Asteroids - Final Implementation Design Document

Mariam Bakkali Kasmi (9186603) & Taha Charef (8947597)

1. Introduction

A single-player arcade shooter (Asteroids) with wave/level progression inspired by COD Zombies. The player controls a spaceship at the bottom of the screen, shooting upward while avoiding and destroying enemies and asteroids. Each wave increases in difficulty, introducing stronger enemies and faster spawn rates. The game features a variety of power-ups that can be collected to temporarily modify gameplay mechanics.

1.1 Implemented Features

The final implementation includes wave-based progression where defeating a required number of enemies automatically starts the next, harder wave. Each wave increases difficulty through:

- Faster enemy movement
- Increased spawn rates
- Higher maximum simultaneous enemies
- More varied enemy type distribution

Power-ups drop from defeated enemies, particularly Martians which guarantee a drop on hit. Available power-ups include:

- Health restoration
- Temporary insta-kill capability
- Screen-clearing nukes
- Score-boosting star pickups

2. Design

2.1 Data Structures

```
type Vec = (Float, Float)
type Seconds = Float
newtype EntityId = EntityId Int
  deriving (Eq, Ord, Show, Num)

-- | Main game state
data GameState = GameState
  { gsPlayer        :: Player
  , gsEnemies       :: [Enemy]
  , gsAsteroids     :: [Asteroid]
  , gsBullets       :: [Bullet]
  , gsPickups       :: [Pickup]
  , gsWave          :: WaveState
  , gsScore         :: Int
  , gsTimeElapsed   :: Seconds
```

```haskell
  , gsRandGen       :: StdGen
  , gsPaused        :: Bool
  , gsNextId        :: EntityId
  , gsGameOver      :: Bool
  , gsGameMode      :: GameMode
  , gsHighScores    :: [HighScore]
  , gsPlayerName    :: String
  } deriving (Show, Generic)

-- | Game mode states
data GameMode = Menu | Playing | GameOverScreen | EnteringName
  deriving (Show, Eq, Generic)

-- | High score entry
data HighScore = HighScore
  { hsName  :: String
  , hsScore :: Int
  } deriving (Show, Read, Generic)

-- | Player state
data Player = Player
  { pId           :: EntityId
  , pPos          :: Vec
  , pVel          :: Vec
  , pHealth       :: Int
  , pMaxHealth    :: Int
  , pFireCooldown :: Seconds
  , pPowerups     :: [ActivePowerup]
  } deriving (Show, Generic)

-- | Enemy types and state
data EnemyType = Alien | Martian | UFO
  deriving (Eq, Show, Enum, Bounded)

data Enemy = Enemy
  { eId      :: EntityId
  , ePos     :: Vec
  , eVel     :: Vec
  , eHealth  :: Int
  , eType    :: EnemyType
  } deriving (Show, Generic)

-- | Asteroid state
data Asteroid = Asteroid
  { aId     :: EntityId
  , aPos    :: Vec
  , aVel    :: Vec
  , aSize   :: AsteroidSize
  } deriving (Show, Generic)

data AsteroidSize = Large | Small
  deriving (Eq, Show, Ord, Enum, Bounded)

-- | Bullet state with owner tracking
```

```haskell
data Bullet = Bullet
  { bId     :: EntityId
  , bPos    :: Vec
  , bVel    :: Vec
  , bOwner  :: BulletOwner
  , bDamage :: Int
  } deriving (Show, Generic)

data BulletOwner = FromPlayer | FromEnemy EntityId
  deriving (Show, Eq, Generic)

-- | Pickup system
data PickupType = HealthDrop Int
                | InstaKill Seconds
                | Nuke
                | ScoreStar Int
                deriving (Show, Generic)

data Pickup = Pickup
  { puId    :: EntityId
  , puPos   :: Vec
  , puVel   :: Vec
  , puType  :: PickupType
  , puTTL   :: Maybe Seconds
  } deriving (Show, Generic)

-- | Active powerups
data ActivePowerup = ActivePowerup
  { apType      :: PickupType
  , apRemaining :: Seconds
  } deriving (Show, Generic)

-- | Wave progression
data WaveState = WaveState
  { waveNumber     :: Int
  , killsThisWave  :: Int
  , killsRequired  :: Int
  , spawnCooldown  :: Seconds
  , spawnTimer     :: Seconds
  , maxEnemies     :: Int
  } deriving (Show, Generic)
```

3. Implementation Details

3.1 Player System

The player controls a spaceship that moves horizontally at the bottom of the screen. Movement is restricted to the horizontal axis to maintain gameplay focus and simplify controls.

Movement Implementation

Movement is handled through velocity-based updates with boundary clamping:

```haskell
updatePlayer :: Float -> GameState -> GameState
updatePlayer dt gstate = gstate { gsPlayer = newPlayer }
  where
    player = gsPlayer gstate
    (px, py) = pPos player
    (vx, vy) = pVel player

    newX = clamp (-screenWidth/2 + 20) (screenWidth/2 - 20) (px + vx * dt)
    newCooldown = max 0 (pFireCooldown player - dt)

    newPlayer = player
      { pPos = (newX, py)
      , pFireCooldown = newCooldown
      }
```

Control System

- A/Left Arrow: Move left
- D/Right Arrow: Move right
- Space: Shoot
- P: Pause game

3.2 Enemy System

Enemy Types and Behaviors

Alien

- Fast but weak enemies
- Health: 1
- Movement: Direct vertical descent at 120% base speed
- Points: 10
- Behavior: Moves straight down, no special abilities
- Common in early waves

Martian

- Medium-strength resilient enemies
- Health: 2
- Movement: Standard vertical descent
- Points: 25
- Special ability: Respawns at top when reaching bottom
- Guaranteed power-up drop on every hit
- More common in middle waves

UFO

- Tough enemies with special abilities
- Health: 3
- Movement: Horizontal movement with slower vertical descent
- Points: 50
- Special ability: Shoots aimed bullets at player
- More common in later waves
- Most challenging enemy type

Enemy Movement and Targeting

UFO shooting uses a sophisticated targeting system that leads shots toward the player:

```
maybeShootUFO :: Enemy -> EntityId -> Vec -> [Bullet]
maybeShootUFO enemy bulletId targetPos
  | eType enemy /= UFO = []
  | otherwise =
      let chance = sin (gsTimeElapsed gstate * 2) * 0.5 + 0.5
          shootThisFrame = chance > 0.95
          (ex, ey) = ePos enemy
          (px, py) = targetPos
          dx = px - ex
          dy = py - ey
          len = sqrt (dx * dx + dy * dy)
          bulletVel = if len == 0
                      then (0, -200)
                      else (dx / len * 200, dy / len * 200)
      in if shootThisFrame
         then [Bullet
                { bId = bulletId
                , bPos = ePos enemy
                , bVel = bulletVel
                , bOwner = FromEnemy (eId enemy)
                , bDamage = 1
                }]
         else []
```

3.3 Asteroid System

The asteroid system was simplified from the original three-size design to a two-size system for better gameplay balance and clearer visual feedback.

Implementation Details

- Asteroids spawn at the top of the screen
- Large asteroids split into two small asteroids when shot
- Small asteroids are destroyed when shot
- Damage values:
  - Large: 30 damage on collision
  - Small: 10 damage on collision

- Point values:
  - Large: 10 points
  - Small: 30 points

Asteroid Movement

Asteroids feature a unique horizontal wrapping movement:

```
wrapHorizontal :: Vec -> Vec
wrapHorizontal (x, y)
  | x < -screenWidth/2 = (screenWidth/2, y)
  | x > screenWidth/2 = (-screenWidth/2, y)
  | otherwise = (x, y)
```

3.4 Power-up System

Types and Effects

1. Health Drop (HealthDrop 25)
   - Restores 25 HP instantly
   - Cannot exceed maximum health (100)
   - Medium spawn chance
2. Insta-Kill (InstaKill 5.0)
   - Duration: 5 seconds
   - Makes all bullets instantly kill enemies
   - Affects both enemies and asteroids
   - Rare spawn
3. Nuke
   - Instantly removes all enemies and asteroids
   - Awards points for each entity destroyed
   - Rare spawn
   - Strategic screen-clearing tool
4. Score Star (ScoreStar 50)
   - Instant 50 point bonus
   - Most common power-up
   - No other effects

Spawn Mechanics

```
maybeSpawnPickup :: Vec -> StdGen -> (Maybe Pickup, StdGen)
maybeSpawnPickup pos gen =
```

```
    let (chance, gen') = randomR (0, 1) gen :: (Float, StdGen)
    in if chance < 0.2
       then let (pickupType, gen'') = randomPickupType gen'
                pickup = Pickup
                  { puId = 9999
                  , puPos = pos
                  , puVel = (0, -50)
                  , puType = pickupType
                  , puTTL = Just 10.0
                  }
            in (Just pickup, gen'')
       else (Nothing, gen')
```

3.5 Wave System

Wave Mechanics

- Each wave requires a specific number of kills to progress
- Required kills increase with wave number
- Enemy type distribution changes with wave number
- Wave difficulty increases through:
  - Faster enemies
  - More simultaneous enemies
  - More challenging enemy types
  - Faster spawn rates

```
checkWaveCompletion :: GameState -> GameState
checkWaveCompletion gstate
  | killsThisWave wave >= killsRequired wave = gstate
      { gsWave = nextWave }
  | otherwise = gstate
  where
    wave = gsWave gstate
    wNum = waveNumber wave + 1
    nextWave = WaveState
      { waveNumber = wNum
      , killsThisWave = 0
      , killsRequired = 5 + wNum * 2
      , spawnCooldown = max 0.5 (2.0 - fromIntegral wNum * 0.1)
      , spawnTimer = 0.0
      , maxEnemies = min 10 (3 + wNum)
      }
```

3.6 Collision System

Implementation

A comprehensive collision system handling all entity interactions:

```haskell
updateCollisions :: GameState -> GameState
updateCollisions gstate = gstate
  & handleBulletEnemyCollisions
  & handleBulletAsteroidCollisions
  & handlePlayerPickupCollisions
  & handleEnemyPlayerCollisions
  & handleAsteroidPlayerCollisions
  & handleEnemyBulletPlayerCollisions
```

Collision Types and Effects

1. Bullet-Enemy
    - Deals damage based on bullet owner
    - May spawn power-ups
    - Updates score and wave kills
2. Bullet-Asteroid
    - Splits large asteroids
    - Destroys small asteroids
    - Awards points
3. Player-Enemy
    - Deals 10 damage per enemy
    - Immediate collision response
4. Player-Asteroid
    - Damage based on asteroid size
    - Large: 30 damage
    - Small: 10 damage
5. Player-Pickup
    - Immediate effect application
    - Pickup removal
6. Enemy Bullet-Player
    - Deals 1 damage per hit
    - Bullet removal on hit

3.7 High Score System

Implementation

The high score system maintains persistent top scores:

```haskell
saveHighScore :: GameState -> IO GameState
saveHighScore gstate = do
  let newScore = HighScore (gsPlayerName gstate) (gsScore gstate)
  let updatedScores = take 10 $ reverse $ sort' $ newScore : gsHighScores gstate
  writeFile "src/highscores.txt" (show updatedScores)
  return $ gstate
    { gsHighScores = updatedScores
    , gsGameMode = Menu
    , gsPlayerName = ""
    }
```

Features:

- Stores top 10 scores
- Saves player names
- Persists between sessions
- Automatic sorting
- File-based storage

4. Pure and Impure Operations

4.1 Pure Functions

The core game logic is implemented purely:

- Entity updates
- Collision detection
- Score calculation
- Wave progression
- Power-up effects
- Movement calculations
- Spawn logic (excluding RNG)

4.2 Impure Operations

IO is strictly limited to:

- File operations (high scores)
- Initial random seed generation
- Game initialization
- Event handling (though handlers are pure)

5. Game Constants

```haskell
screenWidth, screenHeight :: Float
screenWidth = 1024
screenHeight = 768
```

```haskell
playerSpeed :: Float
playerSpeed = 300.0

playerFireRate :: Seconds
playerFireRate = 0.2

bulletSpeed :: Float
bulletSpeed = 500.0

enemySpeed :: Float
enemySpeed = 80.0

asteroidSpeed :: Float
asteroidSpeed = 50.0
```

6. Differences from Original Design

6.1 Simplified Features

1. Movement System
    - Removed vertical player movement
    - Removed ship rotation
    - Added screen boundary clamping
2. Asteroid System
    - Reduced from three sizes to two
    - Simplified splitting mechanics
    - Added horizontal wrapping
3. Animation
    - Removed explosion effects
    - Simplified visual feedback
    - Focused on core gameplay

6.2 Enhanced Features

1. Enemy System
    - Added UFO shooting mechanics
    - Enhanced enemy type distribution
    - Improved spawn patterns
2. Power-up System
    - Added duration tracking
    - Enhanced pickup spawning
    - Improved visual feedback
3. Game States

    - Added proper menu system
    - Enhanced score system
    - Added name entry

7. Future Enhancements

7.1 Visual Improvements

- Explosion animations
- Power-up effects
- Bullet trails
- Background effects

7.2 Gameplay Extensions

- Additional enemy types
- Boss battles
- Special waves
- Alternative game modes

7.3 Technical Improvements

- Sound effects
- Background music
- Particle systems
- Achievement tracking

7.4 User Experience

- Tutorial system
- Difficulty settings
- Control remapping
- Statistics tracking