BH-STG Architecture and Game Design

High Level Overview

When coming up with the basic architectural design for the BH-STG solution, the team decided to follow ***domain-driven design*** patterns and best practices. Figure 1.1 illustrates the high-level overview of the DDD project structure. The **Game Presentation Layer** owns the actual game project (Windows Universal Monogame Project) and acts as the client for the rest of the layers. This is the customer-facing layer in which the user playing the game directly interacts with. The **Game Logic Layer** consists of an abstract factory, an entity manager, and a main game driver. This layer is responsible for the logic behind the game, driving the creating, spawning, points, scores, lives, etc. throughout the duration the user is playing it. The **Game Domain Layer** is the most centric layer consisting of domain entities, value objects and interfaces to interact with each. Entities are objects that have an identity (in our case, an integer) while value objects do not have a specific identity. Entities are a form of aggregates where they utilize composition/association with other entities or value objects to create the final game object. The **Shared Kernel Layer** is a project that is referenced across all other layers of the architecture. Within this project, there exists the base classes for what an entity/value object is, global enums and other abstract base classes. The **Persistence Layer** is used to persist data about core domain objects to a data store via data transfer objects, domain interfaces, and repositories. This layer has an indirect reference to the **Game Domain Layer** via the **Game Logic Layer** so that it can correctly perform CRUD operations on it. The **Integration Layer** is meant to help integrate third-party solutions into the main game or extend the game without changing the core logic. Here, the JSON Parser for scripting changes to the game via a JSON file can be implemented without breaking the game or changing multiple files. The arrows represent references to lower-level projects where each

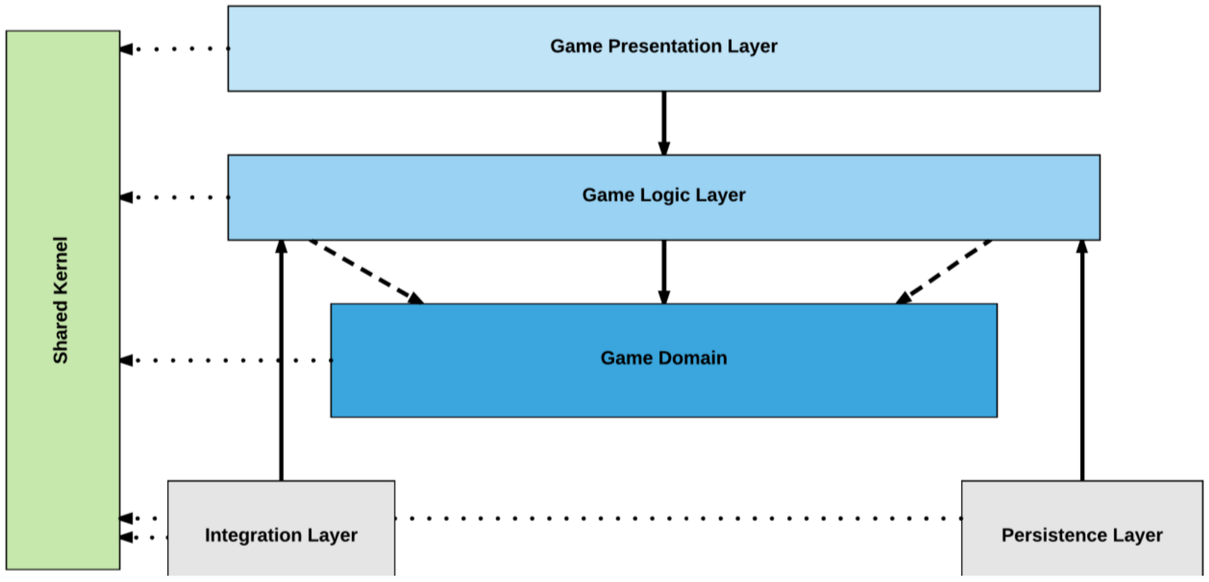

Figure 1.1: BH-STG High Level Architecture Diagram

upper level relies on the level directly below it (except for the dependency inversion implied for the integration and persistence layers). Thus, the core Domain does not need to know about the Game Logic or how it works but the Game Logic needs to know how to interact with the core Domain.

Initial Class Diagram: Domain & Shared Kernel

In Figure 1.2, the concurrent classes used in the BH-STG solution are illustrated with appropriate inheritance arrows, association arrows, etc. At the root of the objects portrayed in the game is the **Entity<TId>** class. This is an abstract base class the acts as a signature for the rest of the entities that inherit from it. Common attributes are stored as properties within the base Entity class and all inheriting classes instantiate them with appropriate values.

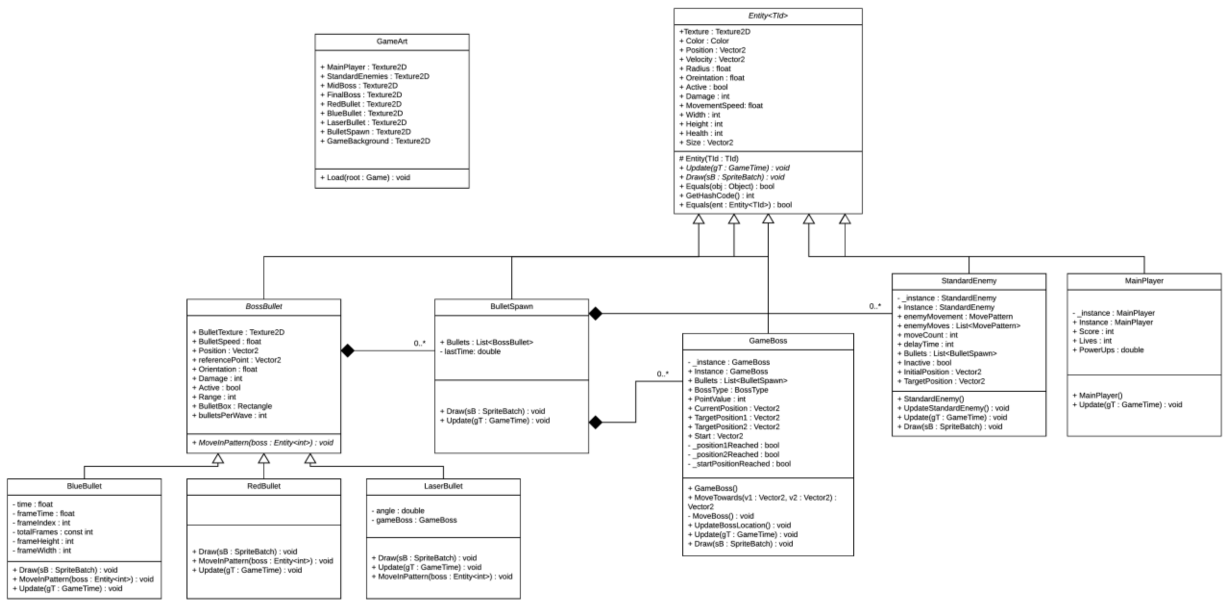
GameArt
+ MainPlayer : Texture2D
+ StandardEnemies : Texture2D
+ MidBoss : Texture2D
+ FinalBoss : Texture2D
+ RedBullet : Texture2D
+ BlueBullet : Texture2D
+ LaserBullet : Texture2D
+ BulletSpawn : Texture2D
+ GameBackground : Texture2D
+ Load(root : Game) : void
Entity<TId>
+Texture : Texture2D
+ Color : Color
+ Position : Vector2
+ Velocity : Vector2
+ Radius : float
+ Oreintation : float
+ Active : bool
+ Damage : int
+ MovementSpeed: float
+ Width : int
+ Height : int
+ Health : int
+ Size : Vector2
# Entity(TId : TId)
+ Update(gT : GameTime) : void
+ Draw(sB : SpriteBatch) : void
+ Equals(obj : Object) : bool
+ GetHashCode() : int
+ Equals(ent : Entity<TId>) : bool
BossBullet
+ BulletTexture : Texture2D
+ BulletSpeed : float
+ Position : Vector2
+ referencePoint : Vector2
+ Orientation : float
+ Damage : int
+ Active : bool
+ Range : int
+ BulletBox : Rectangle
+ bulletsPerWave : int
+ MoveInPattern(boss : Entity<int>) : void
0..*
BulletSpawn
+ Bullets : List<BossBullet>
- lastTime: double
+ Draw(sB : SpriteBatch) : void
+ Update(gT : GameTime) : void
0..*
0..*
GameBoss
- _instance : GameBoss
+ Instance : GameBoss
+ Bullets : List<BulletSpawn>
+ BossType : BossType
+ PointValue : int
+ CurrentPosition : Vector2
+ TargetPosition1 : Vector2
+ TargetPosition2 : Vector2
+ Start : Vector2
- _position1Reached : bool
- _position2Reached : bool
- _startPositionReached : bool
+ GameBoss()
+ MoveTowards(v1 : Vector2, v2 : Vector2) : Vector2
- MoveBoss() : void
+ UpdateBossLocation() : void
+ Update(gT : GameTime) : void
+ Draw(sB : SpriteBatch) : void
StandardEnemy
- _instance : StandardEnemy
+ Instance : StandardEnemy
+ enemyMovement : MovePattern
+ enemyMoves : List<MovePattern>
+ moveCount : int
+ delayTime : int
+ Bullets : List<BulletSpawn>
+ Inactive : bool
+ InitialPosition : Vector2
+ TargetPosition : Vector2
+ StandardEnemy()
+ UpdateStandardEnemy() : void
+ Update(gT : GameTime) : void
+ Draw(sB : SpriteBatch) : void
MainPlayer
- _instance : MainPlayer
+ Instance : MainPlayer
+ Score : int
+ Lives : int
+ PowerUps : double
+ MainPlayer()
+ Update(gT : GameTime) : void
BlueBullet
- time : float
- frameTime : float
- frameIndex : int
- totalFrames : const int
- frameHeight : int
- frameWidth : int
+ Draw(sB : SpriteBatch) : void
+ MoveInPattern(boss : Entity<int>) : void
+ Update(gT : GameTime) : void
RedBullet
+ Draw(sB : SpriteBatch) : void
+ MoveInPattern(boss : Entity<int>) : void
+ Update(gT : GameTime) : void
LaserBullet
- angle : double
- gameBoss : GameBoss
+ Draw(sB : SpriteBatch) : void
+ Update(gT : GameTime) : void
+ MoveInPattern(boss : Entity<int>) : void

Figure 1.2: Domain & Shared Kernel Classes – main classes used in the BH-STG solution

To extend the game, say, adding the ability for the player to shoot at enemies, this design allows us to implement the code without changing code at each level of the project. Instead, we could add a list of BulletSpawns to the MainPlayer class and change the Update() method, within the MainPlayer class, to respond to the spacebar to shoot bullets.