# Galaga SDL Game Design Document

The game has several main classes which function as managers which control several key aspects of the game. It functions as a game framework for developing the main game mechanics and level.

All the manager classes are implemented as singletons.

Timer header and source file - Used to keep track of the time between each reset. A reset is usually called after each frame. It helps control animations and the game is set to play in 60FPS. It runs in delta time so the game has the same performance independent of the computer it is played on.

MathHelper header file - Contains all the math functions that will be needed in the game. Here are all the main functions for determining rotation, translation, position and many more properties of the game entities using vector type of structure.

InputManager source and header files – Here is the player input handled which is two types: mouse and keyboard. The input handled is for the navigation in the menu, for movement and for the ship fire mechanic. It uses SDL\_Scancode predefined class.

Graphics header and source files - Handles the initialization of the graphics related SDL libraries and their release Also handles texture and text fonts loading.

AudioManager header and source files – This class enables the play of sound effects and background music using the SDL\_Mixer library.

AssetManager header and source files - Used to load all assets used in the game. Caches all loaded assets so that nothing is loaded more than once for performance and memory optimization. Each type of asset like audio file, texture or font is stored in a map data structure and loaded from it.

GameManager source and header files - Used to intialize and release all other manager objects. Contains the game loop as well as the Update and Render functions. Used to make sure all functions are called in the correct order like each step of the game update functions. Here is the collision detection handled, as well as the user input and the loading of the game assets. Here is also the rendering done.

The entity classes - Sets up a system to parent GameEntity objects to one another making the child's position, rotation, and scale relative to the parent's instead of the world origin. The game uses a Child->Parent hierarchy system which is implemented in the entity project files.

Texture header and source files - The base class for all textures to be rendered on screen. Can load full textures, or clipped textures from a spritesheet or convert a string into a texture to be rendered. There are several different types of textures which will be used – single textures, sprite sheets, fonts and Animated textures. There is a constructor for each type of texture which takes in to account all its specifications like if the animation of the animated textures will be played in a loop or only once, which element of the sprite sheet will be rendered and more.

The ScreenManager class - handles the switching between the main menu and play screen and handles the update and rendering of the game states.

1. Title Page
   1. Galaga – 2D arcade shooter
2. Game Overview  
   The player controls a ship, which must eliminate waves of enemy ships and avoid getting hit.
3. Genre – 2D arcade shooter.
4. Look and Feel – pixel art in 2D sprite sheets
5. Gameplay and Mechanics

When the game and level are started, a start level label appears on the screen and the start galaga music is played for 6 seconds after which the sidebar is updated with the current level flag and the level itself. If the game is started from the beginning the ready label is shown and after an interval it disappears.

If the game is already started and the first level was loaded a label appears showing only the current level or stage number.

The Player class handles the player ship movement and rendering. It keeps track of the player score and lives and updates the sidebar where they are stored.

The player will be able to move left or right and his input is handled by the InputManager class in the MovementHandler() function of the Player class.

It also handles the movement bounds of the player in the game space, so it prevents the player from moving outside the play area bounds (off screen).

The WasHit() function in the Player class handles the animation when the player is hit if there are any collisions between the player ship and other objects (game entities) in the game scene and subtracts from the player lives and the explosion sound effect is played.

The collision detection itself is done in the CollisionHandler() function of the Level class. Here also the side bar is being updated according to the available player lives.

The PlayerDeathHandler() function handles the player death sequence. If the player is hit a check is made if he has more lives left and if yes he is removed from the screen and after a timer runs out he is respawn in the level.

If the player has no more lives, the game state is switched to game over state and the “GAME-OVER” label is displayed on the screen.

Also here are the different states of the level stored, which are: running, finished and gameover in the enum object LEVEL\_STATES.

The player fires projectiles at the enemies with the ‘SPACE’ button. The player will be able to fire only two bullets at a time. The bullet logic is implemented in the Bullet class. If the bullet object goes out of the screen bounds it is deactivated and reloaded again in the Reload() function. This is implemented in that way for memory save and to reuse the same object, which in this case is a texture.

The firing itself is implemented in the FirignHandler() function in the Player class.

The enemy behavior and logic is implemented in the Enemy class.

Enemy formations and movements are done with Bezier Curves.

Each enemy type inherits this class.

A Bezier Curve structure is defined in the MathHelper file. In the BezierPath class the AddCurve() function is defined and implemented, which enables the addition of curves, with which we can define the movement of the enemies on the screen. The enemy has 4 states:

1. flyIn - this is the first state in which we use the Bezier curves so they can form on a formation at the start of the level.
2. formation - this is when the formation is formed and they start the spread animation in which they spread one from another and then form a closer formation again.
3. dive – in this state the 3 enemy types dive to the player. Each type has a unique dive path and the boss and butterfly types have an escort type dive where 2 of the butterflies in the formation follow the boss when he dives. The dive paths are implemented in the Dive() and CreatePaths() functions in each enemy class, which inherit the base Enemy::Dive() function.
4. dead – the enemies are hit by the player and are destroyd, the animation playes, score is added to the player and if it is a boss type, he has two lives and after the first hit the color is changed.

All the states and their functionalities are handled in the StatesHandler() function of the Enemy class. At the beginning, when an enemy is created, a path for it is created with the CreatePaths() function where the AddCurve() function of the BezierPath class is utilized.

The Formation class handles the enemy formations itself and how they align. Each enemy type has a unique formation.

* + - There are 3 types of enemies in the level:

1. Butterflies – This type of enemy is implemented and defined in the Butterfly class.
2. Wasps – This type of enemy is implemented and defined in the Wasp class.
3. Boss types – This type of enemy is implemented and defined in the Boss class.

Every enemy type inherits the Enemy class and has unique formation patterns and dive capabilities.

Once the formation is former the breath animation is played and the enemies are animated by loading 2 different textures for each enemy type and switching between it.

All enemy types are stored in the enum in the Enemy class and can be accessed from it.

The wasp enemy class has a dive ability in which it breaks from the formation and attacks the player ship. This is implemented in the Dive() function.

The diving behavior itself is implemented in one of the handler functions in the Level class – EnemyDivingHandler() called in the Update() function of the Level class All the enemy handler states are called here. First the enemy formation and dive paths are created in the constructor of the PlayScreen class with the CreatePaths() and CreateDivePaths() functions.

The Butterfly and Boss enemy types have a unique dive in which two butterflies are grouped to a boss type enemy and dive with it as an escort. The Boss enemy type has a special ability which is a beam. The handle of this ability is implemented in the CaptureBeamHandler() function of the Boss class.

The logic of the capture beam itself is implemented in the CaptureBeam class. The RunAimation() function of the CaptureBeam class handles the play of the capture beam animation, which loads it in chunks: the first 2 seconds the beam loads in and then it disappears in a sequence. After the beam animation is played the boss enemy type disappears from the screen and spawns on top of the screen and goes back in the enemy formation.

For the spawning of the enemies I am using an xml file and using the external XML parser TinyXML2.

TinyXML2 is a simple, small, efficient, C++ XML parser that can be easily integrated into other programs. The spawning is implemented in a priority from 0 to 4, so the enemies spawn in a sequence and not all at once. The spawn priority and path are defined in the Level1.xml file and parsed and loaded in the EnemySpawningHandler() function of the Level class.

**Collision Detection**

There is a Collider master class in which the different collider types are stored in an enum class ColliderType (Box and Circle).

There is also a PhysEntity class which inherits the base GameEntity and is for entities which have physics in the level.

The Box and Circle colliders are defined in their classes.

Most of the physics entities like the enemies, player and bullets use circle collider because it is more resource light and only when to circle collider overlap, box collider is used.

The PhysicsManager class handles the whole collision in the game and every physics game entity is divided into layers in the enum class CollisionLayers, which determines which entites will collide with each other.

The Collision Masks for all the physics based entities like the enemies, player and projectiles is set in the GameManager constructor.

In the update function of the PhysicsManager class all the logic of the collision check is done and in the WasHit() and Hit() function of the Player class, the consequences of the player being hit are imolemented like the reduction of lives, explosion effect and animation.

Each enemy type gives a different score to the player when destroyed and when is hit the enemy death animation is played. This is defined in the Hit() function of the Enemy classes.

When all the enemies are killed a new stage is loaded.

6. UI

The game has a main menu:

It is the first thing the player sees when he starts the game. It features several elements. The play mode and a highscore which saves his highstore during play. Also there is a logo animation and a cursor with which the player selects the game mode. The main menu logic is implemented in the StartScreen files.

A background star flickering effect is implemented with 4 different colors of star which flicker at random positions of the screen. The logic behind this is implemented in the Star, BackgroundStar and StarLayer classes. The stars are stored in an array of the singleton BackgroundStars class.

The score functionality is handled by the Scoreboard class. A separate scoreboard class is implemented because of productivity issues if only textures are used for the update of the score, because in SDL we would have to load a texture every time the score is updated.

The PlaySideBar class handles the information about the player stats and level identification. It displays the current player score, the highscore and the player lives. It is defined, updated and rendered in the PlayScreen cpp file.

The player lives are displayed in the form of the ship sprites which amount to 5 on the side bar. If they exceed this amount, a number appears showing the total player lives.

The level indication is implemented by showing flag images, each corresponding to a level which range from 1 to 50 (1, 5, 10, 20, 30, 50). There is a setter function for the current level and the flag image is set corresponding to the current level value set in the PlayScreen file.

**Developer**

**Мартин Калчев, F85994**

Link to project repository: <https://github.com/MartinKalchev/Astro-Game-Project.git>