Exam-2018

Software Development 2018
Department of Computer Science
University of Copenhagen

Bess Alix <Qfd312@alumni.ku.dk>,
Christian Sass Hansen <Qtz972@alumni.ku.dk>,
Kasper Alex Madsen <cdt183@alumni.ku.dk>

Wednesday, June 6th, 15:00

Git url

https://github.com/BessAlix/su18-KMCSBA.git

1   Introduction

In this project we should use the game engine "DIKUArcade" to create a simple version of the game Space Taxi. We were introduced to this game engine in the last project Galaga, where we should learn how to use it. Now we have to use our knowledge from the last project and expand our vision by learning to use the features more effectively to make our next game "Space Taxi".

This report consists of all our work on implmenting a simple version of the game *Space Taxi* in C# using JetBrains Rider, which is part of the course *Software Development 2018* at *The University of Copenhagen*. It features: a backgound explaining the purpose of our work and the history of the game. An analysis, explaining our goals within the project, as well as the requirements to the game. A description of the design we have chosen, and a talk about the implmentation. A user guide, an evaluation and a conclusion on our work process.

2   Background

In the previous course we have been working with another simplification of a game called Galaga. Here we were first introduced to the game-engine DIKUArcade, where the assignments from 4g and up to 7g, was partly used to make us learn how to make use of the game-engine. So we in this project would have easier creating and working on implementing the game *Space Taxi*.

Space Taxi is an action game made for the Commodore 64, that is written by John Kutcher and published by Muse Software in 1984. This game simulates a flying taxi controlled by thrusters and which is being affected by the laws of physics. So when the ship flies upwards or to the left and right, the ship will accelerate gradually. When the player stops flying upwards the ship will start to fall by also accelerating gradually, and when the player starts thrusting upwards again it will de-accelerate gradually as well. The game is then about picking different costumers up and flying them to their desired destination pad, which can be on the same level or on another. The player will then earn some cash from the ride.

The onset of for the game of Space Taxi is basically a 40 x 23 grid, made from simple ASCII based text-file maps made into game levels:

```
%#%#%#%#%#%#%#%#%#%^^^^^^%#%#%#%#%#%#%#%#
#              JW       JW               %
%      h2g                               #
#      222                      >        %
%      H2G                          o    #
#       3                         o     %
%       3                                #
#       3                          o    %
%       3                     j%i       #
#       3                     W Xi       %
%       3                        %       #
#                               xI       %
%       o                       xI       #
#                               xI       %
%                              xI        #
#  o   o                       xI        %
%                             xI         #
#    o                       xI          %
%        o                  xI           #
#  o                        xI           %
%          o                I            #
#          m1111111111111n               %
%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#
```

Figure 1: ASCII-File of Short-n-Sweet

Then, one player will control the space taxi around the levels "Short-n-Sweet"
and "The-Beach":

Figure 2: Game level of Short-n-Sweet

The purpose of this project is not to showcase the DIKUArcade framework, but
to learn how to produce a software product that meets some given specifica-
tions. The game may not be in the final state, but all the components should
work as intended.

The development of our version of the game *Space Taxi* was made over 4 parts,
all comitted by a deliverable. In each of these deliverables where were different
requirements there had to be met for passing the assignments. The process
meeting these has been stated in our previous reports. In this final deliverable
we will do the same, but this time we will be looking more at all the work
we have done up until now. Our development process will focus more on
delivering a code that will be easily readable and maintainable so it also can
be further developed by other people, instead of adding new features. I.e. we
will be focusing on refactoring and testing the code in this last deliverable.

3   Analysis

Our goals for this final deliverable is stated down below. This section will explain these goals, the relevance of them and what specifications that are required in this project.

Goal 1: Meeting the requirements

This goal is one of the most important goals we have. It concerns about the functionality and requirements of our game. It helps us making sure that our game moving towards a playable game. We need to meet the requirements of the *deliverables* 7-10, which all sum up to be the finished game.

Goal 2: Design patterns

This goal concerns utilizing sensible design patterns, where they are relevant. In software development, a design pattern is a general, and reusable solution to a commonly occurring problem within a given context in software design. There exists many different design patterns and each of them are used for different things, while some also recalls other patterns. The approach of using a design pattern can be rewarding for the code quality and can help solve a given problem. The approach can be recognized by many other developers who have to read, understand and edit the code, which also helps supporting goal 3 and goal 6, as it standardizes the code. This is one of our goals, since we have to research different design patterns and get to learn using them in some different occasions.

Goal 3: Maintainability

This goal concerns making the code maintainable, understandable and accessible to other developers who may want to change it. This can be achieved by using names that is appropriate to what each method, class or variable is meant to do. By writing a good documentation will also allow readers to form better mental overview over any solution, and will together with explicit names create a code, that is much more readable. Then updating the code it may also fix flaws thay may be hiding in the program. By keeping the SOLID principles in mind, when designing it will also do much for the maintainability of the code.

The maintainability of this project is a very important part of the development for us, as we are developing the project in a group. We have been sitting together during the designing of the code, and communicated about the design and the quality, ensuring that the code would be readable by everyone in the group. This was important since we were working on differnt parts of the code at the same time. In this deliverable we will return to old pieces of code, which we have to refresh our minds with and do some refactoring, that will make it easier for other people to read.

Goal 4: Tests

This goal concerns tests. Tests can be coded to ensure that methods and classes actually do what they are supposed to do. Tests are also able to help exposing if there are any bugs located in the code. By having tests which checks troughout the program, that the units of the code functions as intended will improve maintainability, and it supports goal 3. If new changes are made, any unintended effects should be caught by the test. That means all classes that we have written requires testing, except the game class, as this heavily relies on the GaneEventBus (has already been tested). The code should also be designed, so methods and classes are suitable for unit-testing.

Goal 5: Design for extensibility

This goal concerns designing the project for further extensibility. Extensibility is a software development and systems design principle where the implementation takes future growth into consideration. The goal here is to accomodate for expected future additions to avoid the need for some major refactoring in order to include additions to the program, while also maintaining a reasonable structure.

Goal 6: Coding style

In this project we should stick to a specific guideline for coding style and naming. This will in many cases, help readers to get an better overview of a code. By using one guideline it also helps ourselves as it will keep away unnecessary confusion which may or may not lead to errors. In our group we have done our best to follow the guideline provided to us, as well as to follow the naming guidelines which are stated by the .NET Framework Naming Guidelines. This goal impacts on readability, which also supports goal 3.

3.1 Requirements for Space Taxi

For the game including this project, it is needed to have a player-taxi that can be controlled by the player. The ship should be affected by the laws of physics, which consists of: When accelerating upwards and sideways it should be accelerating gradually, which means as it is in space it should go faster and faster the longer the player holds down the keys to the left, right or up, also while its being affected by gravity. When the player stops pressing the up-key, which is accelerating the player-ship upwards, the ship should gradually start to fall. The fall should have the same accelerating features as when pressing the left, right or up key, but as it is falling due to gravity, this will go faster than the acceleration from the thrusters. To de-accelerate from the fall when pressing the up-key it should as well de-accelerate gradually. This will overall simulate the laws of physics and gravity. The player should as well be able to pick up customers by landing on platforms and then fly them to their desired next platform, where they will be dropped off and disappear. By "delivering" customers it should give the player money (points basically). It is also needed

to have a timer showing frames and updates pr. second, as well as an event handler, and a state machine which will handle different states in the game, as then the player are put into the main menu or teleporting to another level. Or other examples as: when the game is running, pausing the game and quitting the game.

This is some of the things, that is needed to make Space Taxi. The frame of the game can contain:

1. Start the game.

2. Starts in the main menu, where user inputs can be accepted as ("Key up", "key down" and "key Enter"), to start a new game or quit the game.

3. When starting the game, draw and render game elements (background, objects, obstacles, platforms teleportation device in the top middle screen to change level and player-taxi.

4. Position the player ship up in the right corner, beside the sugar cane on the left side. Accept user inputs (key-left, key-right and key-up).

5. Position customer, which can be picked up at a platform and dropped off on a next one.

6. Depending on the user input, handle the event, change the game state, draw the next frame.

The game should be a single-player game, where the player will be introduced with a main-menu screen. Here the player should be able to select between starting a "New Game" and a "Quit" game button, as you then will have what is needed for a functional working game. The game should be able to quit the game as you hit the quit button, as well start a new game, then hitting the "New Game" button. Then pressing the "New Game" button, the game should start and the player will be put into the game. Here should the player start in the first environment "Short-n-Sweet", which technically should work as the first level. Here the player should see the player-ship (the space taxi), that can be moved left, right and upwards by using the keys for left, right and up. The player will also be able to pick up customers at platforms and deliver them to their desired next platform, and by that earn money. The game should run with 60 frames pr. second and have 60 Updates pr. second, as well be fast and responsive then hitting the buttons. If not, it will be annoying for the player if the buttons does not response immediately, as it will make the game frustrating when you want to de-accelerate at the right moment to avoid hitting obstacles, which will cause the ship to explode. To go to the next level, the player should be able to "collide" with the portal in the top middle of the screen and then get teleported to the next level "The Beach".

To interact with the main-menu and the game, the game needs the implementations:
For instance, to interact with the menu:

⬆ Go one choice up, if possible.

⬇ Go one choice down, if possible.

↵ To select your chosen menu button press enter, if possible.

When the player chooses "New Game", the game should start, while if the player chooses "Quit", it will Quit the game. When the game starts, the player should see the ship placed in the top right corner of the screen on the left side of the sugar cane. The player will also be seeing the level itself, containing a background, walls, obstacles, the portal and the platforms that can be used to land on and pick up costumers as well as deliver customers at. The player should be able to move the ship to left, right and upwards, while also noticing that the ship is being affected by physics. Furthermore, the player should be able to go to the next level by using the teleporter in the top middle of the screen. To make this happen, if possible it is needed to implement, the following keys:

⬆ Will accelerate the ship upwards, if possible.

⬅ Will accelerate the ship to the left, if possible.

➡ Will accelerate the ship to the right, if possible.

If possible the player should also be able to pause the game using the "P" key on the keyboard, where the player should be able to quit the game by pressing the Q-key and start a new game by pressing the N-key. To make this possible it is needed to implement the following things:

P Hit the P-key to pause the game, if possible.

Q After hitting P-key, press Q to quit the game, if possible.

N After hitting P-key, press N to start a new game, if possible.

Design goals relating to technology stack choices:

1. The game should be written in C#, as a JetBrain Rider project, version controlled with Git - the technology stack of our course.

2. The game should run across the different operating systems.

Design goals relating to coding style:

1. The game needs to be made using the game-engine DIKUArcade.

2. The game should be able to handle the player being affected by physics, when the player-ship collides with an obstacle, when it should land on a platform, and when going to the next level. It should draw and render the game and its elements according to the different game states.

3. The game should be designed such that the GameEventBus can be accessed through a GetBus method, so we can access the same GameEventBus from anywhere in our program, and not only in the Game class. Which means that the game should be designed, so we can handle events anywhere in our program.

4. The program should be designed such as, instead of having all of the methods which relates to the player class inside the Game class, it should be in a class of its own named "Player".

5. The game should be designed so it has a state machine, that will handle the different states, such as when game is in the main menu, when the game is running, when the game is paused and quit game. As well when the ship collides with obstacles, it should explode, when it comes near a platform, it should land and when it collides with the teleporter, it should change the level. Furthermore, be able to transition between the different states, changing which state is active.

4 Design

One of our most important design goals is to make the program as conceptually sane and maintainable as possible (see goal 3). Moreover, by having to build on DIKUArcade and meeting all of the given requirements (see goal 1), it has had a great impact on our design decisions.

4.1 Design decisions

In the following first table we have created a responsibility table, where we describe some concepts of our design we need to implement for *Space Taxi*.

Responsibility	
Responsibility description	**Concept name**
Load level to simple data structure.	Loader
Generate level entities.	levelParser
Container which hold level entities.	Level
Player entity.	Taxi
Obstacles where a taxi can land.	Platform
Non-moving entities.	Obstacle

In the following table we will deduct the associations between our concepts of the design. I.e. how our concepts are related.

Responsibility		
Concept Pair	**Association descriptions**	**Associate name**
levelParser ↔ Loader	LP using data structure from Loader to create level.	makeMap, getTaxiPosition, makePlatforms, makeObstacles
Level ↔ levelParser	Level get object from LP.	getLevel
Taxi ↔ Level	Level knows about taxi.	getTaxiPosition
Platform ↔ Level	Platform is part of a level.	platforms (list of platforms)
Platform ↔ Taxi	Platform knows about taxi.	isLanding, Landing
Obstacle ↔ Level	Object is part of a level.	obstacles (list of obstacles)
Obstacle ↔ Taxi	Taxi collide with an object.	collideWith

In the tables before, we have identified the main responsibilities and their associations in our design of the game. Therefore, we will now consider what kinds of attributes our concepts in our game requires.

Responsibility		
Concept	**Attributes**	**Attribute Description**
Loader	Level	list of strings
LP	Height	Number of rows in the map
	Width	Number of chars in a row
	level	List of strings given from Loader
	map	Sublist of level which contain the gamemap
	images	Dictionary of keys and correspondend image
	name	Level name
	xscale	ratio on the x-axis of number of elements
	yscale	ratio on the y-axis of number of elements
	errorMessage	Internal error handler
Taxi	position	Position of the player
	image	Graphical representation of the player
	direction	tracker for direction
	hasCustomer	Boolean value corresponding to the name
	shape	physical parameters of the taxi

4.2   The overall structure of the game

Game

Game is a singleton class, around which the entire game evolves.
This class is a non-terminal method GameLoop, which render the game screen through the different states and entities. This class also maintains the statemachine and the eventbus (SpaceTaxiBus) which send the keyboard events to statemachine.

SpaceTaxiBus

A singleton class, that is a wrapper for the DIKUArcade.EventBus the solo purpose of this is to get a static eventbus to be called from anywhere in the program.

StateMachine

A singleton class, that handle the state switch and direct the events to the active state.

States

    MainMenu – Initiate state where the player can enter a new game or quit.

    NextLevel – This state will handles the state where the ship teleport to a new level, that has the new environment, background, objects, obstacles and platforms.

    GamePaused – This state pauses the game momentarily, where the player should be able to quit the game.

    Quit – This state will quit the game.

LevelBuilder

    IFetcher – will be an interface the loader will use to comply with some expectations of the output.

    IParser – will be an interface the parser will use to comply with some expectations of the output.

    Level – will handle an active level.

    LevelParser – LevelParer is an iparser, an implementation as given a list of strings making all level entities, which level can accept.

    Loader – is an IFetcher implementation, it takes a filepath and returns a list of strings that the file constitutes and removes any blank lines.

SpaceTaxiEntities

    ICollision – is an interface that all the SpaceTaxiEntities will implement, except the player. Uses a boolean CollidWith.

    Customers – implements the interface ICollision that will use this, so when the player collides with the customer, the customer should disappear. It will handle them in general, but it will get all data for all the customers.

    Obstacles – implements the interface ICollision that will use this, when the player collides with an obstacle, it should explode.

    Platforms – implements the interface ICollision, that will use this, when the player gets close to the platform it should land.

    Player – Everything there is about the player will be handled in here. Orientation, images, animations, player speed etc.

    Portal – will be the port that will teleport the player to the next level. It will uses collision detection to detect when the player collides with the portal area, it should teleport the player to the new area.

Movement

> **IMovement** – is an movement interface that will have a method void Move.

> **OnPlatform** – will be used to track movement near a platform. It will also make sure that when the player-ship is on the platform, it should not fall off the platform, which means it turns off gravity temporarily.

> **TrivialMovement** – All the physics in the game will be handled in here.

GameConstants

> This static class handles all constants in the game. For example, player speed, physics, screen size, explosion duration, and the game timers.

4.3   How well did we follow the SOLID principles?

The SOLID-Principles, which stands for:

1. Single responsibility (SRP)

2. Open-closed (OCP)

3. Liskov substitution (LSP)

4. Interface segregation (ISP)

5. Dependency inversion (DIP)

In terms of the SOLID-Principles we have chosen to represent each of them the following way, so we can describe how well we have followed them:

Single responsibility:
The single responsibility principle is about that every class, function, variable should define a single responsibility. That means it has only one reason to change. If you change anything in that class, it will effect only one particular behavior of the software.

So in terms of SRP we have for example the player-class. In this player class, everything there is about the player is inside this class. The player-movement, the physics affecting the player, the player orientation, thrusters, images. And every variable in this class is for one thing only. We have: the GravityDirection variable is only used to make gravity, the ThrusterDirection concerns only about the thrusters and so on. Other examples could be our Obstacles class, Platforms class and Portal class, which only concern about the obstacles, the platform and the portal (which changes level by the player "colliding" with it).

Open-closed:
The open-closed principle concerns that software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification. Such

that an entity can allow its behavior to be extended without modifying its source code. This can be achieved with inheritance. You don't have to touch the class you want to extend if you create a subclass of it.

To cover the open-closed principle we for example use inheritance a lot in our program. We have e.g. an interface "ICollision" which will handle the different types of collisions. As colliding with a platform it should have one action, than colliding with a wall. We use this Interface in the separate classes: Customer, Obstacles, Platforms and Portal, which also inheritances from another class called Entity, which handles entities.

Liskov substitution:
The Liskov substitution principle concerns objects in a program, that should be replaceable with instances of their subtypes without altering the correctness of that program. It can be though of an extension of the Open-Closed principle.

In our game we use this with the parser and the fetcher. The constraints on the substitute of the class are fairly weak, since we only demand that the output has a certain type. But nevertheless we could substitute the loader with one which load the levels of github, or some other place.

Interface segregation:
The interface segregation principle concerns that no client should be forced to depend on methods it does not use. That means classes should be as specialized as possible. You do not want any "god" classes that contain the whole application logic.

In our game we for example have the game elements: Customer, Obstacles, Platforms, Portals and the player, that all should have the feature "collision". Some will use this feature different than the others. That means, for example when the player-ship collides with an obstacle it should explode, but if the ship collides with a portal it should teleport the player to the next game level. So instead of we have one class which performs all the actions, we have created an Interface and 5 different classes, one for each event type. We have therefore also made a ICollision interface so the client will not be forced to depend on methods it does not use. And we have our 6 classes: Customer, Obstacles, Platforms, Portals and the player, where Customer, Obstacles, Platforms, Portals uses the interface ICollision to make their own collision action.

Dependency inversion:
The dependency inversion is about that high-level modules should not depend on low-level modules: both should depend on abstractions. To create specific behavior you can use techniques like inheritance or interfaces.

We have for example use this in the way we load our game level. The concrete way in which the implementation of our interface Ifetcher operates is not dominant of how Iparser reads it. The interface Iparser implementation has only the requirement that the output is the same structure (i.e. List<string>),

besides that Ifetch of course also expects the level text-file to have a specific setup. As well as that level uses the Iparser interface and gets some output, but how it is found and made does not matter to the level class.

Conclusion
Overall, we think we have done the best to follow the SOLID Principles, but as we know, where are always things that can be improved. We haven't used some of the SOLID Principles as much as the others. That could for example be the Liskov substition principle. In our code, we have not used this principle as much, because we have chosen to implement our code leaning more heavily on the simpler, Open-Closed principle.

5   Implementation

5.1   Implementation of SpaceTaxi

Our implementation of the game Space Taxi is a C# Console Project created in Jet Brains Rider. We have chosen to create our unit tests in an neighboring NUnit Library Project. The main entry point is in our Program.cs file which is responsible for calling an instance of Game.Loop from our Game.cs file. The Game.cs file is the most important file though, as it is subscribes to all events that happen while the game is running.

StateMachine is an implementation we have done so the game handle different states. The states are for example if the player are in the main menu (state MainMenu), pressing either "New Game" or "Quit". If the player starts the game, the state that will be happening is "GameRunning". When the player pauses the game, it is a state too (state GamePaused). Inside the GamePaused state, the player can also quit or start a new game, which will be the "New Game" state, and the "Quit" state. When the player telports to the next area using the portal, the state event it uses is the "Next Level" state.

Player (and Physics from TrvialMovements class):
In the Player class here we handles everything there is about the player. That contains the ships orientation, its images for the ship and thrusters (which will animate), when the player hits an obstacle it will explode, the player speed, and uses the implementation of physics from the "TrivialMovements" class to simulate physics. It handles as well when a key is pressed, what will happen when key-left is pressed, key-right and key-up, as well as when they are released again. These physics makes the Taxi movement feel more realistic and simulate gravity. We have moved the Player event handling from Game to player, and made it static too.

ICollision This is a interface and therefore the classes that have assets that can either be collided with or landed on can inherit from this interface. The interface has a method which is returns a bool that indicates if collision has occurred or not.

Level Level is a IGameState, and is the main part of the game, it handle all rendering and collision checking of entities with the player, It hold all objects for the game except from player.

Loader Inheritance from the interface IFetcher. This public class opens a stream and reads the strings that are created in IFetcher. The Loader Class load all the text from the file into a string list and filter all empty strings away from the data.

LevelParser This is a public class which take the simple structure from a Ifetcher and structure it into all the game Entities, find the location and orientation of the player. It use the interface IParser which make constrains on how the expected output are accessed from outside. The LevelParser are structured so, that it does not need to take a specific size of map, say 40 times 23, as long as the map is consistence[1] it will parse it.

IParser This is a public interface that is responsible for our LevelParser Class returns the correct piece.

IFetcher This Interface are used by any loader which the parser shall use, it gives a standard of what to except of the loader.

Platform Inheritance from ICollision and Entity. Rather straightforward, this public class' primary function is to be a helper class for collision detection for both the Taxi and the Customer.

Obstacle Inheritance from ICollision and Entity in DIKUArcade. This public class takes a shape and an image and creates the obstacles in the game.

Customers This public class inherits both from our ICollision interface and Entity in DIKUArcade. The customer is given movement and boundaries so that it doesn't walk off the edges of the platforms.

Portal Inherits from ICollision interface and Entity in DIKUArcade. This public class faciliates the passage between levels.

6   User Guide

To run SpaceTaxi, clone our project (see URL on the front page), open SU18-Exercises/SpaceTaxi-1/SpaceTaxi-1.sln in your preferable IDE, and run the SpaceTaxi-1.sln solution.

When starting the game, you will be introduced to the main menu screen. In the main menu you can choose to start a new game or to quit the game. The controls in the menu, where you can select between each interaction choice,

---

[1]each line in the map has same length

you should use the following keys:

⬆ Go one choice up.

⬇ Go one choice down.

↵ To select your chosen menu button, press enter.

If you choose "New Game", it will start up the game, while if you choose
"Quit", it will Quit the game. So if you choose "New Game" the game will
start, where you will see your ship in the top right corner of the screen placed
beside the sugar cane on the left side. The game will begin when you press one
of the keys. Then the ship will begin to fall and will keep falling until you hit
an obstacle and explode or when you hit the up key that gradually will stop
the ship from falling and push the ship upwards. You can also move the ship
to left and right. However, be careful as your ship is being affected by physics
(gravity and acceleration). If you are going too fast or moving your ship un-
controllable, you may hit an obstacle on the way and the cause will kill both
you and your customer. To move your ship, you will use the following keys:

⬆ Will accelerate the ship upwards.

⬅ Will accelerate the ship to the left.

➡ Will accelerate the ship to the right.

If you want a break from the game, you can do that by pausing the game. If
you want to quit the game, you can do it inside the pause menu, as well. As
well as starting a new game. To do the following things you need to use:

P Hit the P-key on the keyboard to pause the game.

Q After hitting P-key, press Q to quit the game.

N After hitting P-key, press N to start a new game.

6.1   Game guide

When running the game you will start in the Main Menu, where you can choose "New Game", that will start the game and "Quit", that will quit the game. The options which are green, is there you currently is. So if you move one option down, the Quit option will then be green.


When starting the game, your job as a taxi driver in SPACE, is to get to your customer and pick them up at their platform where they are waiting.
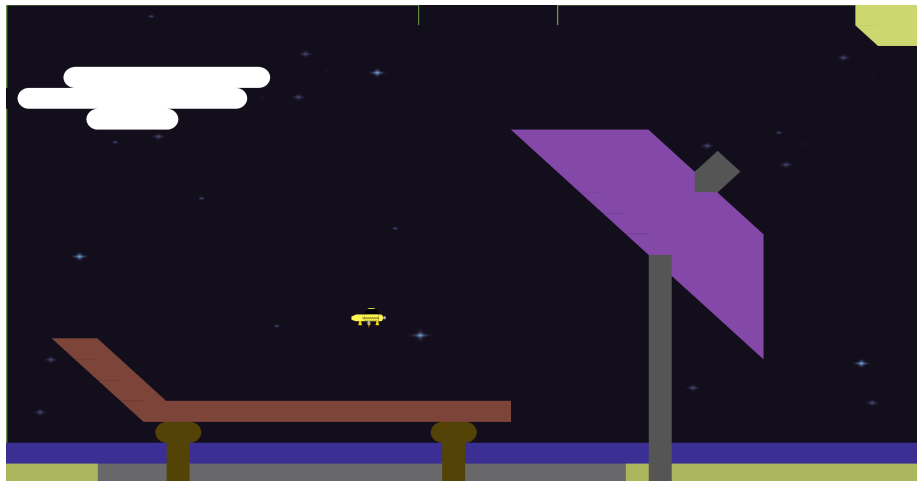

From here on, you should drop off your customers at their desired destination (next platform) in time or they might get angry at you and pay less money.

To get to the next level when finished the current level, you should collide with the portal up in the top middle of the screen, that will teleport you to the next level.
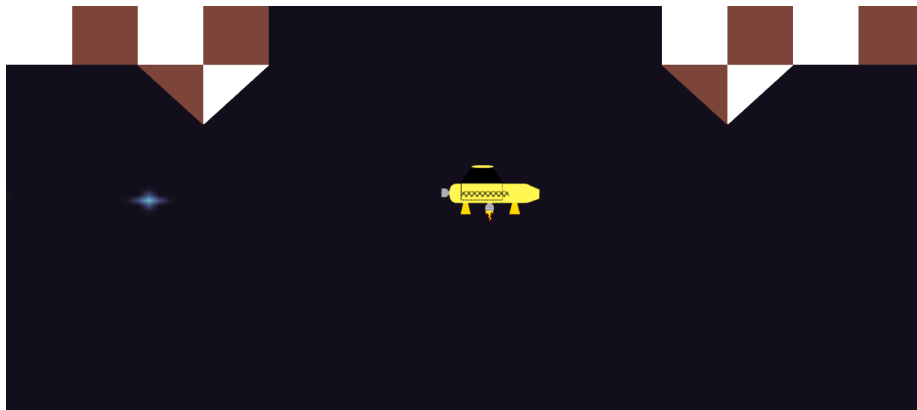
The next level will then be "The Beach":

7   Evaluation

7.1   How we designed and implemented the tests

We has had a lot of problem with the NUnit testing framework and therefore hasn't implemented any test.

Most of our test, has been made along the process of the design of the game in general. We do not test Implementation which only relay on DIKUArcade entities and their methods, since we assume that those are well tested.

- For the Loader, we have tried to do some testing. here we are testing that the loader response accordingly to the expected behavior. This includes what happens when the file isn't are missing, corrupt and under normal behavior.

- The parser we intended to do some unit testing but also a little integration testing. Since the LevelParser only can get input through an implementation of IFetcher, we needs to test if it gets one or not, then we would test through a dummy Ifetcher implementation that the LevelParser git the expected output. That all obstacles, platforms, portals are there and the taxi are set at the right position and orientation. This are trivial since all thoughs data are output of the class.

- IMovement Implementations, are quiet hard to test since, we needs to check if it moves the entity as intended. This would be done by setting a level through the dummie from the LevelParser test, and then call the move method once, then check if it works as intended.

- ICollsion, we would test as IMovement, we would set up different levels which takes into account all situations like the taxi colliding with an obstacle, platform or landing.

Since all game entities are inheriting DIKUArcade Entity and only implement the ICollision interface we has already tested all that isn't the Entity part.

7.2   The non-trivial parts of our implementation

7.2.1   LevelParser

To efficient parse the text to a level, we choose to first split the string list up to several pieces, as the map represented as a list of strings and an image dictionary. Most of the entities are actually first created when asked for, the further preprocessing of the data before creating the entities, enhance the simplicity of how to create entities and searching for the different part of the data.

7.2.2   Collision

The Collision was difficult to design so that it was highly adaptively and easily implemented. We have used a Interface to conform the collision process, since we wished to use our collision design to handle all collision as collision with obstacles, platforms and portals, but also the customers collision with the edges of the platforms. The interfaces makes this possible since we now can us polymorphism to run the checking over all implementation of collision. This Design made it also possible to enhance the single responsibility part since the obstacles, portals and platforms all has their own collision strategy.

7.2.3   Movement

As in Galaga we chose to use movement strategies. This enhance the ability to meet all cases of movement, since we then could specify a movement pattern

when we are near a platform and when we are not. The OnPlatform movement pattern, takes a shape of the platform that the taxi are in the vicinity of and then hold the information need to easily check if the taxi collides with it or lands on it, without having to do it with every platform. Since the taxi change pattern depening on it location, it automatically update which platform it is in the vicinity of.

7.3 10g Implementation

We did not finish the implementation of this exercises, most part of the Customers class are implemented, We chose to use a class which handle all about customers. This simplified the parsing from the file, and the handling of customers in general. The Score would have been a class with a integer attribute score, three methods GetScore, AddToScore and Render.

7.4 Ambiguities in exercise sets

Perhaps the most significant ambiguity in this particular exercise set was when we had to implement physics. We got confused by NewtonianForce class in DIKUArcade and thought that it was from that class that we needed to build further upon. This was not the case and the real solution turned out to be few lines of code in our Player class and our implemented TrivialMovement class. This code snippets can be seen below and is found in at lines 58 and 59 in Player.cs in SpaceTaxi:

Player:

```
1  Vec2F ThrusterDirection = new Vec2F(0.0f, 0.0f);
2  Vec2F GravityDirection = new Vec2F(0.0f,
       GameConstants.GRAVITY);
```

TrivialMovement (line 12 to 30 in TrivialMovement.cs):

```
1  var shape = (DynamicShape)
       Player.GetInstance().Entity.Shape;
2
3  var updates = Game.GameTimer.CapturedUpdates;
4
5  Vec2F thrusterDirection = Player.GetInstance().Thruster;
6  Vec2F gravityDirection = new Vec2F(0.0f,
       GameConstants.GRAVITY);
7
8  float relativeSpeed = 1.0f;
9
10 // Using GameTimer.CapturedUpdates to calculate how many
11 // updates happened during the lastsecond.
12 if (updates > 0) {
13 relativeSpeed = 60.0f / updates;
14 }
15
16 shape.Direction.X += (thrusterDirection.X +
       gravityDirection.X) * relativeSpeed;
```

```
17  shape.Direction.Y += (thrusterDirection.Y +
        gravityDirection.Y) * relativeSpeed;
18
19  shape.Move();
```

8   Conclusion

As shown by our completed project, we have created a functioning playable game using the DIKUArcade game engine, that can be compiled and run on multiple platforms. The game is as well tested for the purpose to test if the functionalities we have implemented works as intended. Our version of the game *Space Taxi* has very limited features, but it is almost complete in the sense that it meets alomst all the given specifications. Sadly we could not figure out how to make so the player can pickup cusomters. However, we still believe we have met all requirements, (except the feature where you can pickup customers) for this project and have succeeded all of our goals we have sat for ourselves (see Goals).

The game for example has a level-builder that currently creates 2 ASCII levels, where the player can control a taxi which is being affected by physics, can teleport to the next level by colliding with the portal in the top middle of the screen. The player can land on platforms where the customer is standing (has no other functions), the ships explodes by colliding with an obstacle and it explodes by landing to fast. We have made so the game contains a statemachine which gives the player the opportunity to start a new game or quit the game in a main menu. As well as pause, quit or start a new game in this state.

In addition to that we have learned more about the programming language C# and its practical application, where in this particular situation, how it can be used to create a computer game. We have also learned to use different design patterns and formed a new way to think about the design of a program. That was, we had to use object oriented analysis and list up different design concepts in 3 responsibility tables. That helped us, to get an overview over which concepts we wanted to use, what their association should be and what they should be a part of. We used those new techniques to develop our game, and learned new ways of thinking and designing of programs, on the way. While developing we have been getting more routined using Github, but it have caused a lot of problems under the way, but not as bad so it have destroyed anything that could keep us away from finishing our work. In this deliverable we have refactored and made the overall structure of the code more readable, and by that more maintainble, which helps understandment for further development by us or other peoples. However, given more time we could have done more refactored, but as our solution is now, we are satisfied enough with what we have.