University of Hradec Králové
Faculty of Informatics and Management
Department of Information Technologies

Multiplatform game development using the Flutter framework

Bachelor Thesis

Author: David Domkář
Branch of study: Applied Informatics

Advisor: Ing. Milan Kořínek

Hradec Králové April 2024

Declaration:

I declare I wrote the Bachelor Thesis „Multiplatform game development using the Flutter framework“ myself, using only the listed bibliography.

The research was done under the support and guidance of Ing. Milan Kořínek.

In Hradec Králové, 19.4.2024 David Domkář

Acknowledgments:
I want to thank my thesis supervisor Ing. Milan Kořínek, for his professional support and guidance during my work on the thesis.

Annotation

Bachelor thesis explores possibilities of multiplatform game development using the Flutter framework as an alternative to other game development tools, such as game engines.

The thesis is divided into theoretical and practical parts. The theoretical part describes the Flutter framework and its ecosystem. In the practical part, the thesis focuses on building a simple example game using the Flutter framework and introduces various game development aspects such as rendering, input processing or physics in the context of the Flutter framework.

The thesis also answers research questions regarding multiplatform game development using the Flutter framework with the results obtained from both parts of the thesis.

Key words: flutter, framework, game development, application, engine, multiplatform

Anotace

Název: Multiplatformní vývoj her ve frameworku Flutter

Bakalářská práce zkoumá možnosti multiplatformního vývoje her s využitím frameworku Flutter jako alternativu k běžným nástrojům pro tvorbu her, jako jsou herní enginy.

Práce je rozdělena na teoretickou a praktickou část. V teoretické části popisuje framework Flutter a jeho ekosystém. V praktické části se práce zaměřuje na vývoj jednoduché hry s využitím frameworku Flutter a předvádí různé aspekty specifické pro vývoj her v kontextu frameworku Flutter.

Práce také zodpovídá výzkumné otázky ohledně multiplatformního vývoje her s využitím frameworku Flutter na základě výsledků z obou částí práce.

Klíčová slova: flutter, framework, vývoj her, aplikace, engine, multiplatformní

Contents

List of Figures

List of Code Snippets

1 Introduction

Developing games for multiple platforms is a complex software engineering problem. Every platform is different. Each has its method of getting input from a user, drawing on a screen with graphics APIs, interacting with other parts of the platform, etc.

Game development tools such as game engines are solving this problem by abstracting away these platform inconsistencies and providing game developers with a set of tools that can be used to develop many kinds of games.

However, game engines are often large, sophisticated pieces of software which come with much functionality that is only needed in some games. Moreover, writing code in these game engines can be a bad developer experience because of the need to recompile and rerun the game when some gameplay code changes. This can be particularly problematic if a game engine of choice uses a scripting language that takes a long time to compile when the project is large. The scripting language can also be slow to run because of the interpreted nature of most scripting languages which can cause problems in games with resource-intensive gameplay code.

When focused on a smaller subset of games, such as UI-heavy games, one can find out that they often use in-engine built-in UI components, which can be slow to work with, and complex UI is challenging to build with them, often requiring complicated supporting code. Smaller, open-source game engines may not even have proper UI support because it is usually not considered a priority.

Ultimately, selecting a game engine is a complicated process and requires a lot of research and knowledge about the game being made. However, some games may not need a game engine at all. They may have many UI elements and little to no interactive parts with simple graphics. There can be a better way to develop these more miniature games than to use a full-blown game engine, especially if these games are also targeting resource-constrained devices, such as mobile phones where battery drain needs to be considered or web browsers where the game is limited in functionality by Web APIs.

Flutter, a multiplatform UI framework developed by Google for building cross-platform apps from a single codebase, could be an interesting alternative for making games without a game engine. It is built from the ground up with its renderer, which uses modern graphics APIs under the hood and gives you control over every pixel on the screen. It supports every major platform and abstracts away the platform code while maintaining easy interoperability if required. This is something many game engines cannot do easily and opens exciting possibilities for games that integrate a lot of third-party libraries dependent on some of the target platforms. Since it is a UI framework, UI developer experience is its primary focus, which is excellent for UI-heavy games.

This thesis explores the possibilities of developing multiplatform games using Flutter. It describes how Flutter can be used to achieve control over rendering, physics, input, and other vital aspects of game development while introducing new ideas unique to the Flutter framework.

2 Objectives

The main goal of this thesis is to explore the possibilities of multiplatform game development using the Flutter framework.

The set of objectives below was determined for the thesis to achieve this goal:

- Describe the Flutter framework and its ecosystem, highlighting its advantages regarding building high performance applications.
- Introduce the main aspects of game development, such as rendering, physics, input, etc., in the context of the Flutter framework.
- Support the thesis by creating a practical example of a simple game developed using the Flutter framework.

These objectives will be used to answer research questions (RQ).

2.1 Research Questions

RQ1: How does Flutter's architecture support the development of multiplatform games?

RQ2: How can game development aspects like input processing, physics simulation, and rendering be leveraged within the Flutter framework?

RQ3: What strategies are employed to manage the layering of UI elements and game objects in Flutter?

RQ4: How does the Flutter framework enable use of special post-processing effects in multiplatform games?

RQ5: What are the deployment options for multiplatform games made using the Flutter framework?

3 Methods

The thesis is divided into theoretical and practical parts. The theoretical part of the thesis uses research to achieve objectives and answer research questions. It explains basic Flutter framework concepts and their inner workings using information from the official Flutter documentation.

The practical part of the thesis achieves objectives and answers research questions by building a simple game using the Flutter framework to support the theoretical part of the thesis with a real-life example. The example is used to introduce various aspects of game development in the context of the Flutter framework and describes how to implement these aspects using the framework's APIs.

4 Theoretical Part

This part of the thesis aims to introduce the Flutter framework and its ecosystem while explaining its features in the context of multiplatform game development.

4.1 Flutter

Mobile applications are part of our everyday lives. While the mobile application market is increasingly competitive, it is a space where many startups and developers focus their efforts.

Over the past years, many technologies have emerged in this space to solve the problem of cross-platform development. In the early days, if an app needed to be developed on both major mobile operating systems – Android and iOS, it had to be built twice using different languages and tooling. On Android, it was Java or the newer Koltin. For iOS, it was Objective-C or the newer Swift. Multiple technologies have been created to allow the development of apps for both platforms from a single codebase. Therefore, saving development time and efficiency. These technologies include PhoneGap, Ionic, Xamarin, Cordova, or React Native [1].

Flutter is trying to solve the same problem while taking a fresh approach to some cross-platform development challenges. It is a free, open-source UI framework for building cross-platform apps developed by Google. It was first shown to the public in 2015 as "Sky" – an experiment for using the Dart language (also developed by Google) on mobile platforms [2]. The first 1.0 version of the framework, now named Flutter, was announced in 2018 [3]. It featured near-native performance compared to similar technologies, mainly thanks to its graphics engine Skia, which also powers Android or Google Chrome. It had a stateful hot reload for fast development and excellent developer experience. And it was shipped with a library of pre-made customizable widgets to build the UI of the apps using the Dart language.

At the time of writing, Flutter is the most popular UI framework, targeting mobile, desktop, and web applications. Many developers around the world use it. In 2022, it surpassed React Native, which up to that year, was the most used cross-

platform framework in developer working hours [1]. A few years ago, it also overtook its rival framework in terms of trends on Stack Overflow [4].

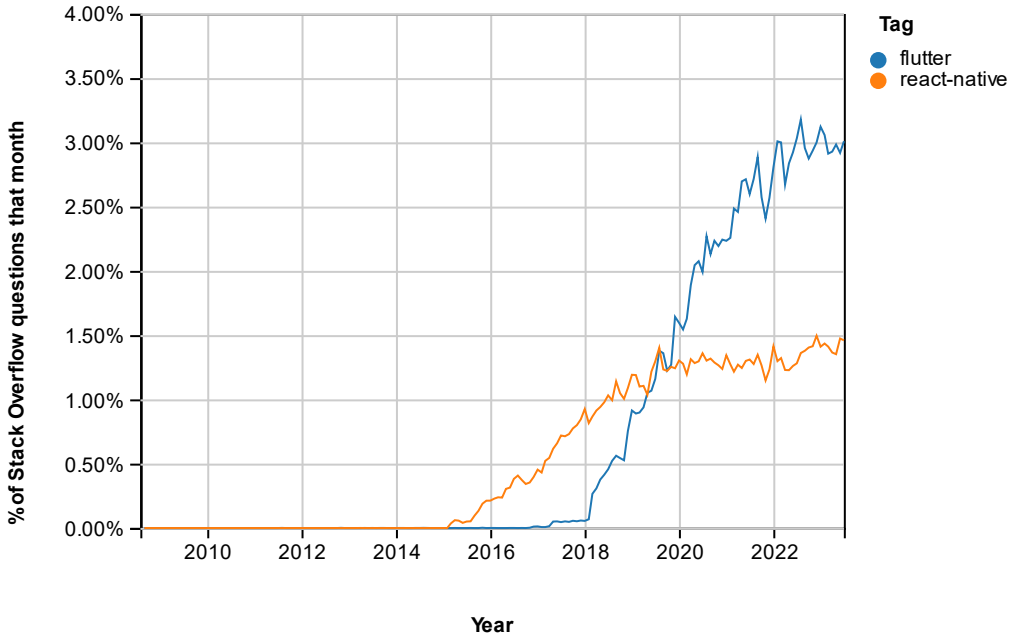

Figure 1: Flutter vs. React Native from Stack Overflow Trends
(source: https://insights.stackoverflow.com/trends?tags=flutter%2Creact-native)

The goal of the Flutter framework is to run on any screen. It can also be used on embedded devices. For example, Armadillo, the system UI for Google's general-purpose operating system Fuchsia, was written using Flutter [5].

4.1.1 Architecture

Flutter framework is a versatile cross-platform UI toolkit that facilitates code reusability across diverse operating systems. Furthermore, it also allows applications to interact with underlying platform services.

In the development phase, Flutter apps run in a virtual machine that offers a stateful hot reload of changes. That way, the app does not need to be recompiled. In the release phase, Flutter apps are compiled straight into CPU instructions for the target platform or JavaScript if targeting the web [6].

Flutter architecture consists of many layers. Each layer depends on the underlying layer; every part of the framework is designed to be optional and replaceable [6]. Specifically, as can be seen in Figure 2, Flutter Architecture is

divided into three main layers – Framework, Engine, and Embedder. This part of the thesis discusses these layers in greater detail.

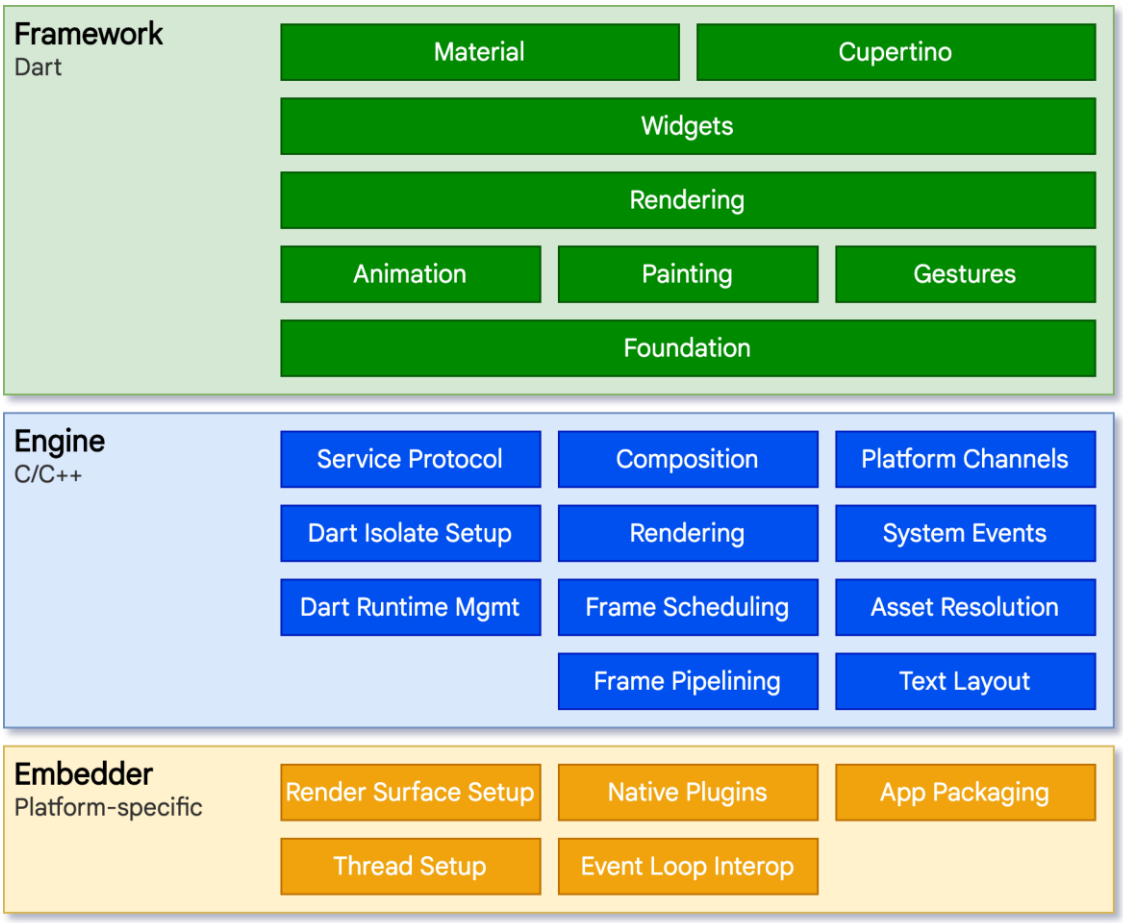

Figure 2: Flutter Architecture Layers
(source: https://docs.flutter.dev/assets/images/docs/arch-overview/archdiagram.png)

Embedder is the only part of the framework that must be implemented for every target platform. It contains the entry point for the application on the target platform, manages the event loop, and has access to services like rendering, accessibility, and input. It is written in a language appropriate to the target platform. Flutter provides embedders for all major operating systems and can also use other third-party embedders [6].

The core layer is the Engine layer. It provides low-level implementations of Flutter core APIs, such as graphics, using the Skia graphics library or newer Impeller. It also facilitates file and network I/O, text layout, accessibility support, plugin architecture, Dart runtime, and compile toolchain. The core layer is mainly written in C++ for high performance and connects with Embedder using the

Embedder API. Currently, the C++ engine implementation is missing from the Flutter web target. Instead, it uses engine implementation written in JavaScript to interface with Web APIs. However, experimental WebAssembly support should perform better than the current JavaScript implementation [6].

The Framework layer is essential for app developers because it is the layer they interact with the most. This layer implements the Flutter framework in the Dart language. It connects to the engine layer using the dart:ui package, which wraps the underlying C++ code in Dart classes for access to lower-level functions of the engine. The framework itself is composed of several layers. At the bottom is the foundation layer supporting basic building blocks such as animation, painting, and gestures. The rendering layer provides abstractions for dealing with layout and building the element tree, which represents the renderable state of widgets. The widgets layer has classes for basic widgets such as Container, Row, Column, etc. It serves as a composition abstraction over the rendering layer. Each render object within the rendering layer has a corresponding class in the widgets layer. This layer also implements the reactive programming model for widgets which is an important concept within the whole framework. Lastly, the Material and Cupertino layers provide widgets that use base widgets from the widgets layer to implement iOS and Android design languages into the framework [6].

The Flutter framework is small and leaves advanced functionality, such as camera, web view, HTTP, or more platform-specific features, such as in-app payments to other packages and plugins within its ecosystem. These packages and plugins can be found on pub.dev, the official package repository for Flutter and Dart, where developers can publish their public packages for use within Flutter projects [6].

This framework architecture gives the developer great flexibility when building the app. Mainly because the generated Flutter project exposes both the app written in Dart and the target platform code, called Runner, for modification. The Embedder sits on top of the Runner, allowing Flutter to be also integrated into existing applications. Moreover, it enables developers to implement application-specific platform-level code without writing plugins or extending Flutter. The app

code in Dart sits on top of the Flutter framework, leveraging its libraries to allow for building UI applications of any kind. Figure 3 shows this layered architecture.

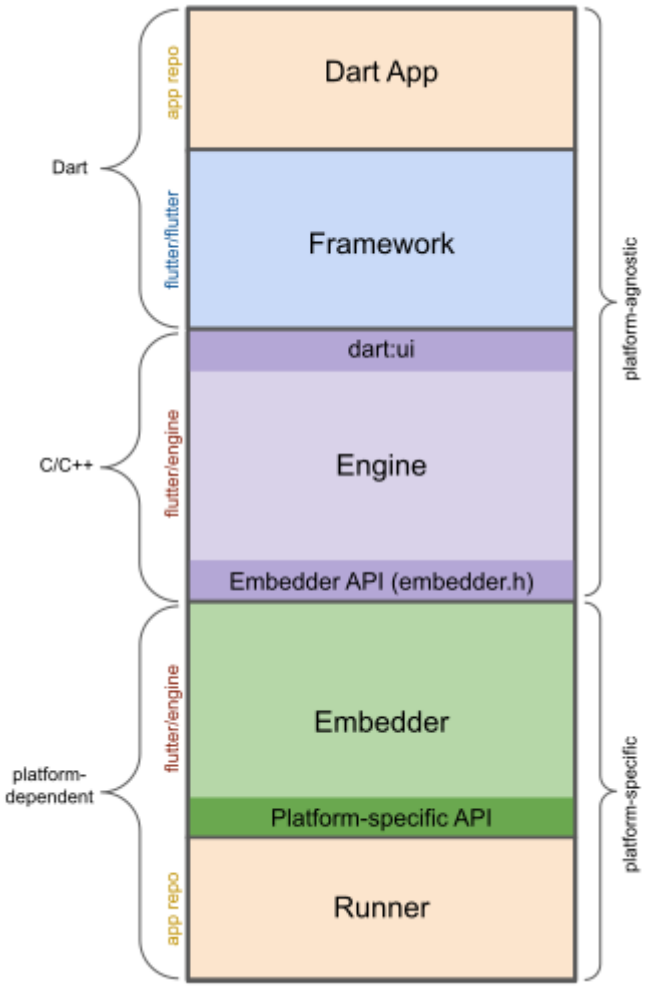

Figure 3: Anatomy of a Flutter app
(source: https://docs.flutter.dev/assets/images/docs/app-anatomy.svg)

Flutter employs a reactive approach to building user interfaces. It was inspired significantly from a model introduced by React, a web UI framework made by Facebook. In this model a UI is represented as a result of a function that gets state as a parameter. This approach of the state producing the UI eliminates the need for imperative UI updates because UI depends on the application state. When the application state changes, the UI reflects that change automatically. This function for building UI is often called many times and should be fast and free of side effects. The execution of this function also creates lots of objects, particularly when called every frame, for example, in the case of animations. Flutter uses Dart language, which is highly optimized for fast object instantiation and deletion. It allows

Flutter to run smoothly on most devices with 120 frames per second framerate. This characteristic makes Flutter apps responsive and fast to use, which is also great for games [6].

4.1.2 Widgets

Widgets are fundamental building blocks of the Flutter app. Each widget is an immutable declaration of a part of the user interface. Flutter uses widgets for almost everything UI related. The whole Flutter app itself is also a widget.

Widgets can be composed together to create a UI. A widget can accept child widgets as arguments which allow for composition. Flutter app's widgets are often composed using smaller single-purpose widgets, forming a widget hierarchy.

Having a widget in the hierarchy does not strictly mean that something will be drawn to the screen. Flutter uses widgets for utilities such as padding or alignment. These widgets only affect the layout of their child widget, positioning it somewhere on the screen.

For example, Container, a commonly used widget, is made up of several widgets responsible for layout, painting, positioning, and sizing. Specifically, Container is made up of the LimitedBox, ConstrainedBox, Align, Padding, DecoratedBox, and Transform widgets [6] .

Widgets are commonly created in Dart code by extending one of the two base widget classes exposed by the framework, “StatelessWidget” and “StatefulWidget”. The created widget is then composed using other widgets inside its overridden “build()” method, which returns an instance of a widget. As widgets are composed together, the framework recursively calls each widget’s “build()” method to form a widget tree, which is then transformed into an element tree for rendering. The first time this occurs is when the root widget of the app is passed into Flutter’s “runApp()” method. This creates the whole widget tree for the first time. As users navigate the app, interact with UI, make network requests, etc., flutter determines what part of the UI changed based on the state changes and rebuilds only parts of the widget tree affected by these changes.

As mentioned, widgets are usually made by extending two base widget classes, “StatelessWidget” and “StatefulWidget”. Most widgets extend

“StatelessWidget”, meaning they cannot have any mutable state. However, when a widget needs to change its state, for example, when a network request returns results and the widget wants to display these results, or when the user clicks on a button to change its color, then the result from the network request and the color of the button represents the state of the widget. To use state with a widget, the widget needs to subclass the latter of the base widget classes, “StatefulWidget”. Because all widgets are immutable, “StatefulWidget” stores the state in a separate class that extends “State”. “StatefulWidget” does not have a “build()” method. Instead, they use the “build()” method of their state object. Whenever the state object mutates, it must call the “setState()” method to signal the framework to call the “build()” method again to update the UI. This separation allows Flutter to destroy and create widgets as needed without worrying about the state, which is separated from the widget tree.

While the Flutter framework offers efficient state management, as applications grow in complexity, managing state across an entire app can become challenging. Various state management solutions, like Provider, Redux, and Riverpod, have emerged to address these challenges. These tools help in maintaining app state consistency, reduce boilerplate code, and facilitate more predictable state changes.

In conclusion, widgets are the cornerstone of Flutter's UI development. Their inherent properties like immutability, reusability, and customizability provide developers with a powerful toolset for building efficient and aesthetically pleasing apps. As the app's complexity grows, developers must also pay attention to global state management to ensure consistent and bug-free user experiences.

4.1.3 Dart

Dart is a client-optimized language for fast apps on any platform developed by Google. Flutter framework is implemented using Dart and Flutter developers use Dart to interact with the Flutter framework to build their apps making it the foundation of Flutter [7]. It is a multi-paradigm language but heavily focuses on object-oriented design using classes, inheritance and mixins. Recent Dart versions

also added patterns, which can be used to employ some functional patterns within the language [8]. Dart uses garbage collection for memory management [9].

Dart has a lot of features of other modern programming languages. It uses static type checking to ensure that variable's value always matches the variable's static type. However, type annotations are optional because the compiler can infer variable type from declaration. Type checking can also be deferred to runtime using the "dynamic" keyword if required. Another useful feature of Dart language is null safety. It means that the variables in the language cannot be null unless the programmer says that they can be using the question mark syntax like in other modern languages. Dart also features async await syntax for asynchronous operations on a single thread. But offers multi-threading (parallelism) support via isolates. It should be mentioned though, that isolates are not supported when compiling into JavaScript because JavaScript runtimes are often single threaded and require web-workers to emulate parallel execution. The async await syntax is also supported for streamed values [9, 10].

Every dart program starts within top level "main()" function. From there it can declare variables, call other functions, use control flow statements, etc... When some function fails an operation, it can throw an exception to signal the error using the "throw" keyword. Exceptions can be caught using try catch block which can specify a specific exception to handle using the "on" keyword. In theory, any arbitrary object can be thrown using "throw" keyword, but best practice is to use one of Darts two base types for exceptions – "Error" and "Exception". "Error" type represents a program failure that the programmer should have avoided and therefore should not be caught by a try catch block but reported to the developer to fix. "Exception" on the other hand is intended to convey information to the user about a failure so it should be caught by the app every time and display some information about the error to the user [11–13].

Dart can be compiled into several targets, but dart compiler mainly differentiates two ways how the code can be run. Native Platform and Web Platform.

Native Platform is intended for apps targeting mobile or desktop devices. Dart includes two compiler modes for these devices – just in time (JIT) and ahead-

of-time (AOT). During development on these devices Dart uses JIT compiler mode to allow for incremental recompilation (enabling hot reload) which boosts developer cycle that is critical for quick iteration. It also enables DevTools support, metrics collection and debugging support. The AOT compiler mode is used when the app is meant to be published in production. AOT compiler can compile to native ARM or x64 machine code. The compiled code runs within an efficient Dart runtime that takes care of memory management, sound type system checking and isolate management.

Web platform is intended for apps targeting web environments powered by JavaScript. Dart code is compiled straight into JavaScript which can be run in the browser. “Web platform” also has two compilation modes. One with incremental development compiler and an optimized production compiler which compiles Dart code to fast, compact, deployable JavaScript [9].

4.1.4 Hot reload

One of the most important features of Dart focusing on developer productivity is hot reload. Hot reload enables developers to iterate quickly, without waiting for the whole app to recompile, thanks to Dart’s JIT compiler incremental recompilation feature. Hot reload is heavily used when developing Flutter apps using Dart. When building UIs, fast visual feedback is necessary to experiment, build new features or fix bugs in the app. After Dart VM updates classes with the new versions of fields and functions, the Flutter framework automatically rebuilds the widget tree, allowing you to quickly view the effects of your changes [14].

When developing a Flutter app, hot reload does not always work because of some special cases, such as compilation error, adding assets, editing native code, or adding fields to widget states which needs to be initialized in “initState()”. In those cases, developers must use either hot restart, which restarts just the Flutter app without recompiling the native code of the app. Or full restart, which restarts the whole app. That takes significantly longer because it also recompiles the native code (Java / Kotlin / Swift / ObjC /…). On the web, it also restarts the Dart compiler. However, this does not occur that often during development.

DevTools

Dart offers a suite of performance and debugging tools that also integrate with Flutter called DevTools. Developers can use DevTools to gain insight into various aspects of their application, including performance, memory usage, and the widget tree which can help with debugging performance problems. DevTools can be integrated with various IDEs with support for Flutter or can be launched as a standalone web app using command line. Most common tools inside DevTools include Flutter inspector for inspecting the widget tree, which is useful when debugging visual errors in application's layout. Timeline view where developers can inspect each frame of the Flutter app and view its call stack. Therefore, knowing which part of code takes the most time to execute in that specific frame. Memory view assists in monitoring and managing the app's memory usage to prevent leaks and other memory-related issues. Performance view provides insights into app's performance metrics enabling developers to optimize the app. Debugger view helps with fixing issues by stepping through the code, setting breakpoints, and inspecting variables to ensure code correctness. Logging view collects and displays log messages from the app. Network view logs every request from the app which is useful for debugging network related issues. And finally, the App Size Tool can be used to analyze the size of the app, assisting developers in reducing app's footprint and optimizing its performance [15].

pub.dev

The official package registry for Dart and Flutter is pub.dev. It hosts libraries, tools, and frameworks published by the community and Google. Developers can explore and use a wide range of reusable packages to accelerate their development process, ensure code quality, and build feature-rich applications. Pub.dev also provides scoring and analysis to help developers choose high quality, well-maintained packages for their projects. Packages on pub.dev can be one of two kinds, a package, or a plugin. The difference is that a package contains only Dart code, while a plugin also contains platform code and is strictly dependent on the Flutter SDK. Platform code can include native Android, iOS, web, or desktop code

to facilitate interoperability with the target platform features with a common interface for the developer. An example of this is the official camera plugin maintained by the Flutter team. Each package on pub.dev has a list of compatible platforms in case of a plugin and information if it requires Flutter SDK or can be used with just Dart. It also provides a changelog, so developers can see changes between versions, example of usage and a link to automatically generated documentation. Package authors can also include a link to a package homepage, which is usually a hosted git repository or custom documentation website [16] .

5 Practical Part

This part of the thesis focuses on building an example game application using the Flutter framework. The game in question is a classic 2D arcade game called Block Breaker. This game was chosen due to its simplicity and physics-based gameplay which allows demonstrating various game development aspects when using Flutter for the development. The game also features multiple screens with custom UI elements to leverage Flutter's UI capabilities and postprocessing effects using custom shader code. The code of the project is available on GitHub: https://github.com/daviddomkar/block_breaker.

5.1 Game Rules

There are a lot of variations of the Block Breaker game with different sets of rules. This project follows its own set of rules which made sense to the author when implementing the gameplay logic and are explained in this section.

The player's objective in Block Breaker is to destroy every block on the board with a ball controlled only by using a paddle at the bottom of the screen. The paddle can only move to the left and to the right within the bounds of the board. Ball bounces off the paddle, board walls and blocks on the board without losing its velocity. When the ball collides with a block, it lowers its tier or destroys the block when at the lowest tier. Collision with a block increments a score counter with a value based on the tier of the target block. When all blocks are destroyed, a win condition is triggered and the player wins. The player can also lose the ball when they fail to catch it with the paddle and the ball falls under the board. When this happens, a player loses one of their 3 lives, leading to a game over condition when no lives are left.

This implementation of block breaker offers 10 levels. Each level is unlocked after completing the previous one and each level is of an increased difficulty. Levels consist of multiple blocks of various tiers positioned on the board. To clarify the rules, provided in Figure 4 are all blocks that can appear in the level, sorted in descendant order by their rarity with score value included. There is also a special

tier of block which is unbreakable by the ball to make levels more interesting. This special tier of block is not required to be destroyed to trigger a win condition.

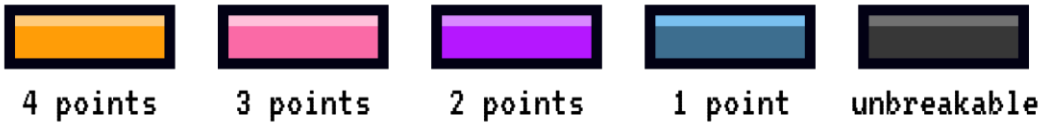

Figure 4: Various block types
(source: own work)

5.2 Application overview

The application itself consists of three screens – home screen with logo and a play button, level selection screen where player can select an unlocked level they want to play, and level screen – interactive screen with score counting, lives and states for various game conditions (won, game over, paused, etc.). The app is set up in such a way that the widget responsible for the game itself is always visible behind the UI, no matter on which screen the user currently is. Figure 5 shows the screens of the finished application.

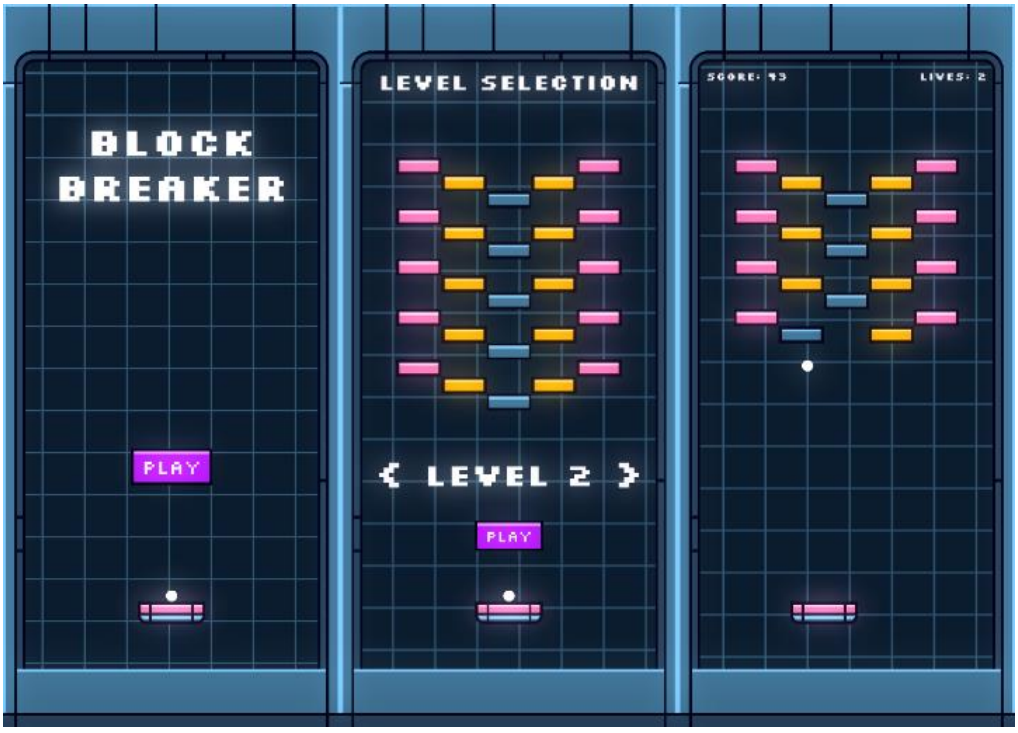

Figure 5: Showcase of the application screens
(source: own work)

5.2.1 Existing packages

It should be mentioned that there are several ready-made packages for Flutter made for game development such as Flame or SpriteWidget. However, one of this theses objectives is to explore the inner workings of Flutter and how to bend the framework to facilitate game development requirements. Because of these goals, the project does not use existing solutions and implements many things from scratch only with inspiration from existing solutions. Another reason to avoid ready-made packages is that they usually hide the inner workings behind abstractions such as component-based system. While this abstraction is proven to be a good model to follow when building games, used by big engines such as Unity or Unreal Engine, it once again goes against the aim of this thesis and implementing such system as a part of this project would just add unnecessary complexity. With the above considered, the project uses a simple object-oriented approach with minimal hierarchy to achieve an easy-to-read codebase considering its relatively small scope. However, when building a bigger project, one should consider existing solutions to avoid reinventing the wheel.

5.2.2 Project Structure

The project structure is conceptually divided into three layers. The first layer is the engine where the core functionality of the game, which does not contribute to actual gameplay, is located. This includes mounting the game inside Flutter's widget tree, running the game loop, processing input, and rendering things to the screen. This part would ideally also include physics. However, to simplify the project, it uses a Dart port of the popular Box2D physics engine called forge2d. Game engine physics by itself is a large topic outside the scope of this thesis.

The second layer of the project is the game itself. The game is just a simple class with update and render method, as well as having method handlers for the input events coming from the engine. The game is completely unaware of Flutter's widget tree and uses the engine to mount itself into it. The game includes all the gameplay logic and variables, such as game state, score count, or player lives. It also includes entities such as the ball, blocks, paddle or the board and their physical bodies while also registering handlers for collision detection between the

entities. The game can also be used as a notifier for rebuilding the Flutter's widget tree when its state changes.

The last part of the project is the UI. The UI part follows Flutter's best practices for making user interfaces and is responsible for composing the game behind its screens, such as home screen, level selection screen and level screen. It is also responsible for routing between the screens and applying postprocessing effects over the whole application.

Flutter Project

This thesis does not go over creating a new Flutter project; however, it mentions some specifics about the created project regarding the project structure outlined earlier.

One such specific is the structure of the lib directory, where all Dart code for the application lives, according to the specified parts of the project structure. There are folders for individual project layers – the engine, the game, and the UI. The structure itself can be seen in Figure 6.

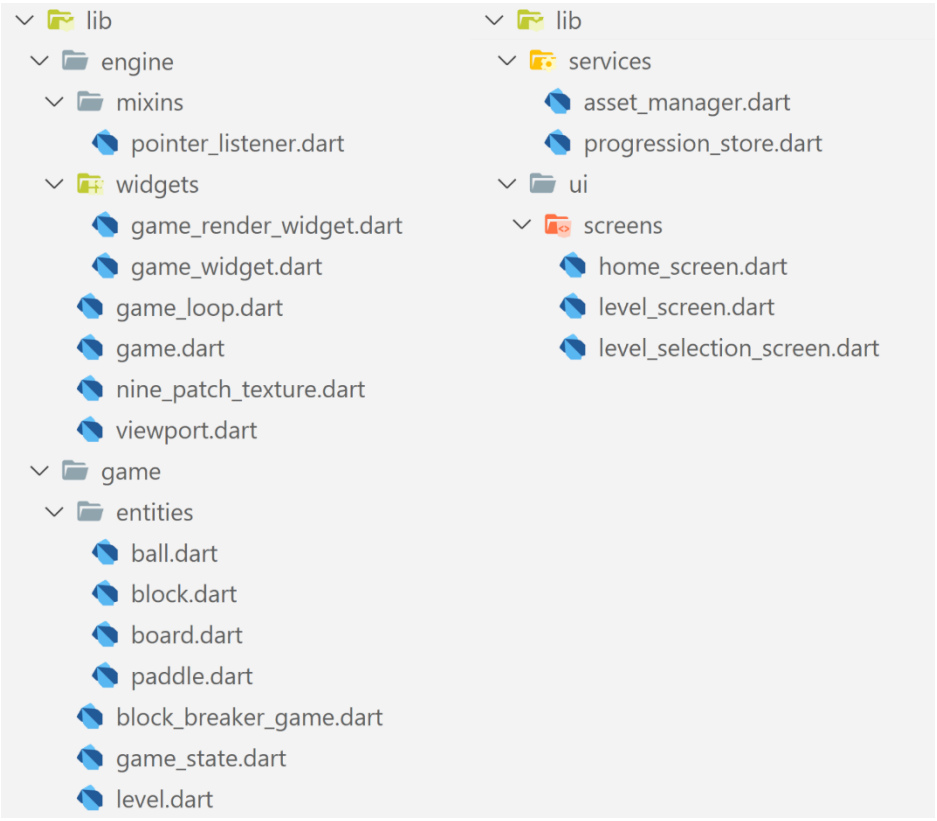

Figure 6: "lib" directory structure
(source: own work)

There are also additional folders and files needed for the app functionality, such as “services” folder, with services for managing application assets and saving and loading player’s progression of levels. The “constants.dart” file provides constants for in-game dimensions of various entities, which are shared with the UI to correctly position widgets according to in-game entity positions. The “theme.dart” file configures the Flutter theme for the app and finally the “main.dart” file serves as the app’s entry point which takes care of the initial configuration and calls the “runApp“ function which bootstraps the main widget – “BlockBreakerApp“.

Another specific aspect of this project is the structure inside assets directory. In Flutter, assets can be anywhere in the project directory as long as they are specified inside the project’s “pubspec.yaml” file. However, it is best practice to put all assets inside “assets” directory to make it clear that these files are assets. In this project, the assets directory is further divided into “fonts”, “images”, “shaders” and “textures” folders containing respective files used inside the app. The whole structure can be seen in Figure 7.

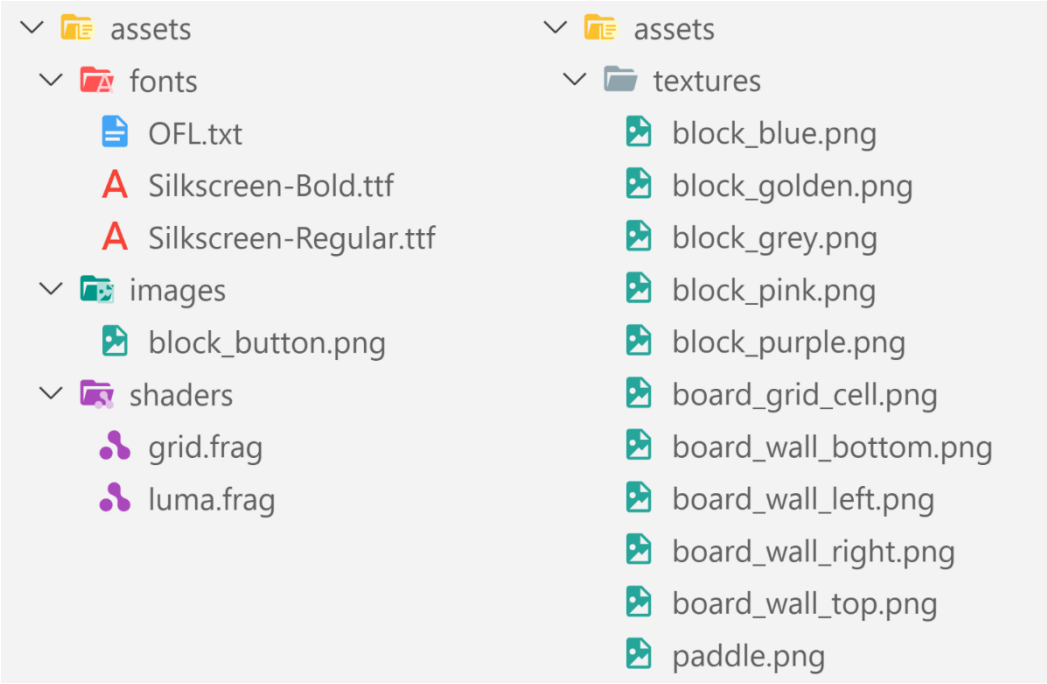

Figure 7: “assets” directory structure
(source: own work)

5.3 Engine

The engine layer of the application is responsible for all low-level game functionality which is not gameplay code. For example, it provides custom "GameWidget", which allows mounting the "Game" class instances into the widget tree. This class can be extended by the game layer to implement the specific game, in the case of this project, such class is called "BlockBreakerGame". Engine also manages the game loop, which runs the game logic every frame. Through the means of "GameWidget", the engine renders the game to the screen and processes all kinds of input from the user (touches, mouse movements, etc.).

5.3.1 Game Widget

As mentioned in the theoretical part of this thesis, Flutter apps are composed of widgets, mainly of two types of widgets – stateless and stateful. Widgets are used to build everything we see on the screen. Based on this information, it would make sense to build the whole game using widgets. For example, the game could have a "BoardWidget" and a "PaddleWidget". Then it could use the Flutter's widget tree to compose those widgets together, the "PaddleWidget" would be a child of a "BoardWidget". There are several problems with this approach. The widget tree is mostly declarative and when there is a need to change the state of a widget, it needs to rebuild all child widgets. The Flutter engine then needs to figure out what elements changed in the element tree and finally propagate those changes to the render tree. While Flutter is heavily optimized for this, even during animations where widgets are rebuilt hundreds of times over a one second period, this approach is not very ideal. Games usually have many dynamic objects and state changes and most importantly, they have a game loop in which they need to process inputs and run gameplay code. The widget hierarchy would need to be rebuilt constantly. For example, when the mouse hovers over the board to move a paddle or when the ball moves every frame in a specific direction, for which specifically the time elapsed between the frames must be known to achieve constant speed. Additionally, the widget hierarchy would be very messy because state needs to be propagated between the widgets. This is usually done by passing the state through constructor parameters of the individual widgets during rebuild

or by using a state management solution such as change notifiers or streams. The state is usually propagated top to bottom, which would make interaction between widgets at the same level of the tree particularly difficult.

The approach the engine uses is having only one widget for the entire game, the "GameWidget". This widget takes care of input processing, updating the game loop and rendering the game to the screen. It accepts abstract "Game" class which can be extended to implement update loop and render loop by specific game class. Code snippet 1 shows the implementation of the "Game" class.

```
abstract class Game extends ChangeNotifier {
  final Viewport _viewport;

  Game({
    required Viewport viewport,
  }) : _viewport = viewport;

  void init() {}
  @mustCallSuper
  @override
  void dispose() {
    super.dispose();
  }

  void update(double dt) {}
  void render(Canvas canvas) {}

  Viewport get viewport => _viewport;
}
```

Code snippet 1: Abstract "Game" class
(source: https://github.com/daviddomkar/block_breaker/blob/main/lib/engine/game.dart#L7)

This class also extends "ChangeNotifier", which means it can be used by the UI layer to react to any state changes by using "notifyListeners()" method from within the specific game class. The class also contains "Viewport" class instance, which is required during its construction to specify the dimensions and scaling behavior of the game world space. It is used by the "GameWidget" class to translate input event coordinates into the game world space coordinates. Additionally, there are two methods – "init()" and "dispose()". The first one is called when the game object is first mounted to the widget tree by the "GameWidget", and the latter is called when

it is being disposed from the widget tree. It also calls “dispose()” on its parent “ChangeNotifier”.

“GameWidget” class itself is a standard “StatelessWidget”, it is composed of the “Listener” widget for receiving pointer events from touches on mobile devices and mouse events on desktop devices. Then it uses “MouseRegion” widget to hide the mouse cursor inside the game. Finally, there is “GameRenderWidget”, which is a part of “GameWidget” responsible for updating the game loop and rendering.

5.3.2 Input

Input processing is managed by the “Listener” widget from the Flutter SDK. It provides handlers for various input events and the “GameWidget” uses these handlers to transform input events coordinates into the game world space and forwards them to the game instance.

Every handler checks if the game is of type “PointerListener”, indicating that it wants to receive pointer events. “PointerListener” is a Dart mixin. Mixins can be thought of as Dart equivalents to interfaces in other languages, but they are less restrictive and can also contain method definitions or member attributes. In the case of “PointerListener”, it only contains declarations of methods for receiving input events. Provided below is a list of those methods.

- onPointerDown – mouse pointer or finger pressed.
- onPointerUp – mouse pointer or finger released.
- onPointerMove – mouse pointer or finger moved.
- onPointerHover – mouse pointer hovered.

The specific game class can then override “PointerListener” methods to receive events with transformed coordinates. “PointerListener” can be expanded with more events from the “Listener” widget as needed.

5.3.3 Rendering

The rendering of the game is facilitated by “GameRenderWidget”. This widget is a special kind of widget that enables custom rendering and contains the game loop.

This widget, like the “GameWidget” just accepts the game as a constructor parameter and calls its methods. The important thing to note is that this widget is neither “StatelessWidget” nor “StatefulWidget”. That is because this widget has nothing to do with the widget tree. It uses completely custom rendering that needs to happen in an entirely different place – the render tree.

The render tree is different from the widget tree. It is not composed of widgets but render objects. Render objects handle sizing, layout, painting, and semantics of widgets that create them. Render objects usually live across multiple frames, until the widgets that created them are disposed. Each frame, render object might be updated by Flutter if something in the render tree changed. This means that unlike immutable widget objects, render objects are highly mutable and they need to correctly manage their state.

There are three important methods in a render object. “performLayout()” used for determining the widget size, “paint()” used for painting to the screen and “describeSemanticsConfiguration()” used for providing values for the semantics tree. When the render object is created, Flutter calls these three methods to initialize the render object. When the widget in the widget tree changes (for example due to “setState()” call), the render object is updated with new values from the newly created widget object and depending on what changed, the render object can mark itself to run any of the three methods again to update its size, paint something new to the screen or to update its semantics [17]. In the case of the render object created by “GameRenderWidget”, the “paint()” method is the most important, because that is where the engine layer does the custom rendering. At the end of each game loop iteration, the code can signal that the render object needs to be redrawn using “markNeedsPaint()” method.

To create a custom render object, the “GameRenderWidget” extends a special kind of widget class called “LeafRenderObjectWidget”. This class further extends from “RenderObjectWidget” which extends from “Widget” from which the already known “StatelessWidget” and “StatefulWidget” also extend. It is clear from this inheritance hierarchy that the “LeafRenderObjectWidget” functions differently from the normal stateless and stateful widgets, but since it is still a widget, it can be used inside the widget tree as any other widget. The word “leaf” in the widget

class name indicates that this widget will not have any child widgets and will reside at the leaf node of the tree. There are also other widget classes that extend “RenderObjectWidget” which can be used to render child widgets in specific ways. However, the “GameRenderWidget” does not need such functionality.

All “RenderObjectWidget” descendants need to implement two methods – “createRenderObject” and “updateRenderObject”. The first is called by the Flutter framework when the widget is mounted to the tree and is responsible for creating a render object. The latter is called when the widget changes and is responsible for updating the render object with new values [17]. The render object itself is another object entirely. In the case of “GameRenderWidget”, its class is named “GameRenderObject” and extends “RenderBox”. “RenderBox” is one of many “RenderObject” descendants which is useful when something needs to be rendered that fits within the bounds of a rectangle, for example a frame the game needs to render [17]. The implementation of “GameRenderObject” is inspired by the implementation of “GameRenderBox” from the Flame library [18].

“GameRenderObject” class has only two members, the game, which is assigned during construction and the game loop variable, which is explained in its own section. The important thing is the implementation of the setter for the game instance, this setter is used from “updateRenderObject()” inside “GameRenderWidget” to update the game when the widget changes. It first checks if the new game instance is different from the current game instance. In that case, the setter just returns because there is nothing to update. However, if the game instance is changed, the widget will call “_detachGame()“ for the old game instance, updates current game instance and then calls “_attachGame()“ for the new instance. These two methods perform initialization and deinitialization of the game which ultimately leads to calling “init()“ and “dispose()“ methods in the „Game“ class. “GameRenderObject“ also overrides methods “attach()“ and “detach()“ which are called when the render object is attached to the render tree or detached from it. These methods also call specific methods for managing the game and are responsible for the first attachment and final detachment of the game.

For the “GameRenderObject“ to be mounted to the render tree, it needs to implement a few things, the “computeDryLayout()“ method is one of them. This

method is responsible for determining how big is this render object going to be. The parameter to this method are the parent widgets constraints and because the game fills the entire screen space, the method just returns the biggest constraints by using “constraints.biggest” getter [19].

Another method to implement is the “performLayout()“ method to decide how big the render object will be in those constraints specified by the “computeDryLayout()“. Because the game fills the entire screen and the render object needs to be as big as it can, it is enough to instead override the boolean getter “sizedByParent“ to return “true“. This means that the render object gets the size attribute set by the parent render object. When this happens, the method “performResize()” is called on the render object whenever the parent size is changed. For example, when changing window resolution or rotating the mobile device. This is useful because the render object can react to this and use the new size to update the game’s viewport, achieving responsive behavior [20].

Last and the most important method to override to render something to the screen is the already mentioned “paint()” method. This method accepts “PaintingContext” and the “Offset” of the widget. “PaintingContext” is used to paint child render objects and can push new painting layers which can be painted using “Canvas” class instance. Since “GameRenderObject” does not have any children and the “PaintingContext” instance comes from the painting layer this render object should be render to, the “Canvas” instance from the “PaintingContext” accessed with “context.canvas” can be used to draw everything. The “GameRenderObject” first translates the canvas to the provided offset using “canvas.translate()” method, then it saves the canvas transform using “canvas.save()” method, transforms the canvas using transform matrix from the game’s viewport to make any next canvas methods draw in game world space using “canvas.transform()” method, calls “render()” method on the game and passes in the “Canvas” instance from the “PaintingContext”. The game can then perform its own rendering using the provided canvas which is already transformed into its viewport coordinates. Thanks to this approach, the game layer does not need to worry about viewport transformations which are abstracted away and can focus only on gameplay code as intended. The last step in the “paint()” method is a call to “canvas.restore()”

method which restores the canvas transform. Code snippet 2 shows the implementation of the “paint()” method.

```
@override
void paint(PaintingContext context, Offset offset) {
  context.canvas.translate(offset.dx, offset.dy);

  final transform = _game.viewport.transform;

  context.canvas.save();
  context.canvas.transform(transform.storage);

  _game.render(context.canvas);

  context.canvas.restore();
}
```

Code snippet 2: "paint()" method (source: https://github.com/daviddomkar/block_breaker/blob/main/lib/engine/widgets/game_render_widget.dart#L92)

5.3.4 Game Loop

The next important part of the engine is the game loop. Game loop is a loop that iterates each frame of the game and is used for performing gameplay logic. It starts when the game is attached and stops when the game is detached from the widget tree. The game loop in this engine is its own class, called “GameLoop”, it uses “Ticker” class from Flutter framework. “Ticker” class expects callback function in its constructor and calls this function on every frame of the application, providing current ticking duration.

The game loop is not actually created inside “_attachGame()” method, but later when the first “performResize()” is called. The reason for this is the fact that the game needs to know the size of the viewport when doing gameplay logic. Before “performRezize()” is called, the viewport has the size unknown. However, stopping and disposing of the game loop happens inside the ”_detachGame()” method.

The “GameLoop” class has methods for starting, stopping, and disposing the ticker as well as the callback tick function where it computes duration between ticks called delta time, to achieve that, it stores the previous reported ticking duration as its private attribute. Delta time can be used in the “onUpdate” callback

of the “GameLoop” class to achieve consistent speed of objects across multiple frames. The implementation of the “_tick” method, computing the delta time and calling custom “_onUpdate” callback can be seen in Code snippet 3.

```
void _tick(Duration duration) {
  final durationDelta = duration - _previousDuration;

  final dt = durationDelta.inMicroseconds /
Duration.microsecondsPerSecond;

  _previousDuration = duration;

  _onUpdate(dt);
}
```

Code snippet 3: "_tick()" method (source: https://github.com/daviddomkar/block_breaker/blob/main/lib/engine/game_loop.dart#L31)

When creating the game loop, “_onUpdate” callback is supplied to it. This update callback calls the update method on the game instance, passing in the delta time. It also marks the render object to be repainted. This is important so the paint method is called, and a new frame is rendered to the screen using the game’s render method. The “_onUpdate” callback can be seen in Code snippet 4.

```
void _onUpdate(double dt) {
  _game.update(dt);
  markNeedsPaint();
}
```

Code snippet 4: "_onUpdate()" callback (source: https://github.com/daviddomkar/block_breaker/blob/main/lib/engine/widgets/game_render_widget.dart#L134)

The game loop should not be running all the time. When the application is in the background or minimized, it should stop wasting CPU and GPU resources. To achieve this, the “GameRenderObject” implements “WidgetsBindingsObserver” mixin. This mixin lets the class observe changes to the widgets binding, which is responsible for binding the main flutter widget to the target platform rendering surface. The class instance needs to register itself as an observer to the widgets binding, this can be done by calling “addObserver()” method on the widgets binding which is globally available as a singleton by calling “WidgetsBinding.instance”. The observer needs to be removed from the widgets

binding when the render object is detached, for that the method "removeObserver()" can be used. These methods are called inside "_attachGame()" and "detachGame()" respectively.

When the "WidgetsBindingObserver" mixin is implemented, the render object can override the "didChangeAppLifecycleState()" method which receives current "AppLifecycleState" as parameter. "AppLifecycleState" is an enumeration of all possible lifecycle states which can be switched over to stop or start the game loop as needed. This method can be seen in Code snippet 5 and concludes the "GameRenderWidget" functionality.

```dart
@override
void didChangeAppLifecycleState(AppLifecycleState state) {
  if (attached) {
    switch (state) {
      case AppLifecycleState.resumed:
        _gameLoop?.start();
        break;
      case AppLifecycleState.detached:
      case AppLifecycleState.hidden:
        _gameLoop?.stop();
      default:
        break;
    }
  }
}
```

Code snippet 5: "didChangeAppLifecycleState()" method (source: https://github.com/daviddomkar/block_breaker/blob/main/lib/engine/widgets/game_render_widget.dart#L106)

5.3.5 Nine-patch Textures

Almost every game needs some texture assets to represent in-game objects. In this project every game object (board, paddle, blocks, and the ball) has a texture. In Flutter, the "Image" class from the "dart:ui" package (not to be confused with "Image" widget) can be used to represent these textures and can be rendered by using the "Canvas" class. But because dimensions of these objects are set in code and can be changed the textures need to be adapted to those sizes without stretching. This can be achieved using nine-patch textures. Such texture is an image divided into 9 segments using two vertical and two horizontal lines, so it forms 3 by 3 grid of regions. The corner regions of this texture are always the same

size, while the side and center regions are stretched when the texture is resized. This means that if the texture is correctly designed, it can be freely resized and adapted to different dimensions. Flutter's canvas has the "drawImageNine()" method, which can be used to draw an image with specified center region of nine-patch texture. The engine contains a custom class "NinePatchTexture" which makes loading and drawing these textures easier and uses the provided "drawImageNine()" method to do so. The class also allows setting scale attribute, which is useful when avoiding problems with nine-patch textures. For example, when the target region to draw to is smaller than the nine-patch texture's 4 corners, nothing is rendered. But if the texture is uniformly scaled down, it will fit the region just fine. The implementation of "NinePatchTexture"'s "render()" method can be seen in Code snippet 6.

```
void render(Canvas canvas, Rect rect) {
  canvas.save();

  canvas.scale(1 / _scale);

  canvas.drawImageNine(
    _image,
    _center,
    Rect.fromLTWH(
      rect.left * _scale,
      rect.top * _scale,
      rect.width * _scale,
      rect.height * _scale,
    ),
    _paint,
  );

  canvas.restore();
}
```

Code snippet 6: "render()" method of the "NinePatchTexture" class (source: https://github.com/daviddomkar/block_breaker/blob/main/lib/engine/nine_patch_texture.dart#L36)

5.4 Game

This part of the thesis focuses on the game layer of the block breaker game application. The game layer uses the engine layer to implement gameplay logic specific for the block breaker game and its rules. The game layer also uses physics

library forge2d to handle collisions of objects and reacts to them as well as counting the score and lives of the player.

5.4.1 Main Game Class

The main part of the game layer is a class called “BlockBrakerGame”, it extends “Game” class from the engine layer and implements “PointerListener” mixin to react to input events. It also overrides “init()”, “dispose()”, “update()” and “render()” methods from the “Game” class to implement gameplay logic and rendering. Moreover, the class overrides “onPointerUp()” , “onPointerMove()” and “onPoinerHover()” methods from the “PointerListener” mixin to implement movement of the paddle based on the pointer position.

The class accepts “AssetManager” class in its constructor. “AssetManager” is custom class which is initialized when the app is started and loads all assets required for the game. This includes all textures and custom shaders.

The game class also contains attributes for the physics world from the forge2d library used for physics simulation, game state variables such as the ”GameState” enum, current time which is accumulated for the duration of the game running, player lives and player score while also containing variables for all entity types such as the board, paddle, ball, and all blocks.

5.4.2 Viewport

The first thing “BlockBreakerGame” class does after initializing its attributes to default values during construction is viewport initialization of its parent “Game” class. This setups game world space dimensions and configures the behavior of the viewport when the window is resized by specifying “minSize” and “maxSize”. The viewport functions in a way that its dimensions are always at least “minSize” but when the window aspect ratio allows it, it can grow in both dimensions up to “maxSize”, the last parameter into the viewport is the location of the ”origin” point of the coordinate system in normalized coordinates. By default, this is the top left corner of the window. Since the game is designed in a way that expects the origin in the center of the screen, the origin is moved by -0,5 in both directions.

The “kViewportSize” constant is used as the “minSize” of the viewport. It is defined as “Size” with a width of “480” and a height of “960”. This is the design size of the game and specifies the dimensions in which the game world will be represented. “maxSize” is defined as infinite, which extends the dimensions in one of the axis when the aspect ratio is not equal to the design size aspect ratio. This additional space is dimmed and only displays animated background using a custom shader to fill the rest of the game window in the horizontal axis. In the vertical axis, the board’s borders are expanded using nine-patch textures to always center the board and fill the remaining vertical space.

5.4.3 Game States

To control the game logic and decide what happens when, the game has several states it can be in. This is represented by the “GameState” enum with following state constants:

- notStarted – The game is not on the game screen.
- ready – The game is ready to start, waiting for player input.
- playing – The game is running with all gameplay logic in place.
- paused – The game is paused; game loop is stopped.
- gameOver – Player lost the game.
- won – Player won the game.

When the game is in background, for example on the home screen or on the level selection screen, the state of the game is “GameState.notStarted” it means that it ignores the input. When the level is selected and loaded, the game starts and transitions into the “GameState.ready” state. In this state, the player can move the paddle, but the ball remains attached to it until the game receives the pointer up event, signaling that the player moved the paddle into the position from which the ball should be fired. After that, the game fires the ball from the paddle and finally transitions to the “GameState.playing” state. During “GameState.ready” or "GameState.playing”, the game can transition into “GameState.paused” by calling the “pause()” method on the “BlockBreakerGame”. When in paused state, the game stops physics simulation which can be then resumed by calling the “resume()”

method, which transitions the game into a state it was in before it was paused. When the game ends, it transitions to “GameState.gameOver” or “GameState.won” state depending on the outcome of the game. From this state the player can either exit the level screen, in which case the game transitions back to the “GameState.notStarted” state or can choose to go to the next level, in which case the game transitions into “GameState.ready” state with the next level loaded.

5.4.4 Physics

The game uses physics library forge2d to simulate physics of objects. This has the advantage of the engine not having to implement anything physics related. Forge2d library is a Dart port of the popular Box2D physics library. “BlockBreakerGame” initializes the forge2d “World” class in its constructor and updates the world with delta time in each frame inside its update method under the condition that the game is not paused. Code snippet 7 shows this logic.

```
@override
void update(double dt) {
  // ...

  if (_state != GameState.paused) {
    _world.stepDt(dt);
  }

  // ...
}
```

Code snippet 7: World updating inside "update()" method (source: https://github.com/daviddomkar/block_breaker/blob/main/lib/game/block_breaker_game.dart#L149)

Every entity accepts the world as constructor parameter and can use the world to create its physical body. Physical bodies have 3 types – static, kinematic, and dynamic. Static bodies never move and behave like having infinite mass. They also do not collide with other static bodies or kinematic bodies. Kinematic bodies move according to their velocity but do not respond to forces. Their position can also be set externally. Kinematic bodies also have infinite mass and do not collide with other static bodies or kinematic bodies. Finally, there are dynamic bodies which are fully simulated, moving according to forces, and having finite mass. Every

physical body type can have multiple fixtures attached. Fixtures bind shapes to physical bodies as well as setting material properties such as density, friction, and restitution. Entities in the game use physical bodies as their in-world representation and rendering themselves accordingly. Due to limitation of the original box2d library, object sizes are defined in meters and the library can work well with objects of 50 meters maximum. Because the game design size is larger than that, whenever physics is needed, the coordinates are transformed to the physics world coordinates which are defined as 10-times smaller than design coordinates. This ensures that physics simulations work correctly [21–23].

5.4.5 Entities

Every object inside the game is considered an entity. In this project there are 4 types of entities – “Board”, “Paddle”, “Ball” and “Block”. It is common in games using OOP design to have one common “Entity” class as a generalization for all entities. However, in this project, there is a relatively small number of entities, and they function very differently from each other. Due to this fact, each entity is its own class and does not have a parent class. “BlockBreakerGame” class initializes each entity in its constructor apart from the “Block” entity because the “Block” entity is the only entity there can be multiple of and the only entity that can be removed from the game world. It is first created when loading a specific game level. This section explains every entity and their relation to the gameplay logic in detail.

Board

Board is an entity that defines the board in which the block breaker game is played. It consists of size which defines the inner size of the board and the physical body containing fixtures for all edges of the board which act as boundaries for the ball to bounce off the edges. Fixture is omitted from the bottom edge of the board because the ball should be able to fall behind the bottom edge when not caught with the paddle. The “Board” class also accepts “AssetManager” in its constructor to access the board wall textures when rendering and “Viewport”, which is

important because it needs to calculate its position inside the viewport, so it is always at the center of the screen, expanding walls in the vertical direction.

When initialized, the board uses the physics world to create its body. The body type of the board is static because it does not move. It consists of top, left and right fixture with edge shape and restitution of 1. It means that when something collides with this fixture shape, it does not lose force. This is important because when the ball collides with the board walls there should be no speed decrease of the ball. To create these fixtures, the board needs to compute its inner bounds using the provided size, assuming the origin at the center of the screen. It then creates fixtures based on those bounds using the private "_createBoundaries()" method which returns the resulting body. The "_createBoundaries()" method calls "_createBoundaryEdge()" method for every edge, which constructs a fixture definition for the edge shape using start and end vectors.

When rendering the board, the code needs to do calculations to correctly position the board to the center of the viewport and expand top and bottom wall textures. This is done by specifying outer bounds in addition to the inner bounds and using them to calculate rectangular area for each board texture from the asset manager. Code snippet 8 shows the rendering code of the board's top wall texture. The magic number "32" in the code is the width of the wall texture design size offset used to correctly position the texture inside the board.

```
void render(Canvas canvas) {
  final boardWallTopRect = Rect.fromLTWH(
    outerBounds.left,
    outerBounds.top,
    outerBounds.width,
    (outerBounds.height - innerBounds.height) / 2 + 32,
  );

  _assetManager.boardWallTopTexture.render(canvas, boardWallTopRect);

  // ... rendering of other board wall textures

}
```

Code snippet 8: Board's wall texture rendering (source: https://github.com/daviddomkar/block_breaker/blob/main/lib/game/entities/board.dart#L75)

From the game class, the board is created inside the constructor and is disposed of when the “dispose()” method on the board is called inside “dispose()” method of the game class. The “dispose()” method of the board is responsible for destroying the physical body and removing it from the physics world.

When rendering the board inside the game’s “render()” method, region outside the board outer bounds is made dimmer to emphasize the board in the center of the screen on wider aspect ratios. This is achieved by multiplying all pixels by a value smaller than 1 excluding the rectangle formed by board’s outer bounds. Inside the “render()” method, this is done using “canvas.saveLayer()” which can be used to paint a new layer which is then combined with previous layer after “canvas.restore()” using a blend mode specified in “canvas.saveLayer()” parameters. Code snippet 9 shows canvas commands performed to achieve the desired effect.

```
// multiplies everything drawn before with pixels from the layer
canvas.saveLayer(null, Paint()..blendMode = BlendMode.multiply);

// draws darker color over the whole viewport
canvas.drawPaint(Paint()..color = const Color(0xFF666666));

// exclude pixels from this layer from pixels from previous layer
canvas.saveLayer(null, Paint()..blendMode = BlendMode.xor);

// draw the cutout using board’s outer bounds
canvas.drawRect(
  _board.outerBounds,
  Paint(),
);

// restore to previous layer
canvas.restore();

// restore to previous layer
canvas.restore();
```

Code snippet 9: Canvas commands for rendering dimmed background effect (source: https://github.com/daviddomkar/block_breaker/blob/main/lib/game/block_breaker_game.dart#L181)

Paddle

Paddle is the entity player is controlling in the game using touch or mouse pointer movements to repel the ball. It has a kinematic physical body because it can be moved but cannot react to forces. The position of the paddle's physical body is updated based on the pointer x position while the y position is always the same. The x position is always clamped to the inner bounds of the board with tolerance for the width of the paddle. The fixture of the paddle has a polygon shape of a box and the restitution of 1 so the ball does not lose its speed when colliding with the paddle.

When rendering the paddle, the paddle nine-patch texture is used from the "AssetManager" class. This texture is adapted to the specified paddle size. And the position and dimensions of the paddle are calculated from the attached physical body. Everything needs to be multiplied by 10 to transform the physical body size to design size.

The paddle is initialized in the "BlockBreakerGame" class constructor and disposed inside the "dispose()" method like the board entity. Additionally, when the input is received inside the "onPointerMove()" or "onPointerHover()" method of the "BlockBreakerGame" class, it is forwarded into "_processMoveInput()" helper method which moves the paddle according to the input event.

Ball

Ball entity is the only entity with dynamic body in the game. The ball is fired from the paddle at the start of the game by applying linear impulse to it in a specific direction based on the paddle position being on the left or right side of the board. Code snippet 10 shows the usage of the "applyLinearImpulse()" method from the physics library inside "fire()" method of the ball.

```
void fire() {
  _body.applyLinearImpulse(
    Vector2(_body.position.x < 0 ? _force : -_force, -_force),
  );
}
```

Code snippet 10: Usage of the "applyLinearImpulse()" method (source: https://github.com/daviddomkar/block_breaker/blob/main/lib/game/entities/ball.dart#L62)

The ball then moves inside the board's inner bounds, colliding with blocks, board walls and with the paddle. When the player fails to move the paddle in time to repel the ball it falls behind the bottom board edge which has a consequence of player losing one life or when having no lives left, transitioning game into "GameState.gameOver" state. This condition is checked every frame of the game inside the "update()" method.

The ball is created like the paddle or the board, during the "BlockBreakerGame" class constructions and is disposed when the "dispose()" method of the class is called. The rendering of the ball is slightly different from other entities since it does not have a texture. It is simply a white circle with the radius of the fixture shape radius. To render the ball "canvas.drawCircle()" method can be used. Code snippet 11 shows the „render()" method implementation of the „Ball" class.

```
void render(Canvas canvas) {
  canvas.drawCircle(
    Offset(_body.position.x * 10, _body.position.y * 10),
    _body.fixtures.first.shape.radius * 10,
    Paint()..color = const Color(0xFFFFFFFF),
  );
}
```

Code snippet 11: "render()" method of the "Ball" class (source: https://github.com/daviddomkar/block_breaker/blob/main/lib/game/entities/ball.dart#L68)

Block

Block is the final entity the game has. Block entities are forming game levels which means there can be multiple of them. They have a static physical body with the restitution of 1 like the walls of the board to repel the ball on collision. Player must destroy all destroyable blocks in the level using the ball to complete the level.

As mentioned in a section about game rules, blocks have tiers defined by the "BlockTier" enum. Tier of the block affects multiple things – the appearance – meaning the texture being rendered, the score value player receives when destroying the block using the ball and the action that happens on collision with the ball. The special block tier "BlockTier.grey" cannot be destroyed and serves only as an obstacle to the ball. When the ball collides with the block, it calls the "destroy()" method on the block which decides what to do with a block based on

its tier and returns score earned for the block. Code snippet 12 shows the implementation of the “destroy()” method.

```
int destroy() {
  switch (_tier) {
    case BlockTier.grey:
      return 0;
    case BlockTier.blue:
      _onDestroy(this);
      return 1;
    case BlockTier.purple:
      _tier = BlockTier.blue;
      return 2;
    case BlockTier.pink:
      _tier = BlockTier.purple;
      return 3;
    case BlockTier.golden:
      _tier = BlockTier.pink;
      return 4;
  }
}
```

Code snippet 12: "destroy()" method inside the "Block" class (source: https://github.com/daviddomkar/block_breaker/blob/main/lib/game/entities/block.dart#L64)

When the tier of the block is “BlockTier.grey”, nothing happens and zero score is returned. If the block is “BlockTier.blue”, which is the lowest tier of the block, the block calls the “_onDestroy()” callback with the current block instance and returns score value of 1. Collision with the block with any other tier lowers the tier and returns the score value for that tier. This means that if the block tier is “BlockTier.golden” it will take 4 hits with the ball to destroy it and the player earns 10 score points in total.

The “_onDestroy()” callback is assigned by the “BlockBreakerGame” class during “Block” class construction. And it allows executing custom code when the block is destroyed. The “BlockBreakerGame” class stores all blocks inside an attribute of type “List<Block>” called “_blocks”. Blocks from this collection are rendered in every frame. Additionally, the class also has another attribute of type “List<Block>” called “_blocksToDispose”. When the “_onDestroy()” callback is called, the “BlockBreakerGame” class forwards the block to be destroyed into “_scheduleBlockDisposal()” method which will add the target block into the

"_blocksToDispose" collection. The update loop then goes through this collection at the beginning of every iteration and calls dispose method for every block inside this collection while also removing the block from the "_blocks" collection. This ensures that the individual block's physical body is removed from the physics world and that is no longer being rendered. Code snippet 13 shows this logic.

```
void _scheduleBlockDisposal(Block block) {
  _blocksToDispose.add(block);
}

@override
void update(double dt) {
  // ...
  for (var block in _blocksToDispose) {
    _blocks.remove(block);
    block.dispose();
  }
  // ...
}
```

Code snippet 13: Block disposal logic inside "BlockBreakerGame" class (source: https://github.com/daviddomkar/block_breaker/blob/main/lib/game/block_breaker_game.dart#L144)

The reason why the block disposal and removal are not done inside „_scheduleBlockDisposal()" method directly is because the „destroy()" method of the block is called from the collision detection code which prevents deleting physical bodies while running. Due to this limitation, it is required to wait until all physics code is processed before deleting the physical body. When all blocks except the ones with the grey tier are destroyed, the win condition is triggered. This is checked in every iteration of the update loop.

5.4.6 Collision Detection

Collision detection of entities is performed by assigning "ContactListener" to the "ContactManager" of the physics world. To create a custom "ContactListener" the "ContactListener" abstract class must be extended. In this case, the descendant class is called "BlockBreakerContactListener" which overrides the "endContact()" method. The "endContact()" method is called when the contact between two fixtures inside physics world ends. To get the two fixtures the contact object can be

used. The method accesses “userData” from the body of each fixture. User data can be any object that is specified on body creation. In this case, when creating the block, the instance to that block is passed into physical body as user data using “this” keyword, same is true for the ball. Code snippet 14 shows an example of such code.

```
final bodyDef = BodyDef(
  userData: this,
  position: Vector2(x * 0.1, y * 0.1),
  type: BodyType.dynamic,
);
```

Code snippet 14: Usage of "userData" parameter inside body definition (source: https://github.com/daviddomkar/block_breaker/blob/main/lib/game/entities/ball.dart#L27)

The “endContact()” method then uses the “userData” to check its type and if one of the bodies is a “Block” and second is the ”Ball”, the “destroy()” method of the block is called, possibly destroying the block and returning score value to the “_onBlockDestroyed()” callback which is assigned during “BlockBreakerContactListener” construction and is responsible for updating the score attribute of the “BlockBreakerGame” class as well as calling “notifyListeners()”, signaling to the UI that the score was updated so it can be rebuilt. Code snippet 15 shows the implementation of the “endContact()” method.

```
@override
void endContact(Contact contact) {
  super.endContact(contact);

  final userDataA = contact.fixtureA.body.userData;
  final userDataB = contact.fixtureB.body.userData;

  if (userDataA is Ball && userDataB is Block) {
    _onBlockDestroyed(userDataB.destroy());
  } else if (userDataA is Block && userDataB is Ball) {
    _onBlockDestroyed(userDataA.destroy());
  }
}
```

Code snippet 15: "endContact()" method inside custom contact listener (source: https://github.com/daviddomkar/block_breaker/blob/main/lib/game/block_breaker_game.dart#L360)

5.4.7 Levels

The game has 10 levels in total. Each level is a collection of blocks at specified coordinates. Because levels are always rectangular, a two-dimensional array can be used to represent them with the type of each field being the tier of the block. To make this even simpler the indexability of Dart enum can be used to only specify the index of the target block tier, which is just an integer number. The -1 index is designated as empty space in the level. The target data structure in Dart would then be “List<List<int>>”. The full “Level” class also contains with and height to validate the input array with asserts. With this class, levels can be easily created in the code using simple syntax. They are all contained in a read-only global levels collection from which they are accessed by other parts of the project. To make formatting nicer, the “E” constant is introduced as an alias for “-1” value which only occupies 1 character. Code snippet 16 shows the definition of the first level.

```dart
// This is used to represent an empty cell in the level data.
// It being one letter makes it easier to read the level data.
const int E = -1;

// Level 1
Level(
  width: 5,
  height: 7,
  data: [
    [E, E, E, E, E],
    [E, E, E, E, E],
    [E, E, E, E, E],
    [4, 4, 4, 4, 4],
    [3, 3, 3, 3, 3],
    [2, 2, 2, 2, 2],
    [1, 1, 1, 1, 1],
  ],
);
```

Code snippet 16: Level definition using the "Level" class (source: https://github.com/daviddomkar/block_breaker/blob/main/lib/game/level.dart#L22)

To load level into the game, “BlockBreakerGame” class includes “loadLevel()” method which accepts “Level” class instance as parameter. It first disposes all the blocks and clears the “_blocks” collection. Then it goes through the level data and creates a “Block” class instance for every element which is not empty constant. It

derives “BlockTier” from the element value and correctly positions the block based on the iteration indexes. It then adds this “Block” instance to the “_blocks” collection. The method itself is mainly composed of two nested loops. Code snippet 17 shows the code for this logic. The magic number “64” is the design offset of the blocks from the top of the board inner bounds.

```
for (int i = 0; i < level.height; i++) {
  for (int j = 0; j < level.width; j++) {
    if (level.data[i][j] == E) {
      continue;
    }

    final block = Block(
      // ...
      tier: BlockTier.values[level.data[i][j]],
      x: (level.width.isEven ? kBlockSize.width / 2 : 0) +
          kBlockSize.width * (j - (level.width ~/ 2)),
      y: _board.innerBounds.top +
          kBlockSize.height / 2 +
          64 +
          kBlockSize.height * i,
    );

    _blocks.add(block);
  }
}
```

Code snippet 17: Level loading algorithm (source: https://github.com/daviddomkar/block_breaker/blob/main/lib/game/block_breaker_game.dart#L202)

5.4.8 Moving Grid Shader

The game background is animated rectangle grid. This is achieved by using a custom fragment shader which repeats the texture of one grid cell over the whole viewport and offsets its position using time uniform, delivering the animated effect. Doing this inside custom shader has performance advantages since the code is executed on the GPU. On the CPU repeating the texture would require loops and multiple draw calls of the same texture which is not ideal by comparison.

Custom fragment shaders in Flutter are written using subset of GLSL language compatible with SkSL – custom shading language used by Skia rendering library which powers Flutter. This is likely to be changed in the future when the

Impeller rendering backend will be implemented on all Flutter supported platforms. However, the limited GLSL support Flutter currently provides is enough to create the effect the game is aiming for [24]. The implementation of the shader in GLSL is shown in Code snippet 18 (some parts have been omitted for clarity).

```
uniform float uNumberOfCells;
uniform float uWidth;
uniform float uTime;
uniform sampler2D uTexture;

void main() {
  vec2 uv = fract(FlutterFragCoord().xy / uWidth * uNumberOfCells *
vec2(-1, 1) + vec2(uTime * 0.25));
  fragColor = texture(uTexture, uv);
}
```

Code snippet 18: Moving grid shader (source: https://github.com/daviddomkar/block_breaker/blob/main/assets/shaders/grid.frag)

The shader accepts several uniforms which are used to calculate the UV coordinates of the texture, such as number of cells, width, and time. To access the local coordinates of the pixel inside the widget space a special "FlutterFragCoord()" function is used which is imported from Flutter. Additionally, a "sampler2D" type uniform is provided for the target texture.

This shader code is placed inside the "grid.frag" file in the "shaders" directory of the "assets" directory from where it is loaded by "AssetManager" as "FragmentProgram" when the app is started. During the initialization of the game the "FragmentShader" instance can be created from "FragmentProgram". "FragmentShader" can live throughout the lifetime of the game instance and is disposed of inside the "dispose()" method of the "BlockBreakerGame" class together with other resources. The shader instance can be used to provide uniforms to the shader code as well as rendering the shader to the screen using "Canvas" and "Paint" object. The "setImageSampler()" and "setFloat()" methods on the "FragmentShader" instance are used for providing values to the shader uniforms. The first argument is the index of the target uniform type and the second is the value of the uniform. The drawing of the shader itself is then performed inside the "render()" method as a first draw call in the frame using "canvas.drawPaint()" method, ensuring it is behind every other later drawn object.

5.5 User Interface

This part of the thesis focuses on the user interface layer of the block breaker game. Unlike other layers, this layer is discussed only briefly because it does not directly contribute to the topic of game development in Flutter and is more focused on standard user interface building principles within the Flutter framework. However, the UI layer is somewhat intricate, interacting with the game layer in various ways while still being reactive thanks to Flutter's design. This is an advantage over user interface systems in various game engines or libraries which are usually imperative and hard to manage when building a complex user interface. Using Flutter for UI on the other hand is much more developer friendly because the framework is made for building user interfaces. Advanced use cases like post-processing effects on the UI are also supported with custom shaders.

5.5.1 Composition

The user interface is composed of widgets. The main widget of the app is the "BlockBreakerApp" widget, which is displayed right after calling "runApp()" method from the entry point of the program – the "main()" function. The "BlockBreakerApp" widget setups Flutter's "MaterialApp" widget using "MaterialApp.router()" constructor. This constructor allows the app to setup routing using a custom router package which is explained in more detail later. Additionally, a „builder" constructor parameter is used to place widgets above the router, creating a scaffold for all the screens managed by the router. Such scaffold is composed of Flutter's „Scaffold" widget, "AnimatedSampler" for applying post processing using a shader on the whole user interface and the „Stack" widget which is responsible for stacking child widgets on top of each other. Thanks to this the "GameWidget" with the "BlockBreakerGame" instance can be placed behind all other widgets. The "BlockBreakerGame" instance is created inside the "initState()" method of the main widget's state and is also an attribute of the state class. From there it can be passed to any other widget requiring control of the game instance. Furthermore, the "SizedBox" widget and "FittedBox" widget together with the information about screen size from the "MediaQuery.sizeOf(context)" call are used to setup correct scaling of the inner router to match the underlying game's

viewport size using the “kViewportSize” constant. Thanks to this, the whole app is responsive, and the widgets align with the game objects inside the game layer.

5.5.2 Routing

The app uses routing for navigating the user to different screens inside the app. Flutter’s routing system is based around the Router API and includes deep linking functionality, history tracking and the address bar URL synchronization when the app is run inside the web browser. Flutter also has Navigator API, which is used to navigate to different screens inside the app or for showing dialogs without the full routing functionalities. The two APIs differ, while Router API is mostly declarative and complex, the Navigator API uses imperative approach and is best suited for small applications without the need for routing capabilities. However, these APIs can also be used together, using the Router API for declaring main application routes and navigating between them while using Navigator API for dialogs and other navigation within the current route [25].

Since working with the Router API directly can be cumbersome because of its complexity, there are several packages abstracting the complexities away and providing simpler interface for declaration of routes and navigation. One of such packages is “go_router” which is officially endorsed by Flutter and mentioned in their documentation. Another popular option is “auto_route”, which heavily relies on codegen to generate application routes. For this project, the “go_router” package is used because it provides easy declaration of routes with imperative navigation while supporting deep linking and URL synchronization on the web.

The router class from the “go_router” package is called “GoRouter” and is created inside the “initState()” method of the “BlockBreakerApp” widget’ state class. Its constructor parameter “routes” accepts a list of “GoRoute” objects. Each “GoRoute” is composed of the route path string and the widget builder. As an example, the route path can look like “/” or “/levels/:index”, with the parts of the path starting with a colon being dynamic parameters. The widget builder is a function returning a widget representing the target route, such as “HomeScreen” or “LevelScreen”. The “GoRouter” constructor can also accept “observers” parameter, which is a list of router observers that can react to routing changes, this

is used in several app screens to modify game state in response to a screen being pushed or being popped to.

5.5.3 Screen widgets

The app is composed of several screens. Each screen has its own widget which is configured as a route inside the app's router. The app is composed of three screens – home screen, level selection screen and level screen.

The home screen contains a play button and app logo and is the initial screen the player sees when they run the app. By pressing the play button, users can navigate themselves to the level selection screen which is pushed onto the page stack.

The level selection screen is a screen where the user can select one of ten levels to play. The screen uses the "PageView" widget to navigate between different levels by swiping on the screen or by using arrow buttons on both sides of the screen. Additionally, the screen contains the play button for starting the level. When navigating between different levels, the screen uses "onPageChanged" callback parameter of the "PageView" to select the target level from the list of levels and load it into the game, rendering the level preview inside the "GameWidget" behind. When some other screen pops to this screen, the screen resets the game to "GameState.notStarted" and loads the last selected level. For the player to play the level, it needs to be unlocked first. This can be done by finishing the previous level. The first level is always unlocked. Locked levels do not have the play button and do not display their preview. Instead, there is a text informing the player to unlock the level first by completing the previous level. The app uses "shared_preferences" package to store information about unlocked levels and their high score between app restarts so the player can continue where they left off next time they play the game. "shared_preferences" package is using different storage mechanisms on each platform to achieve data persistence. For example, local storage on the web, shared preferences on android, or windows registry on windows.

The level screen is the final and most important screen of the app. This screen manages the game being played, displaying information and handling game

states. When this screen is navigated to it loads the selected level passed as a parameter to its constructor from the level selection screen using the router configuration. Because the game instance extends “ChangeNotifier” class, the level screen can call “addListener()” method on the game instance, passing in its listener for changes in game’s state. Level screen uses this listener to know when the game changed its state to “GameState.won” to unlock the next playable level. The screen also uses “ListenableBuilder” widget to listen to changes in game state inside its “build()” method. This is useful to conditionally rebuild the widget when the game state changes. When the state is “GameState.paused”, the level screen builds widgets forming a user interface of a paused game with the options to resume or exit the game represented by buttons. When the state is “GameState.won”, it builds widgets with the information that the player won, how much score they had and a button to go to the next level. Similar happens for “GameState.gameOver”. On “GameState.playing” or “GameState.ready” state, the screen passes through pointer events for the underlying game and displays a HUD at the top of the screen, informing the player what score and how many lives they have. The level screen also uses the “KeyboardListener” widget to listen for the escape key which calls “pause()” or “resume()” methods on the game instance to quickly pause or resume the game.

5.5.4 Post-processing Glow Shader

As mentioned earlier, the scaffolding of the user interface includes “AnimatedSampler” widget. This widget comes from the “flutter_shaders” package and provides a way to apply shader postprocessing effects to child widget tree. It works by rendering the child widgets into an image instead of the screen and then providing this image together with its size and the canvas instance inside “builder” callback parameter. This image can then be rendered to the screen using the provided canvas to display the child widget tree. Additionally, canvas can be used to layer the image together with other layers using blend modes while other layers can use shaders to paint themselves.

Using these techniques, a glow effect is applied to the whole user interface. It works by using a custom shader. This shader accepts the image of the child widget

tree as a “sampler2d” uniform, image size as “vec2” uniform and threshold together with intensity as “float” uniforms. The size is used to find the UV coordinate for a target pixel inside the image and then retrieve the color of that picture. This color is then used to compute luma value of the target pixel using a standard formula. When the luma value of a pixel is smaller than the provided threshold, pixel is erased (all values are 0, including alpha), otherwise the color is multiplied by the provided intensity uniform and preserved. Code snippet 19 shows shader’s main method.

```
void main() {
  vec2 uv = FlutterFragCoord().xy / uSize;
  vec4 color = texture(uTexture, uv);

  float luma = 0.212 * color.r + 0.701 * color.g + 0.087 * color.b;

  if (luma < uTreshold) {
    fragColor = vec4(0);
  } else {
    fragColor = color * uIntensity;
  }
}
```

Code snippet 19: Luma shader (source: https://github.com/daviddomkar/block_breaker/blob/main/assets/shaders/luma.frag)

This shader is then rendered into a separate canvas layer creating a texture with only pixels that should glow. But to sell the effect even more, the “ImageFilter.blur” is applied as an “imageFilter” to this layer. This image filter creates a gaussian blur so the glowing pixels bleed outside based on the sigma value of the blur. The layer is then restored to be applied over the original image of the widget tree using the “BlendMode.plus” additive blend mode, which combines the two layers, creating the final glow effect.

5.6 Deployment

Flutter is a framework for building cross platform applications. Because of it, the final application can be deployed to a wide range of platforms. At the time of writing, these are Android, iOS, macOS, Linux, Windows, and the web. Flutter officially supports these platforms as stable targets. This thesis focuses in detail on deployment on Android, iOS, Web, and Windows. The thesis does not go over

details regarding the setup of distribution services, such as Windows Store, AppStore or Google Play as it is outside the scope of the Flutter framework and the process should be the same as for any non-Flutter app.

Flutter offers CLI tools and IDE integrations to automate the build process which should work out of the box on any platform assuming all required tools are installed. To check this, flutter provides “flutter doctor” command, which verifies the development environment. Figure 8 shows the example output of this command on a Windows machine.

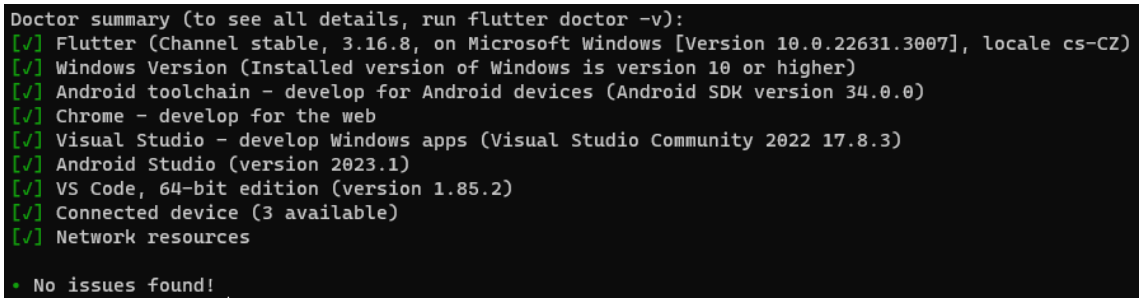

Figure 8: "flutter doctor" output
(source: own work)

The output shows that the development environment is ready for development and deployment on Android, Web, and Windows since the Android toolchain, Chrome and Visual Studio are installed with required components. Unfortunately, there are some platform restrictions that prevent deployment for certain platforms on other platforms. One such example is iOS which can be deployed only on a macOS device because there is no tooling available for other platforms. However, the platform code and the Flutter code of the app can still be edited anywhere. Just the final deployment needs to happen on a platform that supports it. Because of this limitation, CI/CD services supporting Flutter are usually running remote builds on macOS devices.

One other requirement for the deployment of the app on any platform is that the Flutter project needs to be configured for all the target platforms. When creating a new project, Flutter automatically configures all stable platforms and puts their platform code to their specific folders within the project, such folders are named after the target platform – “android”, “ios”, “macos”, “linux”, “windows”, and “web”. Developers can then delete folders for platforms which the application

does not target or create a project with the target platforms specified as arguments, for example by running: “flutter create --platforms=android,windows”. Target platforms can also be enabled or disabled globally for flutter by using “flutter config --enable-<platform>” or “flutter config --no–enable-<platform>” commands. When a new platform is enabled which was not enabled previously the command “flutter create .” must be run inside the project’s directory. Flutter CLI figures out what platforms are missing and creates folders for them [26].

5.6.1 Android Deployment

On Android, there are multiple steps developers need to take before building the final application for distribution. The first step is to change the package name. Package name is a unique app identifier in a form of reversed domain. By default, if a custom package name is not specified using command line arguments, Flutter generates one on project creation. It has the format of “com.example.<project_name>” which is not ideal for distribution. To change a package name completely, it must be edited in multiple places. First and most importantly in “android/app/build.gradle” file in “android.defaultConfig.applicationId” field and the “android.namespace” field. Then it must be also changed inside the “MainActivity.kt” file in “android/app/src/main/kotlin/com/example/<project_name>” directory inside package declaration. Finally, the specified path to the main activity class file needs to be edited to adhere to the new package name.

The next step is to review the app manifest which can be found in “AndroidManifest.xml” file. Manifest should contain all permission declarations used by the app. Since the block breaker app does not use any, this can be skipped. An additional thing to keep in mind is the name of the app which can be modified by setting the “android:label” attribute inside “application” object.

To prepare the app for uploading to a distribution service another step needs to be taken and that is the app signing process. To sign the app a key needs to be created. The process is the same as the process of creating a signing key for a normal android app. This can be done using java’s “keytool” utility or by using Android Studio. Once the key is created and saved into “.jks” file it can be used to

create a new signingConfig inside “android/app/build.gradle” which is then specified inside “buildTypes” block. Recommended way to do this is to create a “key.properties” file inside the “android” directory to store password details of the keystore and then reference this file inside the “build.gradle” file. This file, like the keystore, should not be checked into source control. Anyone in possession of the keystore could then upload a new version of the app to the distribution service .

Because the android folder is a normal Android Kotlin project, it can be opened in an IDE such as Android Studio and do additional configuration there or implement any platform code the app needs, however the steps taken so far should be enough to build and distribute the app. Additionally, it may be beneficial to change the default app icon and splash screen. To avoid doing this on every platform separately packages such as “flutter_native_splash” and “flutter_launcher_icons” can be used to simplify the process.

With the android platform configured, all that remains is to build the app to APK or App Bundle which are formats used for android app distribution to a distribution service such as Google Play. To do this a command must be executed inside the project’s directory. This command is either “flutter build apk” or “flutter build appbundle”, depending on the format. The outputs of these commands can then be found inside “build/app/outputs” directory, and the files can be used to distribute the app to the distribution service [27].

5.6.2 iOS Deployment

Building and releasing the app on the iOS platform requires Xcode software, which is only available on macOS devices, because of this a macOS device is needed for deployment. Additionally, the only distribution method on iOS at the time of writing is through Apple’s AppStore platform. Which means that the app needs to adhere to Apple’s App Review Guidelines and needs to have registered bundle id and application record at App Store Connect.

To configure the app for deployment it is best to use the Xcode UI by opening “ios/Runner.xcworkspace” Xcode workspace. However, some steps can be performed by directly editing the “project.pbxproj” project file inside the “ios/Runner.xcodeproj” folder. The most important thing inside Xcode is to set the

bundle identifier field to the corresponding bundle id registered at App Store Connect and select the App Store Connect team associated with the Apple Developer Account. Furthermore, Xcode can be used to set application name, manually manage signing certificates, selecting a target iOS version, or adding custom app icon and splash screen. To add the app icon and splash screen, Flutter packages such as “flutter_native_splash” and “flutter_launcher_icons” can be used to simplify the process like in the case of Android.

To create an app bundle for distribution a “flutter build ipa” command is used. This command can be extended with options like “--obsfuscate” and “--split-debug-info” to make it harder to reverse engineer. Then the bundle can be validated and uploaded to App Store Connect using Xcode for distribution. Developers can then choose to release the app either on TestFlight for testing or for production on AppStore [28].

5.6.3 Web Deployment

To deploy the Flutter app on the web no configuration is required. The only thing that needs to happen is the “flutter build web” command. The output is saved into the “build/web” directory from which it can be served using a webserver or uploaded to a static hosting, such as Firebase Hosting or GitHub Pages. However, configuration can be supplied to have control over the final output [29].

When building the app for the web one of the two different renderers can be chosen – HTML renderer or Canvaskit renderer. HTML renderer uses HTML elements, CSS styles, canvas elements and SVG elements to render the app. HTML renderer is slower but has a smaller download size. On the other hand, the Canvaskit renderer uses WebGL to render the app, has a bigger download size but is faster when rendering lots of widgets. It is also more consistent with rendering on other platforms since it uses the same Skia API under the hood with desktop and mobile implementations and has better support for shaders. Because of this, the app developed in this thesis uses the Canvaskit renderer. To choose renderer a “--web-renderer” option must be provided to “flutter build web” command. The value of this option can be “canvaskit”, “html” or “auto”. The first two values are self-explanatory. The “auto” value uses “html” renderer on mobile browsers and

“canvaskit” on desktop browsers. Using this option, the renderer can also be configured at runtime. The “auto” is the default value if the “--web-renderer” option is not provided [30].

Another configuration that can be made on the web is overriding how the Flutter app is embedded into a website. By default, Flutter uses “index.html” file inside the web directory of the project. This file contains a script in JavaScript language used for initializing the Flutter engine and bootstrapping the app. This script and the html in the file can be modified to facilitate advanced functionality, such as displaying a loading screen or modifying the embedding configuration. By default, when the application is loaded and displayed Flutter overrides anything inside the body element with its markup. This can be changed by specifying a custom HTML element to the “hostElement” parameter of the “initializeEngine()” method. Flutter then mounts to that HTML element instead of the whole body. Another alternative is to include Flutter web app as an iframe inside another webpage [29].

Flutter web app can also function as a PWA (Progressive Web App). The only thing that is needed for that functionality is the “manifest.json” file inside the web folder. This file can be used to configure the PWA behavior and is generated by default by Flutter when creating the project. PWA allows the app to be installed to the device and work offline, if configured correctly [29].

5.6.4 Windows Deployment

To deploy the app on windows, the simplest way is to generate “.exe” file which can be run directly. To do so a command “flutter build windows” needs to be executed. The resulting “.exe” application can then be found inside the “build\windows\x64\runner\Release\<project_name>.exe” directory. To customize windows build, files inside “windows” folder inside the project’s directory can be modified. One such file is the “windows\runner\Runner.rc” file where the product name or company name can be changed. There is also “windows\runner\main.cpp” file which is a C++ file that contains the main entry point of the Win32 application and creates a “FlutterWindow” object. The origin, size and label properties of the window can be configured there. Additionally, a

custom icon can be configured by changing the "windows\runner\resources\app_icon.ico" icon file [31].

Flutter also provides functionality for building windows package formats used to publish the application to the Microsoft Store. The easiest and the recommended way to do so is using the "msix" pub package [31].

6 Results

This section of the thesis provides results in accordance with defined objectives and answers research questions (RQ).

RQ1: How does Flutter's architecture support the development of multiplatform games?

Answer:

Flutter's architecture supports multiplatform game development with a structured approach that facilitates code reusability and platform-specific services, ensuring games run efficiently across different platforms. Critical layers include the Embedder, Engine, and Framework, each providing unique functionalities like platform-specific operations and core API implementations. Notably, Flutter's ability to render smoothly at high frame rates, up to 120 frames per second, is particularly beneficial for games, ensuring responsive and fluid gameplay. This performance is also attributed to Dart's optimization for fast object instantiation and deletion.

RQ2: How can game development aspects like input processing, physics simulation, and rendering be leveraged within the Flutter framework?

Answer:

Flutter provides built-in ways to handle many of the game development aspects including input processing and rendering. However, to fully utilize them in the context of game development it is required to adapt them to a specific game's requirements and features using a lower-level engine layer. While being more explicit, it allows developers to precisely manage how exactly the game is visually represented on the screen and how the input data gets processed. Some other aspects of game development, like physics simulation can be integrated into Flutter using third-party libraries, such as "forge2d".

RQ3: What strategies are employed to manage the layering of UI elements and game objects in Flutter?

Answer:

While every UI in a Flutter app is represented using a “Widget” object, the game objects need more flexibility and are best rendered within one main game widget. To layer the UI elements over the game widget the “Stack” built-in Flutter widget can be used. Moreover, to align layout and responsive behavior, the UI layer can use layout widgets such as “FittedBox” or “SizedBox” to match the scaling behavior of the underlying game widget by using constant values such as viewport width and height.

RQ4: How does the Flutter framework enable use of special post-processing effects in multiplatform games?

Answer:

Flutter applications can leverage the power of Skia or Impeller rendering backends to implement custom shaders using GLSL shading language. These shaders can be used in combination with “AnimatedSampler” widget from the “flutter_shaders” package to be applied over the portion of the UI wrapped in “AnimatedSampler” widget allowing for real time post processing effects.

RQ5: What are the deployment options for multiplatform games made using the Flutter framework?

Answer:

Flutter offers extensive deployment options for multiplatform games, enabling distribution across Android, iOS, macOS, Linux, Windows, and web platforms with straightforward build processes. Utilizing CLI tools and IDE integrations, developers can automate builds for these platforms, ensuring easy deployment. Key steps include configuring platform-specific settings like package names and app permissions, with unique requirements for each target, such as using Xcode for iOS deployment. The framework's support for a wide range of platforms, combined with efficient build tools, makes Flutter a powerful choice for deploying multiplatform games to a broad audience.

7 Conclusions and Recommendations

In conclusion, this thesis investigated the potential of the Flutter framework for multiplatform game development, covering both theoretical aspects and practical application through the development of a simple game. This comprehensive analysis aimed to assess Flutter's capabilities and limitations as a tool for game developers targeting multiple platforms.

Theoretical part focused on explanation of the Flutter's architecture, emphasizing its widget-based design, the Dart programming language, and features like hot reload and development tools. These insights established a foundation for understanding how Flutter operates and its potential benefits for game development. The practical part of the thesis demonstrated Flutter's ability to handle many game development aspects such as rendering, physics simulation, and input handling through the development of an actual game.

The findings from this thesis confirm that Flutter is a promising framework for developing certain types of multiplatform games, particularly those with simple 2D graphics and strong emphasis on UI. Flutter's single codebase approach offers a streamlined development process and ensures consistent user experience across mobile, desktop, and web platforms. The deployment on those platforms is also very straightforward thanks to Flutter's documentation. On the other hand, the research also identified areas where Flutter falls short, mainly in missing tooling for developing games directly within the framework. Game developers often need to introduce an additional engine layer which facilitates input processing, viewport, rendering, game loop, and other aspects directly on top of the Flutter's low-level APIs. This thesis demonstrates how to do so using an example game application. However, developers can also use existing packages to mitigate this limitation. Other limitations include Flutter's inability to currently render 3D graphics and support other advanced game development features making it not suitable for development of large scale and complex AAA games.

Based on these observations, it is recommended for future research to focus on improving the framework with more game development capabilities. One such improvement could be the new in-development Impeller rendering backend based

on modern graphics APIs that could add support for 3D graphics and improve support for custom shaders which currently have limited subset of features. Additionally, a development of Flutter-specific game engines which, given the necessity of an additional engine layer, could be a significant opportunity to integrate with existing Flutter architecture and offer out-of-the-box solutions for game development aspects tailored to Flutter's environment. Furthermore, an exploration of advanced game development features within Flutter by building a more comprehensive game with emphasis on particle systems, animations, audio processing or support for VR and AR experiences could reveal more shortcomings and opportunities for improvement for Flutter regarding game development.

Ultimately, the journey of exploring Flutter's capabilities in game development does not conclude with this thesis; rather, it marks the beginning of an ongoing exploration into how this versatile framework can be pushed to new heights. The future of game development with Flutter is bright, and it holds the promise of unlocking new possibilities that will enrich the gaming world for developers and players alike. As the Flutter ecosystem grows and adapts, it will undoubtedly open new horizons for innovation in multiplatform game development.

Bibliography

[1] *Cross-platform mobile frameworks used by global developers 2022*. Online. Statista. Available from: https://www.statista.com/statistics/869224/worldwide-software-developer-working-hours/. [viewed 2023-08-15].

[2] *Sky: An Experiment Writing Dart for Mobile (Dart Developer Summit 2015)*. Online. 2015. Available from: https://www.youtube.com/watch?v=PnIWl33YMwA.

[3] *Flutter 1.0: Google's Portable UI Toolkit*. Online. Available from: https://developers.googleblog.com/2018/12/flutter-10-googles-portable-ui-toolkit.html. [viewed 2023-08-15].

[4] *Stack Overflow Trends*. Online. Available from: https://insights.stackoverflow.com/trends?tags=flutter%2Creact-native. [viewed 2023-08-15].

[5] *Fuchsia System UI*. Online. Available from: https://fuchsia.googlesource.com/topaz/+/refs/heads/milestone-1011/shell/README.md. [viewed 2023-08-15].

[6] *Flutter architectural overview*. Online. Available from: https://docs.flutter.dev/resources/architectural-overview. [viewed 2023-08-15].

[7] *Dart programming language*. Online. Available from: https://dart.dev/. [viewed 2023-10-20].

[8] *Patterns*. Online. Available from: https://dart.dev/language/patterns.html. [viewed 2023-10-21].

[9] *Dart overview*. Online. Available from: https://dart.dev/overview.html. [viewed 2023-10-21].

[10] *Introduction to Dart*. Online. Available from: https://dart.dev/language/. [viewed 2023-10-21].

[11] *Exception class - dart:core library - Dart API*. Online. Available from: https://api.dart.dev/stable/3.1.4/dart-core/Exception-class.html. [viewed 2023-10-21].

[12] *Error class - dart:core library - Dart API*. Online. Available from: https://api.dart.dev/stable/3.1.4/dart-core/Error-class.html. [viewed 2023-10-21].

[13] *Error handling*. Online. Available from: https://dart.dev/language/error-handling.html. [viewed 2023-10-21].

[14] *Hot reload*. Online. Available from: https://docs.flutter.dev/tools/hot-reload. [viewed 2023-10-21].

[15] *DevTools*. Online. Available from: https://docs.flutter.dev/tools/devtools/overview. [viewed 2023-10-22].

[16] *The official repository for Dart and Flutter packages.* Online. Dart packages. Available from: https://pub.dev/. [viewed 2024-01-06].

[17] *How to build a RenderObject*. Online. 2023. Available from: https://www.youtube.com/watch?v=cq34RWXegM8.

[18] *flame/packages/flame/lib/src/game/game_render_box.dart at main · flame-engine/flame*. Online. Available from: https://github.com/flame-engine/flame/blob/main/packages/flame/lib/src/game/game_render_box.dart. [viewed 2023-11-18].

[19] *computeDryLayout method - RenderBox class - rendering library - Dart API*. Online. Available from: https://api.flutter.dev/flutter/rendering/RenderBox/computeDryLayout.html. [viewed 2024-01-07].

[20] *sizedByParent property - RenderObject class - rendering library - Dart API*. Online. Available from: https://api.flutter.dev/flutter/rendering/RenderObject/sizedByParent.html. [viewed 2024-01-07].

[21] *forge2d | Dart Package.* Online. Dart packages. Available from: https://pub.dev/packages/forge2d. [viewed 2024-01-07].

[22] *Box2D: Overview*. Online. Available from: https://box2d.org/documentation/index.html#autotoc_md4. [viewed 2024-01-07].

[23] *Box2D: Dynamics Module*. Online. Available from: https://box2d.org/documentation/md__d_1__git_hub_box2d_docs_dynamics.html#autotoc_md52. [viewed 2024-01-07].

[24] *Writing and using fragment shaders*. Online. Available from: https://docs.flutter.dev/ui/design/graphics/fragment-shaders. [viewed 2024-01-07].

[25] *Navigation and routing*. Online. Available from: https://docs.flutter.dev/ui/navigation. [viewed 2024-01-23].

[26] *Desktop support for Flutter*. Online. Available from: https://docs.flutter.dev/platform-integration/desktop. [viewed 2024-01-29].

[27] *Build and release an Android app*. Online. Available from: https://docs.flutter.dev/deployment/android. [viewed 2024-01-29].

[28] *Build and release an iOS app*. Online. Available from: https://docs.flutter.dev/deployment/ios. [viewed 2024-01-29].

[29] *Build and release a web app*. Online. Available from: https://docs.flutter.dev/deployment/web. [viewed 2024-01-29].

[30] *Web renderers*. Online. Available from: https://docs.flutter.dev/platform-integration/web/renderers. [viewed 2024-01-29].

[31] *Building Windows apps with Flutter*. Online. Available from: https://docs.flutter.dev/platform-integration/windows/building. [viewed 2024-01-29].

Univerzita Hradec Králové
Fakulta informatiky a managementu

Zadání bakalářské práce

Autor: **David Domkář**

Studium: I2100193

Studijní program: B1802 Aplikovaná informatika

Studijní obor: Aplikovaná informatika

Název bakalářské práce: **Multiplatformní vývoj her ve frameworku Flutter**

Název bakalářské práce AJ: Multiplatform game development using the Flutter framework

Cíl, metody, literatura, předpoklady:

The goal of the thesis is to explore the possibilities of multiplatform game development using the Flutter framework. The thesis will also include a practical example of a game programmed in Flutter to demonstrate findings from the theoretical part.

Zadávající pracoviště: Katedra informačních technologií,
Fakulta informatiky a managementu

Vedoucí práce: Ing. Milan Kořínek

Datum zadání závěrečné práce: 15.10.2021