

프론트엔드 관점에서의 플러터 기반 퍼즐게임 개발

Development of a Flutter-Based Puzzle Game from
a Front-End Perspective

한국외국어대학교

아랍어과

(AI 융합전공 SW & AI 트랙)

조완기

2024 年 11 月

프론트엔드 관점에서의 플러터 기반 퍼즐게임 개발

Development of a Flutter-Based Puzzle Game from a
Front-End Perspective

위 논문 학사학위 논문으로 제출합니다.

지도교수: 신찬수

2024 년 11 월

대학 : 한국외국어대학교

학과 : 아랍어과


(AI 융합전공 SW & AI 트랙)

학번 : 201903390

이름 : 조완기

조완기의 학사학위 논문을 심사하여

합격으로 판정합니다.

심사위원: 신찬수  인

프론트엔드 관점에서의 플러터 기반 퍼즐게임 개발

Development of a Flutter-Based Puzzle Game from a Front-End Perspective

요약

Abstract

본 논문은 Flutter 기반 퍼즐 게임 개발 과정에서 발생하는 성능 문제를 해결하기 위해 RepaintBoundary와 GetX 상태 관리 도구를 활용한 최적화 방법을 비교 분석하였다. 퍼즐 조각 이동 시 전체 화면을 반복 렌더링하는 문제를 RepaintBoundary를 사용해 개별 조각만 독립적으로 렌더링하도록 하여 성능을 개선하였고, GetX를 활용해 상태 변경 시 필요한 부분만 다시 그리는 방식으로 최적화를 진행하였다. 실험 결과, RepaintBoundary는 PAINT 작업 시간을 99.15% 단축시켰으며, GetX는 PAINT 작업을 97.19% 개선하였다. 두 방법은 활용 조건에 따라 선택적으로 사용될 수 있으며, 본 연구는 이를 통해 모바일 애플리케이션 성능 최적화의 실질적 방안을 제시한다.

목차

1. 서론

- 1.1 논문 주제
- 1.2 주제 선정 동기
- 1.3 목적

2. 시스템 구성

- 2.1 Flutter 프레임워크 개요
- 2.2 퍼즐 게임 User Flow 설계
- 2.3 초기 렌더링 설계

3. RepaintBoundary의 활용과 문제 해결

- 3.1 Flutter 렌더링 파이프라인
- 3.2 RepaintBoundary 도입 배경
- 3.3 적용 방식 및 코드 구현
- 3.4 성능 개선 결과
- 3.5 RepaintBoundary의 부작용

4. GetX의 활용과 문제점

- 4.1 GetX 개요
- 4.2 GetX 적용 및 활용구조
- 4.3 성능 측정 결과
- 4.4 GetX의 한계

5. 결론 및 향후 연구 방향

6. 참고 문헌

1. 서론

1.1 논문 주제

본 논문에서는 Flutter 를 활용하여 개발한 퍼즐 게임의 렌더링 성능을 최적화하는 방법에 대해 다루고자 한다. 퍼즐 게임의 상호작용 과정에서 발생하는 성능 이슈를 해결하기 위해 RepaintBoundary 와 GetX 상태 관리 도구를 활용한 최적화 방안을 비교 분석하며, 각각의 접근법이 렌더링 성능과 사용자 경험에 미치는 영향을 논의한다.

1.2 주제 선정 동기

퍼즐 게임은 조각을 지속적으로 드래그하고 적합한 위치로 이동시키는 상호작용이 핵심인 게임 유형이다. 이러한 상호작용 과정은 모바일 장치의 렌더링 성능에 큰 영향을 미치며, 프론트엔드 에서 부드럽고 안정적인 사용자 경험을 보장하기 위해 최적화된 렌더링이 필수적이다. 본 논문에서는 Flutter 의 렌더링 파이프라인을 활용하여 RepaintBoundary 와 GetX 상태 관리 도구를 각각 적용한 최적화 과정을 설명하고, 이들의 성능 차이를 분석하여 최적화의 필요성과 방향성을 제시하고자 한다.

1.3 목적

이 논문의 주요 목적은 Flutter 기반 퍼즐 게임 개발 과정에서 UI 렌더링 최적화 전략을 설계하고 적용하는 방법을 제시하는 데 있다. RepaintBoundary 와 GetX 를 각각 적용한 사례를 통해 퍼즐 조각 드래그 시 발생하는 렌더링 지연 문제를 해결하고, 상호작용성을 극대화하는 방법을 논의한다. 이를 통해 효율적인 렌더링 구조를 구현하고, 결과적으로 높은 수준의 사용자 경험을 제공하는 최적화된 퍼즐 게임 앱을 개발하는 데 기여하는 것을 목표로 한다.

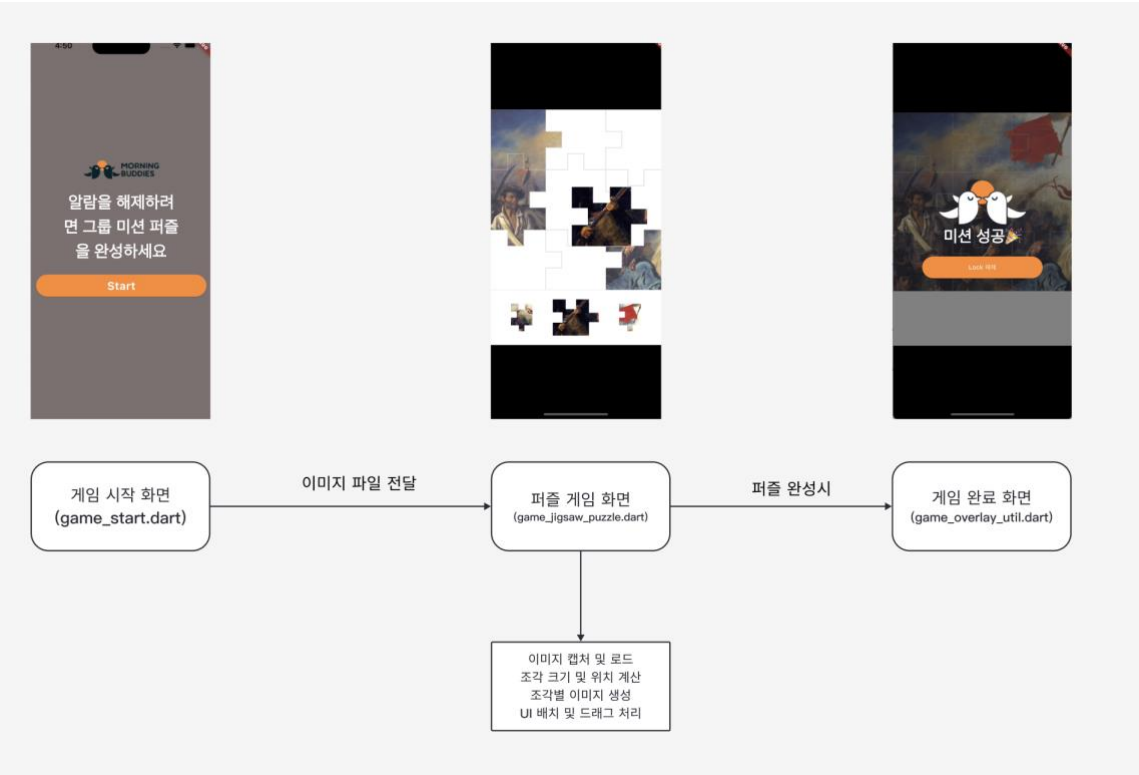
2. 시스템구성

2.1 Flutter 프레임워크 개요

Flutter 는 구글에서 개발한 크로스 플랫폼 앱 개발 프레임워크로, 하나의 코드베이스로 iOS 와 Android, Web 등 다양한 플랫폼에서 동작하는 어플을 개발할 수 있다. Flutter 는 Dart 언어를 기반으로 이루어진 객체 지향 언어로 React Native 와 함께 앱 개발 생태계에서 각광받고 있는 프레임워크 중 하나이다 Flutter 는 네이티브에 가까운

성능과 일관적이고 사용자 지정이 가능한 렌더링, Hot Reload¹를 통해 상태 변경 없이 코드 변경 내용을 미리 확인 할 수 있는 장점을 갖고 있다.

2.2 퍼즐게임 User Flow 설계



[그림 1] 퍼즐게임 UI & Flow Chart

2.2.1 게임시작 화면(game_start.dart)

Start 버튼을 통해 게임을 시작하며, 사용자는 알람을 해제하고 퍼즐을 완성하라는 안내를 받는다.

2.2.2 퍼즐 게임 화면(game_jigsaw_puzzle.dart)

시작화면에서 선택한 이미지 파일이 전달되며, 전달한 이미지를 조각으로 나누어 퍼즐 형태로 로드(load) 한다. 이후 각 퍼즐 조각의 크기 및 위치를 계산하여 화면에 배치, 플레이어는 드래그 앤 드롭을 통해 퍼즐 조각을 맞춘다. 퍼즐 조각은 Carousel 형태로 좌우 Swipe 가 가능한 형태로 주어진다.

¹ Hot Reload: Flutter 의 기능 중 하나로, 가상 머신이 새 버전의 필드 및 함수로 클래스를 업데이트 하여 빠르게 변경사항의 효과를 볼 수 있음

2.2.3 게임 완료 화면(game_overlay_util.dart)

모든 퍼즐 조각이 올바르게 배치되면 callbackFinish 함수를 호출, 게임 완료 화면이 렌더링 된다. 이후 미션 성공 화면이 나타나고 메인 화면으로 돌아갈 수 있는 "Lock 해제" 버튼이 표시된다. 사용자는 이 버튼을 눌러 게임을 종료하고 메인 화면으로 돌아간다.

2.3 초기 렌더링 설계

가장 핵심 로직을 담당하고 있는 퍼즐 게임 화면의 초기 렌더링은 드래그 이벤트를 기준으로 설계하였다. 드래그 이벤트 시작, 드래그 위치 업데이트, 드래그 종료, 퍼즐 조각 위치 검사 및 완료 여부 확인의 총 4 단계로 이루어져 있다.

```
1. Puzzle Initialization:
- 전체 퍼즐 조각 생성 및 초기 위치 설정
- blocksNotifier에 모든 조각 상태 저장

2. Drag Event 시작:
- 사용자가 퍼즐 조각 드래그 시작
- onPanStart 이벤트 발생
- 드래그 중인 조각의 인덱스 (_index) 저장
- 드래그 시작 위치 (_pos) 저장

3. During Drag (드래그 중):
- 사용자가 퍼즐 조각을 이동할 때마다 onPanUpdate 호출
- 모든 조각이 들어있는 blocksNotifier 업데이트
- 모든 퍼즐 조각의 위치가 업데이트됨
- 전체 퍼즐 트리가 다시 빌드됨 (렌더링 지연 발생)

4. Drag Event 종료:
- 사용자가 드래그를 멈추면 onPointerUp 호출
- 현재 조각 위치와 목표 위치 비교
- 목표 위치에 가깝다면 해당 조각 고정 (isDone = true)
- blocksNotifier 업데이트
- 퍼즐 조각 고정 후 전체 퍼즐 트리 다시 빌드됨

5. Puzzle Completion Check:
- 모든 조각이 제자리에 있는지 확인 (checkPuzzleCompletion 호출)
- 완료되면 callbackFinish 호출
- 퍼즐 전체가 완료되었을 때 이벤트 트리거
```

[그림 2] 퍼즐 게임 화면 pseudo code

2.3.1 드래그 이벤트 시작

사용자가 퍼즐 조각을 드래그시, onPanStart 이벤트가 발생하고, 이때 드래그 중인 조각의 인덱스와 드래그 시작 위치가 설정된다.

2.3.2 드래그 위치 업데이트

사용자가 드래그를 할 때마다 onPanUpdate 가 호출되며 퍼즐 조각의 위치가 실시간으로 업데이트 된다. 여기서 드래그 중인 조각뿐만 아니라 전체 blockNotifier 의

상태가 갱신되며, 모든 조각의 상태가 업데이트 되기 때문에 Flutter 의 렌더링 엔진은 전체 퍼즐을 다시 빌드한다.

2.3.3 드래그 종료

사용자가 드래그를 멈추면 onPointerUp 이벤트가 발생한다. 여기서 조각의 위치가 목표 위치에 가까워지면, 해당 조각이 제자리에 고정된다. 모든 조각의 위치가 업데이트된 상태로 유지되므로 Flutter 는 전체 퍼즐을 다시 렌더링한다.

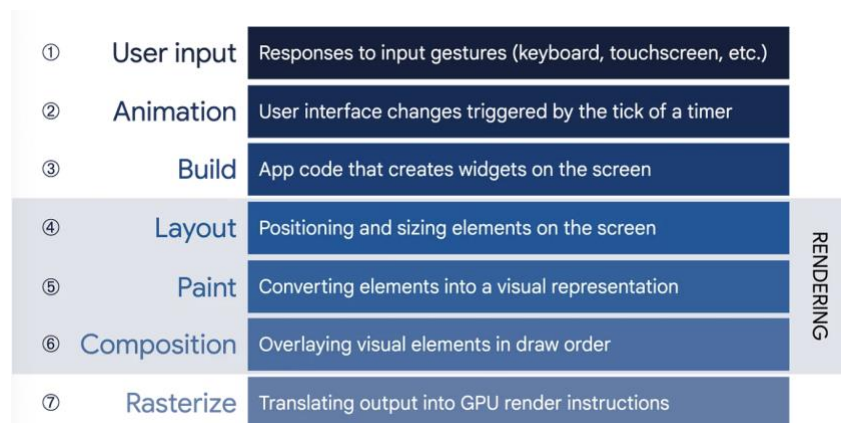
2.2.4 퍼즐 조각 위치 검사 및 완료 여부 확인

checkPuzzleCompletion 함수를 통해 퍼즐이 완성 되었는지 확인한다. 이때 역시 전체 퍼즐이 다시 빌드되며, 리빌드(Re-build) 과정을 진행한다.

3. RepaintBoundary 의 활용과 문제 해결

3.1 Flutter 렌더링 파이프라인

Flutter 의 렌더링 파이프라인을 이해함으로써, 불필요한 빌드와 레이아웃 계산을 피하는데 도움이 될 수 있다. 플러터의 렌더링 아키텍처는 총 7 단계로 구성된다



[그림 3]Flutter 렌더링 아키텍처

그 중 Layout,과 Paint 단계를 주목할 필요가 있다. Layout 단계는 위젯의 크기와 위치를 결정 하는 단계이다. 이때 깊이 우선 탐색(DFS) 방식을 통해 RenderObject 트리에 저장하는 알고리즘을 활용한다. Paint 단계 역시 앞서 Layout 단계서 탐색했던 트리를

깊이 우선 탐색(DFS) 알고리즘을 활용하여 Layer 트리를 생성, 이 Layer 트리는 최종적으로 화면에 표시 될 내용을 포함하게 된다.

이 과정에서 개발자는 RepaintBoundary 매커니즘을 활용해 그래픽 Layer 를 분할, 어느 부분을 그릴지, 말지를 결정할 수 있다. 이를 통해서 자체적 Layer 를 갖게 되며, 독립적으로 렌더링 되는 것이 가능하다.

3.2 RepaintBoundary 도입 배경

Flutter 를 활용한 퍼즐 게임 개발 과정에서 마주한 문제는 퍼즐 조각을 드래그 시 발생하는 지연현상 이다. 원인은 바로 앞 절에서 살펴본 Flutter 의 UI 업데이트 특성 때문이다. Flutter 는 UI 가 변경되면, 다시 위젯을 빌드 하는 구조를 갖고 있다. 즉, 조각을 드래그 할 때 다른 조각들까지 불필요하게 다시 렌더링 하는 상황이 발생했다. 특히 퍼즐 조각의 수가 많아질 수록 이러한 전체 리 렌더링으로 인해 퍼포먼스 저하가 더욱 두드러졌다.

초기 렌더링 설계에서는 모든 조각을 ValueNotifier 를 통해 관리하며, 상태가 변경될 때마다 전체 퍼즐 조각을 Rebuild 하는 방식을 사용하였다. 하지만 이 접근 방식은 조각의 위치를 미세하게 조정하는 드래그 이벤트의 특성으로 인해 CPU 에 많은 부하를 일으켰다. 이로 인해 드래그 동작이 부드럽지 못했고, 지연이 발생하여 사용자 경험에 부정적 영향을 주었다.

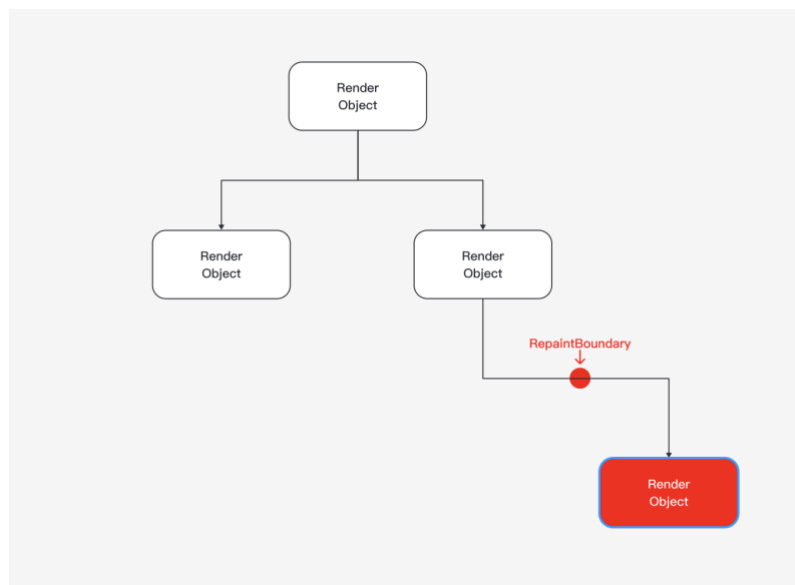
이 문제를 해결하기 위해 RepaintBoundary 를 도입하여 개별 퍼즐 조각이 독립적으로 Repaint 될 수 있도록 최적화 하였다. RepaintBoundary 는 특정 위젯의 Repaint 를 별도로 관리할 수 있게 하는 기능을 수행한다. 즉 이를 통해 조각 하나의 상태가 변경 되더라도 다른 조각이 영향을 받지 않도록 하여, 드래그 중인 퍼즐 조각만 개별적으로 리 렌더링 되며, 전체 퍼즐 게임의 성능을 개선하였다.

3.3 적용방식 및 코드 구현

```
f { (blockNotDone.isNotEmpty)
...blockNotDone.asMap().entries.map{
    (map) {
        return Positioned(
            left: map.value.offset.dx,
            top: map.value.offset.dy,
            child: Offstage(
                offstage: !(_index == map.key),
                child: GestureDetector(
                    onTapStart: (details) {
                        if (map.value.jigsawBlockWidget.imageBox.isDone) {
                            return;
                        }
                        setState(() {
                            _pos = details.localPosition;
                            _index = map.key;
                        });
                    },
                    onTapUpdate: (details) {
                        if (_index != null) {
                            setState(() {
                                blockNotDone[_index!].offset += details.delta;
                            });
                        }
                    },
                    child: Container(
                        child: map.value.jigsawBlockWidget,
                    ),
                ),
            ),
        );
    },
);
},
);
```

[그림 4] RepaintBoundary 적용 전 코드(좌)와 적용 후 코드(우)

초기 렌더링 설계 부분 중 개별 조각 퍼즐을 담는 변수를 담고 있는 Container 위젯을 RepaintBoundary 로 감싸서 구현한다. 이를 통해 드래그 중인 퍼즐 조각만 개별적으로 리렌더링 될 수 있도록 하였다.



[그림 5] Rendering Object Tree 도식

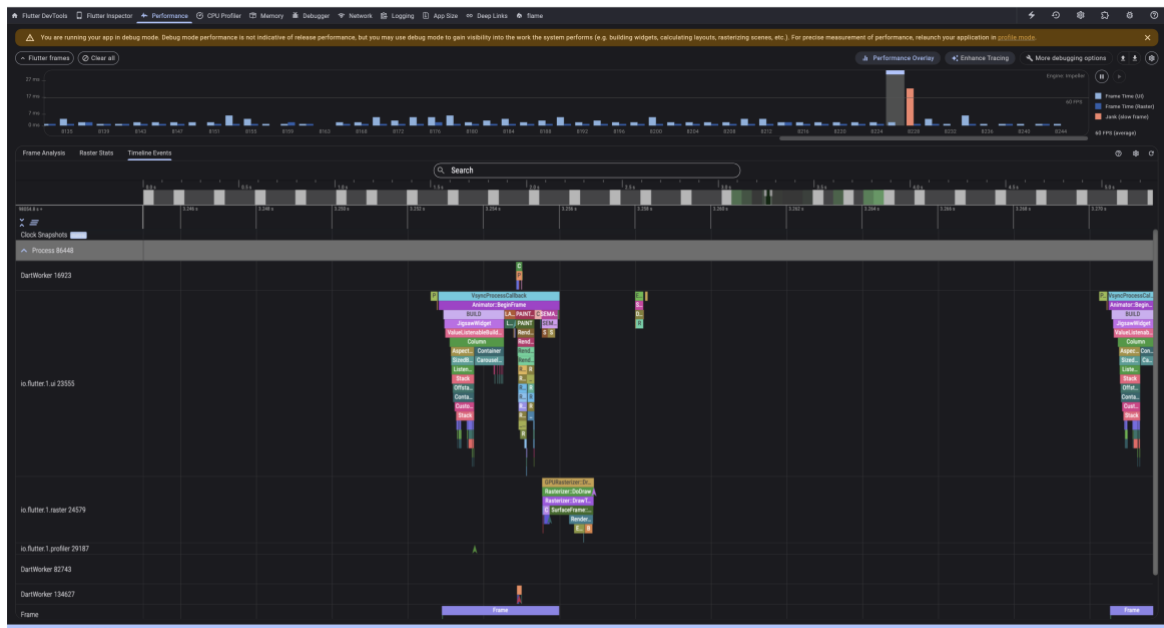
이를 도식화 하면 [그림 4] 처럼 표현할 수 있으며, RepaintBoundary 로 감싸진, UI 객체는(도식 내 붉은 색 노드) 다시 그려지지 않으며, 이를 통해 전체 페이지의 전반적인 성능을 향상시킨다.

3.4 성능 개선 결과

3.4.1 성능 측정 도구

Flutter 는 Devtools 를 제공하며, 이를 통해 UI 레이아웃의 상태와, 성능을 진단할 수 있다. 본 논문에서는 RepaintBoundary 적용 전과 후의 Build 시간과 Paint 시간을 비교하여 유의미한 성능 개선이 이루어졌는지 확인한다.

3.4.2 성능 측정 방법

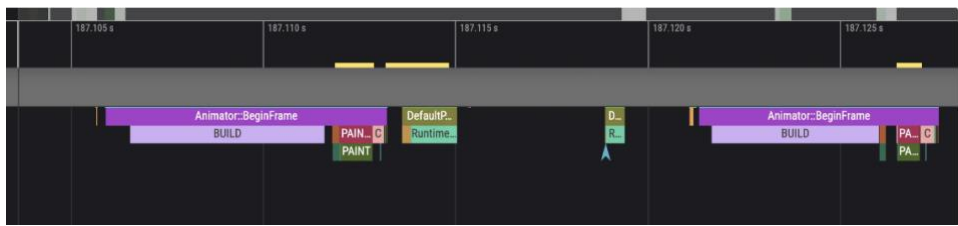


[그림 6] Flutter Devtools

Flutter 의 성능 측정 도구인 DevTools 를 사용하여 RepaintBoundary 적용 후의 렌더링 성능을 iOS 시뮬레이터(iPhone 15, iOS 17.5 버전)에서 동일한 환경에서 비교하였다.

퍼즐 조각을 드래그 하며 UI 상태가 실시간으로 업데이트되는 과정을 타임라인 패널에서 기록하고, Paint 단계를 중심으로 분석하였다. 기록된 데이터 중 첫 번째 사이클은 초기화 및 다른 요소의 빌드가 포함될 가능성이 높아 제외하였으며, 두 번째와 세 번째 사이클의 Paint 시간 중 더 긴 시간을 비교 기준으로 삼았다. DevTools 에서 제공하는 프레임 타임라인 데이터를 활용하여 Paint 단계의 작업량과 소요 시간을 정량적으로 분석하고, 이를 기반으로 RepaintBoundary 적용 후의 성능 차이를 평가하였다.

3.4.3 성능 측정 결과



[그림 7] RepaintBoundary 적용시 UI 렌더링 과정

Slices			Flow Events	
Name	Wall duration (ms)	Avg Wall duration (ms)		
	3560.988			
VsyncProcessCallback	326.209	6.79602		
Animator.BeginFrame	325.073	6.772354		
RenderPhysicalModel	299.175	1.24139		
RenderCustomMultiChildLayoutBox	247.75	1.283678		
RenderConstrainedBox	109.065	0.17367		
RenderPadding	100.414	0.232439		
PAINT (root)	99.444	2.07175		
PAINT	98.691	2.056062		
RenderRepaintBoundary	94.273	1.84849		
RenderCustomPaint	89.747	0.367815		
BUILD	89.309	1.822632		
RenderFlex	75.156	0.260958		
RenderPositionedBox	68.867	0.079523		
DartIsolate.HandleMessage	65.273	1.517976		
SEMANTICS (root)	53.44	1.113333		
SEMANTICS	52.025	1.083854		

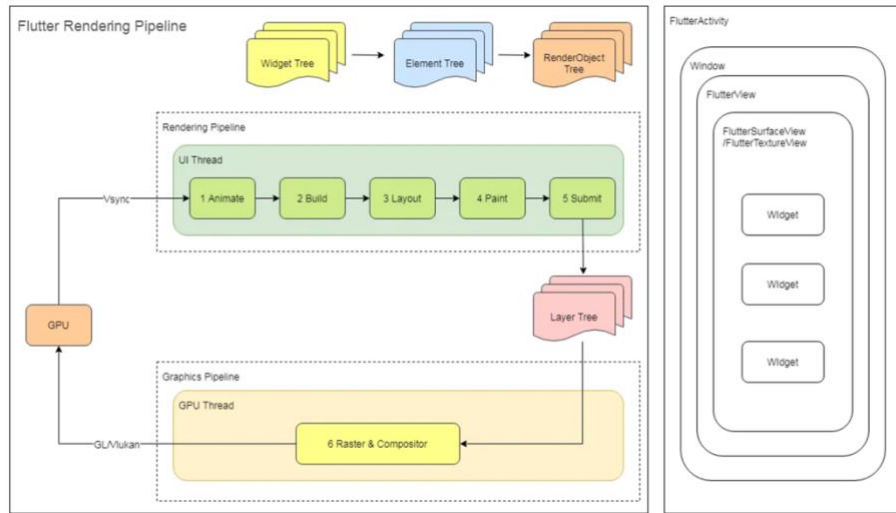
[그림 8] RepaintBoundary 적용 전 성능 측정 결과

Name	Wall duration (ms)	Avg Wall duration (ms)	Occurrences
	26.448		21
VsyncProcessCallback	7.331	7.331	1
Animator.BeginFrame	7.318	7.318	1
BUILD	5.043	5.043	1
Engine.DispatchPointerDataPacket	1.433	1.433	1
DefaultPointerDataDispatcher.DispatchPacket	1.429	1.429	1
RuntimeController.DispatchPointerDataPacket	1.203	1.203	1
PAINT (root)	0.842	0.842	1
PAINT	0.838	0.838	1
COMPOSITING	0.3	0.3	1
ScheduleSecondaryCallback	0.219	0.219	1
LAYOUT (root)	0.191	0.191	1
LAYOUT	0.181	0.181	1
PlatformVsync	0.049	0.049	1
FINALIZE TREE	0.034	0.034	1
POST_FRAME	0.012	0.012	1
VsyncFireCallback	0.009	0.009	1
UPDATING COMPOSITING BITS (root)	0.006	0.006	1

[그림 9] RepaintBoundary 적용 후 성능 측정 결과

테스트 결과, RepaintBoundary 적용 후 성능 최적화가 이루어졌음을 확인할 수 있다. 적용 전에는 PAINT작업이 98.691ms 소요되었고, RepaintBoundary 적용 후에는 PAINT작업이 0.842 ms로 감소하였다. 이를 통해 초기 렌더링 대비 PAINT 작업은 99.15%가 개선되었음을 확인할 수 있다.

3.4.4 RepaintBoundary의 부작용



[그림 10] Flutter의 렌더링 파이프라인

RepaintBoundary는 [그림 10]의 렌더링 과정의 4단계인 Paint와 6단계인 Raster & Compositor에서 부작용이 나타날 수 있다.

Paint 단계에서는 RepaintBoundary가 적용된 위젯이 독립된 Layer Tree로 추가되며, 변경되지 않은 레이어를 재사용하기 때문에 Paint 시간이 단축된다는 장점이 있다. 그러나 각 조각이 독립적인 Layer로 관리되기 때문에, 레이어 관리 비용이 증가하며 조각의 수가 너무 많거나, 고해상도 이미지를 사용하는 경우 GPU 메모리 사용량이 급증할 수 있다는 부작용이 있다.

Raster & Compositor 단계에서는 Paint 단계에서 생성된 레이어를 재사용해 Raster 작업²량이 줄어드는 장점이 있다. 하지만, Paint 단계와 마찬가지로 조각 수가 많아 레이어가 과도하게 생성되거나, 고해상도 이미지가 포함된 경우 GPU와 UI 스레드 간 병목현상이 발생할 수 있다. 특히, 드래그 애니메이션 등 동적 변화가 빈번한 UI에서는 FPS(Frame Per Second)가 감소하는 현상이 나타날 수 있다.

² UI 렌더링의 마지막 단계로, GPU가 데이터를 전달받아 화면에 픽셀로 변환하는 단계이다.

4. GetX의 활용과 문제점

4.4.1 GetX 개요

렌더링 최적화의 다른 방법으로, Get X 상태관리 도구를 활용할 수 있다. Get X는 Flutter의 대표적 상태 관리 라이브러리로, 반응형 상태 관리, 코드 단순화, 종속성 관리, 라우팅 지원 등의 특징을 갖고 있다. 본 논문에서는 퍼즐 조각이 이동 혹은 제자리에 맞추어질 때, 해당 조각의 상태 변화만 감지하여 UI를 다시 그리는 방식으로 최적화를 목표로 한다.

4.4.2 GetX 적용 및 활용구조

```
PuzzleController:
  상태:
    - blocks: 퍼즐 조각 리스트 (현재 위치, 기본 위치, 완료 여부 포함)
    - isCompleted: 퍼즐 완료 여부
    - gridSize: 퍼즐 그리드 크기
    - snapSensitivity: 스냅 민감도

  함수:
    1. initializePuzzle():
      - blocks 리스트 초기화
      - 각 조각에 랜덤 시작 위치 및 그리드 기반 기본 위치 할당

    2. moveBlock(index, newOffset):
      - blocks[index]의 위치를 newOffset으로 업데이트
      - 기본 위치와의 거리를 비교하여 스냅 여부 결정
      - 스냅 성공 시 완료 상태로 업데이트 및 전체 퍼즐 완료 여부 검사

    3. checkCompletion():
      - 모든 blocks가 완료 상태인지 확인
      - 완료 상태라면 isCompleted를 true로 변경
```

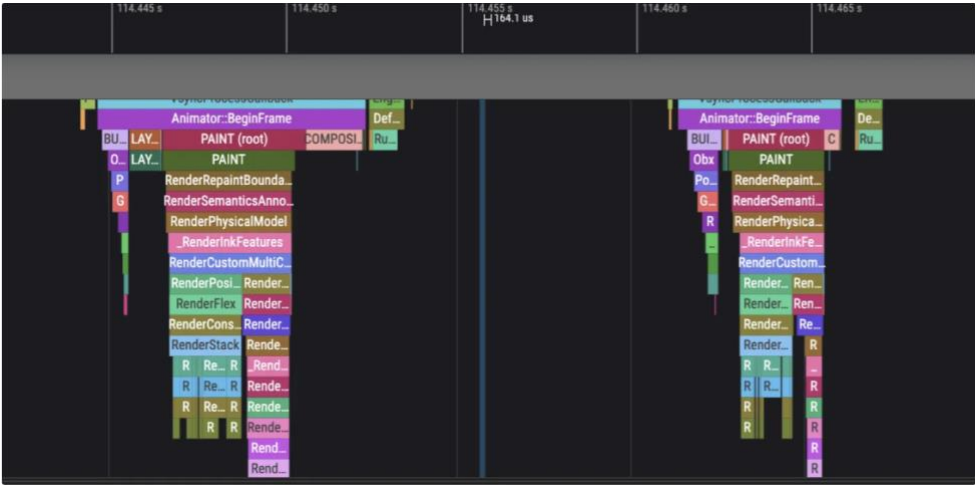
[그림 11] PuzzleController pseudo code

먼저 PuzzleController를 통해 상태관리를 진행하였다. 구성요소를 관리 대상인 상태와 주요 함수로 나누어 살펴보도록 하겠다.

관리되는 상태는 총 3가지로, 퍼즐 조각 리스트인 blocks, 퍼즐 전체 완료 상태인 isCompleted, 마지막으로 gridSize로, 퍼즐의 그리드 크기이다.

주요 함수로는 퍼즐 조각 리스트 초기화를 담당하는 initializePuzzle, 특정 조각의 위치를 업데이트 하는 moveBlock, 모든 조각의 완료 상태를 점검하는 checkCompletion으로 구성하였다..

4.4.3 성능 측정 결과



[그림 12] GetX 적용 후 UI 렌더링 과정

Name	Wall duration (ms)	Avg Wall duration (ms)	Occurrences
	60.54		113
RenderSemanticsAnnotations	6.103	0.6103	10
VsyncProcessCallback	4.674	4.674	1
Animator::BeginFrame	4.667	4.667	1
RenderConstrainedBox	4.404	0.27525	16
_RenderInkFeatures	3.261	1.087	3
RenderPhysicalModel	3.153	1.5765	2
RenderCustomMultiChildLayoutBox	2.864	1.432	2
PAINT (root)	2.772	2.772	1
PAINT	2.769	2.769	1
RenderRepaintBoundary	2.654	2.654	1
RenderPositionedBox	2.648	0.662	4
RenderPointerListener	2.119	0.192636	11
RenderStack	1.532	0.766	2
RenderFlex	1.513	1.513	1
RenderSemanticsGestureHandler	1.459	0.162111	9
RenderCustomPaint	1.25	0.125	10
BUILD	0.988	0.988	1

[그림 13] GetX 적용 후 성능 측정 결과

테스트 결과, GetX 적용 후 성능 최적화가 이루어졌음을 확인할 수 있다. 적용 전에는 PAINT 작업이 98.691 ms 소요되었고, GetX 적용 후에는 PAINT 작업이 2.769 ms로 감소하였다. 이를 통해 초기 렌더링 대비 PAINT 작업은 97.19%가 개선되었음을 확인할 수 있다.

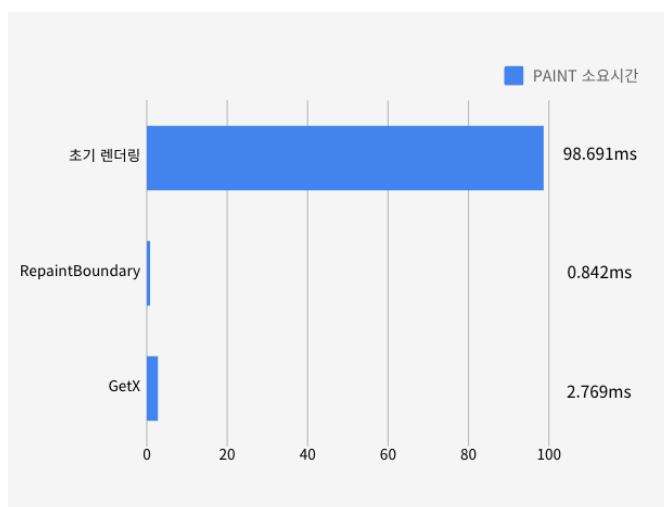
4.4.4 GetX 활용방안의 한계

GetX의 경우 전역 상태를 관리한다는 중요한 특징이 있다. 이러한 특징은 Rx로 관리할 상태를, Obx 객체로 원하는 곳에 그 상태를 렌더링하여 코드의 간소화라는 장점을 제공한다. 그러나, 본 논문에서 다룬 퍼즐 게임 외 다른 기능 추가 등의 이유로 앱의 규모가 커질 경우, 상태를 추적하는 것을 어렵게 만든다. 이러한 단점은 GetX의

간편함을 활용하려는 초기 단계와 달리, 복잡한 앱 구조에서는 장기적으로 문제를 야기할 수 있다.

5. 결론 및 향후 연구 방향

본 연구에서는 Flutter 기반 퍼즐 게임 개발 과정에서 렌더링 성능 최적화를 위한 두 가지 접근법, RepaintBoundary 와 GetX 상태 관리 도구를 비교 분석하였다.



[그림 14] 테스트 결과

Flutter 기반 퍼즐 게임 개발에서 렌더링 성능 최적화를 위해 RepaintBoundary 와 GetX 상태 관리 도구를 비교하였다. 테스트 결과, RepaintBoundary 는 PAINT 작업을 0.842ms 로, GetX 는 2.769ms 로 단축하며 초기 렌더링(98.691ms) 대비 성능을 크게 향상시켰다.

RepaintBoundary 는 정적이고 단순한 퍼즐이나 GPU 메모리 사용량이 문제가 되지 않는 경우 적합하며, 높은 렌더링 성능을 제공한다. 반면, GetX 는 동적 상태 변화가 많은 대규모 퍼즐에서 상태 관리와 코드 유지보수를 간소화하는 데 유리하다.

두 방법은 상황에 따라 보완적으로 사용될 수 있지만, 프로젝트의 요구사항에 따라 적합한 접근법을 선택해야 한다. 향후 연구는 이 두 방법의 장점을 결합하거나 최신 최적화 기술을 도입해 렌더링 성능과 메모리 효율성을 동시에 개선하는 방안을 모색할 필요가 있다..

6.참고문헌

- [1] Flutter 공식 문서, <https://docs.flutter.dev/>, 2024.10.29 검색
- [2] AWS 홈페이지, Flutter 란 무엇입니까?, <https://aws.amazon.com/ko/what-is/flutter/>, 2024.10.29 검색
- [3] Alibaba cloud 기술 블로그, Exploration of the Flutter Rendering Mechanism from Architecture to Source Code, <https://www.alibabacloud.com/blog>, 2024.10.30 검색
- [4] 강설주, 박판우, 배영권. "데이터 시각화를 적용한 클라우드 기반 곱셈구구 연습 애플리케이션 개발". 정보교육학회논문지, 제 26 권, 제 4 호, pp. 285-293, 2022.
- [5] 김근형, 우종민, 이진호, 이홍욱, 정유선, 제환웅, 박세진. "교내 휠체어 이용학생 전용 내비게이션 기초설계". 한국 HCI 학회 학술대회, 2024.1, pp. 1110-1113.
- [6] flutter Jigsaw game Repository, https://github.com/ocompilador/flutter_jigsaw_puzzle, 2024.10.30 검색
- [7]GetX 공식문서, <https://github.com/jonataslaw/getx?tab=readme-ov-file#about-get>, 2024.11.29 검색