Aansturing programma Embedded fitness

Project:	**De vloer is lava**
Client:	**Embedded fitness**
Project Number:	**1.1**
Auteur:	**Sam Hendriks**
Date:	**25-5-2020**
Version:	**1.1**

The undersigned declare their agreement with the content of this Project Plan document

Client	**Project Manager**
Initial Seen:	*Initial Seen:*
Date: _____	Date:
Place: _____	Place:

Contents

Development

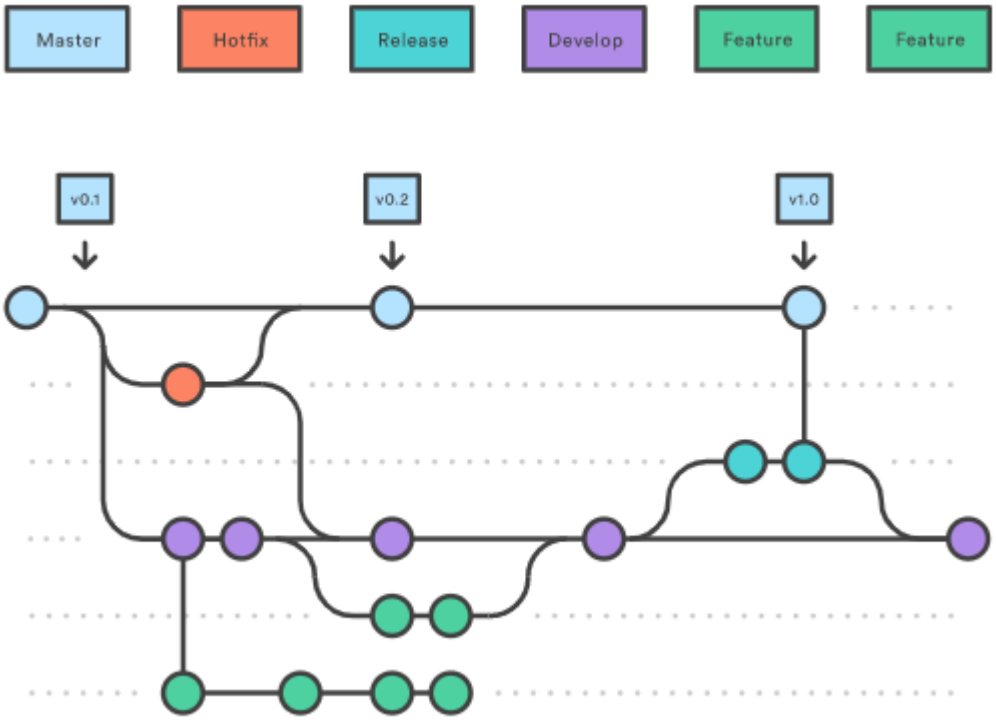
Tools required

C# & XAML capable IDE (Visual studio / VSCode recommended)

git (GitHub desktop client / gitKraken recommended)

The target framework of the application is .NET Framework 4.8.

Git Flow



Branches

Lightspace development follows the git-flow workflow.
master Master branch. Pushes can be made using pull requests.
dev Develop branch. Pushes can be made using pull requests.
A full guide describing the git-flow workflow can be found here.

Branch Naming

Branches must follow the following naming strategy:

feature: feature/"project-name"-"short-implementation-description"

example: feature/SR0-added-hardware-support

release: release/"release_version"

example: release/0.0.1

hotfix: hotfix/"project-name"-"short_bug_name"

example: hotfix/SR0-fix-nullreference

Commit Messages

Commit messages should explain the largest impact change made in the new commit.

An example for how to do this is as follows. Imagine this hotfix branch fixed an issue where a nullreference occurred due to an error when loading a project resource.

hotfix/SR0-fix-nullreference

example: Fixed a nullreference occuring when loading resource.

If the commit/branch is relevant or an update to an Issue, include #[number] in the commit message to link the commit to the relevant issue.

example: Fixed a nullreference occuring when loading resource. #0

Pull requests

Pull request are required to be reviewed by at least one other collaborator. After the check has passed it will be possible to merge the branch. A pull request should always be Squashed when merged.

How do Pull requests work and what are they good for?

Pull requests are a step in between committing(& pushing) and merging. You push your branch with it's changes to the repository, and you create a pull request to the Develop branch (dev). The pull request will go into review, and you can invite other collaborators to review a pull request.

Pull requests should be reviewed by others as a quality gateway to check your code for possible errors or mistakes before it will be pushed onto the main development branch. This is to prevent (most of the) errors from ever entering the main development cycle and possibly interrupting the entire project workflow.

Guidelines

When creating a project, there are certain guidelines to follow. These are important for maximizing performance, and minimizing issues.

- Only update your render visual if it has changed since your last tick.

This means that if your visual render has no changes, it does not need to be updated and pushed to the UI. You need to keep track of this yourself, because comparing the visuals every frame can get expensive very fast. Render visual found as gameRender in TileManager.cs and can be called using SetRenderGraphic() and GetRenderGraphic().

- Be careful when working with different threads.

The Core Loop, Main (UI) and Performance Profiling threads are all separate. This means you can't simply access all their functionality from your game class. Any needed functionality should be available to you, and otherwise you can make a new Issue with the desired feature.

- When removing or changing main program functionality, discuss with other collaborators.

If you are making changes to main program functionality, you may have solved all issues on your end, however other people might need or be working with said functionality. Discuss with other collaborators to work it out, and if needed work together in a Pair Programming manner.

Creating a game

To create a new game, Create a new folder in the Games folder of the LightSpace_WPF_Engine Solution.

In this folder you can create a class. This class should inherit from RunningGameBehavior. Within this class certain standard functions can be used for creating your game:

- Start (public override void Start())

Will be triggered on creating the object.

- Update (public override void Update())

Will be triggered every frame of the game.

- LateUpdate (public override void LateUpdate())

Will be triggered every frame of the game after the entire update step has finished.

There are 2 more functions that can be used, though these are risky since they are able to break the flow of the Core Loop if used incorrectly.

- Destroy (Destroy())

Can be triggered to remove the object from the BehaviorManager, thus breaking possibly the last reference. (**dangerous**)

- OnDestroy (public override void OnDestroy())

Will be triggered before the object gets destroyed. (**dangerous**)

Information on setting game data, specifically your Behavior can be found here.

Adding your game

To add your game to the collection of existing games, we use the GameList class (Models/Models/GameList.cs) and the GameName enum (Models/Enums.GameName.cs).

First off, add your game to the GameName enum (if multiple new games are created at once, discuss with other developers in order to avoid creating duplicate values). After you've added your game to GameName, start filling in the Get functions within GameList.

Setting your game data

GetGameControls()

GetGameControls is the function that loads in your custom controls. (information on creating this found here. Simple add the case for your GameName entry, and return a new object of the type of control you made. If you have no custom controls, you can return null.

GetGameDescription()

GetGameDescription loads a string that is used within the UI to give a short description about what your game does, how it functions, or what it's used for.

GetGameIcon()

GetGameIcon gets a bit more complicated. To use your image, you need to add it to the Solution Resources. First off, drag the image you wish to use into Media/GameIcons. The current sizing standard is 64*64 Pixel .png file.
Once you've added your image, Right-Click the solution and go to the Resources section. Drag your newly imported image into the Resources. this should create a new entry with the name of your file.

Back in GameList you should now be able to return Properties.Resources.<ResourceName> directly.

GetGameBehavior()

GetGameBehavior is the function that loads your game. If you've made your game according to the previous wiki entry, you should have a class that inherits from RunningGameBehavior. Simply return a new object of that class type in this method, and your game should be loaded.

Creating custom ui

UI & Template

Custom UI can be made for your game. The main UI already has a spot reserved for your custom UI, and the previously mentioned GameList will handle the references for you here. Create a new UserControl within (Wpf/Views/UserControls/CustomControls).

the CustomGameControlTemplate.xaml and CustomGameControlTemplate.xaml.cs files are template files you can use as a guide to create your own custom game UI.

Icons & Button Icons

Icons for UI can be found on Modern Ui Icons, or new custom path data icons can be created using this website or any other svg path editor. (respective licenses can be found here)

Logging

Within the application, logs and errors can be handled through the ConsoleLogger. There are two main functions used here.

- ConsoleLogger.WriteToConsole(object data, string message, Exception exception _(nullable)_)
- ConsoleLogger.WriteToUiConsole(object data, string message,bool unique, Exception exception _(nullable)_

WriteToConsole

Allows you to write directly to the console window that opens up at the start of the launcher. Any exceptions passed into this function will automatically be written into a log file saved at %AppData%/Local/LightSpace_WPF_Engine.

A single file will be saved **per** launch of the application if errors are logged, and the file is named after the millisecond the error occurs. Anything passed to the console via this way could be logged if the SaveExceptionsOnly bool is set to **false**, however do note that many writing actions will take it's toll on the application.

WriteToUiConsole

Allows you to write directly to the User Interface console situated below the rendering views. This message is also directly sent to the WriteToConsole in the background, so that errors (or any message in case of SaveExceptionsOnly being **false**) appears in the general log and can be logged to a file.
Using this action is **slower** than WriteToConsole and should only be used to inform the user about critical information regarding the application.