            pm.PowerUps[i] = pm.PowerUps[len(pm.PowerUps)-1]
            pm.PowerUps = pm.PowerUps[:len(pm.PowerUps)-1]
        }
    }
}

func (pm *PowerUpManager) Draw() {
    for _, powerUp := range pm.PowerUps {
        powerUp.Draw()
    }
```

```
}

func (pm *PowerUpManager) RemovePowerUp(powerUp *entities.PowerUp) {
    for i, p := range pm.PowerUps {
        if p == powerUp {
            // Return to pool
            pm.PowerUpPool.Return(powerUp)

            // Remove from list
            pm.PowerUps[i] = pm.PowerUps[len(pm.PowerUps)-1]
            pm.PowerUps = pm.PowerUps[:len(pm.PowerUps)-1]
            break
        }
    }
}
```

3. Enhanced Player Growth & PowerUp Effects

3.1 Player Enhancements

go

```go
// entities/player.go (additions)

// Add to the Player struct:
type Player struct {
    // ... existing fields

    // Growth system
    Size              float32
    BaseSize          float32  // Starting size
    MaxSize           float32  // Maximum possible size
    GrowthMultiplier  float32  // Multiplier for food value to size growth

    // Score system
    Score             int
    TotalFoodCollected int

    // PowerUp effects
    ActivePowerUps    map[entities.PowerUpType]*PowerUpEffect
    MagnetRange       float32  // Range to attract food
    SpeedMultiplier   float32  // Multiplier for movement speed
    HasShield         bool     // Whether shield is active
    SizeBoostFactor   float32  // Multiplier for size growth from food
}

// PowerUpEffect tracks active power-up duration
type PowerUpEffect struct {
    Type         entities.PowerUpType
    RemainingTime float32
    TotalDuration float32
}

// Modify NewPlayer to initialize these fields:
func NewPlayer() *Player {
    player := &Player{
        // ... existing initialization

        // Growth parameters
        Size:             20.0,
        BaseSize:         20.0,
        MaxSize:          100.0,
        GrowthMultiplier: 0.2,  // Each food value point adds 0.2 to size
```