프로젝트명: 실시간 공유 웹 스마트보드

최종 결과 보고서

클라이언트 팀
김윤지
오경우
최진혁
위승현

최종 결과 보고서	프로젝트명: 실시간 공유 웹 스마트보드
작성자 : 김윤지	작성일: 2018/01/06

목 차

I. 프로젝트 개요 3

1 프로젝트 개요 3
2 개발 환경 3
3 기본 구조 3

II. 특정 이슈 5

1 Canvas vs SVG 5
2 실시간 공유 5
3 부드러운 드로잉 5
4 다중 페이지 6

III. 구현 내용 7

1 그림 툴 7
1.1 펜 8
1.2 지우개 11
1.3 도형 (원, 사각형) 12
2 Undo, Redo 13
3 다중 페이지 및 페이지 전환 13
4 페이지 프리뷰 14
5 실시간 공유 15

IV. 개선 필요 사항 16

1 펜 보정 작업과 다중입력 16
2 실시간 공유 17
3 Undo, Redo 17
4 페이지 프리뷰 17
5 데이터 처리 로직 18

최종 결과 보고서	프로젝트명: 실시간 공유 웹 스마트보드
작성자 : 김윤지	작성일: 2018/01/06

Ⅰ. 프로젝트 개요

1 프로젝트 개요

```
/*
      생략
*/
```

2 개발 환경

프로젝트 진행에 앞서 개발을 위한 환경을 선정하였다. 우선, 공유에 사용할 서버를 로컬 서버로 진행 할 경우 실시간 공유의 딜레이가 적어 실제 사용시 걸리는 딜레이를 판단하기 어렵다고 생각하였다. 따라서 사전에 도메인을 등록 해 놓은 리눅스 OS 에서 Node.js 를 통해 개발 및 실행을 하여 실제 사용 시에 걸리는 딜레이를 보다 확실히 알 수 있도록 하였다. 공유를 보다 실시간으로 하기 위해 공유단위와 주기를 최소화 하여 클라이언트간의 실시간 통신을 socket.io 를 통해 중계하기로 하였다.

클라이언트 개발은 React.js 프레임워크를 활용하여 개발하였고, 각 컴포넌트들의 데이터 교류를 위해 React.js 에서 FLUX 패턴을 사용할 수 있게 도와주는 Redux.js 를 적용였다. 그리하여 페이지 정보와 현재 사용중인 툴, 각 툴의 옵션은 모두 store 에서 관리하며, 툴의 전환, 페이지 변경, 페이지의 정보 업데이트는 특정 컴포넌트의 정보가 필요한 경우를 제외하고 모두 action 에 의해 이루어진다.

Redux.js 의 store 의 변화와 action 의 발생을 눈으로 확인할 수 있게 해주는 크롬 확장 프로그램인 Redux DevTool 을 사용하였다. 이를 사용하기 위해선 Redux DevTool 을 설치하고, 프로젝트의 src 디렉터리에 위치한 index.js 의 코드변경이 필요하다. Index.js 에서 store 를 생성하는 구문을 주석처리하고, 다음줄에 주석처리 되어있는 “const store = createStore(reducers, compose(applyMiddleware(ReduxThunk), window.__REDUX_DEVTOOLS_EXTENSION__ && window.__REDUX_DEVTOOLS_EXTENSION__()));” 구문의 주석을 제거해주면, Redux DevTool 을 실행하여 브라우저에서 해당 프로젝트의 action 발생과 store 의 변화를 확인할 수 있다. (참조 : Redux 개발을 더 편하게, Redux-Devtools)

3 기본 구조

우선 화면 전체에 그림을 그릴 수 있는 Canvas 가 존재한다. 보이는 부분은 그림을 그릴 수 있는 Canvas 태그 하나지만, css 의 z-index 속성을 통하여 총 세개의 태그가 전체 화면에 겹쳐져 있다. 제일 바깥쪽에는 현재 페이지에 대해 그림 작업을 수행할 수 있는 Canvas 태그가 있고, 그 아래엔 바탕을 보여줄 수 있는 img 태그가 있다. Canvas 의 경우 background 가 투명이라 배경과 그림을 구분할 수 있었고, 구분하여 사용, 표현함으로써 페이지의 바탕이 변경되더라도 해당 페이지의 그림 데이터에는 영향을 끼치지 않는다.

그리고 제일 안쪽에는, 현재 보고있는 페이지가 아닌, 다른 페이지에서 다른 사용자가 작업할 때의 페이지 프리뷰 업데이트를 위한 또다른 Canvas 가 있다. 이 Canvas 는 사용자에게 직접 보이지 않으며, 오로지 다른 페이지의 프리뷰 업데이트시에만 활용된다.

우측에는 해당 스마트보드에서 사용할 수 있는 ToolBar 가 토글 형식으로 존재한다. >> 버튼과 << 버튼을 통해 ToolBar 를 토글할 수 있으며, ToolBar 는 각각 펜, 지우개, 도형 (삼,사각형), 그리고 Undo 버튼과 Redo 버튼이 존재한다. 각 툴은 툴의 이미지를 클릭하여 전환할 수 있으며, 툴 이미지를 더블 클릭하여 해당 툴의 색상이나 굵기를 설정할 수 있는 창을 띄울 수 있다.

하단에는 현재 만들어 져 있는 페이지의 프리뷰를 볼 수 있는 PageBar 가 존재한다. 사용자는 프리뷰

이미지를 클릭함으로써 해당 페이지로 전환할 수 있으며, PageBar 최우측 + 버튼을 통해 새 페이지를 만들 수 있다. 프리뷰는 한 그림 작업이 완전히 이루어지는 시점에서만 업데이트된다.

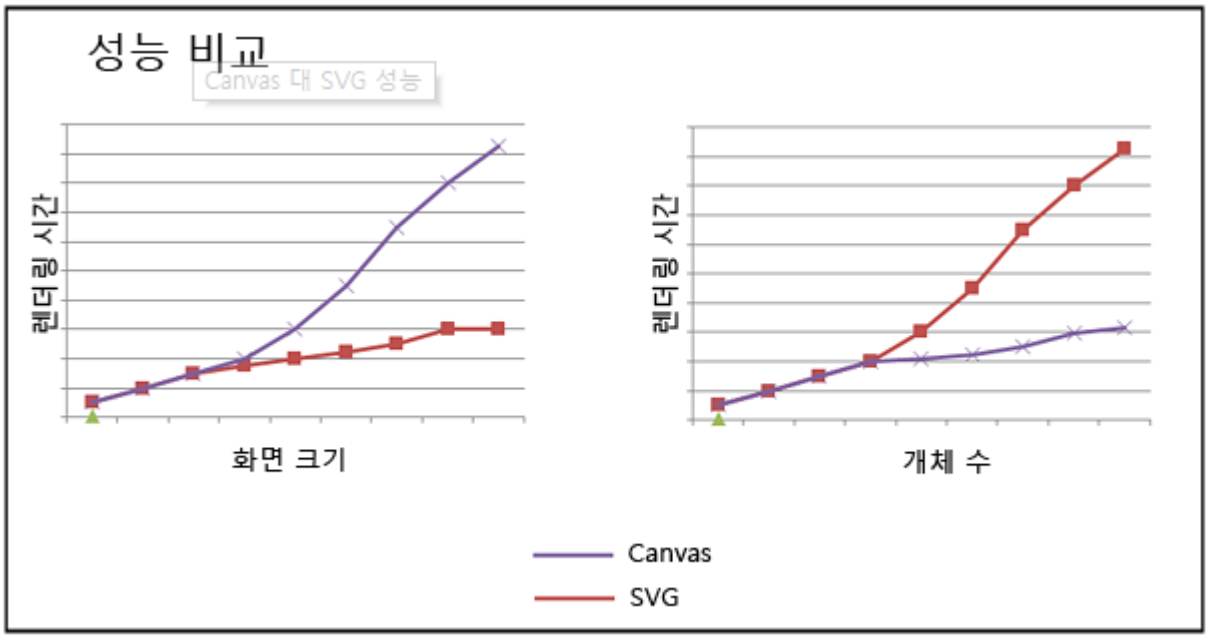
Picture 1 웹 스마트보드의 초기 화면

II. 특정 이슈

1 Canvas vs SVG

그림판의 기능을 구현하기 위해 SVG 와 Canvas 둘 중 하나를 선택하여야 했다. 그렸던 그림을 드래그&드롭을 통하여 옮기는 기능을 구현하기 위해선 SVG 를 사용하여 객체화를 시켜야 했지만, 여러가지 이유에서 객체화와 해당 사항을 포기하고 SVG 대신 Canvas 를 선택하였다.

우선, 그림 데이터 (이하 아이템) 가 많아진 상태에서 페이지를 초기화하여 다시 그려야 할 경우, 다시 그리는 아이템의 개수가 증가할수록 DOM 에 개체를 지속적으로 추가하는 SVG 의 퍼포먼스가 Canvas 보다 떨어지게 된다. 화면의 크기가 클 때의 퍼포먼스는 SVG 가 높으나, 개체 수가 더 많은 영향을 끼칠 수 있을 것 같기에 화면크기에 대한 퍼포먼스는 신경쓰지 않았다.



Picture 2 Canvas 와 SVG 의 성능 비교

또, 아이템을 객체화 시키면 지우개 툴의 기능을 '지나가는 부분을 지우는 것' 이 아닌 객체를 지우는 기능 으로 변경해야 했기에, 지우개 툴의 원 기능을 유지하기 위해 객체화를 포기하였다.

2 실시간 공유

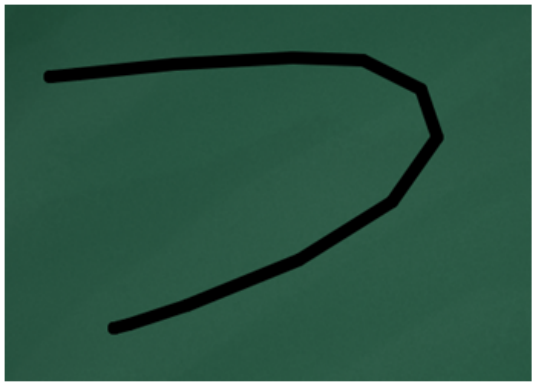
각 클라이언트 간에 드로잉작업을 공유하기 위해서는 아이템의 정보를 넘겨주어야 했고, 실시간 공유를 위해선 공유의 주기를 최소화하여 공유를 자주 해야한다. 이를 위해 Canvas 에 그리는 작업이 시행될 때 각 툴별로 공유하는 주기를 최소화하여 최대한 실시간 공유를 하도록 결정하였다.

3 부드러운 드로잉

Canvas 로 구현하는 일반적인 그림판의 경우, 펜 기능을 MouseMove 이벤트 발생 시 마다 이전 좌표와

이벤트 발생 좌표를 Canvas 의 lineTo 메서드를 통하여 선을 긋는 방법으로 구현하였다. 하지만 해당 방법은 MouseMove 이벤트 발생 주기 사이에 급격한 움직임을 보일 때에 선이 부드럽지 않고 각지게 그려지는 현상을 야기하였다. 이는 lineTo 메서드가 직선을 긋는 것이고, MouseMove 이벤트에서 얻을 수 있는 좌표들의 거리가 멀어서 발생하는 문제였다.

```
const drawLine = (item, start, {x, y}) => {
    context.save();
    context.lineJoin = 'round';
    context.lineCap = 'round';
    context.beginPath();
    context.lineWidth = item.size;
    context.strokeStyle = item.color;
    context.globalCompositeOperation = 'source-over';
    context.moveTo(start.x, start.y);
    context.lineTo(x,y);
    context.closePath();
    context.stroke();
    context.restore();
};
```



Picture 3 lineTo 를 이용한 drawLine 함수(좌)와 각지게 그려지는 현상(우)

이를 해결하기 위해서 많은 고민을 하였고, 여러 가지 방법을 생각 해 내었다. 우선 MouseMove 의 호출 주기를 줄여 좌표간의 차이를 줄이는 방법과 MouseMove 대신 그리는 작업을 수행 할 다른 메서드를 구현하는 방법 등을 구상하였다. 하지만 이는 시스템이나 브라우저 자원에 직접적으로 접근하여야 하는데, 해당 방법들을 수행 하려고 많은 조사를 했으나 찾을 수 없어 사용하지 못하였다. 많은 고민을 하던 중 Canvas 의 quadraticCurveTo 메서드를 활용하는 방법을 구상하였고, 그림을 그릴 때 보정하는 작업을 추가함으로써 보다 부드러운 드로잉을 구현하였다.

4 다중 페이지

한 클라이언트에서 여러 개의 Canvas 를 보여주기 위한 방법을 고민하였다. 우선, 여러 개의 Canvas 를 각 페이지마다 추가하는 것이 있었는데, 이는 페이지의 전환이 쉽지 않고, Canvas 를 전환할 경우 변경된 Context 를 통해 각 툴을 다시 초기화하거나, Canvas 별로 별개의 툴을 가지고 있어야 해서 효율성이 떨어진다. 또 다른 방법으로는 각 페이지 별로 Canvas 의 Context 를 저장하는 것이 있는데, 이 역시 툴 관련 비효율적인 작업이 발생한다.

선택한 방법으로는, Canvas 는 한 개만 두고, 각 페이지별로 해당 페이지의 드로잉 데이터들을 저장하고, 페이지 전환 시 Canvas 를 초기화 한 후 해당 페이지의 아이템들만 다시 그려 다중 페이지를 구현하는 방식을 선택하였다.

III. 구현 내용

1 그림 툴

우선 각 툴의 구조는, Canvas 의 Context 를 인자로 받는 함수를 정의하고, 그 안에 해당 아이템의 정보를 저장할 변수와 마우스 이벤트 발생 시 호출할 내부함수들 (onMouseDown, onMouseMove, onMouseUp, reDrawWithData, getToolType) 을 정의하고 반환하여 사용하는 클로저 패턴을 이용한 방식이다. Canvas 가 있는 컴포넌트에서 각 툴을 초기화한 후, 해당 Canvas 에 대한 툴 셋을 저장해둔다. 이후 아이템에 들어있는 tool 속성을 참조하여 알맞은 툴을 사용한다.

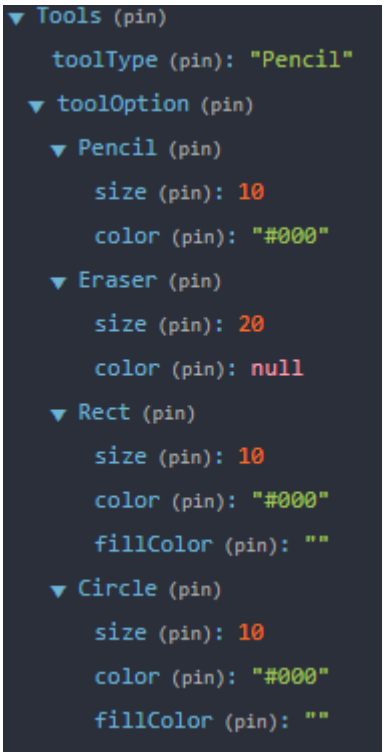
각 함수들은 이름과 매칭되는 마우스 이벤트 발생 시 호출하며, 호출시점의 아이템 상태를 반환한다. onMouseDown 함수는 MouseDown 이벤트 발생 시의 좌표, 현재 설정된 색상, 굵기 등을 인자로 받아 아이템의 정보를 초기화한다. onMouseMove 함수는 MouseMove 이벤트 발생 시 마다 호출되며, 해당 이벤트가 발생 한 좌표를 인자로 받아 아이템의 데이터에 입력받은 좌표를 푸시하고 해당 툴의 드로잉작업을 Canvas Context 에 수행하는 drawLine 함수를 호출한다. onMouseUp 함수는 MouseUp 이벤트가 발생 한 좌표를 받아, 호출주기로 인한 작업의 누락을 막기위해 해당 좌표를 넣은 onMouseMove 함수를 다시 한번 호출을 한 후, 아이템의 최종 정보를 반환하고 외부함수에 정의된 변수를 초기화한다.

getToolType 은 해당 툴의 타입을 반환하고, reDrawWithData 는 아이템의 정보를 인자로 받아 해당 아이템을 다시 그리는 함수이다. drawLine 함수는 각 툴마다 내부가 다르며, 툴에 해당하는 작업을 Context 에 그리는 작업을 수행한다.

```
1  export const TOOL_NAME = 'ToolName';
2
3  export default (context) => {
4
5      let stroke = null;
6      let points = [];
7
8      const onMouseDown = ( x, y, color, size ) => { ... };
9      const drawLine = ( item, start, {x, y} ) => { ... };
10     const onMouseMove = ( x, y ) => { ... };
11     const onMouseUp = ( x, y ) => { ... };
12     const reDrawWithData = ( item ) => { ... };
13     const getToolType = () => { ... };
14
15     return {
16         onMouseDown,
17         onMouseMove,
18         onMouseUp,
19         drawLine,
20         reDrawWithData,
21         getToolType
22     }
23 }
```

Picture 4 기본 툴 함수의 구조

각 툴의 이름은 함수와 별개로 TOOL_'NAME' 을 통해 구분할 수 있게 하였으며, 현재 사용중인 툴과 각 툴의 색상,굵기 등 툴의 정보를 store 에 저장하여 관리 및 참조하고, 툴의 옵션 변경 및 툴의 전환은 각 Action 들을 통해 이루어진다.



Picture 5 Tool 관련 store 의 기본 상태

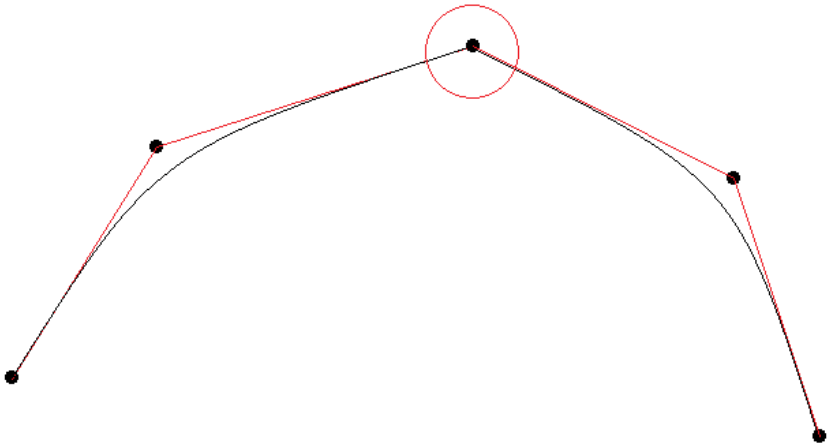
1.1 펜

Canvas 를 사용하여 구현한 일반적인 그림판의 경우, MouseMove 이벤트 발생 시 마다 이전 좌표에서 이벤트 발생 좌표간 lineTo 메서드를 통해 직선을 그리는 방식으로 펜 기능을 구현하였다. 하지만 좌표들의 거리가 멀거나 방향을 급격하게 바꿀 경우, 그려진 선이 각 지는 현상이 발생하였고, 이를 해결하기위해 Canvas 의 quadraticCurveTo 함수를 활용하는 방법을 생각하였다.

quadraticCurveTo 는 Canvas 에서 제공하는 함수로써, 제어점과 목표점을 인자로 받아 2 차 곡선을 그리는 함수이다. 이를 이용해, 임의의 제어점을 활용하여 각 좌표간에 직선이 아닌 곡선으로 연결하여 그릴경우 부드러운 그림을 그릴 수 있지 않을까 생각하였다. 하지만 임의의 제어점을 정하기 위해선 현재 그림의 다음 좌표를 알고, 방향을 정확히 계산해야 하여 상당히 번거롭고 복잡할 것 같아 다른 방법을 찾기로 하였고, 특정 순서에 입력되는 좌표를 제어점으로 두어 그리는 방법을 통해 구현 해 보기로 하였다.

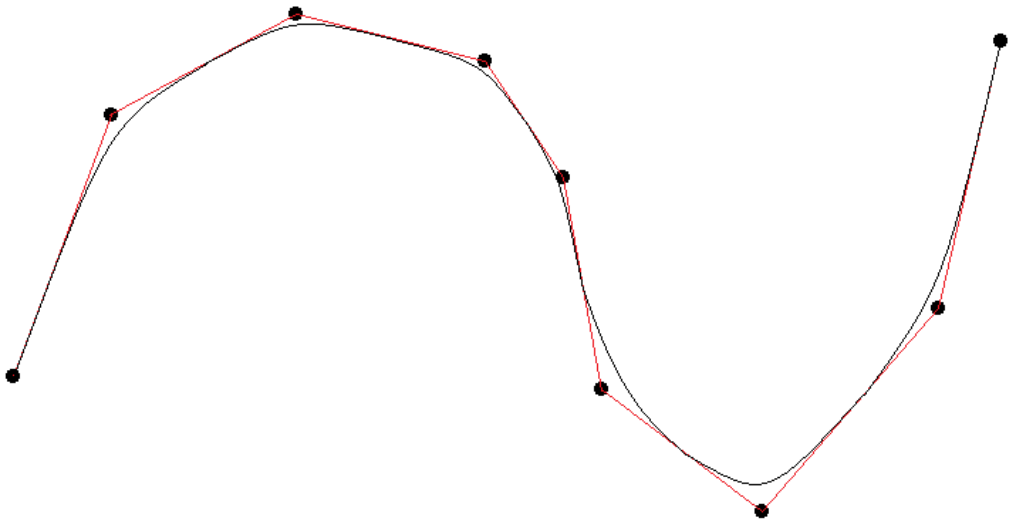
우선, 짝수 번 째 좌표를 제어점으로 한 quadraticCurveTo 함수를 사용하여 그려보았다. MouseMove 이벤트 발생 시, 아이템의 좌표 개수가 짝수일 때에 drawLine 함수를 호출하여 그려지도록 하였다. 하지만 짝수 번 째의 선만 굴곡이 생길 뿐, 그 다음 좌표와의 각이 많이 틀어질경우 곡선 과 곡선

사이가 각이 지는 현상이 발견되었다. 이는 몇 번 째 좌표를 제어점으로 둬도 같은 문제가 발생하였고, 결국 quadraticCurveTo 함수에 사용할 제어점과 목표점을 다르게 접근 할 필요성이 있었다.



Picture 6 짝수 번 째의 좌표를 제어점으로 한 quadraticCurveTo 함수

다음으로, 임의의 좌표 (x1,y1) 를 제어점으로 하고, (x2,y2) 와의 중간 지점을 목표점으로 하여 그려보기로 하였다. N 개의 좌표가 있을 경우, 시작점 (x0, y0) 에서 시작하여 (x1, y1)을 제어점으로, ((x1+y1)/2 , (y1+y2)/2) 를 목표점으로 하여 quadraticCurveTo 함수를 N-2 까지 반복하여 호출하고, N-1 번째 좌표를 제어점, N 번째 좌표를 목표점으로 하는 quadraticCurveTo 함수를 호출하며 선 그리기를 완료한다.



Picture 7 중간점을 목표점으로 하는 quadraticCurveTo 함수

위와 같은 방법을 사용하면 원하는 결과를 얻을 수 있었으나, 좌표가 추가되었을 때 이전 좌표의 끝 지점에서 이어서 그릴경우 역시 위에서 기술한 이슈가 발생할 수 있었다. 이를 위해 새 좌표를 추가할때 이전 좌표를 참조하여 그려 보았으나, 이전 작업과의 결과물이 틀려 선이 이상하게 그어지는 문제도 발생하였다. 이 문제점들은 좌표가 추가될 때 마다 이전에 그렸던 작업을 지우고 좌표가 추가된 새 아이템을 그리는 방식으로 해결할 수 있었다.

우선 툴 함수의 파라미터를 Canvas 의 Context 가 아닌 Canvas 태그로 변경한다. 그리고 Canvas 의 상태를 저장 할 변수를 추가하여 onMouseDown 함수에서 아이템 변수를 초기화하고 나서 해당 Canvas 의 현재 상태를 저장한다. 이후 onMouseMove 함수에서 MouseDown 이벤트 때 저장해둔 Context 를 사용하여 Canvas 를 이전상태로 돌리고 drawLine 함수를 호출함으로써, 이전 drawLine 의 결과물을 지우고 새 좌표를 추가한 아이템을 다시 그리는 작업을 수행한다. 상태를 저장한 변수는 onMouseUp 함수에서 아이템 변수를 초기화할 때 같이 초기화 시킨다.

추가적으로, 점을 찍는 경우에 대하여 보정작업이 필요가 없고, 점의 좌표가 3 개 미만이 경우에 대한 quadraticCurveTo 메서드의 에러를 방지하기 위한 예외처리를 함으로써 부드럽게 그릴 수 있는 펜 기능을 구현하였다.

```
let stroke = null;
let points = [];
let context = canvas.getContext('2d');
let mmxCanvas = document.createElement('canvas');
mmxCanvas.width = 1920;
mmxCanvas.height = 1080;
let mmxCanvasContext = mmxCanvas.getContext('2d');

const onMouseDown = (x, y, color, size) => {
    stroke = {
        tool: TOOL_PENCIL,
        color: color,
        size: size,
        points: [{x, y}],
    };
    mmxCanvasContext.drawImage(canvas, 0,0);
    return [stroke];
};
```

Picture 10 툴 함수의 내부 변수와 onMouseDown 함수

```
const onMouseMove = (x,y) => {
    if( !stroke ) return [];
    const newPoint = { x, y };
    context.clearRect(0,0,1920,1080);
    context.drawImage(mmxCanvas, 0,0);

    stroke.points.push(newPoint);
    points.push(newPoint);
    drawLine(stroke);

    return [stroke];
};

const onMouseUp = (x,y) => {
    if( !stroke ) return;
    onMouseMove(x,y);
    let strokeData = Object.assign({},stroke);
    strokeData.points.push({x:x, y:y});
    mmxCanvasContext.clearRect(0, 0, 1920, 1080);
    stroke = null;
    points = [];

    return [strokeData];
};
```

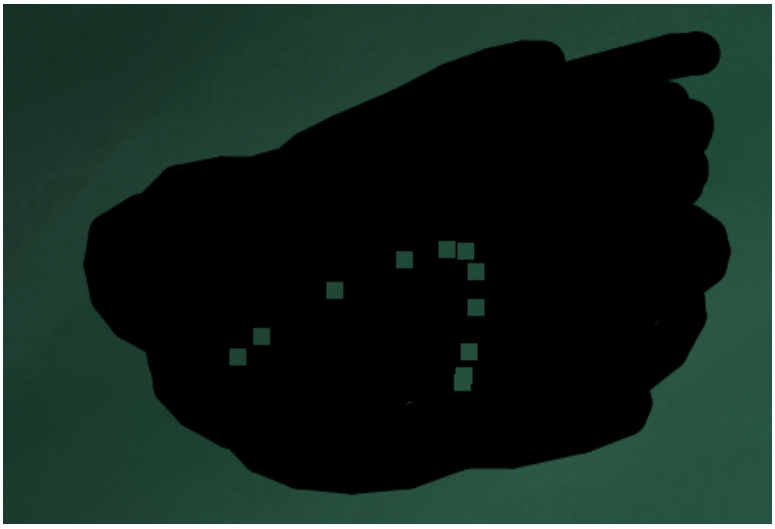
Picture 8 onMouseMove, onMouseUp 함수

```
const drawLine = (item) => {
    let pointt = item.points;
    let i;
    if( pointt.length == 1 ) return;
    context.save();
    context.lineJoin = 'round';
    context.lineCap = 'round';
    context.beginPath();
    context.lineWidth = item.size;
    context.strokeStyle = item.color;
    context.globalCompositeOperation = 'source-over';
    context.moveTo(pointt[0].x,pointt[0].y);
    if( pointt.length == 2 ) {
        context.arc(pointt[0].x, pointt[0].y,
            item.size/2, 0, 2 * Math.PI, true);
        context.closePath();
        context.fill();
    } else if( item.points.length == 3 ) {
        context.lineTo(pointt[1].x, pointt[1].y);
        context.lineTo(pointt[2].x, pointt[2].y);
        context.closePath();
        context.stroke();
    } else {
        for( i = 1; i < pointt.length - 2; i++ ) {
            let c = (pointt[i].x + pointt[i+1].x) / 2,
                d = (pointt[i].y + pointt[i+1].y) / 2;
            context.quadraticCurveTo(
                pointt[i].x, pointt[i].y,
                c, d
            );
        }
        context.quadraticCurveTo(
            pointt[i].x, pointt[i].y,
            pointt[i+1].x, pointt[i+1].y
        );
        context.stroke();
    }
    context.restore();
};
```

Picture 9 quadraticCurveTo 사용과

1.2 지우개

Canvas 에는 특정 영역의 그림 데이터를 지우는 clearRect 함수가 존재한다. 몇몇 Canvas 그림판 예제의 경우 MouseMove 이벤트 발생 시 마다 이벤트가 발생한 좌표의 주위를 clearRect 로 지우는 방식으로 구현하였다. 하지만 이는 지우는 영역이 정사각형이고, MouseMove 이벤트가 발생한 좌표들의 거리가 멀 경우, 각 좌표의 주위만 정사각형으로 지워지게 되어 띄엄띄엄 지워지는 문제가 발생하였다. 그리하여 많은 Canvas 예제는 지우개 툴 대신 Canvas 전체를 초기화 하는 기능을 넣거나, 자체적으로 지우개 기능을 구현하였다.



Picture 11 clearRect 를 활용한 지우개의 문제점

이 프로젝트에서 지우개 툴을 어떻게 개발해야 할지 고민하다가, Canvas 의 속성중 하나인 globalCompositeOperation 속성과 기존의 펜 툴을 혼합하여 해당 프로젝트의 지우개 툴을 완성하였다.

우선 globalCompositeOperation 속성은 Canvas 에 그림을 그릴 때, 이미 그려져 있는 도형과 어떻게 합성할 것인지에 대한 속성이다. 기본값으로 이미 그려진 도형 위에 그리게 설정 되어있고, 변경을 통해 여러 옵션을 선택할 수 있다.

이전 페이지들에서 나온 모든 예제에서, 새로 그리는 도형은 언제나 이미 그려진 도형의 위에 그려졌습니다. 대부분의 상황에서는 이렇게 하는 것이 적절하지만, 도형을 합성하기 위한 순서를 제한하게 되는데, `globalCompositeOperation` 속성을 설정함으로써 이러한 상태를 바꿀 수 있습니다.

globalCompositeOperation

기존 도형 뒤에 새로운 도형을 그릴 수 있을 뿐만 아니라, 도형의 일정 영역을 가려 보이지 않게 하거나 캔버스의 특정 부분을 지우는 (`clearRect()` 메소드는 사각형의 영역만을 지우지만, 이 같은 제한도 없다.) 등의 효과를 얻을 수도 있습니다.

`globalCompositeOperation = type`

새로운 도형을 그릴 때, 도형 합성 방법의 유형을 설정합니다. type에는 12개의 방법이 있습니다.

Picture 12 globalCompositeOperation 속성 설명 (참조 : MDN Canvas Tutorial)

globalCompositeOperation 의 설정값중 하나인 'destination-out' 은 기존 도형을 새 도형과 겹치는 부분을 사라지게 하는 옵션이다. 이를 적용하게 되면, 그려져 있는 아이템 위에 새 아이템을 그릴경우 아이템 위에 겹쳐 그리지 않고, 기존 그림의 겹치는 부분을 지우고 새로 그리게 되는 형식으로 바뀌게 된다.

다음, context 의 속성중 색상을 결정하는 color 속성이 있다. 이 값을 아무것도 주지 않으면 어떤 색상으로 그려질지 확인해 보기 위해 기존의 펜 툴에서 color 속성을 제거하여 기존에 그려놓은 그림 위에 그려보았다. 그 결과 아무것도 그려지지 않았고, globalCompositeOperation 의 기본값이 기존 그림 위에 겹쳐서 그리는 것 이었으므로, color 의 기본값은 투명색이라는 것을 알아냈다.

위에서 알아낸 결론을 바탕으로, 기존의 펜 툴에 color 속성을 제거하고 globalCompositeOperation 속성값을 destination-out 으로 주어 기존의 아이템 위에 그리게 되면 지우개의 기능을 할 수 있지 않을까 하여 적용해 보았다. 그리하여 기존 그림의 겹치는 부분을 지우고 투명한 색상으로 덧그리는 방식의 자연스러운 지우개 툴을 구현하였다. 펜에서 발생한 각이 지는 이슈가 발생하였으나, 지우개에 보정작업은 필요할 것 같지 않아 제외하였다.

```
const drawLine = ( item, start, end ) => {
    context.save();
    context.globalCompositeOperation = 'destination-out';
    context.beginPath();
    context.lineJoin = 'round';
    context.lineCap = 'round';
    context.lineWidth = item.size;
    context.beginPath();
    context.moveTo(start.x, start.y);
    context.lineTo(end.x, end.y);
    context.stroke();
    context.restore();
};
```

Picture 13 속성을 변경 한 지우개의 drawLine 함수(좌) 와 그 결과물(우)

1.3 도형 (원, 사각형)

도형은 기존 Canvas 에서 제공하는 사각형을 그리는 rect 함수와 타원형을 그리는 ellipse 함수를 사용하여 구현하였다. MouseDown 이벤트 발생시 펜 툴과 마찬가지로 아이템의 데이터를 초기화하고 현재 Canvas 의 상태를 저장한다. 이후 MouseMove 이벤트 발생 시 마다 Canvas 를 이전 상태로 되돌리고, MouseDown 의 좌표와 현재 좌표를 바탕으로 도형을 그려줌으로써 도형의 프리뷰를 제공해준다. 그리고 MouseUp 이벤트 발생 시 Canvas 를 이전 상태로 돌리고, MouseDown 발생 좌표와 MouseUp 발생 좌표를 바탕으로 최종 아이템을 Canvas 에 그리고 변수들을 초기화시킨다.

```
const drawLine = ( item, mouse, end = undefined ) => {
    const points = item.points[0];
    const mouseP = item.points[1] ? item.points[1] : mouse;
    const startX = points.x < mouseP.x ? points.x : mouseP.x;
    const startY = points.y < mouseP.y ? points.y : mouseP.y;
    const width = Math.abs(points.x - mouseP.x);
    const height = Math.abs(points.y - mouseP.y);

    context.save();
    context.beginPath();
    context.strokeStyle = item.color;
    context.lineWidth = item.size;
    context.rect(startX, startY, width, height);
    context.stroke();
    context.restore();
};
```

```
const drawLine = ( item, mouse, end = undefined ) => {
    const points = item.points[0];
    const mouseP = item.points[1] ? item.points[1] : mouse;
    const startX = mouseP.x < points.x ? mouseP.x : points.x;
    const startY = mouseP.y < points.y ? mouseP.y : points.y;
    const endX = mouseP.x >= points.x ? mouseP.x : points.x;
    const endY = mouseP.y >= points.y ? mouseP.y : points.y;
    const rediusX = (endX - startX) * 0.5;
    const rediusY = (endY - startY) * 0.5;
    const centerX = startX + rediusX;
    const centerY = startY + rediusY;
    context.save();
    context.beginPath();
    context.lineWidth = item.size;
    context.strokeStyle = item.color;
    context.ellipse(centerX, centerY, rediusX, rediusY, 0, 0, 2 * Math.PI);
    context.stroke();
    context.closePath();
    context.restore();
};
```

Picture 14 사각형을 위한 함수(좌) 와 타원을 위한 함수(우)

2 Undo, Redo

Undo, Redo 기능을 구현하기에 앞서 고민해야 했던 문제가 몇개 있었다. 우선, 여러 사용자가 같은 페이지에서 작업을 하고 있을 때, 각 사용자는 자신의 아이템에 대해서만 Undo, Redo 작업을 수행할 수 있어야 하므로 각 아이템이 어느 사용자의 것인지 구분할 수 있어야 했다. 또한 A,B 사용자가 각각 그린 K 와 L 이 있고 K 위에 L 이 그려져 있는 상태에서, A 사용자가 아이템 K 에 대한 Undo, Redo 작업을 수행하게 되면, 아이템 K 가 L 위에 그려지는 문제가 있었다. 이 문제점들은 아이템 데이터에 식별자 역할로 사용할 변수와 각 페이지별 사용자에 대한 myItemList, RedoList 배열을 추가하여, Undo, Redo 작업 시 해당 변수와 배열을 참고하게 함으로써 해결하였다.

우선 페이지에 현재 아이템이 몇번 입력되었는지에 대한 카운트 변수를 추가한다. 그리고 아이템이 추가될 때마다, 아이템 데이터에 현재 카운트를 식별자로 부여하고 카운트 변수를 증가시킨다. 카운트 변수는 Undo 작업시에도 줄어들지 않고 항상 증가만 하게 하여, 아이템의 순서를 구분할 수 있는 식별자 역할을 수행할 수 있게 하였다. 또, 페이지마다 myItemList 이름의 배열을 추가하여, 자신이 직접 그려서 추가되는 아이템의 경우 해당 아이템의 식별자를 myItemList 배열에 푸시하게 하였다. 이를 통해 식별자를 참조하여 해당 아이템의 순서와, 사용자 본인의 아이템의 식별자를 통해 Undo, Redo 기능을 수행할 수 있게 하였다.

Undo 작업은 myItemList 의 마지막에 있는 식별자를 꺼내어 식별자를 바탕으로 현 페이지의 ItemList 를 탐색하고 아이템을 찾아 ItemList 에서 꺼낸다. 그 후 꺼낸 아이템을 RedoList 에 푸시하고, Canvas 를 초기화한 후 특정 아이템을 제거한 ItemList 를 참조하여 다시 그리게 함으로써 구현하였다.

Redo 작업도 비슷하게, RedoList 의 마지막에 있는 아이템을 꺼내어 식별자를 myItemList 에 푸시하고, 식별자를 참조하여 ItemList 의 몇 번째 아이템이었는지 탐색한다. 그 후 ItemList 의 알맞은 위치에 아이템을 푸시하고, Canvas 를 초기화한 후 아이템을 추가한 ItemList 를 참조하여 다시 그리게 함으로써 구현하였다.

Undo 와 Redo 의 아이템 탐색은 ItemList 에 아이템이 많을 경우 처음부터 탐색하는것에 대한 오버헤드가 발생할 수 있고, 아이템의 식별자를 인덱스로 삼아 찾아가도 해당 위치가 아닐 수 있다. 이 때문에 탐색시점을 식별자에 5 를 뺀 값으로 하여 ItemList 의 탐색을 시작하게 해두었다.

```
undoEvent() {
    let { selectedPage, pageData } = this.props;
    let selectedPageItem = pageData[selectedPage-1].items;

    if( !selectedPageItem.myItem.length ) return;

    let popedMyItemCount = selectedPageItem.myItem.pop();
    let startSearch = popedMyItemCount < 5 ? 0 : popedMyItemCount-5;
    let splicedItemList = null;

    for( ; startSearch < selectedPageItem.itemList.length; startSearch++ ) {
        if( popedMyItemCount == selectedPageItem.itemList[startSearch].count ) {
            splicedItemList = selectedPageItem.itemList.splice(startSearch, 1)[0]
            break;
        }
    }
    if( splicedItemList == null ) return;
    selectedPageItem.undoList.push(splicedItemList);

    this.props.undoItem(selectedPageItem);
    this.refreshCanvasWithItem(selectedPageItem.itemList, selectedPage);
}
```

```
redoEvent() {
    let { selectedPage, pageData } = this.props;
    let selectedPageItem = pageData[selectedPage-1].items;

    if( !selectedPageItem.undoList.length ) return;

    let popedUndoListItem = selectedPageItem.undoList.pop();
    let needCount = popedUndoListItem.count;

    if( !selectedPageItem.itemList.length ||
            needCount > selectedPageItem.itemList[selectedPageItem.itemList.length-1].count )
        selectedPageItem.itemList.push(popedUndoListItem);
    else {
        let startSearch = popedUndoListItem.count < 5 ? 0 : popedUndoListItem.count-5;
        for( ; startSearch < selectedPageItem.itemList.length; startSearch++ ) {
            if( needCount < selectedPageItem.itemList[startSearch].count ) {
                selectedPageItem.itemList.splice(startSearch, 0, popedUndoListItem);
                break;
            }
        }
    }
    selectedPageItem.myItem.push(needCount);

    this.props.redoItem(selectedPageItem);
    this.refreshCanvasWithItem(selectedPageItem.itemList, selectedPage);
}
```

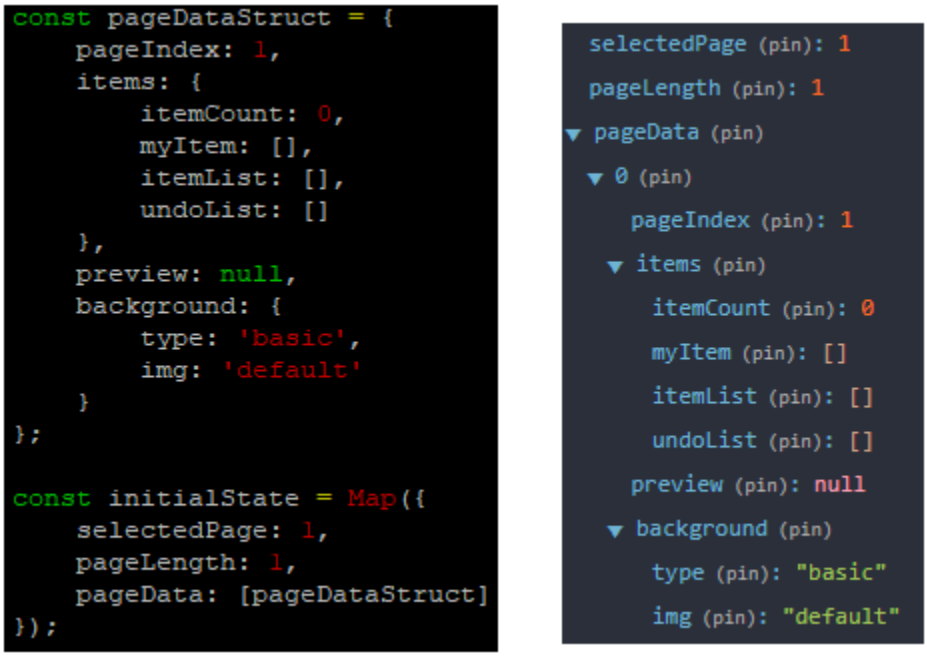
Picture 15 UndoEvent 함수(좌) 와 RedoEvent 함수(우)

3 다중 페이지 및 페이지 전환

각 페이지별로 해당 페이지의 번호, 페이지의 프리뷰, 그리고 해당 페이지에 있는 item 과 관련된 정보를 저장하고, 각 페이지들을 한 배열에 저장하여 관리한다. 그 후 현재 선택 되어있는 페이지의 번호를 저장하는 변수를 두고, 현재 페이지의 번호에 해당하는 페이지 배열의 인덱스에 접근하여 참조,수정을

진행함으로써 다중 페이지와 페이지 전환을 구현하였다.

N 번째 페이지에서 작업을 진행할 경우, 현재 선택되어 있는 페이지의 번호를 저장하는 selectedPage 변수는 N 값을 가지고,현재 Canvas 는 selectedPage 값을 참조하여 페이지 배열의 N-1 인덱스에 위치한 데이터를 참조하여 페이지 N 의 상태를 Canvas 에 표현한다. 그 후 K 번째 페이지로 전환 시 selectedPage 는 K 로 변경되고, Canvas 를 초기화하고 페이지 베열의 K-1 인덱스를 참조하여, 해당 페이지의 ItemList 를 바탕으로 페이지 K 의 Canvas 상태를 다시 표현한다.



```
const pageDataStruct = {
    pageIndex: 1,
    items: {
        itemCount: 0,
        myItem: [],
        itemList: [],
        undoList: []
    },
    preview: null,
    background: {
        type: 'basic',
        img: 'default'
    }
};

const initialState = Map({
    selectedPage: 1,
    pageLength: 1,
    pageData: [pageDataStruct]
});
```

```
selectedPage (pin): 1
pageLength (pin): 1
▼ pageData (pin)
 ▼ 0 (pin)
    pageIndex (pin): 1
  ▼ items (pin)
      itemCount (pin): 0
      myItem (pin): []
      itemList (pin): []
      undoList (pin): []
    preview (pin): null
  ▼ background (pin)
      type (pin): "basic"
      img (pin): "default"
```

Picture 16 페이지 데이터의 구조(좌) 와 페이지 관련 store 의 초기 상태(우)

4 페이지 프리뷰

각 페이지별로 해당 페이지에 대한 프리뷰를 볼 수 있는 기능을 페이지의 Canvas 에서 이미지 데이터를 가져와 저장해두어 PageBar 의 img 태그에 뿌려줌으로써 구현하였다. 그리고 자신의 그림작업 하나가 완료되는 시점 (MouseUp 이벤트 발생) 이나, 다른 사용자의 최종 결과물을 받아올 때, Canvas 에 아이템을 그리고 현재의 Canvas 의 이미지 데이터로 해당 페이지의 프리뷰를 업데이트 하도록 하였다. 하지만 여기서, 한가지 문제점이 발생하였다. 다른 사용자가 다른 페이지에서 작업을 수행할 때에, 다른 페이지에 대한 프리뷰를 어떻게 업데이트할지에 대한 문제였다.

이를 해결하기 위해, 우선 프리뷰 업데이트에 사용 할, 사용자에게 보이지 않는 Canvas(이하 프리뷰 캔버스)를 추가하고, 새 툴 셋을 추가해 프리뷰 캔버스를 인자로 넣어 초기화하였다. 그 후 socket.io 를 통해 다른 사용자의 아이템을 받을 때, 해당 아이템이 어느 페이지의 아이템인지에 따라, 다른 작업을 수행하도록 하여 해결하였다.

공유받은 아이템이 현재 보고 있는 페이지일 경우는 기존의 방법과 같이, 아이템을 받은 시점에서 Canvas 에 그림 데이터를 적용하고, Canvas 의 이미지 데이터를 통해 프리뷰를 업데이트 하도록 한다.

그리고 다른 페이지의 아이템일 경우, 해당 페이지의 ItemList 에 아이템을 추가하고, ItemList 를 참조하여 프리뷰 캔버스에 페이지 아이템들을 그린다. 그 후 프리뷰 캔버스의 이미지 데이터를 통하여 프리뷰를 업데이트하고, 프리뷰 캔버스를 초기화한다.

5 실시간 공유

아이템의 공유는 특정 사용자가 한 그림작업을 마쳤을 때, socket.io 를 통해 서버에 데이터를 전송하고, 서버에서 socket.broadcast 를 통해 다른 사용자들에게 전송하게 하고, Canvas 에서 해당 데이터를 받아 그려주는 방식으로 하면 되었다. 하지만 실시간 공유의 경우는 좀 달랐다.

아이템의 정보를 실시간으로 공유하기 위해서는 공유의 단위와 공유 주기를 줄여야했다. 펜과 지우개 툴은 그림작업의 종료시점에서 공유하게 된다면, 한 클릭으로 많은양의 그림을 그릴경우 그만큼의 공유 시간이 지연되어 결국 실시간 공유가 아니게 되어버린다. 따라서 한 작업이 마무리 되는 시점에서 최종 결과물을 공유하는게 아닌, 작업을 진행하고 있는 도중에도 아이템의 상태가 공유되어야 실시간 공유를 할 수 있었다.

실시간 공유를 구현하기 위해서 그림데이터를 공유하는 주기를 각 작업별로 변경하였다. 우선 툴 작업중 MouseMove 움직임이 실제 Canvas 에 반영되는게 아닌 도형 그리기는 해당 작업이 종료되는 MouseUp 이벤트 발생 시, 해당 아이템의 최종 데이터만 전송한다. 그리고 MouseMove 때도 Canvas 에 작업이 반영되는 펜과 지우개는 MouseMove 이벤트 발생 시마다 그 시점에 아이템의 상태를 공유함으로써, 최대한 실시간 공유가 되도록 하였다.

페이지 추가, Undo, Redo 의 경우는 버튼을 누를 경우에 작업이 공유가 되도록 하였다. 페이지 추가의 경우 특별한 데이터를 전송하지 않고 신호만 전달하여 다른 사용자들도 페이지 추가 작업을 진행하도록 하였다. Undo 의 경우 Undo 하려는 아이템의 페이지와 식별자를 전송하고, Redo 의 경우 Redo 하려는 아이템의 페이지와 아이템의 데이터를 전송한다.

이후, 클라이언트에서 socket.io 로 받는 이벤트들에 대하여, 각 이벤트에 알맞은 데이터 로직을 작성함으로써 최대한 실시간 공유 환경을 구성하였다.

IV. 개선 필요 사항

1 펜 보정 작업과 다중입력

현재 펜 툴은 MouseDown 시 해당 Canvas 의 상태를 저장해두었다가, MouseMove 발생 시 마다 Canvas 를 이전상태로 돌리고 해당 아이템을 다시 그리는 작업을 한다. 하지만 여러 사용자가 같은 페이지에 펜 툴로 동시에 입력하게 될경우, 각 사용자별로 저장해 둔 Canvas 의 상태가 다 다르고, 저장해둔 Canvas 의 상태가 최신상태가 아니게 된다. 이로 인해 자신이 그리는 도중에 다른 사용자가 펜 작업을 진행하게 될경우, 다른 사용자의 작업이 깜빡거리는 현상이 나타난다.

예로, 두 사용자 A,B 가 있고, 다른 시점에 작업을 시작한다. 이때, 마우스를 떼지 않고 계속 그린다고 가정해보자. 사용자 A 가 펜으로 그림을 그리는 도중에, 사용자 B 가 펜 작업을 시작했다. A 가 MouseMove 이벤트를 발생시킬 때 마다 이전의 Canvas 상태로 되돌린다. 이때, A 의 펜 툴은 B 가 작업을 시작하기 이전의 상태를 가지고 있으므로 B 가 그린 작업이 있을 경우 A 의 화면에선 B 의 작업이 사라지게 된다. 그리고 B 가 아이템을 그릴경우 B 작업이 공유되고, A 화면에 B 의 작업이 다시 나타나게 된다. 이 상황이 반복되면, A 의 화면에 B 작업이 깜빡깜빡 거리는 현상이 일어난다.

이는 펜 툴의 보정작업이 Canvas 의 상태 저장을 하고 복구하는 과정에서 일어나고, 임시 저장해둔 Canvas 의 상태가 사용자들간에 동기화가 되지 않아 발생하는 이슈이다. 이를 해결하여 다중입력의 문제점을 해결하여야 한다.

펜 보정작업에 대한 또다른 문제점은 작업 공유 시에 발생한다. 위에 기술한 것과 같이 펜은 MouseDown 시 해당 Canvas 의 상태를 저장하고, MouseMove 발생 시 마다 Canvas 를 이전상태로 돌리고 아이템을 다시 그리며 보정작업을 수행한다. 실시간 공유는 MouseMove 이벤트가 발생할 때 전송해주는데, 다른 사용자의 중간 진행상황을 받아올 때 툴의 drawLine 함수를 직접 호출하여 그리는 방식을 적용하였더니, 받는 사용자의 Canvas 에서는 보정작업이 이루어지지 않는 현상이 발견되었다.

이를 해결하려고 MouseDown, MouseMove, MouseUp 전송을 구분하기위해 아이템에 Identifier 변수와 type 변수를 선언하고, 툴의 onMouseDown 함수에서 아이템을 초기화할 때 Identifier 에 고유 식별자를 두었다. 그 후 각 마우스 이벤트에서 공유할 때, type 변수에 각각 'down', 'move', 'up' 을 대입하여 전송하고, 아이템을 받았을 때 type 별 데이터 처리를 구분하여 어느정도 해결하였다. 하지만 Identifier 는 현재 Math.random 함수로 그냥 임의의 수로 초기화를 시켜뒀고, 각 Identifier 별로 펜 툴을 생성-사용-삭제하기 때문에 상당히 비효율적이라 생각한다. 따라서 실시간 공유에 관한 펜 보정작업 역시 개선이 필요하다.

```
wSocket.on('getonDrawData', (data) => {
    if( data.pageIndex == this.props.selectedPage ) {
        if( data.type == 'down' ) {
            this.shareTool[data.identifier] = Pencil(this.canvasContext, this.canvas);
            this.shareTool[data.identifier].onMouseDown(
                data.points[0].x, data.points[0].y, data.color, data.size
            );
        } else if ( data.type == 'move' ) {
            let { x, y } = data.points[data.points.length-1];
            this.shareTool[data.identifier].onMouseMove(x,y);
        } else if ( data.type == 'up' ) {
            let { x, y } = data.points[data.points.length-1];
            this.shareTool[data.identifier].onMouseUp(x,y);
            delete this.shareTool[data.identifier];
        }
    }
});
```

Picture 17 현재 아이템의 중간상태를 받아 처리하는 socket.io 이벤트

생각하고 있는 개선안으로는 Identifier 를 이용하여 툴을 생성-사용-삭제 하는 부분을 최대한 활용하고,

펜의 MouseMove 이벤트 때 공유하는 데이터를 아이템의 전체 데이터가 아닌 이벤트의 좌표만 전달하는것이다. 이 방법은 Identifier 의 고유성이 보장되도록 설정해야 하지만, 현재 MouseMove 이벤트 발생 시 공유하는 데이터의 양을 많이 줄일수 있어 실시간 공유의 최적화도 기대할 수 있다.

2 실시간 공유

실시간 공유의 경우, 현재 다른 불필요한 작업의 실시간 공유를 최소화하고, 펜과 지우개만 MouseMove 발생 시 마다 공유하도록 하여 실시간 공유를 구현하였다. 이는 MouseMove 발생 시 마다 공유를 하기에, 서버가 조금 부담을 가질 수 있다. 이는 여러 사용자가 작업을 동시에 진행할경우, 작업하는 사용자에 수에 비례하여 부담이 늘아난다

또, MouseMove 발생 주기에 따른 실시간 공유의 속도보다 공유받는 사용자의 클라이언트가 받아 처리하는 속도가 더 느려, 실시간 공유를 해줌에도 0.2~0.5s 의 딜레이가 발생한다. 이는 과도한 이벤트 입력의 영향과 인터넷 통신에 의한 한계로 생각되고,사용자의 인터넷 환경에 따라 더 늘어날 수 있다.

따라서 공유받는 사용자의 클라이언트가 받아서 처리하는 속도에 맞춰 공유 주기를 제한하면, 서버의 부담을 줄어들게 할 수 있고 과도한 이벤트 입력을 방지하여 속도와 성능의 향상을 기대할 수 있을 것 같다.

3 Undo, Redo

Undo, Redo 작업은 식별자를 바탕으로 ItemList 에서 작업대상이나 위치를 찾고 그에 알맞은 작업을 수행한다. 그리고 Canvas 를 초기화하고 변경된 ItemList 를 참조하여 Canvas 에 아이템들을 다시 그린다. 이는 탐색을 처음부터 시작하고 현재 N 개의 아이템이 있다고 가정했을 때, 탐색시간에 최대 O(n) 만큼의 시간이 걸리고, 아이템들을 다시 그리는데 O(n-1)만큼의 시간이 걸린다. 이는 탐색과 Canvas 에 다시 그리는 작업을 개별적으로 수행하여 ItemList 의 순회를 두번 하기 때문이다.

개선 방안으로는 우선 Canvas 의 초기화작업을 먼저 진행하고 ItemList 를 처음부터 순회하게 한다. 그리고 각 아이템이 찾는 아이템이나 위치라면 Undo,Redo 에 맞는 작업을 수행하고, 아니라면 Canvas 에 그리고 다음 아이템을 검사하는 방식의 로직을 적용한다. 이를 통해 탐색과 다시 그리는 작업을 합하여, ItemList 의 순회 횟수를 한번으로 줄이는 방향으로 개선할 수 있을 것 같다.

4 페이지 프리뷰

현재 프리뷰는, 보고 있는 페이지와 다른 페이지의 프리뷰 업데이트를 위해 사용자에게 보이지 않는 Canvas 태그를 추가하여 사용하고 있다. 이는 사용자에게 보이지 않음에도 DOM 에 추가된 Canvas 태그를 랜더링하게 되어 불필요한 자원을 낭비하고 있다고 판단하고 있다.

생각하고 있는 개선방법은 다른 페이지에 대한 프리뷰 업데이트에 사용할 Canvas 를 document.createElement 메소드를 통해 엘리먼트를 생성하되, DOM 에 추가하지 말고 그냥 사용하는것이다. 이는 이전의 방법보다 자원사용의 효율성이 좋은지 알아봐야 할 필요가 있다. 만약 DOM 에 추가하지 않아서 랜더링에 소모되는 자원에 영향을 끼치지 않다면, 해당 개선방법을 통해 효율성을 개선할 수 있을 것 같다.

5 데이터 처리 로직

IV.1~4 에서 기술한 내용 외에도, 현재 프로젝트에는 기능의 구현만을 주로 생각하여 코드를 작성하였기에 상당히 비효율적인 로직이 많이 존재한다. Canvas 를 초기화하고 다시 그리는 것과 같이 많이 사용하는 로직을 필요한곳에 모두 코드로 직접 작성하여 가시성이 좋지 않고 컴포넌트의 코딩 라인이 길어졌다.

자주 사용하는 데이터 로직들을 함수화시키고, 기능을 수행하는 함수에서 필요한 로직의 함수를 호출하는 구조로 코드를 변경하면, 가시성과 재활용성이 늘어나고, 코딩 라인이 축소될것으로 예상한다.