SU 17
Deliverable 4

Group 0xa

https://git.dikunix.dk/su17_0xa/DIKUArcade/tree/exam-2017

Adam Ingwersen,
Peter Friborg,
Thomas Schaer

Datalogisk Institut
Københavns Universitet

June 6, 2017

1 Resumé

This is our submission of the **SpaceTaxi Game - Report** for the exam in **Software Development B3-4, 2017**. This report is accompanied by a C#-solution developed using **Mono**. The latest submission of the software is the cummulative work of this group on the **DIKUArcade** engine - which consists of **d2** and **d3**. This final submission, **su17-exam**, is primarily concerned with refactoring, testing and implementing some minor extensions to the previous work. The end product is a simple version of the *original* **Space Taxi** arcade game.

2 Background

This group's `SpaceTaxiGame` project attempts to implement the directions for the game outlined in Deliverables 2 through 4. As such, this implementation of `SpaceTaxiGame` attempts to mimick some aspects of the classic Commodore 64 arcade game, Space Taxi, whilst disregarding or modifying others. This version of `SpaceTaxiGame` is a simple arcade game, in which a single player tries to earn money by picking up customers standing on platforms. These customers need to go to a space station for further transit - the player can drop them off by warping to the nearest space station. Collisions with surrounding objects are to be avoided.

Most notably, *our* game deviates from the classic game by omitting a game-objective[1], which otherwise states, that the customers should be picked up, and then dropped off on a different platform. This objective was not explicitly stated in the directions of aforementioned deliverables - and has not been implemented.

The background for *our* project is to provide a full implementation of the game described above to elaborate on in this report. The implementation relies on the B4-DIKUArcade Game Engine. The software, which has been developed on top of the engine is designed in accordance with the SU17 course guidelines - and attempts to apply the techniques taught in the SU17 course.

The target audience of this project is the course examinators of the Software Development course of 2017. The purpose of this project is to enable a single player to play the SpaceTaxiGame using a keyboard interface on a machine with Mono (version $\geq$ 4.0.0) installed. This project is intended to showcase the **B4-DIKUArcade** framework - however this report is not concerned with describing the framework itself.

3 Analysis

The game framework introduces a graphical user interface using timers, threads, events and graphical widgets, which are ideal when developing small **Arcade Games**, such as `SpaceTaxi`.

[1] `https://en.wikipedia.org/wiki/Space_Taxi`, Gameplay Section

The game should be a single player game, enabling the Player[2] to travel in space using the keyboard. The player should move in floating motions, which should simulate velocity in space. The player is progressing through levels varying in difficulty.

The `Taxi` should be controlled by the following keys:

- $\boxed{\uparrow}$ Should add an upward going velocity to the player, while visualizing thrusting below the `Taxi`.
- $\boxed{\downarrow}$ Should add a downward going velocity to the player - there are no thrusings on top of the `Taxi`.
- $\boxed{\leftarrow}$ Should add a velocity pulling the player right, while visualizing thrustings on the left-hand side of the `Taxi`.
- $\boxed{\rightarrow}$ Should add a velocity pulling the player left, while visualizing thrustings on the right-hand side of the `Taxi`.

The player earn points, which should allow for a passage to the next level. The points should be collected when the `Taxi` picks up a `Customer`, while standing still next to him.

The game should start up on a random level, which is read from an ASCII-art file. The game should interpret the ASCII-art based on the different objects represented herein.

The player can move the `Taxi` around the game arena, using the controls described above. The Taxi should be sensitive to collisions with sorrounding objects - the Taxi should explode, and the Game end upon collsion.

Other design goals relating to technology stack choices:

1 The game should be written in C#, as a MonoDevelop / Xamarin Studio project, version controlled with Git — the technology stack of our course.

2 The game should run across the variety of operating systems which one might reasonably expect on our course.

Other design goals relating to coding style:

3 Each object should be designed to be in charge of moving and drawing itself, if it's movable and drawable.

4 The classes should each have one responsibility, with methods acting according to this responsibility.

5 The game should be designed according to the principle of least privilege.

6 The project should be designed to have testable methods, which make them suitable for unit-testing.

[2]The Player is visualized through a taxi with thrusts on the back and bottom

4 Design

The overall game is structured as follows:

`SpaceTaxiGame`
: A singleton class, on which the game is based. This singleton class inherits from `Game`, which has a method `OnDrawScene`, which acts according to the state of the game. The game keeps track of the state with the behavioral design pattern (State Pattern), contained by `SpaceTaxiGame`.
The Game contains the Player, the Score, the different levels in `LevelHandler` and all objects in an `EntityContainer`.

`SpaceTaxi Entities`
: All `SpaceTaxiEntities` are contained in a `EntityContainer` class, which is using the iterator design pattern. With this iterator design, it is possible to add and remove entities from the game while iterating through the list of `Entities` contained in the `EntityContainer` class.
Each `SpaceTaxiEntity` contains through inheritance a method `RenderEntity`, which takes care of 'design goal 3'.

`Taxi`
: This class not only inherits from `Player` but also the interface `IVulnerable`. The `Taxi` class implements a method, `ChangeImage()`, which changes the visualization of the `Taxi` according to the last keyboard input.

`Customer`
: This class inherits from `Entity` and the interface `IVulnerable`. The Customer implements two methods `MoveCustomerLeft` and `MoveCustomerRight`. A `Customer` is initialized with boundaries describing when the Customer should turn and move in the other direction. Each Customer contains 3 states: `Unattended`, `WalkingToTaxi` and `OnRide`.

`Platform`
: This Class inherits from `Entity`. A `Platform` is initialised with the name of the image the platform should have. Each platform takes care of a its own `Customer`, and notifies the `Customer` when the `Taxi` lands on the platform. This is done according to the observer design pattern.

`Warpdrive`
: This Class inherits from `Entity` and the interface `IVulnerable`. `Warpdrive` doesn't have any image, only a Shape. The `Warpdrive` takes care of adding local score to global score, while changing to a new random level through the implementation of `Hit()` if the local score is greater than 1 - else the level restarts.

`Obstacle`
: This Class inherits from `Entity`. An `Obstacle` is initialized with the name of the image the platform should have.

`GameParser`

The `GameParser` class is responsible for 'design goal 2' regarding platform compatibility. This is done by using the `Directory` and `Path` class integrated with C#. With these classes it is possible to get parent folders and combine paths on different platforms.
The `GameParser` class maintains the ASCII-art in a `List<string>`, the `Key/Value`-pair mappings in a `Dictionary<string, string>`, the platforms in `List<string>` and the level name in a `string`.

`GameManager`

GameManager contains the functions for loading and saving. `Save` reads the necessary properties from `Game` to recreate the game, and decomposes them into smaller bits to fit them into the simple structure of `SaveSerializables`. `Load` works in the inverse fashion, first reloading the level and clearing the customers away, before writing the fields of `SaveSerializables` into the properties of `Game`.

`SaveSerializable`

Simple `[Serializable]` class from which the `GameManager` saves and loads. To keep `SaveSerializables` light and it's contents to a minimum, as well as to avoid serializing most of our solution, all data have been decomposed to its raw states, i.e. with no reference to types declared in other classes.

`Controls`

SpaceTaxiGame maintains a `TaxiMovement` class. `TaxiMovement` class inherits from `PhysicalMovement`, and maintains two new methods `Keypressed(KeyPressEventArgs)` and `KeyRelease(KeyReleaseEventArgs)`, which adds force-sources according to the description in section 3.

5 Implementation

This section will address some of the topics outlined in the **Design** section 4. However, only the components which have been significantly altered since the hand-in of Deliverable 3 will be discussed.

5.1 `SpaceTaxi`▸`SpaceTaxiEntities`

All Entities are contained as described in **Design**, within an `EntityContainer`. The `EntityContainer` inherits the `IEnumerable` interface, which implements a simple iteration through a collection which makes it possible to iterate through every `Entity` contained.

`Taxi`

The method `ChangeImage()` works as described:
The `Taxi` contains the current `Image`, this design makes it possible to change the current `Image` according to input from `Player`. All images used to visualize the different thrustings are contained by 8 `readonly Images`.

`Customer`

Customer moves according to the state, and moves using the methods described in **Design**.
When the state is `Unattended`, the `Customer` moves from one edge of the platform to the other [3].
When the state is `WalkingToTaxi`, the `Customer` walks to the taxi while locating whether the `Taxi` is on the left- or righthand side. If the `Taxi` takes off from the platform on which the `Customer` walks, the state is changed to `Unattended` again.
When state is `OnRide`, the `Customer` is removed.

`Platform`

The notification to `Customer` (described in **Design**) is handled by a dedicated method `NotifyCustomer()`, which notifies the `Customer` when `Taxi` collides with the `Platform` and changes the state of `Customer` to `WalkingToTaxi`[4].

5.2 `SpaceTaxi` ▸ `SpaceTaxiEntities` ▸ `EntityFactory.cs`

The *public* `EntityFactory`-class can be instantiated using an `EntityContainer` and either a level integer or a `GameParser`-object. The class declares the *private* variables needed for initializing **Entitity**-objects into an **EntityContainer** and then the **SpaceTaxi Game**. The `EntityFactory` class deploys a *private* method for each type of **Entity**; **Obstacle**, **Taxi**, etc. These methods are called from a *public* `AddEntitiesFromAscii()`-method. This method recieves input from a `GameParser`-object, retrieving a dictionary for mappings, ASCII-art, and the list of platform mappings.
Using the fixed dimensions of the ASCII-arena, the method traverses a 40-by-23 grid using the list of strings provided by `GameParser.GetAsciiMap`. Using the mappings, when a match is found, a corresponding object is instantiated using the aforementioned methods. This is true for all objects except **Customer** and **Platform**. In constructing solid collisions, taxi landing and logic for customer-movement, it was decided that a `Platform` should not be a box on the grid - but rather an entire object of a shape, determined by adjacent blocks on the grid.
The method `CreateCustomerPlatformFromAscii`[5] implements this. The team's implementation relies on recording the ajdacent `Platform` objects in a list of tuples and mapping the coordinates to a list of keys. This allows for picking out distinct keys and selecting the set of coordinates needed for constructing the `Platform` as a continous object. Given these coordinates, the limits for a `Customer`'s movement can also be identified, and this is used to initalize the customer on the platform - providing the `Platform` with a `Customer`-object. This allows for signaling between `Platforms` and `Customers` , when e.g. a `Taxi` lands on the `Platform`.

[3]See Appendix: 8
[4]See Appendix: 8
[5]See Appendix: 8

5.3 `GameConcepts`▸`GameParser.cs`

GameParser is intended to find, read, decompose and structure data from a textfile, such that this information can be accessed from the `SpaceTaxi`▸`SpaceTaxiEntities`▸`EntityFactory`.

`GetFiles()`

In order to locate a path to the level files, the `GameParser` implements the *public* `GetFiles()`-method[6]. This method will locate a directory; `GameEngine`▸`bin`▸`Debug`▸`Levels`, given the class' base directory. It will then detect any text-files located in this directory and return the full path to these files in an array for use in other methods. This method can be used both with and without a path-string for the current directory in the constructor. This was implemented in order to test the functionality across classes and paths.

`SetGameLevel(int)`

The *public* `SetGameLevel(int)`-method takes an integer in the constructor, and utilizes `GetFiles()` to retrieve a path for a single level file. This method is the *factory* of the class as it constructs the data needed from `SpaceTaxi`▸`SpaceTaxiEntities`▸`EntityFactory` by calling the other methods and enriching the `GameParser` object. This is conducted using a single *System.IO.StreamReader*-object for all operations. `SetGameLevel(int)` employs the following *void* methods:

`ReadASCII(TextReader)`

This *private* method traverses the first 23 lines of the provided *TextReader*-connection, and decomposes each character into a *private* list of strings, which is then accessible from a setter-method.

`ReadProperties(TextReader)`

This *private* method reads the proceeding 4 lines of the provided *TextReader*-connection, using two regular expressions to match everything on a line proceeding *"Name:"* and *"Platforms:"*, respectively. If the line contains the latter match, the string is split into a list of strings for each platform contained in the line.

`ReadMappings(TextReader)`

This *private* method reads the rest of the file line-by-line, using the provided *TextReader*-connection and two regular expressions to split up each line, using **")"** as the delimiter. The left and right side are then mapped to a dictionary as a key-value pair.

5.4 `SpaceTaxi`▸`SaveSerializables`

SaveSerializables stores the data to be saved or loaded. All data have been decomposed to its raw states, i.e. with no reference to types declared in other classes. This was a necessity in order to keep it serializable: If an instance of a class is referenced, `[Serializable]` will attempt to make that class serializable,

[6]See Appendix: 8

which will then require that all classes within also be serializable, eventually requiring that all connected classes be serialized unless specifically marked as `[NonSerializable]`.

5.5 `SpaceTaxi`▸`GameManager`

GameManager is intended to handle all save and load actions with the serializable class `SaveSerializables` by writing and reading from a text file in the default directory `GameEngine`▸`bin`▸`Debug`. As opposed to just writing or reading from an instance of a class, this allows the user to close the game entirely and reopen it, while still being able to return to the saved game. Further functionally can then also easily be implemented, allowing for several games to be saved or loaded, instead of just one.

`Save`
: This method sets the fields from `SaveSerializables` to the current values of their equal field in `game`. As explained in the previous subsection, some data had to be decomposed before it could be saved, in this case implying that we had to create a foreach statement to run through our customers and pick their individual properties. A formatter is created at the end to serialize the class and write the file.

`Load`
: This method sets the fields and properties of `game` to the contents of `SaveSerializables`. A formatter is created to deserialize the file saved from `Save`. The loaded level and it's objects are initialized and the fields of `Game` are written to from `SaveSerializables`. The customers that are created by default for each platform are cleared, and we can now use the decomposed customer data which was stored as a tuple to combine it with the game data, in order to recreate a `Customer` type.

5.6 `SpaceTaxi`▸`LevelHandler.cs`

The *public* `LevelHandler` class utlizies the `SetGameParser()` and `SetGameLevel(int)` methods of the **GameParser** class in order to operate on an instance of a **GameParser** object. It reads all levels and adds them to a list of **GameParser** objects in the `BuildLevels()` method. `LevelHandler` has an additional method `GetRandomParser()` for picking out a random **GameParser** object from the above created list. This class declares a **GameParser** object in the constructor and can be used from the **EntityFactory** class to render an entire level.

5.7 `SpaceTaxi`▸`SpaceTaxiMovement`▸`TaxiMovement`

The `TaxiMovement` class is implemented as described in **Design/Controls**.

A gravitation force pulling the Taxi down is added to the `PhysicalObject`, `PO`, maintained by the `PhysicalMovement` class, which the `TaxiMovement` inherits from. While the gravitation force is always affecting the `Taxi`, new forces can be temporary added by the two methods described in **Design**.

The forces added upon keypress are removed upon key release, this ensures the `taxi` has the floating motion described in **Analysis**.

`TaxiMovement` keeps track on which thrustings are being used through 4 booleans, these booleans tell whether or not new forces should be added to the `PO` when getting an input from the player. The booleans are accessible to `Taxi` through dedicated properties, and the `Taxi` changes image according to these booleans.

6 User Guide

6.1 Running the game

To run the game, clone the game from our GitLab repository (See URL at front-page). Open the project in MonoDevelop/Xamarin Studio, then **Build** and **Run** the project.
If you prefer to build/run it from a terminal, you can do so by issuing these commands from the project root directory:

```
$ xbuild /p:Configuration=Release DIKUArcade.sln
$ mono GameEngine/bin/Debug/GameEngine.exe
```

6.2 Objective

A player takes on the role as a heliospheric cab-driver trying to make ends meet. The player has to pick up customers on platforms in levels of varying difficulty - trying not to hit the customers, avoid obstacles and landing safely on the platforms. The player will be awarded **$100** for each costumer he or she safely escorts to an international transit point, which can be reached by utilizing the warp-drive placed on the upper-middle of each level. The player only progresses to another level if he's managed to pick up at least one customer properly. Just like real life, the game is a never-ending grind, and as such - the cab-driver is never satisfied with the money earned. The game resets when the taxi crashes. The developers invite you to try and break the high-score.

Regarding Safe Landings

The SpaceTaxi will crash if the player does not land perpendicularly to the platform on which the customer is walking. Furthermore; if the player hits the customer while not safely landed, the customer dies - and **$0** are awarded. If the player manages to land safely on the platform, the customer will rush towards the taxi and be ready for take-off as soon as he steps in.

6.3 Controls

As described in section 3, the movement controls are handled by the arrow-keys, ↑, ↓, ←, →. Furthermore, the **SpaceTaxi Game** supports the following keys:

h	Displays quick help menu - and if a game is in progress, it will pause
p	Pauses a game in progress
q	Quits the current game window
space	Proceeds to load next level - either when starting up or when proceeding to next level
s	Saves the entire game to file
l	Loads the entire game from file

7 Evaluation

In evaluating the implementation of *our* additions to the project, continuous manual testing was conducted. When building the project, we always made sure that the added components ran and worked as intended. Given that many of the base-components were already unit-tested in **B4-DIKUArcade**, unit tests have only been conducted when creating *some* completely new classes. These classes do not rely on building blocks from the **B4-DIKUArcade**-project. All unit tests are deployed on **GitLab CI** using the **NUnit (version = 3.6.1)**-framework, ensuring a simple and concise CI process.
Integreation testing was primarily conducted by making sure that any newly created class worked on its own (with relevant dependencies) - and then simply applying a **Big Bang Approach** to the actual integration. This was convenient in most cases, but sometimes resulted in messy backtraces which slowed down the development process. This approach would not be appropriate for any larger systems with many developers - but overall, this approach worked well for this project.

7.1 `GameConcepts▸Entities▸EntityContainer.cs`

By implementing the `EntityContainer`, all entities are contained in a single class. Not only does this implement a container for entities, it also implements robust iterations through the collection of entities because it is possible to add and remove entities while iterating.
Each `Entity` knows whether or not it collides with another `Entity`, and this made it possible to refactor the collision detection.

7.2 `GameConcepts▸GameParser.cs`

The `GameParser` has been designed to work independently of all other classes - and is tested as such. 5 unit tests are deployed in `GameConceptsTests▸TestGameParser.cs`. Each test makes a *positive* assertion, affirming that the expected output is recieved. The tests focus on the class's ability to correctly find files and extract the expected information. Tests regarding end-of-file issues or files of different dimensions than described in the assignments have not been conducted.
The `GameParser` works well in the following regards:

- It's OS ambivalent.
- It only uses a single reader-object when parsing a **Game**.
- A new ASCII-file is added to the game just by putting it into the correct folder.
- Methods are split up sensibly and the class can easily be modifed in the constructor to accomodate ASCII-art of different dimensions.
- The file-detection methods employ exceptions and error-handling

The `GameParser` could be improved in the following regards:

- The API of the class is not entirely intuitive, and should be simplified.
- Two methods employ *Regular Expressions*. These may be difficult to maintain for an external developer.
- A better construcor interface could reduce the use of getter-methods.
- The methods of the class should be tested more extensively with *negative* assertions.

7.3 `SpaceTaxi`▸`GameManager`

The GameManager is designed to be the binding structure between the SpaceTaxiGame, the created file and SaveSerializables, and as such servers no other purpose than to make this connection. Furthermore, the class can be easily expanded to create different new files instead of overwriting the old saved file - allowing for saving and loading multiple games. With this simple structure, we haven't been able to make the program load wrongfully, however on certain very rare and seemingly random situations, the wrong level is loaded. This could maybe have been avoided with testing.

7.4 `TextSpaceTaxiGame`▸`TaxiMovement`▸`SpaceTaxiMovementTests`

As described in section 4, classes should strive to have individual responsibilities, which make it possible to test individual classes without instantiating new games. The `TaxiMovement` can be tested using only the class and a `Position`. These tests are all changing the `Position` according to a simulation of one or two `KeyPresses`. The tests include testing of movement in all four directions (Up, Down, Right and Left) and the four diagonal directions (Up and Right, Up and Left, Down and Right, Down and Left).

8 Conclusion

We have designed a simple version of the classic **SpaceTaxi** game. Our implementation fulfills the vast majority of requirements to functionality imposed by the deliverables. The solution has been extended to accomodate more entertaining gameplay than would otherwise have been the case, if we'd only

met the bare-minimum requirements. The game plays as intended, runs stably and we've experienced no crashes on the last commit. Overall, we've been successful in solving the task at hand.
In assessing the challenges faced by this group, one of the main issues have been the fractured composition of the group itself. This group has changed 4 times during the semester - and as such, it's taken more effort than usual to establish a proper workflow and group communication. The lack of a steady group has resulted in some of the deliverables to be inadequate - and we've had to compensate over the past two weeks. Given more time, the group would dedicate the majority of it to provide a better API for other developers and conduct more tests. Also, we'd implement a less trivial progression scheme with more levels, and a scoreboard.
The application is not as well structured as we'd like - but taking all else into consideration, we're content with the final result. In refactoring, we quickly discovered, that our previous hand-ins were not easy to maintain and develop further - we've been successful in simplifying and organising the application from that outset. Documentation and testing is deployed on GitLab and makes for a project that is extendable and maintanable - but any new developer will have to put in a considerable effort in order to do so.

Appendix A: Code Examples

Appendix A1

```
/// <summary>
/// Renders unattended customer. Will walk back/forth on the platform, on which he stands.
/// </summary>
/// <param name="area">Area.</param>
/// <param name="scale">Scale.</param>
private void RenderUnattendedEntity (Drawing.GameArea area, Drawing.Scale scale)
{
    area.Pixmap.DrawImage (_standLeft, scale.X, scale.Y);

    while (_state == CustomerState.Unattended) {
        while (_walkDirection == WalkDirection.Right) {
            if (this.X > _toX - _width) {
                StopCustomerMovement (_standLeft, area, scale);
                MoveCustomerLeft (area, scale);
            } else {
                MoveCustomerRight (area, scale);
            }
            return;
        }
        while (_walkDirection == WalkDirection.Left) {
            if (this.X < _fromX + _width) {
                StopCustomerMovement (_standRight, area, scale);
                MoveCustomerRight (area, scale);
            } else {
                MoveCustomerLeft (area, scale);
            }
            return;
        }
    }
}
```

Appendix A2

```
/// <summary>
/// Observes taxi location and executes the WalkToXCoordinate()-method
/// </summary>
/// <param name="area">Area.</param>
/// <param name="scale">Scale.</param>
private void RenderWalkToTaxi (Drawing.GameArea area, Drawing.Scale scale)
{
    var taxiLocation = _spaceTaxiGame.Player.X;
    WalkToXCoordinate (taxiLocation, area, scale);
}

```

```

/// <summary>
/// Walks the customer to the coordinate of a taxi, given that it's landed.
/// </summary>
/// <param name="x">The x coordinate.</param>
/// <param name="area">Area.</param>
/// <param name="scale">Scale.</param>
private void WalkToXCoordinate (int x, Drawing.GameArea area, Drawing.Scale scale)
{
    if (_spaceTaxiGame.physicalMovement.PO._velocity.Y < 0.0) {
        _state = CustomerState.Unattended;
    } else if (x > this.X) {
        MoveCustomerRight (area, scale);
    } else if (x < this.X) {
        MoveCustomerLeft (area, scale);
    }
}
```

Appendix A3

```
/// <summary>
/// Adds all Platform Entities and Customer Entities to entityscripts and initializes them
/// </summary>
/// <param name="platform">Platform.</param>
/// <param name="platformY">Platform y.</param>
/// <param name="platformKeyList">Platform key list.</param>
private void CreateCustomerPlatformFromAscii(List<Tuple<int,
        int>> platform,
        List<int> platformY,
        List<string> platformKeyList)
{
        var uniqueY = platformY.Distinct().ToList();
        var uniqueKeys = platformKeyList.Distinct().ToList();

        for (int i = 0; i < uniqueY.Count; i++)
        {
                var rnd = new Random();
                var y = uniqueY[i];
                var key = uniqueKeys[i];
                var path = _dict[key];
                var q = platform.Where(tp => tp.Item2 == y);
                var z = q.Select(tp => tp.Item1);
                var zMin = z.Min();
                var zMed = z.ElementAt(rnd.Next(0, z.Count()));
                var zMax = z.Max();
                var customer = new Customer(_gameContainer, _script,
                        zMed, y - _boxHeight,
                        zMin, zMax);
                _gameContainer.PendAddEntity(new Platform(_gameContainer,
                        _script, zMin, y, zMax - zMin + _boxWidth, _boxHeight, path, customer));
                _gameContainer.PendAddEntity(customer);
        }
}
```

Appendix A4

```
/// <summary>
/// Retrives the files in the Levels-directory
/// </summary>
/// <returns>The files present text-files in the directory</returns>
public string [] GetFiles ()
{
    string [] dirContents = null;
    try {
var currentDir = AppDomain.CurrentDomain.BaseDirectory;
var parentDir = Directory.GetParent(currentDir).Parent.Parent.Parent.FullName;
        var levelsDir = Path.Combine (parentDir, "GameEngine", "bin", "Debug", "Levels");
        dirContents = Directory.GetFiles (levelsDir, "*.txt");
        return dirContents;
    } catch (UnauthorizedAccessException e) {
        Console.WriteLine (e.Message);
    }
    if (dirContents != null) {
        return dirContents;
    } else {
        Console.WriteLine ("There's nothing contained in the specified folder");
        return dirContents;
    }
}
```