Part II

기본 게임 구현하기

제4장 게임 구현하기

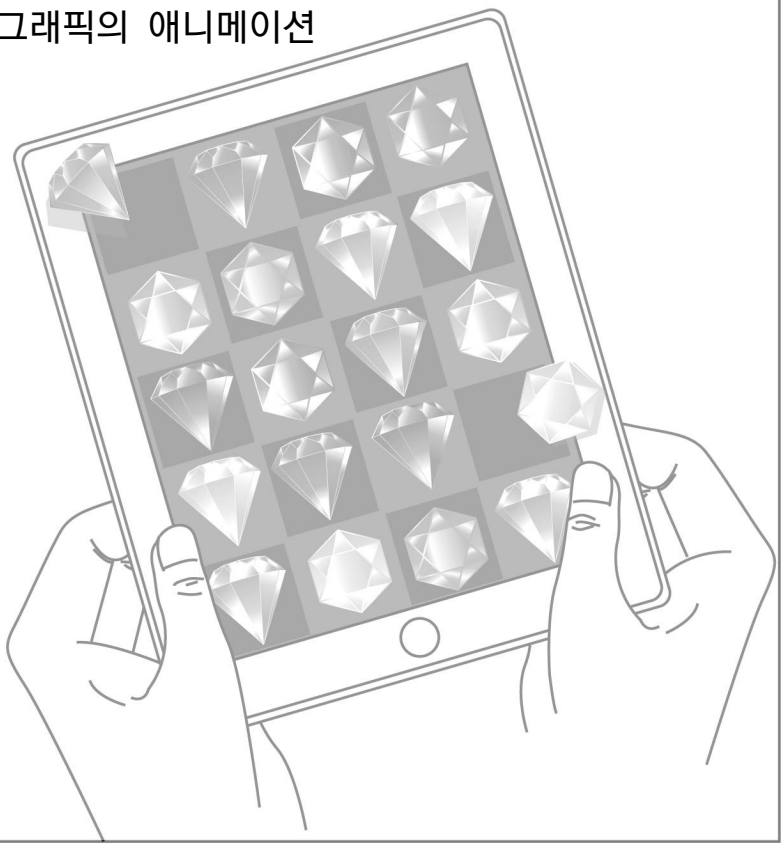
제5장 웹 작업자 스레드의 활용

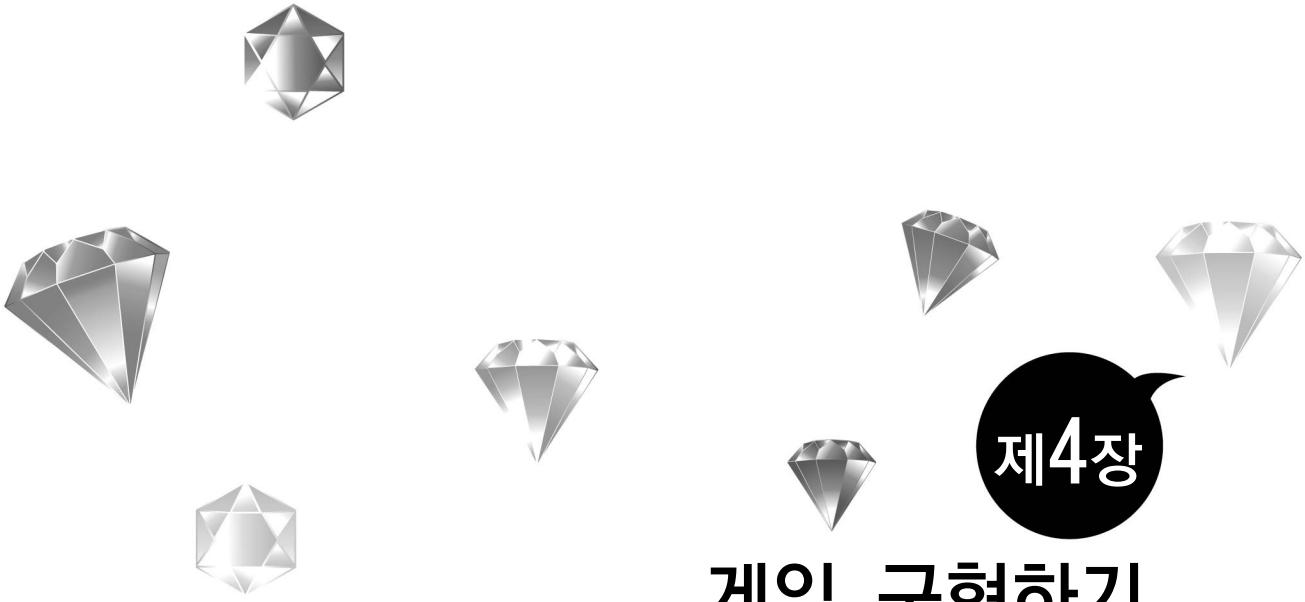
제6장 Canvas를 이용한 그래픽의 처리

제7장 게임 화면 구현하기

제8장 사용자와 게임의 상호작용

제9장 게임 그래픽의 애니메이션





제4장

게임 구현하기

이 장의 주요 내용

게임 보드 모듈 구현하기
게임의 상태 캡슐화하기
보석 블록 보드 설정하기
게임 규칙 구현하기
보석 블록의 교환 처리하기

이번 장에서는 보석 블록 보드를 제어하기 위한 규칙과 게임의 공식들을 구현하는 방법을 소개한다. 즉, 필요한 모든 게임의 로직을 살펴보고 애플리케이션의 나머지 기능들과 상호작용할 수 있는 게임 보드를 구현할 것이다. 또한 모든 게임 규칙이 이루어졌을 때만 게임 보드의 상태를 수정할 수 있는 독립적인 모듈을 통해 게임 보드의 상태를 캡슐화하는 방법도 소개한다. 추가로, 사용자가 보석 블록의 위치를 변경하면 이에 따라 나머지 블록들을 게임 보드가 자동으로 재정렬할 수 있도록 할 것이다.

또한 이번 장에서는 게임이 보드를 제어하기 위한 두 가지 방법을 제공해야 할 때 발생할 수 있는 문제에 대해서도 이야기하고자 한다. 플레이어가 혼자 게임을 플레이하는 사용자 로컬 환경에서의 클라이언트 측 게임 로직은 이번 장에서 설명할 예정이지만, 이 게임은 동일한 규칙으로 서버 측에서도 동작해야 한다.

게임 보드 모듈 구현하기

이 게임의 핵심적인 규칙은 게임의 화면이나 입력에 대한 요소들과는 독립적이다. 이번 장에서 구현할 게임 보드 모듈은 게임 상태를 표현하기 위한 데이터 모델이며, 특히 현재의 보석 블록의 레이아웃과 관련이 있다. 이 모듈은 주로 게임 화면 모듈과 같은 다른 모듈들이 게임의 상태를 제어할 수 있도록 메서드를 노출한다. 게임 보드는 게임 화면의 백엔드 모듈로서 동작하기 때문에 이번 장에서 작성하는 코드는 게임에 시각적 요소를 추가하거나 하지는 않는다.

엄밀히 말하자면, 보드 모듈은 단지 보석 블록 보드를 표현하는 모듈이다. 이 모듈이 외부로 노출하는 기능은 보드 상의 보석 블록을 탐색하는 기능과 보석 블록을 교환하는 기능이다. 보석 블록의 교환은 특정 조건을 만족하는 경우에만 가능하기 때문에 교환 기능은 반드시 보석 블록의 쌍을 교환하도록 구현해야 한다. 또한 보석 블록이 교환되면 교환된 보석 블록이 사라지고 새로운 보석 블록이 내려오는 등의 후속 작업이 필요하다. 게임 보드 모듈은 이러한 상태와 반응들을 자동으로 관리하고 처리해야 한다. 이 모듈은 script 폴더의 board.js 파일에 구현한다. 예제 4.1은 이 모듈의 기본 구현을 보여준다.

예제 4.1 게임 보드 모듈의 기본 구현

```
jewel.board = (function() {

    /* 게임의 기능을 여기에 구현한다. */

    return {
        /* 노출되는 기능을 여기에 정의한다. */
    }

})();
```

그런 후 예제 4.2에서와 같이 loader.js 파일에서 board.js 파일을 두 번째 단계에서 로드하도록 추가해준다.

예제 4.2 게임 보드 모듈 로드하기

```
// 스테이지 2 로드
if (Modernizr.standalone) {
    Modernizr.load([
    {
        load : [
            "scripts/screen.main-menu.js",
            "scripts/board.js"
        ]
    }
    ]);
}
```

지금까지 아무런 기능이 없는 게임 보드 모듈의 뼈대를 구현하였다. 이제 이 모듈에 첫 번째 메서드를 추가해보자.

게임 상태 초기화하기

게임 보드 로직에는 행과 열의 개수, 게임에 사용될 보석 블록의 종류 등 몇 가지 설정이 필요하다. 이런 설정들은 실제 게임 코드와는 별개로 분리해야 전체 코드에 영향을 미치는 일 없이 수정하기가 편리하다. 예제 4.3은 loader.js 파일에서 jewel 네임스페이스에 새로운 설정 객체를 추가하는 코드이다.

예제 4.3 설정 객체 추가하기

```
var jewel = {
    screens : {},
    settings : {
        rows :  8,
        cols : 8,
        baseScore : 100,
        numJewelTypes : 7
    }
}
```

rows와 cols 변수는 게임 보드의 크기를 행과 열로 정의하기 위한 것이다. 이 게임은 작은 화면에서도 잘 동작할 수 있도록 8x8 크기의 격자를 사용한다. baseScore 변수는 보석 블록을 연쇄적으로 맞추었을 때 각 보석 블록마다 매겨질 점수를 의미한다. 나중에 보게 되겠지만 세 개를 초과하는 보석 블록이 연결되었을 때는 이 값은 배가 된다. 기본 점수는 보석 블록마다 100점이지만 이 점수는 여러분이 얼마든지 바꿀 수 있다. 마지막으로 numJewelTypes 변수는 보석 블록의 종류를 의미한다. 이 숫자는 보석 블록의 스프라이트 숫자와 동일하기도 하다.

이제 가장 중요한 게임 보드 모듈을 비롯한 게임의 나머지 모듈에서 이 설정 객체를 사용할 수 있게 되었다.

게임 보드 초기화하기

게임 보드 모듈을 구현하기 위해 첫 번째로 게임 보드를 마련하고 초기화하는 함수가 필요하다. 게임이 시작되면 게임 보드는 이미 임의의 보석 블록들로 채워져 있어야 한다. 예제 4.4는 board.js 파일에 추가된 initialize() 함수를 구현한 코드이다.

예제 4.4 초기화 함수

```
jewel.board = (function() {
    var settings,
        jewels,
        cols,
        rows,
        baseScore,
        numJewelTypes;
```

```
    function initialize() {
        settings = jewel.settings;
        numJewelTypes = settings.numJewelTypes,
        baseScore = settings.baseScore,
        cols = settings.cols;
        rows = settings.rows;
        fillBoard();
    }

    function print() {
        var str = "";
        for (var y = 0; y < rows; y++) {
            for (var x = 0; x < cols; x++) {
                str += getJewel(x, y) + " ";
            }
            str += "\r\n";
        }
        console.log(str);
    }

    return {
        initialize : initialize,
        print : print
    };
})();
```

예제 4.4에서 가장 먼저 수행한 작업은 몇 개의 변수를 선언한 것이다. 첫 번째 변수는 앞서 추가했던 설정 객체를 가져오기 위한 것이며, 두 번째 변수인 jewels는 보석 블록 보드의 현재 상태를 표현할 2차원 배열이다. 각 보석 블록은 jewels 배열 내에 각 보석 블록의 종류를 의미하는 정수 값으로 표현된다. 이렇게 배열을 이용하면 각 블록의 상태에 따라 개별적으로 블록에 액세스하기가 쉬워진다. 예를 들어, 아래의 코드는 보드 상의 특정 위치(x=3, y=2)에 있는 보석 블록의 종류를 조회하는 코드이다.

```
var type = jewels[3][2];
```

나머지 변수들은 설정 모듈의 값을 참조하기 위한 변수들이다. 이 값들에 대한 설명은 다음 섹션으로 잠시 미루자. print() 함수는 디버깅을 위한 함수로 게임 보드의 데이터를 자바스크립트 콘솔에 출력한다. 콘솔에 아래의 구문을 입력하면 게임 보드 모듈을 초기화할 수 있다.

```
jewel.board.initialize()
```

이번 장에 걸쳐 게임 보드 데이터를 조회하고 싶다면 언제든지 콘솔에 아래와 같이 입력하면 된다.

```
jewel.board.print()
```

비동기 함수의 활용

다음 섹션을 진행하기에 앞서 나중을 위해 initialize() 함수를 먼저 수정하고자 한다. 제5장에서는 웹 작업자(web workers)를 이용하여 이번 장에서 구현한 것과 동일한 메서드를 제공하지만, 별도의 스레드(thread)에서 동작하는 웹 작업자 기반 보드 모듈을 구현할 것이다. 웹 작업자는 비동기 메시징 API를 이용하여 애플리케이션의 다른 모듈들과 통신하게 되며, 따라서 웹 작업자 기반 보드 모듈이 노출하는 모든 메서드는 반드시 비동기 형식으로 사용되어야 한다.

마찬가지로 보드 데이터의 유효성 검사를 위한 서버 측 보드 모듈을 추가할 때도 비동기 AJAX 요청을 서버에 전달해야 한다. 게임의 상태를 수정하는 모든 함수는 반드시 서버와의 통신을 필요로 하며, 호출자에게 응답이 도착하기 전에 리턴되어야 한다. 다시 말하면, 함수가 리턴되었다는 것이 어떤 결과가 준비되었다는 것을 의미하지는 않는다.

이 지연 응답(deferred response) 문제는 제대로 구현된 사용자 정의 이벤트 전송 시스템을 사용하거나 혹은 CommonJS와 Node.js에서 사용되는 약속 객체(promise object) 개념을 도입하는 등 몇 가지 방법으로 해결할 수 있다. 가장 간단한 방법은 관련 메서드에 콜백 함수를 제공하여 서버 측의 작업이 완료된 후에 콜백 함수를 호출하도록 구현하는 방법이다. 아마도 여러분은 자바스크립트의 window.setTimeout()이나 DOM 객체의 addEventLister() 메서드와 같은 패턴을 이미 알고 있을 것이다. 이런 함수들은 다른 함수를 매개변수로 전달받아 자신이 해야 할 작업을 완료하는 시점에 호출한다. 그러면 board.js 파일을 열고 initialize() 함수를 예제 4.5와 같이 수정해보자.

예제 4.5 콜백 함수를 이용한 초기화 함수

```
jewel.board = (function() {
    ...
    function initialize(callback) {
        numJewelTypes = settings.numJewelTypes,
        baseScore = settings.baseScore;
        cols = settings.cols;
        rows = settings.rows;
        fillBoard();
        callback();
    }
    ...
})();
```

자바스크립트 콘솔을 통해 게임 보드를 초기화하려면 다음과 같이 빈 함수를 전달하면 된다.

```
jewel.board.initialize(function() {})
```

이제 initialize() 함수 내의 모든 동작이 즉각적으로 발생하므로 결과는 콜백 함수를 사용하지 않은 경우와 완전히 동일하다. 가장 큰 차이점은 웹 작업자 모듈을 적용했을 때 동일한 함수를 사용해도 더 쉽게 통합할 수 있다는 점이다.

게임 보드 채우기

예제 4.5에서 사용한 fillBoard() 함수는 반드시 cols×rows 개수의 격자를 만들고 임의의 보석 블록을 채워야 한다. board.js 파일에 예제 4.6의 코드를 작성해보자.

예제 4.6 보석 보드 채우기

```
jewel.board = (function() {
    ...
    function fillBoard() {
        var x, y;
        jewels = [];
        for (x = 0; x < cols; x++) {
            jewels[x] = [];
            for (y=0; y < rows; y++) {
```

```
                jewels[x][y] = randomJewel();
            }
        }
    }
    ...
})();
```

보석 블록의 종류는 randomJewel() 함수를 통해 결정되며, 이 함수는 단순히 0부터 (numJewelTypes−1) 사이의 값을 리턴한다. 예제 4.7은 이 randomJewel() 함수를 구현한 코드이다.

예제 4.7 임의의 보석 블록 생성하기

```
jewel.board = (function() {
    ...
    function randomJewel() {
        return Math.floor(Math.random() * numJewelTypes);
    }
    ...
})();
```

이제 게임 보드를 임의의 보석 블록으로 채우는 기본 알고리즘이 구현되었다. 사실 이 방법은 완벽하지 않으며, 결과적으로 게임 보드가 잘 구성된다는 것을 보장할 수도 없다. 보석 블록을 임의로 선택하기 때문에 어쩌면 보드 상에 벌써 한두 개의 연결된 보석 블록이 있을 수도 있다. 이럴 경우 플레이어가 아무런 동작도 하지 않고 점수를 얻을 수 있기 때문에 게임이 시작하는 시점에는 연결된 블록이 존재하면 안 된다. 이런 상황이 발생하지 않도록 하기 위해서는 fillBoard() 함수가 두 개 이상의 동일한 보석 블록이 연결되지 않도록 블록을 선택할 수 있어야 한다.

앞서 구현한 블록 채우기 알고리즘은 게임 보드의 왼쪽 상단에서 시작해서 오른쪽 하단에서 끝난다. 이는 현재 채워지고 있는 위치의 왼쪽 위에는 보석 블록이 존재한다는 것을 의미한다. 보석 블록이 없어지려면 적어도 세 개의 동일한 보석 블록이 필요하며, 따라서 임의로 채워진 보석 블록이 하나의 연결을 이루려면 왼쪽 혹은 위쪽으로 두 개의 동일한 보석 블록이 존재해야 한다. 이 게임이 사용하는 것처럼 상대적으로 작은 크기의 게임 보드에서는 간단한게 이 문제를 해결할 수 있다. 예제 4.8은 변경된

채우기 루틴을 구현한 코드이다.

예제 4.8 초기화 단계에서 연결된 블록 제거하기

```
jewel.board = (function() {
    ...
    function fillBoard() {
        var x, y,
            type;
        jewels = [];
        for (x = 0; x < cols; x++) {
            jewels[x] = []’
            for (y = 0; y < rows; y++) {
                type = randomJewel();
                while ((type === getJewel(x - 1, y) &&
                        type === getJewel(x - 2, y)) ||
                       (type === getJewel(x, y - 1) &&
                        type === getjewel(x, y - 2))) {
                    type = randomJewel();
                }
                jewels[x][y] = type;
            }
        }
    }
    ...
})();
```

이 알고리즘에는 새로 추가할 보석 블록이 다른 블록과 연결된 상태가 아닐 때까지 또 다른 종류의 보석 블록을 계속해서 선택하는 반복문이 추가되었다. 대부분의 경우 임의로 선택된 보석 블록이 연결된 상태가 될 수는 없지만 극히 드물게 그런 현상이 발생하는 경우에는 다른 보석 블록으로 빠르게 대체가 가능하다.

그러나 예제 4.9에서 사용한 것과 같은 반복문에 게임 영역을 올바르게 표시하지 않으면 게임 보드를 벗어난 위치를 검사하는 경우도 발생할 수 있다. 그런 이유로 이 예제에서는 jewels 배열을 직접 참조하지 않고 getJewel()이라는 함수를 이용함으로써 보드 영역을 벗어난 위치를 참조하는 경우를 대비하고 있다. 이 함수는 예제 4.9와 같이 구현하였다.

예제 4.9 보드 영역 내에서 지정된 위치의 보석 블록 참조하기

```
jewel.board = (function() {
    ...
    function getJewel(x, y) {
        if (x < 0 || x > cols - 1 || y < 0 || y > rows - 1) {
            return -1;
        } else {
            return jewels[x][y];
        }
    }
    ...
})();
```

getJewel() 함수는 x나 y 변수의 값이 0보다 작거나 혹은 (cols−1)이나 (rows−1)보다 큰 경우에는 게임 보드 영역을 벗어난 위치로 판단하여 −1을 리턴한다. 또한 보석 블록의 종류는 0부터 numJewelTypes−1사이의 값이기 때문에 getJewel() 함수가 −1을 리턴한 경우에는 실제로 존재하지 않는 종류의 보석 블록이 되어 미리 연결된 보석 블록이 만들어질 수 없다.

게임 규칙 구현하기

이제 기본적인 게임 보드가 완성되었으므로 보석 블록을 교환하는 부분을 구현해보자. 게임 보드 모듈은 서로 교환될 보석 블록의 위치를 매개변수로 전달받아 해당 보석 블록을 서로 교환하는 swap() 메서드를 노출한다. 당연히 올바른 위치의 두 블록만 서로 교환되며, 그렇지 않은 경우에는 아무런 처리를 하지 않는다. 그러면 먼저 교환할 두 블록의 위치에 대한 유효성 검사를 수행하는 코드부터 작성해보자.

블록 교환에 대한 유효성 검사

블록 교환은 세 개 이상의 동일한 보석 블록의 연결이 최소한 하나 이상일 때만 유효하다. 이를 검사하기 위해 특정 위치의 보석 블록이 연결된 블록들의 일부인지를 검사하는 checkChain() 함수를 구현해보자. 이 함수는 지정된 위치의 보석 블록의 종류를 알아낸 뒤 그 왼쪽과 오른쪽에 위치한 보석 블록 중 동일한 종류의 블록의 개수를 확인한다. 물론 동일한 탐색을 위와 아래쪽으로도 수행해야 한다. 수평 혹은 수직으로

탐색한 결과 동일한 블록의 개수가 2개(지정된 위치의 블록까지 포함하면 3개) 이상이면 하나의 연결이 만들어진 것이다. 그러면 board.js 파일을 열고 예제 4.10의 코드를 참조하여 checkChain() 함수를 작성해보자.

예제 4.10 블록 연결을 검사하는 함수

```
jewel.board = (function() {
    ...
    // (x,y) 좌표 값을 포함하여 가장 길게 연결된 보석 블록의 개수를 리턴한다.
    function checkChain(x, y) {
        var type = getJewel(x, y),
            left = 0, right = 0,
            down = 0, up = 0;

        // 오른쪽으로 탐색
        while (type === getJewel(x + right + 1, y)) {
            right++;
        }

        // 왼쪽으로 탐색
        while (type === getJewel(x - left - 1, y)) {
            left++;
        }

        // 위쪽으로 탐색
        while (type === getJewel(x, y + up + 1)) {
            up++;
        }

        // 아래쪽으로 탐색
        while (type === getJewel(x, y - down - 1)) {
            down++;
        }

        return Math.max(left + 1 + right, up + 1 + down);
    }
    ...
})();
```

checkChain() 함수는 불린(boolean) 값 대신 자신이 발견한 연결된 블록의 개수를 리턴한다. 이 결과 값은 향후 점수를 계산할 때 사용할 수 있는 추가 정보를 제공한다. 이제 주어진 위치로부터 블록의 연결 여부를 판단할 수 있게 되었으므로 블록의 교환이 가능한지 여부를 판단하는 것이 상대적으로 쉬워졌다. 첫 번째 조건은 두 지점이 서로 인접해야 한다는 것이다. 즉, 서로 붙어 있는 보석 블록만 교환이 가능하다는 것이다. 지정된 두 블록이 서로 붙어 있다면 교환이 가능하며, 따라서 두 블록의 종류를 서로 맞바꾸면 된다. 블록의 교환이 성공적이라면 checkChain() 함수는 한 지점에 대해 2보다 큰 값을 리턴해야 한다. 그런 후에는 보석 블록을 다시 원위치시킨 후 checkChain() 함수의 리턴 값에 따라 판단 결과를 리턴한다. 예제 4.11은 이런 유효성 검사를 수행하는 canSwap() 함수를 구현한 코드이다.

예제 4.11 블록 교환에 대한 유효성 검사

```
jewel.board = (function() {
    ...
    // (x1,y1)과 (x2,y2)가 교환이 가능한 경우에는 true를 리턴한다.
    function canSwap(x1, y1, x2, y2) {
        var type1 = getJewel(x1, y1),
            type2 = getJewel(x2, y2),
            chain;

        if (!isAdjacent(x1, y1, x2, y2)) {
            return false;
        }

        // 보석 블록을 임시로 교환한다.
        jewels[x1][y1] = type2;
        jewels[x2][y2] = type1;
        chain = (checkChain(x2, y2) > 2
              || checkChain(x1, y1) > 2);

        // 보석 블록을 다시 원위치한다.
        jewels[x1][y1] = type1;
        jewels[x2][y2] = type2;

        rethrn chain;
    }

    return {
```

```
        canSwap : canSwap,
        ...
    }
})();
```

canSwap() 함수 내에서는 또 다른 함수인 isAdjacent() 함수를 사용하고 있다. 이 함수는 두 지점이 서로 인접하면 true를, 그렇지 않으면 false를 리턴한다. 이 함수는 소위 맨하탄 거리(Manhattan Distance)라고 불리는 두 좌표 사이의 거리를 이용하여 두 지점이 서로 인접한지를 판단한다. 만일 두 지점이 인접하다면 두 지점 사이의 거리의 합은 반드시 1이어야 한다. 예제 4.12는 isAdjacent() 함수를 구현한 코드이다.

예제 4.12 인접한 두 지점인지 확인하는 함수

```
jewel.board = (function() {
    ...
    // (x1,y1)과 (x2,y2)가 인접해 있으면 true를 리턴한다.
    function isAdjacent(x1, y1, x2, y2) {
        var dx = Math.abs(x1 - x2),
            dy = Math.abs(y1 - y2);
        return (dx + dy === 1);
    }
    ...
})();
```

게임 보드 모듈을 초기화한 후에는 자바스크립트 콘솔을 통해 canSwap() 함수가 정상적으로 동작하는지 테스트할 수 있다. print() 함수를 이용하여 게임 보드의 데이터를 확인하고 jewel.board.canSwap(4,3,4,2)와 같은 구문을 이용하여 두 보석 블록의 교환 가능 여부를 테스트할 수 있다.

연결된 보석 블록 탐지하기

보석 블록을 교환한 후에는 제거해야 할 연결된 보석 블록이 있는지 검색해야 한다. 일단 보석 블록이 교환되면 관련된 몇 개의 보석 블록이 제거의 대상이 된다. 우선은 교환된 두 보석 블록이 연결된 블록의 일부가 되겠지만, 이 블록들이 제거되면 나머지 보석 블록들이 내려오고 게임 보드 위쪽으로 새로운 보석 블록들이 채워져야 한다.

결국 다시 한 번 게임 보드 내에 연결된 블록들이 존재하는지 검사해야 하기 때문에 이제 상황이 쉽지 않아졌다. 보드 내에 모든 연결된 블록을 검사하는 유일한 방법은 전체 보드를 모두 검사하는 방법뿐이다. 이때 앞서 구현했던 checkChain() 함수를 이용하면 어렵지 않게 이 문제를 해결할 수 있다. 예제 4.13은 전체 게임 보드에서 연결된 블록들을 탐색하는 getChains() 함수를 구현한 코드이다.

예제 4.13 게임 보드에서 연결된 블록을 탐색하는 함수

```
jewel.board = (function() {
    ...
    // 연결된 블록들을 2차원 배열로 리턴한다.
    function getChains() {
        var x, y,
            chains = [];

        for (x = 0; x < cols; x++) {
            chains[x] = [];
            for (y = 0;y < rows; y++) {
                chains[x][y] = checkChain(x, y);
            }
        }
        return chains;
    }
    ...
})();
```

getChains 함수의 마지막 부분에서 리턴하는 chains 변수는 보드 내의 연결된 블록 위치를 의미하는 2차원 배열이다. 이 배열은 보석 블록의 종류 대신 연결된 블록에 대한 정보를 가지고 있다. 보드 내의 각 보석 블록은 checkChain() 함수에 의해 연결되었는지 여부를 확인하며, 검사된 지점에 해당하는 배열 원소에는 checkChain() 함수의 리턴 값이 저장된다.

연결된 보석 블록 삭제하기

그러나 연결된 블록들이 존재하는지 알아내는 것만으로는 충분치 않다. 이 게임은 이 정보를 이용하여 뭔가 다른 행동을 취해야 한다. 즉, 연결된 블록들을 삭제하고 그 위에 있던 블록들을 아래로 내려오도록 해야 한다. 연결된 블록들에 대한 정보는

예제 4.14의 check() 함수에 의해 처리된다.

예제 4.14 연결된 블록의 처리

```
jewel.board = (function() {
    ...
    function check() {
        var chains = getChains(),
            hasChains = false, score = 0,
            removed = [], moved = [], gaps = [];

        for (var x = 0; x < cols; x++) {
            gaps[x] = 0;
            for (var y = rows-1; y >= 0; y--) {
                if (chains[x][y] > 2) {
                    hasChains = true;
                    gaps[x]++;
                    removed.push({
                        x : x, y : y,
                        type : getJewel(x, y)
                    });
                } else if (gaps[x] > 0) {
                    moved.push({
                        toX : x, toY : y + gaps[x],
                        fromX : x, fromY : y,
                        type : getJewel(x, y)
                    });
                    jewels[x][y + gaps[x]] = getJewel(x, y);
                }
            }
        }
    }
    ...
})();
```

이 함수는 보드에서 보석 블록을 제거하고 필요한 곳에서 다른 블록을 가져온다. check() 함수는 게임 보드를 수정하는 동시에 삭제되거나 이동한 보석 블록에 대한 정보를 removed와 moved라는 두 배열에 저장한다. 이 데이터는 화면에 보석 블록의 이동을 애니메이션 처리하는 등 이후에 필요한 작업을 처리하기 위한 중요한 데이터이다.

check() 함수는 두 개의 중첩된 반복문을 통해 게임 보드 내의 모든 지점들을 탐색한다. 만일 현재 탐색 중인 지점의 연결 정보가 2보다 큰 값을 갖는다면 이 위치와 해당 위치의 보석 블록의 종류가 removed 배열에 저장된다. 나중에 다른 블록이 내려오게 되면 이 위치의 값을 덮어쓸 것이기 때문에 지금 당장 jewels 배열의 값을 수정할 필요는 없다.

안쪽의 반복문은 각 행을 반복하는데, 일반적으로 위에서 아래로 탐색하는 것이 아니라 아래에서 위쪽으로 탐색해 가는 것이 중요하다. 이렇게 하면 보석 블록이 제거된 후에 다른 보석을 즉시 내려보낼 수 있다. 또한 이 알고리즘은 gaps 배열에 각 열의 개수를 보관한다. 각 열을 처리하기 전에는 각 열에 대해 gaps 배열의 값을 0으로 설정한다. 그리고 보석 블록이 제거될 때마다 이 개수가 증가하게 된다. 만일 보석 블록을 제거할 필요가 없다면 이 숫자에 의해 해당 블록이 아래로 내려가야 하는지를 결정한다. 이 값이 양수라면 현재 보석 블록은 그 수만큼 아래로 이동해야 한다. 이 정보는 원래 위치와 최종 위치를 가지고 있는 moved 배열에 저장된다. 또한 jewels 배열의 정보를 곧바로 수정하여 현재 위치를 다시 탐색하지 않도록 해주어야 한다.

새 보석 블록 채우기

check() 함수는 아직 완벽한 것이 아니다. 기존에 존재하던 보석 블록을 아래로 이동하여 빈 공간을 채우고는 있지만, 보드 위쪽에 새롭게 나타난 빈 공간은 아직 채우지 않고 있기 때문이다. 따라서 현재 열의 모든 보석 블록을 탐색한 후에는 위쪽에 나타난 빈 공간을 새로운 보석 블록으로 채워주어야 한다. 이 작업을 위해 check() 함수를 예제 4.15와 같이 수정해보자.

예제 4.15 새 보석 블록 추가하기

```
jewel.board = (function() {
    ...
    function check() {
        ...
        for (var x = 0; x < cols; x++) {
            gaps[x] = 0;
            for (var y = rows-1; y >= 0; y--) {
                ...
            }
```

```
            // 위쪽을 채운다.
            for (y = 0; y < gaps[x]; y++) {
                jewels[x][y] = randomJewel();
                moved.push({
                    toX : x, toY : y,
                    fromX : x, fromY : y - gaps[x],
                    type : jewels[x][y]
                });
            }
        }
    }
    ...
})();
```

현재 열에 새로 추가될 보석 블록의 개수는 현재 열에서 발견된 빈 공간의 개수와 같다. 새로 채워지는 보석 블록은 보드의 위쪽에 채워지므로 채워질 공간을 계산하는 것은 어렵지 않다. 새로 채워진 보석 블록에 대한 정보 역시 이미 이동한 보석 블록에 대한 정보를 담고 있는 moved 배열에 채우면 된다. 새로 추가된 보석 블록은 기존 위치에 대한 정보가 없기 때문에 이 보석 블록은 이미 게임 보드의 영역 바깥에 존재했던 것으로 가정하고 처리하면 된다.

점수 계산하기

앞서 initialize() 함수에서는 사용자가 얻을 점수를 계산하기 위해 baseScore 변수를 선언했다. 이 값은 게임 보드 전체에 해당하는 총 점수를 계산하는 데 사용된다. 예제 4.16은 check() 함수에 점수를 계산하는 로직을 추가한 코드이다.

예제 4.16 점수를 계산하는 로직

```
jewel.board = (function() {
    ...
    function check() {
        ...
        for (var x = 0; x < cols; x++) {
            gaps[x] = 0;
            for (var y = rows-1; y >= 0; y--) {
                if (chains[x][y] > 2) {
                    hasChains = true;
```

```
                    gaps[x]++;
                    removed.push({
                        x : x, y : y,
                        type : getJewel(x, y)
                    });

                    // 점수 계산하기
                    score += baseScore
                           * Math.pow(2, (chains[x][y] - 3));
                } else if (gaps[x] > 0) {
                    ...
                }
                ...
            }
        }
    }
    ...
})();
```

연결된 블록에 포함되는 모든 보석 블록에 대한 점수는 score 변수에 저장되며, 이 점수는 연결된 블록의 길이에 따라 다르다. 만일 연속된 블록의 개수가 많다면 점수는 배가 된다.

Remember

score 변수는 플레이어의 총점을 의미하는 것이 아니라 check() 함수가 각 보석 블록을 처리하는 동안 발생하는 점수만을 저장한다. 현재의 게임 보드 모듈에는 아직 플레이어에 대한 개념이 들어있지 않기 때문에 이 로직은 단순히 보석 블록을 교환한 후에 얼마나 많은 점수를 얻었는지만을 계산한다.

이제 연결된 블록을 삭제하고 빈 공간을 새로운 블록으로 채우는 작업까지 완료되었다. 그러나 새로 만들어진 블록들이 다시 새로운 연결된 블록을 만들어 낼 가능성이 있기 때문에 아직 완벽하게 끝났다고 할 수는 없다. 연결된 블록이 남아있지 않은지를 판단하기 위해서 check() 함수는 더 이상 연결된 블록이 없다는 것이 확실해질 때까지 자기 자신을 재귀적으로 호출해야 한다. 또한 이 함수는 기록된 변경사항들을 리턴해야 한다. 변경된 check() 함수의 코드는 예제 4.17과 같다.

예제 4.17 재귀적으로 게임 보드 검사하기

```
jewel.board = (function() {
    ...
    function check(events) {
        ...

        events = events || [];

        if (hasChains) {
            events.push({
                type : "remove",
                data : removed
            }, {
                type : "score",
                data : score
            }, {
                type : "move",
                data : moved
            });
            return check(events);
        } else {
            return events;
        }
    }
    ...
}
```

예제에서와 같이 재귀 호출을 통해 removed, moved 및 score 데이터를 수집할 필요가 있다. 그러자면 check() 함수에 events라는 선택적 매개변수를 추가해야 한다. 이 매개변수는 재귀 호출에만 사용된다. 만일 이 매개변수에 아무런 값이 전달되지 않으면 events 매개변수를 빈 배열로 초기화하면 된다. 게임 보드에 대한 처리가 완료되고 누적된 점수를 추가하면 events 배열에 이벤트 객체 형식의 데이터를 추가한다. 각각의 이벤트 객체는 type과 data라는 속성만 가지고 있다. 더 이상 연결된 블록이 존재하지 않으면 게임 보드의 변경을 중단하고 check() 함수를 더 이상 호출하지 않아도 된다. 이 상태에서 events 배열을 리턴하면 이 객체는 최초에 check() 함수를 호출한 호출자에게 리턴된다. 이렇게 하면 호출자는 사용자가 마지막으로 블록을 이동한 시점부터 블록을 제거하거나 움직여 완성된 최종 상태 사이에 발생했던 모든 변경사항에 대한 정보를 얻을 수 있게 된다.

격자 다시 채우기

게임을 진행하다 보면 플레이어는 더 이상 움직일 수 있는 블록이 없는 상황에 맞닥뜨릴 수 있다. 이런 경우를 대비해서 게임 보드를 다시 그려줌으로써 플레이어가 게임을 계속할 수 있도록 해주어야 한다. 이를 위해 움직일 수 있는 블록이 존재하는지 여부를 판단하는 함수를 작성해야 한다. 예제 4.18은 hasMoves() 함수의 코드를 보여준다.

예제 4.18 움직일 수 있는 블록의 존재 여부를 검사하는 함수

```
jewel.board = (function() {
    ...
    // 움직일 수 있는 블록이 존재하면 true를 리턴한다.
    function hasMoves() {
        for (var x = 0; x < cols; x++) {
            for (var y = 0; y < rows; y++) {
                if (canJewelMove(x, y)) {
                    return true;
                }
            }
        }
        return false;
    }
    ...
})();
```

hasMoves() 함수는 블록을 움직여서 연결된 블록을 만들 수 있는 부분이 있다면 true를 리턴하고, 그렇지 않으면 false를 리턴한다. canJewelMove() 함수는 실제로 이동 가능한 위치를 파악하는 역할을 수행하며, 그 코드는 예제 4.19와 같다.

예제 4.19 블록을 이동할 수 있는지 검사하는 함수

```
jewel.board = (function() {
    ...
    // (x, y) 좌표가 유효한 위치이며 이웃한 블록과 교환이 가능하다면 true를 리턴한다.
    function canJewelMove(x, y) {
        return ((x > 0 && canSwap(x, y, x-1, y)) ||
                (x < cols-1 && canSwap(x, y, x+1, y)) ||
```

```
                (y > 0 && canSwap(x, y, x, y-1)) ||
                (y < rows-1 && canSwap(x, y, x, y+1)));
    }
    ...
})();
```

canJewelMove() 함수는 선택한 블록을 상하 혹은 좌우로 이동하여 새로 연결된 블록을 만들 수 있는지 여부를 판단하기 위해 canSwap() 함수를 호출한다. canSwap() 함수는 이웃 블록이 게임 보드의 영역 내에 위치할 때만 수행된다. 즉, canSwap() 함수는 현재 선택한 블록의 왼쪽에 위치한 이웃 블록과 교환할 때는 x 좌표가 최소 1보다 크거나 같으며 (cols−1) 값보다 작은 경우에만 수행하며, 나머지 경우도 마찬가지로 이웃 블록이 유효한 영역 내에 있는지를 검사한다.

만일 플레이어가 더 이상 움직일 블록이 없다면 hasMoves() 함수는 false를 리턴하며 게임 보드는 자동적으로 다시 그려진다. 이 다시 그리기 기능은 check() 함수에 의해 호출된다. 게임 보드에서 연결된 블록들을 검사하고 이들을 제거한 후에는 새로운 블록들이 게임 보드에 생겨나고, 그런 후 hasMoves() 함수를 호출하여 새로운 게임 보드에서 움직일 수 있는 블록이 있는지를 검사하며, 이때 필요하다면 게임 보드를 다시 그려야 한다. 예제 4.20은 이러한 로직을 적용하기 위해 수정된 코드를 보여준다.

예제 4.20 보드 다시 그리는 기능을 추가한 코드

```
jewel.board = (function()
    ...
    function check(events) {
        ...

        if (hasChains) {
            ...

            // 더 이상 움직일 수 있는 블록이 없으면 보드를 다시 그린다.
            if (!hasMoves()) {
                fillBoard();
                events.push({
                    type : "refill",
                    data : getBoard()
                });
```

```
            }
            return check(events);
        } else {
            return events;
        }
    }
})();
```

예제에서는 게임 보드를 다시 그리기 위한 fillBoard() 함수를 호출하는 것 외에도 refill이라는 타입의 이벤트를 events 배열에 추가한다. 이 이벤트는 예제 4.22의 getBoard() 함수에 의해 생성된 현재 게임 보드의 복사본을 데이터로 가지게 된다.

예제 4.21 게임 보드 데이터의 복사본 생성하기

```
jewel.board = (function() {
    ...

    // 게임 보드의 복사본 생성하기
    function getBoard() {
        var copy = [],
            x;
        for (x = 0; x < cols; x++) {
            copy[x] = jewels[x].slice(0);
        }
        return copy;
    }

    return {
        ...
        getBoard : getBoard
    };

})();
```

단순히 fillBoard() 함수를 호출하는 것만으로는 새로 구성된 게임 보드에서 이동할 수 있는 블록이 존재한다는 것을 보장할 수는 없다. 아주 작지만 새로 구성된 게임 보드에서도 더 이상 이동할 수 있는 블록이 없을 경우가 있다. 이런 경우에는 플레이어가 알지 못하도록 다시 한 번 게임 보드를 새로 그려야 한다. 이런 경우를 검사하기

가장 좋은 방법은 바로 fillBoard() 함수 내부에서 처리하는 방법이다. hasMoves() 함수를 호출하면 현재 게임 보드에서 이동할 수 있는 블록이 존재하는지 알 수 있다. 따라서 이동할 수 있는 블록이 없다면 fillBoard() 함수를 다시 한 번 호출하면 된다. 이렇게 하면 이동 가능한 블록이 적어도 하나 이상 존재할 때까지 계속해서 보드를 새로 그릴 수 있다. 예제 4.22는 이런 기능이 추가된 fillBoard() 함수의 코드이다.

예제 4.22 보드를 계속해서 새로 그리는 코드

```
jewel.board = (function() {
    ...
    function fillBoard() {
        ...

        // 이동 가능한 블록이 존재할 때까지 보드를 계속 다시 그린다.
        if (!hasMoves()) {
            fillBoard();
        }
    }
    ...
}) ();
```

이렇게 구현된 다시 그리기 로직을 통해 이동 가능한 블록이 존재하지 않는 게임 보드가 만들어지는 경우를 대비할 수 있다. 게임 보드를 처음 생성할 때 이동 가능한 블록이 존재할 수 있는 것처럼 당연히 블록을 전혀 이동할 수 없을 가능성도 있다. 따라서 예제와 같은 재귀 호출을 통해 이 문제를 해결할 수 있다.

블록 교환하기

이제 게임 보드를 관리하는 모든 기능을 구현하였다. 단 한 가지 아직 구현하지 않은 기능은 실제로 보석 블록을 교환하는 기능이다. 이 작업은 비교적 직관적이다. 우리는 이미 canSwap() 함수를 통해 플레이어가 선택한 두 블록을 교환할 수 있는지 검사하고 있으며, check() 함수는 블록의 이동으로 인해 발생하는 변경사항을 모두 처리한다. 예제 4.23은 보석 블록을 서로 교환하는 swap() 함수를 구현한 코드이다.

예제 4.23 블록을 이동하는 함수

```
jewel.board = (function() {
    ...
    // 블록의 교환이 가능하다면 (x1, y1)의 블록과 (x2, y2)의 블록을 교환하고
    // 콜백 함수를 호출한다.
    function swap(x1, y1, x2, y2, callback) {
        var tmp,
            events;

        if (canSwap(x1, y1, x2, y2)) {

            // 두 블록을 교환한다.
            tmp = getJewel(x1, y1);
            jewels[x1][y1] = getJewel(x2, y2);
            jewels[x2][y2] = tmp;

            // 게임 보드를 검사하여 변경된 이벤트 목록을 가져온다.
            events = check();

            callback(events);
        } else {
            callback(events)
        }
    }

    return {
        ...
        swap : swap
    };
})();
```

swap() 함수는 게임의 나머지 코드에서 호출할 수 있어야 하며, 잠재적으로 게임 보드를 수정할 수 있기 때문에 initialize() 함수와 마찬가지로 비동기 방식으로 동작해야 한다. swap() 함수는 서로 교환할 두 블록의 좌표를 비롯하여 콜백 함수를 지정하기 위한 5번째 매개변수를 정의하고 있다. 블록의 교환의 성공 여부에 따라 이 콜백 함수는 블록 교환이 일어난 후에 발생한 변경사항의 목록을 전달받거나 false를 전달받을 수 있다. 예제 4.24는 board 모듈이 노출해야 하는 함수들을 보여준다.

예제 4.24 board 모듈의 공용 메서드들

```
jewel.board = (function() {
    ...

    return {
        initialize : initialize,
        swap : swap,
        canSwap : canSwap,
        getBoard : getBoard,
        print : print
   };

})();
```

외부에 노출되는 함수를 예제와 같이 제한함으로써 게임의 상태를 변경할 수 있는 유일한 방법은 게임 보드를 새로 그리거나 swap() 함수를 호출하여 블록을 교환하는 방법밖에 없게 되었다. 게임의 모든 규칙은 swap() 함수에 의해 적용되므로 게임 보드의 무결성을 보장할 수 있게 되었다. 또한 보석 블록들의 데이터에 접근하는 유일한 진입점은 getBoard() 함수이지만, 이 함수는 보석 블록 데이터에 대한 수정 기능을 제공하지 않으므로 게임의 나머지 코드들이 예기치 않게 게임 보드를 망가뜨리는 위험을 최소화할 수 있다.

swap() 함수를 테스트하려면 자바스크립트 콘솔에서 직접 호출해보면 된다. 먼저 아래의 코드를 이용하여 게임 보드를 초기화한다.

```
jewel.board.initialize(functino() {})
```

그런 후 jewel.board.print() 함수를 이용하여 서로 교환할 블록의 좌표를 확인한 후 콘솔에 아래와 같은 구문을 입력해보면 된다.

```
jewel.board.swap(4,3,4,2, function(e) { console.log(e) })
```

swap() 함수는 콜백 함수를 필요로 하기 때문에 예제에서는 변경사항이 기록된 배열을 콘솔에 출력하는 함수를 정의하였다.

요약

이번 장에서는 게임의 핵심 메커니즘을 구현했다. 이번 장에서 구현한 코드들은 보석 블록의 교환과 연결된 블록의 처리 및 새로운 블록을 채우는 등 게임의 기본 규칙을 구현하는 데 중점을 두었다. 게임 보드는 이제 독립된 모듈에 깔끔하게 캡슐화되었으며, 제한된 방법을 통해서만 게임의 데이터에 액세스하도록 구현되었기 때문에 반드시 게임의 규칙에 부합하는 변경만 발생할 수 있게 되었다.

또한 게임이 단일 플레이어를 위한 단독 게임 로직과 서버 기반의 다중 플레이어 로직을 지원할 수 있도록 모듈을 다듬어 향후 다중 플레이어 기능을 지원하기 위한 준비를 갖추었다. 몇 가지 핵심 메서드에 콜백 함수를 사용함으로써 두 개의 모듈이 동일한 인터페이스를 공유하게 되었으며, 향후 이를 바탕으로 비동기식 서버 측 보드 모듈을 보다 손쉽게 추가할 수 있다.