Phase 1 Implementation Plan: Core Systems Enhancement

1.1 Large World & Camera System

World Size Expansion

go

```go
// constants/constants.go
const (
    // Current values might be around 800x600
    ScreenWidth  = 1280
    ScreenHeight = 720

    // New world size (10x current size)
    WorldWidth  = 8000
    WorldHeight = 8000

    // Visual boundary thickness
    WorldBorderThickness = 50
)
```

Camera Implementation

go

```go
// game/camera.go
package game

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

type Camera struct {
	Target   *entities.Player
	Position rl.Vector2
	Rotation float32
	Zoom     float32

	// Smoothing parameters
	SmoothingFactor float32
	EdgePadding     float32
}

func NewCamera(target *entities.Player) *Camera {
	return &Camera{
		Target:          target,
		Position:        rl.Vector2{},
		Rotation:        0.0,
		Zoom:            1.0,
		SmoothingFactor: 0.1,  // Lower = smoother camera
		EdgePadding:     100,  // Keep player this far from screen edge
	}
}

func (c *Camera) Update(dt float32) {
	// Calculate target position (centered on player)
	targetPos := rl.Vector2{
		X: c.Target.Position.X - constants.ScreenWidth/2,
		Y: c.Target.Position.Y - constants.ScreenHeight/2,
	}

	// Apply smoothing with lerp
	c.Position.X = util.Lerp(c.Position.X, targetPos.X, c.SmoothingFactor)
	c.Position.Y = util.Lerp(c.Position.Y, targetPos.Y, c.SmoothingFactor)

	// Constrain camera to world bounds
	c.Position.X = util.Clamp(c.Position.X, 0, constants.WorldWidth-constants.ScreenWidth)
	c.Position.Y = util.Clamp(c.Position.Y, 0, constants.WorldHeight-constants.ScreenHeight)
```

```go
}

func (c *Camera) GetRLCamera2D() rl.Camera2D {
    return rl.Camera2D{
        Target:   rl.Vector2{0, 0},
        Offset:   rl.Vector2{constants.ScreenWidth / 2, constants.ScreenHeight / 2},
        Rotation: c.Rotation,
        Zoom:     c.Zoom,
    }
}
```

Mini-map Implementation

go

```go
// ui/minimap.go
package ui

import (
	"github.com/gen2brain/raylib-go/raylib"
	"atomblaster/constants"
	"atomblaster/entities"
	"atomblaster/game"
)

type Minimap struct {
	Position    rl.Vector2
	Size        rl.Vector2
	BorderColor rl.Color
	MapColor    rl.Color
	PlayerColor rl.Color
	FoodColor   rl.Color
	Scale       rl.Vector2
}

func NewMinimap() *Minimap {
	return &Minimap{
		Position:    rl.Vector2{constants.ScreenWidth - 150, 20},
		Size:        rl.Vector2{130, 130},
		BorderColor: rl.Gray,
		MapColor:    rl.DarkGray,
		PlayerColor: rl.Red,
		FoodColor:   rl.Green,
		Scale: rl.Vector2{
			X: 130.0 / float32(constants.WorldWidth),
			Y: 130.0 / float32(constants.WorldHeight),
		},
	}
}

func (m *Minimap) Draw(gameState *game.Game) {
	// Draw minimap background
	rl.DrawRectangleV(m.Position, m.Size, m.MapColor)
	rl.DrawRectangleLinesEx(rl.Rectangle{
		X:      m.Position.X,
		Y:      m.Position.Y,
		Width:  m.Size.X,
		Height: m.Size.Y,
	}, 2, m.BorderColor)

	// Draw viewport area
```

```go
	viewportSize := rl.Vector2{
		X: constants.ScreenWidth * m.Scale.X,
		Y: constants.ScreenHeight * m.Scale.Y,
	}
	viewportPos := rl.Vector2{
		X: m.Position.X + gameState.Camera.Position.X * m.Scale.X,
		Y: m.Position.Y + gameState.Camera.Position.Y * m.Scale.Y,
	}
	rl.DrawRectangleLinesEx(rl.Rectangle{
		X:      viewportPos.X,
		Y:      viewportPos.Y,
		Width:  viewportSize.X,
		Height: viewportSize.Y,
	}, 1, rl.White)

	// Draw player position
	playerMiniPos := rl.Vector2{
		X: m.Position.X + gameState.Player.Position.X * m.Scale.X,
		Y: m.Position.Y + gameState.Player.Position.Y * m.Scale.Y,
	}
	rl.DrawCircleV(playerMiniPos, 3, m.PlayerColor)

	// Draw food entities
	for _, entity := range gameState.Entities {
		food, ok := entity.(*entities.Food)
		if ok {
			foodMiniPos := rl.Vector2{
				X: m.Position.X + food.Position.X * m.Scale.X,
				Y: m.Position.Y + food.Position.Y * m.Scale.Y,
			}
			rl.DrawCircleV(foodMiniPos, 1, m.FoodColor)
		}
	}
}
```

1.2 Optimization for Scale

Quadtree Implementation

go

```go
// util/quadtree.go
package util

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

const (
    MaxEntitiesPerNode = 10
    MaxDepth           = 5
)

type QuadtreeNode struct {
    Bounds    rl.Rectangle
    Depth     int
    Entities  []interface{}
    Children  [4]*QuadtreeNode
    Divided   bool
}

type Quadtree struct {
    Root      *QuadtreeNode
    WorldSize rl.Rectangle
}

// Create a new quadtree with specified world bounds
func NewQuadtree(worldBounds rl.Rectangle) *Quadtree {
    return &Quadtree{
        Root: &QuadtreeNode{
            Bounds:   worldBounds,
            Depth:    0,
            Entities: make([]interface{}, 0, MaxEntitiesPerNode),
            Divided:  false,
        },
        WorldSize: worldBounds,
    }
}

// Clear the quadtree
func (q *Quadtree) Clear() {
    q.Root = &QuadtreeNode{
        Bounds:   q.WorldSize,
        Depth:    0,
        Entities: make([]interface{}, 0, MaxEntitiesPerNode),
        Divided:  false,
    }
```

```go
}

// Interface for objects that can be stored in quadtree
type QuadtreeItem interface {
    GetBounds() rl.Rectangle
}

// Insert an entity into the quadtree
func (node *QuadtreeNode) Insert(entity interface{}, item QuadtreeItem) bool {
    // Check if entity is within this node's bounds
    entityBounds := item.GetBounds()
    if !CheckRectangleOverlap(node.Bounds, entityBounds) {
        return false
    }

    // If not at capacity or max depth, add to this node
    if len(node.Entities) < MaxEntitiesPerNode || node.Depth >= MaxDepth {
        node.Entities = append(node.Entities, entity)
        return true
    }

    // Otherwise, subdivide and add to children
    if !node.Divided {
        node.Subdivide()
    }

    // Try to insert into children
    for i := 0; i < 4; i++ {
        if node.Children[i].Insert(entity, item) {
            return true
        }
    }

    // If doesn't fit in any child (due to bounds checking), add to this node
    node.Entities = append(node.Entities, entity)
    return true
}

// Subdivide the node into four quadrants
func (node *QuadtreeNode) Subdivide() {
    halfWidth := node.Bounds.Width / 2
    halfHeight := node.Bounds.Height / 2
    x := node.Bounds.X
    y := node.Bounds.Y
    nextDepth := node.Depth + 1

    // Create four children nodes
```

```go
node.Children[0] = &QuadtreeNode{ // Top-left
    Bounds:   rl.Rectangle{X: x, Y: y, Width: halfWidth, Height: halfHeight},
    Depth:    nextDepth,
    Entities: make([]interface{}, 0, MaxEntitiesPerNode),
}

node.Children[1] = &QuadtreeNode{ // Top-right
    Bounds:   rl.Rectangle{X: x + halfWidth, Y: y, Width: halfWidth, Height: halfHeight},
    Depth:    nextDepth,
    Entities: make([]interface{}, 0, MaxEntitiesPerNode),
}

node.Children[2] = &QuadtreeNode{ // Bottom-left
    Bounds:   rl.Rectangle{X: x, Y: y + halfHeight, Width: halfWidth, Height: halfHeight},
    Depth:    nextDepth,
    Entities: make([]interface{}, 0, MaxEntitiesPerNode),
}

node.Children[3] = &QuadtreeNode{ // Bottom-right
    Bounds:   rl.Rectangle{X: x + halfWidth, Y: y + halfHeight, Width: halfWidth, Height: h
    Depth:    nextDepth,
    Entities: make([]interface{}, 0, MaxEntitiesPerNode),
}

node.Divided = true

// Redistribute existing entities among children
entitiesToRedistribute := node.Entities
node.Entities = make([]interface{}, 0, MaxEntitiesPerNode)

for _, entity := range entitiesToRedistribute {
    if item, ok := entity.(QuadtreeItem); ok {
        inserted := false
        for i := 0; i < 4; i++ {
            if node.Children[i].Insert(entity, item) {
                inserted = true
                break
            }
        }

        // If it doesn't fit in any child, keep it in this node
        if !inserted {
            node.Entities = append(node.Entities, entity)
        }
    } else {
        // If entity doesn't implement QuadtreeItem, keep it here
        node.Entities = append(node.Entities, entity)
```

```
        }
    }
}

// Query all entities that might collide with the given rectangle
func (node *QuadtreeNode) Query(bounds rl.Rectangle, results *[]interface{}) {
    // If this node doesn't overlap with the query bounds, return
    if !CheckRectangleOverlap(node.Bounds, bounds) {
        return
    }

    // Add all entities in this node to the results
    for _, entity := range node.Entities {
        *results = append(*results, entity)
    }

    // If this node is divided, query children
    if node.Divided {
        for i := 0; i < 4; i++ {
            node.Children[i].Query(bounds, results)
        }
    }
}

// Helper function to check if two rectangles overlap
func CheckRectangleOverlap(a, b rl.Rectangle) bool {
    return !(a.X >= b.X+b.Width || a.X+a.Width <= b.X ||
             a.Y >= b.Y+b.Height || a.Y+a.Height <= b.Y)
}
```

Entity Pooling System

go

```go
// entities/pool.go
package entities

import "sync"

// Generic object pool for entity types
type Pool struct {
    factory   func() interface{}
    instances []interface{}
    mutex     sync.Mutex
}

// Create a new pool with a factory function to create new instances
func NewPool(factory func() interface{}, initialCapacity int) *Pool {
    pool := &Pool{
        factory:   factory,
        instances: make([]interface{}, 0, initialCapacity),
    }

    // Pre-allocate instances
    for i := 0; i < initialCapacity; i++ {
        pool.instances = append(pool.instances, factory())
    }

    return pool
}

// Get an object from the pool
func (p *Pool) Get() interface{} {
    p.mutex.Lock()
    defer p.mutex.Unlock()

    if len(p.instances) == 0 {
        // If pool is empty, create a new instance
        return p.factory()
    }

    // Remove and return the last instance
    instance := p.instances[len(p.instances)-1]
    p.instances = p.instances[:len(p.instances)-1]
    return instance
}

// Return an object to the pool
func (p *Pool) Return(instance interface{}) {
    p.mutex.Lock()
```

```
    defer p.mutex.Unlock()

    p.instances = append(p.instances, instance)
}
```

Viewport Culling

go

```go
// game/game.go (partial implementation for the Update method)

func (g *Game) Update(dt float32) {
    // Update camera
    g.Camera.Update(dt)

    // Get the current viewport as a rectangle
    viewport := rl.Rectangle{
        X:      g.Camera.Position.X,
        Y:      g.Camera.Position.Y,
        Width:  constants.ScreenWidth,
        Height: constants.ScreenHeight,
    }

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

    // Update only entities within or near the viewport
    for _, entity := range g.Entities {
        // Check if entity implements Bounded interface
        if bounded, ok := entity.(interface{ GetBounds() rl.Rectangle }); ok {
            bounds := bounded.GetBounds()

            // Check if entity is visible or close to viewport
            if util.CheckRectangleOverlap(bounds, viewportWithPadding) {
                if updatable, ok := entity.(interface{ Update(float32) }); ok {
                    updatable.Update(dt)
                }
            }
        } else {
            // If entity doesn't implement GetBounds, update it anyway
            if updatable, ok := entity.(interface{ Update(float32) }); ok {
                updatable.Update(dt)
            }
        }
    }

    // Similar logic for drawing entities
```

```
    // ...
}
```

1.3 Player Control Refinement

Enhanced Helicopter Physics

go

```go
// entities/player.go
package entities

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

type Player struct {
    Position       rl.Vector2
    Velocity       rl.Vector2
    Rotation       float32
    TargetRotation float32
    Scale          float32
    Size           float32

    // Physics parameters
    Acceleration   float32
    MaxSpeed       float32
    RotationSpeed  float32
    Friction       float32

    // Visual components
    MainRotor      *RotorComponent
    TailRotor      *RotorComponent

    // Effects
    ExhaustParticles *ParticleEmitter

    // Player state
    Health         float32
    Score          int
    IsInvulnerable bool
}

type RotorComponent struct {
    Rotation      float32
    RotationSpeed float32
    Size          float32
    Offset        rl.Vector2
}

func NewPlayer() *Player {
    player := &Player{
```

```go
        Position:       rl.Vector2{X: constants.WorldWidth / 2, Y: constants.WorldHeight / 2},
        Velocity:       rl.Vector2{X: 0, Y: 0},
        Rotation:       0,
        TargetRotation: 0,
        Scale:          1.0,
        Size:           20.0,

        // Physics tuning
        Acceleration:   500.0,
        MaxSpeed:       200.0,
        RotationSpeed:  5.0,
        Friction:       0.95,

        Health:         100.0,
        Score:          0,
        IsInvulnerable: false,
    }

    // Set up rotors
    player.MainRotor = &RotorComponent{
        Rotation:      0,
        RotationSpeed: 15.0,
        Size:          30.0,
        Offset:        rl.Vector2{X: 0, Y: 0},
    }

    player.TailRotor = &RotorComponent{
        Rotation:      0,
        RotationSpeed: 25.0,
        Size:          15.0,
        Offset:        rl.Vector2{X: -18, Y: 0},
    }

    // Setup particle emitter for exhaust
    player.ExhaustParticles = NewParticleEmitter(
        rl.Vector2{X: -20, Y: 0},  // Offset from helicopter center
        rl.Gray,                   // Base color
        0.5,                       // Particle size
        20,                        // Particles per second
        2.0,                       // Particle lifetime
    )

    return player
}

func (p *Player) Update(dt float32) {
    // Handle input
```

```go
moveDir := rl.Vector2{X: 0, Y: 0}

if rl.IsKeyDown(rl.KeyW) || rl.IsKeyDown(rl.KeyUp) {
    moveDir.Y = -1
}
if rl.IsKeyDown(rl.KeyS) || rl.IsKeyDown(rl.KeyDown) {
    moveDir.Y = 1
}
if rl.IsKeyDown(rl.KeyA) || rl.IsKeyDown(rl.KeyLeft) {
    moveDir.X = -1
}
if rl.IsKeyDown(rl.KeyD) || rl.IsKeyDown(rl.KeyRight) {
    moveDir.X = 1
}

// Normalize direction if moving diagonally
length := float32(math.Sqrt(float64(moveDir.X*moveDir.X + moveDir.Y*moveDir.Y)))
if length > 0 {
    moveDir.X /= length
    moveDir.Y /= length

    // Update target rotation based on movement direction
    p.TargetRotation = float32(math.Atan2(float64(moveDir.Y), float64(moveDir.X))) * 180 /
}

// Smoothly rotate toward target direction
angleDiff := p.TargetRotation - p.Rotation

// Normalize angle to [-180, 180]
for angleDiff > 180 {
    angleDiff -= 360
}
for angleDiff < -180 {
    angleDiff += 360
}

// Apply rotation with smoothing
p.Rotation += angleDiff * p.RotationSpeed * dt

// Apply acceleration in moving direction
if length > 0 {
    p.Velocity.X += moveDir.X * p.Acceleration * dt
    p.Velocity.Y += moveDir.Y * p.Acceleration * dt
}

// Apply friction
p.Velocity.X *= p.Friction
```

```go
    p.Velocity.Y *= p.Friction

    // Limit top speed
    speed := float32(math.Sqrt(float64(p.Velocity.X*p.Velocity.X + p.Velocity.Y*p.Velocity.Y)))
    if speed > p.MaxSpeed {
        p.Velocity.X = (p.Velocity.X / speed) * p.MaxSpeed
        p.Velocity.Y = (p.Velocity.Y / speed) * p.MaxSpeed
    }

    // Update position
    p.Position.X += p.Velocity.X * dt
    p.Position.Y += p.Velocity.Y * dt

    // Constrain to world bounds
    p.Position.X = util.Clamp(p.Position.X, p.Size, constants.WorldWidth-p.Size)
    p.Position.Y = util.Clamp(p.Position.Y, p.Size, constants.WorldHeight-p.Size)

    // Update rotors
    p.MainRotor.Rotation += p.MainRotor.RotationSpeed * speed/p.MaxSpeed * dt * 360
    p.TailRotor.Rotation += p.TailRotor.RotationSpeed * speed/p.MaxSpeed * dt * 360

    // Update particle emitter
    exhaustPos := p.Position
    exhaustOffset := rl.Vector2{
        X: float32(math.Cos(float64(p.Rotation) * math.Pi / 180)) * -20,
        Y: float32(math.Sin(float64(p.Rotation) * math.Pi / 180)) * -20,
    }
    exhaustPos.X += exhaustOffset.X
    exhaustPos.Y += exhaustOffset.Y

    p.ExhaustParticles.Position = exhaustPos
    p.ExhaustParticles.EmissionRate = 5 + speed/p.MaxSpeed*15 // More particles at higher speec
    p.ExhaustParticles.Update(dt)
}

func (p *Player) Draw() {
    // Draw exhaust particles first (behind helicopter)
    p.ExhaustParticles.Draw()

    // Draw helicopter body
    rl.DrawCircleV(p.Position, p.Size, rl.Red)

    // Draw helicopter body as a rectangle
    bodyRect := rl.Rectangle{
        X:      p.Position.X - p.Size,
        Y:      p.Position.Y - p.Size/2,
        Width:  p.Size * 2,
```

```
    Height: p.Size,
}

// Draw rotated body
rl.DrawRectanglePro(bodyRect,
                   rl.Vector2{X: p.Size, Y: p.Size/2},
                   p.Rotation,
                   rl.Red)

// Draw main rotor
rotorPos := p.Position
rotorLength := p.MainRotor.Size

rl.DrawLineEx(
    rl.Vector2{X: rotorPos.X - rotorLength * float32(math.Cos(float64(p.MainRotor.Rotation)
              Y: rotorPos.Y - rotorLength * float32(math.Sin(float64(p.MainRotor.Rotation)*
    rl.Vector2{X: rotorPos.X + rotorLength * float32(math.Cos(float64(p.MainRotor.Rotation)
              Y: rotorPos.Y + rotorLength * float32(math.Sin(float64(p.MainRotor.Rotation)*
    2.0,
    rl.White)

// Draw perpendicular rotor line
rl.DrawLineEx(
    rl.Vector2{X: rotorPos.X - rotorLength * float32(math.Cos(float64(p.MainRotor.Rotation+
              Y: rotorPos.Y - rotorLength * float32(math.Sin(float64(p.MainRotor.Rotation+9
    rl.Vector2{X: rotorPos.X + rotorLength * float32(math.Cos(float64(p.MainRotor.Rotation+
              Y: rotorPos.Y + rotorLength * float32(math.Sin(float64(p.MainRotor.Rotation+9
    2.0,
    rl.White)

// Draw tail rotor
tailOffset := rl.Vector2{
    X: -p.Size * 1.5 * float32(math.Cos(float64(p.Rotation)*math.Pi/180)),
    Y: -p.Size * 1.5 * float32(math.Sin(float64(p.Rotation)*math.Pi/180)),
}

tailRotorPos := rl.Vector2{
    X: p.Position.X + tailOffset.X,
    Y: p.Position.Y + tailOffset.Y,
}

tailRotorLength := p.TailRotor.Size / 2

rl.DrawLineEx(
    rl.Vector2{X: tailRotorPos.X - tailRotorLength * float32(math.Cos(float64(p.TailRotor.R
              Y: tailRotorPos.Y - tailRotorLength * float32(math.Sin(float64(p.TailRotor.Ro
    rl.Vector2{X: tailRotorPos.X + tailRotorLength * float32(math.Cos(float64(p.TailRotor.R
```

```go
                    Y: tailRotorPos.Y + tailRotorLength * float32(math.Sin(float64(p.TailRotor.Rc
        1.0,
        rl.White)
}

func (p *Player) 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,
    }
}

// Particle Emitter implementation for helicopter effects
type Particle struct {
    Position  rl.Vector2
    Velocity  rl.Vector2
    Color     rl.Color
    Size      float32
    Lifetime  float32
    MaxLife   float32
    Active    bool
}

type ParticleEmitter struct {
    Position     rl.Vector2
    BaseColor    rl.Color
    ParticleSize float32
    EmissionRate float32
    ParticleLife float32

    particles    []Particle
    timeSinceLastEmit float32
}

func NewParticleEmitter(position rl.Vector2, color rl.Color, size float32, emissionRate float32
    return &ParticleEmitter{
        Position:     position,
        BaseColor:    color,
        ParticleSize: size,
        EmissionRate: emissionRate,
        ParticleLife: lifetime,
        particles:    make([]Particle, 100), // Preallocate 100 particles
    }
}
```

```go
func (e *ParticleEmitter) Update(dt float32) {
    // Update existing particles
    for i := range e.particles {
        if e.particles[i].Active {
            e.particles[i].Position.X += e.particles[i].Velocity.X * dt
            e.particles[i].Position.Y += e.particles[i].Velocity.Y * dt
            e.particles[i].Lifetime -= dt

            // Fade out particle as it ages
            alpha := e.particles[i].Lifetime / e.particles[i].MaxLife
            e.particles[i].Color.A = uint8(255 * alpha)

            // Deactivate expired particles
            if e.particles[i].Lifetime <= 0 {
                e.particles[i].Active = false
            }
        }
    }

    // Emit new particles
    e.timeSinceLastEmit += dt
    particlesToEmit := e.EmissionRate * dt

    // Handle fractional particles with a time accumulator
    for particlesToEmit > 0 {
        if int(particlesToEmit) > 0 || rand.Float32() < particlesToEmit {
            // Find inactive particle
            for i := range e.particles {
                if !e.particles[i].Active {
                    // Random velocity in a backward cone
                    angle := math.Pi + (rand.Float32() - 0.5) * math.Pi / 2
                    speed := 20.0 + rand.Float32() * 10.0

                    e.particles[i] = Particle{
                        Position: e.Position,
                        Velocity: rl.Vector2{
                            X: float32(math.Cos(float64(angle))) * speed,
                            Y: float32(math.Sin(float64(angle))) * speed,
                        },
                        Color:    e.BaseColor,
                        Size:     e.ParticleSize * (0.7 + rand.Float32()*0.6),
                        Lifetime: e.ParticleLife * (0.8 + rand.Float32()*0.4),
                        MaxLife:  e.ParticleLife * (0.8 + rand.Float32()*0.4),
                        Active:   true,
                    }
                    break
                }
```

```
            }
        }
        particlesToEmit--
    }
}

func (e *ParticleEmitter) Draw() {
    for i := range e.particles {
        if e.particles[i].Active {
            rl.DrawCircleV(e.particles[i].Position, e.particles[i].Size, e.particles[i].Color)
        }
    }
}
```

Now let's create the `Food` entity implementation for our growth system:

go

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

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

type Food struct {
	Position  rl.Vector2
	Color     rl.Color
	Size      float32
	Value     int
	PulseTime float32
	Rotation  float32
}

func NewFood(position rl.Vector2) *Food {
	// Generate random size (which affects value)
	size := 3.0 + rand.Float32()*4.0

	// Value is proportional to size
	value := int(size * 2)

	// Random color based on value
	var color rl.Color
	switch {
	case value < 8:
		color = rl.Green
	case value < 12:
		color = rl.Blue
	default:
		color = rl.Gold
	}

	return &Food{
		Position:  position,
		Color:     color,
		Size:      size,
		Value:     value,
		PulseTime: rand.Float32() * 2 * math.Pi, // Random start phase
		Rotation:  0,
	}
}
```

```go
func (f *Food) Update(dt float32) {
    // Simple pulsing animation
    f.PulseTime += dt * 2
    if f.PulseTime > 2*math.Pi {
        f.PulseTime -= 2 * math.Pi
    }

    // Slow rotation
    f.Rotation += dt * 30
    if f.Rotation > 360 {
        f.Rotation -= 360
    }
}

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

    // Draw a small polygon for variety instead of just a circle
    centerX := f.Position.X
    centerY := f.Position.Y
    sides := 5 // Pentagon

    // Draw a polygon with rotation
    angles := make([]float32, sides)
    for i := 0; i < sides; i++ {
        angle := f.Rotation*math.Pi/180 + float32(i)*2*math.Pi/float32(sides)
        angles[i] = angle
    }

    for i := 0; i < sides; i++ {
        j := (i + 1) % sides
        x1 := centerX + drawSize * float32(math.Cos(float64(angles[i])))
        y1 := centerY + drawSize * float32(math.Sin(float64(angles[i])))
        x2 := centerX + drawSize * float32(math.Cos(float64(angles[j])))
        y2 := centerY + drawSize * float32(math.Sin(float64(angles[j])))

        rl.DrawLineEx(
            rl.Vector2{X: x1, Y: y1},
            rl.Vector2{X: x2, Y: y2},
            1.5,
            f.Color)
    }
```

```go
    // Draw a small dot in the center
    rl.DrawCircleV(f.Position, drawSize*0.3, f.Color)
}

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,
    }
}
```

Now let's create a system for managing the food entities:

go

```go
// game/food_system.go
package game

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

type FoodSystem struct {
	FoodEntities []*entities.Food
	FoodPool     *entities.Pool
	MaxFood      int
	SpawnTimer   float32
	SpawnRate    float32 // Food items per second
}

func NewFoodSystem(maxFood int) *FoodSystem {
	// Create pool for food entities
	foodPool := entities.NewPool(
		func() interface{} {
			return &entities.Food{}
		},
		maxFood,
	)

	return &FoodSystem{
		FoodEntities: make([]*entities.Food, 0, maxFood),
		FoodPool:     foodPool,
		MaxFood:      maxFood,
		SpawnTimer:   0,
		SpawnRate:    5.0, // 5 food items per second
	}
}

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

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

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

```go
        // Create new food at random position
        randomX := rand.Float32() * constants.WorldWidth
        randomY := rand.Float32() * constants.WorldHeight

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

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

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

func (fs *FoodSystem) Draw(camera rl.Camera2D) {
    // Draw all food entities
    for _, food := range fs.FoodEntities {
        // Only draw if within or near camera view
        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,
        }

        // 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()
        }
    }
}

func (fs *FoodSystem) CheckPlayerCollisions(player *entities.Player, quadtree *util.Quadtree) {
    // Get potential food collisions from quadtree
    potentialCollisions := make([]interface{}, 0, 20)
    quadtree.Root.Query(player.GetBounds(), &potentialCollisions)
```

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

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

            // Player size increases with growth, so use that for collision
            minDist := player.Size + food.Size

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

                // Apply effects to player
                player.Score += food.Value

                // Increase player size based on food value
                // (make sure to balance this carefully)
                growthFactor := 0.01 * float32(food.Value)
                player.Size += growthFactor

                // Optional: Increase max speed slightly
                player.MaxSpeed += 0.1 * float32(food.Value)

                // Might want to cap these values
                if player.Size > 40 {
                    player.Size = 40
                }
                if player.MaxSpeed > 300 {
                    player.MaxSpeed = 300
                }

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

                // Create pickup effect (if implemented)
                // effectsSystem.CreatePickupEffect(food.Position)
            }
        }
    }
}
```

```go
func (fs *FoodSystem) RemoveFood(food *entities.Food) {
    // Find and remove food from list
    for i, f := range fs.FoodEntities {
        if f == food {
            // Return to pool
            fs.FoodPool.Return(food)

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

Now, let's integrate all of these components into the main game loop:

go

```go
// game/game.go
package game

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

type Game struct {
	Player      *entities.Player
	Camera      *Camera
	Quadtree    *util.Quadtree
	FoodSystem  *FoodSystem
	Minimap     *ui.Minimap
	Score       int
	GameTime    float32
}

func New() *Game {
	// Initialize player
	player := entities.NewPlayer()

	// Initialize camera focused on player
	camera := NewCamera(player)

	// Create quadtree for collision detection
	quadtree := util.NewQuadtree(rl.Rectangle{
		X:      0,
		Y:      0,
		Width:  constants.WorldWidth,
		Height: constants.WorldHeight,
	})

	// Create food system
	foodSystem := NewFoodSystem(500) // Allow up to 500 food items

	// Initialize 200 food items to start with
	for i := 0; i < 200; i++ {
		randomX := rand.Float32() * constants.WorldWidth
		randomY := rand.Float32() * constants.WorldHeight

		food := entities.NewFood(rl.Vector2{X: randomX, Y: randomY})
```

```go
        foodSystem.FoodEntities = append(foodSystem.FoodEntities, food)

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

    // Create minimap
    minimap := ui.NewMinimap()

    return &Game{
        Player:     player,
        Camera:     camera,
        Quadtree:   quadtree,
        FoodSystem: foodSystem,
        Minimap:    minimap,
        Score:      0,
        GameTime:   0,
    }
}

func (g *Game) Update(dt float32) {
    // Update game time
    g.GameTime += dt

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

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

    // Add food entities to quadtree
    for _, food := range g.FoodSystem.FoodEntities {
        g.Quadtree.Root.Insert(food, food)
    }

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

    // Update food system
    g.FoodSystem.Update(dt, g.Quadtree)

    // Check for player-food collisions
    g.FoodSystem.CheckPlayerCollisions(g.Player, g.Quadtree)

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

```go
func (g *Game) Draw() {
    rl.BeginDrawing()
    rl.ClearBackground(rl.Black)

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

    // Draw world boundaries
    borderColor := rl.DarkGray
    borderThickness := constants.WorldBorderThickness

    // Top border
    rl.DrawRectangle(0, 0, constants.WorldWidth, borderThickness, borderColor)

    // Bottom border
    rl.DrawRectangle(0, constants.WorldHeight - borderThickness, constants.WorldWidth, borderTh

    // Left border
    rl.DrawRectangle(0, 0, borderThickness, constants.WorldHeight, borderColor)

    // Right border
    rl.DrawRectangle(constants.WorldWidth - borderThickness, 0, borderThickness, constants.Worl

    // Draw background grid for visual reference
    gridSize := 200
    gridColor := rl.Color{R: 20, G: 20, B: 20, A: 255}

    for x := 0; x < constants.WorldWidth; x += gridSize {
        rl.DrawLine(x, 0, x, constants.WorldHeight, gridColor)
    }

    for y := 0; y < constants.WorldHeight; y += gridSize {
        rl.DrawLine(0, y, constants.WorldWidth, y, gridColor)
    }

    // Draw all food entities
    g.FoodSystem.Draw(g.Camera.GetRLCamera2D())

    // Draw player
    g.Player.Draw()

    rl.EndMode2D()

    // Draw UI elements

    // Draw score
```

```go
    rl.DrawText(TextFormat("Score: %d", g.Player.Score), 20, 20, 20, rl.White)

    // Draw time
    minutes := int(g.GameTime) / 60
    seconds := int(g.GameTime) % 60
    rl.DrawText(TextFormat("Time: %02d:%02d", minutes, seconds), 20, 50, 20, rl.White)

    // Draw size indicator
    rl.DrawText(TextFormat("Size: %.1f", g.Player.Size), 20, 80, 20, rl.White)

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

    rl.EndDrawing()
}

func (g *Game) Cleanup() {
    // Any cleanup needed when game ends
}

// Helper function for string formatting (similar to C's printf)
func TextFormat(format string, args ...interface{}) string {
    return fmt.Sprintf(format, args...)
}
```

Finally, let's update the main.go file to use our enhanced game:

go

```go
// main.go
package main

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

func main() {
	// Seed the random number generator
	rand.Seed(time.Now().UnixNano())

	// Initialize window
	rl.SetConfigFlags(rl.FlagVsyncHint)
	rl.InitWindow(constants.ScreenWidth, constants.ScreenHeight, "Atomblaster 2.5D")
	rl.SetTargetFPS(60)

	// Initialize audio
	rl.InitAudioDevice()

	// Create game state
	gameState := game.New()

	// Main game loop
	for !rl.WindowShouldClose() {
		// Calculate delta time
		dt := rl.GetFrameTime()

		// Update game state
		gameState.Update(dt)

		// Draw game
		gameState.Draw()
	}

	// Clean up
	gameState.Cleanup()
	rl.CloseAudioDevice()
	rl.CloseWindow()
}
```

This completes the detailed implementation for Phase 1 of our helicopter game development. With this code, you'll have:

1. A greatly expanded game world with a camera that follows the player
2. A quadtree for efficient collision detection to support many entities
3. Enhanced helicopter physics with satisfying movement
4. Visual feedback with particle effects and animated rotors
5. A basic food system that allows the helicopter to grow
6. A minimap to navigate the larger world

The next phase would focus on enhancing the power-up system and special abilities, but this implementation gives you a solid foundation to start with.