권기동
웹 프론트엔드 포트폴리오
2025

팀 프로젝트: aya.gg v1

Private | 2021. 01 - 2024. 01 | React, TypeScript, Next.js, Radix UI, Stitches, React Query, Jotai, i18next, Git, Google Analytics, Microsoft Clarity

쿼터뷰 배틀로얄 게임 이터널 리턴의 전적 검색, 통계, 도감 정보, 루트 시뮬레이터 등을 제공하는 웹사이트입니다. 제가 팀에 참가했던 3년의 운영 기간 총 활성 세션 수 2,359,539를 기록하였습니다. 실사용자들과 메신저, 메일, 댓글 등으로 피드백을 주고받으면서 제품을 유용하게 개선하거나, 게임사와 직접 소통을 하며 의견 교환을 하는 등 소중한 경험을 할 수 있었습니다.

총 2인 팀 프로젝트로, 백엔드/프론트엔드로 나뉘어 저는 프론트엔드를 맡았습니다. 통계 페이지의 차트 구현을 제외한 모든 프론트엔드 코드를 구현했으며, 모든 UI 디자인, UX 설계도 맡았습니다.

루트 시뮬레이터 구현 | SVG, Image Map, ResizeObserver, CSS Transform

루트 시뮬레이터는 최적의 아이템 파밍 경로를 찾아주는 도구입니다. 이터널 리턴은 특정 지역에서 나오는 재료 아이템을 조합하여 더 강한 아이템을 만들어 싸워야 하는 게임인데, 당시 플레이어들은 지역에서 나오는 아이템 목록만을 가지고 스스로 실험하여 루트 경우의 수와 최적의 루트를 찾아야 하는 불편함이 있었습니다. 이러한 불편을 해소하기 위해 개발하게 되었습니다.
원하는 최종 아이템을 입력한 후, 자동 계산을 누르거나 직접 지도에서 방문할 순서를 지정하여 어느 지역에서 어떤 아이템을 찾아야 하는지, 인벤토리 상태는 어떤지 시뮬레이션하여 보여줍니다.
이후 **이터널 리턴 본 게임에 UX 측면에서 거의 그대로 이식**되었습니다.

- **화면 크기 변경 퍼포먼스 / 모바일 환경에서의 입력 반응 속도 개선**
  루트 시뮬레이터는 게임 지도 위에 각 지역을 image map 엘리먼트(<map>, <area>)로 표현하고, 시뮬레이션 된 경로를 SVG 엘리먼트로 표현합니다. 처음에는 지역 엘리먼트와 경로 엘리먼트의 크기를 동기화하기 위해 ResizeObserver를 사용했는데, 저사양 데스크탑/모바일 환경에서 느린 문제가 있었습니다. 이를 SVG viewBox와 CSS transform 속성을 활용해 ResizeObserver 기술을 사용하지 않아도 되는 형태로 수정했습니다. 이에 따라 데스크탑 환경에서 화면 크기를 변경할 때의 간헐적 끊김과, 모바일 환경에서의 입력 반응 속도가 개선되었습니다.

방송용 위젯 구현 | `Stitches`

- **OBS Browser Plugin 레이아웃 트러블슈팅**
  OBS 크로미움의 버전이 77로 낮아 CSS grid와 flex를 제대로 쓸 수 없었습니다. 이미 사이트의 다른 부분은 완성되어 있었기 때문에 공통 사용 컴포넌트를 특정 환경 전용으로 작성하여 두 벌 관리해야 하는 상황에 처해있었습니다. CSS-in-JS 라이브러리 Stitches로 Flex 컴포넌트를 구현하여 레이아웃을 구성하고 있었기 때문에, 최대한 외부의 기존 레이아웃 코드를 건드리지 않고 Flex 컴포넌트 내부에서 분기하여 negative margin으로 gap 동작을 폴리필하는 코드를 추가하는 방식을 택하였습니다. wrap 동작의 정확성을 희생하여 코드베이스의 복잡도를 크게 늘리지 않는 선에서 관리할 수 있었습니다.

SSR | `Next.js 12, React Query`

- **FCP 개선**
  Cloudflare CDN을 붙이고 SSR을 구현한 후 FCP가 비정상적으로 높아졌고, 원인을 분석한 결과 두 가지 문제를 발견했습니다. 하나는 React Query를 통해 hydration 되는 데이터의 크기가 비정상적으로 큰 것이었고, 또 하나는 BFF와 API 서버 간 전송이 의도치 않게 Cloudflare를 통해 이루어지는 것이었습니다. 이를 해결하기 위해 브라우저 캐시를 활용할 수 있도록 hydration 과정을 없애고 SSR과 CSR시 각각 fetch 하도록 구조를 바꾸고, BFF와 API 서버 간의 통신을 Cloudflare를 거치지 않고 직접 통신하도록 변경했습니다. 이에 따라 첫 접속 기준 FCP가 **10초에서 1.5초 이내로 560% 개선**되었습니다.

데이터 페칭 관리 | `React Query`

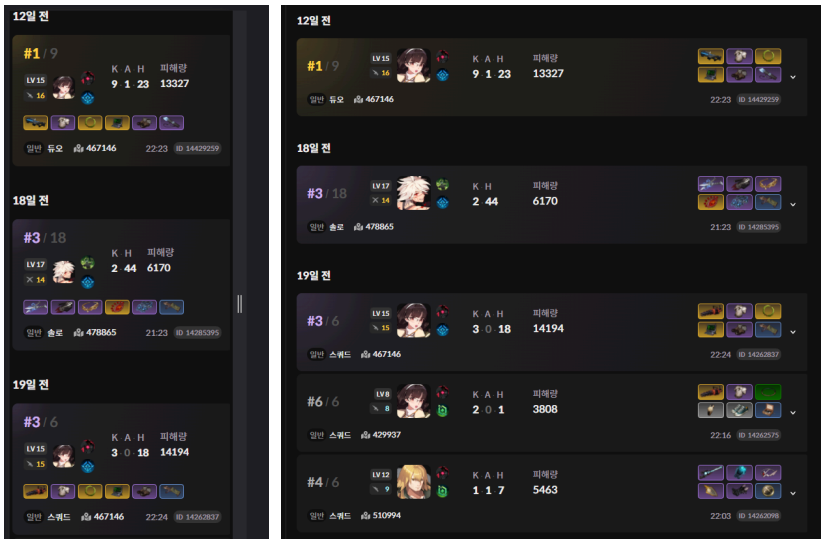
- **SWR vs React Query**
  fetch한 데이터를 관리하는 여러 방법론 중 당시 문서와 커뮤니티 자료가 잘 갖춰진 해결책으로는 SWR과 React Query가 있었습니다. 두 프로젝트는 거의 비슷한 API를 제공하지만, 후자를를 선택했습니다. 그 이유는 다음과 같습니다.
  - 갱신의 용이함: API에서 배열 형태의 key(hierarchical key)를 지원하여 key의 prefix가 같은 리소스들을 동시에 갱신 가능한 점
  - 나은 DX: 화면에서 바로 접근 가능한 Devtools를 지원하여 데이터의 상태가 어떤지, 마지막으로 갱신된 타이밍은 언제인지 쉽게 확인이 가능한 점

국제화 대응 | `i18next, next-i18next, typesafe-i18n`

aya.gg는 처음부터 국제화를 염두에 두고 개발되었습니다. 처음에는 한국어와 영어만을 지원했는데, 이후 프랑스의 게이머에게 메일로 연락을 받아 프랑스어 지원을 추가하게 된 게 기억에 남습니다.

- **next-i18next으로 SSR 대응 번역 구현**
  국제화를 지원하기 위해 직접 구현하는 옵션과 함께 여러 라이브러리를 비교하였고, react-i18next와 next-i18next를 선정하여 작업하였습니다. 이 라이브러리를 선정하게 된 이유는 다음과 같습니다.
  - 번역 파일을 불러오는 여러 방법(번들링, HTTP 요청 등)을 지원해 당시 국제화를 처음 작업하는 상황에서 최선의 방법이 무엇인지 실험이 쉽게 가능한 점
  - 초기 언어 설정, 선호 언어 저장, 그에 따른 SSR 단계부터의 올바른 언어 렌더링 등의 까다로운 동작을 Next.js 환경에서 쉽게 구현할 수 있도록 제공하는 점
- **typesafe-i18n 마이그레이션**
  v1에서 v2로 넘어가면서 Next.js에서 SvelteKit으로 스택을 바꿔 작업하게 되었는데, 국제화 관련 코드를 작성할 때의 불편한 점을 개선하고자 했습니다. 이때 다른 여러 솔루션 중에서 typesafe-i18n을 고르게 되었는데, 이 라이브러리를 선정하게 된 이유는 다음과 같습니다.
  - 더 작은 번들 크기(5.6kB gzipped vs 1.3kB gzipped)
  - 여러 언어 파일 사이에 누락된 키가 있는 것을 타입 단계에서 감지하여 특정 언어에서 번역이 되지 않은 문자열이 노출되는 것을 방지
  - 번역 키에 대해 타입 힌트가 제공되어 개발 속도 향상
  - 코드젠 방식으로 타입을 생성하기 때문에 복잡한 타입을 통해 타입 안정성을 제공하는 것에 비해 개발 단계에서 타입스크립트 컴파일러의 부담이 적을 것이로로 판단

반응형 대응 | Stitches, CSS Media Query



- **Mobile-first 방식의 반응형 구현**
  Stitches의 Responsive variants 기능을 사용하여 특정 breakpoint에서의 컴포넌트 모습을 지정하는 방식으로 구현했습니다. 루트 시뮬레이터를 구현하는 게 최초 관심사였기 때문에 처음에는 Desktop-first 방식으로 구현했는데, 플레이어 전적 페이지 등을 작업하며 모바일 화면에서 좁은 밀도에 많은 정보를 표시해야 하는 상황에서 작은 화면을 먼저 디자인하는 것이 편하다는 것을 느끼고 Mobile-first로 재작업하는 과정을 거쳤습니다.

오픈소스 기여: Nugget

GitHub | 2025. 04 - | Lit, TypeScript, Zustand, Git, Canvas, WebGL

오픈소스 영상 편집 소프트웨어입니다. SNS에서 메인테이너를 모집하는 글을 보고 제작자분께 연락하여 기여하게 되었습니다.

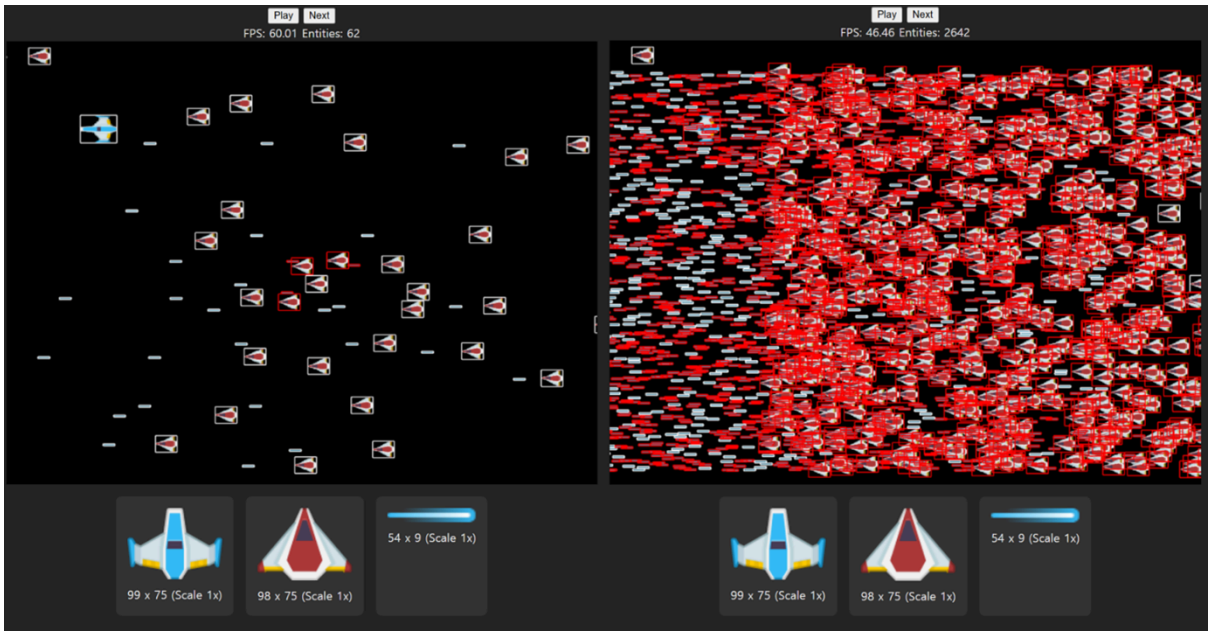
렌더러 리팩토링 / 버그 수정 | TypeScript, Zustand, Canvas, WebGL

- **캔버스 기반 렌더러를 리팩토링하여 약 2,000줄의 중복 코드 제거** PR#38, PR#36, PR#33
  프로젝트 오너와 프로젝트에 산재한 문제 중 가장 시급한 게 무엇인지 우선순위를 논의했습니다. 1순위로 영상 익스포트에 사용되는 렌더러와 프리뷰(타임라인 에디터)에 사용되는 렌더러 코드가 중복되지만 미묘하게 달라 결과물에 일관성이 없는 문제를 선정했습니다. 이후 코드를 분석하여 코드가 담고 있는 관심사를 추출하고, 여러 관심사가 하나의 클래스나 함수에 모여있지 않도록 리팩토링하는 과정을 거쳤고, 이렇게 적절히 추상화한 빌딩 블록을 바탕으로 익스포트 렌더러와 프리뷰 렌더러를 재작성했습니다. 결과 중복된 코드를 약 2,000줄 제거하고, 일관된 렌더링 동작을 얻을 수 있었습니다.
- **WebGL 필터 로직 재설계를 통한 UI 스터터링 제거** PR#36
  리팩토링 과정 중 비디오 효과 처리를 위한 WebGL 필터 구현에 결함이 있는 것을 발견하였습니다. 변경이 없어도 매번 셰이더를 컴파일하고 있었고, 비디오마다 WebGLContext를 들고 있는 형태였기 때문에 동시 렌더링 수에 제한도 있었습니다. 작업의 큰 카테고리는 리팩토링이었으나 근본적으로는 버그 수정을 겸하고 있었기 때문에, 수정하는 쪽으로 작업 방향을 설정했습니다. 코드를 통해 짐작되는 문제를 바로 수정하는 대신 디버거를 통해 실제 성능 병목이 어딘지 분석하는 과정을 거쳤습니다. 추상화를 포함한 기본 설계부터 다시 진행한 결과 제일 연산이 오래 걸리는 필터 기준으로, UI 반응 지연시간을 **평균 48ms → 0.262ms, 18,434% 개선**했습니다.

팀 프로젝트: ecs-collision-webworker

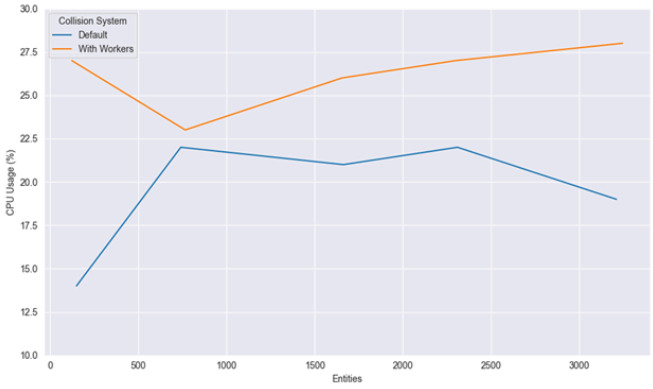
GitHub | 2024. 11 - 2024. 12 | `TypeScript, Git, WebGL, Web Worker`

빅데이터처리 과목의 팀 프로젝트입니다. 제 졸업 작품인 웹 게임 엔진의 충돌 처리 성능을 개선하는 프로젝트로, 모든 코드 작성을 맡았습니다. 해당 엔진은 Entity-Component-System 설계로 만들어졌습니다. Entity는 핸들, Component는 데이터, System은 데이터를 받아 처리하는 함수 형태로 구성되는 설계입니다.

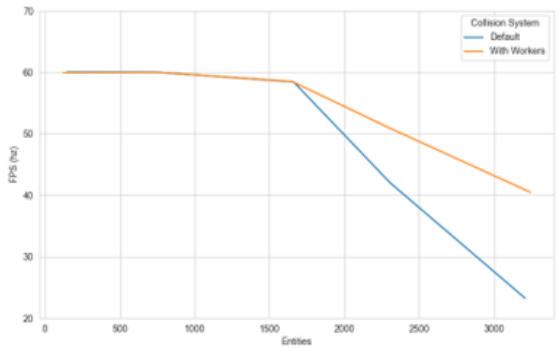


흰색 사각형: 충돌 영역, 빨간색 사각형: 같은 충돌 영역에 대하여 충돌이 감지됨

Web Worker를 사용한 성능 향상 실험 | `WebGL, Web Worker`



CPU Utilization 개선 그래프



FPS 그래프

- **Spatial Hash 기반의 충돌처리 병렬화**
  collider가 장착된 엔티티의 수가 1,500개를 넘어가면 퍼포먼스가 급격하게 저하되는 문제가 있었습니다. $O(N^2)$ 방식의 나이브한 충돌 처리 대신 Spatial Hash나 Quad-tree 등을 사용하여 충돌처리를 최적화하는 방식을 고민하게 되었고, 여기에서는 embarrassingly parallel 하여 병렬화하기 쉬운 Spatial Hash 방식을 채택하여 실험했습니다. Spatial Hash는 화면을 특정 크기의 cell로 나눠 그 안에 들어있는 객체들과 경계에 걸쳐 있는 객체들만 체크하는 기법입니다. 흐름을 파악하기 위한 핵심 코드는 다음과 같습니다.

```
// 작업을 Promise로 만들어 대기
const workerPromises = cellEntries.map(([_, cellObjects], i) => {
  return new Promise<void>((resolve) => {
    const worker = workerPool[i];
    worker.onmessage = (e: MessageEvent) => {
      collisionsFromWorkers[i] = e.data as [number, number][];
      resolve();
    };
    worker.postMessage(Array.from(cellObjects));
  });
});

// 모든 워커의 작업 완료를 대기
await Promise.all(workerPromises);
```

해당 충돌처리를 구현하기 위해 원래 async하지 않았던 시스템 정의를 async하게 바꾸면서 메인루프에서 마이크로태스크가 추가로 생기는 손해가 있었지만, 그것을 상회할 정도의 성능 이득을 검증할 수 있었습니다.

업무: 쿠키런 킹덤

2018. 03 - 2020. 10 | Unity, C#, Git, Spine

DX 개선 | Node.js

- **Unity Cache Server 도입으로 초기 프로젝트 로딩 속도 개선**
  프로젝트 규모가 커짐에 따라 초기 프로젝트 로딩 속도가 느려지는 문제가 있었습니다. 이를 해결하기 위해 빌드 서버에 Unity Cache Server(이후 Unity Accelerator)를 구축하여 10분 이상 걸리던 초기 프로젝트 로딩을 2분 이내로 **400% 개선**했습니다. 엔지니어뿐만 아니라 아트, 기획팀 동료들에게도 좋은 반응을 얻었습니다.