Programmer Documentation for Jetpack Runner

Specification

Jetpack Runner is a playable endless runner game based on the mobile game Jetpack Joyride. The goal of the game is to travel the longest distance possible while avoiding zappers. The game was written in c# using the .NET WPF. WPF was chosen because it is more modern than WinForms and also uses hardware acceleration, which can help the game run more smoothly (ie. with less stutter, etc.).

For more information about the original specification, read the `PRG - Semestral Project - 01 - Specification - Švarc` in the Jetpack Runner folder.

List of functional requirements:

1. A player must run at 30 FPS
2. The game must be stable
3. The game must end after the player hits a zapper
4. The zappers must generate in a way that it's not impossible to move between them
5. The game must filter out all inputs other than the spacebar/mouse button

Architecture/Design

Game states

The game has 2 different states which are dependent on the state of the game character. If the character is alive, the game is running (ie. generating obstacles and moving background). If the character is dead, the game ends and is waiting for an imput from a user. When the game is in the running state, it refreshes the screen every 20 milliseconds, which results in a framerate of 50. Even if some frames are lost, the game should run at at least 30 FPS.

Input

The game itself recieves input from either the keyboard or the mouse and filters out all other input. This is achieved by using the funtions built-in to WPF.

Map generation

The zappers itself generate pseudo-randomly. There are 3 different positions on the y-axis where the zappers can generate. When a zapper goes off screen it is then generated at a position with coordinates of 950 + a random number (from 0 to 40) on the x-axis. And one of the 3 positions on the y-axis. The background generates similarly to this, but when it goes off screen it is generated at coordinates 1260 on the x-axis.

Data structures and global variables

No special data structures are used and global variables include a DispatcherTimer which keeps refreshing the game loop. Another global variables are ImageBrushes, Rectangles, an instance of a class Game, and 2 double values used for animation of the character and zappers.

Main algorithms used:

- the main game loop - invoked every 20 milliseconds
- generation of a background and zappers based on their location on the x-axis - invoked everytime gameloop is ran
- collision detection between the character and zappers, ground or the roof - invoked everytime gameloop is ran

Technical documentation

Functions

MainWindow():

- creates a new window and initializes all window components
- sets the refresh rate of game loop function
- sets background image

Procedures

InitWindow():

- sets the character and zapper image
- starts the game timer

GameEnge():

- the main game loop
- increases the speed of the background by a small amount every invocation
- refreshes the position of all components: character, zapper, background
- creates new hitboxes for all components : character, floor, zapper, roof
- checks if collision occures
- if the character is dead it displays score

DrawNew():

- checks if a background or a zapper is off the screen and then places it on appropriate appropriate coordinates
- this creates the illusion of the infinitely long corridor

- adds +1 to the score when a zapper gets of the screen

Animations: RunAnimation() and ZapperAnimation():

- gets a double value as a parameter
- switch statement then uses the parameter to switch between images, which creates the animation

CheckCollision():

- different types of collisions:
    - Character with the roof: game and gravity is set to 0 -> player is moving horizontaly
    - Character with a zapper: the character dies and the game ends
    - Character with the ground: the character running animation is displayed

SetStage():

- sets position for character, background and zapper

Input functions:

- Canvas_KeyDown & Canvas_MouseDown:
    - KeyDown: if space is pressed
    - MouseDown: if mouse button is pressed
    - both set the uplift variable and grabity variable in class Game
    - this results in a player going up
- Canvas_KeyUp & Canvas_MouseUp:
    - KeyDown: if space is not pressed
    - MouseDown: if mouse button is not pressed
    - this results in a player going down