**Engine Documentation**

This document provides an overview of the engine's structure and functionality.

Tasks: <https://quire.io/w/Gasper_Kadivec/5?myTask=me&view=board>

**Structure**

* **main.lua**: The primary file called at launch.
* **GLOBALS.lua**: Contains all global variables. Add any new global variables here if they are not already present.
* **conf.lua**: Currently contains only the orientation code.
* **functions/**: This folder contains all necessary functions for the game, including game logic and basic functions. It also handles input logic.
* **idea/**: A boilerplate folder, leave as is.
* **initialization-engine/**: Contains the necessary libraries and the require function, which is called at the end of main.lua. It also includes all settings variables.
* **main/**: Hosts the actual game. It includes the initialization of the game (screen, level for now), and functions for game initialization. This folder also contains the load, update, and draw files, which are the core of the engine.
* **objects/**: Contains all objects in the game/engine. Each folder either includes subfolders or directly contains the class file. The structure starts with a subfolder (if any), and the subfolder includes a class file. Each class has a basic class that can be inherited later. The folders may also include an images folder, which stores imagesData for the specific class (loaded into the variable ImagesData["BasicExample"]) and Images variable (actuall image). Every basic class must start with ! for proper loading.

**Important Variables**

* **CollisionGroups**: This variable stores all indexes of collections, that are used for default collision beetwen objects. Use self.collisionGroups in an object to se custom collision.
* **ImagesData**: This variable stores all image data in the project, excluding the actual images.
* **Images**: This variable stores all images used in the project.
* **Screen**: This variable manages the interactions of objects with the screen borders.
* **Canvas**: This variable is used for drawing purposes and has the same dimensions as the screen.
* **mc**: This variable represents the main character that the player controls. It can be disabled for different game types.
* **Touches**: This variable contains all touch inputs on the screen.

**Logic**

**Hit Registration of Objects (e.g., Main Character vs Enemies)**

1. **Collision Detection**:
   * First, check if the projectile has collided with the enemy using playerProjectile:hasCollided(enemy).
   * If a collision is detected, call playerProjectile:hit(enemy). This function contains the logic for the player's projectile.
   * If hit() returns true, the projectile has been used and is subsequently removed from the table (i.e., removed from the game).
2. **Enemy Hit Calculation**:
   * After handling the projectile, the code enemy:hit(proj) is executed. This function processes the logic for the enemy being hit and returns the result of this function.
   * Each enemy has a specific function, BasicEnemy:hit(proj), which takes the projectile as an argument and calculates the damage to the enemy's health (self.hp).
   * This function returns true if the enemy accepts the hit (e.g., the enemy is not immune).

## Yet to Implement

### Layer Management

* **Layer Array**:
  + Create an array called layers to manage the states of the game. This array will contain instances of the Layer class.
  + Each Layer instance will encapsulate its own Collections.
* **Layer Execution**:
  + Only the latest layer in the array is active and running.
  + If the user presses the home button, all layers are cleared except for the home layer. Alternatively, a new home layer can be created, and all existing layers can be deleted.
  + Inactive layers remain in the array but do not run.
* **Resuming Layers**:
  + To resume a previous layer, clear the latest layer from the array.
* **Collections Management**:
  + Decide whether the engine will utilize the Collections within the layer itself or overwrite the global Collections array when loading a new layer.
  + It may be more straightforward and less prone to bugs to use the Collections specific to each layer. Each layer will manage its own Collections and main character (mc).
* **Layer Functionality**:
  + Consider implementing separate update and potentially draw functions for each layer. This would allow each layer to handle its own logic and rendering independently, similar to activities in Android Studio.

**Movement Logic**

1. **angle**:
   * Represents the angle in degrees at which the object is moving.
2. **velocityX and velocityY**:
   * The speed at which the object moves per frame, measured in pixels. This values are calculated based on the forces applied to the object.
3. **applyForce(angle, force)**:
   * This function takes an angle (in degrees) and a force to be applied in that direction.
   * It calculates the new velocity components (velocity\_x and velocity\_y) considering the current angle and velocity.
   * Updates the overall velocity and movement angle.
4. **reduceVelocity()**:
   * This function is called every frame to simulate friction, gradually reducing the object's velocity.
   * It multiplies the velocity components by a friction coefficient to decrease the velocity over time.
5. **move()**:
   * This function calculates the new position of the object based on the current velocity and angle.
   * It updates the object's position and applies friction by calling reduceVelocity().
6. **fun()**:
   * This function is called every frame to update the object's state.
   * It calls the move() function to update the position and velocity.