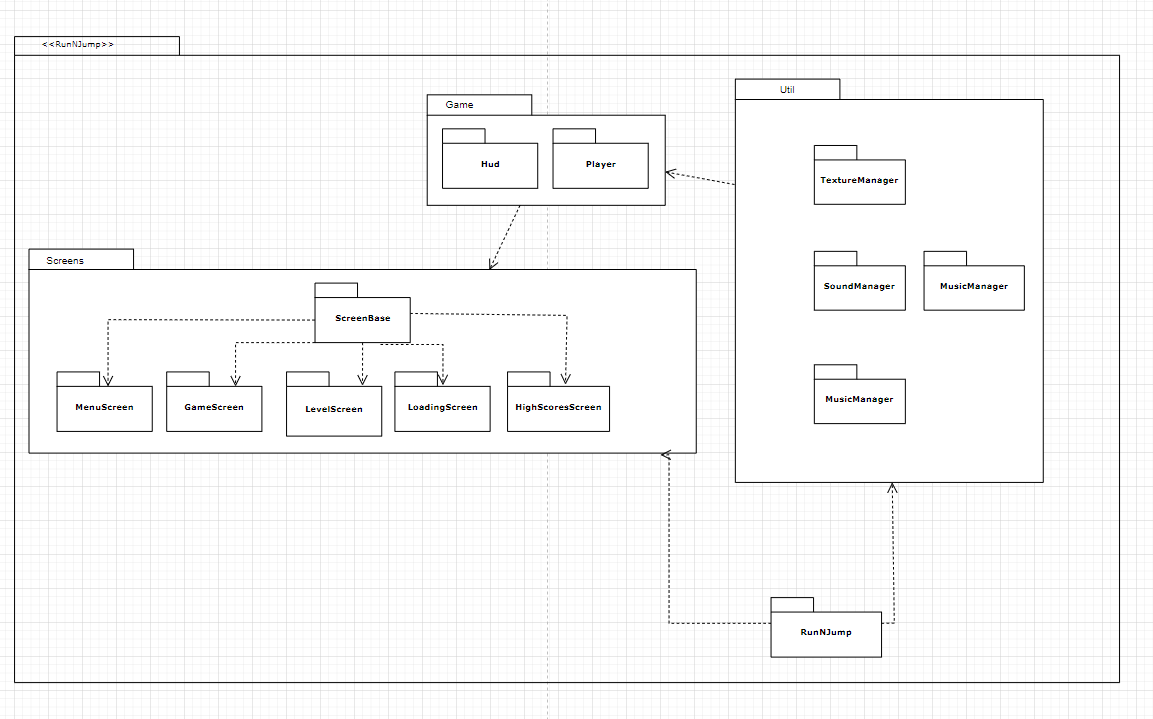
Technical documentation

Package Diagram



Main project directories

Android

* This is where the platform specific code for android is located.
* In the ‘assets’ folder within is where all the assets, sound and graphics are kept - since android assets need to be placed in an android project’s resources folder otherwise they won’t work.

Core

* This is where all the platform-independent code is located, all the code aside from specific android and desktop launchers are within sub-directories of this folder.

Desktop

* Only the desktop Java application entry point/launcher is in this directory.

Class Runnjump

This is the actual ‘main’ class of the aplication. It stores all the managers. So they can easily be accessed from the entire app.

Methods

setLevelSelected(int level): this sets the level selected by the levelscreen. So it can be accessed from the game screen.

Create(): creates the app-wide elements such as the managers and the sound button.

changeScreen(ScreenEn screenId): takes an enum of the screen the app should switch to. If that screen doesn’t exist it will create it.

Package Game

In this package the classes which represent the actual game elements which appear in the GameScreen are.

Class Hud

This class represents the in-game UI the player sees while actually playing the level. The constructor creates the layout and determines weather to render the android specific HUD or desktop, depending on which device is running the app.

Object public fields: stage - it is the highest element in the hierarchy of actors and on which all the UI elements are placed.

Methods:

setScore(int score): this method is used to update the score displayed on the ui.

setLives(int lives) does the same thing but for lives reamining.

getMovementJoystick(): a getter for the touchpad

getJumpBt(): getter for the jump button.

gameOver(int score): this method prints the game over messages onto the hud, informing the player he can play again. It gets score so it can be printed.

gameWon(int score, int level): this method similarly prints the game won messages onto the screen/hud informing the player that he has successfully completed the level.

Class Player

This class represents the player character on the game map. The constructor initialises the player, and his initial values for score, lives remaining and gets its textures from the textureManager. It also sets up the platform-specific touch detection listeners for Android touchpad and jump buttons.

Methods:

setLogicalSize(float width, float height): largely redundant method for changing the logical size of the player, ignoring his sprite’s size, for collision detection. Only useful for debugging purposes.

setFrame(Texture texture): this sets the frame(for the purposes of animation) of the sprite which will be displayed upon drawing.

Draw(Batch batch): this draws the sprite after updating.

collidesSouth(), collidesEast(), collidesNorth(),collidesSouth(): methods used for testing collisions on all sides of the player character, returns true if a collision is found, false if there is no collision. Also detects collectibles and things that kill the player.

handleMisc(float x, float y): used for handling a detected collision with miscellaneous items, currently only the victory point, ie the win flag is handled here. Float x and y are the positions on the tile map where the tile the player collided with is located.

handleCollectible(float x, float y): same as above but for collectibles such as power-ups, coins and stars.

removeCollectible(int tileX, tileY): removes the collectible.

update(float delta): updates the position of the player, checks all the collisions and determines the frame of animation to be drawn. Additionally checks if a power up was used and grants the player the power up’s effects. Delta is the time since last rendering occured.

isCellMisc(float x, y), isCellCollectible(float x, float y), cellKillsPlayer(float x, float y), isCellBlocked(float x, float y): All these methods are similar, they check if the cell specified by x and y are miscellaneous, collectibles, blocked or kill the player return true if yes false if no.

keyDown(int keycode): This method allows for keyboard input for controlling the player character on windows desktop.

keyUp(int keycode): this method stops the movement of the player character when A or D keys stop being held down by the player, on windows.

Die(): this method removes a life from the player and sets him as dead.

isRunning(): checks if the player is currently moving on the x axis

inAir(): checks if the player is currently in the air

isIdle(): checks if the player is currently idle

respawn(): this method is called when the player is respawning, if the player has no lives left false is returned.

setLayers(TiledMapTileLayer visualLayer, TiledMapTileLayer collisionLayer): this method is used for setting the layers of the tile map which need to be checked for collisions for the player.

Restart(): this method is used for restarting the player character to its default state, is called when the level is restarted.

isGameWon(): checks if the level has been won.

getTimeWon(): gets how much time in seconds passed since the game has been won.

Class Enemy -

A class used to represent an enemy NPC and its assorted AI and the logic related to it. Unfinished as of week 11.

Package screens

This package is used for all the screens which appear in the game, a screen is like a window which is displayed, this can be a menu window or the game window.

abstract class ScreenBase

This class is the parent of all the screens in the package, it does all the necessary processing required by all the screens. The constructor initialises the skin used by the whole application, the background for the menu system and the batch used for drawing.

Methods

Show(): this method is called every time the screen changes, it sets up a new stage for every screen and a camera. Additionally it sets up the input processing multiplexer which processes all the inputs from the different stages.

Resize(int width, int height): this is called upon resizing of the game window.

Render(float delta): This is the method which actually draws everything onto the display. delta is the time in seconds since the last render call.

Dispose(): this method disposes of all the resources, to prevent memory leaks.

keyDown(int keycode): this method implements the ‘back’ key functionality on android and windows. (by pressing the ESCAPE key). Switches the screen to the previous screen.

Class GameScreen

This is used to represent the screen with the actual game and its level. The constructor initialises the chosen level.

Methods

respawnPlayer(): this method respawns the player at the (currently hard-coded) spawn point, but only if the player has lives left. If not, a game over boolean is set.

startGame(int level): this method is used for re-starting the level after a game over.

Show(): this method is used for starting the chosen level after the level/game mode is chosen. It sets up the whole tiled map and input processors.

Render(float delta): renders the game world and the hud also makes the camera follow the player. Checks if the game has been won or a game over happened.

touchDown():this method is used for restarting the level or leaving the level upon winning by the player and going back to the main menu, opon touching of the screen. This is android-specific.

keyUp(): the same method as above but for windows.

Dispose(): prevent memory leaks by disposing all game elements with are disposable.

Class HighScoresScreen

This screen shows the high scores screen, the interface is finished but the functionality isn’t.

Class LevelScreen

This class is the level selection screen, it allows the player to select the level on which to play on.

Methods

updateLevelSelectIndicator(): this updates the level selected label to show which level has been selected.

Class LoadingScreen

This class represents the loading screen which is shown when the game begins, currently it has no look of its own but this class does all the loading of assets necessary for the rest of the game.

Methods:

loadPlayer(): This loads all the player assets and all the frames of animation.

loadGraphics(): this loads all the graphical elements of the game aside from the skin, the tilemap and its map elements themselves.

loadSoundFX(): this loads all the sound effects of the game.

loadMusic(): this loads all the songs in the game.

Show(): loads everything.

Render(float delta): once loading is completed the screen is changed to the main menu.

Class MenuScreen

This class represents the main menu window.

Methods

initGui(): this method initialises the GUI of the menu and sets up iuts input processors.

Show(): this method inits the gui and plays the background music.

Render(): draws the gui.

Package Util

This package contains the utility classes used in the game.

Class MusicManager

This class loads and manages all the music tracks in the game.

Methods

addMusic(String name, String path): this method loads and stores a song in the manager, under the name specified. It grabs the file using the path.

addMusicSet(String name, String[] paths): Similar as above but instead it grabs multiple songs and puts them under a single name, useful if you want a specific soundtrack for a specific level for example.

playMusic(String name): this plays a previously loaded song specified by the name, assuming no song is being played at the moment.

playRandom(String name): this plays a random song from the song set added by music set under the name specified.

muteMusic(): this method mutes music.

stopMusic(): this stops the currently playing song.

stopAll(): this method is used for stopping all currently playing songs, it is redundant since the current implementation only allows for one song to be playing at the same time, so currently in practice is identical to stopMusic().

Dispose(): disposes of all the music assets, to prevent memory leaks.

Class SoundManager

Similar to the MusicManager but for simpler/shorter sound effect sounds like the sound effect upon picking up of a coin.

Methods

muteSound(): mutes and unmutes all sound effects.

addSoundSet(String name, String[] paths): adds and loads multiple sounds under the same name.

addSound(string name, String path): adds and loads a single sound under the name specified.

playSound(String name): plays the sound specified by the name

playRandom(String name): plays a random sound from a sound set added under the specified name.

Dispose(): disposes of all the sound assets and prevents memory leaks.

Class TextureManager

This class is similar to the other managers, but is used for pre-loading textures and animation frames.

Methods

addPlayerFrameSet(String name, String path, int number): this adds a set of player images which show an animation if played after one another, the name of the file must end with a 3 digit number and .png. Number is the number of images in the set.

getPlayerFrameSet(String name): this returns a player frame set specified by the name, returns in the form of an arraylist.

Dispose(): disposes of all the texture assets and prevents memory leaks.

Class SoundHandler

This class is used for handling the input for the sound button. It mutes and unmutes sounds as the sound button is checked or unchecked.

**General Implementation**

The game at the high level consists of a series of screens, with the actual gameplay happening on the ‘GameScreen’. The game renders the chosen level based on its TMX map (a text based data file which represents tile-based game levels). The gamescreen parses that tiled map by each layer placing the appropriate tiles where they should be and rendering the layers in their chosen order, furthermore it parses the objects placed on the tiled map. Based on those objects on the TMX map it creates game objects from my classes such as Enemy & Player and places them in the appropriate positions.

**Dialogue System**

Dialogue is stored in text files. Each line in the text file represents a single dialogue screen/display. First part of the line before the @ sign represents the name of the character speaking, or the tag for the line. The following are tags -

* Character name
* CHECK - this represents a check for a certain condition being met, ie a quest/task completion.
  + It is followed by a tag for the CHECK, this can be any label which identifies the type of checking needed to be done in code to verify if the condition was met.
* GIVE - this represents an action the NPC takes to give you a reward/gift.
  + It is followed by a description of what is given
    - Either a power-up, score or a life.
* CONDITION - this can be used to give a condition.
  + It is followed by a name/tag of the condition to be granted. For example when the player rescues an NPC the condition CONDITION@RESCUED1 would be written on the line.
* END@END - indicates the end of the dialogue, and the beginning of a new context. So next time the player tries to talk to this NPC the line directly following the END@END line will be the one shown.
* KILL@KILL - this indicates that the NPC will dissappear at the end of the dialogue.