Halloween Puzzle

Einleitung

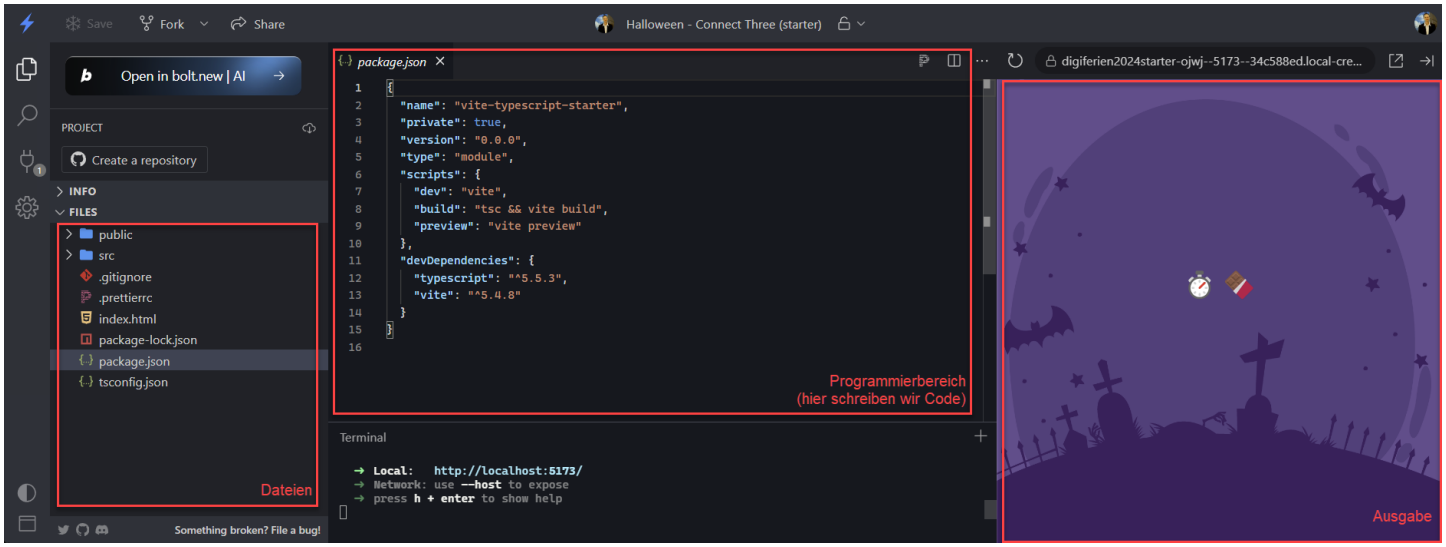
Willkommen beim Halloween Puzzle! Diese Übung eignet sich gut für alle, die zwar schon einiges mit Scratch gemacht, jedoch noch wenig oder keine Erfahrung mit textueller Programmierung haben. Wir wollten gemeinsam ein *Halloween Puzzle* bauen, bei dem man Punkte sammeln kann, indem man mindestens drei gleiche Symbole zusammenbringt. Als Programmiersprache werden wir *TypeScript* verwenden.

Entwicklungsumgebung laden

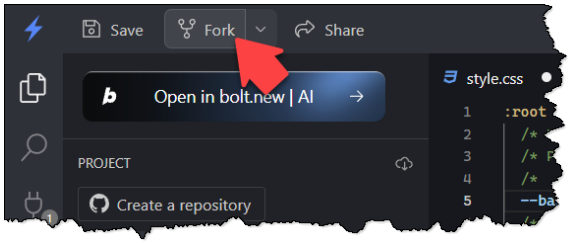
Als erstes müssen wir die *Entwicklungsumgebung* laden. Dafür brauchst du eine moderne Version von dem Internetbrowser *Chrome* oder *Edge*. Starte den Browser und öffne die folgende Webseite:

https://stackblitz.com/edit/digiferien-2024-starter

Wenn alles klappt, dann solltest du unsere Entwicklungsumgebung *Stackblitz* sehen. Sie sieht in etwa so aus:

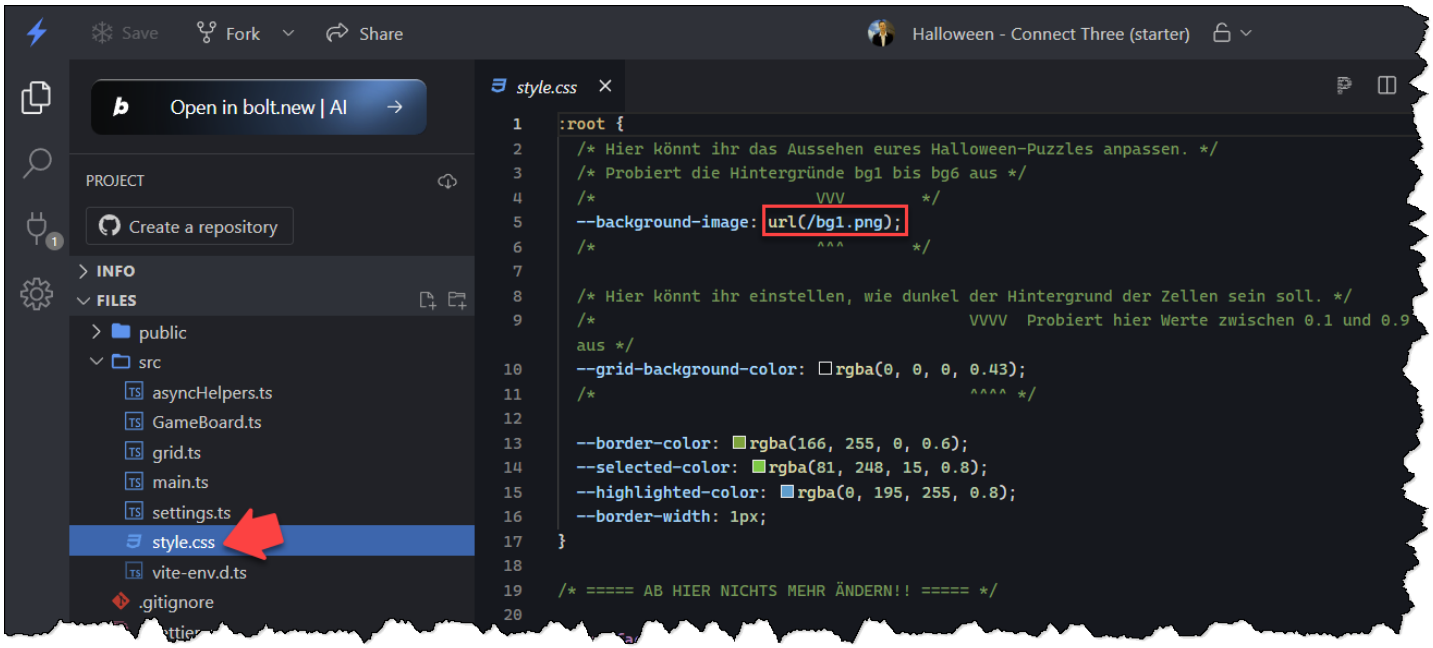


Du musst einmal auf *Fork* klicken, damit du deine eigene Kopie von dem Spiel bekommst. **Lade zur Sicherheit danach den Inhalt des Internetbrowsers einmal neu.**



Hintergrund ändern

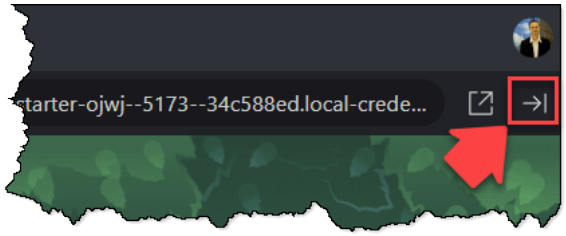
Fangen wir mit ersten, einfachen Programmierschritten an. Deine Aufgabe ist es, den Hintergrund für unser Spiel zu wählen. Öffne dafür die Datei *style.css* im Ordner *src*, indem du sie anklickst. Ganz oben findest du einen Hinweis, was du ändern musst, um verschiedene Hintergründe auszuwählen. Die folgende Grafik zeigt auch, wo du etwas ändern musst. Probiere die Hintergründe *bg1.png* bis *bg6.png* aus. Wähle den Hintergrund, der dir am besten gefällt.

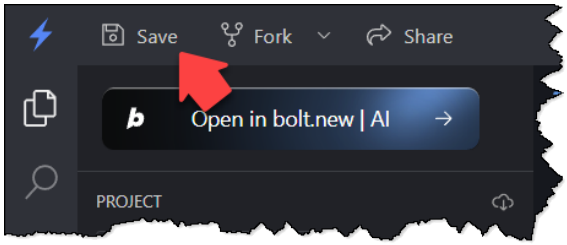


Wenn du genau schaust, dann siehst du, dass in der *style.css*-Datei noch weitere Einstellungen, die du ändern kannst. Lass sie im Moment noch so wie sie sind. Wir kommen später auf sie zurück.

Tipp: Wenn du etwas im Code änderst, aktualisiert sich die Ausgabe automatisch. Dafür ist kein explizites Speichern notwendig. **Trotzdem ist es empfehlenswert, öfter auf *Save* zu klicken.** Wenn nämlich dein Programm abstürzt und du den Browser neu landen musst, wird der Code auf den letzten Stand zurückgesetzt, den du gespeichert hast.

Tipp: Wenn du etwas im Code änderst, aktualisiert sich die Ausgabe automatisch. Dafür ist kein explizites Speichern notwendig. Falls das automatische Aktualisieren einmal nicht klappt, blende die Ausgabe einmal aus- und dann wieder ein (siehe Bild rechts). Spätestens dann solltest du die neue Ausgabe sehen.



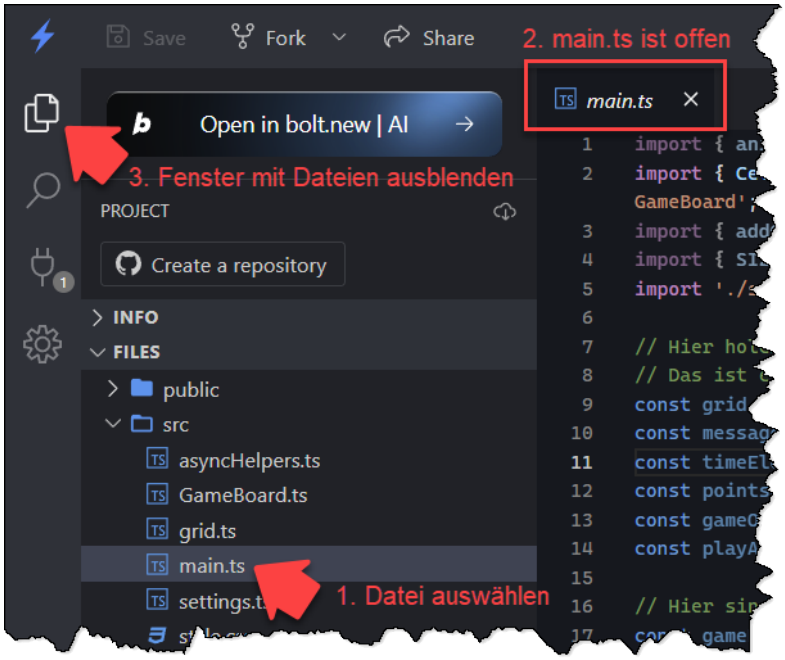


Trotzdem ist es empfehlenswert, öfter auf *Save* zu klicken (siehe Bild links). Wenn nämlich dein Programm abstürzt und du den Browser neu landen musst, wird der Code auf den letzten Stand zurückgesetzt, den du gespeichert hast.

Raster anzeigen

Jetzt legen wir mit TypeScript los. Ein Spiel wie das Halloween Puzzle von Grund auf zu programmieren ist nicht einfach. Das würde zu lange dauern. Daher haben wir einiges für dich vorbereitet. Du musst den Code in der Datei *main.ts* vervollständigen. Diese Anleitung führt dich durch alle notwendigen Schritte.

Öffne die Datei *main.ts* über das Dateifenster (siehe Bild rechts). Falls du einen relativ kleinen Bildschirm hast, kannst du anschließend das Dateifenster ausblenden, da wir uns im Moment ganz auf *main.ts* konzentrieren.



Suche die Funktion *buildGrid()* in deinem Programm. Die Zeile die du suchst, enthält *function buildGrid() {*. Hier ist der Code, den du **innerhalb der geschweiften Klammern** von *buildGrid()* eintippen musst.

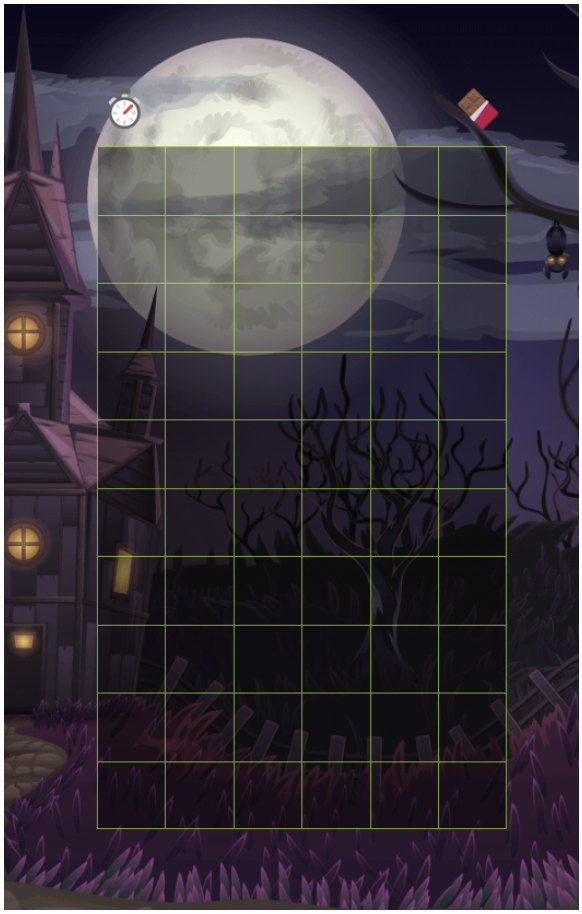
Tipp: Die Zeilen, die mit *//* beginnen, sind *Kommentare*. Man verwendet sie, um sich Notizen zu machen. Du musst die Kommentarzeilen **nicht** abschreiben. Sie sollen dir nur erklären, was der Code macht. Wenn du etwas nicht verstehst, frag erfahrene Programmiererinnen oder Programmierer im CoderDojo oder in der Schule.

```
function buildGrid() {
  // Der "for"-Befehl ist eine sogenannte Schleife. Vielleicht kennst du Schleifen
  // aus Scratch. Dort heißen sie "Wiederhole fortlaufend", "Wiederhole bis",
  // "Wiederhole ___ mal", etc.

  // Diese Schleife wird je Zeile wiederholt. "SIZE_Y" ist eine sogenannte "Konstante".
  // Sie legt fest, wie viele Zeilen wir haben (in unserem Fall 10).
  for (let row = 0; row < SIZE_Y; row++) {
    // Diese Schleife wird je Spalte wiederholt. "SIZE_X" ist wie oben eine Konstante,
    // die die Anzahl der Spalten festlegt (in unserem Fall 6).
    for (let col = 0; col < SIZE_X; col++) {
      // Diese Funktion fügt eine Zelle zu unserem Raster hinzu. Dabei übergeben
      // wir unter andere die Spalte ("col" Abkürzung für "column") und die Zeile ("row").
      addCell(grid, col, row);
    }
  }
}
```

Geschafft? Super! Jetzt probieren wir das aus. Suche dir die Funktion *startup()* heraus. Sie ist direkt über der Funktion *buildGrid*, die du gerade geändert hast. Schreib in *startup* die folgenden Befehle:

```
async function startup() {
  // Der Code in dieser Funktion wird am Beginn des Programms ausgeführt.

  // Als erstes müssen wir das Styling initialisieren. Das ist ein technisches Detail,
  // auf das wir hier nicht näher eingehen werden.
  initializeStyling();

  // Jetzt erstellen wir das Raster
  buildGrid();
}
```

Und? Siehst du das Raster im Ausgabebereich, wo unser Puzzle erscheinen wird (siehe Bild rechts)? Gratulation! Du hast die erste, große Hürde geschafft.



Falls du das Raster nicht siehst, kontrolliere nochmal genau, ob du den Code so geschrieben hast, wie er in dieser Anleitung enthalten ist. Jedes Zeichen ist wichtig, auch Leerzeichen, Groß- und Kleinschreibung, Klammern - beim Programmieren muss man ganz genau darauf achten, was man eintippt. Computer sind da heikel.

Raster mit Symbolen füllen

Bereit für das nächste Level? Jetzt füllen wir unser Raster mit Symbolen. Natürlich verwenden wir Symbole, die zu Halloween passen. Suche dir die Funktion *fill()*. Füge den folgenden Code zur Funktion *fill* hinzu:

```
function fill() {
  // In dieser Methode füllen wir unser Raster mit Symbolen (z.B. Geister, Kürbis,
     Süßigkeiten, etc.).

  // Später müssen wir wissen, ob wir überhaupt etwas hinzugefügt haben. Deshalb
  // merken wir uns in der Variable "addedSomething", ob wir etwas hinzugefügt haben.
  let addedSomething = false;

  // Kannst du dich noch an die "for"-Schleife aus der buildGrid-Methode erinnern?
  // Hier funktionieren die Schleifen genau gleich.
  for (let row = 0; row < SIZE_Y; row++) {
    for (let col = 0; col < SIZE_X; col++) {
      // Wir überprüfen, ob die Zelle leer ist.
      if (gameBoard.isEmpty(row, col)) {
        // Wenn ja, fügen wir ein zufälliges Symbol hinzu.
        gameBoard.addRandomSymbol(col, row, select);

        // Wie erwähnt merken wir uns in der Variable "addedSomething", dass wir etwas
           hinzugefügt haben.
        addedSomething = true;
      }
    }
  }

  // Wir geben zurück, ob wir etwas hinzugefügt haben.
  return addedSomething;
}
```

So wie vorhin müssen wir am Beginn unseres Programms in *startup* die neue Funktion *fill* aufrufen. Suche dir die Funktion *startup* heraus und füge **am Ende** den Aufruf von *fill* hinzu:

```
async function startup() {
  ...

  // Wir füllen das Raster mit Symbolen
  fill(); // <<<< DAS MUSST DU AM ENDE VON startup EINFÜGEN
}
```

Und? Siehst du in der Ausgabe bereits die Symbole im Raster? Dann hast du alles richtig gemacht. Falls nicht, kontrolliere nochmal aufmerksam den Code.

Markieren von Symbolen

Bei unserem Spiel muss man Gruppen von mindestens drei gleichen Symbolen zusammenstellen, indem man benachbarte Symbole vertauscht. Der Benutzer wählt dir zu vertauschenden Systeme durch Mausklick. Im ersten Schritt wollen wir das Auswählen eines Symbols ermöglichen. Wenn der Benutzer ein zweites Mal auf das Symbol klickt, wir die Auswahl entfernt. Das eigentliche Vertauschen machen wir etwas später.

Such dir die Funktion *select()*. Füge den folgenden Code hinzu. **Nicht vergessen:** Kommentare brauchst du nicht abzutippen!

```
export async function select(selectedSymbol: HTMLDivElement, col: number, row: number) {
  if (message.innerText || gameOver) {
    return;
  }

  // Der Benutzer hat ein Symbol angeklickt. Es gibt drei Fälle:
  // 1. Der Benutzer hat das erste Mal ein Symbol angeklickt
  //    -> wir müssen es markieren und uns die Auswahl merken
  // 2. Der Benutzer hat das gleiche Symbol ein zweites Mal angeklickt
  //    -> wir müssen die Markierung entfernen
  // 3. Der Benutzer hat zwei unterschiedliche Symbole angeklickt
  //    -> wir müssen prüfen, ob sie nebeneinander liegen und - falls ja - dann tauschen

  if (!firstSelection) {
    // Der Benutzer hat das erste Mal ein Symbol angeklickt.
    // Markieren wir es und merken wir uns die Auswahl.
    gameBoard.markAsSelected(selectedSymbol);
    firstSelection = selectedSymbol;
    firstCol = col;
    firstRow = row;
  } else if (firstSelection === selectedSymbol) {
    // Der Benutzer hat das gleich Symbol ein zweites Mal angeklickt.
    // In diesem Fall entfernen wir die Selektion.
    gameBoard.removeSelectionMark(firstSelection);
    firstSelection = null;
  } else {
    // Der Benutzer hat ein zweites Symbol angeklickt.

    // <<<< HIER FÜGEN WIR GLEICH ALS NÄCHSTES CODE HINZU.
    //      Im Moment bleibt das noch leer
  }
}
```

Probiere aus, auf ein Symbol im Raster zu klicken. Wird es farbig hervorgehoben? Wenn ja, dann hast du alles richtig gemacht. Wenn du nochmal auf das selbe Symbol klickst, wird die Auswahl entfernt.

Tauschen von Symbolen

Jetzt machen wir die Funktion *select* fertig. Im vorigen Kapitel ist ein *else*-Zweig unvollständig geblieben. Wenn du nicht mehr sicher bist, wo das war, blättere eine Seite zurück und suche den Kommentar *HIER FÜGEN WIR GLEICH ALS NÄCHSTES CODE HINZU*. Genau dort programmieren wir jetzt weiter. Innerhalb der geschweiften Klammern des *else*-Zweigs schreiben wir den folgenden Code:

```
export async function select(selectedSymbol: HTMLDivElement, col: number, row: number) {
  ... // DEN CODE DAVOR HABEN WIR SCHON!
  } else {
    // Der Benutzer hat ein zweites Symbol angeklickt.
    const secondSelection = selectedSymbol;
    const secondCol = col;
    const secondRow = row;

    // Prüfen wir, ob die Symbole nebeneinander liegen.
    // Nur wenn die Symbole Nachbarn sind, ist die Auswahl des Benutzers gültig.
    if (isNeighbour(firstRow, secondRow, firstCol, secondCol)) {
      // Wenn ja, markieren wir das zweite Symbol
      gameBoard.markAsSelected(secondSelection);

      points--;
      updateStatus();

      // Tauschen wir die Symbole miteinander
      gameBoard.switchSymbols(firstSelection, secondSelection);

      // Wir müssen herausfinden, in welche Richtung die Symbole bewegt werden müssen.
      const firstDir = gameBoard.getDirection(firstRow, firstCol, secondRow, secondCol);
      const secondDir = gameBoard.getDirection(secondRow, secondCol, firstRow, firstCol);

      // Jetzt können wir die Symbole bewegen.
      gameBoard.moveAnimation(firstSelection, firstDir);
      gameBoard.moveAnimation(secondSelection, secondDir);

      message.innerText = ''; // Hier können wir später eine Meldung eingeben
      await animation();
      firstSelection.classList.remove('selected');
      secondSelection.classList.remove('selected');
      firstSelection = null;
      await cleanup();
    }
  }
}
```

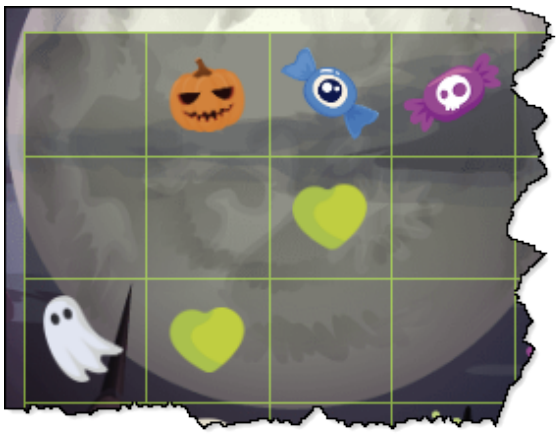
Probiere aus, ob du zwei benachbarte Symbole austauschen kannst. Klappt es? Wow, du hast schon eine Menge geschafft heute! **Vergiss nicht, regelmäßig Pausen zu machen.** Jetzt wäre eine gute Gelegenheit dafür.

Entfernen von Gruppen

Der nächste Schritt ist das Erkennen und Entfernen von Gruppen von mindestens drei gleichen Symbolen. Wir bezeichnen solche Gruppen hier als *Cluster*. Das Erkennen von Clustern machen wir in der Funktion *cleanup()*. Suche sie in deinem Code heraus und füge den folgenden Code ein:

```
async function cleanup() {
  // Alle Gruppen mit 3 gleichen, benachbarten Symbolen finden
  const clusters = gameBoard.findClusters();
  if (clusters.length > 0) {
    // Wir haben Gruppen gefunden. Lassen wir die Symbole der
    // Gruppen verschwinden.
    for (const cluster of clusters) {
      for (const item of cluster) {
        gameBoard.letDisappear(item.symbol);
      }
    }

    // Die Symbole verschwinden nicht sofort, sondern werden
    // langsam ausgeblendet. Daher müssen wir einen Moment warten,
    // bis die Animation fertig ist.
    await animation();

    for (const cluster of clusters) {
      // Punkteberechnung
      points += cluster.length * cluster.length - 6;
      updateStatus();

      // Gruppen löschen
      for (const item of cluster) {
        gameBoard.removeSymbol(item.symbol);
      }
    }

    fallDown();
  } else {
    message.innerText = '';
  }
}
```

Verschiebe Symbole, sodass Cluster entstehen. Verschwinden die Symbole dann? Rechts siehst du ein Bild, bei dem bereits Symbole eines Clusters verschwunden sind. So sollte die Ausgabe deines Programms jetzt aussehen.



Fällt dir auf, dass in unserem Spiel Cluster, die gleich am Anfang da sind, nicht gleich entfernt werden? Das können wir beheben, indem wir in unsere *startup*-Funktion **ganz am Ende** etwas Code hinzufügen:

```
async function startup() {
  ...
  fill(); // <<<< DIESE ZEILE HAST DU SCHON. Den Code ab hier einfügen!

  // Punkte und Zeit anzeigen
  updateStatus();

  // Die Symbole sind nicht plötzlich da, sie werden langsam eingeblendet.
  // Daher müssen wir an dieser Stelle warten, bis die Animation abgeschlossen ist
  // und alle Symbole eingeblendet sind.
  await animation();

  // Es kann sein, dass gleich am Beginn Cluster mit mehr als 3 Symbolen entstanden.
  // Diese müssen wir entfernen und die Symbole fallen lassen.
  message.innerText = ''; // Hier können wir später eine Meldung einfügen
  cleanup();
}
```

Symbole fallen lassen

Du kommst super voran! Jetzt müssen wir die Symbole nach unten fallen lassen, wenn Cluster verschwinden. Die Lücken werden dann mit neuen, zufälligen Symbolen aufgefüllt. Das machen wir in der Funktion *fallDown*. Suche dir die Funktion heraus und füge den folgenden Code ein:

```
async function fallDown() {
  // Hier lassen wir die Symbole nach unten fallen.

  // Wir merken uns, ob wir überhaupt etwas bewegt haben.
  let somethingMoved = false;

  // Unser alter Bekannter, die "for"-Schleife...
  for (let row = 1; row < SIZE_Y; row++) {
    for (let col = 0; col < SIZE_X; col++) {
      // Prüfen, ob das Symbol nach unten fallen kann (d.h. die Zelle darunter ist leer)
      if (gameBoard.canMoveDown(row - 1, col)) {
        // Wenn ja, lassen wir es fallen.
        gameBoard.fallDown(row - 1, col);

        // Und merken uns, dass wir etwas bewegt haben.
        somethingMoved = true;
      }
    }
  }

  if (somethingMoved) {
    // Falls wir etwas bewegt haben, warten wir, bis die Animation abgeschlossen ist
    await animation();

    // Durch das nach unten fallen könnten wieder neue Lücken enstanden sein, in die
    // Symbole fallen können. Daher rufen wir die Methode erneut auf.
    fallDown();
  } else {
    // Falls wir nichts bewegt haben, füllen wir die enstandenen, leeren Zellen neu auf.
    if (fill()) {
      // Warten, bis die Symbole eingeblendet sind
      await animation();

      // Vielleicht hat der Benutzer Glück und es gibt wieder Cluster mit mehr als 3
         Symbolen.
      cleanup();
    }
  }
}
```

Los gehts! Verschiebe Symbole, sodass Cluster entstehen. Diesmal dürfen die Symbole nicht nur verschwinden, die Lücken müssen sich automatisch wieder schließen. Wow, dein Spiel ist schon fast fertig!

Zeit und Punkte

Was fehlt noch? Damit es ein richtiges Spiel wird, brauchen wir ein Ziel. In unserem Fall bekommt man Punkte für Cluster. Für jedes Verschieben von Symbolen werden Punkte abgezogen. Die Logik dafür hast du schon programmiert. Wir müssen nur noch die verbleibende Zeit anzeigen. Das machen wir in der Funktion *updateStatus*. Suche sie dir heraus und füge den folgenden Code ein:

```
function updateStatus() {
  // Hier aktualisieren wir die Anzeige der verbleibenden Zeit und der Punkte.

  // Die nächste Zeile ist eine echte Herausforderung. Sie enthält viiiele Sonderzeichen.
  // Wir verwenden sie, um die verbleibende Zeit anzuzeigen.
  timeElement.innerText = `${Math.floor(seconds / 60)}:${seconds % 60 < 10 ? '0' : ''}${
      seconds % 60}`;

  // Und hier zeigen wir die Punkte an.
  pointsElement.innerText = points.toString();
}
```

Am Beginn des Programms in der *startup*-Funktion müssen wir jetzt noch ganz zum Schluss die Zeit ”ticken”lassen. Dafür füge folgenden Code **ganz am Anfang von *startup*** ein:

```
async function startup() {
  // Der Code in dieser Funktion wird am Beginn des Programms ausgeführt.

  // Das ist unser Timer. Er zählt die verbleibende Zeit herunter.
  const timer = setInterval(() => {
    seconds--;
    updateStatus();
    if (seconds <= 0) {
      // Zeit ist abgelaufen, wie zeigen "Game Over" an
      gameOver = true;
      gameOverDialog.showModal();
      clearInterval(timer);
    }
  }, 1000);

  // Am Spielende wird der Button "Nochmal spielen" angezeigt.
  playAgainButton.addEventListener('click', () => {
    // Lade die Seite neu, um das Spiel neu zu starten.
    location.reload();
  });

  // ... AB HIER FOLGT DER CODE, DER SCHON IN startup WAR
}
```

Fertig!

Du kannst stolz auf dich sein. Das war kein einfaches Spiel, das du hier programmiert hast!