UNIVERSITY OF TARTU
Institute of Computer Science
Computer Science Curriculum

Pelle Holden Porila

Golfotron – A fast-paced 2D game

Bachelor's Thesis (9 ECTS)

Supervisor: Mark Muhhin, MSc

Tartu 2022

Golfotron – A fast-paced 2D game

Abstract:

This thesis describes the design and implementation of a fast-paced 2D action game called *Golfotron*. The main goal of the game is to provide players with an enjoyable experience through its unique game mechanics and visual design. The thesis starts with an introduction to the action game genre, moving on to game design, implementation, and visual design. Additionally, there is an analysis of similar games, from which *Golfotron* drew its inspiration, within the chapters. The game was playtested on potential players to find problems and assess the sleekness of the game. Based on the test results many improvements were made, fixing already existing issues and implementing new ideas for enhancing *Golfotron* further.

Keywords:

Computer game, action game, playtesting, Unity, computer graphics, game design, 2D game, programming patterns

CERCS: P170 Computer science, numerical analysis, systems, control

Golfotron – Kiire tempoline 2D mäng

Lühikokkuvõte:

See lõputöö kirjeldab kiire tempoga 2D-märulimängu nimega "Golfotron" kavandamist ja programmeerimist. Mängu peamine eesmärk on pakkuda mängijatele nauditavat kogemust selle ainulaadse mängumehaanika ja visuaalse disaini kaudu. Lõputöö algab sissejuhatusega märulimängu žanri ning liigub edasi kirjeldades mängukujundust, teostust ja visuaalset disaini. Lisaks on ka analüüs sarnastest mängudest, millest Golfotron inspiratsiooni ammutas. Mängu testiti potentsiaalsete mängijate peal, et leida võimalikke probleeme ja hinnata mängu ladusust. Testimiste tulemuste põhjal tehti palju täiustusi, parandades juba olemasolevaid probleeme ja rakendades uusi ideid Golfotroni edasiseks täiustamiseks.

Võtmesõnad:

Arvutimäng, märulimäng, mängu testimine, Unity, arvutigraafika, mängudisain, 2D mäng, programmeerimismustrid

CERCS: P170 Arvutiteadus, arvutusmeetodid, süsteemid, juhtimine (automaatjuhtimisteooria)

Table of contents

1. Introduction....4

2. Game Design....6

2.1 Similar games....6

2.1.1 *Geometry dash*....6

2.1.2 *Golf with your friends*....7

2.2 Game Mechanics....8

2.2.1 Throwing the disk....9

2.2.2 Catching the disk and redirecting it....10

2.3 Level Design....11

3. Implementation....12

3.1 Unity Game engine....12

3.1.1 Scripting in Unity....13

3.2 Overall game flow....14

3.3 Structure of the core game loop....15

3.3.1 Structure of a level....15

3.3.2 Structure of a player game object....16

4. Visual design....19

4.1 Visual style of *Golfotron*....19

4.1.1 Visual design inspiration....20

4.2 Implementation of visuals....20

4.2.1 Signed distance fields....20

4.2.2 Neon effect....23

5. Testing....26

5.1 Playtesting and its importance....26

5.2 Methodology....27

5.3 First test iteration....28

5.3.1 Results........28
5.4 Second test iteration........29
5.4.1 Results........29
6. Conclusion........30
References........31
Appendix........32
I. Launch Guide........32
II. Accompanying files........33
III. Assets........34
IV. Source code........35
V. License........36

1. Introduction

The video game industry is a fast-growing industry that has a lot of potential for creating revenue. In 2021 the video game market was valued at approximately 198.40 billion USD, with predictions of it reaching 339.95 billion USD by 2027.[1] In order to benefit from this market video games need to be developed.

For this thesis, I developed a fast-paced action-platformer genre video game *Golfotron* that includes a combination of game mechanics and visual design that would create an unique and enjoyable experience for the player. The game design follows the main principles of the action genre and contains challenging aspects which require fast reflexes and good eye-hand coordination to overcome.

In the game *Golfotron*, the player controls a neon disk that they can throw with a desired direction and speed, which provides the primary interface for the player to interact with the game. Furthermore, the gameplay is divided into ten different levels. The goal in the level mode is to reach the end of the level while avoiding numerous obstacles. While playing a certain level, the disk can be caught and redirected a limited number of times, which can slow the player down or, when hitting an obstacle, cause the player character to perish. Due to the set time limit, the end of the level has to be reached fast. If the time limit expires before reaching the end, clearing the level is deemed unsuccessful, and, again, the player character perishes. Moreover, a scoring system incentivizes the player to clear the level as fast as possible, rewarding the player with a higher score.

Chapter 2 focuses on describing the game design of *Golfotron*. It defines the genre, similar games, and main design points. Additionally, it elaborates on the game mechanics and level design. Chapter 3 gives an overview of the game engine used in creating *Golfotron* and explains the game logic, structure, and programming patterns used to implement it. Chapter 4 describes visual style, inspirations, and implementation processes. Chapter 5 discusses the testing of *Golfotron* with an overview of the given methodology. Two testing sessions and their results were analyzed.

[1] https://www.mordorintelligence.com/industry-reports/global-gaming-market

The launch guide of *Golfotron* is in Appendix I and the games' build is in Accompanying files (Appendix II) with a demo of the game. Assets used from the internet are in Appendix III and the repository link for the source code is in Appendix IV. A level of *Golfotron* is shown in the Illustration 1.

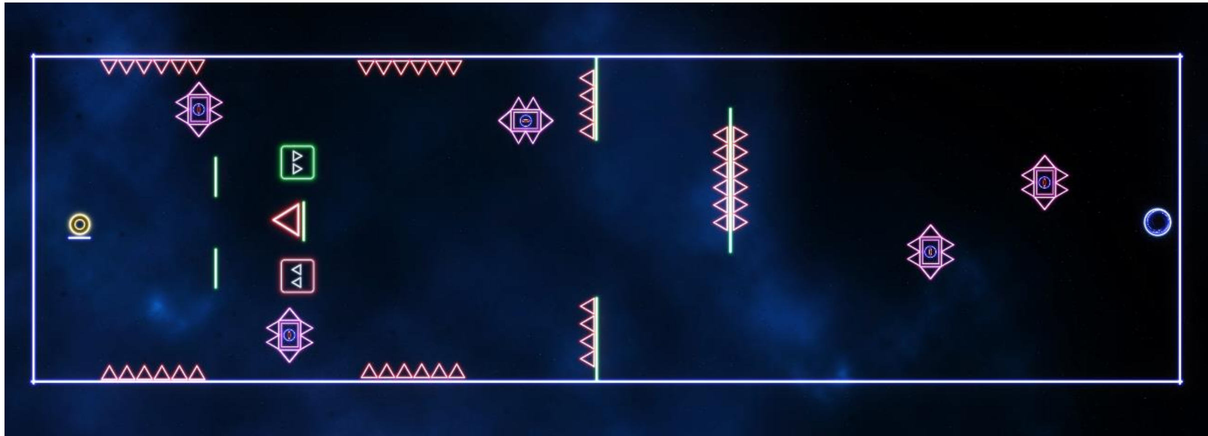

Illustration 1. Screenshot of a level in *Golfotron*

2. Game Design

Golfotron was designed to be an action genre video game, which is a game that relies on eye-hand coordination, skill, and fast reflexes [1]. Additionally, the game's design intends to create a unique gameplay experience characterized by its fast-paced, unique combination of gameplay mechanics, distinct visuals, and modular levels with a high potential for variety.

Golfotron is designed following a *playcentric design process*, in which from the initial conception of the game, the player is involved in the designing process [2]. Before the game development began, potential players were asked about the game's concept and if they saw any potential in it. Development ideas were prototyped and playtested with players while gathering their feedback. Further on, changes were implemented to prioritize the players' game experience.

Firstly, subchapter 2.1. gives an overview of similar games and how they've inspired *Golfotron.* Secondly, subchapter 2.2. explains the game mechanics and the player's interaction with them. Lastly, subchapter 2.3. defines the main philosophies of the level design process.

2.1 Similar games

There are thousands of different games globally. Thus when creating a game, it needs to stand out from the competition by creating something unique or improving the already existing game concept. I took different ideas from other games and improved them, making *Golfotron*.

Golfotron is an action game with many similarities to the platformer subcategory of the action genre. For this reason, action/platformer genre games were researched to improve my own game and make it stand out. Additionally, as *Golfotron* aims to have some unique gameplay mechanics, games that were conceptually similar were also researched.

2.1.1 *Geometry dash*

Geometry dash[2] (2013) is a 2D platformer game developed by Swedish developer Robert Topala (Illustration *2*). In the game, the player has to avoid numerous obstacles to reach the end of the level. The gameplay mechanics are simple, making the player constantly move

[2] https://geometrydash.io/

forward at a certain pace, and in the event of an obstacle appearing, the player has to jump to avoid it.

Even though the gameplay mechanics used were simple, the levels can get quite complex. The combination of different obstacles is almost endless and leads to challenging situations within the game. *Geometry dash* also utilizes speed in the game; slowing or increasing the speed increaseing the number of possible scenarios. The players need good timing to be successful at clearing a level.

This level of varying complexity and an abundance of possible obstacle combinations is something that *Golfotron* also aspires to accomplish. *Golfotron* draws inspiration from the level design and looks to improve on it, with additional gameplay mechanics leading to more skillful and complex gameplay.

Illustration 2. *Geometry Dash*[2]

2.1.2 *Golf with your friends*

Golf with your friends[3] is a golf video game made by the Australian developer Blacklight (Illustration *3*). In the game, the player has to clear various golf courses or "levels" within a specific time limit, which must be done by launching the golf ball in the cup at the end. The levels include a variety of obstacles to hinder the player's ability to reach the cup. The game can be played in multiplayer mode with other people, making it a contest to get to the cup first.

As in the name, *Golfotron* takes inspiration from golf games overall - to launch an object, navigate around obstacles, and reach the endpoint. *Golf with your friends*' time limit is something that makes it most similar to *Golfotron* out of all other golf games. *Golfotron* looks

[3] https://store.steampowered.com/app/431240/Golf_With_Your_Friends/

to improve the ball launching mechanic by supplementing it with an additional mechanic to redirect the ball mid-flight, leading to a greater variety of gameplay.

Illustration 3. *Golf with your friends*[3]

2.2 Game Mechanics

Every game has a core mechanic that provides the player with an interface to interact with the game. A core mechanic can be a single action within a game or a collection of activities. It is essential for the game designer to clearly define and focus on the game's core mechanic, as often, if a game fails to be fun, the game's core mechanic is to blame [3]. The core mechanic of *Golfotron* is a combination of the following game mechanics: throwing the disk, catching and redirecting the disk, navigating around obstacles, and doing the aforementioned fast. During the development of *Golfotron*, the described game mechanics were extensively focused on providing the player with a fun experience.

Illustration 4. End of level

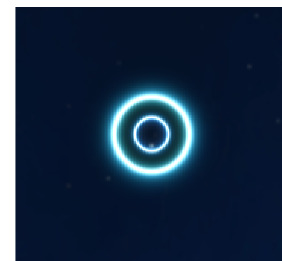

Illustration 5. Disk of player character

Golfotron consists of levels. Each level has a beginning and an end (Illustration 4), and obstacles are between them. Each level has a time limit; the player must reach the end before the time runs out otherwise, clearing the level is unsuccessful. In-game movement is done by throwing the player object characterized by a disk (Illustration 5). The disk can also be caught and redirected at any point after the start of the game. Redirecting can be done a limited number of times.

Clearing the level is deemed unsuccessful, when the player runs out of time, disk redirects or perishes. There are various obstacles in each level, some of them slow the player down, and

some cause the player to perish, which means clearing the level is unsuccessful. If the player reaches the end within the time limit, the level is cleared successfully, in which case a score is calculated for the completed level.

As described before, *Golfotron* aims to have a unique combination of gameplay mechanics that complement the game's fast-paced nature and skillful play and offer one of a kind gameplay experience. This aforementioned experience was achieved with the game mechanics explained in the following subchapters.

2.2.1 Throwing the disk

The primary way the player interacts with the game is by throwing a disk in a chosen direction with the desired speed. The throwing action can be started by clicking on the disk, holding the mouse down, and dragging the cursor in the opposite direction of where the disk will be thrown. The throw direction is opposite of the cursor because it adds some extra difficulty, feels more intuitive, and doesn't obstruct the player's vision when traversing through the level. A line is rendered to give the player visual feedback of the started throwing action and inform the player of the direction they are currently aiming at (Illustration 6).

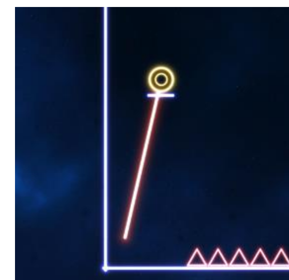

Illustration 6. Aiming the disk

The speed of the throw is determined by how far the mouse cursor has been dragged from the disk. The farther the mouse is from the disk, the greater the speed, which can go up to a specific limit. Visual feedback (Illustration 7) is given; the rendered line changes color based on the rate the disk will be thrown. Once the mouse button is released, the disk will be thrown with the chosen speed and direction. While the disk is moving, it's affected by drag, so over time, it loses speed until coming to a complete stop and mimicking the physics of real life. Furthermore, the disk is affected by gravity, so as it loses force over time, it will start falling towards the ground, again mimicking the physics of real life. Adding gravitational pull is vital because otherwise, at one point, the disk would just hover in the "air," giving the game an unintuitive feel.

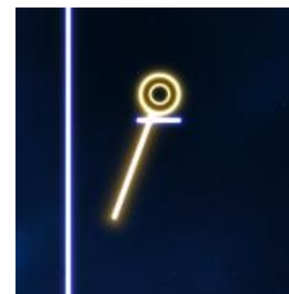

Illustration 7. Line indicating medium speed

The main philosophy followed during the design of this mechanic was for it to feel as intuitive as possible for the player, as it is the action that is used the most.

2.2.2 Catching the disk and redirecting it

After the disk is thrown, it can be caught by clicking on it and holding down the click button. This, in turn, activates the disk throwing mechanic, where the player can redirect the disk in chosen direction with the desired speed. While the disk has been caught and the player calculates how to throw it again, its position is frozen. During that time, the disk is not affected by gravity and will not move until the mouse click has been released and the disk is thrown. This action can be done a limited amount of times during a level because otherwise, the player would just throw the disk at a shallow speed, catch it and repeat. Since *Golfotron* aims to be a fast-paced game, this is not desired from the player.

This unique mechanic of catching the disk and throwing it again provides the player with a wide variety of possibilities on how to traverse a level. This mechanic has many functionalities: clearing the level faster, traversing the level slower while minimizing risk, navigating around obstacles, or avoiding certain death by catching the disk millimeters away from deadly spikes (Illustration 8). It also opens up new ways to design levels exponentially. For example, all the walls could be covered with spikes, and colliding with them would result in a game over. Still, as the player can catch the disk, he can carefully navigate through the obstacles and eventually reach the end of the level.

Illustration 8. Thrown disk caught and being redirected

Catching and redirecting the disk makes the game more skillful since catching the disk requires accurate clicks and good eye-hand coordination, especially at high speeds. Furthermore, it allows for more complex levels to be designed since it provides the player with the tools to navigate around any obstacles. Finally, it plays well with the fact that there is limited time to clear a level and many obstacles; navigating around them swiftly is crucial to a level.

The goal while designing this mechanic was for it to feel responsive and not frustrating for the player. For this reason, the hitbox[4] to start redirecting the disk was made slightly larger than the visual representation of the disk (Illustration 9). It proved to improve the gaming experience for the player while maintaining a good level of difficulty.

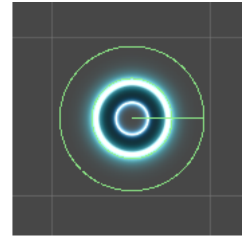

Illustration 9. Hitbox of disk

2.3 Level Design

Rudolf Kremes' statement best describes the level design of *Golfotron*: "good level design teaches the player how to play and enjoy the game" [4, p. 26]. This means that a level should encourage the player to make use of the game's gameplay mechanics to clear the level in a way that produces the player an enjoyable experience. The described philosophy is used in all of Golfotrons' levels. It is accomplished by designing a level around *Golfotrons'* core gameplay mechanics: throwing the disk, redirecting the disk, obstacles, and time limit, as explained in chapter 2.2.

One example of those mentioned above would be *Golfotrons* level 3; in Illustration 10, it is seen that the placement of obstacles makes it extremely difficult for the player to reach the end by only throwing the disk waiting for it to stop and throwing it again within the time limit. Additionally, the blue spiked objects are in constant movement and need to be navigated around. So the player is encouraged to make use of the catch and redirect mechanic to avoid obstacles and reach the end, which in turn teaches the player how to play the game. In the testing of *Golfotron,* it was apparent that the players enjoyed that experience.

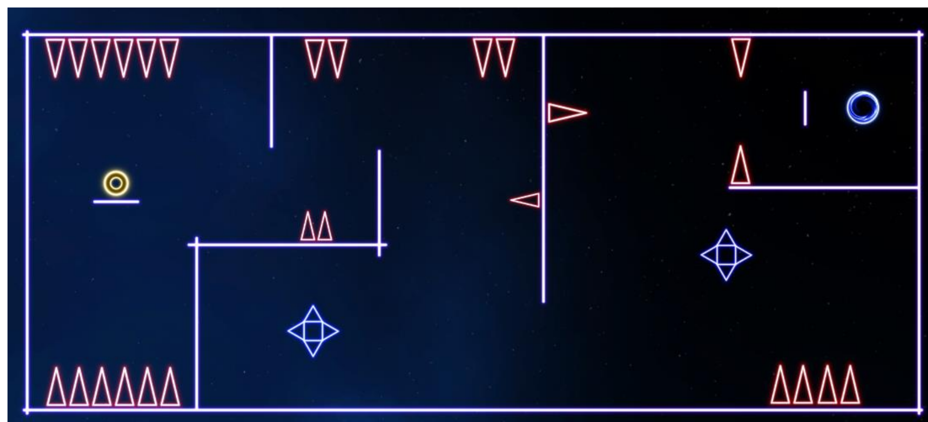

Illustration 10. Level 3 of *Golfotron*

---

[4] A hitbox is an invisible shape used for collision detection
https://en.wiktionary.org/wiki/hitbox

3. Implementation

One of the more critical parts of game development is implementation. An excellent technical implementation of a game has many benefits. It provides better performance, which leads to a better game experience for the player. Additionally, a well-thought-out architecture provides an interface for an easier and faster development process for the game as it scales in size with time [5].

To achieve the previously described benefits in *Golfotron*, various programming patterns were used that are considered by many to be best practices in combination with some object-oriented programming principles to create solid code structure. The following subchapters describe the Unity[5] game engine and code structures for the principal systems of *Golfotron*.

3.1 Unity Game engine

For *Golfotron*, an essential piece of technology was the game engine. The game engine is a software framework that provides the necessary functionality for game design: rendering graphics, physics simulation, detecting a collision, and scripting[6]. The game engine is vital for the game designer because the mentioned functionalities take a significant amount of time to implement and are very difficult to do, so reaching the stage of designing a game is hard when so much groundwork has to be done. For this reason, it is often better to use an existing game engine that has all the core technologies already implemented. Thus, the game designer can focus on the game design and creating a good player experience rather than on the game engine.

The game engine for *Golfotron* is *Unity*, and I chose it over its competitors for four main reasons. First, it supports over 25 different platforms, from Android to MacOS, which is very important from a business standpoint[7]. The potential players of *Golfotron* have diverse preferences for platforms on which to play games; making the game available on as many platforms as achievable helps the game designer reach as many people as possible. Second, *Unity* has a great deal of the best tools for creating 2-dimensional games, and there is a large community around it providing support and tips to new and old developers. Third, the C# programming language can be used to develop a game in *Unity*, which is fast and object-

---

[5] https://unity.com/
[6] https://en.wikipedia.org/wiki/Game_engine
[7] https://www.gamesindustry.biz/articles/2020-01-16-what-is-the-best-game-engine-is-unity-the-right-game-engine-for-you

oriented, preferred in game development. Fourth, *Unity* has various built-in tools that are essential to game development. For example, *Golfotron* relied heavily on its Shader Graph tool, which lets the developer create shaders while getting a visual preview of every step. Furthermore, *Unity*'s particle system and animation tools were fundamental to *Golfotron* as they were easy to use and saved precious development time.

3.1.1 Scripting in Unity

Since *Golfotron* was developed using *Unity*, it's essential to define a "scene" in Unity and how a game developer can create custom game logic within it. A scene[8] in Unity is an asset that contains all or a part of a game, and there can be multiple scenes that can work independently or dependently of each other. As presented in Illustration 11 a scene in Unity consists of game objects, and game objects consist of components[9]. A game object's behavior is defined by the components attached to it; for example, a camera game object that displays the game to the players' screen is a game object that has a camera component attached to it. In order to control a game object and the behavior of its components via a script, a new component needs to be created for it; this can be done by creating a script that derives from the `MonoBehaviour`[10] built-in class and attaching it to a game object[11]. The created script can now interact with other game objects and their components in the scene using the derived methods of `MonoBehaviour`, which in turn provides the developer with the necessary interface to create game logic.

[8] https://docs.unity3d.com/Manual/CreatingScenes.html
[9] https://courses.cs.ut.ee/LTAT.02.018/2021_fall/uploads/Main/UnitySlides
[10] Monobehaviour is the base class from which every Unity script derives. https://docs.unity3d.com/ScriptReference/MonoBehaviour.html
[11] https://docs.unity3d.com/Manual/CreatingAndUsingScripts.html

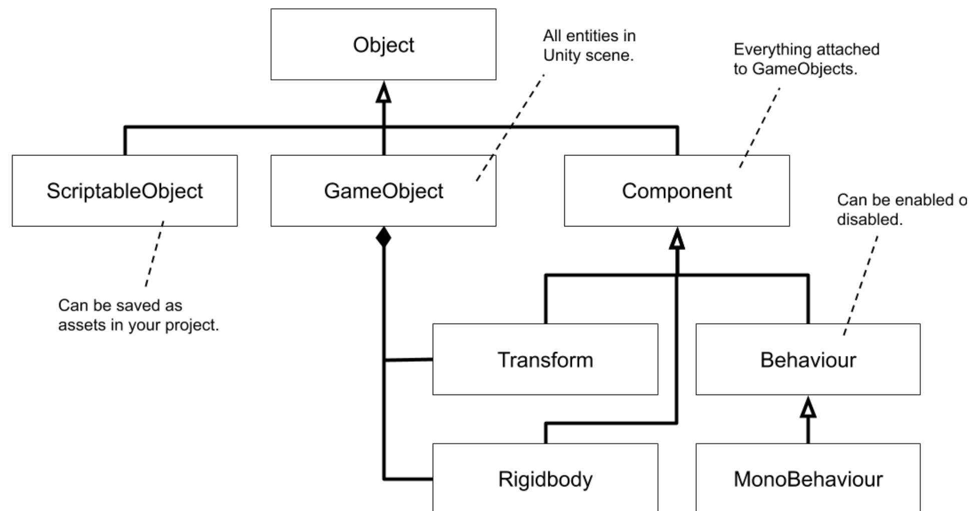

Illustration 11. Diagram[9] showing the structure of a `GameObject`

3.2 Overall game flow

The overall game flow of *Golfotron* describes the system in which the player can navigate through its main menu and levels. It's based on Unity's scene system, in which every scene has a separate game loop - for every level, there is a scene with the addition of the main menu, which is also a scene. Every scene in *Golfotron* transitions to another scene, with the start scene being the main menu. The player can choose any of the levels from the main menu or quit the game. As shown in Illustration 12, in every level, the player can either go to the menu, restart the level or go to the next level, except for the last level, which directs the player back to the main menu. The created system is foolproof because the player is never stuck in one scene and can always move to another. Also, only one scene is active at a time, so it's like a finite state machine that is also scalable as the code doesn't need any significant rewriting on adding new levels. Additionally, if multiple levels use the same component, its properties can be customized through Unity's editor to fit the required level without creating new components.

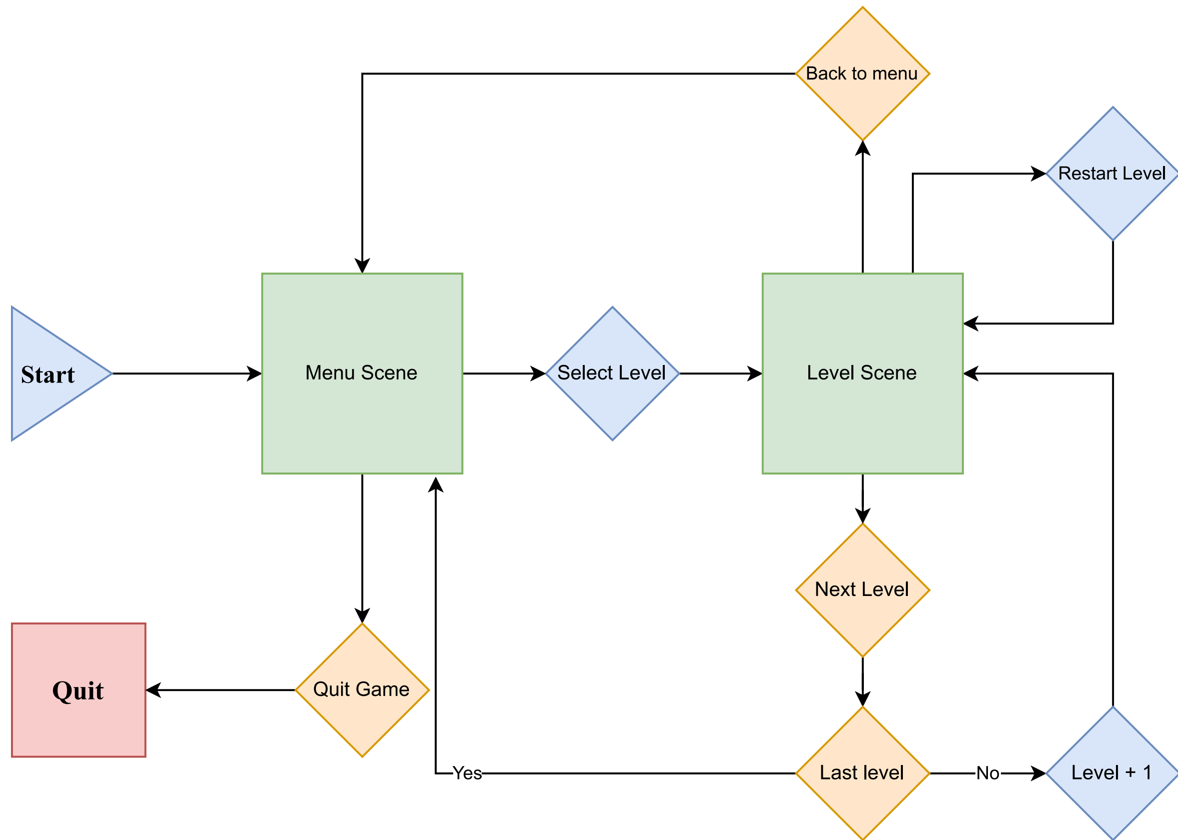

Illustration 12. Activity diagram of the overall game flow

3.3 Structure of the core game loop

The core game loop in *Golfotron* consists of two main structures: a level and the player game object within it, the implementations of these structures are explained in the following subchapters.

3.3.1 Structure of a level

The structure of a level in *Golfotron* is defined by the game objects within it. The main game objects that regulate the flow of a level are `GameManager`, `GameOverManager`, `TimerController`, and the player. The `GameManager` handles the initializing of level-specific properties like time and redirects given and handling game lost and game won events and delegating them to the `GameOverManager`. The `GameOverManager` is responsible for displaying the game over overlay, calculating the score of the level, and handling transitions between scenes. The `TimerController` tracks and displays the time of the level. All three of the game objects mentioned above are singletons, as there should only exist one instance of

them on a level. The player game object handles logic regarding the player character in the game, including user input, physics, collision, and rendering effects.

The `GameManager` and `TimerController` interact with the player game object by the observer pattern. The observer pattern is software design pattern that defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically[12]. The observer pattern is useful to decouple code which is recommended in game development [5]. In *Golfotron,* the observer pattern is implemented with the C# built-in event system; this is preferred over custom implementation to avoid boilerplate code.

The interactions of the game objects via the observer pattern are described as follows: player notifies the `GameManager` of the game started, game lost and game won events. Additionally, the `TimerController` notifies the player of the time ran out event which is handled and then delegated to the `GameManager` as a game lost event. The described interaction via observers could be useful, for example, if there were two player game objects in a level because instead of coupling them to the `GameManager`, it can keep track of both player game objects within its own logic through the received events and decide whether a level is lost or won.

3.3.2 Structure of a player game object

An important part of a level is the player game object, which has a limited amount of states in which it can be in: not thrown, being thrown, redirected, released, and game over, each of which implement different behavior. Additionally, the player game object must only be in one state at a time since it is not logical that the player object can be thrown and not thrown simultaneously. Considering the aforementioned, the programming pattern that matches the player object's movement structure the best is the state programming pattern.

A state programming pattern imitates a finite state machine from the automata theory[13]. Putting the theory into the context of *Golfotron*, the theory's main principles according to Robert Nystrom [5] are that there is a fixed set of states, and the player object can be in one state at a

[12] https://www.vogella.com/tutorials/DesignPatternObserver/article.html
[13] https://en.wikipedia.org/wiki/Finite-state_machine

time; the player object receives inputs or events. Furthermore, according to the theory, each state has a set of transitions that lead to other states triggered by the mentioned inputs or events.

The implementation of the state pattern for *Golfotron* is shown in Illustration 13. The `PlayerContext` class derives from *Unity*'s `MonoBehaviour` class as described in chapter 3.1.1., using its lifecycle events and functions necessary for scripting game logic. The `PlayerContext` class holds information related to the player game object, possible states, and a reference to the active state. On a lifecycle event, the `PlayerContext` class delegates it to the active state through `PlayerBaseState` class.

The `PlayerBaseState` is an abstract class that defines the valid methods for the concrete state classes. It also contains a reference to the context for the different concrete states to alter the context and provide a way for the concrete state classes to transition between each other. The concrete state classes' behavior and transitions (Illustration 14) vary depending on each concrete state class, as shown in Illustration 13.

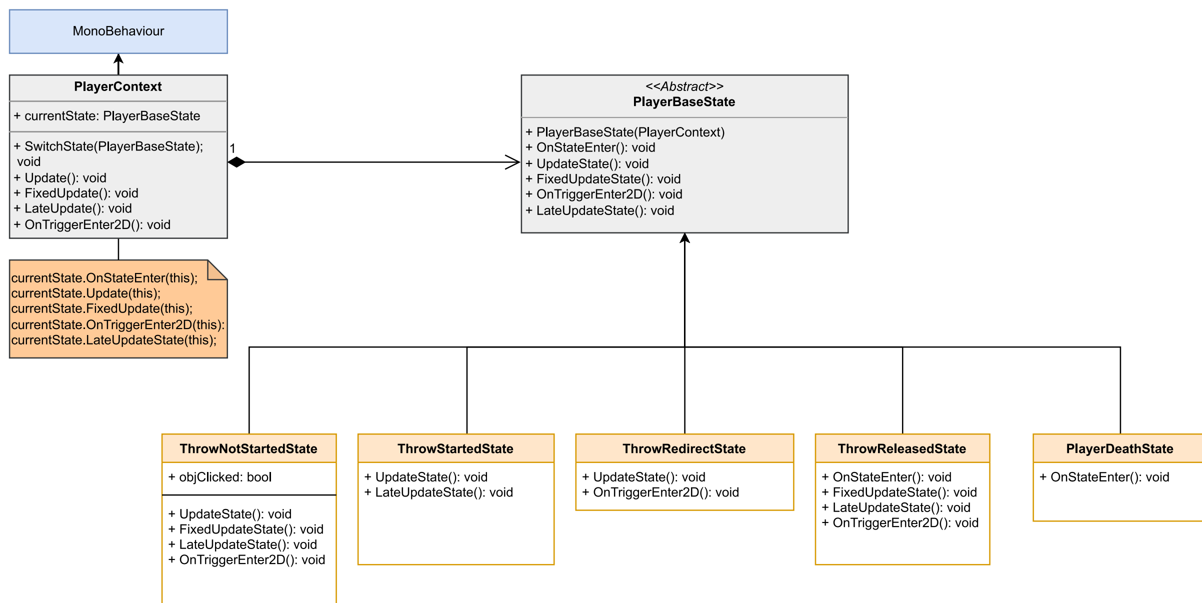

Illustration 13. UML diagram of the implemented State pattern

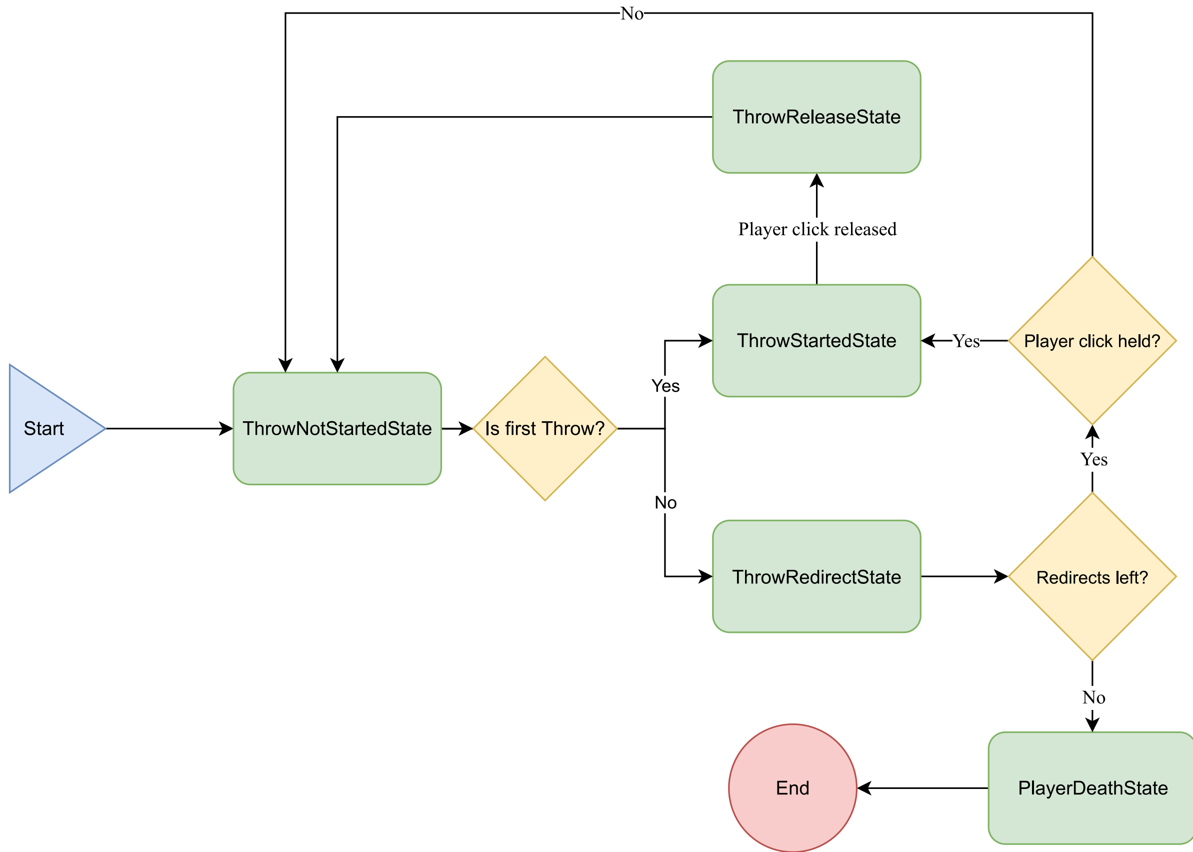

Illustration 14. State diagram depicting transitions between state

The result of the implemented state pattern is that the player game object's behavior changes depending on the active state, and states' behaviors are defined independently of each other. The created structure for the player game object had three main benefits for *Golfotron*. First, debugging *Golfotron* was easier since it could be immediately seen - which state contained problematic logic because only one state could be active at a time. Second, it reduced the dependency on messy conditional logic significantly, which in turn made the code more readable, understandable, and less prone to bugs. Third, it made the code more scalable since state-specific behavior was defined independently; adding new states wouldn't affect the behavior of already existing states.

4. Visual design

The book *The Art of Game Design – A Book of Lenses* [6] states: "But we must always remember that we are not designing just game mechanics, but an entire experience." Thus appearance is a vital part of any game – it can immerse the player within the game, supplement the game mechanics, or lure a player into playing through the beauty of the aesthetics.

Subchapter 4.1 defines the distinct visual style of Golfotron and illustrates the inspirations from which the visual design of Golfotron is derived. Subchapter 4.2 describes the technical implementation of the described visual style.

4.1 Visual style of *Golfotron*

Golfotron's visual style is based on various primitive shapes and combinations that emit a neon lighting effect. Neon lighting is characterized by a bright white center that emits a hue of bright color. One example of an object with neon lighting would be the lightsaber from the cult-classic movie *Star wars* [7] (Illustration 15). The neon effect works best on simple lines or, in the case of shapes, their outlines, as on a solid body, the white center area is too large and not visually pleasing. Therefore *Golfotron* uses only annular shapes in combination with the neon effect.

Illustration 15. A lightsaber from *Star Wars* [9]

The main reasons for the neon theme in *Golfotron* are that it goes well with the fast-paced nature of the game. Since *Golfotron*'s levels are designed to be modular, having primitive shapes that can be combined and applied to the neon effect creates the possibility of coming up with unique visuals that don't have to be made separately but with a combination of already existing visuals (Illustration 16). Making the creator's job easier and the game more varied.

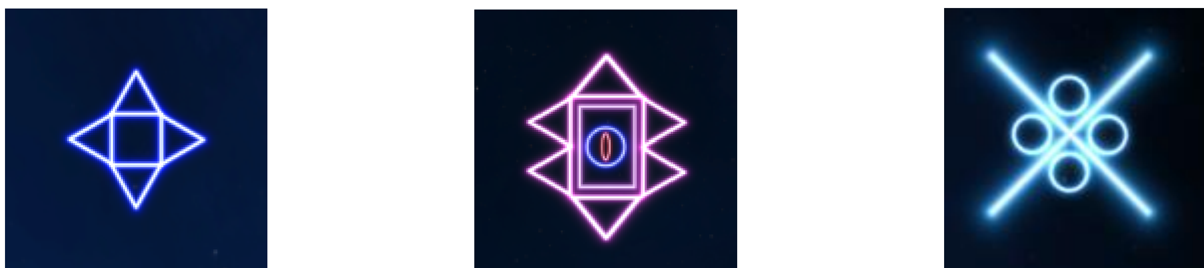

Illustration 16. Different neon shapes

4.1.1 Visual design inspiration

The game's visual design inspiration comes from two primary sources. The first source of inspiration is the 2010 released movie *Tron: Legacy*[14] (**Error! Reference source not found.**), a sequel to the American cult classic *Tron*[15] (1982). *Tron: Legacy* is a sci-fi movie about venturing into a digital world, containing enticing shape outlines with a neon color palette drawn on a dark background. This contrast creates a dark futuristic appearance that is enjoyable to the eye. The neon look supplements the fast movement in the movie, which directly transfers over to the fast-paced nature of *Golfotron*.

The second inspiration comes from the popular 2D platformer *Geometry Dash* (**Error! Reference source not found.**), which contains a variety of simplistic shapes outlined with a distinct neon-like glow. This style creates an enjoyable and modular look. The simplicity and modularity of the figures are also something that *Golfotron* benefits from.

Illustration 17. *Tron: Legacy*[14]

Illustration 18. *Geometry Dash*[2]

4.2 Implementation of visuals

The following subchapters 4.2.1 and 4.2.2 describe how the visual design of *Golfotron* was implemented.

4.2.1 Signed distance fields

Since *Golfotron* aimed to be a modular and customizable game characterized by combinations of various primitive shapes, it was crucial for there to be a way to create said shapes so that their shape and scale would be easily adjustable during the creation of levels or at runtime

[14] https://www.imdb.com/title/tt1104001/mediaviewer/rm1565857792/
[15] https://www.imdb.com/title/tt0084827/

without losing any quality and this would be done quick without making a new texture for every object with a different shape or scale. The solution was to use signed distance fields to generate the required shapes procedurally.

A distance field is a scalar field that specifies the minimum distance to a shape. If the distance is signed it's a signed distance field, this is done to distinguish between inside and outside of the shape[16].

In the context of this thesis, signed distance fields were used to create procedural textures. For creating the procedural textures a fragement's UV was sampled and it's texture coordinates were used as `input p` for the signed distance field function. The method for defining the shape and calculating the `input p`-s distance from it's closest point varies depending on the shape. The methods to calculate the SDF[17]-s of different primitive shapes were made by Inigo Quilez[18]. The calculated distances directly correlate to grayscale color, from 0 to 1 or from black to white, values below 0 are still black and values above 1 are still white. So the closer a point is to the shape (isosurface) the darker its color is (Illustration 19). Since we're dealing with a signed distance field, not a distance field, we can take the absolute value of every pixel and get an annular texture, this only affects the pixels inside the shape since their distance from the shape is negative (Illustration 20). With a regular distance field, this would not be possible. The calculated distances as colors are multiplied with `-1` in order to invert the colors since neon is white in the center of the shape and the result is passed to the neon effect sub shader. (See subchapters 4.2.2)

---

[16] https://www.ics.uci.edu/~gopi/ICS280Win02/ADF.pdf

[17] SDF – Signed distance field

[18] https://www.iquilezles.org/www/articles/distfunctions2d/distfunctions2d.htm

Distance zero means pixel in on shape

1.2	1.2	1.2	1.2	1.2	1.2	1.2	1.2	1.2	1.2	1.2
1.2	1.2	0.894	0.8	0.8	0.8	0.8	0.8	0.894	1.2	1.2
1.2	0.894	0.565	0.4	0.4	0.4	0.4	0.4	0.565	0.894	1.2
1.2	0.8	0.4	0	0	0	0	0	0.4	0.8	1.2
1.2	0.8	0.4	0	-0.4	-0.4	-0.4	0	0.4	0.8	1.2
1.2	0.8	0.4	0	-0.4	-0.8	-0.4	0	0.4	0.8	1.2
1.2	0.8	0.4	0	-0.4	-0.4	-0.4	0	0.4	0.8	1.2
1.2	0.8	0.4	0	0	0	0	0	0.4	0.8	1.2
1.2	0.894	0.565	0.4	0.4	0.4	0.4	0.4	0.565	0.894	1.2
1.2	1.2	0.894	0.8	0.8	0.8	0.8	0.8	0.894	1.2	1.2
1.2	1.2	1.2	1.2	1.2	1.2	1.2	1.2	1.2	1.2	1.2

Negative distance means pixel is inside the shape

Distances directly correlate to grayscale color

Illustration 19. Square SDF example

Distance zero means pixel in on shape

1.2	1.2	1.2	1.2	1.2	1.2	1.2	1.2	1.2	1.2	1.2
1.2	1.2	0.894	0.8	0.8	0.8	0.8	0.8	0.894	1.2	1.2
1.2	0.894	0.565	0.4	0.4	0.4	0.4	0.4	0.565	0.894	1.2
1.2	0.8	0.4	0	0	0	0	0	0.4	0.8	1.2
1.2	0.8	0.4	0	0.4	0.4	0.4	0	0.4	0.8	1.2
1.2	0.8	0.4	0	0.4	0.8	0.4	0	0.4	0.8	1.2
1.2	0.8	0.4	0	0.4	0.4	0.4	0	0.4	0.8	1.2
1.2	0.8	0.4	0	0	0	0	0	0.4	0.8	1.2
1.2	0.894	0.565	0.4	0.4	0.4	0.4	0.4	0.565	0.894	1.2
1.2	1.2	0.894	0.8	0.8	0.8	0.8	0.8	0.894	1.2	1.2
1.2	1.2	1.2	1.2	1.2	1.2	1.2	1.2	1.2	1.2	1.2

After taking the absolute value of distances
the shape is annular on the inside

Distances directly correlate to grayscale color

Illustration 20. Annular square SDF example

Since textures based on signed distance fields are procedurally generated, it is possible to rescale them as much as needed without a loss in quality. It's also possible to change their shape or rotation at runtime without any impact on quality.

4.2.2 Neon effect

The goal was to create high-quality game elements with a neon look. The formed elements varied from different shapes to text and numbers, which could move, fade, etc.

At first, the effect was done with *Adobe Illustrator* using its' raster effects. This approach proved unfruitful - when the created image was exported from *Adobe Illustrator* to png format and imported to *Unity*, the created texture of the elements was low-quality despite the resolution used. Furthermore, different visual artifacts made the result look bad, and the asset did not blend well into the game. I tried different import settings to fix the quality problem, but the achieved result was not up to expectations, even when using mipmaps. It's important to note that the elements' quality was tested at pixel-perfect texture resolutions with the correct camera size.

The second problem was that the textures made in an image editing program were not dynamically editable. When the editor changed the neon texture's color, the texture would not retain the exact composition of the neon effect as it had been initially. Additionally, there was no way to modify the intensity or scale of the neon texture without significant loss in quality.

Since the effects added by image editing software were of insufficient quality and not dynamically editable, the solution to fix this was to use a custom shader to add the neon effect. The created shader works for two inputs: a texture in which only black-white gradient color is used or a shape represented by a signed distance function (Illustration 21). The neon effect shader used is a modified version of Arthur Deleynes' shader [8].

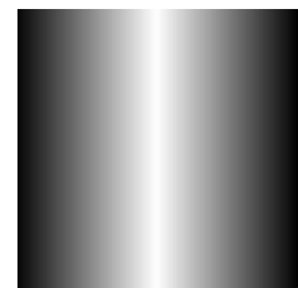

Illustration 21. An inverted line SDF

The shader to apply the neon effect was unlit, which isn't affected by lighting. The created texture was sampled twice (Illustration 22 and Illustration 23).

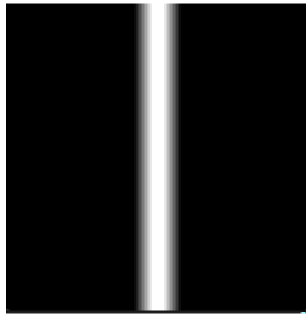

Illustration 22. The heart of the laser

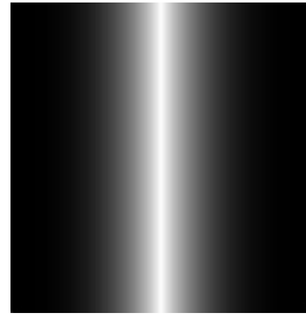

Illustration 23. The aura of the laser

The first sample is for the aura or atmosphere of the neon shape. It is created through a power function with a custom float value as the power; this way, the intensity of the aura can be controlled by the editor (Illustration 24 and Illustration 25). The aura is multiplied with HDR color for the result to work with the post-processing effect bloom (Illustration 26).

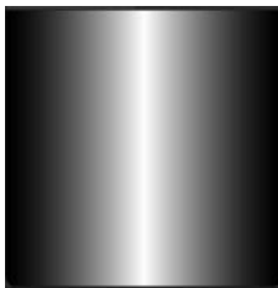

Illustration 24. Aura with a lower intensity

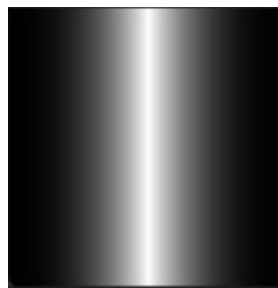

Illustration 25. Aura with a higher intensity

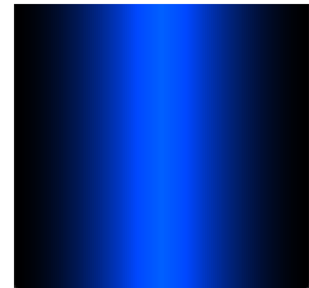

Illustration 26. Aura multiplied with a HDR color

The second sample is for the heart of the neon shape, which is created via *Unity*'s color mask[19] function. The color mask variable defines the color that the person wants to use for the heart of the neon shape from the input. We choose white because neon shapes typically have a lighter center, which visually appears white or almost white (Illustration 27). Further on, we can control the size of the heart by modifying the range variable, which selects the range of values that the mask considers white (Illustration 28). The edges of the shape can be softened to disappear gradually via the fuzziness variable. The fuzziness variable reduces hard edges and

[19] https://docs.unity3d.com/Packages/com.unity.shadergraph@6.9/manual/Color-Mask-Node.html

creates a more believable and enjoyable effect (Illustration 29). The result is then again passed through a power function to either intensify or unintensify the heat.

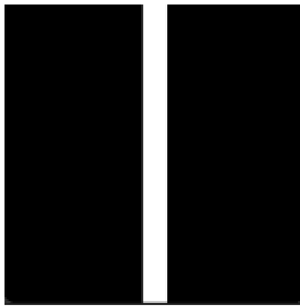

Illustration 27. Range of area considered to be white

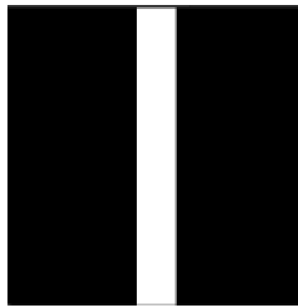

Illustration 28. Higher range of area considered to be white

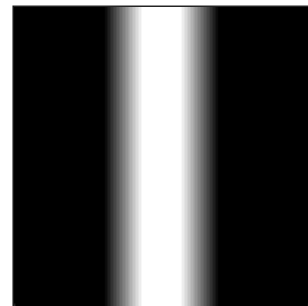

Illustration 29. High fuzziness with high range

In the next step, the aura and the heart of the shape are blended togetherusing *Unity*'s screen blend[20] function. This function inverts the values of the heart and the aura pixels, multiplies them, and then inverts the result again, resulting in a brightening effect (Illustration 30).

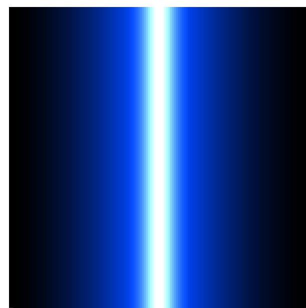

Illustration 30. Shape after blend function

The output from the blend function is directed into the fragment's base color, and the shader is complete. Providing the shader with a gradient texture/SDF, color, range, power, and fuzziness values, creates a believable neon effect unto the texture that looks good in-game and is dynamically customizable even at runtime.

[20] https://docs.unity3d.com/Packages/com.unity.shadergraph@6.9/manual/Blend-Node.html

5. Testing

Testing is important for any kind of software. Subchapter 5.1 explains playtesting and why it's essential for game development. Subchapter 5.2 describes how the testing sessions were conducted, subchapters 5.3 and 5.4 give an overview of the first and second testing iterations, and subchapter 5.5 sums everything up.

5.1 Playtesting and its importance

Playtesting is a form of testing in which a person plays the created game and describes the received experience. It is important to note that playtesting differs from quality assurance testing, which focuses on finding the technical errors of the game, and from usability testing, which focuses on the intuitive usage of the games' systems and control interface. In the following chapter, primarily two sources will be used - Schell J. "The Art of Game Design – A Book of Lenses" [6] and Salen K. and Zimmerman E. "Rules of Play – Game Design Fundamentals" [3] - which help to decipher the theory and set upcoming goals.

Playtesting is vital for the game designer to understand if the designed game experience he is trying to portray is met in reality with the experience an actual player experiences from playing the designed game.

If the playtesting provides negative feedback, it is essential to take it into consideration and make improvements to the game based on it. If a player does not enjoy the game, the game cannot succeed, and the game designer has failed.

Playtesting should be performed multiple times and as early as possible in the game development cycle. Playtesting in the different game development cycles brings out the shortcomings or abundances of the game. When these problems are discovered early in the game development process, these issues can be addressed. Otherwise, they will make it into the final product, in which case large-scale changes need to be made that could have been avoided.

For playtesting, the tester needs to prepare as specific and clear questions as possible for the player. This is important to get as specific and constructive feedback as possible from the player, which the tester can take into consideration and improve the game on it. Some examples

of these questions would be "Was the third level too long?" and "Which level did you enjoy the most?". It is also important for the tester to decide which parts of the game he wants feedback on and concentrate the test on these parts. In addition, the tester can observe different aspects of the gameplay, for example, what strategy the player used to reach the goal or how long clearing a level took.

After a playtest, the game designer needs to collect the players' feedback. An interview or a quiz can do it. Quizzes are a good way to get a quantitative overview of the test results. As mentioned before, the questions should be specific, and they should be appreciable on a numerical scale, which provides quantitative data. It would be helpful to ask for some personal details of the player in the quiz, for example, the gender, age, and favorite video games.

With an interview, the game designer can get more specific feedback from the players than from a quiz by asking clarifying questions when needed. The body language and manner of speech can be observed to understand the opinion of the player better. Before the interview, the game designer needs to prepare the format and questions of the interview and, if possible, record it to review it thoroughly. A good practice is to give the players a chance to add some comments or thoughts about the game.

5.2 Methodology

The testing of *Golfotron* was done in two iterations. Both iterations of testing were done with five testers. The testing took place via an online call due to the COVID-19 pandemic. The testers were asked to share their screen, and the testing sessions were recorded in order to document the sessions and further analyze the tester's approach to the game and strategies used to play.

The first testing iteration was more informal and focused more on getting the parameters of the gameplay mechanics right through the testers' feedback and discovering any significant bugs. The second testing iteration was more formal and focused on all aspects of the game. In both iterations, an interview with the testers took place to get qualitative feedback, and an simple online quiz was provided to them to gather quantitative feedback.

5.3 First test iteration

The first testing iteration focused mainly on playtesting the game mechanics. The testers were provided with five different levels and were asked to try each one and optionally complete them. After the testers were finished with the levels, a short interview took place. In the interview, testers were asked about the feel of different game mechanics, for example, does launching the disk feel intuitive, does the speed of throwing feel right, does the bouncing of the disk off walls seem natural, etc. Additionally, the testers were asked what they thought about the game mechanics and if they would change something. After the interview, the testers were provided with a quiz to rate the game and its mechanics.

5.3.1 Results

From the testers' feedback regarding the game mechanics, multiple facts appeared. First, catching the disk was too tricky and frustrated the player. Secondly, the bouncing of the disk off walls under certain angles felt weird. Thirdly, the time limit and the number of redirects allowed in the levels were too high. The camera's behavior at the start of the game was also inadequate. These problems were taken into consideration and adjusted for the next testing iteration. Also, various bugs and unintended behavior appeared during the testing, which were fixed for the next iteration. The testers suggested the following change to the game: make a hotkey for restarting a level, which was implemented.

From the quiz's quantitative feedback (Illustration 31) on a scale of 1 ("terrible") to 6 ("excellent), it is shown that despite the testers encountering some problems during testing, they rated the overall experience and the potential of the game over the average, which is a good indication that the concept of Golfotron could actually appeal to people in the actual video game market.

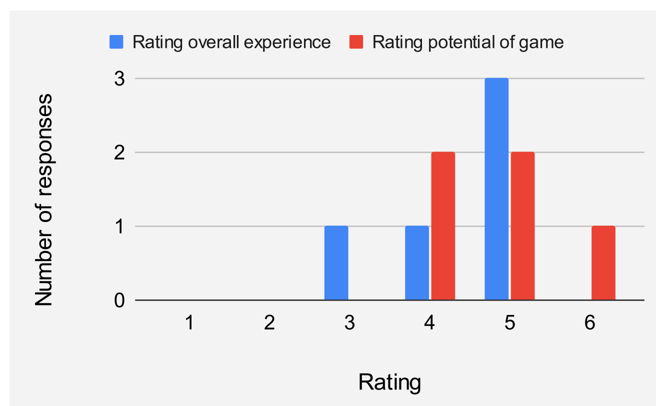

Illustration 31. The rating of Golfotron regarding its potential and overall game experience

5.4 Second test iteration

The second test iteration was similar to the first one but focused on all aspects of the game, like the visuals and level design, not just the gameplay mechanics. After the testing an short interview took place to gather feedback on the different aspects of the game, and a quiz was provided.

5.4.1 Results

In the second test iteration, the testers had three primary criticisms towards *Golfotron*. First, they felt that the level design of some levels was not adequate and should be improved upon. Second, they expressed that the game should have more types of obstacles and interactions; otherwise, the game would become repetitive. Third, the camera following the disk would at times act clunky. As the game's developer, I accept these criticisms and believe them to be valid. The described problems were not entirely fixed within the scope of this thesis but will be in the future.

Design-wise the aspect that *Golfotron* focused most on was its disk throwing mechanic and the unique disk redirecting mechanic. From the second iteration's questionnaire feedback (Illustration 32) on a scale of 1 ("terrible") to 6 ("excellent"), it can be seen that the mentioned game mechanics are liked by the testers, which shows the concept and implementation of them was adequate.

Illustration 32. The ratings of *Golfotrons'* disk throw and redirect mechanics

6. Conclusion

As a result of this thesis, a 2D action game *Golfotron* was developed. The goal was to create a game that provides the player with an enjoyable experience through its unique game mechanics and visual design. Similar games were researched, and playtesting was done by the end of the development process.

The development of *Golfotron* consisted of two parts: game logic implementation and visual implementation. Programming patterns were used to create a scalable and structured game logic system to ease the process of future development. The main programming patterns used were the state and observer patterns. The visual implementation was done with signed distance fields and shader programming in order to create textures that are adjustable at runtime and maintain good quality while doing so.

The games' testing was divided into two iterations; there were five testers in both. During the testing, multiple problems were discovered regarding the game, some of which were fixed. Additionally, feedback from the tests were gathered, and it showed that the gameplay mechanics, overall experience, and the concept of the game were rated highly. Despite the positive feedback, the testers' also indicated that the game still has some significant shortcomings, which will be improved upon in the future.

This was my first time developing a game alone, and it was a very teaching experience. I learned a lot throughout the process and became interested in the field of computer graphics. I think that the goal of this thesis was achieved, and I will develop this game further to one day release it into the video game market.

Sincere gratitude goes to the supervisor Mark Muhhin, whose feedback and suggestions greatly improved the quality of this thesis. Special thanks go to all of the testers of *Golfotron* and Raimond-Hendrik Tunnel for giving the Bachelor's Thesis Seminar course and helpful advice regarding the thesis.

References

[1] S. Rogers, *Level Up! The Guide to Great Video Game Design*, 2nd ed. England, Chichester: John Wiley and Sons, Ltd., 2014.

[2] T. Fullerton, *Game Design Workshop: A Playcentric Approach to Creating Innovative Games*, 3rd ed. United States of America, Boca Raton: CRC Press/Taylor & Francis, 2014.

[3] K. Salen and E. Zimmerman, *Rules of Play - Game Design Fundamentals*. England, Massachusetts London: The MIT Press Cambridge, 2004.

[4] R. Kremers, *Level Design: Concept, Theory, and Practice*. United States of America, Hoboken: CRC Press, 2009.

[5] R. Nystrom, *Game Programming Patterns*. Genever Benning, 2014.

[6] J. Schell, *The Art of Game Design – A Book of Lenses*. United States of America, Burlington: Morgan Kaufmann Publishers, 2008.

[7] "Star Wars (1977) - IMDb." https://www.imdb.com/title/tt0076759/ (accessed May 10, 2022).

[8] A. Deleye, "ArtStation - Simple 2D Laser Shader with Unity ShaderGraph." https://www.artstation.com/blogs/arthurdeleye/XKBN/simple-shader-de-laser-2d-avec-unity-shadergraph (accessed May 08, 2022).

[9] "Star Wars lightsaber pulled from auction over authenticity issue | Star Wars | The Guardian." https://www.theguardian.com/film/2018/dec/11/star-wars-lightsaber-pulled-auction-authenticity-luke-skywalker (accessed May 10, 2022).

Appendix

I. Launch Guide

Instructions to play *Golfotron*:

1. Extract archive "AccompanyingFiles.zip".
2. Open directory "/Build".
3. Run file "Golfotron.exe".
4. Press Start Game and choose a level.

II. Accompanying files

- /Build – the folder containing the build of *Golfotron* with other files needed to run it.
- demo.mkv – a video showing the gameplay of *Golfotron*

III. Assets

Table 1. Assets from the internet

Asset	Author	Source
Nebula Blue background	DinV Studio	https://assetstore.unity.com/packages/2d/textures-materials/dynamic-space-background-lite-104606#description
Nebula Red background	DinV Studio	https://assetstore.unity.com/packages/2d/textures-materials/dynamic-space-background-lite-104606#description
Nebula Aqua-Pink Background	DinV Studio	https://assetstore.unity.com/packages/2d/textures-materials/dynamic-space-background-lite-104606#description
Stars small	DinV Studio	https://assetstore.unity.com/packages/2d/textures-materials/dynamic-space-background-lite-104606#description
Stars big	DinV Studio	https://assetstore.unity.com/packages/2d/textures-materials/dynamic-space-background-lite-104606#description

IV. Source code

The source code and assets of *Golfotron* can be found at:

https://bitbucket.org/pelleholdenporila/golfotron/src/master/

V. License

Non-exclusive licence to reproduce the thesis and make the thesis public

I, **Pelle Holden Porila,**

(author's name)

1. grant the University of Tartu a free permit (non-exclusive licence) to reproduce, for the purpose of preservation, including for adding to the DSpace digital archives until the expiry of the term of copyright, my thesis

Golfotron, A fast-paced 2D game,

(title of thesis)

supervised by Mark Muhhin.

(supervisor's name)

2. I grant the University of Tartu a permit to make the thesis specified in point 1 available to the public via the web environment of the University of Tartu, including via the DSpace digital archives, under the Creative Commons licence CC BY NC ND 4.0, which allows, by giving appropriate credit to the author, to reproduce, distribute the work and communicate it to the public, and prohibits the creation of derivative works and any commercial use of the work until the expiry of the term of copyright.
3. I am aware of the fact that the author retains the rights specified in points 1 and 2.
4. I confirm that granting the non-exclusive licence does not infringe other persons' intellectual property rights or rights arising from the personal data protection legislation.

Pelle Holden Porila

08/05/2022