O'REILLY®

CSS BEZ TAJEMNIC

47 SEKRETÓW KREATYWNEGO PROJEKTANTA

LEA VEROU

Helion

Tytuł oryginału: CSS Secrets

Tłumaczenie: Piotr Cieślak (wstęp, rozdz. 1 – 6, 8); Karolina Waśko (rozdz. 7)

ISBN: 978-83-283-1716-1

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/scss47
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

Spis treści

Efekty wizualne 4

15 Jednostronne cienie

Problem

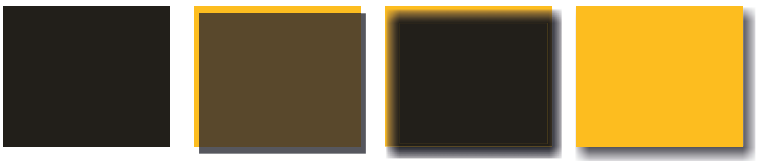
Jedno z najczęściej zadawanych pytań odnoszących się do atrybutu `box-shadow`, jakie widuję na stronach z poradami, dotyczy tego, jak można uzyskać cień padający tylko z jednej strony (albo, rzadziej, z dwóch stron) określonego obiektu. Szybkie wyszukiwanie w serwisie *stackoverflow.com* zwraca niemal tysiąc rezultatów związanych z tym zagadnieniem. I ma to sens, ponieważ wyświetlenie cienia tylko z jednej strony elementu daje subtelniejszy, a zarazem realistyczny efekt. Sfrustrowani programiści piszą nawet na listę dyskusyjną CSS Working Group, prosząc o nowe właściwości, takie jak `box-shadow-bottom`, które pozwalałyby uzyskać tego rodzaju efekt. Okazuje się jednak, że jest to wykonalne już dziś dzięki sprytnemu zastosowaniu starej, dobrej, znanej i lubianej przez nas właściwości `box-shadow`.

Cień z jednej strony

Większość projektantów, używając właściwości `box-shadow`, podaje cztery argumenty — trzy odległości oraz kolor, na przykład:

```
box-shadow: 2px 3px 4px rgba(0,0,0,.5);
```

Rysunek 4.1 ilustruje sekwencję kroków (choć niezupełnie technicznie poprawną), ułatwiającą wyobrażenie sobie metody rysowania takiego cienia przez przeglądarkę:



RYSUNEK 4.1. Ilustrowanie tworzenia cieni przy użyciu atrybutu box-shadow

1. Najpierw jest kreślony prostokąt w kolorze `rgba(0,0,0,.5)` o wymiarach i położeniu zgodnych z parametrami elementu bazowego.
2. Prostokąt jest przesuwany o `2px` w prawą stronę i o `3px` w dół.
3. Prostokąt zostaje poddany rozmyciu (przy użyciu algorytmu Gaussa albo podobnego) o promieniu `4px`. Tak naprawdę oznacza to, że przejście na krawędziach między nieprzezroczystą częścią cienia a całkowitą przezroczystością będzie w przybliżeniu dwukrotnie dłuższe od promienia rozmycia (czyli `8px` w naszym przykładzie).
4. Rozmyty prostokąt jest następnie przycinany zgodnie z kształtem jego części wspólnej z oryginalnym elementem, a w rezultacie wydaje się, że znajduje się pod nim. Większość programistów wyobraża sobie taki cień trochę inaczej — jako cały, rozmyty prostokąt pod elementem. W niektórych sytuacjach warto uzmysłowić sobie to, że pod elementem tak naprawdę nie ma żadnego cienia. Na przykład jeśli zmienimy tło elementu na półprzezroczyste, to nie zobaczymy cienia pod spodem. To inaczej niż w przypadku atrybutu `text-shadow`, generującego cień, który nie jest przycinany pod znakami.

Jeśli nie zostało powiedziane inaczej, wymiary elementu oznaczają tutaj wymiary *pudełka ramki*, czyli *border box*, a *nie* szerokość i wysokość tego elementu w CSS.

Zastosowanie promienia rozmycia o wartości `4px` oznacza, że wymiary cienia są w przybliżeniu o `4px` większe niż wymiary elementu, pewna część cienia będzie więc widoczna z każdej strony tego elementu. Aby ukryć cień z lewej strony i od góry, moglibyśmy zmienić wartości przesunięć — zwiększyć je do co najmniej `4px`. Tak otrzymany cień jednak bardzo rzucałby się w oczy i nie wyglądałby najlepiej (rysunek 4.2). Poza tym nawet jeśli taki wygląd nie stanowiłby problemu, to chcieliśmy uzyskać efekt cienia tylko z jednej strony elementu, a nie z dwóch, pamiętasz?

Gwoli ścisłości — u góry zobaczymy cień o wielkości `1px` (`4px – 3px`), po lewej stronie — o wielkości `2px` (`4px – 2px`), po prawej — `6px` (`4px + 2px`), na dole zaś — `7px` (`4px + 3px`). W praktyce cień będzie się jednak wydawał mniejszy, bo przejście od koloru do przezroczystości na krawędziach nie ma charakteru liniowego jak w przypadku gradientów.

Rozwiązanie kryje się w rzadko stosowanym czwartym parametrze opisującym odległość, podawanym po promieniu rozmycia i zwanym promieniem rozproszenia (ang. *spread radius*). Promień ten zwiększa albo zmniejsza (w przypadku wartości ujemnej) wielkość cienia o podaną wartość. Na przykład promień `-5px` spowoduje zmniejszenie szerokości i wysokości cienia o `10px` (po `5px` z każdej strony).

RYSUNEK 4.2.
Próba ukrycia cienia po lewej stronie i od góry poprzez przesunięcie go na odległość równą promieniowi rozmycia

Logiczne jest więc, że jeśli zastosujemy promień ujemny, którego wartość bezwzględna jest równa promieniowi rozmycia, to cień będzie miał takie same wymiary jak element, na podstawie którego powstał. To oznacza, że jeśli nie przesuniemy cienia za pomocą parametrów decydujących o jego położeniu (dwa pierwsze argumenty), to nie zobaczymy go w ogóle. Stąd zaś już tylko krok do stwierdzenia, że jeśli przesunięciu w pionie nadamy wartość dodatnią, cień pojawi się pod dolną krawędzią elementu, ale nie będzie widoczny z żadnej z pozostałych stron, a dokładnie na tym nam zależy:

```
box-shadow: 0 5px 4px -4px black;
```

RYSUNEK 4.3.
Cień typu box-shadow widoczny tylko od dołu

▸ **WYPRÓBUJ** *play.csssecrets.io/shadow-one-side*

Cienie z dwóch sąsiadujących stron

Kolejne często zadawane pytanie dotyczące cieni ma związek z ich tworzeniem z dwóch stron elementu. Jeśli strony te sąsiadują ze sobą (na przykład prawa i dolna krawędź), to zadanie jest łatwiejsze: możesz zdecydować się na efekt taki jak na rysunku 4.2 albo zastosować prostą odmianę triku opisanego w poprzednim przykładzie, z uwzględnieniem dwóch poniższych różnic:

- Tym razem nie chcemy zmniejszać cienia tak, by wziąć pod uwagę rozmycie z obu stron, ale chcemy to zrobić tylko z jednej strony. To oznacza, że nie trzeba nadawać promieniowi rozproszenia wartości przeciwnej do wartości promienia rozmycia; wystarczy połowa tej wartości.

Musimy użyć obu argumentów decydujących o położeniu cienia, ponieważ chcemy go przesunąć zarówno w poziomie, jak i w pionie. Ich wartość powinna być większa niż wartość promienia rozmycia lub równa jej połowie, ponieważ w ten sposób nie będzie widać cienia z dwóch pozostałych stron elementu.

RYSUNEK 4.4.
Cień typu box-shadow widoczny z dwóch sąsiednich stron

Na przykład tak możemy utworzyć cień w kolorze ■ `black` o promieniu rozmycia `6px`, który to cień będzie widoczny po prawej stronie i na dole:

```
box-shadow: 3px 3px 6px -3px black;
```

Rezultat pokazano na rysunku 4.4.

▸ **WYPRÓBUJ** *play.csssecrets.io/shadow-2-sides*

Cień z dwóch przeciwległych stron

Kłopoty powstają z chwilą, gdy zaczyna nam zależeć na cieniu z dwóch przeciwległych stron elementu, na przykład z lewej i prawej strony. Ponieważ promień rozproszenia ma ze wszystkich stron takie samo działanie (to oznacza, że nie ma sposobu na takie skonfigurowanie argumentów, by powiększyć cień w poziomie i zmniejszyć w pionie), jedyną możliwością uzyskania takiego efektu jest zastosowanie dwóch cieni, po jednym z każdej strony. Potem zaś możemy zastosować trik omówiony w przykładzie „Cień z jednej strony", z tym że dwa razy:

Na łonie CSS WG był już omawiany pomysł wprowadzenia osobnych promieni rozpraszania dla kierunków poziomego i pionowego, co uprościłoby tego rodzaju operacje.

```
box-shadow: 5px 0 5px -5px black,
            -5px 0 5px -5px black;
```

Efekt został przedstawiony na rysunku 4.5.

RYSUNEK 4.5. Cień typu box-shadow widoczny po dwóch przeciwległych stronach

▸ **WYPRÓBUJ** *play.csssecrets.io/shadow-opposite-sides*

POWIĄZANE SPECYFIKACJE

- **CSS Backgrounds & Borders**
  w3.org/TR/css-backgrounds

16 Nieregularne cienie

Wymagania

Atrybut `box-shadow`.

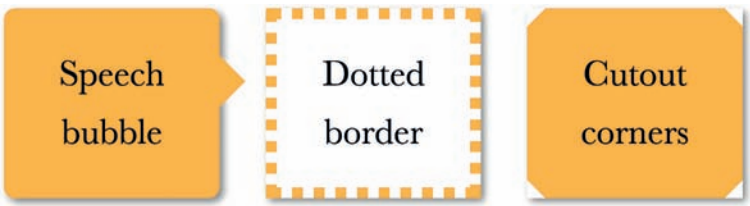
Problem

Atrybut `box-shadow` sprawdza się znakomicie, gdy chcemy uzyskać efekt cienia rzucanego przez prostokąt albo przez inny kształt — możliwy do uzyskania za pomocą właściwości `border-radius` (kilka przykładów znajdziesz w sekrecie 9. „Elastyczne elipsy"). Przydatność `box-shadow` staje się jednak wątpliwa, gdy mamy do czynienia z pseudoelementami albo innego rodzaju półprzezroczystymi ozdobnikami, ponieważ atrybut ten „bezczelnie" ignoruje przezroczystość. Oto kilka przykładów obiektów, które mogą sprawiać kłopoty:

- półprzezroczyste obrazy, obrazy w tle albo obramowania uzyskane za pomocą `border-image` (na przykład grafika naśladująca staromodną złotą ramę obrazu);
- kropkowane, kreskowane lub półprzezroczyste ramki bez tła (albo elementy, w których wartość atrybutu `background-clip` została zmieniona na inną niż `border-box`);
- dymki dialogowe ze strzałką utworzoną za pomocą pseudoelementu;
- ścięte narożniki, takie jak omówione w sekrecie 12. „Przycięte narożniki";

- większość efektów zawiniętych rogów, w tym jeden opisany w dalszej części tego rozdziału;
- kształty utworzone przy użyciu ścieżek `clip-path`, takie jak romboidalne obrazy opisane w sekrecie 11. „Obrazy w kształcie rombu”.

Bezowocne próby dodania cieni do niektórych z wymienionych obiektów za pomocą atrybutu `box-shadow` zostały pokazane na rysunku 4.6. Czy da się w takich przypadkach coś zrobić, czy też po prostu musimy pożegnać się z cieniami w ogóle?



RYSUNEK 4.6. Elementy ze stylami CSS, przy których atrybut box-shadow staje się bezużyteczny. Wartość zastosowanych tutaj parametrów box-shadow to 2px 2px 10px rgba(0,0,0,.5)

Rozwiązanie

OGRANICZONA OBSŁUGA

Specyfikacja **Filter Effects** (*w3.org/TR/filter-effects*) przewiduje rozwiązanie tego problemu za pośrednictwem nowej właściwości `filter`, zapożyczonej z SVG. Warto wiedzieć, że choć filtry CSS są tak naprawdę filtrami SVG, nie wymagają żadnej wiedzy na temat SVG. Ich użytkowanie zostało sprowadzone do kilku wygodnych funkcji, takich jak `blur()`, `grayscale()` oraz… nie zgadniesz: `drop-shadow()`! Jeśli chcesz, możesz nawet utworzyć sekwencję takich filtrów, rozdzielając nazwy funkcji spacjami, na przykład tak:

```
filter: blur() grayscale() drop-shadow();
```

Filtr `drop-shadow()` przyjmuje te same argumenty co atrybut `box-shadow`, ale nie zmusza do ekwilibrystyki z promieniem rozpraszania, słowem kluczowym `inset` czy kilkoma osobnymi definicjami cieni. Na przykład zamiast:

```
box-shadow: 2px 2px 10px rgba(0,0,0,.5);
```

…wystarczy napisać:

```
filter: drop-shadow(2px 2px 10px rgba(0,0,0,.5));
```

Efekt zastosowania filtra `drop-shadow()` względem elementów z rysunku 4.6 ilustruje rysunek 4.7.

RYSUNEK 4.7. Filtr drop-shadow() użyty dla elementów z rysunku 4.6

Omawiana funkcja może używać różnych algorytmów rozmywania, które będą wymagały dobrania wartości rozmycia innych niż podane!

Największą zaletą filtrów CSS jest ich elegancka degradacja w sytuacji awaryjnej: jeśli nie są obsługiwane, nic się nie „rozsypie" — po prostu efekt nie zostanie wyświetlony. Jeśli oprócz omawianej techniki zastosujesz prawdziwy filtr SVG, zyskasz trochę pełniejsze wsparcie w przeglądarkach, co może się przydać w sytuacji, gdy bezwzględnie będzie Ci zależało na obsłudze jak największej liczby przeglądarek. Filtry SVG odpowiadające poszczególnym funkcjom znajdziesz w specyfikacji **Filter Effects** (*w3.org/TR/filter-effects*). Istnieje możliwość zawarcia w kodzie zarówno filtra SVG, jak i jego uproszczonego odpowiednika z CSS; o tym, który wariant zostanie uwzględniony, zadecyduje kaskada:

```
filter: url(drop-shadow.svg#drop-shadow);
filter: drop-shadow(2px 2px 10px rgba(0,0,0,.5));
```

Niestety filtry SVG znajdujące się w osobnych plikach nie są tak łatwe w konfiguracji jak przyjazne, wygodne funkcje dostępne wprost z poziomu kodu CSS, ich użycie zaś w wersji *inline* powoduje pogorszenie czytelności kodu. Parametry działania filtra są na stałe zdefiniowane w pliku, a zwiększanie liczby plików z filtrami w przypadku, gdy zależałoby nam na kilku różnych wariantach cienia, nie jest praktyczne. Moglibyśmy wprawdzie posłużyć się odsyłaczami URI (co przy okazji pozwoliłoby zaoszczędzić dodatkowe żądanie HTTP), ale filtry mimo wszystko przyczyniałyby się do wzrostu objętości strony. Ponieważ jednak traktujemy to jako rozwiązanie awaryjne, możemy pokusić się o opracowanie jednego albo dwóch wariantów filtra `drop-shadow()`, nawet jeśli ich parametry będą się różnić tylko trochę.

RYSUNEK 4.8. Cień uzyskany za pomocą text-shadow rzuca własny cień dzięki filtrowi drop-shadow()

Kolejna kwestia, o której warto pamiętać — w opisywanej sytuacji cień będzie rzucany przez każdy nieprzezroczysty obszar i obiekt, w tym tekst (jeśli tło jest przezroczyste), o czym mogłeś się przekonać na rysunku 4.7. Być może sądzisz, że da się temu zapobiec za pomocą deklaracji `text-shadow: none;`, ale niestety: właściwość `text-shadow` jest zupełnie oddzielnym narzędziem i nie spowoduje

wyłączenia filtra `drop-shadow()` dla tekstu. Co więcej, jeśli użyłeś atrybutu `text-shadow` do utworzenia cienia tekstu, to ten cień także będzie podlegał działaniu funkcji `drop-shadow()`, co doprowadzi do wygenerowania cienia rzucanego przez cień! Weźmy na przykład poniższy kod CSS (wybacz kiczowaty efekt; chodziło mi o zademonstrowanie dziwaczności tego problemu):

```
color: deeppink;
border: 2px solid;
text-shadow: .1em .2em yellow;
filter: drop-shadow(.05em .05em .1em gray);
```

Przykład działania tego kodu został zilustrowany na rysunku 4.8, na którym widać oba cienie: ten rzucony przez atrybut `text-shadow` oraz cień tego cienia, uzyskany dzięki filtrowi `drop-shadow()`.

▸ **WYPRÓBUJ** *play.csssecrets.io/drop-shadow*

POWIĄZANE SPECYFIKACJE

- **Filter Effects**
  w3.org/TR/filter-effects

17 Barwienie obrazów

Wymagania

Model barw HSL, atrybut `background-size`.

Problem

Zabarwienie obszaru w skali szarości (albo zdjęcia przekształconego w skalę szarości) konkretnym kolorem jest często stosowanym i eleganckim sposobem na wizualne ujednolicenie grupy zdjęć o bardzo zróżnicowanych stylach. Często efekt ten jest dodawany domyślnie i wyłączany w przypadku zajścia zdarzenia `:hover` lub innego rodzaju interakcji (rysunek 4.9).

Tradycyjnie za pomocą programu do edycji grafiki przygotowywało się dwa warianty zdjęcia, a potem pisało się prosty kod CSS odpowiadający za ich podmienianie. Ta metoda działa, ale zwiększa objętość kodu oraz liczbę żądań HTTP, a przy okazji jest koszmarnie uciążliwa w utrzymaniu. Wyobraź sobie, że chciałbyś zmienić kolor uzyskanego efektu: musiałbyś otworzyć wszystkie źródłowe zdjęcia i utworzyć ich nowe, monochromatyczne warianty!

RYSUNEK 4.9. Ten trik został wykorzystany do pokazania zdjęć mówców na stronie konferencji CSSConf 2014, z tym że po wskazaniu i kliknięciu danego zdjęcia wyświetlał się jego kolorowy wariant

Inne rozwiązania polegają na nałożeniu półprzezroczystego koloru na zdjęcie lub zmianie stopnia przejrzystości samego zdjęcia, wyświetlonego na jednolitym tle. Te metody tak naprawdę jednak nie barwią zdjęcia: nie tylko nie przekształcają wszystkich barw oryginalnego obrazu w odcienie żądanego koloru, ale też znacząco zmniejszają kontrast.

Istnieją skrypty przekształcające zdjęcie na element obiektu `<canvas>` i tam poddające je barwieniu za pomocą JavaScriptu. To rozwiązanie daje prawidłową tintę, ale jest stosunkowo wolne i ma ograniczone możliwości.

Czy nie byłoby łatwiej zabarwić zdjęcie wprost przy użyciu CSS?

Rozwiązanie z użyciem filtrów

OGRANICZONA OBSŁUGA

Ponieważ żaden z filtrów nie daje dokładnie takiego efektu, na jakim nam zależy, musimy wykazać się pewną pomysłowością i połączyć ich kilka.

Pierwszym filtrem, jaki zastosujemy, będzie `sepia()`, która nadaje zdjęciom blade, słomkowe zabarwienie. Parametr `hue` większości pikseli otrzymanego w ten sposób obrazu wynosi około 35 – 40 (rysunek 4.10). Jeśli takie zabarwienie okaże się wystarczające, to problem z głowy, ale w większości przypadków tak nie jest. Jeśli chcemy uzyskać bardziej soczysty kolor, możemy użyć filtra `saturate()` do zwiększenia nasycenia barw poszczególnych pikseli. Załóżmy, że zamierzamy nadać zdjęciu odcień [140-01] `hsl(335, 100%, 50%)`. W tej sytuacji powinniśmy dość mocno zwiększyć nasycenie barw, użyjemy więc argumentu o wartości 4. Dobór wartości jest uzależniony od konkretnego przypadku i na

RYSUNEK 4.10

U góry: oryginalne zdjęcie

Na dole: po zastosowaniu filtra sepia()

ogół należy ją określić „na oko". Jak widać na rysunku 4.11, wskutek zastosowania obu wymienionych filtrów nasze zdjęcie nabrało złotego, ciepłego odcienia.

Choć zdjęcie wygląda teraz bardzo ciekawie, to nie chcieliśmy przecież barwić go na bursztynowy kolor, ale na głęboki jaskrawy róż. Z tego względu musimy dodatkowo zastosować filtr `hue-rotate()`, zmieniający barwę każdego piksela o podany kąt (na kole barw). Aby uzyskać barwę znajdującą się pod kątem 335° ze źródłowej, której odpowiada kąt mniej więcej 40°, musimy dodać do niej 295° (335 – 40), czyli:

```
filter: sepia() saturate(4) hue-rotate(295deg);
```

W ten sposób rzeczywiście udało się nam nadać zdjęciu oczekiwaną kolorystykę. Jego aktualny wygląd ilustruje rysunek 4.12. Jeśli efekt ten miałby być włączany po zajściu zdarzenia `:hover` lub wskutek innego rodzaju zmiany stanu elementu, można byłoby dodać do niego odpowiednie przejścia CSS:

```
img {
  transition: .5s filter;
  filter: sepia() saturate(4) hue-rotate(295deg);
}
img:hover,
img:focus {
  filter: none;
}
```

RYSUNEK 4.11.
Zdjęcie w wyniku zastosowania filtra saturate()

RYSUNEK 4.12.
Zdjęcie po zastosowaniu kolejnego filtra, hue-rotate()

▸ **WYPRÓBUJ** *play.csssecrets.io/color-tint-filter*

Rozwiązanie z trybami mieszania

OGRANICZONA OBSŁUGA

Metoda z filtrami działa, ale być może zwróciłeś uwagę na to, że jej zastosowanie daje trochę inne rezultaty niż można uzyskać przy użyciu programu graficznego. Pomimo że wybraliśmy bardzo jaskrawy kolor barwiący, efekt jest raczej mdły. Jeśli z kolei spróbowalibyśmy temu zaradzić przez zwiększenie wartości argumentu filtra `saturate()`, to otrzymalibyśmy cukierkowy, przesadzony efekt. Na szczęście jest lepszy sposób, by zrealizować nasz cel: tryby mieszania!

Jeśli kiedykolwiek posługiwałeś się programem graficznym takim jak Adobe Photoshop, to tryby mieszania zapewne nie są Ci obce. Pokrótce: gdy dwa elementy nakładają się, tryby mieszania decydują o tym, jak będą się łączyły kolory górnego elementu z kolorami tego, który znajduje się pod spodem. Jeśli chodzi o barwienie zdjęć, najbardziej przydatnym trybem jest `luminosity`. Tryb `luminosity` zachowuje jasność (składowa L w modelu HSL) górnego elementu i przejmuje barwę oraz nasycenie tła. Wobec tego, jeśli temu tłu nadamy docelowy kolor barwiący i nałożymy na nie zdjęcie z użyciem wspomnianego trybu mieszania, to czy nie powinniśmy w ten sposób otrzymać dokładnie takiego zabarwienia, na jakim nam zależało (rysunek 4.13)?

RYSUNEK 4.13. Porównanie metody z filtrami (u góry) z metodą z trybami mieszania (na dole)

Zmianę trybów mieszania wybranych elementów umożliwiają dwie właściwości: `mix-blend-mode` (służąca do zmieniania trybów mieszania całych elementów) oraz `background-blend-mode` (umożliwiająca stosowanie trybów mieszania do każdej warstwy tła z osobna). To oznacza, że przy barwieniu zdjęć tą metodą mamy dwie możliwości (niestety każda z nich ma swoje wady):

- umieścić zdjęcie w kontenerze wyposażonym w tło w żądanym kolorze;
- zamiast zwyczajnie wstawić zdjęcie, zastosować element `<div>` z właściwością `background-image` (odwołującą się do zdjęcia przeznaczonego do zabarwienia) oraz drugim tłem, pod spodem, w żądanym kolorze.

Wybór jednej z tych metod jest podyktowany wymogami konkretnego projektu. Na przykład jeśli chcielibyśmy zastosować opisywany efekt względem elementu `<img>`, to musielibyśmy umieścić go w kolejnym elemencie. Ale jeżeli taki element już istnieje, może nim być na przykład element `<a>`, to możemy posłużyć się następującym kodem:

HTML

```
<a href="#something">
  <img src="tiger.jpg" alt="Rawrrr!" />
</a>
```

Potem zaś do zastosowania omawianego efektu wystarczą już tylko dwie deklaracje:

```
a {
  background: hsl(335, 100%, 50%);
}

img {
  mix-blend-mode: luminosity;
}
```

Tak jak w przypadku filtrów CSS, brak obsługi trybów mieszania nie wiąże się z poważnymi skutkami ubocznymi; efekt po prostu nie zostanie wyświetlony, ale samo zdjęcie będzie widoczne.

Warto też pamiętać, że o ile filtry można animować, to trybów mieszania animować się nie da. Zobaczyłeś już, jak wykonać płynną animację od kolorowego zdjęcia do monochromatycznego obrazu za pomocą prostego przejścia CSS z użyciem właściwości `filter`. Takiego triku nie da się wykonać z trybami mieszania. Nie martw się jednak, nie oznacza to, że zastosowanie animacji w ogóle nie wchodzi w grę — trzeba jednak pomyśleć trochę nieszablonowo.

Jak już wspomniałam, właściwość `mix-blend-mode` powoduje „zmieszanie" całego elementu z tym, co znajduje się pod spodem. To oznacza, że jeśli za pośrednictwem tej właściwości nadamy obrazowi tryb mieszania `luminosity`, ów obraz zawsze *z czymś* się połączy. Z kolei właściwość `background-blend-mode` miesza wszystkie warstwy tła elementu zawierające obrazy z warstwami znajdującymi się pod nimi, ale bez uwzględniania czegokolwiek, co nie należy do tego elementu. W takim razie, co się stanie, jeśli w tle elementu będzie tylko jedna warstwa z obrazem i jedna warstwa z kolorem przezroczystym? Zgadłeś: nie dojdzie do żadnego mieszania barw!

Możemy wykorzystać to spostrzeżenie i użyć właściwości `background-blend-mode` do naszych celów. Przy okazji będziemy musieli odrobinę zmodyfikować kod HTML:

HTML

```
<div class="tinted-image"
     style="background-image:url(tiger.jpg)">
</div>
```

Następnie wystarczy napisać kod CSS dla jednego elementu `<div>`, bo opisywana metoda nie wymaga dodatkowych elementów:

```
.tinted-image {
  width: 640px; height: 440px;
  background-size: cover;
  background-color: hsl(335, 100%, 50%);
  background-blend-mode: luminosity;
  transition: .5s background-color;
}
.tinted-image:hover {
  background-color: transparent;
}
```

Jak już jednak wspomniałam, żadna z tych dwóch metod nie jest idealna. Główne problemy, z jakimi mamy tutaj do czynienia, są takie:

- konieczność sztywnego określenia wymiarów obrazu w kodzie CSS;
- to, że pod względem składniowym nie mamy do czynienia z obrazem i nie zostanie on potraktowany jak obraz przez czytniki ekranowe.

Jak to w życiu bywa, żadne wyjście nie jest idealne, ale miałeś przynajmniej okazję zapoznać się z trzema różnymi metodami uzyskiwania omawianego efektu, a każda z nich miała pewne zalety i wady. Wybór jednej z tych metod jest uzależniony od specyfiki konkretnego projektu.

▸ WYPRÓBUJ *play.csssecrets.io/color-tint*

WYRAZY UZNANIA

Wyrazy uznania dla Dudleya Storeya (*demosthenes.info*), który wpadł na pomysł umożliwiający animowanie trybów mieszania (*demosthenes.info/blog/888/Create-Monochromatic-Color-Tinted-Images-With-CSS-blend*).

POWIĄZANE SPECYFIKACJE

- **Filter Effects**
  w3.org/TR/filter-effects
- **Compositing and Blending**
  w3.org/TR/compositing
- **CSS Transitions**
  w3.org/TR/css-transitions

18 Efekt matowego szkła

Wymagania

Modele barw RGBA/HSLA.

Problem

Użyłam tutaj wyrażenia „drugi plan” do określenia tej części strony, która znajduje się pod spodem jakiegoś elementu, ale jest widoczna dzięki przezroczystości tego elementu.

Jedno z pierwszych praktycznych zastosowań przezroczystych kolorów w CSS polegało na nakładaniu ich na zdjęcia albo innego rodzaju skomplikowany wizualnie „drugi plan” w celu zmniejszenia kontrastu tła i poprawienia czytelności tekstu znajdującego się na wierzchu. Sztuczki tego typu robiły wrażenie, ale pomimo ich zastosowania tekst czasami nadal trudno było odczytać, zwłaszcza przy kolorach o dużej przejrzystości i (lub) bardzo skomplikowanych tłach. Przyjrzyj się na przykład rysunkowi 4.14, gdzie główny kontener na tekst ma półprzezroczyste białe tło. Kod tego przykładu ma taką postać:

HTML

```
<main>
  <blockquote>
    "The only way to get rid of a temptation[…]"
```

```
    <footer>—
      <cite>
        Oscar Wilde,
        The Picture of Dorian Gray
      </cite>
    </footer>
  </blockquote>
</main>
```

CSS przedstawia się natomiast tak (dla ułatwienia wszystkie nieistotne fragmenty zostały pominięte):

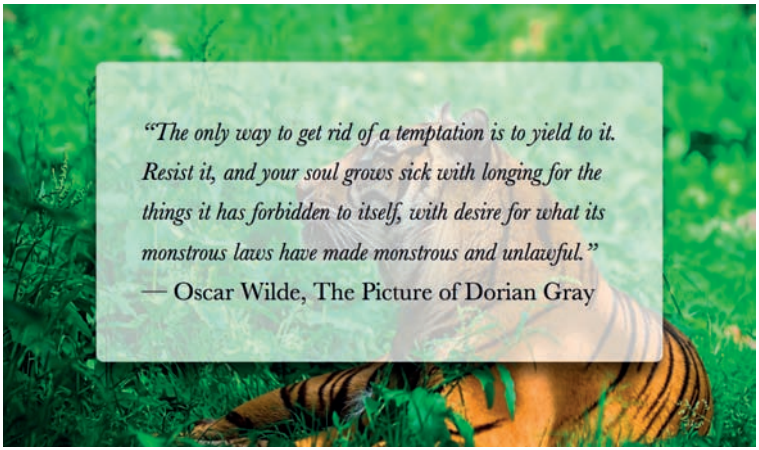
```
body {
  background: url("tiger.jpg") 0 / cover fixed;
}
main {
  background: hsla(0,0%,100%,.3);
}
```

RYSUNEK 4.14. Pomimo przezroczystego białego tła tekst nadal jest mało czytelny

Jak widać, ze względu na to, że na zdjęciu na drugim planie sporo się dzieje, a stopień krycia (alfa) białego tła wynosi tylko 30%, tekst jest niemal nieczytelny. Oczywiście moglibyśmy bez trudu poprawić czytelność tekstu

poprzez zwiększenie wartości parametru alfa dla koloru tła, ale wtedy całość prezentowałaby się znacznie mniej interesująco (zob. rysunek 4.15).

RYSUNEK 4.15.
Zwiększenie wartości alfa dla białego tła poprawia czytelność tekstu, ale zarazem pogarsza wizualną atrakcyjność projektu



W druku problem ten jest często rozwiązywany przez rozmycie tych fragmentów zdjęcia, które znajdują się bezpośrednio pod ramką z tekstem. Rozmyte tła nie są tak „gęste" pod względem wizualnym, dzięki czemu umieszczony na nich tekst jest czytelniejszy. Rozmywanie obrazu wymaga jednak dość dużej mocy obliczeniowej, przez co stosowanie tego rodzaju trików na stronach WWW oraz w interfejsach aplikacji do niedawna nie wchodziło w rachubę. Jednak w miarę rozwoju procesorów GPU i coraz powszechniejszej sprzętowej akceleracji grafiki sztuczek z rozmyciem obrazu używa się coraz częściej. W ciągu ostatnich kilku lat pojawiły się one w najnowszych wersjach systemów Microsoft Windows, Apple iOS oraz Mac OS X (rysunek 4.16).

RYSUNEK 4.16.
W ciągu kilku ostatnich lat popularność przejrzystych elementów UI oraz efektu rozmycia tła zdecydowanie wzrosły, w miarę jak proces rozmywania przestał stanowić poważne obciążenie dla sprzętu (na rysunku po lewej stronie został pokazany Apple iOS 8.1, po prawej zaś Apple OS X Yosemite)

Za sprawą filtra `blur()` pojawiła się też możliwość rozmywania elementów z poziomu CSS. Filtr ten jest sprzętowo przyspieszoną wersją prostego rozmycia dostępnego w SVG, które zawsze mieliśmy do dyspozycji. Jeśli jednak zastosujemy filtr `blur()` dla elementu użytego w naszym przykładzie, element ten ulegnie rozmyciu w całości wraz z tekstem, który stanie się przez to kompletnie nieczytelny (rysunek 4.17). Czy jest jakiś sposób, by poddać rozmyciu tylko drugi plan elementu (na przykład fragment tła znajdujący się dokładnie pod nim?).

RYSUNEK 4.17. Zastosowanie filtra blur() względem elementu zawierającego tekst tylko pogarsza sprawę

Rozwiązanie

Jeśli przyjąć, że pozycjonowanie naszego elementu zostało ustalone względem tła (`background-attachement`) albo strony (`fixed`), zadanie jest wykonalne, choć trochę kłopotliwe. Ponieważ nie możemy zastosować rozmycia bezpośrednio do elementu z tekstem, rozmyjemy pseudoelement umiejscowiony dokładnie za tym elementem. Tło pseudoelementu będzie idealnie pasować do tła elementu `<body>`.

Można to zrobić także w przypadku teł o elastycznym pozycjonowaniu, ale jest to jeszcze bardziej zagmatwane.

Zacznijmy od utworzenia pseudoelementu i zdefiniowania jego położenia w sposób bezwzględny z przesunięciami równymi `0`, dzięki czemu będzie on przesłaniał cały element `<main>`:

```
main {
  position: relative;
  /* [Reszta stylów] */
```

```
}
main::before {
  content: '';
  position: absolute;
  top: 0; right: 0; bottom: 0; left: 0;
  background: rgba(255,0,0,.5); /* Na potrzeby debugowania */
}
```

Uważaj podczas wykorzystywania ujemnych wartości `z-index` do przesuwania elementu potomnego pod element rodzica: jeśli ów rodzic jest zagnieżdżony w innych elementach z własnymi tłami, potomek znajdzie się także pod tymi tłami.

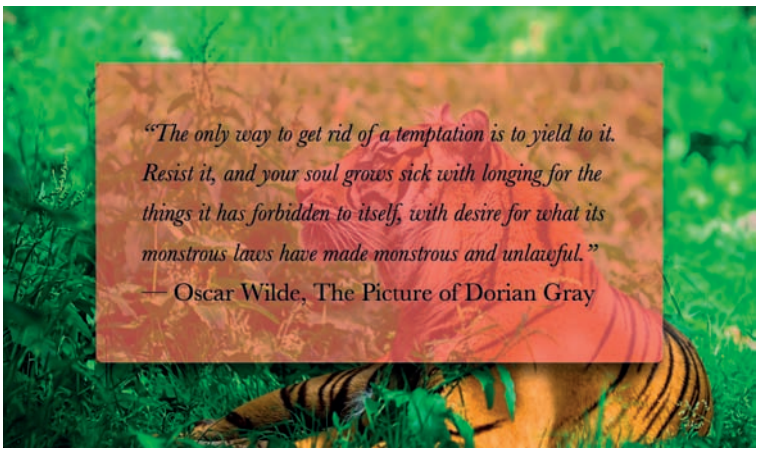
Zastosowaliśmy ponadto półprzezroczyste tło w kolorze red, aby lepiej widzieć, co robimy — w przeciwnym razie usunięcie ewentualnych błędów byłoby bardzo kłopotliwe, jako że mamy do czynienia z przezroczystym (a co za tym idzie niewidocznym) elementem. Na rysunku 4.18 pokazano, że nasz pseudoelement znajduje się na razie ponad tekstem, a tym samym go zasłania. Możemy to naprawić, dodając deklarację `z-index: -1;` (rysunek 4.19).

Teraz możemy już zastąpić półprzezroczyste czerwone tło takim, jakie rzeczywiście będzie pasowało do naszego „drugiego planu”. Możemy to zrobić albo przez skopiowanie tła elementu <body>, albo przez utworzenie osobnej reguły. Czy po tych zabiegach da się zastosować rozmycie? Spróbujmy:

Dlaczego nie użyliśmy `background:inherit` dla elementu `main::before`? Ponieważ wtedy zostałyby odziedziczone właściwości elementu `main`, a nie body, a zatem pseudoelement również otrzymałby półprzezroczyste białe tło.

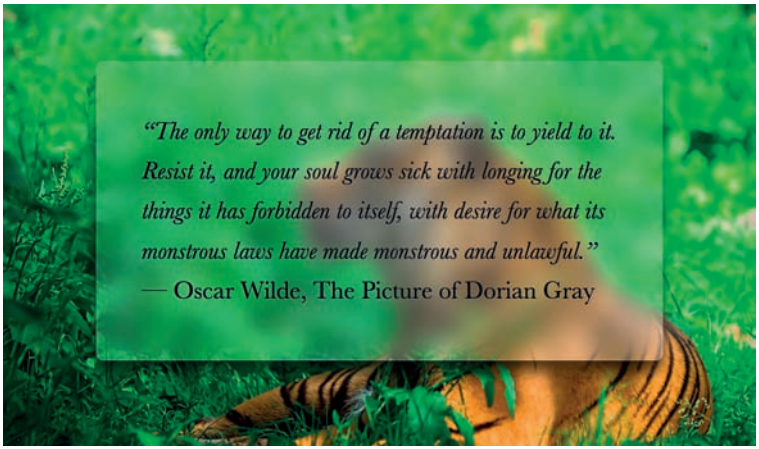
```
body, main::before {
  background: url("tiger.jpg") 0 / cover fixed;
}
main {
  position: relative;
  background: hsla(0,0%,100%,.3);
}
main::before {
  content: '';
  position: absolute;
  top: 0; right: 0; bottom: 0; left: 0;
  filter: blur(20px);
}
```

RYSUNEK 4.18.
Pseudoelement zasłania tekst

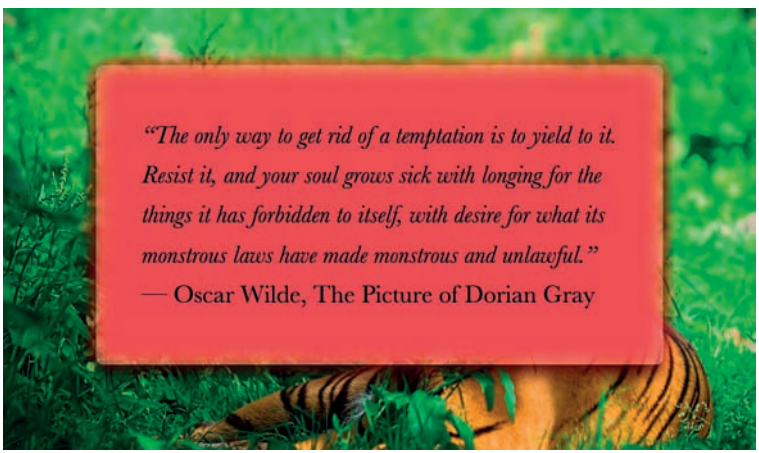


RYSUNEK 4.19.
Pseudoelement został przesunięty pod swojego rodzica za pomocą deklaracji z-index: –1;

Jak widać na rysunku 4.20, niemal się udało. Efekt rozmycia w środku elementu wygląda doskonale, ale jego natężenie w pobliżu krawędzi wyraźnie spada. Dzieje się tak dlatego, że rozmycie zmniejsza zasięg obszaru pokrytego jednolitym kolorem o wartość równą promieniowi rozmycia. Zastosowanie tła w kolorze red dla pseudoelementu dobrze pokazuje, na czym to zjawisko polega (rysunek 4.21).

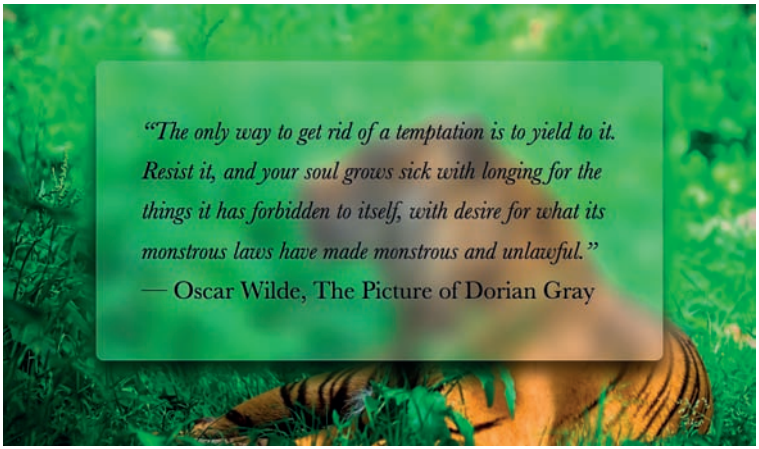


RYSUNEK 4.20. Rozmycie pseudoelementu zadziałało… prawie: intensywność rozmycia jest mniejsza przy krawędziach kontenera, co zaburza iluzję matowego szkła

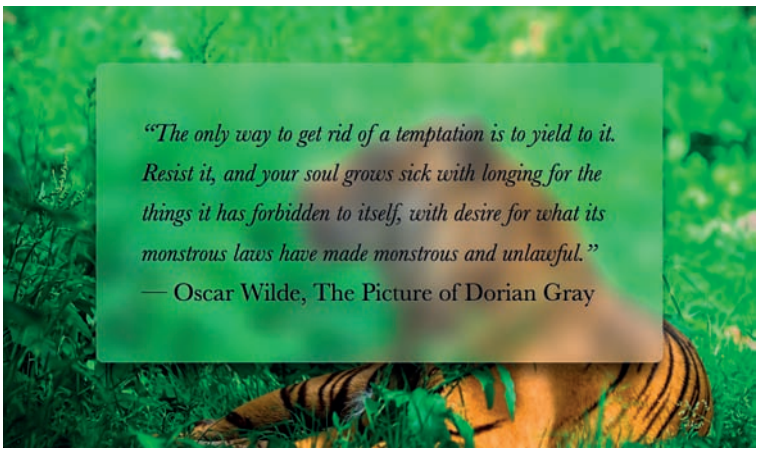


RYSUNEK 4.21. Dodanie tła w kolorze red ułatwia zrozumienie przyczyny tego zjawiska

Aby zapobiec temu problemowi, powiększymy pseudoelement tak, by był o co najmniej `20px` (tyle, ile wynosi promień rozmycia) większy od kontenera; ponadto zastosujemy margines `-20px` lub mniejszy, na wszelki wypadek, w razie gdyby w różnych przeglądarkach funkcjonowały różne algorytmy rozmycia. Na rysunku 4.22 pokazano, że zabiegi te rozwiązują problem z zanikaniem rozmycia przy krawędziach, ale zarazem sprawiają, iż rozmycie jest widoczne poza granicami kontenera i przypomina szeroką smugę, a nie kawałek matowego szkła. Na szczęście można to łatwo naprawić: wystarczy dodać deklarację `overflow: hidden;` do elementu `main`, co spowoduje przycięcie nadmiarowego rozmycia. Ostatecznie kod wygląda tak jak poniżej, natomiast efekt jego działania ilustruje rysunek 4.23.



RYSUNEK 4.22.
Efekt rozmycia przy krawędziach został skorygowany, ale wskutek tego rozmycie sięga poza granice elementu



RYSUNEK 4.23.
Gotowy efekt

```
body, main::before {
  background: url("tiger.jpg") 0 / cover fixed;
}
main {
  position: relative;
  background: hsla(0,0%,100%,.3);
  overflow: hidden;
}
main::before {
  content: '';
  position: absolute;
```

```
    top: 0; right: 0; bottom: 0; left: 0;
    filter: blur(20px);
    margin: -30px;
}
```

Zauważ, o ile czytelniejszy stał się teraz projekt strony i jak ładnie się prezentuje. Trudno ocenić, czy degradacja tego efektu w przypadku braku obsługi filtrów jest akceptowalna, czy nie: w takiej sytuacji otrzymamy bowiem efekt pokazany na początku (rysunek 4.14). W ramach rozwiązania awaryjnego możemy zwiększyć krycie koloru tła, aby poprawić czytelność tekstu.

▸ **WYPRÓBUJ** *play.csssecrets.io/frosted-glass*

POWIĄZANE SPECYFIKACJE

- **Filter Effects**
  w3.org/TR/filter-effects

19 Efekt zagiętego rogu

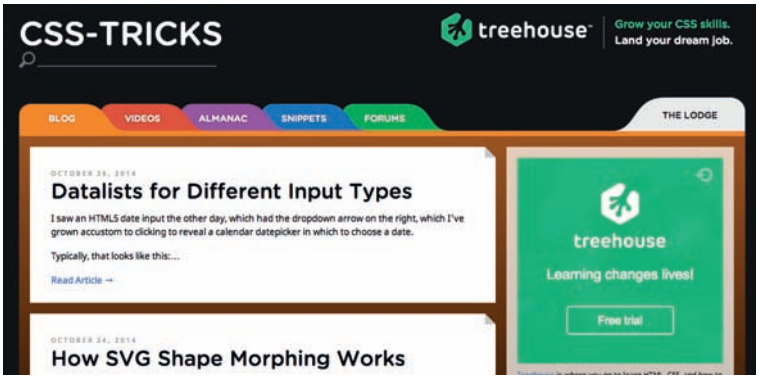
Wymagania

Przekształcenia CSS, gradienty CSS, sekret 12. „Przycięte narożniki”.

Problem

Zmiana wyglądu jednego z rogów elementu (zwykle prawego górnego albo prawego dolnego) w taki sposób, by sprawiał wrażenie mniej lub bardziej realistycznie zagiętego, to efekt od lat cieszący się niesłabnącą popularnością (rysunek 4.24).

Obecnie istnieje kilka metod pozwalających uzyskać ten efekt, opracowanych w czystym CSS. Na pomysł jednego z nich już w 2010 roku wpadł mistrz zastosowań pseudoelementów, Nicolas Gallagher (*nicolasgallagher.com/pure-css-folded-corner-effect*). Główna zasada tego rodzaju trików polega zwykle na dodaniu w rogu elementu dwóch trójkątów: jeden symbolizuje zagięcie, a drugi jest biały i ma na celu zasłonięcie „prawdziwego” narożnika głównego elementu. Trójkąty te są zwykle tworzone przy użyciu starej sztuczki z ramką.



RYSUNEK 4.24. W kilku starszych wersjach serwisu css-tricks.com występowały zagięte narożniki, umiejscowione w prawym górnym rogu każdego artykułu

Choć na owe czasy rozwiązania tego rodzaju bezsprzecznie były godne uznania, dziś są niewystarczające i ograniczają nasze możliwości w pewnych sytuacjach:

- gdy tło elementu nie jest jednolite, lecz ma postać deseniu, tekstury, zdjęcia, gradientu albo w zasadzie dowolnego obrazu;
- gdy zależy nam na tym, by róg został zagięty pod kątem innym niż 45°, albo jeśli chcemy, by kąt zagięcia był animowany.

Czy da się uzyskać w CSS bardziej elastyczny efekt zagiętego rogu, pozbawiony powyższych mankamentów?

Rozwiązanie dla kąta 45°

Rozpoczniemy od utworzenia elementu z przyciętym prawym górnym rogiem, uzyskanym za pomocą metody z gradientem, opisanej w sekrecie 12. „Przycięte narożniki”. Aby przy użyciu tej metody otrzymać przycięty prawy górny róg o wielkości `1em`, należy zastosować poniższy kod. Efekt jego działania został pokazany na rysunku 4.25.

```
background: #58a; /* Rozwiązanie awaryjne */
background:
  linear-gradient(-135deg, transparent 2em, #58a 0);
```

> “The only way to get rid of a temptation is to yield to it.” — Oscar Wilde, The Picture of Dorian Gray

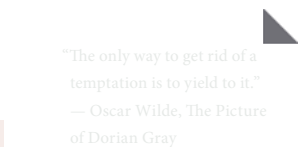
RYSUNEK 4.25. Punkt wyjścia: element z prawym górnym rogiem przyciętym za pomocą gradientu

Na tym etapie połowę zadania mamy właściwie z głowy: wystarczy dodać ciemniejszy trójkąt symbolizujący zagięty róg. Taki trójkąt można utworzyć za pomocą kolejnego gradientu, któremu przy użyciu atrybutu `background-size` nadamy odpowiednią wielkość, a potem umiejscowimy go w prawym górnym rogu.

Potrzebujemy w tym celu ustawionego pod kątem gradientu liniowego z dwoma znacznikami koloru, spotykającymi się w jego środku:

```
background:
  linear-gradient(to left bottom,
    transparent 50%, rgba(0,0,0,.4) 0)
    no-repeat 100% 0 / 2em 2em;
```

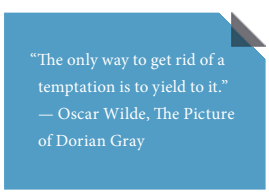
Na rysunku 4.26 pokazano tło tylko z tym jednym trójkątem. Wydaje się, że teraz wszystko sprowadza się do połączenia obu efektów… Ale czy aby na pewno? Spróbujmy to zrobić, pamiętając o tym, że trójkąt symbolizujący zagięty róg znajduje się ponad gradientem odpowiadającym za przycięcie narożnika:



RYSUNEK 4.26. Drugi gradient, odpowiedzialny za efekt zagiętego rogu, pokazany niezależnie od kontenera. Jasnoszary tekst pozwala się zorientować co do położenia rogu

```
background: #58a; /* Rozwiązanie awaryjne */
background:
  linear-gradient(to left bottom,
    transparent 50%, rgba(0,0,0,.4) 0)
    no-repeat 100% 0 / 2em 2em,
  linear-gradient(-135deg, transparent 2em, #58a 0);
```

Rezultat odbiega od naszych oczekiwań, co pokazano na rysunku 4.27. Dlaczego wielkości rogów się nie zgadzają? Przecież obydwu rogom nadaliśmy rozmiar `2em`!



RYSUNEK 4.27. Połączenie dwóch gradientów nie wygląda dokładnie tak, jak tego oczekiwaliśmy

Przyczyna tej sytuacji (wspomniana już w sekrecie 12. „Przycięte narożniki") jest taka, że wielkość `2em` w drugim gradiencie odpowiada położeniu znacznika koloru, jest więc mierzona wzdłuż ukośnej linii gradientu. Z kolei długość `2em` w przypadku właściwości `background-size` jest mierzona względem szerokości i wysokości elementu w tle, czyli poziomo i pionowo.

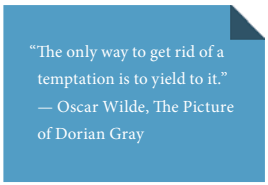
Jeśli chcemy, by oba narożniki spotkały się w tym samym miejscu, musimy wykonać jedną z następujących operacji, w zależności od tego, który z dwóch rozmiarów wolimy uzyskać:

- aby pozostawić wielkość `2em` mierzoną ukośnie, powinniśmy pomnożyć wartość użytą w deklaracji `background-size` przez $\sqrt{2}$;
- aby pozostawić wielkość `2em` mierzoną w poziomie i w pionie, powinniśmy podzielić położenie znacznika koloru w gradiencie przez $\sqrt{2}$.

Ponieważ atrybut `background-size` powtarza się dwukrotnie, a większość innych długości w CSS nie jest mierzona ukośnie, na ogół lepszym wyjściem będzie drugie z podanych rozwiązań. Położenie znacznika koloru będzie zatem wynosiło $\frac{2}{\sqrt{2}} = \sqrt{2} \approx 1{,}414213562$, co można zaokrąglić do `1.5em`:

```
background: #58a; /* Rozwiązanie awaryjne */
background:
  linear-gradient(to left bottom,
    transparent 50%, rgba(0,0,0,.4) 0)
    no-repeat 100% 0 / 2em 2em,
  linear-gradient(-135deg,
      transparent 1.5em, #58a 0);
```

Jak widać na rysunku 4.28, dopiero ten kod daje nam elegancki, elastyczny i minimalistyczny efekt zagiętego rogu.



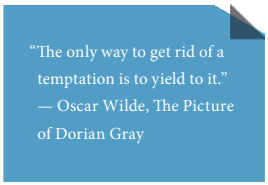
RYSUNEK 4.28. Po zmianie położenia znacznika koloru niebieskiego gradientu efekt zagiętego rogu wreszcie działa jak trzeba

▸ **WYPRÓBUJ** *play.csssecrets.io/folded-corner*

Rozwiązania dla innych kątów

W rzeczywistości rzadko spotyka się rogi zagięte dokładnie pod kątem 45°. Jeśli chciałbyś uzyskać „ośle ucho" wyglądające nieco naturalniej, mógłbyś wybrać trochę inny kąt, na przykład 30°, co będzie wymagało podania w parametrach gradientu wartości `-150deg`. Jeśli jednak po prostu zmienisz kąt zagięcia rogu, to trójkąt imitujący tę część strony nie dopasuje się do nowego ustawienia i będzie wyglądał podobnie jak na rysunku 4.29. Okazuje się, że skorygowanie wymiarów takiego rogu wcale nie jest proste. Wielkość tego trójkąta nie jest bowiem podyktowana przez kąt ułożenia, ale przez jego szerokość i wysokość. Jak wobec tego możemy wyliczyć potrzebne parametry? No cóż, przyszedł czas na szczyptę… tak, trygonometrii!

! Upewnij się, że wartość `padding` dla kontenera jest co najmniej równa wielkości narożnika, bo w przeciwnym razie tekst będzie częściowo przesłaniał ów narożnik (pamiętaj, że to tylko tło), niwecząc uzyskane wrażenie.

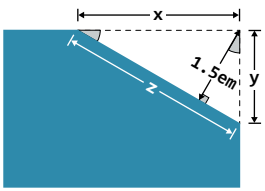


RYSUNEK 4.29. Zmiana kąta przycięcia rogu powoduje taki oto problem

Kod ma teraz taką postać:

```
background: #58a; /* Rozwiązanie awaryjne */
background:
  linear-gradient(to left bottom,
    transparent 50%, rgba(0,0,0,.4) 0)
    no-repeat 100% 0 / 2em 2em,
  linear-gradient(-150deg,
    transparent 1.5em, #58a 0);
```

Trójkąt prostokątny *30-60-90* jest trójkątem prostokątnym, w którym dwa pozostałe kąty wynoszą 30° i 60°.



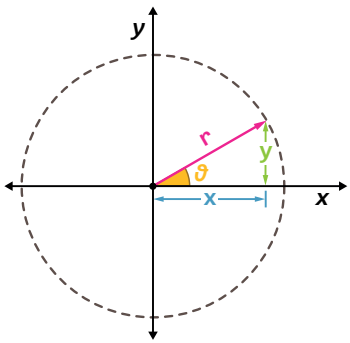
RYSUNEK 4.30.
Nasz ścięty narożnik w powiększeniu (na szaro oznaczono kąty o wartości 30°)

Jak wynika z rysunku 4.30, musimy obliczyć długości przeciwprostokątnych dwóch trójkątów prostokątnych *30-60-90*, znając długość jednej z ich przyprostokątnych. Na podstawie wykresu zależności trygonometrycznych, pokazanego na rysunku 4.31, widzimy, że znając kąty oraz długość jednego z boków trójkąta prostokątnego, za pomocą funkcji sinus, cosinus i twierdzenia Pitagorasa możemy wyliczyć długości jego pozostałych dwóch boków. Z lekcji matematyki (albo dzięki kalkulatorowi) wiemy, że $\cos 30° = \frac{\sqrt{3}}{2}$, natomiast $\sin 30° = \frac{1}{2}$. Na podstawie wykresu wiemy też, że w naszym przypadku $\sin 30° = \frac{1{,}5}{x}$ oraz że $\cos 30° = \frac{1{,}5}{y}$. To zaś oznacza, że:

$$\frac{1}{2} = \frac{1{,}5}{x} \Rightarrow x = 2 \times 1{,}5 \Rightarrow x = 3$$

$$\frac{\sqrt{3}}{2} = \frac{1{,}5}{2} \Rightarrow \frac{2 \times 1{,}5}{\sqrt{3}} \Rightarrow y = \sqrt{3} \approx 1{,}732050808$$

RYSUNEK 4.31.
Funkcje sinus i cosinus umożliwiają obliczenie długości przyprostokątnych w trójkątach prostokątnych na podstawie znanych kątów i długości przeciwprostokątnej



$$x^2 + y^2 = r^2$$

$$\cos\vartheta = \frac{y}{r}$$

$$\sin\vartheta = \frac{x}{r}$$

Na tym etapie możemy też obliczyć wartość z, korzystając z twierdzenia Pitagorasa:

$$z = \sqrt{x^2 + y^2} = \sqrt{\sqrt{3}^2 + 3^2} = \sqrt{3+9} = \sqrt{12} = 2\sqrt{3}$$

Dysponując tą wiedzą, możemy skorygować wielkość trójkąta:

```
background: #58a; /* Rozwiązanie awaryjne */
background:
  linear-gradient(to left bottom,
    transparent 50%, rgba(0,0,0,.4) 0)
    no-repeat 100% 0 / 3em 1.73em,
  linear-gradient(-150deg,
    transparent 1.5em, #58a 0);
```

Na tym etapie zagięty róg prezentuje się tak jak na rysunku 4.32. Jak widać, trójkąt pasuje do ścięcia, ale wygląda jeszcze mniej naturalnie niż poprzednio! Początkowo trudno się zorientować, dlaczego tak się dzieje, ale widzieliśmy w życiu wystarczająco wiele zagiętych rogów, by mieć pewność, że uzyskany efekt zdecydowanie różni się od tego, do czego przywykliśmy. Możemy jednak ułatwić sobie zrozumienie, dlaczego ów róg wygląda tak sztucznie, jeśli zagniemy kartkę papieru pod mniej więcej takim kątem, jaki próbujemy uzyskać. No cóż — po prostu nie sposób zagiąć jej tak, by choć w pewnym zakresie przypominała efekt uzyskany na rysunku 4.32.

> "The only way to get rid of a temptation is to yield to it." — Oscar Wilde, The Picture of Dorian Gray

RYSUNEK 4.32. Choć w jakimś sensie zrealizowaliśmy swój pierwotny zamiar, to okazuje się, że efekt jest jeszcze mniej przekonujący niż wcześniej

Na przykładzie prawdziwego zagiętego rogu, takiego jak na rysunku 4.33, możemy zobaczyć, że powinniśmy utworzyć trójkąt nieznacznie obrócony i o tych samych wymiarach co trójkąt „wycięty" z rogu elementu. A ponieważ tła nie da się obrócić, musimy przenieść efekt na pseudoelement:

```
.note {
  position: relative;
  background: #58a; /* Rozwiązanie awaryjne */
  background:
    linear-gradient(-150deg,
      transparent 1.5em, #58a 0);
```

RYSUNEK 4.33. Analogowa wersja zagiętego rogu (ozdobną kartkę przygotowały Leonie i Phoebe Verou)

```
}
.note::before {
  content: '';
  position: absolute;
  top: 0; right: 0;
  background: linear-gradient(to left bottom,
    transparent 50%, rgba(0,0,0,.4) 0)
    100% 0 no-repeat;
  width: 3em;
  height: 1.73em;
}
```

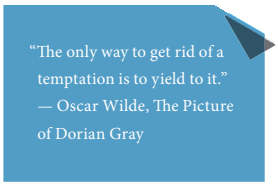
Ten kod powiela efekt z rysunku 4.32, lecz przy użyciu pseudoelementów. Następny krok będzie polegał na zmianie orientacji istniejącego trójkąta przez zamianę miejscami wartości parametrów `width` i `height`, co pozwoli uzyskać lustrzane odbicie wyciętego rogu zamiast jego dopełnienia. Następnie obrócimy uzyskany kształt o 30° ((90° – 30°) – 30°) w kierunku przeciwnym do ruchu wskazówek zegara, dzięki czemu przeciwprostokątna trójkąta stanie się równoległa do krawędzi ściętego rogu:

```
.note::before {
  content: '';
  position: absolute;
  top: 0; right: 0;
  background: linear-gradient(to left bottom,
    transparent 50%, rgba(0,0,0,.4) 0)
    100% 0 no-repeat;
  width: 1.73em;
  height: 3em;
  transform: rotate(-30deg);
}
```

Po wprowadzeniu zmian nasza „kartka z oślim uchem" wygląda tak jak na rysunku 4.34. Powoli zmierzamy we właściwym kierunku, musimy jednak doprowadzić do tego, by przeciwprostokątne dwóch trójkątów (ciemniejszego oraz tego, który pełni funkcję ściętego rogu) pokrywały

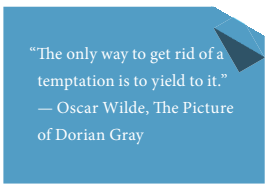
się. W tej chwili powinniśmy przesunąć trójkąt zarówno w poziomie, jak i w pionie, trudno jest więc oszacować parametry takiego przekształcenia. Możemy jednak ułatwić sobie zadanie, zmieniając wartość atrybutu `transform-origin` na `bottom right`, dzięki czemu punktem odniesienia obrotu stanie się prawy dolny wierzchołek trójkąta. To zaś oznacza, że nie ulegnie on przesunięciu podczas przekształcania:

```
.note::before {
  /* [Reszta stylów] */
  transform: rotate(-30deg);
  transform-origin: bottom right;
}
```



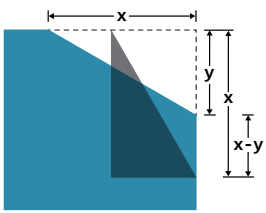
RYSUNEK 4.34. Zmierzamy we właściwym kierunku, ale musimy jeszcze przesunąć trójkąt

Jak widać na rysunku 4.35, teraz wystarczy przesunąć trójkąt pionowo w górę. Aby wyznaczyć dokładną wartość tego przesunięcia, możemy ponownie skorzystać z geometrii. Z rysunku 4.36 wynika, że pionowe przemieszczenie, jakiego powinniśmy dokonać, ma wartość $x - y = 3 - \sqrt{3} \approx 1{,}267949192$, co możemy zaokrąglić do `1.3em`:



RYSUNEK 4.35. Dodanie deklaracji transform-origin: bottom right; trochę upraszcza sprawę: teraz musimy jeszcze tylko przesunąć trójkąt w pionie

```
.note::before {
  /* [Reszta stylów] */
  transform: translateY(-1.3em) rotate(-30deg);
  transform-origin: bottom right;
}
```

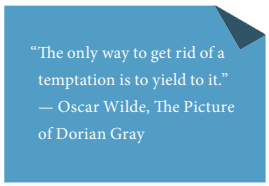


RYSUNEK 4.36. Wyznaczenie odległości, na jaką należy przesunąć trójkąt, wcale nie jest tak trudne, na jakie wygląda

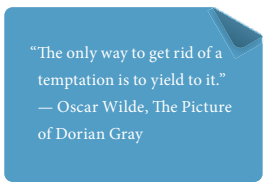
Przykład pokazany na rysunku 4.37 potwierdza, że wreszcie udało się nam uzyskać zamierzony efekt. *Trochę* to kosztowało zachodu! Ale ponieważ nasze ośle ucho zostało teraz wygenerowane przy użyciu pseudoelementów, to możemy nadać mu jeszcze bardziej realistyczny wygląd poprzez dodanie zaokrąglonych rogów, (prawdziwych) gradientów i cieni typu `box-shadow`! Ostateczna wersja kodu jest taka:

Upewnij się, że przekształcenie `translateY()` zostanie wykonane **przed** obrotem, ponieważ w przeciwnym razie trójkąt przemieści się nie w pionie, ale pod kątem 30°. Stanie się tak dlatego, że każde przekształcenie obejmuje cały układ współrzędnych elementu, a nie tylko sam element!

```
.note {
  position: relative;
  background: #58a; /* Rozwiązanie awaryjne */
  background:
    linear-gradient(-150deg,
      transparent 1.5em, #58a 0);
  border-radius: .5em;
}
.note::before {
  content: '';
  position: absolute;
  top: 0; right: 0;
  background: linear-gradient(to left bottom,
    transparent 50%, rgba(0,0,0,.2) 0, rgba(0,0,0,.4))
    100% 0 no-repeat;
  width: 1.73em;
  height: 3em;
  transform: translateY(-1.3em) rotate(-30deg);
  transform-origin: bottom right;
  border-bottom-left-radius: inherit;
  box-shadow: -.2em .2em .3em -.1em rgba(0,0,0,.15);
}
```



RYSUNEK 4.37. Trójkąty wreszcie pasują do siebie jak ulał



RYSUNEK 4.38. Po dodaniu kilku efektów zagięty narożnik wygląda jak prawdziwy!

Efekty naszej pracy możesz podziwiać na rysunku 4.38.

▸ WYPRÓBUJ *play.csssecrets.io/folded-corner-realistic*

Efekt wygląda ładnie, ale czy jest SUCHY? Pomyślmy o typowych poprawkach i modyfikacjach, jakie moglibyśmy chcieć tutaj wprowadzić:

- Zmiana wielkości elementu oraz innych związanych z nim wymiarów (na przykład odstępu wewnętrznego) wymaga tylko jednej poprawki w kodzie.
- Dwóch poprawek (jedna dotyczy rozwiązania awaryjnego) potrzeba do zmiany koloru tła.

- Zmiana wielkości zagiętego rogu wymaga czterech modyfikacji i kilku nieoczywistych obliczeń.
- Pięciu poprawek i jeszcze bardziej skomplikowanych obliczeń potrzeba do zmiany kąta zagięcia rogu.

Dwie ostatnie kwestie rzeczywiście są uciążliwe. W tej sytuacji warto się pokusić o napisanie domieszki dla preprocesora:

SCSS

```
@mixin folded-corner($background, $size,
                     $angle: 30deg) {
position: relative;
background: $background; /* Fallback */
background:
  linear-gradient($angle - 180deg,
    transparent $size, $background 0);
border-radius: .5em;

$x: $size / sin($angle);
$y: $size / cos($angle);

&::before {
  content: '';
  position: absolute;
  top: 0; right: 0;
  background: linear-gradient(to left bottom,
    transparent 50%, rgba(0,0,0,.2) 0,
    rgba(0,0,0,.4)) 100% 0 no-repeat;
  width: $y; height: $x;
  transform: translateY($y - $x)
             rotate(2*$angle - 90deg);
  transform-origin: bottom right;
  border-bottom-left-radius: inherit;
  box-shadow: -.2em .2em .3em -.1em rgba(0,0,0,.2);
}
}
```

```
/* zastosowanie... */
.note {
  @include folded-corner(#58a, 2em, 40deg);
}
```

W chwili gdy piszę te słowa, preprocesor SCSS nie oferuje wbudowanych funkcji trygonometrycznych. Aby móc się nimi posługiwać, możesz użyć frameworka Compass (*compass-style.org*), a także jednej z kilku innych bibliotek. Możesz też napisać potrzebne funkcje sam, korzystając z szeregu Taylora! Z kolei preprocesor LESS ma wbudowaną obsługę funkcji trygonometrycznych.

▸ **WYPRÓBUJ** *play.csssecrets.io/folded-corner-mixin*

POWIĄZANE SPECYFIKACJE

- **CSS Backgrounds & Borders**
  w3.org/TR/css-backgrounds
- **CSS Image Values**
  w3.org/TR/css-images
- **CSS Transitions**
  w3.org/TR/css-transitions

Pierwsza specyfikacja standardu CSS została opublikowana w 1996 roku. Była bardzo krótka, zwięzła i stosunkowo prosta, po wydrukowaniu zajmowała niespełna 70 stron. Od tego czasu język CSS stał się na tyle złożonym i zaawansowanym narzędziem, że dziś właściwie niemożliwe jest, aby jedna osoba biegle poznała wszystkie jego tajniki. Obecny CSS wciąż zaskakuje nieoczekiwanymi możliwościami, dalece wykraczającymi poza proste ozdabianie stron WWW — gradienty, przejścia, cienie, ciekawe narożniki, animacje, łamanie tekstu. Sięgnij po tę książkę i spraw, by pisany przez Ciebie kod był zgodny z dobrymi praktykami.

Jeśli w miarę płynnie posługujesz się CSS i chcesz udoskonalić swój warsztat programisty, a przy tym lubisz nietuzinkowe, inspirujące rozwiązania, wykorzystaj ciekawe propozycje autorki — masz ich do dyspozycji 47. Dzięki nim poradzisz sobie z często spotykanymi trudnościami związanymi z projektowaniem stron WWW. Co więcej, dzięki tej książce nauczysz się tworzyć kod zwięzły, łatwy w utrzymaniu, elastyczny, lekki i zgodny z obowiązującymi standardami.

ODKRYJ TAJEMNICE CSS
— I PROJEKTUJ ZASKAKUJĄCO PIĘKNE STRONY, PISZĄC CZYSTY, ZWIĘZŁY KOD!

W tej książce znajdziesz wskazówki dotyczące:

- **dobrych praktyk kodowania w CSS i standardów w tym zakresie**
- **programowania kształtów, ramek, cieni, gradientów**
- **tworzenia interesujących efektów wizualnych, w tym uwzględniających zasady perspektywy**
- **reguł typografii**
- **wymogów funkcjonalności strony**
- **projektowania struktury i układu strony**

Lea Verou jest utalentowaną programistką, znaną z niekonwencjonalnego podejścia do tworzenia stron WWW. Jest niezależną ekspertką w grupie roboczej W3C CSS. Wcześniej była rzecznikiem programistów w W3C. Obecnie pracuje w Massachusetts Institute of Technology (MIT) — zajmuje się badaniem interfejsów służących do komunikacji między człowiekiem a maszyną. Chętnie dzieli się swoimi doświadczeniami z innymi — prowadzi bloga, występuje na międzynarodowych konferencjach i jest autorką popularnych projektów open source, ułatwiających pracę programistom.

Helion
41644 numer katalogowy
księgarnia internetowa
http://helion.pl
zamówienia telefoniczne
0 801 339900
0 601 339900
Informatyka w najlepszym wydaniu

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

O'REILLY®

ISBN 978-83-283-1716-1
9 788328 317161
cena: 59,00 zł