# HTML5

## TWORZENIE GIER Z WYKORZYSTANIEM CSS I JAVASCRIPT

KARL BUNYAN


Helion

Tytuł oryginału: Build an HTML5 Game: A Developer's Guide with CSS and JavaScript

Tłumaczenie: Jakub Hubisz

ISBN: 978-83-283-1770-3



Wydawnictwo HELION
ul. Kościuszki 1c, 44-100 GLIWICE
tel. 32 231 22 19, 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *http://helion.pl* (księgarnia internetowa, katalog książek)

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
*http://helion.pl/user/opinie/htcsjs*
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:
ftp://ftp.helion.pl/przyklady/htcsjs.zip

Printed in Poland.

# Spis treści

# Część II. Ulepszanie gry za pomocą HTML5 i kanwy

# 3

# Logika gry

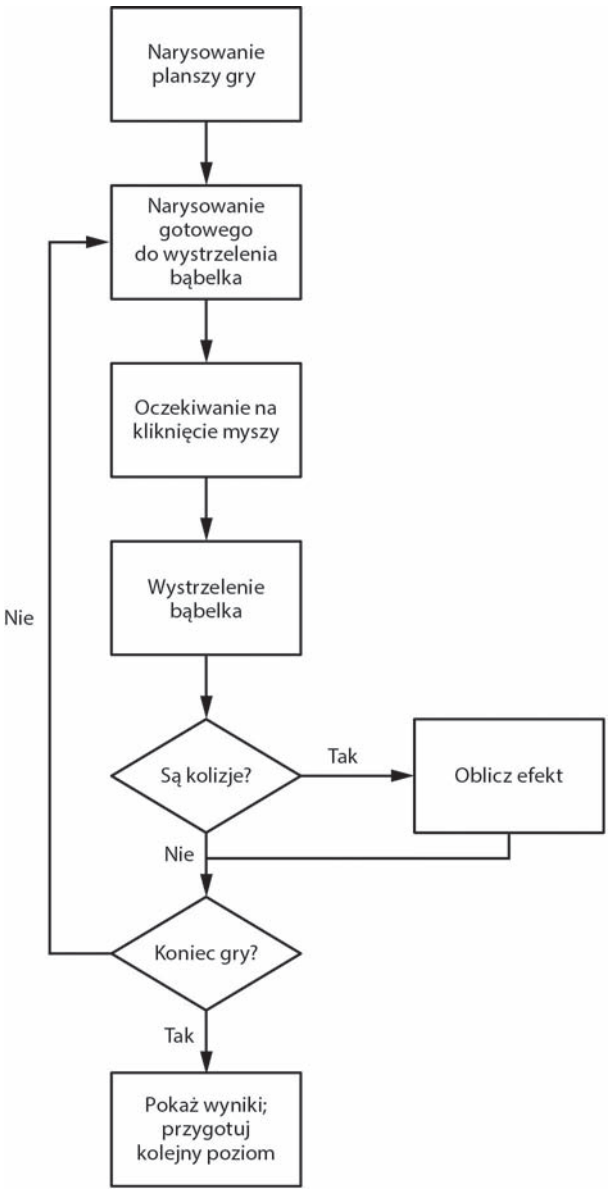
Do tej pory stworzyliśmy ekran powitalny, przycisk rozpoczynający nową grę i pojedynczy bąbelek, który może zostać wystrzelony poza ekran. W tym rozdziale *Bubble Shooter* zacznie bardziej przypominać grę. Dowiesz się, jak narysować planszę gry i wyświetlić informacje na temat poziomu, a następnie nauczysz się, jak wykrywać kolizje.

**Kolizje** są głównym elementem wielu gier i mają miejsce, jeżeli sprite'y się zetkną. Kiedy będziesz już w stanie wykrywać kolizje, będziesz mógł napisać kod, który sprawi, że sprite'y będą na nie reagować. W naszej grze kolizje następują, gdy wystrzelony bąbelek zderza się z bąbelkiem znajdującym się w siatce gry. Zaimplementujemy dwie reakcje: jeżeli bąbelek nie uformuje grupy co najmniej trzech bąbelków tego samego koloru, przyklei się do siatki, a w przeciwnym wypadku stworzy grupę i wszystkie bąbelki pękną.

Zanim jednak zaczniemy obliczać kolizje, potrzebujemy obiektu, z którym bąbelki będą mogły się zderzyć. W pierwszej części tego rozdziału omówię rysowanie początkowej planszy gry i przygotowanie stanu gry. Zrobimy to zgodnie z procesem obejmującym kilka kroków, pokazanym na rysunku 3.1.

Najpierw narysujemy planszę gry, a potem dodamy wykrywanie kolizji dla wystrzelonego bąbelka. W kolejnym rozdziale zaimplementujemy mechanizm pękania grup bąbelków oparty na identyczności kolorów.

Przejdźmy przez kolejne kroki i zamieńmy je na kod.




*Rysunek 3.1. Pętla gry rozpoczyna się od narysowania planszy, a kończy na pokazaniu wyniku*

# Rysowanie planszy gry

Plansza gry ma podobną strukturę dla każdego poziomu, a każda plansza zawiera rzędy bąbelków w czterech kolorach. Kolejne rzędy zawierają albo parzystą, albo nieparzystą liczbę bąbelków, zależnie od tego, czy numer rzędu jest parzysty, czy nie. Wszystkie te informacje będziemy przechowywać w obiekcie `Board`, a aktualny obiekt będzie przechowywany jako zmienna w obiekcie `Game`.

Wybrana struktura obiektów powinna być uzależniona od projektu gry, ale cele powinny być takie same jak podczas podejmowania decyzji na temat struktury kodu w aplikacjach webowych: należy grupować obiekty wykonujące podobne operacje i próbować osiągnąć równowagę w ilości abstrahowanych wspólnych funkcji. Nie definiuj kilku klas zawierających bardzo mało kodu, ale nie twórz też zbyt mało klas zawierających bardzo dużo kodu, które będą trudne do czytania i zrozumienia. Twórcy gier często opierają decyzje dotyczące struktury kodu na instynkcie i doświadczeniu oraz na ściśle zdefiniowanych zasadach. Zawsze bądź przygotowany, aby refaktoryzować swój kod, jeżeli uważasz, że Twoje początkowe wybory nie są już odpowiednie.

Rzędy składające się na planszę będą tablicą obiektów `Bubble`. Stworzymy tę tablicę podczas tworzenia instancji obiektu `Board`. Później przeniesiemy rysowanie elementów planszy do pliku *ui.js*. Wstawienie dużej ilości kodu do klasy `Game` jest bardzo proste, ale niepożądane — dlatego korzystaj z okazji i rozdzielaj odpowiedzialność pomiędzy klasami, kiedy tylko będzie to możliwe, szczególnie jeżeli chodzi o renderowanie obiektów na ekranie.

W pliku *game.js* musimy stworzyć zmienną przechowującą planszę i nową instancję obiektu `Board`. Plansza jest generowana po kliknięciu przycisku nowej gry. Dodaj poniższy kod do pliku *game.js*:

*game.js*

```
var BubbleShoot = window.BubbleShoot || {};
BubbleShoot.Game = (function($){
  var Game = function(){
    var curBubble;
    var board;
    --cięcie--
    var startGame = function(){
      $(".but_start_game").unbind("click");
      BubbleShoot.ui.hideDialog();
      curBubble = getNextBubble();
      board = new BubbleShoot.Board();
      BubbleShoot.ui.drawBoard(board);
      $("#game").bind("click",clickGameScreen);
    };
    --cięcie--
  };
  return Game;
})(jQuery);
```

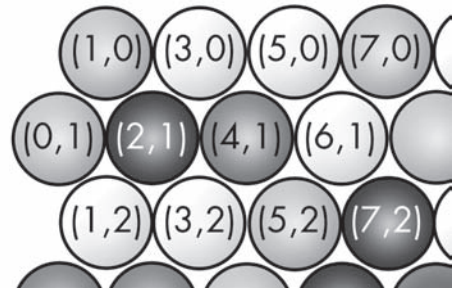
Board to nowy konstruktor, który musimy stworzyć. Stwórz nowy plik o nazwie *board.js* i dodaj go do listy plików ładowanych przez `Modernizr.load` w pliku *index.html*. Dodaj poniższy kod do nowego pliku:

*board.js*

```
var BubbleShoot = window.BubbleShoot || {};
BubbleShoot.Board = (function($){
  var NUM_ROWS = 9; (1)
  var NUM_COLS = 32; (2)
  var Board = function(){
    var that = this;
    var rows = createLayout(); (3)
    this.getRows = function(){ return rows;}; (4)
    return this;
  };
  var createLayout = function(){
    var rows = [];
    for(var i=0;i<NUM_ROWS;i++){ (5)
      var row = [];
      var startCol = i%2 == 0 ? 1 : 0; (6)
      for(var j=startCol;j<NUM_COLS;j+=2){
        var bubble = BubbleShoot.Bubble.create(i,j); (7)
        row[j] = bubble;
      };
      rows.push(row);
    };
    return rows;
  };
  return Board;
})(jQuery);
```

`NUM_ROWS` (1) i `NUM_COLS` (2) to stałe determinujące liczbę rzędów i kolumn dla siatki bąbelków. Liczba kolumn może się wydawać wysoka, ponieważ na pewno nie będziemy mieli aż 32 bąbelków w rzędzie. Powodem tak dużej liczby kolumn jest to, że stworzymy wpis siatki co połowę szerokości bąbelka, ponieważ rzędy parzyste i nieparzyste są przesunięte względem siebie. Ta decyzja projektowa pozwala osiągnąć lepszy efekt wizualny, a bąbelki będą wyglądały jak ułożone w stos. Pozwala to również opracować ciekawsze kąty wystrzału.

Wszystkie bąbelki z pierwszego rzędu i każdy kolejny nieparzysty rząd będą miały nieparzyste koordynaty $y$. Rzędy są numerowane kolejnymi liczbami całkowitymi, ale tablica, którą wykorzystamy, będzie indeksowana od zera: pierwszy rząd będzie miał indeks 0, drugi będzie miał indeks 1 i tak dalej. Dlatego koordynaty bąbelka $(x, y)$ rozpoczynające się od lewej górnej krawędzi planszy będą numerowane zgodnie z rysunkiem 3.2. Ten sposób określania koordynat w przypadku częściowo obsadzonej siatki pozwala uniknąć wartości cząstkowych i ułamków. Ponadto możemy przechowywać układ planszy w tablicach indeksowanych liczbami całkowitymi. Praca z liczbami całkowitymi zamiast liczb zmiennoprzecinkowych nie wpływa na proces obliczania kolizji, ale sprawia, że kod jest bardziej czytelny.




*Rysunek 3.2. Koordynaty bąbelków w siatce gry*

W kodzie wywołamy teraz funkcję `createLayout` ❸, która zwraca dwuwymiarową tablicę rzędów i kolumn. W następnym wierszu udostępniamy publiczny dostęp do tej tablicy ❹. Kiedy mamy już obiekt `Board`, możemy pobrać bąbelek na dowolnej pozycji. Na przykład aby pobrać bąbelek na koordynatach (4,1), napisalibyśmy:

```
var rows = board.getRows();
var row = rows[1];
var bubble = row[4];
```

Dostęp do bąbelków uzyskujemy za pośrednictwem numeru rzędu i kolumny. Najpierw pobieramy wszystkie rzędy za pomocą `board.getRows`, a potem pierwszy rząd przechowujemy jako `row`. Następnie uzyskujemy dostęp do czwartego bąbelka za pomocą numeru kolumny. Ponieważ tablica `row` ma tylko połowę wartości, wszystkie nieparzyste wpisy w parzyście zaindeksowanych rzędach (począwszy od 0) i wszystkie parzyste wpisy dla nieparzystych kolumn będą miały wartość `null`.

Funkcja `createLayout` zawiera pętlę ❺. Dla każdego rzędu, który chcemy stworzyć, `startCol` ❻ oblicza, czy zacząć od kolumny 1 czy 0, w zależności od tego, czy rząd jest nieparzysty czy parzysty. Następnie kolejna pętla przechodzi do maksymalnej liczby kolumn, tworząc nowy obiekt `Bubble` ❼ i dodając go do tablicy rzędu, która jest potem zwracana.

Aby ta funkcja mogła zadziałać, musimy przystosować klasę `Bubble` tak, żeby przyjmowała koordynaty rzędu i kolumny, i wprowadzić zmiany w metodzie `Bubble.create`. To, że obiekt `Bubble` — dzięki temu, iż przechowuje koordynaty — będzie znał swoje położenie na planszy, przyda się również podczas obliczania grup, które mają pęknąć. Jeżeli znamy pozycję bąbelka, możemy uzyskać do niego dostęp w strukturze przechowywanej w obiekcie `Board`. Następnie, mając bąbelek, możemy odpytać go o jego pozycję. Każdy bąbelek będzie miał właściwość `type`, która odnosi się do jego koloru, a właściwość ta będzie ustalana podczas tworzenia.

Kiedy zaczniesz tworzyć gry według swoich pomysłów, kluczowe staną się decyzje na temat tego, gdzie przechowywać dane i jak uzyskiwać do nich dostęp. Twoje rozwiązanie będzie uzależnione od typu tworzonej gry. W grze *Bubble*

*Shooter* przechowujemy relatywnie małą liczbę bąbelków w ramach obiektu `Board`. Aby uzyskać informację o bąbelku, możemy uzyskać dostęp do tych danych, przeglądając tablicę `rows`.

W zależności od tego, jak chcielibyśmy wykorzystać dane dotyczące bąbelków, ta metoda może nie być najbardziej elegancka. Na przykład załóżmy, że chcemy znaleźć wszystkie czerwone bąbelki w grze. Aktualnie musielibyśmy przejść po wszystkich elementach planszy, sprawdzić, czy dany bąbelek jest czerwony, a następnie zapisać wynik. Plansza gry jest mała, więc nowoczesne procesory szybko uporają się z tym zadaniem. O ile nie będziemy uruchamiać sprawdzania kolorów zbyt wiele razy na sekundę, aktualna struktura kodu powinna zadziałać.

Ale wyobraź sobie teraz, że na tablicy znajdują się *tysiące* bąbelków. Przechodzenie w pętli po wszystkich bąbelkach tylko po to, aby znaleźć czerwone, byłoby zbyt czasochłonne. Moglibyśmy natomiast przechowywać bąbelki w tablicach wielowymiarowych — jedna dla wszystkich czerwonych bąbelków, jedna dla zielonych i tak dalej — aby mieć natychmiastowy dostęp do wszystkich bąbelków danego koloru. Takie rozwiązanie również ma swoje wady: aby sprawdzić, czy dane miejsce na planszy jest zajęte przez bąbelek dowolnego koloru, musielibyśmy przejrzeć wiele tablic.

Jeżeli masz tylko ogólne pojęcie na temat tego, jak szybko procesor może wykonać daną operację, najlepiej, aby kod był jasny i prosty. Jeśli w Twoją grę da się grać i działa wystarczająco szybko, nie będziesz musiał eksperymentować z różnymi metodami dostępu do danych. Jeśli zidentyfikujesz wąskie gardła, będziesz musiał zrefaktoryzować część kodu, aby podnieść ich wydajność. Tworzenie gier to proces iteracyjny — analizowanie i poprawianie istniejącego kodu jest równie częste co pisanie nowego.

To, jak zaprojektujesz obiekty i gdzie będziesz przechowywał ich dane, zależy od rodzaju gry. Pamiętaj jednak o tym, że jeżeli obiekt `Game` musi z tych danych korzystać, będziesz musiał w ten czy inny sposób zapewnić mu do nich dostęp. To, czy dane będą przechowywane bezpośrednio w zmiennej czy w tablicy wewnątrz obiektu `Game` lub wewnątrz obiektu pośredniego, do którego `Game` ma dostęp (takiego jak obiekt `Board` w naszej grze), nie zmienia faktu, że jeżeli kod potrzebuje tych danych do podejmowania decyzji, będzie musiał mieć dostęp do stanu obiektu.

Aby dodać wsparcie dla przechowywania koloru i pozycji bąbelka na planszy, zmodyfikuj plik *bubble.js* zgodnie z poniższym:

*bubble.js*

```
var BubbleShoot = window.BubbleShoot || {};
BubbleShoot.Bubble = (function($){
  var Bubble = function(row,col,type,sprite){
    var that = this;
    this.getType = function(){ return type;};
    this.getSprite = function(){ return sprite;};
    this.getCol = function(){ return col;};
    this.getRow = function(){ return row;};
  };
  Bubble.create = function(rowNum,colNum,type){
    if(type === undefined){
```

```
      type = Math.floor(Math.random() * 4); ❹
    };
    var sprite = $(document.createElement("div"));
    sprite.addClass("bubble");
    sprite.addClass("bubble_" + type);
    var bubble = new Bubble(rowNum,colNum,type,sprite);
    return bubble;
  };
  return Bubble;
})(jQuery);
```

Obiekt `Bubble`, oprócz obiektu `sprite`, przyjmuje teraz koordynaty siatki i typ bąbelka ❶. Typ odnosi się do koloru określonego w pliku *game.css*. Metoda `Bubble.create` przyjmuje te same parametry ❷ — jeżeli typ nie zostanie przekazany ❸, jeden z czterech kolorów zostanie wybrany losowo ❹.

Mamy teraz obiekt `Board`, mnóstwo bąbelków oraz ich typy i pozycje. Jednak wszystkie te informacje znajdują się w pamięci i są przechowywane w ramach właściwości `rows` obiektu `Board`. Teraz, korzystając z tych informacji, wyrenderujemy poziom, dzięki czemu gracze będą mogli zobaczyć planszę.

## Renderowanie poziomu

Rysowanie poziomu to idealne zadanie dla klasy `ui`, ponieważ `ui` reprezentuje stan gry, ale na niego nie wpływa.

Odseparowanie kodu, który oblicza pozycję obiektu, od kodu, który renderuje obiekt na ekranie, jest zasadą, której powinieneś przestrzegać podczas pracy nad wszystkimi swoimi grami. Dzięki temu nie tylko odseparujesz kod renderujący od logiki gry, poprawiając tym samym czytelność, ale będziesz mógł również łatwiej zmieniać sposób renderowania. Gdyby, na przykład, plansza naszej gry była większa i nie mieściła się na ekranie, a my chcielibyśmy zaimplementować funkcjonalność zbliżania i oddalania planszy, moglibyśmy zmienić kod renderujący tablicę tak, aby przesuwał pozycje renderowania albo skalował w górę czy w dół w celu uzyskania różnych rozmiarów tablicy. Waga odseparowywania kodu renderowania od logiki stanie się oczywista, kiedy w rozdziale 6. przejdziemy od sprite'ów tworzonych w oparciu o elementy DOM do korzystania z elementu `canvas`.

Ponieważ stworzenie obiektu `bubble` wiąże się ze stworzeniem elementu DOM dla sprite'a, proces renderowania musi umieścić ten element w dokumencie i poprawnie ustawić jego pozycję. Musimy wykonać poniższe kroki:

1. W pętli przejść po wszystkich rzędach i kolumnach i wyciągnąć każdy z obiektów `bubble`.
2. Zapisać kod HTML bąbelka w drzewie DOM.
3. Ustawić bąbelek na odpowiedniej pozycji.

Kod, który dodasz teraz, będzie wykonywał te kroki. Otwórz *ui.js*, dodaj nową metodę (`drawBoard`) po metodzie `fireBubble`, a następnie na początku pliku dodaj stałą `ROW_HEIGHT`:

*ui.js*

```
var BubbleShoot = window.BubbleShoot || {};
BubbleShoot.ui = (function($){
  var ui = {
    BUBBLE_DIMS : 44,
    ROW_HEIGHT : 40,
    init : function(){
    },
    fireBubble : function(bubble,coords,duration){
      --cięcie--
    },
    drawBoard : function(board){
      var rows = board.getRows(); ❶
      var gameArea = $("#board");
      for(var i=0;i<rows.length;i++){
        var row = rows[i];
        for(var j=0;j<row.length;j++){ ❷
          var bubble = row[j];
          if(bubble){ ❸
            var sprite = bubble.getSprite(); ❹
            gameArea.append(sprite); ❺
            var left = j * ui.BUBBLE_DIMS/2;
            var top = i * ui.ROW_HEIGHT;
            sprite.css({ ❻
              left : left,
              top : top
            });
          };
        };
      };
    }
  };
  return ui;
})(jQuery);
```

Metoda `drawBoard` pobiera rzędy i kolumny planszy ❶ i przechodzi po nich w pętli ❷. Jeżeli na aktualnej pozycji znajduje się bąbelek ❸ (pamiętaj, że ze względu na sposób rozmieszczenia bąbelków na siatce co druga pozycja w tablicy ma wartość `null`), `drawBoard` pobiera obiekt `sprite` ❹, dodaje go na planszę ❺ i przed ustaleniem jego pozycji oblicza koordynaty ❻.

Aby określić pozycję bąbelka, `drawBoard` najpierw oblicza koordynatę `left`, poprzez przemnożenie numeru kolumny bąbelka przez połowę jego szerokości. Aby obliczyć koordynatę `top`, użyjemy wartości nieco mniejszej niż wysokość `BUBBLE_DIMS`. Rzędy parzyste i nieparzyste ustawione są naprzemiennie i chcemy, aby bąbelki wyglądały na dobrze dopasowane. W celu uzyskania efektu ułożenia w warstwach separacja pionowa będzie nieco mniejsza niż odległość pozioma. Na początku pliku *ui.js* stała `ROW_HEIGHT` została ustawiona na 40, czyli o 4 piksele mniej niż wysokość. Wartość ta została dobrana metodą prób i błędów, a nie za pomocą obliczeń geometrycznych: modyfikuj wartości, aż plansza będzie wyglądała tak jak chcesz.

Przeładuj stronę i kliknij przycisk *Nowa gra* — powinieneś zobaczyć wyrenderowaną planszę. Możesz nawet wystrzelić bąbelek, aczkolwiek niestety przeleci on przez pozostałe bąbelki, nie zderzając się z żadnym z nich, i wyleci poza ekran.

Ponieważ mamy tylko jeden bąbelek, to aby strzelić ponownie, musimy odświeżyć stronę. Zanim zaczniemy pracę nad wykrywaniem kolizji, upewnimy się, czy możemy strzelać kolejnymi bąbelkami.

## Kolejka bąbelków

Chociaż gracz będzie miał do dyspozycji tylko skończoną liczbę bąbelków, gra musi zapewnić kolejne bąbelki do wystrzelenia. Dlatego musimy dodać funkcję, która stworzy nowy bąbelek, doda go do drzewa DOM i ustawi go w kolejce zaraz po wystrzeleniu poprzedniego.

W pliku *game.js* dodaj poniższe zmienne i funkcje oraz zmień inicjalizację dla `curBubble` tak, aby wywoływała nową funkcję `getNextBubble`:

*game.js*

```
var BubbleShoot = window.BubbleShoot || {};
BubbleShoot.Game = (function($){
var Game = function(){
  var curBubble;
  var board;
  var numBubbles; ❶
  var MAX_BUBBLES = 70; ❷
  this.init = function(){
    $(".but_start_game").bind("click",startGame);
  };
  var startGame = function(){
    $(".but_start_game").unbind("click");
    numBubbles = MAX_BUBBLES; ❸
    BubbleShoot.ui.hideDialog();
    curBubble = getNextBubble();
    board = new BubbleShoot.Board();
    BubbleShoot.ui.drawBoard(board);
    $("#game").bind("click",clickGameScreen);
  };
  var getNextBubble = function(){
    var bubble = BubbleShoot.Bubble.create();
    bubble.getSprite().addClass("cur_bubble");
    $("#board").append(bubble.getSprite());
    BubbleShoot.ui.drawBubblesRemaining(numBubbles); ❹
    numBubbles--;
    return bubble;
  };
  var clickGameScreen = function(e){
    var angle = BubbleShoot.ui.getBubbleAngle(curBubble .getSprite(),e);
    var duration = 750;
    var distance = 1000;
    var distX = Math.sin(angle) * distance;
    var distY = Math.cos(angle) * distance;
    var bubbleCoords = BubbleShoot.ui.getBubbleCoords(curBubble .getSprite());
    var coords = {
      x : bubbleCoords.left + distX,
      y : bubbleCoords.top - distY
```

```
    };
    BubbleShoot.ui.fireBubble(curBubble,coords,duration);
    curBubble = getNextBubble(); ❺
  };
  return Game;
})(jQuery);
```

Nowy kod najpierw tworzy zmienną ❶ do przechowywania liczby wystrzelonych przez użytkownika bąbelków. Ponieważ liczba wystrzelonych bąbelków jest liczbą całkowitą (to podstawowy typ danych), przechowamy ją jako zmienną klasy `Game`. Jeżeli na przykład mielibyśmy ograniczenie czasowe na ukończenie poziomu, to zamiast tworzyć kolejne zmienne w klasie `Game` moglibyśmy stworzyć obiekt przechowujący pozostały czas i liczbę pozostałych bąbelków. Aktualnie jednak zmienna dobrze odpowiada naszym potrzebom.

W kodzie ustawiana jest również stała zawierająca maksymalną liczbę dozwolonych bąbelków ❷. Kiedy rozpoczyna się poziom, liczba pozostałych bąbelków ustawiana jest na wartość `MAX_BUBBLES` ❸, a z pliku *ui.js* wywoływana jest nowa funkcja, wyświetlająca na ekranie liczbę pozostałych bąbelków ❹. I wreszcie, za każdym razem, kiedy wystrzelony zostanie bąbelek, wywoływana jest funkcja `getNextBubble` przygotowująca następny ❺.

Chcemy, aby gracz widział liczbę pozostałych w zapasie bąbelków, dlatego w pliku *ui.js*, w ramach obiektu `ui`, stworzymy metodę `drawBubblesRemaining`:

*ui.js*

```
var BubbleShoot = window.BubbleShoot || {};
BubbleShoot.ui = (function($){
  var ui = {
    BUBBLE_DIMS : 44,
    ROW_HEIGHT : 40,
    --cięcie--
    drawBoard : function(board){
      --cięcie--
    },
    drawBubblesRemaining : function(numBubbles){
      $("#bubbles_remaining").text(numBubbles);
    }
  };
  return ui;
})(jQuery);
```

Dodatkowo musimy wyświetlić liczbę pozostałych bąbelków, dlatego dodamy nowy element w pliku *index.html*:

*index.html*

```
<div id="game">
  <div id="board"></div>
  <div id="bubbles_remaining"></div>
</div>
```

W pliku *main.css* trzeba też dodać style dla elementu `bubbles_remaining`:

*main.css*

```
#bubbles_remaining
{
  position: absolute;
  left: 479px;
  top: 520px;
  width: 50px;
  font-size: 26px;
  font-weight: bold;
  color: #000;
  text-align: center;
}
```

Teraz odśwież grę. Powinieneś móc wystrzeliwać bąbelki, otrzymywać nowy bąbelek zaraz po wystrzeleniu poprzedniego (dopóki nie wykorzystasz 70 bąbelków lub innej liczby określonej w stałej `MAX_BUBBLES`) i natychmiast go wystrzeliwać.

Zazwyczaj możesz rozbić grę na powtarzające się **pętle tur**. Pętla jest przeważnie inicjowana przez działanie gracza i zamykana po zakończeniu tego działania. W naszej grze pętla rozpoczyna się, kiedy gracz kliknie bąbelek, aby go wystrzelić, i kończy się, gdy kolejny bąbelek jest gotowy do wystrzelenia. W tym momencie mamy podstawową pętlę tur, ale aby utworzyć grę, musimy dopracować środkową fazę pętli i dzięki temu móc określać, gdzie bąbelek powinien się zatrzymać i czy jakieś bąbelki powinny pęknąć.

# Wykrywanie kolizji

Chociaż możesz już strzelać bąbelkami, przelatują one przez całą planszę bez żadnego wpływu na nią. Projekt gry wymaga, aby zderzały się z pozostałymi bąbelkami, albo stając się częścią planszy, albo powodując pęknięcie grup bąbelków o tym samym kolorze. Kolejnym zadaniem jest wykrycie wystąpienia kolizji.

Możemy obliczać kolizję na dwa sposoby:

- Przesunąć sprite o kilka pikseli do przodu dla każdej klatki i wtedy próbować wykrywać nakładanie się na siebie sprite'ów. Jeżeli takie nałożenie wystąpi, będziemy wiedzieć, że trafiliśmy w inny bąbelek.
- Jeszcze przed rozpoczęciem ruchu bąbelka wykorzystać geometrię do obliczenia, w którym miejscu sprite może zderzyć się z innym.

W szybkich grach zręcznościowych możesz wybrać pierwszą opcję, pod warunkiem że nie będzie możliwości, iż obiekty przelecą przez siebie bez wykrycia kolizji. Takie przenikanie może mieć miejsce, kiedy obiekty poruszają się z dużą szybkością, a wykrycie kolizji następuje dopiero po przemieszczeniu obiektu o wiele pikseli od ostatniego sprawdzenia. Na przykład w grze, w której wystrzeliwujesz kulę w ścianę o grubości 30 centymetrów, to, że kula zderzy się ze ścianą, będzie pewne tylko wówczas, jeżeli będziesz sprawdzał wystąpienie kolizji co 30 centymetrów. Jeśli zamiast tego będziesz sprawdzał kolizje co 60 centymetrów, sprawdzenie może wypaść tuż przed ścianą i kolizja nie zostanie wykryta.

Po 60 centymetrach, kiedy wykonane zostanie kolejne sprawdzenie, kula będzie już za ścianą i znowu kolizja nie wystąpi.

Aby rozwiązać problem szybko poruszających się obiektów, możemy zapewnić, aby kroki były zawsze wystarczająco małe, by przeniknięcie nigdy nie miało miejsca, wymaga to jednak więcej obliczeń, które mogą okazać się niemożliwe bez znacznej mocy obliczeniowej. Wystąpienie tego problemu jest bardziej prawdopodobne w środowisku przeglądarkowym: ponieważ nie możemy wiedzieć, z jakiego sprzętu korzysta użytkownik, nie możemy zakładać dostępności mocy obliczeniowej.

Druga opcja — wykorzystanie geometrii — jest bardziej dokładna, jeżeli tylko jest możliwa. Na szczęście nasza gra ma stosunkowo proste właściwości geometryczne. Niestety ta opcja nie jest możliwa w grach, w których sprite'y mają bardziej skomplikowane kształty. W takim przypadku będziesz musiał w poszczególnych klatkach sprawdzać, czy sprite'y nachodzą na siebie, i dokładnie przetestować grę, aby się upewnić, że nie ma skutków ubocznych. Dla *Bubble Shootera* wykorzystamy podejście geometryczne, ponieważ gra ma następujące właściwości:

- gra odbywa się na regularnej planszy;
- wszystkie obiekty (bąbelki) są identyczne;
- pracujemy tylko w dwóch wymiarach;
- gracz przemieszcza tylko jeden obiekt;
- wszystkie obiekty to proste kształty geometryczne (okręgi), dlatego obliczenie zetknięcia się krawędzi jest proste.

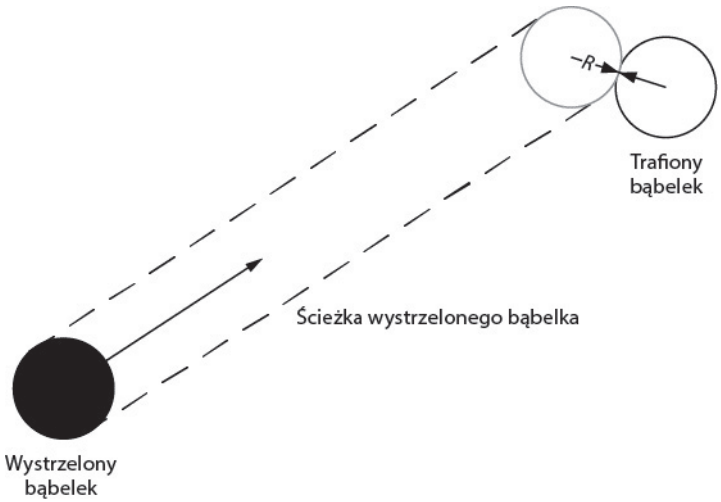
Te warunki powodują, że obliczenia kolizji są relatywnie proste. Ponieważ tworzenie gier z reguły wiąże się z wykorzystaniem geometrii, dobre podstawy z trygonometrii i wektorów są nieodzowne. W kolejnym podrozdziale omówię obliczenia geometryczne wykorzystywane w grze. Potem przekształcimy teorię w kod.

## Geometria kolizji

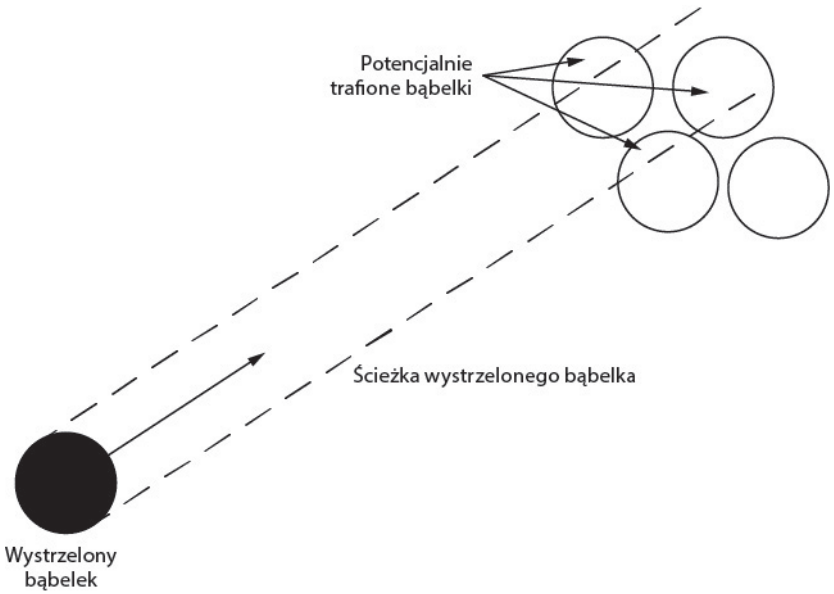
Kiedy musisz obliczyć kolizje, przed napisaniem kodu wykrywającego rozrysuj potrzebne obliczenia na kartce. Będziesz mógł wtedy zwizualizować wartości potrzebne do obliczeń (zobacz rysunek 3.3).

Wystrzelony bąbelek powinien spowodować kolizję, kiedy jego środek minie punkt znajdujący się w odległości $2R$ (gdzie $R$ to promień bąbelka) od środka innego bąbelka. Będzie to oznaczać, że dwa obwody się stykają. Ponieważ punkt styku zawsze będzie pod kątem 90 stopni do krawędzi bąbelka, z którym nastąpiła kolizja, musimy sprawdzać kolizje tylko wówczas, jeżeli ścieżka środka poruszającego się bąbelka znajdzie się w odległości $2R$ od środka innego bąbelka.

Aby określić, gdzie wystąpi kolizja, musimy sprawdzić każdy bąbelek na planszy i ustalić, czy przechodzi przezeń ścieżka wystrzelonego bąbelka. Jeżeli ścieżka koliduje z wieloma bąbelkami, jak na rysunku 3.4, musimy się upewnić, że wybierzemy bąbelek, który spowoduje pierwszą kolizję, czyli ten, do którego bąbelek wystrzelony ma najkrótszą drogę.

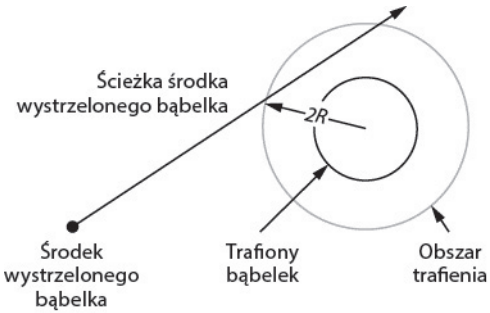



*Rysunek 3.3. Wizualizacja geometrii potrzebnej do obliczenia kolizji bąbelków*




*Rysunek 3.4. Wystrzelony bąbelek może być na torze kolizyjnym z wieloma bąbelkami*

Wykrycie kolizji jest równoznaczne z wykryciem, kiedy wektor narysowany ze środka wystrzeliwanego bąbelka przetnie się z kołem o promieniu dwukrotnie większym od promienia bąbelków. Będziemy to nazywać *obszarem trafienia*. Rysunek 3.5 przedstawia inny sposób na narysowanie tej koncepcji, dzięki któremu będziemy mogli łatwiej o tym myśleć.




*Rysunek 3.5. Kolizja następuje, jeżeli ścieżka przemieszczania się wystrzelonego bąbelka przetnie się z okrągłym obszarem trafienia bąbelka stacjonarnego*

Na tym schemacie małe wypełnione kółko oznacza środek wystrzeliwanego bąbelka. Bąbelek, z którym nastąpi kolizja, to wewnętrzny okrąg, a przecięcie z obszarem trafienia (punkt oznaczony strzałką $2R$, która jest dwukrotnym promieniem bąbelka) to miejsce zatrzymania się wystrzelonego bąbelka.

Zmienienie schematu we wzór matematyczny wiąże się z wykorzystaniem wektorów. Zamiast omawiać matematykę przed programowaniem, przejdźmy bezpośrednio do potrzebnego kodu JavaScriptu wraz z adnotacjami.

### UPRASZCZANIE OBSZARÓW TRAFIENIA

Ponieważ pracujemy z okręgami, stworzenie obszaru trafienia jest łatwiejsze, niż gdybyśmy, na przykład, animowali biegającą i skaczącą postać, jak w grze platformowej. W takim przypadku możesz nie chcieć wykrywać kolizji na podstawie nachodzących na siebie pikseli ze względu na możliwe problemy wydajnościowe; zamiast tego mógłbyś uprościć geometrię głównego bohatera i stworzyć prostokątny obszar trafienia. Nie wszystkie gry mogą skorzystać z tego rozwiązania. Jeżeli jednak będziesz w stanie zredukować skomplikowany kształt do pojedynczych figur geometrycznych, będziesz mógł bardziej precyzyjnie wykrywać kolizje, a proces ten będzie wymagał mniejszej mocy obliczeniowej niż w przypadku sprawdzania, czy piksele się nakładają. Zawsze szukaj kreatywnych, wydajnych rozwiązań, które pozwolą uniknąć korzystania z brutalnych, zasobożernych technik.

Obliczenia to duży blok kodu o konkretnym zadaniu, dlatego umieścimy go w osobnym pliku. Stwórz plik o nazwie *collision-detector.js* i dodaj go do wywołania `Modernizr.load` w pliku *index.html*. Zapisz w nim poniższy kod:

*collision-detector.js*

```
var BubbleShoot = window.BubbleShoot || {};
BubbleShoot.CollisionDetector = (function($){
  var CollisionDetector = {
    findIntersection : function(curBubble,board,angle){
      var rows = board.getRows();
      var collision = null;
      var pos = curBubble.getSprite().position();
```

```
var start = {
left : pos.left + BubbleShoot.ui.BUBBLE_DIMS/2,
top : pos.top + BubbleShoot.ui.BUBBLE_DIMS/2
};
var dx = Math.sin(angle);
var dy = -Math.cos(angle);
for(var i=0;i<rows.length;i++){
  var row = rows[i];
  for(var j=0;j<row.length;j++){
    var bubble = row[j];
    if(bubble){
      var coords = bubble.getCoords(); ❶
      var distToBubble = {
      x : start.left - coords.left,
      y : start.top - coords.top
      };
      var t = dx * distToBubble.x + dy * distToBubble.y;
      var ex = -t * dx + start.left;
      var ey = -t * dy + start.top;
      var distEC = Math.sqrt((ex - coords.left) * (ex - coords.left) +
      (ey - coords.top) * (ey - coords.top));
      if(distEC<BubbleShoot.ui.BUBBLE_DIMS * .75){
        var dt = Math.sqrt(BubbleShoot.ui.BUBBLE_DIMS * BubbleShoot.
        ui.BUBBLE_DIMS - distEC * distEC);
        var offset1 = {
          x : (t - dt) * dx,
          y : -(t - dt) * dy
        };
        var offset2 = {
          x : (t + dt) * dx,
          y : -(t + dt) * dy
        };
        var distToCollision1 = Math.sqrt(offset1.x * offset1.x +
        offset1.y * offset1.y);
        var distToCollision2 = Math.sqrt(offset2.x * offset2.x +
        offset2.y * offset2.y);
        if(distToCollision1 < distToCollision2){
        var distToCollision = distToCollision1;
          var dest = {
            x : offset1.x + start.left,
            y : offset1.y + start.top
          };
        }else{
          var distToCollision = distToCollision2;
          var dest = {
          x : -offset2.x + start.left,
          y : offset2.y + start.top
          };
        }
        if(!collision || collision.distToCollision>distToCollision){
          collision = {
            bubble : bubble,
            distToCollision : distToCollision,
            coords : dest
          };
        };
      };
```

```
          };
        };
      };
      return collision;
    }
  };
  return CollisionDetector;
})(jQuery);
```

Za chwilę omówimy kod w pliku *collision-detector.js*. Najpierw jednak zwróć uwagę na wywołanie z pliku *bubble.js* nowej metody `getCoords` ❶, która zwraca koordynaty *(x, y)* środka bąbelka obliczone na podstawie jego pozycji w rzędzie i kolumnie. Będziesz musiał zmodyfikować klasę bąbelka i dodać nową metodę:

*bubble.js*

```
var BubbleShoot = window.BubbleShoot || {};
BubbleShoot.Bubble = (function($){
  var Bubble = function(row,col,type,sprite){
    var that = this;
    this.getType = function(){ return type;};
    this.getSprite = function(){ return sprite;};
    this.getCol = function(){ return col;};
    this.getRow = function(){ return row;};
    this.getCoords = function(){
      var coords = {
        left : ❶that.getCol() * ❷BubbleShoot.ui.BUBBLE_DIMS/2 +
        ❺BubbleShoot.ui.BUBBLE_DIMS/2,
        top : ❸that.getRow() * ❹BubbleShoot.ui.ROW_HEIGHT +
        ❺BubbleShoot.ui.BUBBLE_DIMS/2
      };
      return coords;
    }
  };
  Bubble.create = function(rowNum,colNum,type){
    --cięcie--
  };
  return Bubble;
})(jQuery);
```

Koordynaty bąbelka są łatwe do obliczenia: zaczynasz od znalezienia poszczególnych koordynat punktu w lewym górnym rogu. Koordynata *x* (od lewej) to liczba kolumny ❶ pomnożona przez połowę szerokości sprite'a bąbelka ❷. Koordynata *y* (od góry) to numer rzędu ❸ pomnożony przez wysokość rzędu ❹, co stanowi wartość nieznacznie mniejszą niż wysokość sprite'a bąbelka. Aby znaleźć środek bąbelka, do obu koordynat wystarczy dodać połowę wymiarów bąbelka ❺.

Podczas opracowywania logiki gry najczęściej skupiać się będziesz na koordynatach środka obiektu, natomiast do celów renderowania podawane są koordynaty lewego górnego rogu obiektu oraz jego szerokość i wysokość. Wbudowanie w obiekcie wygodnej metody konwertującej z koordynat środka na koordynaty lewego górnego rogu zaoszczędzi Ci konieczności pisania obliczeń za każdym razem, kiedy zechcesz zmienić koordynaty.

## Logika wykrywania kolizji

Omówmy teraz całą metodę `findIntersection` z pliku *collision-detector.js*. Jeżeli nie chcesz teraz zagłębiać się w obliczenia matematyczne, możesz pominąć tę część — omówię tu wyłącznie matematykę pozwalającą obliczyć kolizje i nie będę przedstawiał żadnych nowych koncepcji z dziedziny HTML5 ani tworzenia gier. Wiedz jednak, że w prawie każdej grze, jaką będziesz tworzył, będziesz musiał rozbijać złożone interakcje między obiektami w modelu, w którym będziesz mógł nimi operować za pomocą relatywnie prostej matematyki.

### Pozycja początkowa i wektor kierunku

Pierwszą częścią pliku *collision-detector.js* jest standardowy początek biblioteki:

```
var BubbleShoot = window.BubbleShoot || {};
BubbleShoot.CollisionDetector = (function($){
  var CollisionDetector = {
```

Stworzyliśmy obiekt o nazwie `CollisionDetector`. Spójrzmy teraz na pierwszą metodę tego obiektu:

```
findIntersection : function(curBubble,board,angle){
```

Podczas wywoływania `CollisionDetector` będziesz korzystał z `BubbleShoot.CollisionDetector.findIntersection`. Metoda ta przyjmuje parametry `curBubble` (instancję klasy `Bubble`), zmienną `board` (instancję klasy `Board`) i kąt, pod którym wystrzeliwany jest bąbelek — są to wystarczające informacje o sytuacji początkowej.

Przyjrzyjmy się pierwszym zmiennym wewnątrz funkcji `findIntersection`:

```
var rows = board.getRows();
var collision = null;
```

Będziemy przechodzić w pętli po każdym z rzędów i sprawdzać kolizję, dlatego do zmiennej lokalnej zapisujemy rzędy planszy. Zakładając, że domyślnie nie ma kolizji, będzie to stan zwracany przez funkcję, jeżeli nie nastąpią żadne interakcje. W rezultacie, jeżeli wystrzelony bąbelek nie trafi w inny, będzie poruszał się do przodu.

Początkowa wartość zmiennej `collision` to `null` zamiast `false`, ponieważ jeżeli nastąpi przecięcie, zamiast wartości boolowskiej informującej o tym, czy kolizja wystąpiła, czy nie, zmienna będzie przechowywała bąbelki, z którymi ta kolizja miała miejsce, a także inne informacje. Musimy wiedzieć, że kolizja wystąpiła (co byłoby informacją „prawda – fałsz"), ale co ważniejsze, musimy odesłać informacje o tym, gdzie i z czym zderzył się bąbelek:

```
var pos = curBubble.getSprite().position();
var start = {
  left : pos.left + BubbleShoot.ui.BUBBLE_DIMS/2,
  top : pos.top + BubbleShoot.ui.BUBBLE_DIMS/2
};
```

Kolejna para zmiennych pobiera początkową pozycję bąbelka (na ekranie) jako obiekt z właściwościami `top` i `left`:

```
var dx = Math.sin(angle);
var dy = -Math.cos(angle);
```

Na koniec `dx` i `dy` określają, jak bardzo bąbelek przemieszcza się w lewo lub prawo (`dx`) oraz w górę (`dy`) w stosunku do całkowitej odległości, o jaką się przemieści. Mając zdefiniowane te zmienne, możemy w pętli przejść po rzędach i kolumnach planszy:

```
for(var i=0;i<rows.length;i++){
  var row = rows[i];
  for(var j=0;j<row.length;j++){
    var bubble = row[j];
    if(bubble){
```

Zaczniemy od lewego górnego rogu planszy i będziemy posuwać się w dół i w prawo. Ponieważ bąbelki wystrzeliwujemy tylko w górę, wiemy, że bąbelek nigdy nie wejdzie w kolizję z innym bąbelkiem znajdującym się wyżej na planszy. Wiemy również, że jeżeli na ścieżce bąbelka znajdzie się kilka bąbelków, z którymi potencjalnie nastąpiłaby kolizja, chcemy zwrócić tego, do którego wystrzelony bąbelek ma najkrótszą drogę — czyli tego, z którym kolizja nastąpi jako pierwszym. Pamiętaj, że ponieważ w kolumnach co druga wartość to `null`, zanim zaczniemy obliczenia, musimy też upewnić się, czy na danej pozycji znajduje się bąbelek — stąd sprawdzenie `if(bubble)`.

## Obliczanie kolizji

Teraz do sprawdzenia, czy obszar zderzenia wystrzelonego bąbelka zderzył się z innym bąbelkiem, musimy wykorzystać geometrię. Określimy, gdzie wektor definiowany przez $(dx, dy)$, który zaczyna się w środku wystrzeliwanego bąbelka, przecina się z okręgiem z rysunku 3.4. Zacznijmy od równania okręgu:

$$(x - c_x)^2 + (y - c_y)^2 = r^2$$

Gdzie $x$ i $y$ to punkty na obwodzie okręgu, $c_x$ i $c_y$ są koordynatami środka okręgu, a $r$ jest promieniem. Punkty te będą potrzebne do znalezienia odległości do bąbelka początkowego.

```
var coords = bubble.getCoords();
var distToBubble = {
  x : start.left - coords.left,
  y : start.top - coords.top
};
```

Ta część pętli zawiera bąbelek, dla którego sprawdzamy kolizję, dlatego musimy pobrać koordynaty środka $c_x$ i $c_y$ (w powyższym kodzie zapisane są w zmiennej `coords`) i obliczyć odległość między tym punktem a koordynatami wystrzelonego bąbelka. Nie wiemy jeszcze, czy kolizja wystąpi.

Wystrzeliwany bąbelek podąża ścieżką wyznaczaną przez punkty definiowane przez równania:

$$p_x = e_x + td_x$$
$$p_y = e_y + td_y$$

Gdzie $p_x$ i $p_y$ to punkty na trajektorii środka bąbelka. Obliczanie punktów $p_x$ i $p_y$ ma miejsce w funkcji jQuery `animate` i jest standardowym równaniem przemieszczania punktu wzdłuż linii. Następnie obliczymy $t$ w punkcie na tej linii najbliższym środka bąbelka, dla którego sprawdzamy kolizję:

```
var t = dx * distToBubble.x + dy * distToBubble.y;
```

Ten wiersz mówi nam, w jakim zakresie ruchu bąbelka znajdzie się on najbliżej środka bąbelka, dla którego sprawdzamy kolizję. Mając tę informację, możemy obliczyć koordynaty tego miejsca na ekranie:

```
var ex = -t * dx + start.left;
var ey = -t * dy + start.top;
```

Dzięki tym koordynatom możemy znaleźć odległość $e$ (punkt na linii środka wystrzelonego bąbelka najbliższy środka bąbelka, dla którego sprawdzamy kolizję):

```
var distEC = Math.sqrt((ex - coords.left) * (ex - coords.left) + (ey - coords.top)
↪* (ey - coords.top));
```

Jeżeli odległość `distEC` jest mniejsza niż podwójny promień bąbelka, następuje kolizja. Jeżeli nie, to wystrzelony bąbelek nie wejdzie w kolizję ze sprawdzanym bąbelkiem.

Jeżeli `distEC` ma wartość mniejszą niż trzy czwarte szerokości sprite'a, wiemy, że ścieżka wystrzelonego bąbelka w jakimś miejscu przecina się z obszarem trafienia bąbelka, dla którego sprawdzamy:

```
if(distEC < BubbleShoot.ui.BUBBLE_DIMS * .75){
```

### METODA PRÓB I BŁĘDÓW KONTRA OBLICZENIA

Zauważ, że chociaż `BubbleShoot.ui.BUBBLE_DIMS` podaje wysokość i szerokość sprite'a, porównujemy `distEC` z obrazem, który w rzeczywistości jest odrobinę mniejszy. Przemnożenie `BUBBLE_DIMS` przez wartość `0,75` (która została ustalona metodą prób i błędów) daje średnicę bąbelka funkcjonującą w grze.

Możemy uzyskać dokładniejszą wartość dla `distEC`, mierząc szerokość bąbelka, która dla obrazów w tej książce wynosi `44` piksele. Dzieląc ją przez wartość `BUBBLE_DIMS`, która wynosi `50` pikseli, otrzymamy wynik 0,88. Ta większa wartość może być dokładniejsza, ale wymaga również, aby gracz był celniejszy, próbując trafić bąbelkami w wąskie przesmyki. Dlatego wartość 0,75 daje lepszą grywalność, umożliwia bowiem wykonywanie strzałów, które byłyby bardzo trudne, gdyby obliczenia były dokładniejsze.

Podczas tworzenia gier będziesz często musiał podejmować decyzje na podstawie prób i błędów w równym stopniu co na podstawie obliczeń. W tym przypadku, używając nieznacznie mniejszej wartości, dajesz graczowi okazję do strzelania bąbelkami przez wąskie przestrzenie na planszy. Gracze nie zauważą nieznacznego nagięcia praw fizyki, za to sama gra będzie dla nich przyjemniejsza.

Najprawdopodobniej drugi punkt przecięcia znajdzie się w miejscu wyjścia linii z obszaru trafienia (zobacz rysunek 3.5, który przedstawia linię środka przechodzącą przez obszar trafienia w dwóch miejscach), ale nas interesuje tylko pierwszy punkt. Dwa obliczenia pozwolą mieć pewność, że otrzymaliśmy poprawny punkt przecięcia. Spójrzmy na pierwsze z nich:

```
var dt = Math.sqrt(BubbleShoot.ui.BUBBLE_DIMS * BubbleShoot.ui.BUBBLE_DIMS
↪- distEC * distEC);
```

Znajdujemy tutaj odległość pomiędzy środkiem uderzonego bąbelka i najbliższym punktem ścieżki wystrzelonego bąbelka. Drugie obliczenie to:

```
var offset1 = {
  x : (t - dt) * dx,
  y : -(t - dt) * dy
};
var offset2 = {
  x : (t + dt) * dx,
  y : -(t + dt) * dy
};
```

Punkty na linii przecinającej środek stacjonarnego bąbelka są obliczane jako przesunięcie względem ścieżki wystrzelonego bąbelka w punkcie *t*.

## Odnalezienie poprawnego punktu kolizji

Teraz chcemy wybrać, które przecięcie wystąpi jako pierwsze — czyli który punkt znajduje się najbliżej pozycji, z której wystrzelony został bąbelek — musimy więc znaleźć odległości do wszystkich potencjalnych punktów kolizji:

```
var distToCenter1 = Math.sqrt(offset1.x * offset1.x + offset1.y * offset1.y);
var distToCenter2 = Math.sqrt(offset2.x * offset2.x + offset2.y * offset2.y);
```

Następnie wybierzemy poprawny punkt kolizji i obliczymy, gdzie ma się zatrzymać `curBubble` — w tym celu znów przydadzą się koordynaty początkowe:

```
if(distToCollision1 < distToCollision2){
  var distToCollision = distToCollision1;
  var dest = {
    x : offset1.x + start.left,
    y : offset1.y + start.top
  };
}else{
  var distToCollision = distToCollision2;
  var dest = {
    x : -offset2.x + start.left,
    y : offset2.y + start.top
  };
}
```

W większości przypadków, jeżeli środek wystrzeliwanego bąbelka wejdzie w kolizję z krawędzią innego bąbelka, przejdzie przez nią w dwóch miejscach: raz wchodząc i raz wychodząc z obszaru zderzenia. W rzadkich przypadkach, kiedy bąbelek przeleci obok i wystąpi tylko jeden punkt kolizyjny, otrzymamy dwa identyczne rezultaty, nie ma więc znaczenia, który wykorzystamy.

W tej chwili funkcja przejdzie w pętli po wszystkich bąbelkach na planszy i sprawdzi dla nich kolizje, nie chcemy jednak być informowani o *wszystkich* kolizjach — interesuje nas tylko ta *najwcześniejsza*, która wystąpi najbliżej na trasie wystrzelonego bąbelka.

Aby przechować aktualnie najlepszą kolizję, używamy zmiennej `collision`, która przed rozpoczęciem pętli została ustawiona na `null`. Następnie, za każdym razem, kiedy znajdziemy kolizję, sprawdzamy, czy nowa kolizja znajduje się bliżej od poprzedniej najbliższej kolizji. Jeżeli nie było żadnej wcześniejszej kolizji, pierwsza, jaką znajdziemy, będzie najlepsza. Obiekt `collision` przechowuje referencję do stacjonarnego bąbelka, z którym nastąpiła kolizja, odległość do kolizji i koordynaty miejsca, w którym ta kolizja nastąpiła:

```
        if(!collision || collision.distToCollision>distToCollision){
          collision = {
            bubble : bubble,
            distToCollision : distToCollision,
            coords : dest
```

```
            };
          };
        };
      }
    }
  };
  return collision;
};
```

Teraz funkcja `findIntersection` zwróci obiekt ze wszystkimi potrzebnymi danymi na temat znalezionej kolizji lub wartość `null`, jeżeli żadna kolizja nie zostanie znaleziona. Wszystkie te obliczenia mają miejsce, jeszcze zanim bąbelek zacznie się poruszać.

# Reagowanie na kolizje

Musimy teraz wykorzystać koordynaty kolizji w zmodyfikowanej wersji `clickGame ↪Screen` w pliku *game.js*, dzięki czemu będziemy mogli wystrzeliwać i zatrzymywać bąbelki. Zrobiliśmy pierwszy krok wykrywania kolizji, znajdując bąbelek, z którym nastąpiła kolizja (lub stwierdzając, że kolizja *nie* miała miejsca). Teraz klasa `Game` musi zdecydować, jak na ewentualną kolizję zareagować.

Najpierw sprawdzamy, czy miała miejsce kolizja. Jeżeli tak, przemieszczamy bąbelek do miejsca wystąpienia kolizji. Jeżeli kolizja nie wystąpi, bąbelek wystrzeliwujemy poza ekran. Zmień istniejącą funkcję `clickGameScreen` w pliku *game.js* na następującą:

*game.js*

```
var clickGameScreen = function(e){
  var angle = getBubbleAngle(e);
  var bubble = $("#bubble");
  var duration = 750;
  var distance = 1000;
  var collision = BubbleShoot.CollisionDetector.findIntersection
  ↪(curBubble,board,angle);
  if(collision){
    var coords = collision.coords;
    duration = Math.round(duration * collision.distToCollision / distance); ❶
  }else{
    var distX = Math.sin(angle) * distance;
    var distY = Math.cos(angle) * distance;
    var bubbleCoords = BubbleShoot.ui.getBubbleCoords(curBubble.getSprite());
    var coords = {
      x : bubbleCoords.left + distX,
      y : bubbleCoords.top - distY
    };
  };
  BubbleShoot.ui.fireBubble(curBubble,coords,duration);
  curBubble = getNextBubble();
};
```

Jeżeli odległość, jaką przemierza bąbelek, zmieniła się z powodu kolizji, czas potrzebny na dotarcie do punktu końcowego również powinien zostać zmieniony, aby wszystkie bąbelki były wystrzeliwane z tą samą prędkością. Do obliczenia zmiany prędkości wykorzystamy dane z kolizji ❶.

Przeładuj grę i wystrzel bąbelka. Bąbelek powinien się zatrzymać po trafieniu w główną grupę. Jednak cała animacja nie wygląda jeszcze dobrze. Bąbelek zatrzymuje się, ale nie integruje się z planszą gry. Przykleja się po prostu do miejsca, w które trafił. Jeżeli wystrzelisz więcej bąbelków, zauważysz również, że się na siebie nakładają — nowe bąbelki nie powodują kolizji z wystrzelonymi wcześniej. Problemem jest to, że stan gry nie zmienia się wraz ze stanem wyświetlanym — poprawimy ten błąd w dwóch krokach:

1. Dodamy wystrzelony bąbelek do stanu gry w odpowiednim rzędzie i kolumnie.
2. Kiedy wystrzelony bąbelek się zatrzyma, umieścimy go na odpowiedniej pozycji w siatce.
   W drugim kroku wykorzystamy informacje z pierwszego.

## Dodanie obiektu bąbelka do planszy

Obiekt `curBubble` klasy `bubble` jest częścią DOM i powinien znaleźć się blisko poprawnej pozycji na ekranie, dlatego kiedy już będziemy wiedzieć gdzie, możemy dodać go do tablicy rzędów i kolumn.

Aby obliczyć numer rzędu, dzielimy koordynatę *y* przez wysokość rzędów, a wynik zaokrąglamy. Obliczanie numeru kolumny wygląda podobnie, z tym że musimy umieścić bąbelek w kolumnie o numerze nieparzystym dla parzystego rzędu (włączenie z zerem) lub w kolumnie parzystej dla nieparzystego rzędu. Możemy wreszcie dodać bąbelek do właściwości `rows` obiektu `Board`, ponieważ to tam przechowujemy informacje o pozycjach wszystkich bąbelków.

Funkcja dodająca wystrzeliwane bąbelki jest trywialna, więc umieścimy ją w pliku *board.js*. W ramach definicji klasy planszy dodaj, po metodzie `getRows`, poniższy kod:

*board.js*

```
var BubbleShoot = window.BubbleShoot || {};
BubbleShoot.Board = (function($){
  var NUM_ROWS = 9;
  var NUM_COLS = 32;
  var Board = function(){
    var that = this;
    var rows = createLayout();
    this.getRows = function(){ return rows;};
    this.addBubble = function(bubble,coords){
      var rowNum = Math.floor(coords.y / BubbleShoot.ui.ROW_HEIGHT);
      var colNum = coords.x / BubbleShoot.ui.BUBBLE_DIMS * 2;
      if(rowNum % 2 == 1)
        colNum -= 1;
      colNum = Math.round(colNum/2) * 2;
      if(rowNum % 2 == 0)
        colNum -= 1;
```

```
      if(!rows[rowNum])
        rows[rowNum] = [];
      rows[rowNum][colNum] = bubble; ❶
      bubble.setRow(rowNum); ❷
      bubble.setCol(colNum); ❸
    };
    return this;
  };
  var createLayout = function(){
    --cięcie--
  };
  return Board;
})(jQuery);
```

Zauważ, że wraz z dodaniem bąbelka na odpowiedniej pozycji w rzędach i kolumnach w `rows[][]` ❶ do obiektu `bubble` przekazujemy obliczony rząd ❷ i kolumnę ❸, tak aby znał swoją relatywną pozycję na planszy. Nie mamy jeszcze metod, które są wywoływane, stwórzmy je zatem w klasie `Bubble` w pliku *bubble.js*:

*bubble.js*

```
var Bubble = function(row,col,type,sprite){
  var that = this;
  this.getType = function(){ return type;};
  this.getSprite = function(){ return sprite;};
  this.getCol = function(){ return col;};
  this.setCol = function(colIn){ col = colIn;};
  this.getRow = function(){ return row;};
  this.setRow = function(rowIn){ row = rowIn;};
  this.getCoords = function(){
    --cięcie--
  }
};
```

Dodaj teraz w pliku *game.js* wywołanie nowej metody `clickGameScreen`:

*game.js*

```
var clickGameScreen = function(e){
  var angle = BubbleShoot.ui.getBubbleAngle(curBubble.getSprite(),e);
  var duration = 750;
  var distance = 1000;
  var collision = BubbleShoot.CollisionDetector.findIntersection(curBubble,
  board,angle);
  if(collision){
    var coords = collision.coords;
    duration = Math.round(duration * collision.distToCollision / distance);
    board.addBubble(curBubble,coords);
  }else{
    var distX = Math.sin(angle) * distance;
    var distY = Math.cos(angle) * distance;
    var bubbleCoords = BubbleShoot.ui.getBubbleCoords(curBubble.getSprite());
    var coords = {
      x : bubbleCoords.left + distX,
      y : bubbleCoords.top - distY
    };
  };
```

```
  BubbleShoot.ui.fireBubble(curBubble,coords,duration);
  curBubble = getNextBubble();
};
```

Przeładuj grę i wystrzel kilka bąbelków. Powinny zacząć się nawarstwiać, chociaż niektóre mogą się wciąż nakładać, gdyż jeszcze nie wpasowują się dokładnie w planszę. Osiągnęliśmy postęp, ale chcemy, aby więcej bąbelków ładnie rozkładało się po wystąpieniu kolizji — jest to nasze następne zadanie.

## Ustawianie obiektu bąbelka na siatce

Kiedy wystrzelone bąbelki trafiają w bąbelki na planszy, zamiast pozostawić je w miejscu, w które trafią, powinniśmy dopasować je do układu pozostałych. Aktualny ruch działa dobrze, ale musimy dodać kolejny krok, który poprawi pozycję bąbelka, kiedy ten dotrze do punktu docelowego.

Po uruchomieniu `board.addBubble` obiekt `bubble` wie, do której kolumny i rzędu został dopasowany — dzięki temu wywołanie jego metody `getCoords` (która wylicza koordynaty na podstawie pozycji w rzędzie i kolumnie) zwróci koordynaty miejsca, w którym powinien znajdować się bąbelek, a nie miejsca, w którym się zatrzymał. Aby przemieścić go na miejsce, dodamy funkcję `complete`, która może zostać stworzona jako część funkcji jQuery `animate` i wykorzystywać informacje, które bąbelek już ma. W rezultacie możemy wystrzelić bąbelek i zapomnieć o nim — bez konieczności projektowania procesu ustawiającego bąbelki na miejscach. Funkcja `complete` jest dobrym miejscem na umieszczenie kodu, który ma być wykonany po zakończeniu animacji. Na przykład w grze wykorzystującej efekt eksplozji elementy DOM, z których składała się eksplozja, po zakończeniu animacji mogłyby być usuwane.

Właściwość `complete` jest wywoływana po zakończeniu animacji. W pliku *ui.js* zmodyfikuj `fireBubble` jak poniżej:

*ui.js*

```
fireBubble : function(bubble,coords,duration){
  bubble.getSprite().animate({
    left : coords.x - ui.BUBBLE_DIMS/2,
    top : coords.y - ui.BUBBLE_DIMS/2
  },
  {
    duration : duration,
    easing : "linear",
    complete : function(){
      if(bubble.getRow() !== null){ (1)
        bubble.getSprite().css({
          left : bubble.getCoords().left - ui.BUBBLE_DIMS/2,
          top : bubble.getCoords().top - ui.BUBBLE_DIMS/2
        });
      };
    }
  });
},
```

Po przeładowaniu wystrzeliwane bąbelki powinny dopasowywać się do siatki bąbelków na planszy. Zauważ, że do sprawdzenia, czy kolizja miała miejsce, wykorzystujemy metodę `getRow` ❶, ponieważ `getRow` powinien zwrócić `null` dla bąbelka, który nie trafi w nic i wyleci poza obszar gry.

# Podsumowanie

Teraz, kiedy wystrzeliwane bąbelki trafiają w inne na tablicy, *Bubble Shooter* zaczyna wyglądać jak gra. Wykorzystując jQuery, przemieszczamy sprite'y po ekranie, sprawiliśmy, że gra reaguje na działania gracza, i zaimplementowaliśmy podstawową logikę. Jednakże aktualnie nie ma możliwości pękania bąbelków, a bez tej funkcjonalności gra nie będzie kompletna. Logika pękania i odpowiadająca mu animacja będą tematem kolejnego rozdziału.

# Dalsze ćwiczenia

1. Każdy rząd planszy gry jest przesunięty względem poprzedniego, aby uzyskać efekt warstw. Zmień kod w `createLayout`, aby bąbelki układały się w regularną siatkę. W jaki sposób ta zmiana wpłynie na pozostałe elementy gry?
2. Teraz, kiedy już wiesz, jak za pomocą `createLayout` tworzyć różne wzory siatki, stwórz kod, który wygeneruje zupełnie nowy układ bąbelków. Na przykład mógłbyś umieszczać bąbelki tylko na co drugiej kolumnie lub nawet wymyślić bardziej kreatywny układ.
3. *Bubble Shooter* ma prostą strukturę obiektów, która składa się z `Game`, `Board` i zestawu obiektów klasy `Bubbles`. Jakich obiektów potrzebowałbyś, gdybyś budował grę podobną do *Angry Birds*, *Bejeweled* albo *Candy Crush*?

# Skorowidz

# Z

**SPRÓBUJ SWOICH SIŁ I ZREALIZUJ POMYSŁY NA WŁASNĄ GRĘ!**

*Pasjans, Bubble Shooter, Puzzle Bubble, Mahjong, Candy Crush…* Ile już godzin spędziłeś nad jedną z tych prostych i… wyjątkowo uzależniających gier? Żadna z nich nie wymaga nośników instalacyjnych ani potężnych mocy obliczeniowych, te gry działają na różnych platformach i urządzeniach, niezależnie od systemu operacyjnego, a do ich uruchomienia wystarczy jedynie przeglądarka. Zbudowanie takiej gry jest proste — wystarczy sprawny duet HTML5 i JavaScript.

Jeśli masz podstawową wiedzę o HTML5, CSS i JavaScript, to dzięki tej książce możesz nauczyć się pisania gier działających w przeglądarce. Autor książki, Karl Bunyan, pokazuje, jak zaplanować logikę gry, jak korzystać z bibliotek jQuery i Modernizr, jak renderować elementy gry i tworzyć płynne animacje. Uczy korzystania z efektownych przejść i transformacji CSS, a także sposobu implementowania efektów dźwiękowych i zapisywania wyników uzyskanych przez gracza. Ten kompletny przewodnik w każdym rozdziale przedstawia nowe koncepcje i techniki, od razu prezentując ich działanie w praktyce. Dzięki temu czytelnik płynnie przechodzi od zagadnień podstawowych (tworzenie struktury plików gry czy reagowanie na zachowanie gracza), przez bardziej zaawansowane (wprowadzanie poziomów i wykorzystanie kanwy), do tak istotnych spraw, jak zarządzanie pamięcią i optymalizacja szybkości działania aplikacji.

**Karl Bunyan** — swoją pierwszą grę przygodową opublikował w 1990 r. dla ZX Spectrum i od tamtej pory zajmuje się tworzeniem gier. Tworzył prototypy HTML5 dla Game Show Network, a obecnie jest właścicielem Wedu Games, niezależnej firmy budującej gry sieciowe i mobilne.


Helion

38568 numer katalogowy

księgarnia internetowa

http://helion.pl

zamówienia telefoniczne

0 801 339900

0 601 339900

Sprawdź najnowsze promocje:
http://helion.pl/promocje
Książki najchętniej czytane:
http://helion.pl/bestsellery
Zamów informacje o nowościach:
http://helion.pl/nowosci

Helion SA
ul. Kościuszki 1c, 44-100 Gliwice
tel.: 32 230 98 63
e-mail: helion@helion.pl
http://helion.pl

Informatyka w najlepszym wydaniu

ISBN 978-83-283-1770-3

cena: 39,90 zł