Asteroids - Game design

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

1. Introduction

Single-player arcade shooter (Asteroids) with wave/level progression inspired by COD Zombies: the player survives waves of enemies/asteroids; finishing a wave spawns the next, harder wave. The player moves horizontally along the bottom and shoots upward; enemies/asteroids spawn and move around the field.

1.1 Twist

After you kill the required number of enemies in the current wave, the next wave starts automatically. Each wave increases difficulty (spawn rate, max simultaneous enemies, speed). Occasionally, defeated enemies drop pickups (health, insta-kill, nuke, score stars). Pickups can be collected to temporarily change mechanics (e.g. insta-kill: single shot kills; nuke: clears screen). Waves are kill based like COD Zombies.

2. Design

2.1 Data Structures

```
type Vec = (Float, Float)
type Seconds = Float

newtype EntityId = EntityId Int
  deriving (Eq, Ord, Show)

data GameState = GameState
  { gsPlayer          :: Player
  , gsEnemies         :: [Enemy]
  , gsAsteroids       :: [Asteroid]
  , gsBullets         :: [Bullet]
  , gsPickups         :: [Pickup]
  , gsWave            :: WaveState
  , gsScore           :: Int
  , gsTimeElapsed     :: Seconds
  , gsRandGen         :: StdGen -- impure RNG kept in state for
deterministic stepping
  , gsStatus          :: GameStatus
  } deriving (Generic)
```

```
data GameStatus
  = Running
  | Paused
  | GameOver GameOverInfo
  deriving (Show, Eq)

data GameOverInfo = GameOverInfo
  { goReason   :: GameOverReason } deriving (Show, Eq)

data GameOverReason = PlayerDied
  deriving (Show, Eq)

-- Player: single guaranteed player
data Player = Player
  { pId           :: EntityId
  , pPos          :: Vec
  , pVel          :: Vec
  , pHealth       :: Int
  , pFireCooldown :: Seconds       -- time until next allowed shot
  , pPowerups     :: [ActivePowerup]
  } deriving (Show, Generic)

-- Enemies
data EnemyType = Alien | Martian | UFO deriving (Eq, Show)
data Enemy = Enemy
  { eId      :: EntityId
  , ePos     :: Vec
  , eVel     :: Vec
  , eHealth  :: Int
  , eType    :: EnemyType
  } deriving (Show, Generic)

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

data AsteroidSize = Large | Medium | Small deriving (Eq, Show)

-- Bullets (used by player and enemies)
data Bullet = Bullet
  { bId     :: EntityId
  , bPos    :: Vec
  , bVel    :: Vec
  , bDamage :: Int
  } deriving (Show, Generic)
```

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

-- Pickups / Upgrades
data PickupType = HealthDrop Int    -- heals X
                | InstaKill Seconds -- temporary insta-kill duration
                | Nuke              -- instant clear
                | ScoreStar Int
                deriving (Show, Generic)

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

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

-- Wave state (keeps track of wave number, kills needed, spawn
limits)
data WaveState = WaveState
  { waveNumber     :: Int
  , killsThisWave  :: Int
  , killsRequired  :: Int
  , spawnCooldown  :: Seconds -- time between spawns (ctrl. diffic.)
  , spawnTimer     :: Seconds -- countdown
  , maxEnemies     :: Int
  } deriving (Show, Generic)
```

3. Implementation - Minimum requirements

3.1 Player

The player will control a spaceship using keyboard input. The user can shoot only from the nose (front side) of the spaceship, in the direction that the ship is currently facing.

We will start with linear movement (horizontal and vertical using keys W, A, S, and D). After that, we will implement diagonal movement (combinations of keys) and velocity. Finally, we will add support for ship rotation.

Code structure to support this requirement:

Data:
Player (single).

Controls:
keyboard events handled by

```
handleInput :: Event -> GameState -> GameState
```

Shooting:
handled by

```
playerShoot :: GameState -> (Maybe Bullet, GameState)
```

3.2 Enemies

The game will have several antagonistic characters that try to make the player lose the game. Initially, the antagonists are asteroids. As the game progresses, new enemy types such as aliens and UFOs appear. UFOs will be able to shoot bullets towards the player's current position when firing, requiring the player to actively dodge.

Code structure to support this requirement:

Data:
`Enemy` list.

Behavior:
We use a pure AI update function
`updateEnemy :: Seconds -> Enemy -> GameState -> Enemy` (enemy can reach player pos but function is pure).

Spawning:
`spawnEnemy :: StdGen -> WaveState -> (Maybe Enemy, StdGen, WaveState)`

3.3 Randomness

Enemies spawn at random locations just outside the screen boundaries. The type of enemy is chosen based on randomized weighted ratios. Additionally, power-ups/bonuses can randomly appear after killing an enemy at the position of the defeated enemy.

Code structure to support this requirement:

We put `StdGen` inside `GameState`. Spawn functions take a `StdGen` and return a new `StdGen` (pure). Example:

```
spawnAsteroid :: StdGen -> (Asteroid, StdGen)
spawnEnemy    :: StdGen -> WaveState -> (Maybe Enemy, StdGen,
WaveState)

spawnPickup  :: StdGen -> (Maybe Pickup, StdGen)
```

3.4 Animation

We will implement multiple animations. When an enemy dies, an explosion effect will appear at the location of the defeated enemy. Furthermore, when the player gets hit, the color of the player will slightly flash to make the player know that the spaceship got hit.

Code structure to support this requirement:

Main Gloss step function calls `stepGame :: Seconds -> GameState -> GameState`.

`stepGame` is pure: it advances positions, timers, resolves collisions and returns a new `GameState` (consuming/producing `StdGen` as needed).

`render :: GameState -> Picture returns the Gloss Picture`.

3.5 Pause

Our idea is to implement a `gsStatus :: GameStatus`, which turns to `Paused` when the game gets paused. At that moment, the game state stops getting updated, until the player resumes the game.

Code structure to support this requirement:

`gsStatus :: GameStatus`. Gloss event handler toggles it. `stepGame` early returns same state if paused.

3.6 Interaction with the file system (IO)

We will maintain a high score list, stored in a .txt file. The game will keep only the top 5 or top 10 scores. On game over, the current score is compared to existing ones and stored if it qualifies.

Code structure to support this requirement

> IO only in `main` or dedicated IO functions: save/load high score:
> `saveHighScore :: FilePath -> Int -> IO ()` and
> `loadHighScore :: FilePath -> IO (Maybe Int)`.
> Asset load (images) happens in `main` and converted into `Pictures` (we will store the `Pictures` in a read-only `Assets` record).

4. Implementation - Optional requirements

Planned: wave-based progression, pickups (health, insta-kill, nuke, score stars), different enemy types, nukes, temporary powerups.

Data support:

`PickupType`, `Pickup` and `ActivePowerup` support all pickups.

`WaveState` controls wave progression and difficulty scaling.

Spawn & Difficulty scaling:

A wave clenches `killsRequired` and `spawnCooldown`.

On wave complete (`killsThisWave >= killsRequired`) do

```
advanceWave :: StdGen -> WaveState -> (WaveState, StdGen)
```

which increases wave number and adjusts `spawnCooldown`, `maxEnemies`, `killsRequired` to make it more difficult.

Nuke:

`applyNuke :: GameState -> GameState` removes all enemies/asteroids and awards points.

Insta-kill:

`ActivePowerup` with `InstaKill duration` reduces all enemy hp to 1 on hit (or sets bullet damage to very large while active, we'll see what's best when we start working on it).

5. Pure and impure

Pure

All game logic: movement updates, collision detection, damage calculation, spawning decisions (but not RNG generation), powerup timers, wave progression rules. These are pure functions of `GameState`, `Seconds` and (optionally) `StdGen -> (result, StdGen)`.

Impure

IO: loading assets, reading/saving highscore files, starting Gloss main loop, obtaining initial `StdGen` from system, any file reads/writes.

Gloss callbacks (event loop) appear in `IO`, but we keep the event handlers pure functions that the IO code calls.

Concrete pattern

`main :: IO ()` does:

1. load assets (`IO Assets`)
2. `g <- getStdGen` or `newStdGen`
3. `let initialState = initGameState g`
4. `playIO` or `play` from Gloss:
    - if `playIO` used, the step/render/handle can be IO; maybe `play` with pure `render`/`step`/`handle` if we keep `StdGen` in `GameState`.
5. Save high score on exit (IO).

Example: pure step using StdGen in state

```
stepGame :: Seconds -> GameState -> GameState
stepGame dt gs
  | gsPaused gs = gs
  | otherwise   =
      let gs1 = updatePhysics dt gs
      (maybeSpawned, gen') = spawnEntitiesIfNeeded (gsRandGengs)
      (gsWave gs) dt
          gs2 = gs1 { gsRandGen = gen', gsWave = snd maybeSpawned,
      gsEnemies = maybeAddEnemy (fst maybeSpawned) (gsEnemies gs1) }
           gs3 = resolveCollisions gs2
       in gs3
```

6. Abstraction

Use typeclasses sparingly for cross-cutting behavior:

```
class HasPos a where
  getPos :: a -> Vec
  setPos :: Vec -> a -> a

class Drawable a where
  toPicture :: Assets -> a -> Picture

class Collidable a where
  hitbox :: a -> Circle  -- or Rect
```

```
  onHit  :: Int -> a -> (a, Maybe Pickup) --damage and possible drop
```

We'll implement `HasPos/Collidable for Player, Enemy, Asteroid, Bullet, Pickup.` This lets collision code be generic:

```
checkCollision :: (Collidable a, Collidable b) => a -> b -> Bool
```

Other abstractions:

`Spawner` module encapsulating spawn logic: functions `spawnEnemy`, `spawnAsteroid`, `spawnPickup`.

`Powerup` module with pure `applyPowerupToPlayer :: PickupType -> Player -> Player`.