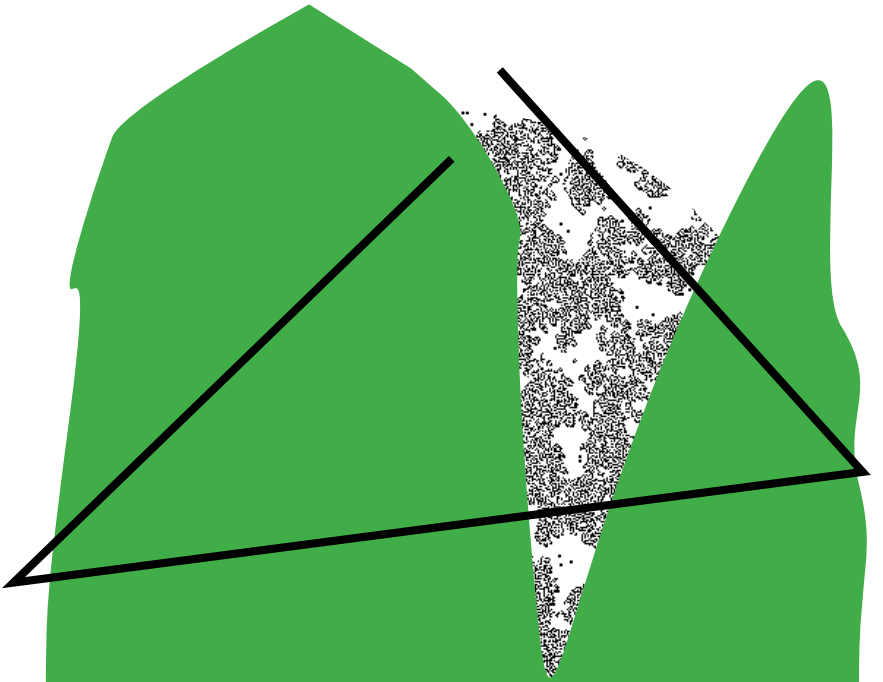


PROCESSING

podstawy technologii multimedialnych dla artystów

Przemysław Sanecki

«**Przemysław Sanecki** pisze, że "jeżeli współczesny artysta chce zmierzyć się z szerszą perspektywą sztuki, to nie może ignorować podłoża technologicznego świata, w którym żyje". Jest to trafna uwaga, którą potwierdzają medioznawcy, jak np. Peter Lunenfeld, który uważa komputer za podstawową maszynę kulturową. Podobny pogląd sformułował już w latach 90. XX wieku niemiecki teoretyk mediów Friedrich Kittler, zauważając, że współczesny artysta powinien znać przynajmniej jeden język programowania, podobnie jak powinno znać się języki obce. Koncepcja mediów technologicznych, rozwijana przez Kittlera, opiera się na założeniu, że we wszystkich praktykach codzienności, w tym praktykach twórczych, jak np. pisanie, pośredniczy technologia, a pisanie programów jest logiczną, dziejową konsekwencją pisania w sensie literackim.»
(z recenzji dr Ewy Wójtowicz)

PROCESSING

podstawy technologii multimedialnych dla artystów

Przemysław Sanecki

spis

treści

wstęp/ wprowadzenie

Dla praktyki artystycznej kilka zagadnień wydaje się wyjątkowo znaczących. Mam na myśli takie pojęcia jak interaktywność, generatywność czy komunikacja. Oczywiście każde z nich zasługuje na szersze omówienie, a przede wszystkim na usprawiedliwienie użytych do ich skonstruowania uogólnień. Unikam jednoznacznych definicji w skrypcie, z dwóch powodów – pozostawiam trud zbudowania definicji studentom oraz wybieram mniej bezpośrednie podejście – przykłady w skrypcie wskazują na te pojęcia, pozostawiając ich dosłowną artykulację w domyśle. Przede wszystkim jednak podstawową techniką nauczaną w skrypcie jest transkrypcja myśli i zamiarów na język, w którym możliwe jest sterowanie maszyną. Stąd też nacisk na składnię języka programowania.

Celem niniejszej publikacji jest dostarczenie studentowi pomocy naukowej do przedmiotu propedeutycznego realizowanego przeze mnie na pierwszym roku studiów licencjackich w Katedrze Intermediów UA w Poznaniu. Ze względu na specyfikę studiów, podręcznik ten przeznaczony jest dla studenta posiadającego niezbyt zaawansowaną wiedzę na temat obsługi komputera. Skrypt będzie zawierał znikomą ilość formalnych informacji z zakresu informatyki z korzyścią, mam nadzieję, dla programistycznej heurystyki[1]. Skrypt nie obejmuje całości materiału przerabianego w ramach zajęć, choć stanowi jego znaczną cześć, skupioną na programistycznych podstawach praktyki multimedialnej. Nie było też moim celem stworzenie podręcznika, nadającego się w pełni do samodzielnego studiowania, zwłaszcza dla osób, które po raz pierwszy stykają się z programowaniem. Stąd też nie należy się obawiać, jeśli na pierwszy rzut oka materiał wyda się czytelnikowi zbyt „gęsty". Treści zawarte w tekście stanowią bazę dla wykładów i ćwiczeń realizowanych pod moją opieką. Należy zaznaczyć, że każdy z poruszonych w skrypcie tematów jest potraktowany z wielu względów skrótowo i zasługuje na szersze omówienie, które dalece wykracza poza ramy skryptu. Pełne przyswojenie materiału przerabianego na zajęciach wymaga dodatkowo realizacji ćwiczeń oraz osobistego zaangażowania studenta w proces pisania programów.

W pierwszych rozdziałach omawiana jest głównie składnia języka programowania. Rozdziały te inspirowane są serią podręczników „How to think like

1 heurystyka [fr. *heuristique* < gr. *heurískō* 'znaduję'], *metodol.*, umiejętność wykrywania nowych faktów i związków między faktami (a zwłaszcza stawiania hipotez), prowadząca do poznania nowych naukowych prawd (http://encyklopedia.pwn.pl/haslo.php?id=4008452)

a computer scientist"[2]. Do tej pory opublikowano kilka jego wersji dla różnych języków – *Python, C++,* Java[3]. Skrypt nie jest jednak czymś w rodzaju *How to think like a computer artist*, mam bowiem duże wątpliwości, czy coś takiego jak „sztuka komputerowa" w ogóle istnieje. Technologie informacyjne (IT) kształtują od dziesięcioleci nie tylko nasz odbiór rzeczywistości, ale i samą rzeczywistość w jej nieskończonych wymiarach (ekologicznym, politycznym, kulturowym, itd.). Proces ten ciągle przybiera na sile. Jeżeli współczesny artysta chce się zmierzyć z szerszą perspektywą sztuki, to nie może ignorować zaplecza technologicznego świata w którym żyje. Bez tego nie będzie możliwa żadna adekwatna do naszych czasów teoria i praktyka. Termin „sztuka komputerowa" jest mylący chociażby dlatego, że sugeruje, iż chodzi tu o praktykę zawężoną do jakiegoś urządzenia elektronicznego.

Dla praktyki artystycznej kilka zagadnień wydaje się wyjątkowo znaczących. Mam na myśli takie pojęcia jak interaktywność, generatywność czy komunikacja. Oczywiście każde z nich zasługuje na szersze omówienie, a przede wszystkim na usprawiedliwienie użytych do ich skonstruowania uogólnień. Unikam jednoznacznych definicji w skrypcie, z dwóch powodów – pozostawiam trud zbudowania definicji studentom oraz wybieram mniej bezpośrednie podejście – przykłady w skrypcie wskazują na te pojęcia, pozostawiając ich dosłowną artykulację w domyśle. Przede wszystkim jednak podstawową techniką nauczaną w skrypcie jest transkrypcja myśli i zamiarów na język, w którym możliwe

2 Patrz http://www.openbookproject.net/thinkcs/archive/
3 Wszystkie podręczniki należą do domeny publicznej.

jest sterowanie maszyną. Stąd też nacisk na składnię języka programowania.

Od użycia języków algorytmicznych w twórczości artystycznej odstrasza artystów następująca trudność. O artystycznej wartości programów nie decyduje, tak jak w przypadku poezji, umiejętne użycie nieścisłości języka naturalnego w celu wywołania określonych efektów poetyckich, lecz coś zgoła przeciwnego – taka konfiguracja jednoznacznych instrukcji, która w efekcie doprowadza użytkownika do niealgorytmicznych zachowań. Nie planujemy zatem kontroli uczuć, lecz poprzez kontrolę stwarzamy podstawę wolności. Jest to magia kultury, która może się wykształcić jedynie w cywilizacji dużo bardziej zinformatyzowanej niż ta, w której żyjemy obecnie.

W przypadku programowania, zrozumienie przychodzi często w miarę użycia. Pomocny jest więc od czasu do czasu „skok wiary". Przedstawianie programowania jako aktywności właściwej tylko umysłom inżynierów, bierze się bądź z nieznajomości przedmiotu, bądź z deprecjonowania własnych zdolności. Tymczasem mamy do czynienia z techniką, do nauki której należy podejść bez uprzedzeń. Na poparcie tej tezy przywołam słowa Josepha Weizenbauma, autora legendarnego bota[4] ELIZA[5] – „It happens that programming is a relatively easy craft to learn. Almost anyone with a reasonably orderly mind can become a fairly good programmer with just a little instruction and practice."[6] Zdaję sobie jednak sprawę z tego, że programowanie nie dla wszystkich twórców jest odpowiednim narzę-

4 Programu z którym można prowadzić konwersację w języku naturalnym

5 Możesz z „Nią" porozmawiać, np. http://www-ai.ijs.si/eliza-cgi-bin/eliza_script

6 http://www.smeed.org/1735

dziem artystycznego rozwoju. Niemniej mam nadzieję, że zawarta w niniejszym opracowaniu wiedza, pozwoli choćby w niewielkim stopniu, zredukować uczucie wyobcowania, które w kontakcie z techniką często towarzyszy humanistom.

Kilka słów o przykładach. Rdzeniem realizacji programistycznych są algorytmy – efektywne i powtarzalne procedury postępowania. Jest nawet książka autorstwa Niklasa Wirtha, której tytuł brzmi dosłownie „Algorytmy + struktury danych = program"[7]. Pomimo tego, że algorytmika jest fascynującą dziedziną, niekoniecznie może być interesująca dla studenta sztuki. Problem w tym kontekście pojawia się następujący – jak rozwiązać kwestię stylu oraz kreatywności w programowaniu? Jest to kwestia daleka od jednoznacznego rozwiązania. Własna implementacja algorytmu stosowanego przez tysiące programistów nie musi być koniecznie skazana na odtwórczość. Stanąłem przed tym problemem pisząc programy do podręcznika, bowiem w zbiorze przykładów dołączonych do PDE Processingu można znaleźć odmienne realizacje wielu technik obecnych w skrypcie.

Kod jest wyróżniony w tekście czcionką techniczną. Mogę śmiało powiedzieć, że te fragmenty są nawet istotniejsze od tekstu pisanego zwykłą czcionką. W każdym razie nie powinno się ich omijać w trakcie lektury. Starałem się zewrzeć je semantycznie na tyle, żeby konieczność komentarza ograniczyć do minimum. Zdecydowałem się też na dość śmiały krok i w przykładach stosuję nazewnictwo polskie. Może się to wydawać stosunkowo dziwne, zważywszy na fakt, że języki

7 Wirth Niklaus „Algorytmy + struktury danych = programy", WNT, 2004, wyd.4

programowania opierają się na języku angielskim. Nie jest jednak moim celem wykształcenie korporacyjnego programisty, lecz zapoznanie przyszłego artysty z technologią, którą może posłużyć się w celach innych niż utylitarne IT. Ponieważ Processing jest dialektem Java, starać się będę przestrzegać, na swój polski sposób, standardów nazewnictwa Java*Beans*. W przeważającej części jest on gotowym skryptem, który wystarczy przekopiować do Processingu i uruchomić.

Pomimo tego, że istnieje kilkaset różnych języków programowania, to wiele pojęć związanych z problematyką pisania programów, takie jak zmienna, funkcja, itd., pozostają jednakowe lub bardzo do siebie zbliżone, niezależnie od języka w którym pracujemy. Składnia większości najbardziej popularnych z nich zawdzięcza wspólne cechy językowi proceduralnemu o nazwie C, który powstał około 1972 roku i zyskał niesamowitą popularność, głównie za sprawą systemu operacyjnego UNIX. Pomimo swojego wieku C jest w dalszym ciągu jednym z najczęściej stosowanych języków programowania. Oznacza to, że najtrudniejsze będzie poznanie pierwszego języka z tej rodziny, a używanie każdego kolejnego pozostanie kwestią zapoznania się z jego specyficznymi cechami. Mój wybór padł na Processing z kilku względów. Powstał w 2001 roku w Massachusetts Institute of Technology (MIT) w Stanach Zjednoczonych z inicjatywy dwóch studentów tamtejszego Media Lab, Ben Fry'a i Casey Reasa. Od tamtego czasu jest rozwijany jako projekt *open source* przez deweloperów z całego świata. Został on też doceniony nagrodą na festiwalu *Ars Electronica* w 2005 roku[8]. Na przestrzeni ostatnich dziesięciu lat stał się

8 http://www.aec.at/prix/en/gewinner/2005/

jednym z głównych narzędzi artystów pracujących z nowymi mediami. Poza tym istotne jest to, że Processing został zaprojektowany przez artystów i dla artystów. Ilość dziedzin, które możemy zgłębiać z jego pomocą jest olbrzymia – od graficznej animacji, przez cyfrową obróbkę wideo po sztuczne sieci neuronowe i wiele innych. Nie byłoby to możliwe bez wzajemnego wspierania się artystów i programistów, dzielących się własną pracą i doświadczeniem. Jak już wspomniałem, Processing jest „dialektem" Java, co oznacza, że uczyć się będziemy równocześnie jakby dwóch języków. Chciałbym podkreślić, że pod koniec kursu wiele innych języków programowania zacznie się wydawać kursantowi dziwnie znajome.

1 ZMIENNE I TYPY

1.1 PRZYGOTOWANIE ŚRODOWISKA PRACY

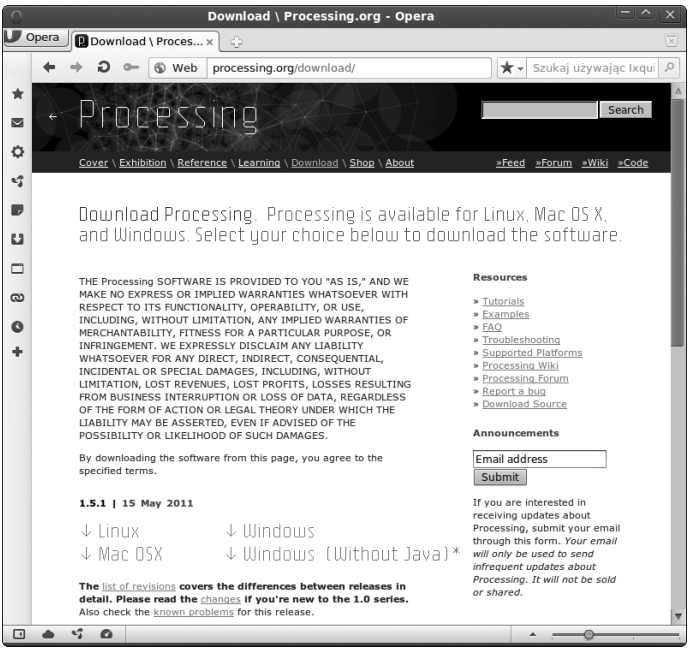
Poprzez środowisko pracy należy rozumieć zbiór podstawowych narzędzi potrzebnych jednostce do napisania i uruchomienia programu. Można środowisko pracy podzielić na dwie warstwy – warstwę urządzenia i warstwę oprogramowania – inaczej na hardware i software. Pod pojęciem hardware rozumiemy tu komputer. Do realizacji programu zawartego skrypcie wystarczy urządzenie o parametrach sprzed kilku lat, (zazwyczaj pracuję na czteroletnim laptopie marki Lenovo z procesorem Intel® Celeron® M 1,73GHz i RAM o pojemności 1,5GB i to w zupełności wystarcza). Warto też nabyć jakąś standardową kamerę internetową. Jeśli chodzi o software, to Processing jest dystrybuowany w trzech różnych wersjach na następujące systemy operacyjne – Linux, Windows oraz Mac OS X. Wszystkie wersje działają identycznie, więc którą z nich wybierzemy zależy od systemu na którym pracujemy. Ponieważ tylko Linux jest systemem nieodpłatnym, gorąco zachęcam do zainteresowania się tą platformą. Wspaniałą cechą Processing*u* jest to, że bez znaczenia w jakim systemie pisaliśmy program, uruchomimy go na każdej z pozostałych platform – mówimy wówczas, że kod jest przenośny (ang. *cross-platform*).

Dysponując komputerem z zainstalowanym systemem operacyjnym, kolejną rzeczą którą potrzebujemy jest oprogramowanie, pozwalające nam na edytowanie oraz uruchamianie kodu. Słowa Processing używam

w dwóch znaczeniach – w pierwszym mam na myśli język programowania oparty na Java, w drugim mam na myśli IDE. IDE jest skrótem od angielskiego wyrażenia Integrated Development Environment, tłumaczonego dosłownie na polski jako zintegrowane środowisko programistyczne. „Środowisko", ponieważ jest to grupa aplikacji; „zintegrowane", gdyż funkcje poszczególnych aplikacji są podporządkowane nadrzędnemu celowi całego środowiska; „programistyczne", gdyż celem środowiska jest tworzenie programów. Processing wygląda na pierwszy rzut oka jak niepozorny edytor tekstu, w rzeczywistości jednak edytor pełni tylko jedną z wielu innych funkcji IDE – edycję kodu źródłowego. Inne funkcje oferowane przez Processing to testowanie (konsola), interpretacja napisanego kodu oraz kompilacja kodu do samodzielnej aplikacji[9]. Podstawą każdego IDE jest przede wszystkim następująca funkcjonalność – edycja, testowanie i kompilacja kodu. Processingu jest również językiem skryptowym, czyli wykonywanym w nadrzędnym programie, zwanym interpretatorem. Z wersji skryptowej Processingu będziemy korzystać w toku zajęć. Kompilacją zajmiemy się w końcowych fragmentach podręcznika.

W celu instalacji Processingu musimy odwiedzić stronę internetową projektu i przejść do zakładki o nazwie *Download*. Należy ściągnąć wersję korespondującą z systemem operacyjnym, na którym pracujemy. W przypadku Windows wybierzmy wersję z Java.

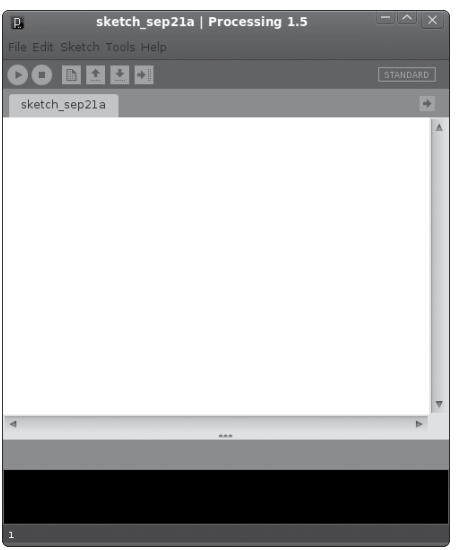
9 IDE Processingu jest też nazywane Processing Development Environment (PDE), które spełnia wszystkie podstawowe cechy środowiska pracy.



STRONA INTERNETOWA PROGRAMU (ZAKŁADKA Z PROGRAMEM DO ŚCIĄGNIĘCIA)

Processing jest dostarczany jako archiwum. Zapisujemy je w dowolnym miejscu na dysku. Po rozpakowaniu otrzymamy folder w którym znajduje się kilka folderów oraz plik o nazwie Processing. Przenieśmy ten folder w dowolne miejsce na dysku, w przypadku Windows może to być np. *Program Files*. Nie jest to jednak konieczne. Powinniśmy być w stanie uruchomić program poprzez dwukrotne kliknięcie w ikonę pliku Processing, bez względu na miejsce położenia folderu. Na tym właściwie kończy się instalacja.

FOLDER PROGRAMU PO ROZPAKOWANIU ARCHIWUM

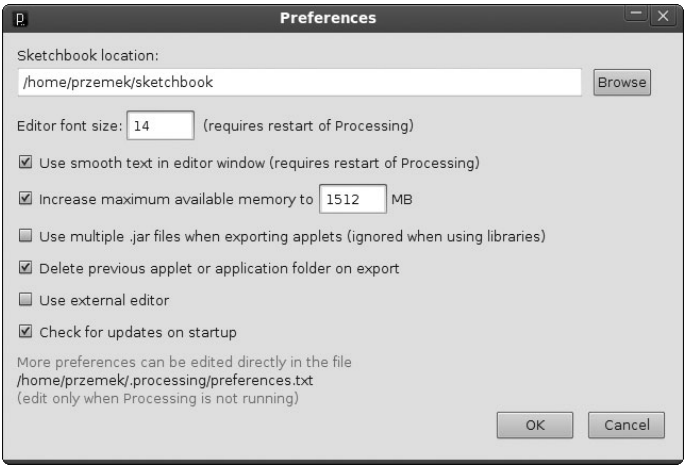


OKNO PROGRAMU PO PIERWSZYM URUCHOMIENIU

Ostatnią rzeczą, którą zrobimy przed rozpoczęciem pracy, jest ustalenie preferencji programu. W tym celu naciskamy jednocześnie dwa przyciski klawiatury – ctrl i przecinek. Możemy też wybrać myszką z menu pozycję *File->Preference*. Na poniższej ilustracji przedstawiam przykładowe ustawienia. Jedna z pozycji wymaga szczególnego omówienia. Mowa o polu o nazwie *Sketchbook location*. Processing podczas pierwszego uruchomienia tworzy domyślny katalog w którym będzie przechowywał pliki programów pisanych przez użytkownika. Podaję tu absolutną ścieżkę do katalogu – pod Linuxem jest to */home/<nazwa_użytkownika>/sketchbook*, pod Mac OS X */Users/<nazwa_użytkownika>/sketchbook*, natomiast pod Windows *C:\\Documents and Settings\<nazwa_użytkownika>*[10]*\Moje Dokumenty\Processing*. Możemy pozostawić domyślną lokalizację niezmienioną lub wybrać inne dogodne miejsce na twardym dysku. Należy pamiętać, że to w tym katalogu będą zapisywane wszystkie nasze programy, chyba, że w momencie zapisu świadomie zadecydujemy inaczej. Zalecam jednak na początku trzymać wszystkie pliki właśnie w tym miejscu. Ten folder jest ważny jeszcze z jednej przyczyny, którą poznamy później.

Z pozostałych ustawień zalecam ustawić wygładzanie tekstu w edytorze (wpłynie to na komfort pracy), jak również zwiększyć rozmiar pamięci operacyjnej z której mogą korzystać uruchomione skrypty do rozmiaru zależnego od możliwości naszego hardware, tzn. nie powinno się wpisywać liczby przekraczającej fizyczną pojemność RAM. (W moim wypadku zwiększyłem ją z domyślnych 256 MB do 1512 MB).

10 Ciąg znaków <*nazwa_użytkownika*> oznacza login osoby pracującej na komputerze, np. konkretny adres może brzmieć */home/Przemek/sketchbook*.



OKNO PREFERENCJI PROGRAMU

Początkowo nasza praca nie będzie się różnić od pisania w notatniku. Wszystkie najbardziej znane skróty klawiaturowe, tj. *kopiuj* (*ctrl+c)*, *wklej* (*ctrl+v*), *cofnij* (*ctrl+z)*, *zapisz* (*ctrl+s*), itd., w dalszym ciągu obowiązują. Kilka dodatkowych, które warto poznać to *formatuj* (*ctrl+t)* i przede wszystkim *uruchom* (*ctrl+r).*

1.2 TYPY PRYMITYWNE

Uruchamiamy Processing klikając w ikonę programu dwukrotnie. Program[11] napisany w Processingu składa się z wyrażeń, które oddzielamy od siebie średnikiem ';'. Jest to podobne do kropki, oddzielającej od siebie zdania. Program jest więc rodzajem wypowiedzi językowej, której ogólny schemat możemy przedstawić następująco.

11 W tekście wymiennie będziemy używać terminu „skrypt" na określenie programu pisanego w *IDE* Processingu

<WYRAŻENIE_1>
<WYRAŻENIE_2>
(…)
<WYRAŻENIE_N>

<WYRAŻENIE_#> (gdzie **#** jest kolejnym numerem) oznacza zdanie napisane w języku programowania. Takie zdanie jest sensowne, jeśli zawiera instrukcję, które mogą zostać wykonane przez komputer. <**WYRAŻENIE_#>** w powyższym zapisie oznacza symbol nieterminalny, tzn. taki, który należy zastąpić innym symbolem. Nowy symbol może składać się ze słów należących do leksykonu używanego języka, których już nie będziemy mogli podmienić, tzw. terminali, lub z kolejnych symboli nieterminalnych, do których ponownie stosuje się procedurę podmiany. Najczęściej jest to połączenie obu tych kategorii. Zobaczymy to zaraz na przykładzie definicji zmiennej. Ogólna zasada zapisu reguł gramatycznych przyjęta w tym podręczniku opiera się na notacji Backusa-Naura[12] (BNF), gdzie każdy symbol nieterminalny jest reprezentowany przez pisany majuskułą ciąg znaków, umieszczonych pomiędzy ogranicznikami '**<**' oraz '**>**'. W poniższej definicji deklaracji zmiennej tylko średnik jest symbolem terminalnym, natomiast **<TYP>** oraz **<NAZWA>** wymagają dalszej podmiany na symbole końcowe.

<TYP> <NAZWA>;

Zmienna przechowuje wartość i jest rodzajem wirtualnego pudełka, do którego wkładamy dane. Umieszczenie zmiennej w kodzie rozpoczyna się od de-

12 Więcej na temat tej notacji – http://encyklopedia.helion.pl/index.php/Notacja_BNF

klaracji. Deklaracja składa się zawsze z dwóch elementów – słowa określającego typ zmiennej, które umieszczamy w miejsce **<TYP>**, oraz identyfikatora zmiennej, zastępującego symbol **<NAZWA>**. Nazwa musi być jednoznacznie przypisana do zmiennej – nie może być dwóch zmiennych o tej samej nazwie. Poniższy przykład pokazuje, jak to robimy. Utworzymy teraz zmienną, która będzie mogła jako wartość przechowywać ciąg znaków.

```
String jakiesSlowo;
```

Nazwa musi składać się z jednego słowa i nie może rozpoczynać się od cyfry, choć cyfry są dozwolone jako wewnętrzny lub ostatni znak. Ponadto należy posługiwać się podstawowym alfabetem łacińskim, tzn. nie używać polskich znaków diakrytycznych (*ą, ę, ..*). Dopuszczalnymi znakami specjalnymi są: znak podkreślający i znaki walut (*_, $, ¢, £, ¥*). Nie możemy używać *!, @, #, %, ^*, itd... Od znaku podkreślającego możemy rozpocząć nazwę, głównie przydaje się jednak do połączenia dwóch lub większej ilości słów w jeden ciąg znaków, tzn. **to_tez_jest_poprawna_nazwa_zmiennej**. Nazwa zmiennej powinna też nie pokrywać się z którymś ze słów kluczowych zarezerwowanych dla samej składni języka, np. nie możemy nazwać zmiennej słowem **void**. Poniżej przedstawiam listę tzw. *keywords*[13], będących podstawowym leksykonem języka – każde z tych słów pełni istotną rolę w tworzeniu programów. Nie wszystkich z nich będziemy używać w podręczniku, niektóre z nich są bowiem związane z bardziej zaawansowanymi zastosowaniami języka.

13 Źródło internetowe: http://download.oracle.com/javase/tutorial/java/nutsandbolts/_keywords.html

abstract	continue	for	new	switch
assert	default	goto	package	
boolean	do	if	private	this
break	double	implements	protected	throw
byte	else	import	public	throws
case	enum	instanceof	return	transient
catch	extends	int	short	try
char	final	interface	static	void
class	finally	long	strictfp	volatile
const	float	native	super	while

W Javie, a więc i w Processingu, jest przyjęta pewna stylistyczna konwencja w sytuacji, gdy nazwa jest zbitką kilku słów. Jak wcześniej zaznaczyłem, w nazwie nie może być przerwy (tzw. białego znaku). Rozwiązujemy ten problem następująco – zaczynamy nazwę od małej litery i pisząc w jednym ciągu, każde kolejne słowo rozpoczynamy dużą literą. Tak też zrobiłem w naszym przykładzie – **`jakiesSlowo`**. Ta konwencja oparta na bikapitalizacji nasi nazwę camelCase – przez luźną analogię do „wielbłądzich garbów".

Nazwy zmiennych powinny sugerować rolę, jaką odgrywają w kodzie. Unikajmy używania czasowników. Zmienne reprezentują dane, więc w zależności od jej rodzaju, wskazane jest posłużyć się rzeczownikiem, przymiotnikiem lub przysłówkiem.

Po zadeklarowaniu zmiennej możemy „umieścić" w niej wartość. Wartość ta musi być zgodna z zadeklarowanym typem zmiennej; np. jeśli wprowadziliśmy zmienną przechowująca ciąg znaków, to nie możemy przypisać do niej wartości liczbowej.

```
<NAZWA> = <WARTOŚĆ>;
```

Przypisanie dokonuje się za pomocą operatora przypisania, znaku '=', który wbrew pierwszemu skojarzeniu, nie jest w Processingu znakiem równości. Operator przypisuje operandowi stojącemu z jego lewej strony, wartość znajdującą się po jego prawej stronie. Używając uprzednio zadeklarowaną zmienną **jakiesSlowo** przypisanie wyglądać może następująco.

```
jakiesSlowo ="to jest wartość";
```

W przypadku zmiennej typu **String**, wartością jest dowolne wyrażenie złożone ze znaków alfabetu i zamknięte w cudzysłowie.

Możemy deklarację i przypisanie zapisać w jednej linijce.

```
<TYP> <NAZWA> = <WARTOŚĆ>;
String jakiesSlowo ="to jest wartość";
```

Jest to równoznaczne wcześniejszemu zapisowi, rozbitemu na dwa wyrażenia. Jeżeli chcemy zadeklarować kilka zmiennych tego samego typu, to możemy nazwy zmiennych wypisać w jednej linii rozdzielając je przecinkami.

String pierwszy, drugi, trzeci;

Tym sposobem zadeklarowaliśmy trzy zmienne typu **String**. Dopuszczalne jest nawet następująca składnia.

```
String pierwszy ="raz", drugi = "dwa", trzeci = "trzy";
```

Tu pozwoliliśmy sobie na jeszcze większą zwięzłość i w jednej linijce zawarliśmy zarówno deklarację jak i utworzenie instancji (ang. *instantiation*) kilku zmiennych tego samego typu. Instancja jest to po prostu zmienna, której została przypisana jakakolwiek wartość.

Zmienna, jak sama nazwa sugeruje, może zmieniać swoją zawartość. Operację przypisania możemy powtórzyć dowolną ilość razy, zachowując jednak zawsze zgodność typu zmiennej z typem przypisywanej wartości. Poniższy kod jest więc niepoprawny.

```
jakiesSlowo = "to jest wartość";
jakiesSlowo = 11;
```

Natomiast ten jest poprawny:

```
jakiesSlowo = "to jest wartość";
jakiesSlowo = "11";
```

String jest typem danych, który przechowywać może tylko literały łańcuchowe (napisy).

Poniższa tabela przedstawia podstawowe typy danych używanych w Processingu.

NAZWA	ROZMIAR	WARTOŚĆ MIN.	WARTOŚĆ MAKS.	PRZYKŁAD UŻYCIA	OPIS
boolean	zależny od platformy	false, 0	true, 1	**boolean** prawda = true;	Wartości logiczne, tj. prawda lub fałsz.
char	16 bitów	0	$2^{16}-1$	**char** znak_r = 'r';	Wartość znaku w kodowaniu UNICODE.
int	32 bitów	-2^{31}	$2^{32}-1$	**int** dziesiec = 10;	Liczba całkowita.
float	32 bitów	-3.4028235E38	3.4028235E38-1	**float** e = 2.7182817;	Komputerowa reprezentacja liczby rzeczywistej.
String	dowolny	nd.	nd.	**String** slowo = "Słowo";	Ciąg znaków tworzący tekst.

Z wyjątkiem **`String`** wszystkie wymienione w tabeli rodzaje zmiennych to typy prymitywne. Oprócz typów prymitywnych istnieją jeszcze typy złożone, będące podstawą programowania obiektowego. Typy prymitywne są w programowaniu tym, czym pierwiastki w chemii. **`String`** jest szczególnego rodzaju, należy jakby jednocześnie do tych dwóch światów.

Poszczególne typy będziemy wprowadzać stopniowo. W dalszej części rozdziału skupimy się na **`String`** i **`int`**. W połowie kursu nauczymy się tworzyć własne typy (klasy), które znacznie lepiej będą odpowiadać naszym oczekiwaniom. Na chwilę obecną najistotniejsze jest jednak zapamiętanie, w jaki sposób przebiega opisana wyżej procedura utworzenia instancji zmiennej. Składnia deklaracji i przypisania wartości dla wszystkich typów prymitywnych jest jednakowa.

1.3 URUCHAMIANIE PROGRAMU I DRUKOWANIE KOMUNIKATÓW

W Processingu uruchomienie napisanego programu jest bardzo proste. Wystarczy z menu z pozycji *Sketch* wybrać *Run*. Istnieje też skrót klawiaturowy – *ctrl* + *'r'* (lub Apple key pod Mac OSX). Możemy również kliknąć okrągły przycisk na pasku pomiędzy menu a polem edytora – tuż pod pozycją *File*. Po wykonaniu tego, program powinien zostać uruchomiony. Przede wszystkim zobaczymy nowe okno, jak na razie puste i jednolicie szare. Jest to miejsce, w którym wyświetlany będzie wizualny aspekt naszej pracy. Jednak nie mniej istotny może się okazać ten element IDE Processingu, który nazywa się powszechnie konsolą. Jest to czarny obszar poniżej edytora. Możemy

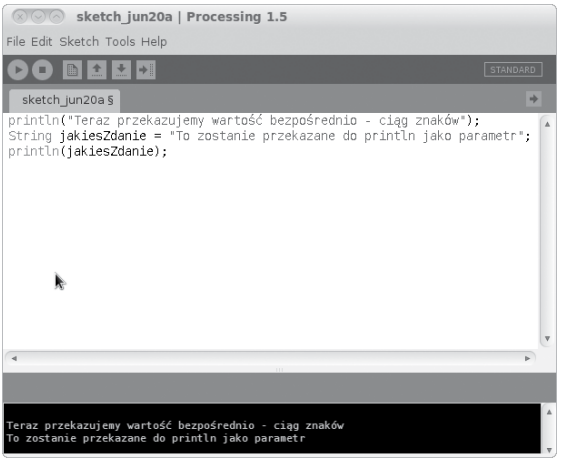
w nim wydrukować komunikat z wnętrza działającego programu. Inną funkcją konsoli jest system zgłaszania błędów. Zostanie on omówiony później.

Metoda to zbiór poleceń, często obliczeń, o unikalnym identyfikatorze (nazwie i sygnaturze), który wywołujemy jeżeli chcemy od programu ich wykonania. Poniżej pokażemy uogólnienie tej procedury.

```
<NAZWA>(<ARGUMENTY>);
```

Symbol nieterminalny **<NAZWA>** podmieniamy na nazwę metody. Na razie poprzestaniemy na używaniu metod, które zostały zdefiniowane przez twórców Processingu. Będziemy korzystać z gotowego leksykonu języka, jednak już niedługo poznamy reguły, które pozwolą nam rozszerzyć zakres dostępnych w nim pojęć. Symbol **<ARGUMENTY>** może być ciągiem zmiennych, wartości wpisanych bezpośrednio (tzw. literałów) lub złożeniem jednego i drugiego. Umieszczenie argumentu w wywołaniu metody należy rozumieć jako przekazanie informacji do wnętrza metody. Termin „informacja" jest tu niezwykle istotny, ponieważ w programowaniu wszystko jest informacją. Przekazanie argumentu metodzie polega na wpisaniu w nawias następujący po nazwie albo bezpośrednio wartości (w liczbie zero lub więcej), albo nazwy zmiennej z przypisaną wartością, która jest zgodna z typem ustalonym w sygnaturze metody. Jeśli argumentów jest więcej niż jeden, to rozdzielamy je przecinkami. Nie zawsze jest jednak wymagane przekazanie informacji. Metoda może być tak jednoznacznie zdefiniowana, że nie potrzebuje do wykonania żadnych dodatkowych danych. Wówczas po-

zostawiamy nawias pustym. Są też i takie metody, dla których argumenty przekazujemy opcjonalnie. Przykładem jest **`println`**[14], której używamy do wyświetlania tekstu w konsoli. Oczywiście wywoływanie jej bez argumentu jest pozbawione uzasadnienia



PRZYKŁAD PROGRAMU Z UŻYCIEM METODY **`PRINTLN`**

Należy tu zaznaczyć, że metodzie przekazujemy wartość. Zatem nie wystarczy, żeby zmienna była tylko zadeklarowana. Powinna również być stworzona jej instancja. Liczbę i rodzaj argumentów akceptowanych przez metodę określa tzw. sygnatura metody. Każda wartość przechowywana przez zmienną prymitywną dowolnego typu, może zostać zaprezentowana przez ciąg znaków (**`String`**), dlatego też sygnatura metody **`println`** jest tak tolerancyjna. Druga metoda o nazwie **`print`** różni się tym, że nie jest w niej dołączany znak

14 Nazwa metody wywodzi się od wyrażenia angielskiego *print line* (pl. drukuj linijkę)

przejścia do nowej linii na końcu drukowanego komunikatu. Napiszmy kilka wywołań **print**.

```
print(1);
print(", ");
print(2);
print(", ");
print(3);
```

Powinniśmy otrzymać w konsoli poniższy wpis.

```
1, 2, 3
```

1.4 OPERATORY I KOLEJNOŚĆ WYKONANIA

Z pojęciem operatora spotkaliśmy się już przy okazji znaku **'='.** Operator jest to specjalny znak, który reprezentuje operację, np. którąś z elementarnych operacji arytmetyki – dodawanie, odejmowanie, itd.. Operator zawsze potrzebuje co najmniej jednego operandu, tzn. obiektu na którym wykonuje daną operację, np. w przypadku dodawania operandami są liczby. W Processingu operandami mogą być wartości liczbowe podane wprost (literały) lub reprezentowane przez zmienne. Operację z użyciem operatora i operandu nazywamy wyrażeniem i najczęściej wynik operacji przypisujemy do zmiennej. Poniżej znajduje się przykładowy program.

```
int godzina, minuta;
godzina = 11;
minuta = 59;
print("Liczba minut, które upłynęły od północy: ");
println(godzina*60+minuta);
print("Ułamek z godziny, która właśnie upływa: ");
println(minuta/60);
```

Po uruchomieniu powinniśmy zobaczyć w konsoli wydrukowane dwie linijki.

```
Liczba minut, które upłynęły od północy: 719
Ułamek z godziny, która właśnie upływa: 0
```

Wynik z drugiej linijki może nas zaskoczyć – podzieliliśmy 59 przez 60, co powinno dać w przybliżeniu 0,983333333. Ta nieścisłość bierze się stąd, że Processing wykonał dzielenie dwóch liczb całkowitych, z którego wynik również jest liczbą całkowitą. Jeśli dzielenie daje liczbę rzeczywistą, zostaje ona zaokrąglona w dół. Możemy jednak obliczyć procent, który jest wartością całkowitą.

```
int procent = minuta*100/60;
print("Procentowa część godziny,
która właśnie upływa: ");
println(procent);
```

W pierwszej linii zadeklarowaliśmy zmienną i przypisaliśmy do niej wartość z wyrażenia matematycznego obliczającego procent. Teraz powinniśmy otrzymać następujący wynik.

```
Procentowa część godziny, która właśnie upływa: 98
```

Powinniśmy pamiętać o kolejności wykonywania operacji matematycznych – mnożenie i dzielenie mają pierwszeństwo przed dodawaniem i odejmowaniem, tzn. w wyrażeniu **`4*2+3*5`** dodane zostaną iloczyny. Operacje o tej samej regule pierwszeństwa (np. "**`*`**" i "**`/`**", "**`+`**" i "**`–`**") są wykonywane od lewej do prawej – **`2*8/6`** daje **`3`**. Jeśli nie jesteśmy pewni interpretacji kodu lub też chcemy wymusić kolejność wykonania, możemy ustrukturyzować wy-

rażenie poprzez użycie nawiasów, np. wynik z działania **2*(8/6)** równy jest **2**, a nie jak wcześniej **3**.

Szczególnym przypadkiem są operacje na String. Dość oczywiste jest, że nie możemy wykonać operacji matematycznych na zmiennych tego typu. Możemy jednak zastosować operator dodawania. Była o tym mowa przy okazji omawiania metody **println**. Użycie '**+**' nie jest w tym wypadku działaniem arytmetycznym, w wyniku bowiem dostajemy nowy ciąg znaków, będący złączeniem ciągów na których dokonano operacji. Taką operację na ciągach znaków nazywamy kontkatenacją. Oto przykład jej użycia.

```
String pierwszy = "program", drugi = "owanie";
String trzecie = pierwszy + drugi;
print(trzecie);
```

Nie będzie niespodzianką, jeśli w oknie konsoli otrzymamy jedno słowo:

```
programowanie
```

Jak wspomniałem wcześniej, ten typ zmiennej jest szczególnego rodzaju. Oprócz dodawania możliwe jest wiele innych operacji jak np. zamiana małych liter na duże, przestawienie kolejności, itd.

2 TWORZENIE METOD

2.1 LICZBY RZECZYWISTE

Do tej pory dla operacji matematycznych używaliśmy typu zmiennej **int**. Szybko zauważyliśmy, jakie wiążą się z tym ograniczenia – nie możemy wykonywać poprawnych obliczeń z użyciem liczb rzeczywistych. Staje się to możliwe po wprowadzeniu kolejnego typu liczbowego o nazwie **float**. Mechanizm utworzenia instancji zmiennej tego typu wygląda analogicznie do poznanego wcześniej.

```
float pi = 3.14159;
```

W krajach anglojęzycznych zapis liczby rzeczywistej różni się od konwencji przyjętej w Polsce tym, że część ułamkową oddziela się kropką (‘**.**’) a nie przecinkiem (‘**,**’). Stąd też taki zapis w Processingu. Taki zapis jest więc niedopuszczalny:

```
float e = 2,71828;
```

W Processingu powinniśmy podać powyższą wartość stosując zapis z kropką:

```
float e = 2.71828;
```

Zmienna **float** może przechowywać również wartości całkowite. Porównajmy dwie linijki wydrukowane w konsoli po wykonaniu kodu z poniższego listingu.

```
int a = 1;
float b = 1;
println(a);
print(b);
```

```
1
1.0
```

Typ danych **float** traktuje liczby całkowite z większą precyzją niż **int** i przechowuje część ułamkową liczby, nawet jeśli wynosi ona 0. Powyższe przypisanie liczby całkowitej do **float** nie zwraca komunikatu błędu, ale powinniśmy jednak dla większej jasności w kodzie przypisywać wartości do **float** w następujący sposób.

```
float c = 1.0;
```

Jeżeli jako wartość zmiennej przypisujemy wynik z wyrażenia, np. z dzielenia, ważne jest zwrócenie uwagi na to, czy operacja nie jest wykonywana na liczbach całkowitych. Wynik z wyrażenia będzie wartością zmiennoprzecinkową jedynie pod warunkiem, że w wyrażeniu znajdzie się choć jeden **float**. Porównajmy dwa komunikaty poniżej.

```
println(1/3);
println(1.0/3);
```

```
0
0.33333334
```

2.2 KONWERSJA INT NA FLOAT

Konwersja wartości **int** na **float** odbywa się bez żadnej straty. Jednak ze względu na różnicę w precyzji pomiędzy **float** i **int**, czasami przy zamianie w drugą stronę należy się liczyć ze znacznym przekłamaniem. Wartości są bowiem zaokrąglane w dół. Na przykładzie wygląda to następująco.

```
int nieprecyzyjnePi = int(3.14159);
int nieprecyzyjneE = int(2.71828);
```

W drugiej linijce w wyniku konwersji straciliśmy prawie **1/4** wartości liczby. Wydaje się, że mając do wyboru dwie precyzje, lepiej jest pracować z dokładniejszym typem danych. Nie zawsze jest to jednak możliwe; np. wtedy gdy wynik jakiejś operacji zmuszeni jesteśmy przekazać metodzie, która przyjmuje tylko argumenty typu **`int`**.

2.3 FUNKCJE MATEMATYCZNE

Mitem jest, że umiejętność programowania musi wiązać się z ponadprzeciętną znajomością matematyki. Oczywiście znajomość matematyki wyższej (analiza matematyczna, algebra liniowa, matematyka dyskretna) może okazać się przydatna, ale wszystko zależy głównie od charakteru programów, które chcemy pisać. W tym podręczniku nie wyjdziemy poza matematykę na poziomie szkoły średniej. Warto jednak zainteresować się matematyką dla niej samej. Formalizm matematyki jest istotny nie tylko dla programowania, ale również dla myślenia w ogóle[15].

Wyrażenie **`sin(PI)`** powinno wydać się nam znajome z lekcji matematyki. Tak bowiem przywykliśmy zapisywać funkcję trygonometryczną sinus dla kąta o wartości przybliżonej **`3.14159`** wyrażonej w radianach. Wywołanie tej funkcji w Processingu wygląda identycznie. Co więcej, nie musimy pamiętać wartości liczby **`pi`**, ponieważ jest dostępna za pośrednictwem stałej o nazwie **`PI`**.

15 Sam źródłosłów terminu „matematyka" może to zasugerować. Termin ten pochodzi mianowicie od greckiego *mathēmatikḗ*, co tłumaczymy jako „poznanie", „umiejętność".

```
println(PI);
float wartoscSinusa = sin("pi = "+PI);
print(wartoscSinusa);
```

```
-8.742278E-8
```

Wydruk w drugiej linijce może wyglądać zaskakująco, bowiem wartość sinusa dla kąta **180** stopni (inaczej pi radianów) powinna wynieś **0**. W rzeczywistości uzyskany wynik jest bardzo zbliżony do **0**. Wynik jest liczbą rzeczywista zapisaną według notacji naukowej – przy użyciu znaku liczby, mantysy, podstawy systemu liczbowego oraz potęgi.[16] Może wydawać się to skomplikowane, ale w gruncie rzeczy jest elementarne – w naszym konkretnym przypadku wynik, **-8.742278E-8**, to inaczej liczba **-8,742278** pomnożona przez **10** do potęgi **-8**, co daje w przybliżeniu wartość **-0,000000087**, a więc w ostateczności wynik bardzo bliski zeru.

Oprócz sinusa, mamy do dyspozycji również inne funkcje trygonometryczne, takie jak cosinus, tangens, jak również przeciwne do nich – arcus sinus, arcus cosinus, arcus tangens. Trud ich implementacji (czyli zaprogramowania procedur obliczających te wartości) został nam oszczędzony przez twórców języka.

W Processingu zostały zaimplementowane również takie funkcje proste jak funkcja potęgowa (x^y), logarytm naturalny (ln(x)), funkcja wykładnicza (e^x), pierwiastkowanie, i wiele innych. Możemy teraz wprowadzić do terminologii języka programowania rozróżnienie na metody i funkcje. Metoda jest to funkcja niezwracająca żadnego wyniku, który moglibyśmy następnie przypisać do zmiennej. O funkcji możemy

16 Podstawa matematyczna tego zapisu jest omówiona m.in. na https://pl.wikipedia.org/wiki/Liczba_zmiennoprzecinkowa

powiedzieć odwrotnie – jest metodą, która zwraca wartość. W odniesieniu do funkcji matematycznych jest to w miarę oczywiste – dokonuję obliczenia np. pierwiastka, ponieważ interesuje mnie wynik – czyli to, co przez operację spierwiastkowania otrzymaliśmy na wyjściu. Z matematycznego punktu widzenia nie nazwiemy funkcją operacji, która nie daje żadnego wyniku, chociażby zera. Wywołując funkcję pierwiastkującą zakładamy, że algorytm (tzn. sekwencja procedur) da nam ostatecznie poprawny wynik i nie interesują nas wszystkie szczegóły operacji.

Pamiętamy o problemie utraty precyzji przy zamianie liczby zmiennoprzecinkowej na całkowitą. Jednak poprzez zastosowanie odpowiednich funkcji możemy zadecydować o rodzaju zaokrąglenia. Porównajmy trzy konwersje.

```
float f = 0.5;
int wDol = floor(f);
int zaokraglone = round(f);
int wGore = ceil(f);
println(wDol);
println(zaokraglone);
print(wGore);
```

```
0
1
1
```

W pierwszym przypadku skorzystaliśmy z funkcji **floor**, która zaokrągla w dół każdą przekazaną wartość, a więc w działaniu nie różni się od zwykłej konwersji typów. Kolejna funkcja – **round** zaokrągla w zależności od wartości ułamkowej – jeśli jest większa lub równa **0,5**, liczba zaokrąglona zostanie w górę. Wreszcie ostat-

nia – **ceil** zaokrągla wszytko w górę. Nazwy funkcji sugerują ich zachowanie – **floor** (pl. podłoga), **ceil** (pl. sufit).

2.4 SKŁADNIA DEFINICJI METODY

Dotychczas korzystaliśmy z gotowych metod – tworzyliśmy wyrażenia poprzez wywołanie i przekazanie argumentu. Jeżeli będziemy chcieli stworzyć własną metodę, będziemy musieli przestrzegać ogólnych reguł składni definiowania metod.

```
void <NAZWA>(<SYGNATURA>) {
    <WYRAŻENIE_1>
    (…)
    <WYRAŻENIE_N>
}
```

Definicję metody zaczynamy od słowa kluczowego **void**. Następnie umieszczamy nazwę. Reguły formalne jej utworzenia są identyczne jak dla nazw zmiennych. O ile jednak nie powinniśmy używać czasowników w nazwach tych drugich, to jest to jak najbardziej wskazane w nazwach metod. Często też można dla większej czytelności zastosować zbitkę czasownika i rzeczownika, np. **rysujKolo**, **ustawDatę**, itp. Symbol nieterminalny **<SYGNATURA>** w definicji zamieniamy na ciąg oddzielonych od siebie przecinkami deklaracji zmiennych. Ich ilość i typ mogą być dowolne. Sygnatura decyduje o ilości i typie przekazywanych metodzie danych na wejściu. Przestrzeń zawierająca wyrażenia w nawiasach klamrowych to blok kodu. W definicji metody nosi on nazwę ciała metody. Ciało może być dowolnie długie, nie ma ograniczeń co do liczby

umieszczonych w nim wyrażeń. Wyrażeniami mogą być deklaracje zmiennych, operacje przypisania, wywołania innych metod oraz nie poznane jeszcze przez nas wyrażenia boolowskie i iteracje. Nie możemy natomiast zagnieździć w ciele metody definicji innej metody. W praktyce lepiej nie tworzyć zbyt długich bloków kodu. W wypadku skomplikowanego problemu stosowne jest definiowanie kilku krótszych metod zawierających tylko te procedury, których efekt jest jasno zasugerowany przez nazwę metody. Należy zwrócić uwagę, że nie umieszczamy średnika po nawiasie klamrowym zamykającym ciało.

Zanim zaczniemy wykorzystywać tę strukturę do tworzenia całkiem nowych metod, wprowadzimy do naszego programu definicję bardzo ważnej metody **`setup`**.

2.5 METODA SETUP

Do tej pory instrukcje w programie były wykonywane linijka po linijce. Program był czytany od góry do dołu. Ten tryb jest nazywany w Processingu podstawowym i ma wiele zastosowań. Będziemy go czasami używać w celu szybkiego zaprezentowania jakiegoś zagadnienia. Nie pozwala on jednak na umieszczanie w kodzie definicji nowych metod. Jesteśmy ograniczeni do używania tych zdefiniowanych przez twórców języka. Jeżeli chcemy utworzyć własną metodę, musimy przejść do trybu pracy zwanego trybem ciągłym (*ang. continuous*). Przejście do tego trybu następuje po wprowadzeniu do naszego kodu następujące wyrażenie.

```
void setup(){
}
```

Może tego jeszcze nie widać, ale jest to duży krok naprzód. Przede wszystkim przejście do tego trybu otwiera nam drogę do struktur wcześniej niedostępnych. Słów kilka o samej metodzie **setup**. Jest to metoda, która zostanie wywołana automatycznie jako pierwsza na początku działania programu. Nie ma więc potrzeby wywołania jej w kodzie. Na początek możemy przenieść do niej niedawno napisany kod.

```
void setup(){
        float f = 0.5;
        int wDol = floor(f);
        int zaokraglone = round(f);
        int wGore = ceil(f);
        println(wDol);
        println(zaokraglone);
        print(wGore);
}
```

Tym sposobem zdefiniowaliśmy naszą pierwszą, nie do końca własną, metodę. Efekt w konsoli będzie identyczny jak w trybie podstawowym. Należy zapamiętać, że nie jest możliwe połączenie obu trybów. W trybie ciągłym możemy wywołać metody tylko z ciała innej metody. Program w trybie ciągłym składa się przede wszystkim z definicji metod. Wszystko co nie jest deklaracją, utworzeniem instancji oraz przypisanie wartości do zmiennej musi być napisane w ciele jakiejś metody. Użycie operatora przypisania poza metodami jest możliwe wyłącznie pod warunkiem, że w operacji korzystamy co najwyżej z jednoznacznych wartości (literałów), instancji zmiennych, wyrażeń arytmetycznych (w przypadku danych liczbowych) lub konkatenacji (w przypadku **String**). Wykonanie programu w trybie ciągłym

rozpoczyna się od tych wyrażeń. Następnie program przechodzi do metody **setup**. Dalszy proces wykonania programu jest określany kolejnością wywołań metod. W ten sposób program może się cofać do linijek kodu powyżej miejsca w którym aktualnie się znajduje. Metoda **setup** jest wywołana tylko raz i nie możemy jej wywołać w żadnym miejscu kodu. Możemy ją tylko definiować.

Jest przyjęte, że w definicji **setup** określamy podstawowe parametry naszego programu takie jak rozmiary okna, anty-aliasing czy częstotliwość odświeżania. Zmieniamy rozmiar okna programu wywołaniem metody **size**, która przyjmuje dwa argumenty typu **int**.

```
void setup(){
        size(200,200);
}
```

Programy, które nie mają implicite określonych wymiarów okna uruchamiać się będą z domyślnym wymiarem 100 na 100 pikseli. Obecność tego okna, pomimo faktu że większość tego co piszemy jest adresowane do konsoli, bierze się stąd, że każdy skrypt pisany w Processingu jest ostatecznie apletem Java, czyli rodzajem tzw. programu okienkowego. Możemy jednak, w ramach ćwiczenia, w bardzo prosty sposób stworzyć symulację konsoli w głównym oknie programu.

Zaczniemy od stworzenia zmiennej typu **String**, która będzie przechowywać całość wyświetlanego tekstu. Przypisaliśmy jej pusty ciąg znaków. Zmienne, które deklarujemy na zewnątrz metod, znajdują się na tym samym poziomie, co one. Będziemy mogli takich zmiennych użyć wewnątrz każdej z metod dzięki tej samej zasadzie, która pozwala na używanie przez metody

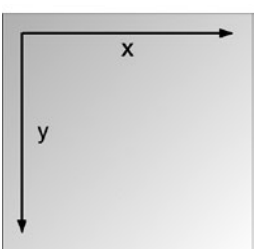
innych metod. Mówimy też, że takie zmienne mają zasięg globalny dla odróżnienia od zmiennych o zasięgu lokalnym, czyli zadeklarowanych wewnątrz bloku kodu i dostępnych tylko w jego granicach oraz poniżej linijki, w której miała miejsce deklaracja. Następnie definiujemy metodę **setup**, w której ustalamy wymiary okna. Metoda, która ustala kolor drukowanej czcionki to **fill**. Argument **255** oznacza biały. O kolorze będziemy mówić szerzej w jednym z kolejnych rozdziałów.

```
String napis = "";
void setup(){
        size(320,240);
        fill(255);
}
```

Napiszemy teraz definicję naszej własnej metody. Zaczynamy od słowa kluczowego **void**. Po nim następuje nazwa. Metodę nazwiemy **drukuj**. Dobrze jest przypisać metodzie nazwę zgodną z czynnością przez nią wykonywaną. Nadaje to spójności znaczeniowej programom. Nasza metoda będzie wymagała przekazania jednej zmiennej typu String, dlatego w nawiasie po nazwie wpiszmy jej deklarację. Należy pamiętać, że w sygnaturze umieszczamy tylko deklaracje. Przypisanie wartości odbywa się przy wywołaniu metody. Nazwiemy tą zmienną wiersz. Definicja wygląda następująco.

```
void drukuj(String wiersz){
        napis = napis + wiersz + "\n";
        background(0);
        text(napis, 12, 12, width-24, height-24);
}
```

Przypisywanie wartości do zmiennej jest wykonywane od lewej strony, tzn. w pierwszej kolejności jest wykonywane wyrażenie stojące po prawej stronie operatora przypisania. Następnie wartość zwrócona przez to wyrażenie jest przypisywana do zmiennej po lewej stronie. Dzięki temu możliwy jest sposób przypisania, którego użyliśmy w pierwszej linijce – najpierw dodaliśmy do starej wartości zmiennej napis wartość zmiennej wiersz oraz znak nowej linii, a dopiero później wynikiem tej operacji zastąpiliśmy poprzednią wartość **napis**. Stąd też po każdym kolejnym wywołaniu tej metody, ciąg znaków przechowywany przez tę zmienną będzie się wydłużał o nową linię, reprezentowaną przez zmienną **wiersz**. Reszta kodu to wywołanie metody **background**, która barwi tło na kolor określony argumentem (0 oznacza tu czarny) oraz wywołanie metody **text,** która umieszcza pole tekstowe na płótnie okna. Metoda **text** wymaga pięciu argumentów w następującej kolejności – ciąg znaków, który powinien zostać wyświetlony, współrzędne **x, y** oraz szerokość i wysokość pola tekstowego. Słowa **width** i **height** są zmiennymi wbudowanymi w język, które przechowują szerokość i wysokość okna programu. Należy tu też wspomnieć, że wartości współrzędnych na płótnie Processingu rosną w prawo i w dół od punktu zerowego znajdującego się w lewym górnym rogu okna.

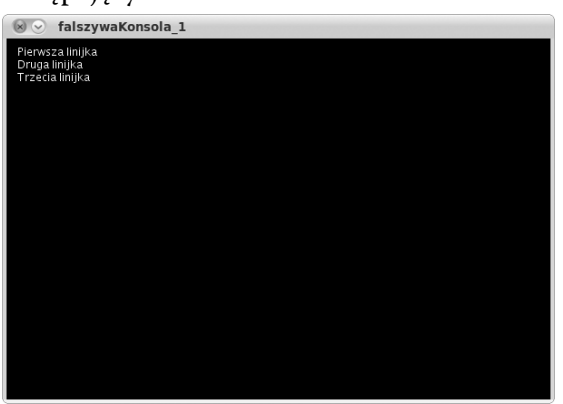


UKŁAD WSPÓŁRZĘDNYCH W PROCESSINGU

Mając zdefiniowaną metodę, możemy ją wywołać z wnętrza metody **setup**, przekazując jej kilka linijek tekstu, tak jakbyśmy wywoływali processingową metodę **println**. To, że definicja **drukuj** jest umieszczona poniżej metody **setup** nie ma większego znaczenia. Napiszmy następujący program.

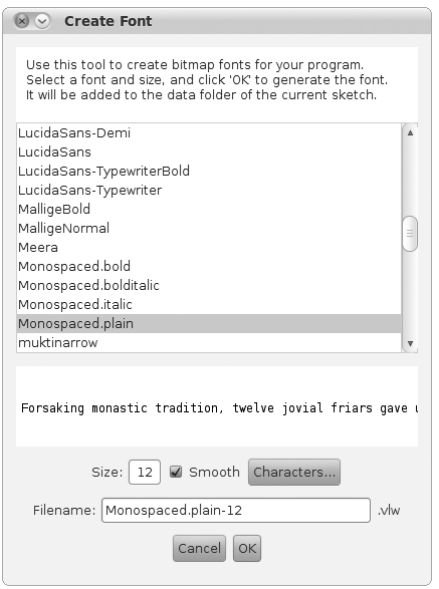
```
String napis = "";
void setup() {
        size(320, 240);
        fill(255);
        drukuj("Pierwsza linijka");
        drukuj("Druga linijka");
        drukuj("Trzecia linijka");
}
void drukuj(String wiersz) {
        //(…)
}
```

Po uruchomieniu programu powinniśmy uzyskać następujący widok.



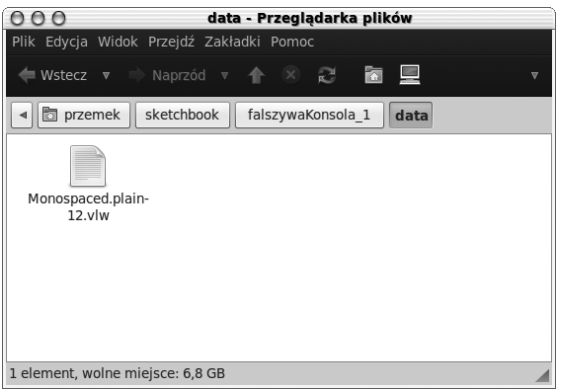
OKNO URUCHOMIONEGO PROGRAMU

Jeśli porównamy go z wydrukiem, może się nam rzucić w oczy różnica czcionek. W polu tekstowym na płótnie widzimy czcionkę bezszeryfową, natomiast wydruk w konsoli operuje krojem technicznym – tym samym, którego używamy w edytorze. Możemy to zmienić następująco. W menu wybieramy *Tools -> Create Font...*. Uruchomi to narzędzie umożliwiające stworzenie i dodanie czcionki do naszego programu. Wybierzmy z listy dostępnych czcionek taką, która ma w nazwie słowo „monospace". W moim przypadku jest to *Monospaced.plain*. Ustawmy rozmiar czcionki na 12. W polu o nazwie *Filename* zostanie automatycznie zaproponowana nazwa pliku, którą możemy zachować lub dowolnie zmodyfikować.



OKNO NARZĘDZIA DO TWORZENIA CZCIONEK

Po kliknięciu przycisku *ok* czcionka zostanie zachowana na dysku jako plik o podanej nazwie i rozszerzeniu *vlw*. Możemy to sprawdzić następująco. Wybierzmy z menu *Sketch -> Show Sketch Folder*. Systemowa przeglądarka plików powinna otworzyć folder, w którym znajduje się plik naszego programu. Czcionka została zapisana w katalogu o nazwie *data*. Katalog ten powstał w chwili utworzenia czcionki.



ZAWARTOŚĆ FOLDERU *DATA*

Przy zapisywaniu projektów struktura plików tworzy się automatycznie w taki sposób, że zapisywany plik zostaje umieszczony wewnątrz katalogu o tej samej nazwie. Projekty zazwyczaj zapisujemy w katalogu o nazwie *sketchbook*. Jeśli się przyjrzymy jego strukturze, to zobaczymy, że w rzeczy samej zawiera on katalogi o nazwach edytowanych przez nas plików. Zapisywanie projektów w tym miejscu jest wygodne, ponieważ dowolny z nich możemy następnie otworzyć klikając w jego nazwę na liście dostępnej pod pozycją *Sketchbook ->File*.

Żeby zrobić użytek ze stworzonej czcionki, musimy napisać kilka dodatkowych linijek kodu w definicji **setup**. Zaczniemy od stworzenia zmiennej lokalnej typu **Pfont**, w której umieścimy czcionkę. Dla przypomnienia zmienna lokalna oznacza taką zmienną, do której dostęp jest możliwy tylko w obrębie tego samego bloku kodu, w którym została zadeklarowana. Typ **PFont** jest typem bardziej złożonym niż te poznane dotychczas, ale podstawowa zasada pozostaje taka sama jak w przypadku poznanych do tej pory typów prymitywnych – zmienna będzie przechowywać przypisane do niej dane. Do typu **PFont** przypisujemy czcionki w formacie *vlw*. poprzez wywołanie funkcji **loadFont** z argumentem **String**, będącym nazwą pliku czcionki przechowywanej na dysku twardym. Zrobimy tak dla zmiennej o nazwie **czcionka**. Następnie poprzez wywołanie metody **textFont**, ze stworzoną przed chwilą zmienną jako jej argumentem, sprawimy, że od tego momentu do każdego pola tekstowego będzie stosowana ta czcionka. Ostatecznie po uruchomieniu następującego kodu ponownie zobaczymy okno, które jeszcze wierniej naśladuje konsolę.

```
String napis = "";
void setup() {
        size(320, 240);
        PFont czcionka =
                loadFont("Monospaced.plain-12.vlw");
        textFont(czcionka);
        fill(255);
        drukuj("Pierwsza linijka");
        //(…)
}
void drukuj(String wiersz) {
        //(…)
}
```

2.6 RYSOWANIE W PROCESSINGU

Processing jest z założenia środowiskiem programistycznym sprzyjającym tworzeniu struktur wizualnych, stąd też obecność wielu metod przeznaczonych wyłącznie do pracy graficznej. Oto kilka z nich.

Metoda **point** pozwala na umieszczenie punktu na płótnie. Przejmuje ona dwa argumenty liczbowe – współrzędną **x** i **y**, np.

```
point(50,50);
```

Metoda **line** wyrysowuje linie pomiędzy dwoma punktami. Zakładając, że chodzi nam o rysunek w dwóch wymiarach, musimy przekazać cztery argumenty – dwa jako współrzędne pierwszego punktu i dwa dla drugiego.

```
line(0,0,100,100);
```

Są też metody, które rysują podstawowe figury geometryczne takie jak prostokąt czy okrąg. W przypadku **rect** rysującego prostokąt, przekazujemy cztery argumenty – współrzędne lewego górnego rogu oraz szerokość i wysokość figury. Metoda rysująca okrąg lub elipsę – **ellipse**, potrzebuje również czterech współrzędnych: środka oraz średnicę w poziomie i w pionie.

```
rect(25, 25, 50, 50);
ellipse(0, 0, 100, 100);
```

Kolejne to **triangle** i **quad**. Rysują one odpowiednio trójkąt oraz dowolny czworokąt. W ich przypadku argumenty to współrzędne wierzchołków – trzech w **triangle** i czterech w **quad**.

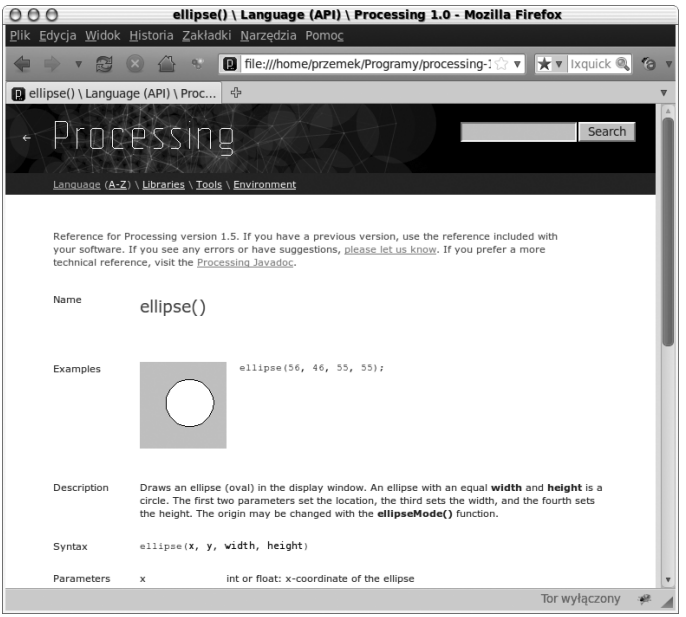
```
triangle(50, 0, 0, 100, 100, 100);
quad(35, 35, 65, 35, 65, 65, 35, 65);
```

Szczegółowe omówienie zagadnień związanych z rysowaniem przekroczyłoby format tego podręcznika. Processing posiada bardzo obszerny system pomocy. Mamy do niego dostęp poprzez pozycję *Help* w menu i wybranie interesującego nas zagadnienia. Najczęściej jednak będziemy korzystać z pomocy na bieżąco, chcąc zasięgnąć szczegółowych informacji o konkretnych elementach kodu pisanego w danym momencie. Jeśli chcielibyśmy dowiedzieć się więcej o metodzie **`ellipse`**, to najpierw możemy napisać jej nazwę w edytorze a następnie zaznaczyć ją, wcisnąć prawy przycisk myszki (pod Mac OS X – wcisnąć myszkę jednocześnie z klawiszem *cmd*) i wybrać z menu kontekstowego pozycję *Find in Reference*, tak jak przedstawia to ilustracja poniżej.

MENU KONTEKSTOWE W EDYTORZE

Poskutkuje to uruchomieniem przeglądarki internetowej, w której oknie pojawi się strona z do-

kumentacją wybranej metody. Pomoc Processingu ma postać strony internetowej zapisanej na naszym dysku w miejscu instalacji programu. Aby ją przeglądać, nie musimy być podłączeni do internetu.



PRZYKŁADOWA DOKUMENTACJA WYRAŻENIA

3 PRZEPŁYW STEROWANIA

3.1 WYRAŻENIA WARUNKOWE

Przepływ sterowania to zestaw elementów składniowych języka, które pozwalają programiście na podejmowanie decyzji o kolejności wykonania bloków kodu. Podstawową instrukcją sterującą, spotykaną w niemal wszystkich językach programowania, jest **`if`** (*ang. jeśli*). W Processingu składnia tej konstrukcji wygląda następująco.

```
if (<WARUNEK>){
    <WYRAŻENIA>
}
```

`<WARUNEK>` jest formą wyrażenia, które po ewaluacji zwraca wartość typu logicznego (boolowskiego – od nazwiska angielskiego matematyka Georga Boola). Typ ten deklarujemy słowem kluczowym **`boolean`**. Zmienna boolowska może przechowywać jedną z dwóch wartości logicznych – **`true`** (prawda) lub **`false`** (fałsz). Wyrażenia zawarte w nawiasach klamrowych zostaną wykonane jedynie w sytuacji, gdy wartość wyrażenia w nawiasie po **`if`** ma wartość **`true`**, innymi słowy – wtedy i tylko wtedy, jeśli warunek jest spełniony.

`<WARUNEK>` konstruujemy za pomocą operatorów porównania. Są to operatory dwuargumentowe, porównujące ze sobą operandy. Operandy nie muszą to być koniecznie zmiennymi typów liczbowych, mogą to być również dwa ciągi znaków lub też inne typy, które można ze sobą porównać którymkolwiek z operatorów porównania, w szczególności zmienne boolowskie. W przypadku zmiennych **`String`** tylko dwa ostatnie z podanych w tabeli operatorów mogą zostać zastosowane.

OPERATOR PORÓWNANIA		OPIS
>	x > y	true jeśli x jest większe od y, inaczej false
<	x < y	true jeśli x jest mniejsze od y, inaczej false
>=	x >= y	true jeśli x jest większe lub równe y, inaczej false
<=	x <= y	true jeśli x jest mniejsze lub równe y, inaczej false
==	x == y	true jeśli x jest równe y, inaczej false
!=	x != y	true jeśli x jest różne od y, inaczej false

Poniższy kod umieszczony w **setup** wydrukuje do konsoli słowo **true**.

```
int x = 2;
if(x>0){
        print(x>0);
}
```

Tworząc łańcuch bloków **if**, możemy stworzyć logiczną alternatywę wykonania kodu. W poniższym przykładzie zostanie wykonana druga klauzula.

```
int x = 2;
if(x>2){
        print("x>0");
}
if(x==2){
        print("x==2");
}
```

Choć **if** daje nam możliwości sporej manipulacji przepływem, poleganie wyłącznie na tej konstrukcji bywa często niepraktyczne. W niektórych przypadkach

bylibyśmy wręcz niezdolni do skonstruowania koniecznych warunków logicznych. Uzupełnieniem **if** jest **else.**

```
if (<WARUNEK>){
    <WYRAŻENIA_1>
} else {
    <WYRAŻENIA_2>
}
```

Program przejdzie do bloku po **else** w wypadku, gdy wcześniejszy warunek nie zostanie spełniony. Jeśli dodamy jeszcze jedną instrukcję sterującą, będziemy w posiadaniu pełnej składni pozwalającej na logiczne pokierowanie przepływem programu.

```
if (<WARUNEK_1>){
    <WYRAŻENIA_1>
} else if (<WARUNEK_2>) {
    <WYRAŻENIA_2>
} else {
    <WYRAŻENIA_3>
}
```

Przy takiej strukturze blok kodu skorelowany z warunkiem nr. **2**, zostanie wykonany jedynie w sytuacji, gdy równocześnie jego ewaluacja przyniesie wynik **true** oraz gdy nie został spełniony warunek nr.**1**. Kod przy trzeciej możliwości, zostanie wykonany wyłącznie jeśli nie udało się to w żadnym z poprzednich przypadków.

Tak jak to zrobiliśmy w przykładzie z **if**, wyrażenia warunkowe możemy łączyć w szeregi budując bardziej skomplikowane struktury logiczne. Operatory te są dwuargumentowe, co oznacza, że jednorazowo możemy porównać tylko dwie wartości. Jeżeli nasze reguły zawie-

rają więcej składowych, musimy zagnieżdżać jeden warunek w drugim.

```
if(i > 0) {
        if(i < 10) {
                print("i jest większe"
                      + "od 0 i mniejsze od 10");
        }
}
```

Warunkiem w instrukcjach sterujących, niekoniecznie musi być wyrażenie skonstruowane z operatora logicznego i operandów. Tę rolę spełniać mogą również zmienne boolowskie. Wówczas przepływ jest uzależniony od wartości logicznej przechowywanej w zmiennej.

3.2 REKURENCJA

Rekurencja jest jednym z najbardziej fascynujących zagadnień nie tylko w programowaniu ale również w wielu innych dyscyplinach, nie wyłączając sztuk pięknych. Najogólniej rzecz ujmując, rekurencja jest odwołaniem się pewnej całości do samej siebie. Obrazowo – uzyskamy idealny przykład rekurencji stawiając naprzeciw siebie dwa lustra. W programowaniu rekurencja występuje wtedy, gdy we wnętrzu metody znajduje się wywołanie jej samej. Żeby uniknąć tzw. regresji w nieskończoność (w przypadku dwóch luster mamy właśnie do czynienia z czymś takim – odbicie odbicia, które jest odbiciem odbicia, itd...), potrzebujemy mechanizmu, który pozwoli nam na przerwanie egzekucji kodu w odpowiednim momencie. Umieszczając w ciele metody słowo kluczowe **`return`** sprawimy, że po napotkaniu tej komendy przez program nastąpi natychmiastowe wyjście z metody, z pominięciem tego, co zdefiniowane będzie w linijkach następujących po wyrażeniu ze słowem

return. Ten mechanizm jest często używany w przypadkach, gdy chcemy sprawdzić poprawność przekazanych metodzie argumentów i zabezpieczyć program przed wykonywaniem operacji z użyciem danych, które nie powinny zostać przekazane. Pokazuje to poniższy przykład.

```
void tylkoDodatnie(int i){
        if(i < 0){
                return;
        }
        println(i +"jest liczbą parzystą");
}
```

Program przejdzie do linijki z wywołaniem **println** jedynie w sytuacji, gdy warunek **if(i<0)** zostanie spełniony, tzn. jeśli argument przekazany metodzie nie będzie liczbą ujemną – w przeciwnym razie wykonanie bloku warunkowego poskutkuje natychmiastowym wyjściem z metody poprzez wywołanie **return**. Zanim przejdziemy do właściwej rekurencji, przedstawimy ogólny schemat metody rekurencyjnej.

```
void <NAZWA>(<SYGNATURA>){
    if (<WARUNEK_WYJŚCIA>){
          <WYRAŻENIA>
          return;
    }
    <WYRAŻENIA>
    <NAZWA>(<ARGUMENTY>);
}
```

Poniższa definicja metody pokazuje jak bezpiecznie użyć rekurencji. Konieczne jest umieszczenie mechanizmu wyjścia jeszcze przed wywołaniem samej siebie.

```
void drukujNLinijek(int n){
        if(n==0){
                return;
        }
        println("Linijka nr. "+n);
        drukujNLinijek(n-1);
}
```

Po umieszczeniu w **setup** wywołania tej metody z argumentem **10**, powinniśmy otrzymać następujący wydruk.

```
Linijka nr. 10
Linijka nr. 9
Linijka nr. 8
Linijka nr. 7
Linijka nr. 6
Linijka nr. 5
Linijka nr. 4
Linijka nr. 3
Linijka nr. 2
Linijka nr. 1
```

Działanie powyższego kodu można opisać następująco. Najpierw wywołujemy metodę z argumentem **10**. Ponieważ ta wartość jest różna od **0**, blok kodu w wyrażeniu warunkowym nie jest wykonywany. Program omija ten blok, wywołuje metodę **println**, a w następnej linijce odwołuje się do metody, w której wnętrzu się aktualnie znajduje, lecz z argumentem zmniejszonym o **1**. W kolejnym wykonaniu dzieje się identycznie. Dopiero za **10** razem, gdy przekazany argument jest równy **0**, warunek **if** zostaje spełniony, program wchodzi do bloku kodu z wy-

wołaniem **return**, co skutkuje wyjściem z metody i zakończeniem rekurencji.

3.3 OPERATOR MODULO

Dodamy teraz do listy operatorów matematycznych operator modulo, który jest ściśle powiązany z tzw. arytmetyką reszt, mającą szerokie zastosowanie w informatyce, przede wszystkim w zagadnieniach związanych z szyfrowaniem danych (kryptografią). Operacje z modulo są również bardzo użyteczne w programowaniu grafiki i wszędzie tam, gdzie mamy do czynienia z cyklicznością. Modulo, jako działanie matematyczne, jest operacją wyznaczania reszty z dzielenia dwóch liczb. Zapis *a mod b = r* oznacza, że *r* jest resztą z dzielenia *a* przez *b*. W Processingu operator modulo reprezentowany jest przez znak **%**.

```
int i = 10;
int j = 5;
print(i%j);
```

Ponieważ **5** jest dzielnikiem **10**, powyższy fragment kodu wydrukuje **0**. Jeśli natomiast zmienimy wartość zmiennej **i** o **1**, wynik będzie równy **1**.

```
int i = 11;
int j = 5;
print(i%j);
```

Jeśli chcielibyśmy sami zaimplementować tę operację, możemy to zrobić tak jak na listingu poniżej. Należy wówczas pamiętać, że choć dzielimy dwie liczby całkowite, to działanie musi zostać wykonane z precyzją liczb rzeczywistych. Wytłuszczony fragment jest właściwą operacją modulo na liczbach całkowitych.

```
void setup(){
        modulo(11,5);
}
void modulo(float a, float b){
        float r = a - b*floor(a/b);
        println("Modulo z dzielenia " + (int)a +
        "  przez "+(int)b+" wynosi " + (int)r);
}
```

```
Modulo z dzielenia 11 przez 5 wynosi 1
```

Możemy skorzystać z mechanizmu rekurencji, żeby na przykładzie analogii do zegara, pokazać cykliczny charakter wyników generowanych przez tę operację.

```
int limit = 48;
void setup(){
        zegar(0);
}
void zegar(int n){
        if(n>=limit){
                return;
        }
        int godzina = n % 12;
        println("Godzina "+godzina+":00");
        zegar(n+1);
}
```

```
Godzina 0:00
Godzina 1:00
Godzina 2:00
Godzina 3:00
```

```
Godzina 4:00
Godzina 5:00
Godzina 6:00
Godzina 7:00
Godzina 8:00
Godzina 9:00
Godzina 10:00
Godzina 11:00
Godzina 0:00
Godzina 1:00
Godzina 2:00
Godzina 3:00
Godzina 4:00
Godzina 5:00
Godzina 6:00
...
```

3.4 WYKORZYSTANIE REKURENCJI W GRAFICE

Zjawisko rekurencji możemy również zilustrować używając narzędzi rysujących. Napiszemy metodę do której przekażemy liczbę okręgów, które chcielibyśmy wyrysować na ekranie. Ustawmy w **`setup`** rozmiar okna na kwadrat o boku równym **200** pikseli oraz wywołanie poniższej metody.

```
void setup(){
        size(200,200);
        rysujOkregi(10);
}
```

```
void rysujOkregi(int i) {
        if (i==0) {
                return;
        }
        fill(i*25.5);
        ellipse(width/2, height/2, i*20, i*20);
        rysujOkregi(i-1);
}
```

W oknie powinniśmy uzyskać następujący obrazek.

EFEKT ZASTOSOWANIA REKURENCJI

4 FUNKCJE

4.1 ZWRACANIE WARTOŚCI PRZEZ METODY

Nie będziemy się zajmować matematyczną definicją funkcji. Z punktu widzenia programowania, funkcją jest każda metoda, która zwraca wartość. Słowo kluczowe **return** ma tu jeszcze jedno zastosowanie prócz przerywania rekurencji i natychmiastowego wyjścia z metody. Umieszczenie zmiennej lub literału (tzn. wartość wpisaną wprost) po **return**, pozwala zamienić metodę w pełnoprawną funkcję, tzn. w metodę, której wynik działania można przypisać do zmiennej. W Processingu zmienne mogą przechowywać tylko wartości, nie mogą natomiast przechowywać metod czy funkcji. Jeśli przypisujemy do zmiennej wywołanie jakiejś funkcji, na przykład sinus, to w rzeczywistości przypisujemy do niej wynik działania, a nie samą funkcję.

Aby naszą metodę można było użyć w jakimkolwiek przypisaniu, tak jak to jest możliwe w przypadku wbudowanych funkcji matematycznych, musimy zaopatrzyć ją w mechanizm przekazujący wartość z wnętrza metody na zewnątrz. Rozważmy prosty przykład, w którym napiszemy definicję działania obliczającego pole prostokąta.

```
float obliczPoleProstokata(float bokA, float bokB){
        float pole = bokA * bokB;
        return pole;
}
```

Do tej pory definicje metod zaczynaliśmy od słowa **void**. Oznaczało to, że metoda stanowiła zamkniętą całość, której wkład w program polegał na tzw. efektach ubocznych, np. metoda drukowała komunikat

w konsoli lub wyrysowywała figurę w oknie programu. Jeżeli weźmiemy funkcje matematyczne, to na ich przykładzie jasno widać, że niezależnie od skomplikowania procedury obliczeniowej, istotny jest dla nas tylko wynik. Jeżeli zależy nam na tym, żeby konsekwencją ciągu procedur zawartych w ciele funkcji, (czyli w bloku kodu w nawiasie klamrowym po sygnaturze), była wartość, która następnie zostanie przekazana do zmiennej na prawo od miejsca wywołania funkcji, musimy zastąpić słowo **void** terminem deklarującym typ pożądanego wyniku. Określamy w ten sposób typ metody, analogicznie jak to robiliśmy przy deklaracji zmiennych. Tak więc funkcja jest metodą z typem. Drugą niezbędną czynnością, zamieniającą zwykłą metodę w funkcję, jest dodanie do ciała funkcji wyrażenia skonstruowanego ze słowa **return** i następującej po nim wartości zgodnej z typem funkcji. Oba warunki muszą zostać spełnione – w przeciwnym razie zostanie zgłoszony błąd. Ogólnie składnia funkcji wygląda następująco.

```
<TYP> <NAZWA>(<SYGNATURA>){
    <WYRAŻENIA>
    return <WARTOŚĆ>;
}
```

<WARTOŚĆ> powinniśmy zastąpić wyrażeniem, literałem lub zmienną, która koresponduje z typem występującym przed **<NAZWA>**.

W ciele funkcji może być więcej niż jedno wywołanie **return**. Należy przy tym pamiętać, że wyrażenie z tym słowem sprawia, że po jego ewaluacji następuje wyjście z funkcji. Dla przykładu zaimplementujemy funkcję obliczającą wartość bezwzględną liczby przekazanej jako argument. Wspomnę tu, że nie jesteśmy do

tego zmuszeni – w Processingu jest gotowa funkcja matematyczna o nazwie **abs**, która robi dokładnie to samo.

```
float wartoscAbsolutna(float liczba){
        if(liczba >= 0) {
                return liczba;
        }
        else {
                return -1.0*liczba;
        }
}
```

4.2 PRZECIĄŻANIE METOD I FUNKCJI

Dla wygody, od tej pory będę używał w tekście jednego terminu „funkcja", gdy będzie mowa o zasadach wspólnych zarówno dla metod jak i funkcji.

Na identyfikator funkcji, inaczej niż w zmiennych gdzie jest nim tylko nazwa, składa się dodatkowo jej sygnatura. Jak pamiętamy – nie jest możliwe stworzenie dwóch zmiennych o tej samej nazwie lecz różnym typie. Możemy natomiast zdefiniować dowolną ilość funkcji o tej samej nazwie lecz różnym typie, jeśli będą się różnić sygnaturą. Należy zaznaczyć, że zmiana wyłącznie nazw argumentów nie zmienia sygnatury. Technika definiowania kilku funkcji o tej samej nazwie nosi miano przeciążania funkcji (*ang. method overloading*).

```
float obliczPoleProstokata(float bok){
        float pole = sq(bok);
        return pole;
}
```

Powyższa definicja po dodaniu do programu zawierającego uprzednio zdefiniowaną funkcję **poleProstokąta** z dwoma argumentami, nie wchodzi z nią

w konflikt. Możemy zatem użyć pierwszej bądź drugiej w zależności od tego, czy chcemy obliczyć pole kwadratu czy prostokąta.

Możemy też zmienić tylko typy argumentów, pozostawiając niezmienioną ich liczebność. To bowiem zmienia już sygnaturę. Poniższa funkcja zostanie wywołana, jeśli zamiast **float** przekażemy **int**.

```
float obliczPoleProstokata(int bok){
        float pole = sq(bok);
        return pole;
}
```

Niedopuszczalne jest jednak zdefiniowanie w jednym kodzie dwóch funkcji o tej samej nazwie i sygnaturze lecz innego typu. Jeżeli dodamy do kodu następującą definicję **obliczPoleProstokata**, to przy próbie uruchomienia pojawi się komunikat o błędzie, informujący o niedozwolonej próbie zdefiniowania dwóch funkcji o tym samym identyfikatorze.

```
int obliczPoleProstokata(int bok){
        int pole = (int)sq(bok);
        return pole;
}
```

4.3 WYRAŻENIA BOOLOWSKIE

Nauczyliśmy się już, jak konstruować proste wyrażenia boolowskie przy pomocy operatorów porównania. Z racji tego, że operatory te są dwuargumentowe, zmuszeni byliśmy do zagnieżdżania warunków. Dzięki operatorom logicznym będziemy w stanie tworzyć wieloczłonowe wyrażenia. Porównajmy obie struktury.

```
if(i > 0) {
        if(i < 10) {
            print("i jest większe od 0 i mniejsze od 10");
        }
}
if(i > 0 && i < 10) {
        print("i jest większe od 0 i mniejsze od 10");
}
```

Obie działają identycznie. W drugiej użyliśmy operatora koniunkcji logicznej **&&** (**AND**), który pozwolił na połączenie dwóch prostych wyrażeń w jedno dłuższe, bez konieczności zagnieżdżania dwóch bloków kodu.

W rachunku zdań, wyrażenie ze spójnikiem koniunkcji **AND** będzie prawdziwe jedynie wówczas, gdy wszystkie wyrażenia cząstkowe również będą prawdziwe. Te i inne logiczne reguły obowiązują także w Processingu. Niektóre funktory zdaniotwórcze (koniunkcja, alternatywa, równoważność, negacja) zostały włączone do języka i posługujemy się nimi analogicznie jak w klasycznym rachunku zdań. Poniżej znajduje się tabela operatorów w formie, w jakiej są one reprezentowane w Processingu oraz tablice prawdy dla każdego z nich.

NAZWA	OPERATOR
Koniunkcja logiczna AND	&&
Alternatywa OR	||
Negacja NOT	!

p	q	p OR q
0	0	0
0	1	1
1	0	1
1	1	1

p	q	p AND q
0	0	0
0	1	0
1	0	0
1	1	1

p	NOT p
1	0
0	1

Chociaż Processing nie jest językiem przeznaczonym do programowania logicznego (w przeciwieństwie do np. języka PROLOG), możemy w nim wyrazić szereg tautologii rachunku zdań. I tak na przykład pierwsze prawo De Morgana możemy zapisać jako **`!(p && q) == (!p || !q)`**. Oczywiście umieszczenie tego jako warunku w implikacji **`if`** jest bezcelowe – niezależnie od wartości zmiennych boolowskich **`p`** i **`q`**, blok kodu po **`if`** zostanie zawsze wykonany. Pokazuje to działanie poniższego kodu.

```
void setup(){
        deMorgan1(false, false);
        deMorgan1(true, false);
        deMorgan1(false, true);
        deMorgan1(true, true);
}
void deMorgan1(boolean p, boolean q){
        if(!(p && q) == (!p || !q) ){
                println("p = "+p+" ; q = " +q);
        }
}
```

```
p = false ; q = false
```

```
p = true ; q = false
p = false ; q = true
p = true ; q = true
```

4.4 KOMENTOWANIE KODU

Do tej pory wszystko, co pisaliśmy w edytorze, stanowiło fragmenty wykonywalnego kodu. Kod był złożonym wyrażeniem, którego adresatem była przede wszystkim maszyna. Oczywiście komunikaty wydrukowane do konsoli stanowią tu swego rodzaju wyjątek. Pokażemy teraz sposób wplatania komentarzy w kod. Komentarze to fragmenty kodu, które nie będą wykonywane przez maszynę. Mogą natomiast stanowić cenną wskazówkę dla innych programistów, czytających źródło programu. Umieszczanie komentarzy jest też pomocne dla samego autora, piszącego rozbudowany kod.

```
// To jest komentarz. Mogę go  kontynuować do końca linii.
/* Natomiast ten komentarz
może być dowolnie długi,
jeśli znajduje
się pomiędzy znacznikiem
otwierającym i zamykającym */
```

4.5 METODA DRAW

Istnieją dwa sposoby umieszczenia komentarza w kodzie. Jak widzimy w powyższym kodzie, podwójny ukośnik `//` **(backslash)** wprowadził komentarz w jednej linii. Znaczniki **`/* i */`** pełnią rolę ograniczników komenta-

rza rozciągającego się na większą ilość linii. Komentarze możemy umieszczać w dowolnym miejscu kodu. Mamy również pełną swobodę składniową oraz możemy używać wszystkich znaków alfabetu, dostępnych w kodowaniu UTF-8, (łącznie ze znakami polskimi).

Metoda **setup** jest wyrażeniem wykonywanym zaraz po uruchomieniu skryptu i tylko raz podczas całego jego działania. Ten fakt sprawia, że jeśli chcielibyśmy wprowadzić do programu ruch czy też inną jakość zależną od czasu, potrzebujemy dodatkowego mechanizmu, opartego na cyklicznym wywoływaniu metod czy funkcji. Podstawowym mechanizmem tego typu jest w Processingu metoda **draw,** która wywoływana jest w stałych odstępach czasu. Składnia tej metody wygląda następująco.

```
void draw(){
    <CIAŁO>
}
```

Definicję **draw** umieszczamy pod zdefiniowaną metodą **setup**. Ciało **draw** definiujemy tak jak ciało każdej innej metody typu **void**. Symbol **<CIAŁO>** w definicji występuje w zastępstwie dotychczas używanego **<WYRAŻENIA>.** Oba wymagają identycznych reguł podstawienia na symbole terminalne. Musimy oczywiście pamiętać, że w ciele funkcji nie możemy zamieścić wyrażenia będącego definicją innej funkcji. Metodę **draw** wyróżnia fakt, że nie piszemy jej wywołania w żadnym miejscu kodu. Jest ona wywoływana automatycznie, z domyślną częstotliwością **60** razy na sekundę przez cały czas działania uruchomionego skryptu. Można wyobrazić sobie listwę czasu, na której umieszczono

jej wywołanie klatka po klatce. Metoda **draw** zawiera najczęściej wywołania metod tworzących wizualną zawartość skryptów. Stąd też jej nazwa (pl. rysuj). Nie jest to jednak żelazna zasada, ponieważ możemy **draw** zdefiniować na dowolny (w granicach poprawności) sposób, tak jak każdą inną metodę. Pamiętajmy jednak, że podobnie jak **setup**, wywołanie **draw** odbywa się bez naszego udziału z ustaloną z góry regularnością. Rozważmy taki program:

```
int predkoscX = 2;
int predkoscY = 2;
int pozycjaX = 0;
int pozycjaY = 0;

void setup() {
        size(320, 240);
}
void draw() {
        pozycjaX = pozycjaX + predkoscX;
        pozycjaY = pozycjaY + predkoscY;
        ellipse(pozycjaX, pozycjaY, 10, 10);
}
```

Po uruchomieniu skryptu, zobaczymy małe koło przemieszczające się po przekątnej okna. Koło rozpocznie swój ruch w punkcie (0,0) układu współrzędnych obowiązującego w Processingu. Właściwie zobaczymy szereg białych kół wzdłuż trajektorii ruchu. Wynika to z tego, że przy każdym wywołaniu metody **draw** umieszczony zostaje na płótnie nowy element. Nie zostaje przy tym usunięta zawartości z poprzedniego wywołania **draw**. Okno uruchomionego skryptu jest niczym palimpsest – żeby zapisać na jego powierzchni coś nowego, konieczne jest uprzednie wymazanie po-

przedniego zapisu. Umieśćmy zatem w pierwszej linijce wywołanie metody **background(200)**. Dzięki temu każdorazowe wyrysowanie koła na nowych współrzędnych odbędzie się na czystej „karcie". Zmienne **pozycjaX** i **pozycjaY** przechowują aktualną pozycję figury i są aktualizowane za każdym razem, gdy wykonywane jest **draw**. Stąd też w niedługim czasie koło znika poza granicami okna, ponieważ wartość zmiennych **X** i **Y** niezmiennie rośnie. Żeby temu zapobiec, możemy dodać prosty mechanizm utrzymujący koło w obszarze okna. Dopiszmy do **draw**, zaraz pod **background** następujące klauzule.

```
if(pozycjaX > width || pozycjaX < 0){
        predkoscX = -1 * predkoscX;
}
if(pozycjaY > height || pozycjaY < 0){
        predkoscY = -1 * predkoscY;
}
```

Warunki w **if** sprawdzają, czy wartości zmiennych, przechowujących współrzędne, nie przekroczyły wielkości, które umieszczają obiekt poza oknem. Jeśli tak się stało, przemnożenia przez **-1** spowodują, że składnik sumy uaktualniający pozycję zmieni znak na przeciwny. Uzyskamy dzięki temu efekt odbijania się obiektu od ściany.

Możemy oba te warunki przenieść do funkcji, która zwraca wartość boolowską. Taki rodzaj funkcji nosi nazwę predykatu. W tym celu napiszmy definicję następującej funkcji poniżej definicji **draw**.

```
boolean jestZderzenie(int pozycja, int sciana) {
        if (pozycja > sciana || pozycja < 0) {
                return true;
        }
```

```
        else {
                return false;
        }
}
```

Domyślną dla **draw** częstotliwość **60** razy na sekundę możemy zmienić poprzez wywołanie w **setup** metody **frameRate**. Przekazujemy argument liczbowy – **int** lub **float** – określający nową prędkość. Dodatkowo możemy włączyć antyaliasing, który poprawi jakość wyświetlanego obrazu. W tym celu dopisujemy **smooth()**.

Skrypt przez nas napisany posiada sporo niedoskonałości. Jeśli chodzi o pożądaną przez nas detekcję ruchu, to oczekujemy, że znakiem sygnalizującym zderzenie powinien być moment zrównania się wartości którejś ze współrzędnych krawędzi koła ze współrzędną krawędzi okna. Tymczasem aktualny kod sprawia, że program sprawdza położenie środka koła względem krawędzi. Skorygujemy to poprzez wprowadzenie kilku nowych zmiennych i modyfikację wartości przekazywanej funkcji **jestZderzenie**. Dodamy następujące zmienne typu **int: srednicaX, srednicaY, kierunekX, kierunekY**. Na tym etapie kod wygląda następująco.

```
int predkoscX = 2;
int predkoscY = 2;
int pozycjaX = 0;
int pozycjaY = 0;
int srednicaX = 12;
int srednicaY = 12;
int kierunekX = 1;
int kierunekY = 1;
```

```
void setup() {
        size(320, 240);
        smooth();
        frameRate(30);
}
void draw() {
        background(200);
        boolean zderzenieX =
                jestZderzenie(pozycjaX+srednicaX/2,width);
        boolean zderzenieY =
                jestZderzenie(pozycjaY+srednicaY/2,height);
        if (zderzenieX) {
                kierunekX = -1*kierunekX;
                predkoscX = kierunekX * predkoscX;
                srednicaX = kierunekX * srednicaX;
        }
        if (zderzenieY) {
                kierunekY = -1*kierunekY;
                predkoscY = kierunekY * predkoscY;
                srednicaY = kierunekY * srednicaY;
        }
        pozycjaX = pozycjaX + predkoscX;
        pozycjaY = pozycjaY + predkoscY;
        ellipse(pozycjaX, pozycjaY,
        srednicaX, srednicaY);
}
boolean jestZderzenie(int pozycja, int sciana) {
        if (pozycja > sciana || pozycja < 0) {
                return true;
        }
        else {
                return false;
        }
}
```

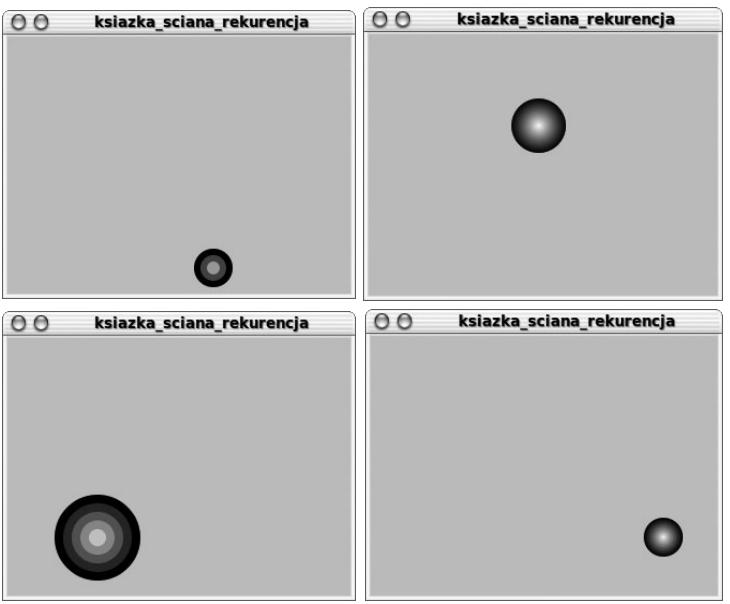
Tym razem koło odbija się od krawędzi okna zgodnie z naszymi oczekiwaniami. Wartości zmiennych **kierunek** (dla ruchu w poziomie i pionie) są współczynnikami, przez które mnożymy promień koła (połowę jego średnicy). W zależności od tego, czy koło przemieszcza się w dół czy w górę, w lewo lub w prawo, zmienna ta przechowuje **1** bądź **-1**. Wpływa to na wynik operacji, który następnie przekazujemy jako pierwszy argument funkcji **jestZderzenie**. W rzeczywistości więc na przemian dodajemy lub odejmujemy promień koła od współrzędnych jego środka.

Do naszego kodu dodamy jeszcze jedną metodę, która zdefiniowaliśmy przy omawianiu rekurencji. Metodzie **okregi** nadajmy nową, bardziej obrazową nazwę **rysujKola**. Ponadto do sygnatury dodamy dwa argumenty określające współrzędne, oraz jeden dla promienia każdego kolejnego koła. Metoda powinna być zdefiniowana następująco:

```
void rysujKola(int pozX, int
pozY, int promien, int r) {
        if (r==0) {
        return;
        }
        fill(255-r*(255/R));
        noStroke();
        ellipse(pozX, pozY, r*promien, r*promien);
        okregi(pozX, pozY, promien, r-1);
}
```

Do deklaracji zmiennych na samej górze kodu (tzn. przed **setup**) dopiszmy dwie typu **int — promien**, przechowującą promień najmniejszego koła, oraz **R**, do której przypiszemy liczbę pożądanych rekurencji. Śred-

nica uzyskanej figury będzie wynikiem przemnożenia wartości zmiennej **promien** przez wartość **R**. Musimy więc przenieść utworzenie zmiennych **srednicaX** oraz **srednicaY** poniżej i umieścić wynik operacji opisanej w poprzednim zdaniu w tych wyrażeniach. Ostatnim krokiem jest zamiana w **draw** linijki **ellipse(pozycjaX, pozycjaY, srednicaX, srednicaY);**, na **rysujKola(pozycjaX, pozycjaY, promien, R);**. Po uruchomieniu programu zobaczymy coś na kształt piłki odbijającej się od ścian prostokąta. Poeksperymentujcie z rozmiarem promienia i ilością rekurencji. W zależności od tych parametrów, uzyskamy kule o różnych rozmiarach i precyzji gradientu.



CZTERY URUCHOMIENIA TEGO SAMEGO PROGRAMU Z RÓŻNYMI ARGUMENTAMI PRZEKAZANYMI METODZIE REKURENCYJNEJ

5 ITERACJA

5.1 STAŁE. BŁĘDY PODCZAS KOMPILACJI

Zmienne możemy wyobrazić sobie jako swego rodzaju pudełka, w których umieszczamy wartości. W zależności od rodzaju pojemnika, przechowujemy w nim liczby całkowite, rzeczywiste, ciągi znaków lub wartości boolowskie. Możemy w każdym momencie sięgnąć po dane pudełko nie tylko w celu odczytania jego aktualnej zawartości, ale również w celu zamiany tej zawartość na nową. Innymi słowy, nową wartość możemy przypisywać do zmiennej dowolną ilość razy, pod warunkiem, że nie następuje niezgodność typów, tzn. **`int`** możemy zastąpić tylko **`int`**, itp.

Istnieją sytuacje, w których chcielibyśmy przypisać wartość do zmiennej na stałe i być pewni, że nie zmieni się ona w toku działania programu. Tego rodzaju zmiennymi są np. stałe matematyczne. Jest jasne, że wartość liczby pi nie może ulec zmianie. Sposobem na uniknięcie tego, jest konstruowanie kodu w taki sposób, żeby zapobiec powtórnym przypisaniom. Niemniej w Processingu istnieje mechanizm, dzięki któremu zmiennej możemy przypisać wartość tylko raz. Każda próba ponownego przypisania do niej wartości będzie z góry skazana na niepowodzenie. To jest gwarancja, że wartość pozostanie przy zmiennej na stałe. Aby określić taki typ zmiennych, deklarację zaczynamy od słowa kluczowego **`final`** – poprzedza ono termin określający typ zmiennej. Właściwe jest w takich wypadkach zaniechać nazywania tego typu danych zmiennymi i mówić dla

odróżnienia o stałych. Poniżej przedstawiona jest składnia stworzenia instancji, możemy jednak rozbić ją na deklarację i przypisanie, tak jak to robiliśmy ze zwykłymi zmiennymi.

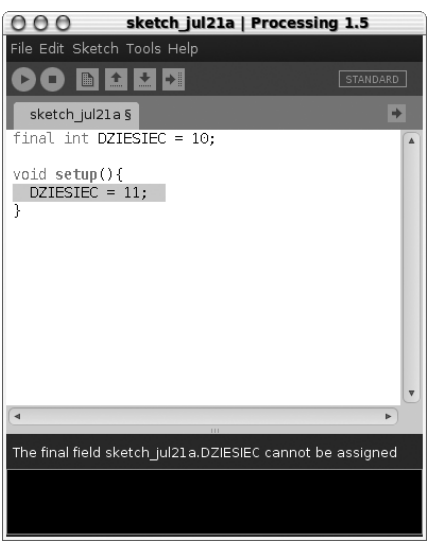
`final <TYP> <NAZWA> = <WARTOŚĆ>;`

Np.

```
final int DZIESIEC = 10;
```

W powyższym kodzie stworzyliśmy stałą typu int. Nazwy stałych, dla odróżnienia od zmiennych, będziemy pisać dużymi literami stosując znak dolnej linii w miejsce bikapitalizacji w nazwach złożonych z kilku słów. Napiszemy więc np. **`WAZNA_WARTOSC`** zamiast **`waznaWartosc`**.

Jeśli spróbujemy przypisać nową wartość do stałej, otrzymamy komunikat o błędzie przy próbie uruchomienia kodu. Widać to na poniższej ilustracji.



PRZYKŁAD ŚWIADOMIE WYWOŁANEGO BŁĘDU CZASU KOMPILACJI

Istnieją zasadniczo dwa rodzaje błędów – kompilacji oraz wykonania. Ten przed chwilą przez nas sprowokowany należał do pierwszego typu. Łatwiej sobie z nim radzić, gdyż o jego istnieniu dowiadujemy się przy próbach uruchomienia kodu. Drugi typ błędu jest trudniejszy do wykrycia, ponieważ pojawia się dopiero na poziomie wykonania programu, z reguły z „niewiadomych przyczyn". Taki błąd jest z reguły błędem typu logicznego, tzn. istnieje pomimo formalnej (syntaktycznej) poprawności kodu. (Dlatego też program w ogóle działa, choć działania te są błędne). Sama poprawność składni kodu nie gwarantuje, że program uniknie stanów, w których dalsze jego działanie przestaje być możliwe. Program przerywa wówczas swoje działanie lub doprowadza do konieczności restartu systemu.

Nie powinniśmy się przejmować tym, że początkujący programista popełnia błędy, najczęściej typu pierwszego, o naturze gramatycznej. Błędy są oczywiście frustrujące, ale na nich uczymy się najwięcej. Processing posiada system, który pozwala radzić sobie z błędami – moduł testowania jest jednym z podstawowych podzespołów każdego IDE. W poprzednim przykładzie komunikat o błędzie poinformował nas precyzyjnie o przyczynie błędu.

```
The final field sketch_jul21a DZIESIEC cannot be assigned.
```

W edytorze pojawiła się też wizualna podpowiedź – linijka, zawierająca błędny fragment kodu, została „podświetlona" na pomarańczowo. Początkującemu programiście podpowiedzi te mogą wydawać się niejasne, ale z biegiem czasu staną się coraz bardziej użyteczne. Poza tym popełniane przez nas błędy składniowe staną się coraz rzadsze, w miarę coraz lepszego opanowania reguł składni.

5.2 OPERATORY IN- I DEKREMENTACJI

Do tej pory, gdy chcieliśmy zmienić wartość zmiennej z użyciem jej aktualnej wartości, korzystaliśmy z następującej konstrukcji.

```
<NAZWA> = <NAZWA> <OPERATOR> <WARTOŚĆ>;
```

<OPERATOR> jest dowolnym operatorem dopuszczonym do pracy z danym typem zmiennej, np. dla zmiennych liczbowych były to znaki działań matematycznych.

```
int i = 1;
i = i + 4;
print(i);
```

```
5
```

Operację ponownego przypisania z użyciem aktualnej wartości zmiennej możemy skrócić używając następującej konstrukcji.

```
<NAZWA> <OPERATOR>= <NAZWA>;
```

Tak więc równoważnie z poprzednim przykładem, możemy napisać jak poniżej.

```
// i = i + 4;
i += 4;
```

Jeśli chcemy wartość zmiennej zwiększyć lub zmniejszyć o 1, możemy użyć operatorów inkrementacji (‘**++**’) i dekrementacji (‘**--**’).

```
<NAZWA>++;
<NAZWA>--;
```

Są jeszcze dwa warianty tych wyrażeń, z operatorami stojącymi przed nazwą zmiennej.

```
++<NAZWA>;
--<NAZWA>;
```

One również zwiększają i zmniejszają wartości, z tą różnicą, że robią to przed ewaluacją wyrażenia. Ilustruje to następujący przykład.

```
int a = 0;
println(a++);
println(a);
println(++a);
println(a);
```

```
0
1
2
2
```

Zmienna a o wartości **0** została przekazana metodzie **println** w konstrukcji z operatorem inkrementacji. Wartość zmiennej została wydrukowana i następnie zwiększyła się o **1**. Kolejne **println** wydrukuje już tą nową wartość. Następnie zostaje przekazana **println** zmienna **a**, tym razem z operatorem dekrementacji umieszczonym przed jej nazwą. W wyniku tego, jeszcze przed wydrukowaniem jej w konsoli, wartość zmiennej zostanie zwiększona o **1**.

5.3 REPREZENTACJA KOLORU W PROCESSINGU. MAPOWANIE

Komputer jest urządzeniem elektronicznym. W konsekwencji – to, co piszemy w edytorze, posiada przede wszystkim postać sygnału elektronicznego, a raczej sekwencji dwóch dyskretnych stanów – napięcia lub

spoczynku. Centralnym modułem każdego komputera jest procesor, zwany też odpowiednio po angielsku CPU (Central Processing Unit). Przedmiotem jego pracy jest wykonywanie ściśle określonych operacji na danych. Dane i rozkazy zapisane są jako słowa składające się z sekwencji bitów. Bit, identycznie jak zmienna boolowska, może przyjąć tylko dwie wartości: **0** lub **1**. Słowo przekazane do procesora jest w gruncie rzeczy liczbą całkowitą zapisaną w dwójkowym systemie pozycyjnym. Fragment tej liczby jest odczytywany jako rozkaz – identyfikator operacji procesora, a reszta jako dane, na których operacja ma zostać wykonana. Procesory mogą się różnić długością akceptowanych słów i częstotliwością ich odczytywania. W zależności od długości słowa mówimy, że procesor jest **8**, **16**, **32**, **64** bitowy. Wszystkie te liczby są potęgą liczby **2**. Częstotliwość wymiany słów określa szybkość taktowania procesora. Tak więc wykonanie przez procesor rozkazu prowadzi do nowej liczby, która ponownie jest skonstruowana z rozkazu i danych, itd. Przejście z jednego stanu w drugi jest więc rodzajem działania matematycznego na liczbach dwójkowych. Nie przez przypadek nazwa urządzenia pochodzi od angielskiego słowa *compute* – liczyć.

Na rozkazy wydane procesorowi składają się m.in. działania arytmetyczne, działania logiczne na bitach (**0** – fałsz, **1** – prawda) oraz kopiowanie danych. Program przetłumaczony na ciąg rozkazów dla procesora nazywa się kodem maszynowym. Pierwsze komputery programowało się bezpośrednio w takim kodzie. Ogólnie mówi się, że im bardziej program przypomina język maszynowy, tym język służący do wyrażenia tego programu jest niższego rzędu. Na szczęście dzisiaj, pomimo oczywistej sztuczności, możemy pozwolić sobie

na komunikację z komputerem w języku zbliżonym do języka naturalnego. Języki takiego typu (do których zalicza się Processing) nazywa się dla odróżnienia językami wysokiego rzędu. Program napisany w Processingu jest wykonywany w wirtualnej maszynie Javy, dlatego przetłumaczony zostaje przez kompilator na tzw. kod pośredni. Kod pośredni nazywa się też bajtowym, ponieważ każdy rozkaz skierowany do procesora posiada identyfikator w postaci numeru zapisanego jako liczba o długości nie dłuższej niż **1** bajt[17].

AAAAAAAARRRRRRRRGGGGGGGGBBBBBBBB

Kolor jest zapisany w przestrzeni barw **ARGB**. Analogia do przestrzeni bierze się stąd, że pojedynczy kolor pojmuje się jako punkt w czterech wymiarach, którego współrzędnymi są jego składowe. Są to odpowiednio przezroczystość (**A** od ang. alpha), czerwony (**R** od ang. red), zielony (**G** od ang. green), i niebieski (**B** od ang. blue). Jak widać każdy kanał **ARGB** zawiera bajt informacji (czyli wartość w zakresie (**0,255**) w systemie dziesiętnym.), a kolor jako całość jest zbitką tych czterech liczb określających intensywność poszczególnych kanałów. Przykładowo kolor fioletowy można zapisać jako **11111111011111111000000011111111**.

Obraz na ekranie komputera jest siatką złożoną z pojedynczych punktów zwanych pikselami, które z kolei składają się z trzech subpikseli, świecących jednym z trzech wymienionych kolorów z różną intensywnością. Wiązki emitowanego światła są następnie mieszane na siatkówce naszego oka. Proces ten nosi nazwę syntezy addytywnej. Synteza sprawia, że postrzegamy poszczególny piksel jako jeden z ponad szesnastu milionów kolorów

17 Bajt jest jednostką informacji zawierającą 8 bitów.

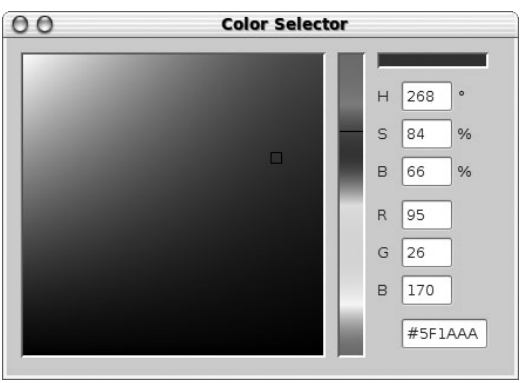
(**2^{24}**). Ponieważ każdy z trzech subpikseli może świecić na 256 sposobów, otrzymujemy liczbę wszystkich dostępnych kombinacji mnożąc przez siebie liczbę ich potencjalnych stanów. Kanał przezroczystości jest dla naszego oka nieistotny. Składowa ta służy programowi do określenia stopnia przenikania się płaszczyzn kolorów.

SYNTEZA ADDYTYWNA RGB

W grafice komputerowej używa się również zapisu w systemie szesnastkowym, który jest blisko spokrewniony z systemem dwójkowym (**16** jest potęgą liczby **2**). W Processingu możemy pozostać przy dobrze znanym systemie dziesiętnym i określać wartości kanałów z przedziału **<0, 255>**. Zapis dwójkowy przydaje się wówczas, gdy zależy nam na wydajności operacji na poszczególnych pikselach, choć coraz mniejsze ma to znaczenie z powodu coraz większej wydajności maszyn.

Dotychczas wywoływaliśmy metody **fill** i **background** o różnej liczbie argumentów (1, 3 bądź 4) o typie **int**. Jeśli, tak jak to robiliśmy dotychczas, przekażemy tylko jeden argument, to ustalimy wszystkie **3** składowe (**RGB**) koloru na tą samą wartość, uzyskując szarość o określonym stopniu nasycenia. W przypadku **3** argumentów, każda liczba określa poszczególny kanał. Kanał przezroczystości jest podawany jako czwarty argument i używamy go tylko wówczas, gdy zależy nam na efekcie przenikania.

Processing posiada przydatne narzędzie do pracy z kolorami. Uruchamiamy je wybierając z Menu *Tools->Color Selector*. Możemy tu kursorem myszki wybrać z palety dowolny kolor, w bardzo podobny sposób w jaki pracujemy w programach do obróbki plików graficznych. W prawej części okna pokażą się nam wartości każdej z trzech składowych, zarówno w przestrzeni **RGB**, jak i **HSB**, o której będzie mowa za chwilę. Dodatkowo mamy też zapis w kodzie szesnastkowym. Możemy też używać tego narzędzia w drugą stronę, wpisując wartości w pola i sprawdzając ich wpływ na zmianę barwy.



OKNO NARZĘDZIA DO WYBORU KOLORU

W Processingu istnieje specjalny typ zmiennej do przechowywania koloru, która nosi nazwę **color**. Deklarujemy ją jak każdy inny typ prymitywny (w istocie jest **int**) lecz wartość możemy przypisać na kilka sposobów. Najczęściej wykorzystujemy do tego funkcję o nazwie **color**, która posiada identyczną sygnaturę jak **fill** i **background** lecz zwraca wartość, która jest reprezentacją liczbową pożądanej barwy. Drugi sposób polega na użyciu literału skonstruowanego ze znaku **#** i wartości liczbowej w kodzie szesnastkowym, tj. **#RRGGBB**. Oba sposoby zaprezentowane są w przykładzie poniżej.

```
color czerwony1 = color(255,0,0);
color czerwony2 = #FF0000;
println(czerwony1==czerwony2);
println(Integer.toBinaryString(czerwony1));
println(Integer.toBinaryString(czerwony2));
true
11111111111111110000000000000000
11111111111111110000000000000000
```

Pomimo różnic w metodzie przypisania, obie zmienne przechowują identyczną wartość. Zmienne tego typu można przekazać do metod określających kolor tła (metoda **background**) czy wypełnienie figur (metoda **fill**). Gdy chcemy zdecydować o kolorze konturu, linii lub pojedynczego punktu, kolor przekazujemy metodzie **stroke**. Wszystkie te metody przyjmują argumenty liczbowe (**float** lub **int**) o wartościach z przedziału (**0,255**) – czyli z zakresu pojemności **8** bitów informacji. Jeżeli któryś z kanałów (bądź wszystkie) jest powiązany ze zmienną przyjmującą wartości spoza tego zakresu, zarówno poniżej wartości minimalnej jak i powyżej maksimum, możemy przed przekazaniem

przygotować argumenty funkcją mapującą, której ogólny wzór wygląda następująco.

```
<X_NOWE> = ((<X> – <MIN_STARE>)
 / (<MAX_STARE> – <MIN_STARE>))
 * (<MAX_NOWE> – <MIN_NOWE>)
 + <MIN_NOWE>
```

Mapowanie jest funkcją przenoszącą dane z jednego modelu (zakresu) w drugi. W praktyce może to wyglądać następująco. Załóżmy, że zmienna **<X>** może przyjąć wartość pomiędzy **0** i **10000**. Przyjmijmy też, że **<X>** będzie wynosiło 5032. Chcielibyśmy teraz zamienić tą wartość na inną spomiędzy **0–255**, zachowując przy tym relację wartości zmiennej do dolnej i górnej granicy zakresu. Musimy więc podstawić do wzoru odpowiednie wartości:

```
<X_NOWE> =(( 5032 – 0 ) /
(10000 – 0)) * (255 – 0) + 0
= 0,5032 * 255 = 128,316
```

Mając dany wzór matematyczny, nie powinno nam przysporzyć trudności napisanie funkcji mapującej. Możemy jednak użyć gotowej funkcji o nazwie **map**, której wywołanie wygląda następująco.

```
map(<X>, <MIN_STARE>, <MAX_STARE> ,
<MIN_NOWE>, <MAX_NOWE>);
```

Możemy teraz sprawić, czy nasze poprzednie obliczenia pokrywają się z działaniem tej funkcji.

```
int x_stare = 5032;
float x_nowe = map(x_stare, 0, 10000, 0, 255);
print(x_nowe);
```

```
128.316
```

5.4 ITERACJA

Dzięki rekurencji byliśmy w stanie pokierować przepływem programu w taki sposób, że wykonał on zbiór komend wielokrotnie. Podobnym mechanizmem automatyzacji jest iteracja (ang. *iterate* – powtarzać). Wyrażenie tworzące iterację nazywa się często wymiennie pętlą (ang. *loop*). Podstawową strukturą iteracyjną w Processingu jest pętla **for** o następującej składni.

```
for (<LICZNIK>; <WARUNEK_PĘTLI>;
<DZIAŁANIE_NA_LICZNIKU>) {
    <WYRAŻENIA>
}
```

Liczba wykonań bloku pętli jest określona w nawiasie po słowie for trzema wyrażeniami. Pierwsze z nich, **<LICZNIK>** jest wykonywane tylko raz, w momencie rozpoczęcia pętli przez program. W tym wyrażeniu wskazujemy na zmienną numeryczną, najczęściej typu **int**, która pełnić będzie rolę licznika wykonań pętli. Możemy w roli licznika użyć zmiennej wprowadzonej w kod wcześniej lub utworzyć nową instancję w tym właśnie miejscu. Najczęściej wybieramy tą drugą możliwość. Zakres zmiennej (zasięg jej dostępności), utworzonej w wyrażeniu, ograniczony jest do struktury for. Kolejny symbol nieterminalny, **<WARUNEK_PĘTLI>**, reprezentuje wyrażenie boolowskie, porównujące bieżącą wartość licznika do limitu, tj. liczby będącej kresem górnym lub dolnym wartości przyjmowanych przez licznik. Limit może występować w postaci literału liczbowego lub zmiennej liczbowej, której zakres obowiązuje w pętli. Jeśli warunek zostaje spełniony, program wykonuje polecenia z bloku kodu. Po wykonaniu ostatniego z nich, program przechodzi do **<DZIAŁANIE_NA_LICZNIKU>**.

Jest to działanie arytmetyczne zwiększające lub zmniejszające wartość licznika. Następnie program sprawdza ponownie warunek i jeśli ten test przyniesie wartość logiczną **true**, blok kodu zostaje wykonany kolejny raz. Scenariusz się powtarza do momentu, gdy warunek nie zostanie spełniony. Wówczas program opuszcza pętlę i przechodzi do kodu znajdującego się poniżej. Brzmi to może zawile, lecz w praktyce jest bardzo proste. Zauważcie jakie wartości w poniższym przykładzie przyjmuje zmienna typu **int** o nazwie **i**.

```
for(int i = 0; i < 10; i = i + 1) {
        print(i+ ",");
}
```

```
0,1,2,3,4,5,6,7,8,9,
```

Metoda `print` została wywołana dziesięć razy. Wartość zmiennej **i** rosła w przedziale **(0,9)**. Następny przykład jest równoznaczny w działaniu z poprzednim, z tą różnicą, że deklaracja licznika znajduje się na zewnątrz struktury pętli. To sprawia, że zmienna będąca licznikiem, jest dostępna również poniżej bloku pętli zachowując dodatkowo wynik przeprowadzonych na niej działań.

```
int i;
for(i = 0; i < 10; i = i + 1) {
        print(i+ ",");
}
print(i);
```

```
0,1,2,3,4,5,6,7,8,9,10
```

Możemy również działanie na liczniku przeprowadzać w przeciwnym kierunku. Wówczas porównujemy bieżącą wartość licznika do jego dolnego zakresu.

```
for(int i = 10; i > 0; i = i - 1) {
        print(i+ ",");
}
```

```
10,9,8,7,6,5,4,3,2,1,
```

Strukturę pętli możemy zastosować do tworzenia bardzo ciekawych form wizualnych. Jedną z nich jest gradient, czyli przejście tonalne między dwoma kolorami.

```
void setup() {
        size(400, 400);
        background(0);
        smooth();
        color kolorOd = color(255, 0, 0);
        color kolorDo = color(5, 228, 0);
        float deltaR = red(kolorDo)-red(kolorOd);
        float deltaG = green(kolorDo)-green(kolorOd);
        float deltaB = blue(kolorDo)-blue(kolorOd);
        for (int i = 0; i < 400; i++) {
                float wspolczynnik
                        = map(i, 0, 400, 0, 1);
                float r = red(kolorOd)
                        + deltaR * wspolczynnik;
                float g = green(kolorOd)
                        + deltaG *wspolczynnik;
                float b = blue(kolorOd)
                        + deltaB * wspolczynnik ;
                color kolorPomiedzy = color(r,g,b);
                stroke(kolorPomiedzy);
                line(0, i, width, i);
        }
}
```

W powyższym kodzie zostały użyte trzy nowe funkcje o nazwach **red**, **green** i **blue**. Pobierają one jedną z trzech składowych przekazanego im koloru, korespondującą z nazwą funkcji. Znając te wartości możemy określić odległości pomiędzy dwoma kolorami (w języku matematycznym różnicę nazywa się często deltą). W pętli korzystamy też z funkcji **map**. Z jej pomocą określamy współczynnik, który na początku pętli jest równy **0**, dlatego też początkowo **kolorPomiedzy** nie różni się od **kolorOd**. Niemniej z biegiem pętli wartość zmiennej **współczynnik** rośnie, przyjmując wartości coraz bliższe **1**, a to z kolei pociąga za sobą zmianę wartości nasycenia poszczególnych kanałów. Zmienna koloru sukcesywnie przyjmują wartości coraz bardziej zbliżone do składowych docelowego koloru.

Gradient można animować, co często daje bardzo interesujące wizualnie efekty. Poniższy przykład jest jednym z wariantów takiej animacji.

```
color KOLOR_OD, KOLOR_DO;
float R, G, B;
color kolorOd, kolorDo;
float DELTA_R;
float DELTA_G;
float DELTA_B;
float wspolczynnik;
int licznik;
boolean znak = false;

void setup() {
        size(400, 400);
        KOLOR_OD = #3708FA;
        KOLOR_DO = #00FF30;
        R = red(KOLOR_OD);
```

```
        G = green(KOLOR_OD);
        B = blue(KOLOR_OD);
        DELTA_R = red(KOLOR_DO)-R;
        DELTA_G = green(KOLOR_DO)-G;
        DELTA_B = blue(KOLOR_DO)-B;
        frameRate(25);
        smooth();
}

void draw() {
        licznik = frameCount%100;
        if (licznik==0) {
                znak=!znak;
        }
        if (znak) {
                wspolczynnik =
                        map(licznik, 0, 100, 1, 0);
        }
         else {
                wspolczynnik =
                        map(licznik, 0, 100, 0, 1);
        }
        float r = R + DELTA_R * wspolczynnik;
        float g = G + DELTA_G * wspolczynnik;
        float b = B + DELTA_B * wspolczynnik;
        kolorOd = color(r, g, b);
        fill(kolorOd);
        rect(0, 0, 400, 200);
        r = red(KOLOR_OD) + DELTA_R * (1-wspolczynnik);
        g = green(KOLOR_OD) + DELTA_G *(1-wspolczynnik);
        b = blue(KOLOR_OD) + DELTA_B * (1-wspolczynnik);
        kolorDo = color(r, g, b);
        fill(kolorDo);
        rect(0, 200, 400, 200);
}
```

Na powyższą animację składają się dwa pulsujące kolorem prostokąty. Kod wygląda trochę bardziej zawile, niemniej nie ma w nim żadnego elementu, którego nie poznalibyśmy wcześniej. Warunki na początku **draw** pełnią tę samą funkcję co w animacji z piłką. W tym przypadku to kolor przejściowy „odbija się" na przemian od kolorów będących podstawą gradientu. Drugi prostokąt jest wypełniony kolorem, który znajduje się w stałej odległości do pierwszego. Mamy więc dwie instancje wymijające się dokładnie w pół drogi.

Możemy wykorzystać przedstawioną mechanikę animacji wcześniejszego gradientu. Delta pomiędzy **kolorOd** i **kolorDo** posłuży jako podstawa do obliczeń koloru poszczególnej linii. Kod tej pętli jest w istocie nieznaczną modyfikacją wcześniejszej pętli gradientu. Usuńmy z **draw** linijki zawierające wywołania **fill** i **rect** oraz dopiszmy poniższy kod pod linijką zawierającą wyrażenie **kolorDo = color(r, g, b);**

```
for (int i = 0; i < width; i++) {
        wspolczynnik = map(i, 0, width, 0, 1);
        float deltaR = red(kolorDo)-red(kolorOd);
        float deltaG = green(kolorDo)-green(kolorOd);
        float deltaB = blue(kolorDo)-blue(kolorOd);
        r = red(kolorOd) + deltaR * wspolczynnik;
        g = green(kolorOd) + deltaG * wspolczynnik;
        b = blue(kolorOd) + deltaB * wspolczynnik;
        color kolorPomiedzy = color(r, g, b);
        stroke(kolorPomiedzy);
        line(0, i, width, i);
}
```

5.5 PRZESTRZEŃ HSB Alternatywną do **RGB** przestrzenią barw jest przestrzeń **HSB**. Nazwa jest akronimem od angielskich słów **hue** (barwa), **saturation** (nasycenie), **brightness** (jasność). Czasami na określenie tej przestrzeni wymiennie używa się akronimu **HSV** gdzie v pochodzi od **value** (wartość). **V** opisuje identyczną jakość co jasność. Podobnie jak w **RGB** jednostkowy kolor w przestrzeni **HSB** jest punktem w trzech wymiarach, z tą różnicą, że współrzędne są określone przez barwę, nasycenie i jasność. W modelu **HSB** przestrzeń barw jest stożkiem, którego podstawą jest koło barw. Składową **hue** (barwę) zapisujemy jako wielkość kątową w stopniach, np. odcienie trzech podstawowych barw w **RGB** (czerwony, zielony, niebieski) są umieszczone odpowiednio na promieniach o następujących stopniach: **0**° (lub **360**°), **120**° i **240**°. Następnie w definicji koloru określamy **saturation** (nasycenie). W tym przypadku wartość oznacza odległość punktu koloru od osi stożka. Kolor traci na intensywności proporcjonalnie do odległości punktu od osi. Wreszcie jako trzecią współrzędną definiujemy **brightness** (jasność). Punkt traci na jasności przesuwając się w kierunku wierzchołka. W narzędziu *Color* możemy zaobserwować jak identyczny kolor jest reprezentowany zarówno w przestrzeni **RGB** jak i w **HSB**. Wartości nasycenia i jasności w tym ostatnim są podawane w procentach, a więc w zakresie (0, 100). Poniższy program wyrysuje nam podstawę stożka **HSB**.

```
void setup() {
        size(400, 400);
        background(0);
        smooth();
        colorMode(HSB, 360, 100, 100);
```

```
        background(360);
        strokeWeight(4);
        for (int h = 0; h < 360; h++) {
          float katWRadianach = radians(h);
             for (int s = 0; s < width/2;s++) {
                float x = width/2
                        + cos(katWRadianach) * s;
                float y = height/2
                        + sin(katWRadianach) * s;
                stroke(h, s/2, 100);
                point(x, y);
                }
        }
        stroke(360);
        noFill();
        strokeWeight(5);
        ellipse(width/2, height/2, width, height);
        noLoop();
}
```

Zwróćmy przede wszystkim uwagę na linijkę z metodą **colorMode**. Metoda ta przyjmuje od **1** do **4** argumentów. W naszym przypadku przekazaliśmy **4**, z których pierwszy jest stałą określającą jedną z dwóch dostępnych w Processingu przestrzeni barw. Do tego momentu domyślnie używaliśmy **RGB**, lecz wpisując **HSB** przeszliśmy do alternatywnego modelu. Poza tym możemy określić zakresy poszczególnych kanałów i zamiast domyślnego zakresu **(0,255)** możemy ustalić bardziej nam odpowiadający. Wpisaliśmy **360** dla kąta barwy oraz po **100** dla nasycenia i jasności ze względu

na fakt, że odpowiada to przyjętemu standardowi zapisu. Niemniej w tym miejscu możemy ustalić zakresy zupełnie dowolnie, również dla przestrzeni **RGB**. Często oszczędza nam to mapowania argumentów przy parametryzacji koloru. W następnej linijce wywołaliśmy **background** z wartością **360**, co odpowiada wcześniejszym wywołaniom z argumentem **255** w **RGB** , określającym nasycenie szarości w domyślnym zakresie **(0,255)**.

Możemy zauważyć jak powoli nasze programy stają się strukturalnie coraz bardziej wyrafinowane. W tym przypadku zagnieździliśmy jedną pętlę w drugiej. To wewnątrz tych pętli są umieszczone najistotniejsze dla programu polecenia. Z kolei polecenia w zewnętrznej pętli służą głównie kosmetycznym celom. Na początku pierwszej pętli tworzymy instancję zmienną lokalną **katWRadianach**, która przechowywać będzie wartość kąta. Pętla jest wywoływana **360** razy, więc wartość ta zmienia się w taki sposób, jakbyśmy przy każdej iteracji pokonywali jednostopniowy odcinek okręgu. Kąt ten jest przechowywany w jednostkach zwanych radianami, przyjmuje więc wartości od **0** do **2***pi. Konwersja stopni na radiany jest konieczne ze względu na użycie we wewnętrznej pętli funkcji trygonometrycznych, które wymagają argumentów podanych w radianach. Zagnieżdżona pętla jest wywoływana za każdym przebiegiem pętli zewnętrznej i przy innej wartości kąta. Wewnątrz zagnieżdżonej pętli umieściliśmy większość rysujących metod. Wcześniej jednak obliczamy współrzędne punktu. Wykorzystujemy do tego znaną z trygonometrii zależność współrzędnych punktu okręgu od trzech argumentów – promienia okręgu, długości odcinka łączącego środek okręgu ze środkiem układu współrzędnych oraz cosinusa (sinusa dla **y**) kąta

(mierzonego w radianach) pomiędzy promieniem a osią **x** (inaczej osią odciętych). W następnych krokach ustalamy kolor dla punktów i konturów oraz umieszczamy kolejny punkt na płótnie. Jak można łatwo obliczyć, wewnętrzna pętla jest wywoływana **360*200** razy, czyli umieszcza na płótnie dokładnie **72000** punkty, każdy w innym kolorze. Oczywiste jest, że nie byłoby to możliwe bez posłużenia się mechanizmem pętli.

5.6 ANIMACJA PARAMETRYCZNA

Wrócimy teraz do programu z piłką. Naszym celem będzie kontrolowanie koloru tła poprzez pozycję piłki w poziomie. Połączymy więc dwa szkice – animację piłki z animacją gradientu. Właściwie wykonaliśmy już większość pracy. Jedyna rzeczą, którą musimy zrobić, to umiejętnie przypisać wartości zmiennej, odpowiedzialnej za kolor przejściowy do współrzędnej piłki względem osi rzędnych. Definicja **draw** wygląda następująco.

```
void draw() {
        wspolczynnik = map(pozycjaY,
        0, height, 0, 1);
        float r = R + DELTA_R * wspolczynnik;
        float g = G + DELTA_G * wspolczynnik;
        float b = B + DELTA_B * wspolczynnik;
        kolorOd = color(r, g, b);
        background(kolorOd);
        zderzenieX =
               jestZderzenie(pozycjaX+srednicaX/2, width);
        zderzenieY =
               jestZderzenie(pozycjaY+srednicaY/2, height);
```

```
        if (zderzenieX) {
                kierunekX = -1*kierunekX;
                predkoscX = kierunekX * predkoscX;
                srednicaX = kierunekX * srednicaX;
        }
        if (zderzenieY) {
                kierunekY = -1*kierunekY;
                predkoscY = kierunekY
                        * (int)random(3, 7);
                srednicaY = kierunekY
                        * (int)random(3, 7);
        }
        pozycjaX = pozycjaX + predkoscX;
        pozycjaY = pozycjaY + predkoscY;
        rysujKola(pozycjaX, pozycjaY, promien, REK);
}
```

Nie umieściłem na listingu powyżej utworzenia instancji zmiennych globalnych, definicji **setup** oraz definicji dwóch funkcji – **jestZderzenie** i **rysujKola**, ponieważ pozostały identyczne jak w przykładzie w podrozdziale poświęconym rekurencji. Należy jedynie zwrócić uwagę na fakt, że zmienna **R**, określająca ilość rekurencji, została przemianowana na **REK**, w celu uniknięcia konfliktu ze zmienną przechowującą wartość kanału czerwonego. Efekt parametryzacji uzyskaliśmy podmieniając zmienną o nazwie **licznik** na **pozycjaY**. Wartość zmiennej **pozycjaY** po przeniesieniu w odpowiedni zakres, przekazana zostaje zmiennej **wspolczynnik**. Ta druga zmienna ogrywa ważną rolę w tworzeniu koloru, który przekazujemy jako parametr

do **background**. Ten przykład obrazuje idee stojącą za specyficznym sposobem pracy – większe programy powstają w efekcie hybrydyzacji mniejszych fragmentów, zwanych czasami po angielsku *snippets (pl. skrawki, fragmenty)*. Środowisko Processingu sprzyja temu w wyjątkowym stopniu.

5.7 LICZBY PSEUDOLOSOWE

W kodzie z poprzedniego przykładu użyliśmy po raz pierwszy funkcji **random**. Z jej udziałem wprowadzamy do programu efekt przypadkowości. W tym konkretnym przykładzie sprawiliśmy, że prędkości piłki w obu wymiarach zmieniają się po zetknięciu z przeszkodą z góry założoną nieokreślonością. Funkcja **random** przyjmuje jeden bądź dwa argumenty. W przypadku wywołana z jednym argumentem losowana jest liczba z przedziału pomiędzy 0 a wartością argumentu. Po podaniu dwóch argumentów otrzymamy na wyjściu liczbę z zakresu ograniczonego ich wartościami. Należy podkreślić, że uzyskane liczby nie są czysto przypadkowe lecz powstają w wyniku działania deterministycznego algorytmu. Złudzenie przypadkowości jest jednak bardzo przekonujące. Zbiory uzyskane w ten sposób mają rozkład normalny. Możemy wobec tego stworzyć przekonującą symulację procesu stochastycznego, zwanego błądzeniem losowym.

```
int x1, y1;
int x2, y2;

void setup() {
        size(200, 200);
        background(255);
```

```
        smooth();
        x1 = width/2;
        y1 = height/2;
}

void draw() {
        noStroke();
        fill(255,220);
        rect(0, 0, 400, 400);
        stroke(0);
        strokeWeight(2);
        x2 = x1 + (int)random(-5, 5);
        y2 = y1 + (int)random(-5, 5);
        line(x1, y1, x2, y2);
        x1 = x2%width;
        y1 = y2%height;
        if (x1 < 0) {
                x1+=width;
        }
        if (y1 < 0) {
                y1+=height;
        }
}
```

Pierwsze dwie linijki w **`draw`** wprowadzają prostą implementację efektu zwanego rozmyciem w ruchu (ang. *motion blur*) – zakrywamy półprzezroczystą płaszczyzną poprzedni rysunek. Dwa ostatnie wyrażenia warunkowe służą pozostawieniu obiektu w obrębie płótna. W poprzednich przykładach z piłką, odbijała się ona od krawędzi, w tym przypadku rysowana „cząsteczka" pojawia się po przeciwległej stronie okna.

Możemy wreszcie stworzyć ciąg liczb pseudolosowych. Poniższy program rysuje dwuwymiarowy wykres takiego ciągu. Dodatkowo umieszcza wylosowane liczby w pliku tekstowym, który możemy następnie odczytać w dowolnym edytorze tekstowym. Do tego celu użyliśmy obiektu typu **`PrintWriter`**. Jeżeli zmienną tego typu połączymy kropką z dobrze znaną metodę nazwie **`println`**, to przekazany jej tekst zostanie zapisywany do pliku tekstowego a nie jak dotychczas wyświetlony w konsoli .

```
void setup() {
        size(600, 200);
        PrintWriter liczby =
                createWriter("liczby pseudolosowe.txt");
        int x1 = 0;
        int y1 = height/2;
        int y2;
        for (int x2 = 5; x2<=width; x2+=5) {
                y2 = (int)random(height);
                line(x1, y1, x2, y2);
                x1 = x2;
                y1 = y2;
                liczby.println(y2);
        }
        liczby.flush();
        liczby.close();
}
```

Podkreślić należy, że w przypadku komputerowych generatorów liczb losowych, nie możemy mówić o czystej przypadkowości. Kolejne wyrazy otrzymanego przez nas ciągu są wynikiem dobrze określonych procedur, których złożoność jest niewielka w porównaniu

z bogactwem przyrody. Oczywiście algorytmiczne generatory liczb losowych są bardzo przekonujące i trudno jest odróżnić efekt ich działania od tych, będących zapisem zdarzeń występujących w naturze. Należy jednak pamiętać, że w przypadku komputerów mamy zawsze do czynienia mniej lub bardziej z symulacją. Dla przykładu, w poniższym kodzie zaimplementujemy algorytm wykorzystujący operację modulo do przeprowadzenia symulacji przypadkowości.

```
int m = 1466; /* moduł, im większy, tym
większa różnorodność wyników */
int a = 3; /* mnożnik, dowolny, byle był
większy lub równy 2 oraz mniejszy od m*/
int c = 23; /* inkrementacja, dowolna
wartość ze zbioru {0,m} */
int x = 66; /* y(0) wartość początkowa,
warunek jak przy c */

/*
y(n+1) = (a* y(n) + c)mod m
Iteracja generująca ciąg liczb pseudolosowych.
Warunek konieczny:
2 <= a < m, 0 <= c <= m, 0 <= y(0) < m
*/

void setup() {
        println("Iteracja:");
        pseudolosuj(x);
        println("Rekurencja:");
        pseudolosuj(x,50);
}
```

```
void pseudolosuj(int _x){
        for (int n = 1; n < 50; n++) {
                _x = ((a * _x) + c) % m;
                println(_x);
        }
}
void pseudolosuj(int _x, int i) {
        if (i == 0) {
                return;
        }
        else {
                _x = ((a * _x) + c) % m;
                println(_x);
                pseudolosuj(_x, i-1);
        }
}
```

Wzór matematyczny i wymogi gwarantujące poprawne działanie zostały podane w komentarzu do kodu. Zaimplementowaliśmy funkcję na dwa różne sposoby, dające identyczne wyniki – jako iterację lub rekurencję. Działanie rekurencji trudniej jest prześledzić, niemniej ta struktura charakteryzuje się trudną do wytłumaczenia elegancją, czy wręcz tajemniczością. Działania przeprowadzaliśmy na liczbach całkowitych. Dział matematyki zajmujący się relacjami pomiędzy liczbami całkowitymi nazywa się teorią liczb i odgrywa bardzo istotną rolę m.in. w kryptografii. W przypadku naszego pseudolosowego generatora możemy oczywiście zamienić **`int`** na **`float`**, bez żadnych konsekwencji. Jeśli zależy nam na liczbach z przedziału **`(0-1)`** tzw. znormalizowanych, możemy podzielić wyniki przez zmienną **`m`**. Warto też spędzić trochę czasu na eksperymenty

z użyciem tego prostego algorytmu. Może to zaowocować ciekawymi sekwencjami cyfr stanowiących bazę do mniej abstrakcyjnych zastosowań.

W kolejnych paragrafach zobaczymy w jaki sposób możemy odrobinę zakłócić determinizm maszyny.

5.8 CHAOS Struktura **for** pozwali nam zaimplementować metody numeryczne, związane ze zjawiskiem chaosu[18]. Iterując proste równanie kwadratowe, zwane odwzorowaniem logistycznym, uzyskamy obraz zachowania się układu dynamicznego. Układ dynamiczny charakteryzuje się m.in. wrażliwością na warunki początkowe, tzn. niewielka zmiana w doborze wartości argumentów przekazanych funkcji opisującej taki układ przynieść może diametralnie różne wyniki. System dynamiczny zachowuje się w sposób wykraczający poza ekstrapolację statycznego opisu. Oznacza to, że jeśli będziemy modelować zachowanie takiego układu wielokrotnie, za każdym razem rozpoczynając od innego zbioru argumentów reprezentujących warunki początkowe, to wyniki iteracji będą dla nas nieprzewidywalne. W przypadku odwzorowania logistycznego, które ma zastosowanie m.in. w opisie zmian wielkości populacji zwierząt, warunek początkowy jest symbolizowany przez współczynnik **R**, przybierający wartości z przedziału **(0,4)**. Powyżej wartości zbliżonej do **3.569954672**, układ zaczyna zachowywać się chaotycznie. Uruchamiając następujący kod będziemy się mogli przekonać o następującej prawidłowości – im wartości **R** będą bliższe **4**, tym wykresy skojarzone z różnymi wartościami tego argumentu zaczną odbiegać od siebie co-

18 Bardzo przystępnym i „niematematycznym" wprowadzeniem w zagadnienia związane z chaosem jest książka Jamesa Gleicka „Chaos", Zysk i S-ka, Poznań 1996,

raz widoczniej. Dzieje się tak nawet wówczas kiedy różnice pomiędzy R są bardzo niewielkie – rzędu **1.0E-6**.

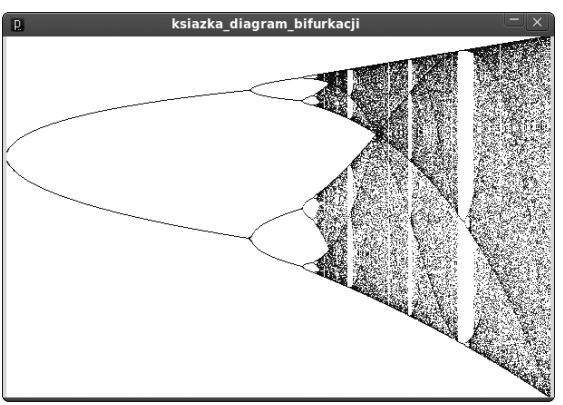
```
size(600, 400);
background(128);
float R = random(3.569954672,4);
for (int i = 0; i < 2; i++) {
        stroke(i*255);
        float xn;
        float px = 0.5;
        float py = 0;
        int n = 5;
        for (n = 5; n < width; n+=5) {
                xn =R*px*(1.0-px);
                float y = height - xn*height;
                line(n, y, n-5, py);
                px = xn;
                py = y;
        }
        println("R: "+R);
        R += 0.000001;
}
```

Jeżeli zmodyfikujemy odrobinę powyższy kod, uzyskamy tzw. diagram bifurkacji, posiadający cechy zbioru samopodobnego (fraktalu).

```
float R;
void setup() {
        size(600, 400);
        background(255);
        stroke(10, 220);
        smooth();
```

```
        for (int x = 0; x <= width; x++) {
                R = map(x, 0, width, 0.75, 1);
                float xn = 0.5;
                for (int n = 0; n <= height; n++) {
                        xn = 4*R*xn*(1.0-xn);
                        int y = height - (int)(xn*height);
                        if (n > height/2) {
                                point(x, y);
                        }
                }
        }
}
```

W powyższym kodzie użyliśmy drugiej wersji odwzorowania logistycznego, w której **R** przyjmuje wartości ze zbioru **{0,1}**. Przemnażamy go następnie przez współczynnik równy **4**.



WYKRES BIFURKACJI ODWZOROWANIA LOGICZNEGO

6 INTERAKCJA I OBIEKTY

6.1 OBSŁUGA ZDARZEŃ MYSZKI

Dotąd nasze programy w momencie ich uruchomienia stawały się dla nas niedostępne. W czasie, gdy program realizował ustalony przez nas plan, nie mieliśmy już żadnego wpływu na jego działanie. Oczywiście jest to w dużym stopniu sprzeczne z doświadczeniem użytkownika komputera, który dzięki klawiaturze i myszce jest w ciągłej interakcji z programem.

Processing posiada dwie pary zmiennych systemowych umożliwiających pracę z myszką. Pierwsza przechowuje aktualną pozycję myszki, druga natomiast jej położenie z poprzedniego wywołania metody **`draw`**. Pokazane jest to w poniższym przykładzie.

```
void setup() {
        size(400,400);
}


void draw(){
        point(mouseX,mouseY);
        int predkoscX = mouseX - pmouseX;
        int predkoscY = mouseY - pmouseY;
        if(predkoscX > 0 || predkoscY > 0 ){
                println("myszka porusza się z predkoscia "
                        +predkoscX+" piksel/klatka");
                println("myszka porusza się z predkoscia "
                        +predkoscX+" piksel/klatka");
        }
}
```

Wprowadzamy następnie mechanizm odpowiadający na zmianę stanów myszki. Każdy ze stanów myszki – czyli ruch, wciśnięcie przycisku, przeciąganie, itd. – jest ujęty jako zdarzenie (ang. *Event*). Mechanizm, który reaguje na zmianę (różnie pojmowanych) stanów, nazywamy obsługą zdarzeń (ang. *event handling*). Najprościej rzecz ujmując, obsługa zdarzeń polega na zdefiniowaniu metod, które są wywoływane tylko w sytuacji wystąpienia określonego zdarzenia. Zdarzenie i metoda najczęściej nazwane są bardzo podobnie, np. wciśnięcie przycisku myszki wywołuje metodę **`mousePressed`** (dosłownie *myszkaWciśnięta*), zwolnienie przycisku – **`mouseReleased`** (pol. *myszkaZwolniona*), itp. Rozpatrzmy poniższy przykład.

```
void setup() {
}

void draw(){
}

void mousePressed(){
        print("a");
}

void mouseReleased(){
        print("e");
}

void mouseDragged(){
        print("i");
}

void mouseMoved(){
        print("o");
}
```

Zasada programowania poleceń wydawanych za pośrednictwem myszki jest bardzo prosta. Definicje powyższych metod służą do grupowania poleceń, które chcielibyśmy uruchomić poprzez relewantny stan urządzenia. Wynik wszelkich metod rysunkowych zostanie pokazany w oknie programu dokładnie tak, jakbyśmy rysowali wewnątrz **setup** lub **draw.** Należy pamiętać, że jeżeli chcemy korzystać z obsługi zdarzeń, to w kodzie konieczne jest umieszczenie definicji tych dwóch metod, nawet jeśli pozostawimy ich ciała puste. Obecność **draw** sprawia, że program będzie „nasłuchiwał" komunikatów, których źródłem jest urządzenie wskazujące.

Dysponujemy już wystarczającymi środkami, żeby stworzyć nasz własny program rysunkowy, na podobieństwo klasycznego szkicownika Paint. Zaczniemy od definicji metody rysującej paletę kolorów.

```
void rysujPalete(int x, int szerokosc, int wysokosc) {
        for (int h = 0; h < 360; h++) {
                int b = 100;
                for (int s = 0; s < 100; s++) {
                        if (s<100) {
                             stroke(h, s, 100);
                        }
                        else {
                             stroke(h, 100, b--);
                        }
                        strokeWeight(5);
                        int _y = (int)map
                              (h, 0, 360, 0, wysokosc);
                        int _x = (int)map
                              (s, 0, 100*2, 0, szerokosc);
                        point(x+_x, _y);
                }
```

```
        }
        stroke(360);
        line(x,0,x,wysokosc);
}
```

Ta metoda wyrysuje prostokąt o parametrach przekazanych zgodnie z sygnaturą – lewym górnym punkcie zakotwiczenia, szerokości i wysokości. Pętle w jej ciele, są bardzo zbliżone w strukturze do tych w przykładzie rysującym koło **HSB**. Metoda **stroke** jest w nich wywoływana **72000** razy i tyle teoretycznie kolorów powinniśmy mieć do dyspozycji w naszym szkicowniku. W praktyce jednak będzie ich tyle, ile wyniesie przemnożenie szerokości i wysokości prostokąta palety. Wywołanie metody **rysujPalete** możemy umieścić zarówno w **draw** jak i w **setup**. Nie planujemy jednak odświeżać płótna w **draw** z przyczyn, które staną się za moment oczywiste. Ponadto wyrysowanie palety jest nam potrzebne tylko raz, zaraz po uruchomieniu programu. Oszczędzi to ponadto mocy obliczeniowej procesora – program nie będzie musiał wykonywać pętli za każdym odświeżeniem płótna. Kod definicji **setup**, **draw** oraz utworzenie instancji zmiennych programu jest przedstawiony na poniższym listingu.

```
int szerokoscPalety = 50;
color kolorPedzla = 0;
void setup() {
        size(450, 400);
        colorMode(HSB, 360, 100, 100);
        background(10, 5, 100);
        rysujPalete(0, szerokoscPalety, height);
        smooth();
}

void draw() {
}
```

Rozmiar okna programu jest kwestią wyboru. Wybrałem format płótna zbliżony do kwadratu o podstawie **400** pikseli. Dodałem do podstawy **50** pikseli, rezerwując te miejsce na paletę. Ustalenie przestrzeni barw na **HSB** jest konieczne dla poprawnego działania metody **rysujPalete**, można się więc zastanowić nad przeniesieniem do wnętrza ciała **rysujPalete** linijki w której ustalamy przestrzeń barw. Możemy też kod opatrzyć stosownym komentarzem, informującym o warunku poprawności działania kodu. Ustaliliśmy arbitralnie kolor podłoża na biel w ciepłym odcieniu. Jeśli ktoś woli inny kolor, może dowolnie zmodyfikować ten parametr. Istotne jest jednak umieszczenie **background** przed wywołaniem **rysujPalete**. W przeciwnym wypadku zasłonimy to co wcześniej zostało umieszczone w oknie. Jest to też główny powód, dla którego definicję **draw** pozostawiliśmy pustą. Chcemy bowiem zachować wszystko to, co zostało wyrysowane w poprzednich klatkach. Przestrzeń okna będzie spełniać funkcję cyfrowego palimpsestu.

Jeżeli uruchomimy kod na tym etapie, uzyskamy obraz notatnika, lecz bez możliwości interakcji. Dodamy teraz kod umożliwiający rysowanie i wybór koloru narzędzia rysującego. Obraz wyświetlany w oknie programu jest grafiką rastrową (inaczej bitmapą), co oznacza, że jest siatką niezależnych pikseli. Rysowanie w programie komputerowym polega na wpływaniu poprzez urządzenie wskazujące na wartości kolorystyczne przyjmowane przez pojedynczy piksel. Pokażemy teraz w jaki sposób będziemy mogli to uzyskać za pośrednictwem mechanizmu obsługi zdarzeń myszki. Dla przypomnienia – pusta definicja **draw** sprawia, że program monitoruje stany myszki. Umieścimy teraz de-

finicje trzech poznanych stanów – **mouseReleased, mouseDragged** oraz **mouseMoved**. Pierwszy z nich występuje, kiedy zwalniamy przycisk myszki, drugi oznacza sytuację gdy poruszamy myszką przy wciśniętym przycisku, nazywamy to przeciąganiem i stąd nazwa metody. Metoda **mouseReleased** sprawdza czy wydusiliśmy przycisk. Nas interesuje, czy te zdarzenie będzie miało miejsce nad obszarem palety. Jeśli tak, pobierany zostaje kolor piksela znajdujący się na aktualnej współrzędnej myszki. Następnie jest on przekazywany metodzie **stroke**, które w ogólności ustala kolor wyrysowywanych punktów oraz linii. – a w tym konkretnym przypadku narzędzia rysującego. Dodatkowo poprzez metodę **cursor**, zmieniamy znak kursora na krzyżyk (ang. *cross*).

```
void mouseReleased() {
        if (mouseX < szerokoscPalety) {
                cursor(CROSS);
                kolorPedzla = get(mouseX, mouseY);
                stroke(kolorPedzla);
        }
}
```

W **mouseDragged** znajduje się główna metoda rysująca. Jest to po prostu dobrze znane wywołanie metody **line** ze współrzędnymi dwóch punktów. Linia jest rysowana pomiędzy aktualnym i poprzednim położeniem myszki. Dodatkowo kursor zmienia się na rączkę. Aby uniknąć sytuacji zachodzenia linii na paletę, metodę rysującą umieściliśmy w bloku poprzedzonym warunkiem, który sprawdza współrzędne obu położeń myszki, tj. aktualnego i poprzedniego. Ponadto sprawdza czy zmienna **kolorPedzla** przechowuje inną wartość niż domyślna.

```
void mouseDragged() {
        if (mouseX > szerokoscPalety
        && pmouseX > szerokoscPalety
        && kolorPedzla!= 0) {
        cursor(HAND);
        line(mouseX, mouseY, pmouseX, pmouseY);
        }
}
```

Wreszcie piszemy metodę dotyczącą wyglądu kursora. Jej zadanie jest natury kosmetycznej – zmienia sposób jego wyświetlania w zależności od położenia.

```
void mouseMoved(){
        if(mouseX > szerokoscPalety){
                cursor(MOVE);
        } else {
                cursor(CROSS);
        }
}
```

Zakończymy pisanie tej aplikacji zaimplementowaniem mechanizmu monitorującego komunikaty nadchodzące z klawiatury. Dopiszmy więc kolejną definicję metody, tym razem uruchomianej przez zdarzenie zwolnienia klawisza '**s**'. Wzbogaci to program o możliwość zapisania naszej pracy rysunkowej w formie pliku graficznego.

```
void keyReleased() {
        if (key=='s') {
                PImage obrazek = get(szerokoscPalety,
                        0, width-szerokoscPalety, height);
                obrazek.save("obrazek.jpg");
        }
}
```

Nazwa metody jest analogiczna do metody nasłuchującej zwolnienia klawisza myszki. Dodaliśmy do powyższej definicji warunek sprawdzający, czy zwolnionym klawiszem jest ten z etykietą `'s'`. Zwróćcie uwagę, że litera `'s'` jest umieszczona w pojedynczym cudzysłowie. To istotne, gdyż zmienna systemowa **`key`** jest typu **`char`** a nie **`String`**. Do pobrania fragmentu okna, które pełni rolę płótna, posłużyliśmy się funkcją **`get`**, która, jeśli wywołana z czterema parametrami w miejsce dwóch, zwróci nie kolor jednego piksela, lecz fragment bitmapy określony przez dwie skrajne współrzędne (lewą górną i prawą dolną). Zostaje on zapisany w zmiennej typu **`PImage`**. Następnie, posługując się metodą **`save`**, zapisujemy ten obszar jako plik graficzny w folderze programu. Format graficzny ustalamy poprzez podanie rozszerzenia w nazwie pliku. W tym wypadku jest to jpg, ale nic nie stoi na przeszkodzie, żeby zapisać obrazek jako **`tiff`**, **`png`** czy **`targa`**.

W ostatniej definicji użyliśmy po raz drugi wywołania metody w taki sposób, że kolejno napisaliśmy nazwę zmiennej, kropkę i wywołanie metody. Taka notacja nazywa się nieformalnie notacją kropki i mam nadzieję, że stanie się zrozumiała kiedy poznamy podstawy programowania obiektowego.

Na tym zakończyliśmy pisanie prostego programu graficznego. Moglibyśmy go jednak w dalszym ciągu udoskonalać – nawet tak rozbudowany program jak Photoshop musiał się kiedyś rozpocząć od kilku linijek kodu.

6.2 PROGRAMOWANIE OBIEKTOWE

Nasze programy stają się coraz bardziej wyrafinowane pod względem używanych pojęć. Rozpoczęliśmy od pojęcia naj-

prostszego – zmiennej. Następnie wprowadziliśmy metody, funkcje, iteracje, wyrażenia warunkowe. Dzięki nim byliśmy w stanie pisać programy, które inaczej nie byłby możliwe lub w najlepszym wypadku bardzo trudne do zrealizowania. Każde kolejne pojęcie istotnie wpłynęło równocześnie na naszą zdolność formułowania problemów jak i ich rozwiązywania. Jednocześnie pamiętajmy, że program w dalszym ciągu pozostaje zbiorem wyrażeń.

Wprowadzimy teraz kolejne pojęcie, które w sposób diametralny wpłynie na nasze myślenie o programowaniu, mianowicie pojęcie klasy. Nie ogranicza się ono do programowania – odgrywa znaczącą rolę w logice formalnej, a także w naukach ścisłych, takich jak fizyka czy chemia. Wszędzie klasa znaczy rzecz podobną – definicję obiektów lub zjawisk mających wspólną cechę. W taki też sposób posługujemy się nim w języku potocznym. Oznacza to, że wprowadzenie klas pozwala na organizację kodu w jednostki znaczeniowo zbliżone do obiektów w realnym świecie. Ma to dwie zalety. Po pierwsze problemy programistyczne mogą zostać sformułowane w sposób analogiczny do tych napotykanych w otaczającym nas środowisku. Będziemy więc w stanie rozpatrywać interesujący nas problem za pomocą pojęć należących do jego przestrzeni. Po drugie może się stać zupełnie na odwrót, tzn. zagadnienia, które do tej pory trudno było umieścić w kontekście programowania, będzie można modelować poprzez adekwatne do świata zewnętrznego relacje między obiektami klas. Taki paradygmat programowania nosi nazwę programowania zorientowanego obiektowo lub krócej programowania obiektowego.

Definicja klasy jest ogólnym wzorcem służącym tworzeniu wielu, autonomicznych obiektów, zwa-

nych instancjami klasy. Tworzone obiekty są więc jakby wytłaczane z jednej matrycy, zawierającej ogólną specyfikację cech charakteryzujących rodzinę obiektów. Zatem definicja klasy pozwala na stworzenie instancji w dowolnej ilości, które łączyć będzie zbiór cech i zachowań a różnić ich indywidualna realizacja, np. możemy zdefiniować klasę materialnego obiektu poprzez założenie, że wszystkie obiekty we wszechświecie posiadają masę (wyłączając foton), prędkość, przyśpieszenie, rozmiar oraz znajdują się w jakimś określonym współrzędnymi punkcie przestrzeni. Różnią się natomiast co do indywidualnej realizacji tych cech, bez znaczenia czy mówimy o krześle na którym właśnie siedzimy, przelatującym samolocie czy odległej o tysiące lat gwiezdnych planecie. W takim ujęciu, zarówno te wymienione przed chwilą obiekty jak i jakikolwiek inny istniejący we wszechświecie obiekt (o ile możemy w nim wyróżnić powyższe cechy), będzie dla nas przede wszystkim instancją bardzo ogólnej klasy materialnego przedmiotu.

6.3 DEFINIOWANIE KLASY

Składnia definiująca klasę w Processingu ma kilka nowych pojęć, ale nie różni się zbytnio od tego co już poznaliśmy. Najogólniej mówiąc klasa jest zbiorem właściwości oraz stanów, czyli inaczej zbiorem zmiennych i funkcji, które nazywa się ogólnie członkami (ang. *members*).

```
class <NAZWA_KLASY> {
    <ZMIENNA_1>
    ...
    <ZMIENNA_N>
```

```
        <KONSTRUKTOR_1>
        ...
        <KONSTRUKTOR_N>



        <METODA/FUNKCJA_1>
        ...
        <METODA/FUNKCJA_N>
    }
```

Definicję klasy należy rozumieć jako następne wyrażenie wprowadzone do kodu. Umieszczamy je na tym samym logicznym poziomie co definicje metod, funkcji i zmiennych o zakresie globalnym. Definicję zaczynamy od słowa kluczowego **`class`**, po którym następuje nazwa klasy. Nazwa ta może być dowolnym ciągiem znaków, (z zastrzeżeniem reguł obowiązujących również w nazewnictwie zmiennych i funkcji). Do konwencji należy rozpoczynanie nazwy klasy od dużej litery, nie jest to jednak wymóg. Po nazwie otwieramy nawias klamrowy, w którym umieszczamy wyrażenia charakteryzujące tworzony typ. W tym nawiasie umieszczamy wszystko, co jest relewantne dla danej klasy (tzn. deklaracje i stworzenie instancji zmiennych, definicje metod i funkcji), tak jakbyśmy pisali w ciele klasy samodzielny program. Wszelkie reguły składniowe, które do tej pory poznaliśmy, nadal obowiązują. Definiujemy więc funkcje, metody w taki sam sposób, jak robiliśmy to dotychczas.

Jedno z wyrażeń umieszczonych w definicji klasy zasługuje na specjalne omówienie, zarówno ze względu na rolę, którą pełni w definicji, jak i na skład-

nię. Specjalnym wyrażeniem w klasie jest konstruktor, czyli funkcja, która służy do utworzenia instancji danego typu. Rolą konstruktora jest stworzenie unikalnego obiektu z ogólnej definicji. Pomimo wyjątkowej roli, składnia konstruktora odbiega od definicji zwykłych funkcji w niewielkim stopniu. Różni się tylko tym, że nazwa klasy jest jednocześnie nazwą funkcji i jej typem, oraz tym, że nie umieszczamy w ciele konstruktora słowa kluczowego **return**. W definicji klasy możemy umieścić kilka konstruktorów różniących się sygnaturą. Sygnaturę projektujemy analogicznie do sygnatur metod i funkcji.

```
<NAZWA_KLASY> (<SYGNATURA>){
        <WYRAŻENIA>
}
```

Konstruktor wywołujemy w trakcie tworzenia zmiennej, wskazującej na instancją klasy (inaczej obiekt). Wywołanie umieszczamy po operatorze przypisania i poprzedzamy je słowem kluczowym **new**.

```
<NAZWA_KLASY> <NAZWA> = new
<NAZWA_KLASY>(<ARGUMENTY ...>);
```

Obiekty stworzone ze wzorca klasy są danymi złożonymi z wielu różnorakich danych cząstkowych oraz procedur, określonych definicją klasy. To implikuje bardzo istotną właściwość. Mianowicie wartością zmiennej, do której przypisujemy obiekt, nie jest sam obiekt, lecz adres do miejsca w pamięci komputera, gdzie został on przez program umieszczony w chwili utworzenia. Obiekt więc znajduje się jakby na zewnątrz zmiennej. Porównanie takich zmiennych do pudełka, tak jak to robiliśmy w przypadku zmiennych prymi-

tywnych, przestaje być adekwatne. Zmienne do których przypisujemy obiekty, są raczej wskaźnikami i stąd też ich ogólna nazwa – zmienne wskaźnikowe. Ponieważ obiekt rezyduje na zewnątrz, możliwa jest też sytuacja, w której wskazuje na niego kilka zmiennych.

Stworzymy klasę **Pilka** definiującą typ obiektu, który symulowaliśmy już wcześniej w przykładach z rekurencją i gradientem. W tym przypadku zobaczymy jak możemy konstruować programy poprzez analogię do świata fizycznego.

```
class Pilka {
        float srednica;
        Pilka() {
                srednica = 30.0;
        }
        Pilka(float s) {
                srednica = s;
        }
}
```

Powyższy kod umieśćmy poniżej definicji **setup**. Na razie nasza piłka posiada tylko jedną cechę – średnicę, do której przypisujemy wartość w momencie tworzenia instancji klasy **Pilka**. Utworzenie instancji klasy należy rozumieć jako nadanie jednostkowej formy przedmiotowi, który dotychczas był pojmowany tylko jako idea. Tak też należy myśleć o definicji klasy – jako idei, która służy za wzorzec obiektom uosabiającym jej cechy i zachowania. Pierwszy konstruktor w definicji klasy posiada często pustą sygnaturę. W **Pilka** zdefiniowaliśmy jeszcze jeden, który wymaga liczby jako argumentu. Zostanie on przekazany w ciele konstruktora do zmiennej instancji o nazwie **srednica**.

```
Pilka pilka = new Pilka(30.0);
print(pilka.srednica);
```
```
30.0
```

W pierwszej linijce stworzyliśmy obiekt klasy i przypisaliśmy go do zmiennej. W kolejnej wydrukowaliśmy wartość zmiennej należącej do instancji. Dostęp do zmiennych obiektu jest możliwy poprzez notację, którą nazywa się nieformalnie notacją kropki. Przede wszystkim możemy pobrać wartość przechowywaną przez zmienną, znajdującą się w instancji i przypisać pobraną wartość do innej zmiennej, znajdującej się na zewnątrz struktury obiektu. Możemy też przypisać nową wartość do zmiennej wewnątrz instancji. Musimy tylko pamiętać, że identyfikator zmiennej należącej do obiektu składa się z nazwy zmiennej, wskazującej na obiekt oraz nazwy samej zmiennej, połączone ze sobą kropką. Ogólnie, reguły postępowania ze zmiennymi prymitywnymi należącymi do obiektów nie różnią się od tych, dotyczących zwykłych zmiennych prymitywnych. Klasa służy jedynie do grupowania (agregacji) zmiennych różnych typów, w taki sposób, że wspólne ich występowanie jest znaczeniowo usprawiedliwione.

Powyższy, bezpośredni sposób obchodzenia się ze zmiennymi instancji, jest często zastępowany pośrednim, polegającym na posługiwaniu się funkcjami zwanymi po angielsku **`getters`** oraz **`setters`**. Będziemy mówić o nich później.

6.4 SŁOWO KLUCZOWE THIS

Rozbudujemy teraz klasę **Pilka** dodając do niej inne kluczowe właściwości oraz stany, takie jak współrzędne w układzie kartezjańskim, prędkość, przyśpieszenie, wygląd i detekcję kolizji.

Mamy co prawda już gotową metodę wykorzystującą rekurencję do rysowania gradientowej kuli, lecz uprościmy sobie na razie sprawę i potraktujemy wygląd w sposób schematyczny, jako pojedyncze koło o średnicy przekazanej do konstruktora. Dodamy jeszcze jeden konstruktor, tym razem wymagający trzech argumentów – współrzędnych x, y oraz średnicy. Musimy zatem dodać zmienne x i y do ogólnej charakterystyki klasy. Obecność w definicji kilku konstruktorów daje nam wybór sposobu utworzenia instancji. Obiekt klasy **Pilka** będzie mógł być utworzony każdym z konstruktorów. Dodanie dodatkowego konstruktora umożliwi zmianę domyślnych wartości niektórych zmiennych w momencie tworzenia instancji klasy.

```
class Pilka {
        float x = random(200);
        float y = random(200);
        float srednica = 20;
        Pilka() {
                srednica = 30.0;
        }
        Pilka(float s) {
                srednica = s;
        }
        Pilka(float x, float y,float s) {
                this.x = x;
                this.y = y;
                srednica = s;
        }
}
```

Zmienne **x** i **y** posiadać będą wartości domyśle. W zależności od użytego konstruktora możemy je pozostawić niezmienione, bądź przypisać do nich nowe wartości. Zwróćcie uwagę, że w sygnaturze drugiego konstruktora nazwaliśmy zmienne występujące tylko w bloku kodu konstruktora identycznie jak zmienne powyżej, o zasięgu całej klasy. Żeby uniknąć dwuznaczności i być pewnym, że przekazana wartość trafi do właściwej zmiennej, użyliśmy słowa kluczowego **this** (ang. *ten, ta, to, obecny*) i połączyliśmy je kropką z nazwą zmiennej. Słowo **this** należy rozumieć jako wskazanie, że zmienna po nim występująca ma zasięg klasy i nie należy ją mylić ze zmienną lokalną o tej samej nazwie, stworzoną wewnątrz bloku kodu zagnieżdżonego w definicji klasy. W obecnym przykładzie zmienna **x** zadeklarowana w sygnaturze konstruktora jest użyta tylko w chwili wywołania konstruktora, z kolei wyrażenie **this.x** reprezentuje zmienną zdefiniowaną o jeden stopień wyżej, będącą jedną z cech klasy.

Ogólnie mówiąc, referencja **this** odsyła zawsze do obiektu, w definicji którego się znajduje.

6.5 METODY I FUNKCJE W KLASIE

Jesteśmy teraz gotowi do umieszczenia metody rysującej naszą piłkę na ekranie. Jej definicję piszemy pod konstruktorami. Jest to zwyczajna metoda, nie wyróżniająca się niczym nowym od tych napisanych do tej pory. Metoda ma pustą sygnaturę i zawiera tylko wywołanie **ellipse** z parametrami o wartościach zmiennych **x** i **y** instancji.

```
class Pilka {
```

```
        float x = random(200);
        //(...)

        Pilka() {
                //(...)
        }
        Pilka(float s) {
                //(...)
        }
        Pilka(float x, float y,float s) {
                //(...)
        }

        void rysuj() {
                ellipse(x,y,srednica,srednica);
        }
}
```

Dostęp do metody **rysuj** jest zapośredniczony poprzez obiekt klasy, więc musimy użyć notacji kropki. Jeżeli umieścimy w **setup** lub **draw** poniższy kod, wyrysujemy obiekt w oknie programu.

```
pilka.rysuj();
```

Jak na razie będzie to statyczne kółko. Chcąc je animować, dodamy zmienne oznaczające prędkość w pionie i poziomie oraz metodę, która w oparciu o te zmienne, uaktualni położenie obiektu. Dopiszmy więc do definicji klasy następujący kod.

```
/* zmienne umieśćmy
```

```
nad konstruktorami */
float vx = random(-10,10);
float vy = random(-10,10);

/* natomiast tę metodę
pod metodą 'rysuj' */
void poruszaj(){
        x += vx;
        y += vy;
}
```

Zanim przejdziemy dalej, poznamy przydatną właściwość środowiska Processingu. W miarę pracy nad aplikacją kod staje się coraz dłuższy, co prowadzić może szybko do zmniejszenia przejrzystości. Mamy jednak możliwość pogrupowania kodu w osobnych zakładkach. Zakładkę możemy utworzyć na dwa sposoby, wciskając konfigurację przycisków Ctrl (jabłko Mac OS X), Shift i 't', albo też klikając na ikonę ze strzałką po prawej stronie paska ikon edytora. Kiedy to zrobimy, pojawi się pytanie o nazwę pliku. Nazwa może być dowolna, lecz na ten moment nazwijmy po prostu **`Pilka`**. Przenieśmy teraz całą deklarację klasy z pierwszej zakładki do tej nowo utworzonej. W folderze projektu powstał w międzyczasie nowy plik o nazwie przez nas nadanej i rozszerzeniu pde. Możemy utworzyć dowolną ilość takich zakładek, rozdzielając kod ze względu na rolę pełniącą w projekcie – w głównej zakładce umieścić metody **`setup`** i **`draw`** a w pozostałych definicje funkcji, klas czy obsługę zdarzeń. Ułatwia to w znacznym stopniu pracę.

W naszej symulacji piłki będziemy używać praw mechaniki klasycznej w sposób mniej formalny. W metodzie **`poruszaj`** uaktualniamy pozycję piłki po-

przez dodanie w obu osiach wielkości, która określa o jaką odległość w pikselach zmieni się jej położenie po każdym wywołaniu tej metody. Jeżeli umieścimy wywołanie w **draw**, otrzymamy ciągły ruch figury. Ruch zawdzięczamy niezerowej prędkości przedmiotu przy bezruchu układu odniesienia. Dlatego też użyłem **v** (ang. *velocity*) jako prefiksu nazw zmiennych, tak jak to się przyjęło w notacji fizycznej. Ponieważ nie umieściliśmy jeszcze detekcji zderzeń, piłka bardzo szybko zniknie z naszego widoku. Żeby to naprawić zdefiniujemy predykat (funkcję zwracającą wartość boolowską), sprawdzający czy dochodzi do zderzenia obiektu z przeszkodą. Dodamy również metody kierujące reakcją przedmiotu na taką sytuację. Dodajmy do definicji klasy następujący kod.

```
boolean jestZderzenie(float x, float y) {
        float odleglosc = dist(this.x, this.y, x, y);
        if (odleglosc <= srednica/2) {
                return true;
        }
        else {
                return false;
        }
}

void odbijX() {
        vx*=(-1);
}
void odbijY() {
        vy*=(-1);
}
boolean blokada = false;

void zablokuj(){
```

```
        blokada = true;
}

void odblokuj(){
        blokada = false;
}

boolean jestZablokowana(){
        return blokada;
}
```

W pierwszej linijce zastosowaliśmy funkcję matematyczną **dist**, obliczającą odległość między dwoma punktami w przestrzeni kartezjańskiej. Pełna formuła nie jest skomplikowana – jest to pierwiastek z sumy kwadratów różnic współrzędnych x i y. W zależności od tego, czy odległość jest mniejsza czy większa od promienia koła, predykat zwraca odpowiednio prawdę lub fałsz. Dwie proste metody **odbijX** i **odbijY** zmieniają znak liczb określającej prędkość w poziomie i pionie na przeciwny. Zdefiniowanie oddzielnych metod dla każdej osi jest użyteczne w symulacji odbicia piłki od płaszczyzny (w poziomie lub pionie), gdy trajektoria ruchu zmienia się o **90** stopni (inaczej – pi radianów). Ten sposób detekcji kolizji posłuży nam do sprawdzania, czy piłka nie napotkała na swojej drodze krawędzi okna. Dodaliśmy też zmienną boolowską **blokada**. Metody **zablokuj** oraz **odblokuj** pozwolą nam na zmianę bieżącej jej wartości. Zmodyfikujemy też metodę **poruszaj**, uzależniając wykonanie działań **x += vx** i **y += vy** od warunku przyjęcia wartości **false** przez zmienną **blokada**. Dodając ponadto obsługę zdarzeń, stworzymy ciekawą interakcję. Będziemy mogli złapać piłkę, przeciągnąć ją, jak również

nadać jej nową prędkość bazując na aktualnej i poprzedniej pozycji myszki. Umieśćmy poniższy kod w głównej zakładce.

```
Pilka pilka;
color c;

void setup() {
        size(400, 400);
        pilka = new Pilka(200, 200, 100.0 );
        c = color(random(255),
        random(255), random(255));
}

void draw() {
        background(c);
        pilka.poruszaj();
        /* Sprawdza czy piersza
        piłka nie napotyka 'ściany'*/
        if (pilka.jestZderzenie(pilka.x, 0)) {
                pilka.odbijY();
        }
        if (pilka.jestZderzenie(pilka.x, width)) {
                pilka.odbijY();
        }
        if (pilka.jestZderzenie(0, pilka.y)) {
                pilka.odbijX();
        }
        if (pilka.jestZderzenie(height, pilka.y)) {
                pilka.odbijX();
        }
        pilka.rysuj();
}
```

```
void mousePressed() {
        if (pilka.jestZderzenie(mouseX, mouseY)) {
                pilka.zablokuj();
        }
}
void mouseReleased() {
        if (pilka.jestZablokowana()) {
                pilka.odblokuj();
                pilka.vx = mouseX-pmouseX;
                pilka.vy = mouseY-pmouseY;
        }
}
void mouseDragged() {
        if (pilka.jestZablokowana()) {
                pilka.x = mouseX;
                pilka.y = mouseY;
        }
}
```

Jeśli pobieramy lub zmieniamy bieżącą wartość jakiejkolwiek zmiennej należącej do klasy, lecz nie w sposób bezpośredni (tj. poprzez notację kropki), ale poprzez dedykowane metody (tak jak to możemy zaobserwować na przykładzie zmiennej **`blokada`** i skojarzonych z nią **`metod zablokuj`**, **`odblokuj`** oraz predykatu **`jestZablokowana`**), to mówimy o takiej zmiennej, że jest chroniona[19]. Funkcja przekazująca wartość zmiennej jest tak zwaną funkcją otrzymującą (ang. *getter*) i według konwencji nazwę takiej funkcji powinniśmy zacząć od słowa angielskiego **`get`**. Z kolei metodę zmieniającą

19 Nie jest to do końca prawda, gdyż pomijamy w rozważaniach zagadnienie tzw. modyfikatorów dostępu (ang. *access modifiers*). Celem jednak jest zapoznanie czytelnika z konwencją JavaBeans.

wartość nazywamy metodą ustalającą (ang. *setter*) i nazwę tej metody powinniśmy zacząć od słowa **`set`**. W naszym kodzie używamy języka polskiego tam gdzie jest to możliwe, stąd też nazwy metod i funkcji o wspomnianych rolach spełniają tylko pośrednio konwencję przyjętą przez angielskojęzycznych programistów.

W ten sposób napisaliśmy nasz pierwszy program z użyciem własnej klasy. Wrócimy jeszcze na jakiś czas do klasy **`Pilka`**, żeby zdefiniować jej kolejne właściwości i stany w celu lepszej symulacji fizycznego obiektu. Od tej chwili klasy będą nam towarzyszyć na każdym kroku, umożliwiając formułowanie znaczeń w programie poprzez intuicje nieodbiegające zbytnio od codziennej interakcji z materialnym światem.

7 TABLICE

7.1 TABLICE ZMIENNYCH PRYMITYWNYCH

Z pojęciem tablic łączy się procedura indeksowania. Niemal na każdym kroku spotykamy się z sytuacją, gdy pewne przedmioty czy nawet osoby są ponumerowane, np. domy, mieszkania, osoba na liście obecności, tramwaje, itd. Idea, która za tym stoi, zakłada istnienie zbioru podobnych do siebie obiektów (np. ulica, jako zbiór domów) oraz procedury dostępu do indywidualnych obiektów poprzez wskazanie na jednoznacznie przypisany do obiektu numer, nazywany też indeksem. Z matematycznego punktu widzenia, indeksowanie jest funkcją przekształcającą zbiór liczby naturalnych w inny, dowolnie zdefiniowany zbiór. Poznamy teraz najprostszą ze zmiennych złożonych, których rolą jest przechowywanie innych zmiennych. Utworzenie instancji tzw. tablicy (ang. *array*) może się odbyć na dwa sposoby. Składnia pierwszego z nich wygląda następująco.

```
<TYP>[] <NAZWA> = {<WARTOŚĆ_0,
WARTOŚĆ_1, .... , WARTOŚĆ_N>};
```

Jeżeli więc chcielibyśmy mieć tablicę zmiennych typu **int**, zawierającą **5** różnych wartości, możemy to osiągnąć w taki sposób:

```
int[] hasla = {123456, 23415, 93450, 23110, 34253};
```

Stworzyliśmy zmienną **hasla**, która faktycznie składa się z **5** różnych wartości. Możemy posługiwać się nią podobnie jak z każdą inną zmienną, np. przekazać ją metodzie jeśli wymaga tego sygnatura. Jedną z takich metod, z sygnaturą akceptującą tabelę, jest **println**.

```
println(hasla);
```

```
[0] 123456
[1] 23415
[2] 93450
[3] 23110
[4] 34253
```

Wydruk odkrywa bardzo ważną właściwość tablic – indeksowanie rozpoczyna się od liczby **0** a kończy na liczbie mniejszej o **1** od długości (ang. *length*) tablicy, np. jeżeli chcemy dowiedzieć jaka liczba znajduje się na **4** miejscu tablicy, musimy użyć indeksu o numerze **3**. Ogólnie, procedura dostępu do zmiennej stojącej na n-miejscu wygląda następująco.

```
<NAZWA>[N-1];
```

Jeżeli więc chcemy pobrać wartość z tablicy, musimy zastosować powyższą składnię. Przykład zastosowania znajduje się poniżej.

```
int pinDoKarty = hasla[3];
```

Nic nie stoi na przeszkodzie działania w drugą stronę. Nową wartość możemy przypisać bezpośrednio lub za pośrednictwem zmiennej.

```
hasla[0] = 123999;
// albo
hasla[4] = pinDoKarty;
```

Drugi sposób na wprowadzenie tablicy do kodu wygląda następująco.

```
<TYP>[] <NAZWA> = new
<TYP>[<DŁUGOŚĆ_TABLICY>];
```

Chcąc utworzyć tablicę przechowującą **wartości float** napiszemy:

```
float [] jakiesLiczby = new float [5];
```

Zadeklarowana w ten sposób tablica nie posiada jeszcze żadnych wartości. Zapełniamy ją używając poznanej przed chwilą syntaksy z zastosowaniem indeksu.

```
jakiesLiczby[0] = 0.3245;
jakiesLiczby[1] = 0.435;
jakiesLiczby[2] = 0.3655;
jakiesLiczby[3] = 0.133;
jakiesLiczby[4] = 0.9005;
```

Należy podkreślić, że tablice są jednolite jeśli chodzi o typ przechowywanych przez nie wartości, tzn. tablica **float[]** może przechowywać tylko **float**, itp. Niezależnie od tego jaki sposób utworzenia wybierzemy, rozmiar tablicy jest stały, tzn. możliwość ustalenia długości tablicy mamy tylko w momencie jej utworzenia. Musimy też wystrzegać się przed próbą pobrania lub przypisywania wartości z pozycji powyżej ostatniego indeksu tablicy. Jeśli jednak zdarzy się to, np. przy próbie pobrania wartości – **print(jakiesLiczby[5]);**, to pojawi się następujący błąd.

```
Exception in thread "Animation Thread" java.
                                    : 5
```

Często jednak nie pamiętamy rozmiaru wszystkich tablic, na których dokonujemy operacji. Na całe szczęście tablice posiadają użyteczny parametr, który przechowuje jej długość – **`length`** (pol. *długość*). Jeżeli chcemy sprawdzić lub przypomnieć sobie rozmiar tablicy, sięgamy po prostu po ten parametr tak jak to robiliśmy w przypadku klas – następując kropką i słowem `lenght` nazwę zmiennej tablicy. Przydaje się to najczęściej przy iteracji. Mamy wówczas pewność, że nie wykroczymy poza dopuszczony zakres.

```
for(int i = 0; i < jakiesLiczby.length; i++){
        print(jakiesLiczby[i]);
}
```

Tablice są obiektami złożonymi, czego konsekwencją jest to, że przekazując funkcji zmienną tego typu, nie przekazujemy kopi zmiennej lecz odnośnik do niej. Oznacza to, że każda operacja na argumencie wpłynie na stan przekazanej zmiennej. Pokazuje to następujący przykład.

```
void setup(){
        int[] a = {
                1,2,3,4};
        zmien(a);
        println(a);
}
void zmien(int[] b ){
        for(int i = 0; i < b.length; i++){
                b[i] = b.length - i;
        }
}
```

```
[0] 4
[1] 3
[2] 2
[3] 1
```

7.2 PRACA Z GOTOWYMI PLIKAMI GRAFICZNYMI

Poznaliśmy już wcześniej klasę **PImage**. Za jej pomocą zapisaliśmy owoc naszej pracy rysunkowej do pliku graficznego. Załóżmy jednak, że chcielibyśmy wzorem edytora graficznego, móc nie tyle rysować, co zmieniać istniejący już obraz. Procedura jest bardzo prosta, wystarczy bowiem funkcją **loadImage** przypisać plik graficzny do instancji klasy **PImage** i następnie wywołać metodę **image** z tą instancją jako argumentem. Jeśli wywołamy metodę **selectInput**, to otworzy się nam systemowa przeglądarka plików i będziemy w stanie wskazać myszką plik graficzny niekoniecznie znajdujący się w katalogu *data*. Zmodyfikujmy **setup** w starym kodzie szkicownika w sposób przedstawiony na poniższym listingu.

```
String plik = selectInput();
PImage tlo = loadImage(plik);
size(tlo.width+szerokoscPalety, tlo.height);
colorMode(HSB, 360, 100, 100);
image(tlo, szerokoscPalety, 0);
rysujPalete(0, szerokoscPalety, height);
smooth();
```

Na uwagę w powyższym kodzie zasługuje sposób ustalenia wymiarów okna programu. Przekazanie wymiarów obrazka metodzie **size** sprawi, że okno programu dynamicznie dostosuje się do wymiarów pliku graficznego.

7.3 CYFROWE PRZETWARZANIE OBRAZU

Wiemy już, że obraz wyświetlany na ekranie składa się z ustawionym obok siebie pikseli, potrafimy też pobierać wartość pojedynczego piksela funkcją get, jak również ustalać piksel metodą set. W Processingu dostępna jest dodatkowo tablica zawierająca wszystkie piksele wyświetlanego okna. Piksele tworzą obraz w kolejności podobnie jak litery tekst na stronie książki – pierwszy piksel o indeksie **0** znajduje się w lewym górnym rogu ekranu. Indeks następnie rośnie w prawo do końca rzędu, przechodzi do kolejnej linii, i tak sukcesywnie do prawego dolnego rogu okna. Zmienna przechowująca piksele obrazu nazywa się po prostu **pixels**. Żeby mieć do niej dostęp, musimy ją najpierw umieścić w buforze programu poprzez wywołanie metody **loadPixels**. Bufor jest rodzajem tymczasowej pamięci, w której program przechowuje dane potrzebne do realizacji zadania zanim zostaną one przekazane do procesu programu. Należy pamiętać, że zmiany wartości pikseli pozostaną dla nas niezauważone do momentu wywołania metody **updatePixels**, która przemieszcza tablicę pikseli do głównego wątku programu.

Okno programu jest ostatecznie bitmapą, tak jak każdy plik grafiki. Nic dziwnego zatem, że również **PImage** posiada tablicę **pixels**. Sposób postępowania z nią jest analogiczny. Możemy więc wyświetlić zewnętrzny plik obrazu poprzez przekopiowanie pikseli z obrazka do pikseli okna obrazu. Przedstawia to poniższy program.

```
void setup(){
        PImage nebula =
                loadImage("http://farm8.staticflickr.com/"
                        +"7087/7136928779_57afbfd088_d.jpg");
        size(nebula.width,nebula.height);
        loadPixels();
        nebula.loadPixels();
        for(int i = 0; i < width*height; i++){
                pixels[i] = nebula.pixels[i];
         }
        updatePixels();
}
```

Przekopiowaliśmy wartości z jednej tablicy do drugiej. Obie były o tej samej długości, równej przemnożeniu przez siebie szerokości i wysokości obrazka. Pojawia się jednak pytanie dlaczego zadawać sobie trud i używać dłuższej, i bardziej technicznej procedury, skoro mogliśmy posłużyć się metodą **image**. W powyższym przykładzie przekopiowaliśmy piksele z jednej tablicy do drugiej. Jeśli jednak pomiędzy pobraniem piksela a umieszczeniem go w tablicy docelowej, zmienimy jego wartość lub przypiszemy go do innego indeksu, efekt będzie zgoła inny niż proste wyświetlenie pliku graficznego. Innymi słowy, zaczniemy wówczas cyfrowo przetwarzać obraz. Poniżej pokazuję, jak wyświetlić obraz do góry nogami.

```
for(int i = 0; i < width*height; i++){
        int odwrotnyIndeks = width*height - i -1;
        pixels[i] = nebula.pixels[odwrotnyIndeks];
}
```

Ale naprawdę ciekawe rozpoczyna się kiedy, zaczynamy manipulować wartościami poszczególnych kanałów. W poniższy sposób uzyskamy negatyw.

```
for(int i = 0; i < width*height; i++){
        color kolorWejsciowy = nebula.pixels[i];
        float r = 255-red(kolorWejsciowy);
        float g = 255-green(kolorWejsciowy);
        float b = 255-blue(kolorWejsciowy);
        color kolorWyjsciowy = color(r,g,b);
        pixels[i] = kolorWyjsciowy;
}
```

Zamiana obrazu kolorowego na czarno-biały nie przysparza dużego problemu.

```
for(int i = 0; i < width*height; i++){
        color kolorWejsciowy = nebula.pixels[i];
        float r = red(kolorWejsciowy);
        float g = green(kolorWejsciowy);
        float b = blue(kolorWejsciowy);
        float bw = (r+b+g)/3;
        color kolorWyjsciowy = color(bw);
        pixels[i] = kolorWyjsciowy;
}
```

Idąc tym tropem, możemy również ustawić wartość progową, która zadecyduje o pokolorowaniu pojedynczego piksela na czarny lub biały.

```
int prog = 122;
for(int i = 0; i < width*height; i++){
        color kolorWejsciowy = nebula.pixels[i];
        float r = red(kolorWejsciowy);
        float g = green(kolorWejsciowy);
        float b = blue(kolorWejsciowy);
        float bw = (r+b+g)/3;
        color kolorWyjsciowy;
```

```
        if (bw > prog) {
                kolorWyjsciowy = color(255);
        }
        else {
                kolorWyjsciowy = color(0);
        }
        pixels[i] = kolorWyjsciowy;
}
```

Bez trudu ograniczymy też paletę barw do określonej liczby kolorów, na przykład do **8**.

```
int poster = 2;
for (int i = 0; i <width*height; i++) {
        color kolorWejsciowy = nebula.pixels[i];
        int r = 255/poster*(int)
         ceil(map(red(kolorWejsciowy), 0, 255, 0, poster));
        float g =255/poster*(int)
          ceil(map(green(kolorWejsciowy), 0, 255, 0, poster));
        float b = 255/poster*(int)
          ceil(map(blue(kolorWejsciowy), 0, 255, 0, poster));
        color kolorWyjsciowy = color(r, g, b);
        pixels[i] = kolorWyjsciowy;
}
```

Powyższa operacja matematyczna tylko pozornie wygląda skomplikowanie. W rzeczywistości mapujemy liczbę z przedziału **(0, 255)** w przedział **(0,2)**, usuwamy część ułamkową i przemnażamy ją przez współczynnik, który sprawia, że wartość przyjmuje jedną z **2** wartości leżących w wejściowym zakresie. Współczynnik **poster** jest równy **2**, ponieważ liczba możliwych kolorów jest permutacją **3** kanałów – czerwonego, zielonego i niebieskiego – a ponieważ każdy z kanałów przy takim

ustawieniu może przyjąć tylko jedną z dwóch wartości, w rezultacie da to **8** (2^3) różnych kolorów.

7.4 KOLOR JAKO PARAMETR ANIMACJI

Zanim przejdziemy do bardziej zaawansowanej formy przetwarzania obrazu, pokażę, jak wykorzystać iterację tablicy, w tym wypadku **pixels**, do sterowania animacją. Zaczniemy jednak od stworzenia klasy, która będzie modyfikacją wcześniej przez nas stworzonej klasy **Pilka**.

Zdefiniujemy ogólny model wirtualnej cząsteczki – z prędkością, przyśpieszeniem i kilkoma jeszcze innymi właściwościami. O klasie **Pilka** możemy pomyśleć jako o abstrakcji specyficznego rodzaju cząsteczki. Stąd też większość kodu definicji klasy, która zaraz napiszemy, przekopiujemy z definicji **Pilka**. Nowa klasa o nazwie **Czasteczka** wygląda następująco (pogrubione zostały fragmenty w których dokonaliśmy modyfikacji klasy **Pilka**).

```
class Czasteczka {
        float srednica = 20;
        float x = 0;
        float y = 0;
        float vx = 0;
        float vy = 0;
        boolean blokada = false;
        float kat = 0;
        float predkosc = 0;
        float przyspieszenie = 0;

        Czasteczka() {
        }
```

```
Czasteczka(float s) {
        srednica = s;
}

Czasteczka(float x, float y, float s) {
        this.x = x;
        this.y = y;
        srednica = s;
}

void rysuj() {
        pushMatrix();
        translate(x, y);
        if (predkosc > 0){
                rotate(atan2(vy,vx));
        } else {
                rotate(atan2(vy,vx)+PI);
        }
        ellipse(0, 0, srednica, srednica);
        stroke(0);
        float dlugosc =predkosc*5;
        line(srednica/2,
        0, dlugosc+srednica/2, 0);
        line(dlugosc+srednica/2,
        0, srednica/2+cos(HALF_PI/2)*dlugosc/2,
        sin(HALF_PI/2)*dlugosc/2);
        line(dlugosc+srednica/2,
        0, srednica/2+cos(-HALF_PI/2)*dlugosc/2,
        sin(-HALF_PI/2)*dlugosc/2);
        popMatrix();
}

void poruszaj() {
        x += vx;
        y += vy;
}
```

```
        void zmienKatO(float delta) {
                kat+= delta;
                ustawPredkosc(kat, predkosc);
        }

        void ustawPredkosc(float kat, float predkosc) {
                this.kat = kat;
                this.predkosc = predkosc;
                vx = cos(kat)*predkosc;
                vy = sin(kat)*predkosc;
        }

        void przyspieszO(float przyspieszenie) {
                this.przyspieszenie
                        += przyspieszenie;
                ustawPredkosc(kat,
                predkosc+przyspieszenie);
        }


// Poniższy kod jest identyczny z poprzednią wersją.
        boolean jestZderzenieZ(float x, float y) {
                //(...)
        }
        void odbijX() {
                //(...)
        }
        void odbijY() {
                //(...)
        }
        void zablokuj(){
                //(...)
        }
```

```
        void odblokuj(){
                //(...)
        }
        boolean jestZablokowana(){
                //(...)
        }
}
```

Wprowadziliśmy trzy nowe zmienne **float**, przechowujące kąt, prędkość oraz przyśpieszenie obiektu. Kąt obrotu obiektu będzie ściśle powiązany z jego prędkością oraz przyśpieszeniem. Wystarczy skorzystać z trygonometrycznej reprezentacji współrzędnych x i y. Prędkość będzie zatem wektorem o początku znajdującym się w centrum obiektu. Wartość przyśpieszenia jest dodawana do aktualnej długości wektora prędkości. W zależności od znaku wartości przyśpieszenia, obiekt przyśpiesza lub hamuje. Dodaliśmy też do definicji klasy metody, które pozwolą nam na sterowanie obiektem, poprzez zmianę wartości przechowywanych przez powyższe zmienne. Metody **zmienKatO** oraz **zmienPrzyspieszenie** zmieniają kąt i przyśpieszenie względem aktualnej wartości, co ułatwia płynne sterowanie ruchem obiektu.

Znacznej zmianie została poddana metoda **rysuj**. Przede wszystkim zastosowanie pary metod **pushMatrix/popMatrix**, ułatwia kontrolę nad translacją i rotacją obiektu. Obiekt porusza się i obraca względem zewnętrznego układu współrzędnych, nie naruszając relacji wewnątrz swojej własnej struktury – **ellipse** jest rysowane niezmiennie w centrum wewnętrznego układu współrzędnych. Inną modyfikacją

jest dodanie strzałki, która zmienia swoją długość i nachylenie w zależności od prędkości i obrotu cząsteczki. To wizualizuje nam zarówno kierunek jak i prędkość obiektu. Możemy teraz dodać w głównej zakładce definicje **setup**, **draw** oraz obsługę zdarzeń klawiatury, dzięki której będziemy mogli wywoływać odpowiednie metody z wnętrza instancji obiektu klasy **Czasteczka**.

```
Czasteczka czasteczka;
void setup() {
        size(400, 400);
        czasteczka = new Czasteczka(200, 200, 20);
        czasteczka.ustawPredkosc(PI, 3);
}
void draw() {
        background(200);
         czasteczka.poruszaj();
        if (czasteczka.x < 0) {
                czasteczka.x+=width;
        }
        if (czasteczka.y < 0) {
                czasteczka.y+=height;
        }
        czasteczka.x%=width;
        czasteczka.y%=height;
        czasteczka.rysuj();
}

void keyPressed() {
        println(czasteczka.przyspieszenie);
        if (key==CODED) {
                if (keyCode==LEFT) {
                        czasteczka.zmienKatO(-PI/32);
                }
```

```
                if (keyCode==RIGHT) {
                        czasteczka.zmienKatO(PI/32);
                }
                if (keyCode==UP) {
                        czasteczka.przyspieszO(0.2);
                }
                if (keyCode==DOWN) {
                        czasteczka.przyspieszO(-0.2);
                }
        }
}
```

Stosunkowo niewielka ilość kodu pozwoliła nam uzyskać symulację sterowania obiektem. Następnie wykorzystamy tablicę pikseli jako listę komend, decydujących o zachowaniu się cząsteczki. Dodajmy więc kilka zmiennych do kodu powyżej **setup**.

```
PImage obrazek;
int indeks = 0;
int kolejnaZmiana;
int klatka = 0;
```

W **setup** dopiszmy polecenia tworzące obiekt `PImage` oraz ładujące tablicę pikseli.

```
obrazek =
        loadImage("http://farm8.staticflickr.com/"
        +"7087/7136928779_57afbfd088_d.jpg");
nebula.loadPixels();
kolejnaZmiana = (int)map(red(obrazek.
        pixels[indeks]), 0, 255, 32, 64);
```

Wreszcie dodajmy do **draw** następujący fragment.

```
klatka++;
if (kolejnaZmianaCo==klatka) {
        indeks++;
        klatka=0;
        color piksel = obrazek.pixels[indeks];
        kolejnaZmiana = (int)
                map(red(piksel), 0, 255, 2, 8);
        czasteczka.zmienKatO(map(green(piksel),
        0, 255, -PI/16, PI/16));
        czasteczka.przyspieszO(map(blue(piksel),
        0, 255, -.2,0.2));
}
if(indeks == obrazek.pixels.length){
        indeks = 0;
}
```

Po uruchomieniu kodu, animacja cząsteczki będzie determinowana zawartością kanałów **RGB** pojedynczych pikseli. Jeżeli zachowamy obsługę zdarzeń klawiatury, to będziemy mogli ingerować w ruch cząsteczki przez wciśnięcie odpowiednich klawiszy za pomocą klawiatury, co umożliwi nam na interakcję z autonomicznym obiektem.

7.5 TABLICE OBIEKTÓW

Zorkiestrowanie trzech struktur – tablicy, pętli oraz obiektu – może się stać naprawdę potężnym środkiem służącym tworzeniu generatywnej animacji. To co będziemy chcieli uzyskać w następnym przykładzie, to symulacja fontanny, tzn. grupy cząstek, które wyłanianą

się z jednego punktu, by następnie rozproszyć się w różnych kierunkach. Kod na listingu poniżej pokazuje, jak łatwe jest przejście od operowania jednym obiektem do całej ich grupy. Przede wszystkim musimy stworzyć tablicę obiektów. W naszym przypadku będzie to tablica przechowująca **100** instancji **Czasteczka**. Oczywiście długość tablicy określiliśmy w sposób dowolny – może być zarówno znacznie większa jak i mniejsza. Procedura dostępu do pojedynczego obiektu jest już nam znana – odbywa się za pośrednictwem indeksu. W gruncie rzeczy jedyną nowością jest tutaj odpowiednie połączenie znanych nam już struktur języka.

```
Czasteczka[] czasteczki = new Czasteczka[100];

void setup() {
        size(400, 400);
        for (int i = 0; i < czasteczki.length; i++) {
                czasteczki[i] = new
                        Czasteczka(200, 100, 5);
                czasteczki[i].
                ustawPredkosc(random(PI,
                  TWO_PI), random(3, 6));
        }
}

void draw() {
        background(255);
        for (int i = 0; i <czasteczki.length; i++) {
                czasteczki[i].poruszaj();
                if (czasteczki[i].x >
                        width || czasteczki[i].x <0) {
                        czasteczki[i].x = 200;
                        czasteczki[i].y = 100;
```

```
                czasteczki[i].
                        ustawPredkosc(random(PI,
                          TWO_PI), random(3, 6));
            }
            if (czasteczki[i].y >
                    height || czasteczki[i].y < 0) {
                    czasteczki[i].x = 200;
                    czasteczki[i].y = 100;
                    czasteczki[i].
                        ustawPredkosc(random(PI,
                          TWO_PI), random(3, 6));
            }
            czasteczki[i].rysuj();
        }
}
```

Do definicji klasy dodajmy jeszcze jedną właściwość – grawitację. Będzie to wartość dodawana do prędkości w pionie – rodzaj przyśpieszenia w jednym kierunku. Konsekwentnie będziemy musieli dodać jedną linijkę do metody o nazwie **poruszaj**.

```
float grawitacja = 0.2;
void poruszaj() {
        x += vx;
        vy += grawitacja;
        y += vy;
}
```

Warty odnotowania jest fakt, że stosunek zmiany w złożoności kodu do zmian w zawartości animacji nie jest proporcjonalny, tzn. stosunkowo niewielka zmiana w długości kodu znacznie przyczyniła się do widocznego efektu działania programu.

Jak było to powiedziane wcześniej, zmienne prymitywne przekazują wartość, natomiast zmienne złożone referencję. Gdy przekazujemy metodzie argument w postaci zmiennej złożonej, to wskazujemy tylko na obiekt, na którym chcemy wykonać operacje. Nie przekazujemy instancji samej w sobie. Wszystkie zmiany zostaną wykonane na źródłowym obiekcie na który zmienna wskazuje. Tą właściwość zmiennych wskaźnikowych wykorzystuje inna wersja pętli **for** o następującej strukturze.

```
for(<TYP> <INSTANCJA> : <TABLICA>){
    <WYRAŻENIA>
}
```

Możemy użyć powyższej struktury tylko do iterowania tablic złożonych ze zmiennych wskaźnikowych. Nie musimy się wówczas martwić o indeks i pojemność tablicy. Mamy też pewność, że zdefiniowane w bloku kodu instrukcje, zostaną wykonane dokładnie tyle razy ile, zmiennych znajduje się w tablicy. Zobaczmy jak możemy tę składnię użyć do obsługi zdarzeń w naszej symulacji cząsteczek.

```
void keyPressed() {
       for (Czasteczka czasteczka : czasteczki) {
              if (key==CODED) {
                     if (keyCode==LEFT) {
                     czasteczka.zmienKatO(-PI/32);
                     }
                     if (keyCode==RIGHT) {
                          czasteczka.zmienKatO(PI/32);
                     }
                     if (keyCode==UP) {
                         czasteczka.przyspieszO(0.1);
                     }
```

```
                    if (keyCode==DOWN) {
                        czasteczka.przyspieszO(-0.1);
                    }
                }
        }
}
```

Zmienna lokalna **czasteczka** reprezentuje po kolei każdą zmienną w tablicy **czasteczki**. Poprzez tą zmienną lokalną manipulujemy oryginalnymi obiektami.

7.6 KONWOLUCJA

Processing jest bardzo dogodnym narzędziem do pracy z grafiką rastrową, ponieważ dostęp do wartości piksela jest bardzo ułatwiony poprzez takie funkcje jak **get** czy **set**. Jednak dopiero użycie tablicy **pixels** sprawia, że praca z bitmapami może stać się wyjątkowo interesująca. Konwolucja, lub inaczej mnożenie splotowe, jest bardzo elastyczną techniką pracy z grafiką. Jest to operacja, którą wykonujemy na każdym pikselu obrazka. Polega na przemnożeniu wartości zarówno piksela jak i innych, bezpośrednio z nim sąsiadujących pikseli, przez wartości pobrane ze specjalnej macierzy. Następnie sumujemy wszystkie iloczyny i zastępujemy wartość środkowego piksela otrzymaną sumą.

W tym celu przechodzimy przez tablicę pikseli w następujący sposób. Zaczynając od drugiego rzędu i drugiej kolumny, pobieramy wartość piksela. Następnie tworzymy tymczasową macierz o wymiarach **3x3**, składającą się z bieżącego piksela (umieszczonego w centrum macierzy) oraz **8** otaczających go pikseli. „Nakładamy" na tę macierz inną macierz o tych samych wymiarach, w taki sposób, że wartość stojącą

w **n-rzędzie** i **n-kolumnie** w jednej macierzy, mnożymy przez wartość stojącą na tym samym miejscu w drugiej macierzy. Ostatecznie sumujemy wszystkie dziewięć iloczynów i wynikiem zastępujemy wartość początkowego piksela. W ten sposób postępujemy ze wszystkimi pikselami obrazu, z wyjątkiem pikseli znajdujących się na jego krawędziach, ze względu na fakt, że nie posiadają one wszystkich potrzebnych do operacji „sąsiadów". Strukturę macierzy można bardzo łatwo odtworzyć w Processingu poprzez tablicę składającą się z innych tablic. Wówczas pierwszy indeks tablicy oznaczać będzie wiersz, a drugi kolumnę.

W zależności od wartości współczynników mnożenia, możemy uzyskać bardzo różne efekty – od rozmycia obrazu po efekt reliefu. Oto program, w którym sprawdzimy działanie kilku najczęściej stosowanych konfiguracji „jądra".

```
// Rozmycie 1
float [][] rozmycie1 = {
{ 1.0/9, 1.0/9, 1.0/9 } ,
{ 1.0/9, 1.0/9, 1.0/9 } ,
{ 1.0/9, 1.0/9, 1.0/9 } };

// Rozmycie 2
float [][] rozmycie2 = {
{ 1.0/9, 2.0/9, 1.0/9 } ,
{ 2.0/9, 4.0/9, 2.0/9 } ,
{ 1.0/9, 2.0/9, 1.0/9 } };

// Wyostrzenie 1
float [][] wyostrzenie1 = {
{ -1, -1, -1 } ,
```

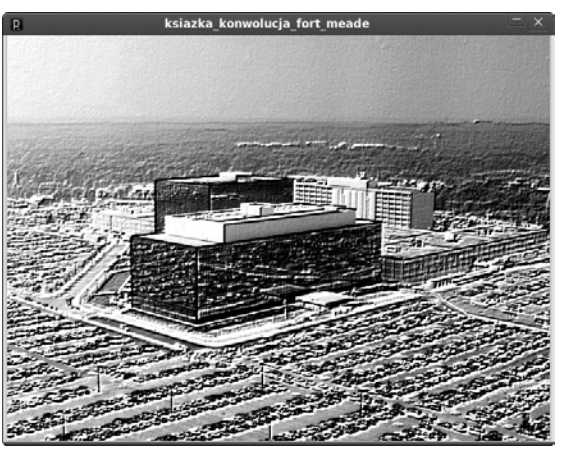
```
{ -1, 9, -1 } ,
{ -1, -1, -1 } };

// Wyostrzenie 2
float [][] wyostrzenie2 = {
{ 0, -1, 0 } ,
{ -1, 5, -1 } ,
{ 0, -1, 0 } };

// Wykrywanie krawędzi
float [][] wykrywanieKrawedzi = {
{ 0, 1, 0 } ,
{ 1, -4, 1 } ,
{ 0, 1, 0 } };

// Wzmocnienie krawędzi
float [][] wzmocnienieKrawedzi = {
{ 0, 0, 0 } ,
{ -1, 1, 0 } ,
{ 0, 0, 0 } };

// 'relief'
float [][] relief = {
{ -2, -1, 0 } ,
{ -1, 1, 1 } ,
{ 0, 1, 2 } };

float [][] jadro = wyostrzenie1;

void setup() {
```

```
        PImage obrazek
                = loadImage("Fort_Meade.jpg");
        size(obrazek.width, obrazek.height);
        loadPixels();
        obrazek.loadPixels();
        for (int x = 1; x < width-1; x++) {
                for (int y = 1; y < height-1; y++) {
                        float czerwonySuma = 0;
                        float zielonySuma = 0;
                        float niebieskiSuma = 0;
                        for (int jx = 0; jx < 3; jx++) {
                           for (int jy = 0; jy < 3; jy++) {
                                int xy = (y+jy-1) * width + (x+jx-1);
                                czerwonySuma
                                   += jadro[jy][jx]*red(obrazek.pixels[xy]);
                                zielonySuma
                                   += jadro[jy][jx]*green(obrazek.pixels[xy]);
                                niebieskiSuma
                                   += jadro[jy][jx]*blue(obrazek.pixels[xy]);
                                        }
                                }
                            pixels[y* width + x]
                                = color(czerwonySuma,
                                zielonySuma, niebieskiSuma);
                    }
         }
        updatePixels();
}
```

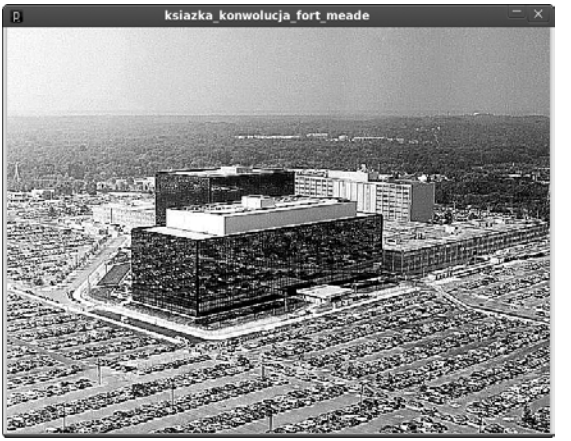
EFEKT ROZMYCIA NR.1

EFEKT ROZMYCIA NR.1

EFEKT WYKRYCIA KRAWĘDZI



EFEKT RELIEFU



EFEKT WYOSTRZENIA

7.7 AUTOMATY KOMÓRKOWE

Technika operacji na sąsiadujących pikselach przyda się nam nie tylko do manipulacji gotowym obrazem, ale także do tworzenia dynamicznych abstrakcji. Automaty komórkowe, (ang. *cellurar automaton*) są samoreplikującymi się strukturami. Mogą one symulować zachowanie się wielu zjawisk w przyrodzie, zarówno w środowisku nieożywionym jak i z udziałem organizmów żywych. Do wynalezienia automatu komórkowego doszło w wyniku wzajemnej inspiracji dwóch naukowców – Stanisława Ulama oraz Johna von Neumanna. Pierwszy z nich pracował nad problemem rozrostu kryształów, drugi natomiast nad problemem z dziedziny teorii obliczeń – samoreprodukcją robotów. W Wikipedii nieformalna definicja automatu komórkowego brzmi następująco.

„Automat komórkowy to system składający się z pojedynczych komórek, znajdujących się obok

siebie. Ich układ przypomina szachownicę lub planszę do gry. Każda z komórek może przyjąć jeden ze stanów, przy czym liczba stanów jest skończona, ale dowolnie duża. Stan komórki zmieniany jest synchronicznie zgodnie z regułami mówiącymi, w jaki sposób nowy stan komórki zależy od jej obecnego stanu i stanu jej sąsiadów."

Matematyczna definicja automatu komórkowego formalizuje go w następujący sposób.

Automat komórkowy to:

- sieć komórek {i} w przestrzeni D-wymiarowej
- zbiór $\{s_i\}$ stanów pojedynczej komórki, zwykle ten sam dla wszystkich komórek, zawierających k elementów
- reguła F określająca stan komórki w chwili t+1 w zależności od stanu w chwili t tej komórki i komórek ją otaczających; $s_i(t+1) = F(\{sj(t)\})$, j należy do O(i), gdzie O(i) jest otoczeniem i-tej komórki. [20]

W kontekście programowania grafiki, jako komórkę możemy rozumieć pojedynczy piksel. Naturalne też będzie pracowanie w dwóch wymiarach. Zasady natomiast zdefiniujemy jako wyrażenia boolowskie. Zamiast jednak utożsamić komórkę z pikselem, wybierzemy podejście obiektowe – zdefiniujemy klasę, której instancje posiadać będą właściwości komórki.

```
class Komorka {
        boolean stan = false;
        Komorka(){
        }
        Komorka(boolean $stan) {
                stan = $stan;
```

20 (K.Kułakowski, Automaty komórkowe, OEN AGH, Kraków 2000)

```
        }

        boolean jestZywa() {
                return stan;
        }
        void zmienStan() {
                stan=!stan;
        }
}
```

Klasa **Komorka** posiada pole przechowujące jeden z dwóch jej możliwych stanów – komórka może być martwa lub żywa. Domyślnie, jeśli stwarzamy instancję klasy używając pierwszego konstruktora, wartość tej zmiennej jest ustalona na **false**. Zdefiniowaliśmy również metodę, która zmienia stan komórki na przeciwny oraz predykat, informujący nas o bieżącym stanie komórki.

Napiszemy teraz dwuwymiarową wersję systemu dynamicznego opartego na automatach komórkowych. Reguły zmian stanów komórek zapisujemy według następującej konwencji. Szereg cyfr oznaczający ilość żywych sąsiadów, która pozwala przetrwać komórce do następnego pokolenia oddzielamy ukośnikiem '/' od szeregu cyfr określających ilość żywych sąsiadów powołujących nową komórkę do życia. Przykładowo reguły, które noszą nazwę „Gry w życie *Conwaya*[21]", możemy zapisać jako **23/3**. Oznacza to, że komórka przetrwa jeśli w jej bezpośrednim sąsiedztwie znajdą się dwie lub trzy inne żywe komórki, w przeciwnym wypadku „umrze". Jeśli natomiast w otoczeniu martwej komórki znajdą się

21 Obszernym źródłem wiedzy o tym zagadnieniu jest strona internetowa http://www.conwaysgameoflife.net/

trzy żywe, komórka ta „ożyje". Co czyni interesującymi automaty komórkowe dla naukowców z różnych dziedzin, to fakt, że kilka prostych reguł tworzy dynamiczne i złożone struktury, których zachowanie może stać się modelem problemów z obszaru fizyki, socjologii czy ekonomii.

Zespół reguł zachowania się komórki możemy przedstawić w Processingu jako koniunkcję dla żywej i alternatywę dla martwej komórki, np. reguła **245/368** będzie wyglądać następująco.

```
//REGUŁY DLA ŻYWEJ KOMÓRKI
if (wCentrum.jestZywa() ) {
        if (zywe != 2 && zywe !=4 && zywe != 5) {
                wCentrum.zmienStan();
        }
}
//REGUŁY DLA MARTWEJ KOMÓRKI
else {
        if (zywe == 3 || zywe == 6 || zywe == 8) {
                wCentrum.zmienStan();
        }
}
```

Sprawdzenie liczebności żywych sąsiadów wykorzystuje strukturę zagnieżdżonych pętli, którą stosowaliśmy przy konwolucji. Poniżej podany jest kod animacji z ewoluującym środowiskiem komórek.

```
Komorka[] komorki;
Komorka[] nastepnePokolenie;
Komorka wCentrum;
```

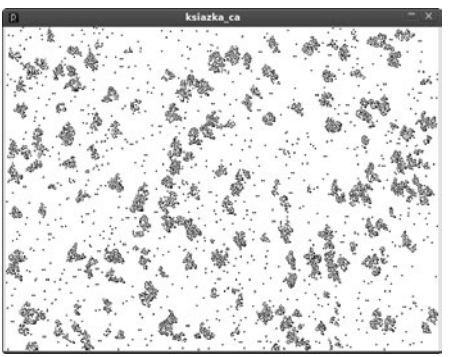
```
float gestosc = 0.01;
void setup() {
        size(640, 480);
        komorki = new Komorka[width*height];
        nastepnePokolenie = new
                Komorka[width*height];
        for (int i = 0; i < komorki.length; i++) {
                boolean zywa = (random(1)
                   < gestosc && random(1)>0.05) ? true : false;
                komorki[i] = new Komorka(zywa);
                nastepnePokolenie[i] = new Komorka();
        }
        background(255);
        smooth();
}

void draw() {
        loadPixels();
        for (int x = 1; x < width-1; x++) {
                for (int y = 1; y < height-1; y++) {
                    int zywe = 0;
                    for (int i = -1; i < 2; i++) {
                        for (int j = -1; j < 2; j++) {
                              if (i==0 && j==0) {
                                  continue;
                              }
                              int sasiad
                                  = (y+j) * width + (x+i);
                              if (komorki[sasiad].jestZywa()) {
                                  zywe++;
                              }
                        }
                     }
```

```
                    int indeks = y * width + x;
                    wCentrum = komorki[indeks];

                //REGUŁY DLA ŻYWEJ KOMÓRKI
                //(…)
                //REGUŁY DLA MARTWEJ KOMÓRKI
                //(…)
                nastepnePokolenie[indeks] = wCentrum;
                pixels[indeks] =
                   (nastepnePokolenie[indeks].
                   jestZywa()) ?color(0) : color(255);
                }
        }
        updatePixels();
        for (int k = 0; k < komorki.length; k++) {
                komorki[k] = nastepnePokolenie[k];
        }
}
```

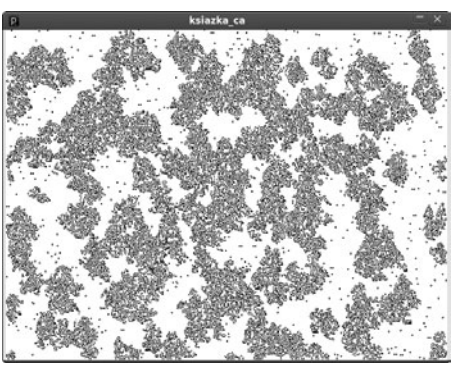
Zmienna **gestosc** determinuje ilość żywych komórek stworzonych podczas zapełniania tablicy. W kodzie wprowadziliśmy użyteczną składnię, która skraca wyrażenie warunkowe **if(...) {...} else {...}** do jednej linijki. Co więcej, ta skrócona wersja wyrażenia warunkowego może występować w połączeniu z operatorem przypisania po lewej stronie.

```
<ZMIENNA> = (<WARUNEK>) ?
<ALTERNATYWA_1> : <ALTERNATYWA_2>;
```

Używając tego wyrażenia mogliśmy przypisać **`true`** lub **`false`** do zmiennej boolowskiej **`zywa`**, w zależności od wyniku ewaluacji warunku w nawiasie. W **`draw`** natomiast użyliśmy tego zapisu w linijce, w której kolor piksela zmienia się na czarny jeśli komórka jest żywa lub na biały, jeśli jest inaczej.



„GRA W ŻYCIE" W DZIAŁANIU



MOŻEMY ZAOBSERWOWAĆ PROCESY ROZROSTU I SAMOORGANIZACJI KOMÓREK

7.8 ALGORYTMICZNOŚĆ Pojęcie algorytmu jest bardzo popularne i na dobre zagościło w języku potocznym. Używamy go np. w sytuacji, gdy chcemy podkreślić, że czyjeś działanie kierowane jest dobrze określonymi regułami. W internetowej encyklopedii PWN hasło objaśniające algorytm zaczyna się od następujących słów.

algorytm, przepis postępowania prowadzący do rozwiązania ustalonego problemu, określający ciąg czynności elementarnych, które należy w tym celu wykonać.

Słowo „algorytm" pochodzi od nazwiska perskiego matematyka, Muhammada ibn Musa al-Chuwarizmiego, który żył pomiędzy ok. **780** i ok. **850** n.e. Z kolei staroangielskie *„algorism"* oznaczało wykonywanie działań za pomocą liczb arabskich.

Wiele algorytmów nosi nazwę swojego wynalazcy (autentycznego lub domniemanego). Autorstwo algorytmu znajdującego największy wspólny dzielnik przypisuje się Euklidesowi i stąd jego nazwa. Algorytm, który znajduje najkrótszą ścieżkę w grafie, nosi nazwę algorytmu Dijkstry, od nazwiska jego twórcy – holenderskiego informatyka Edsgera Dijkstry. Najczęściej jednak algorytmy nazywa się podobnie jak funkcje. Ich nazwy po prostu informują nas o rodzaju wykonywanego przez algorytm działania.

Współczesne rozumienie słowa algorytm ma źródło w koncepcji *calculus ratiocinator* z ok. roku 1680, autorstwa Gottfrieda Wilhelma Leibniza. Koncepcja uniwersalnej maszyny liczącej została następnie sformalizowane przez matematyków w pierwszej połowie

XX w. Najbardziej znane formalizacje to „Formulation 1" Emila Posta, oraz przede wszystkim uniwersalna maszyna Alana Turinga[22].

Języki programowania są stworzone do implementacji algorytmów. Powstały bowiem z myślą o komunikowaniu jednoznacznych instrukcji maszynie liczącej. Z przyczyn historycznych (algorytmy są starsze od języków programowania) oraz różnic stylistycznych pomiędzy językami, do opisu algorytmów często używa się pseudokodu, który ma mniej formalną składnię i może zostać łatwo przetłumaczony na dowolny język programowania. W gruncie rzeczy nie ma jednego, ogólnie obowiązującego zestawu reguł pisania pseudokodu – najistotniejsza jest jednoznaczność wyrażanych instrukcji.

Najbardziej znanym i najszerzej w literaturze dydaktycznej omówionym algorytmem jest Algorytm Euklidesa, który oblicza największy wspólny dzielnik (**NWD**) z dzielenia dwóch liczb. Możemy go zapisać za pomocą pseudokodu w następujący sposób.

```
Algorytm Euklidesa:
Dane wejściowe: liczba całkowita
a, liczba całkowita b
Dane wyjściowe: liczba całkowita NWD z liczb a i b
dopóki b != 0:
        c = reszta z dzielenia a przez b
        a = b
        b = c
NWD = a
```

22 Książka Rogera Penrose'a „Nowy Umysł Cesarza" (PWN, Warszawa 2000), zawiera bardzo przystępne omówienie idei Alana Turinga.

Zaimplementowanie algorytmu jako funkcji w Processingu nie powinno przysporzyć nam większych problemów.

```
int NWD(int a, int b){
        if(b = 0){
                int nwd = a;
                return nwd;
        }
        else {
                int c = a%b;
                return NWD(b, c);
        }
}
```

W powyższej implementacji algorytm używa rekurencji. Jeśli zaimplementujemy go jako iterację, działający kod może wyglądać jak niżej.

```
int NWD(int a, int b) {
        while (b!=0) {
                int c = a%b;
                a = b;
                b = c;
        }
        return a;
}
```

Przy okazji poznaliśmy drugi sposób na skonstruowanie pętli – przy pomocy słowa **`while`**. Konstrukcję można przetłumaczyć następująco „Wykonuj polecenia zawarte w bloku pętli dopóty, dopóki warunek w nawiasie po **`while`** pozostaje prawdziwy." W przypadku pętli **`for`** sprawdzaliśmy, czy licznik mieścił się

w określonym zakresie. Natomiast w warunku skojarzonym z **while**, możemy umieścić dowolne wyrażenie boolowskie, tak jak to robiliśmy z **if** przy kierowaniu przebiegiem wykonania programu. Sprawdzimy teraz, czy obie implementacje algorytmu Euklidesa dają identyczny wynik.

```
void setup() {
        int a = 17*1213451;
        int b = 17*4334;
        if ( NWD(a, b) == NWD_rekur(a, b)) {
                print( NWD(a, b) );
        }
}
```

17

Przy strukturach z **while** musimy upewnić się, podobnie jak to było przy konstruowaniu funkcji rekurencyjnych, że w bloku kodu jest zawarty mechanizm wyjścia z pętli. W przeciwnym wypadku doprowadzimy do sytuacji tzw. „nieskończonej pętli" (ang. *infinite loop*), w której musielibyśmy wymusić zakończenie programu, a być może nawet zrestartować komputer. Przykład złego kodu znajduje się poniżej. Wyłączona spod egzekucji linijka zawiera niezbędny mechanizm zapobiegający błędowi.

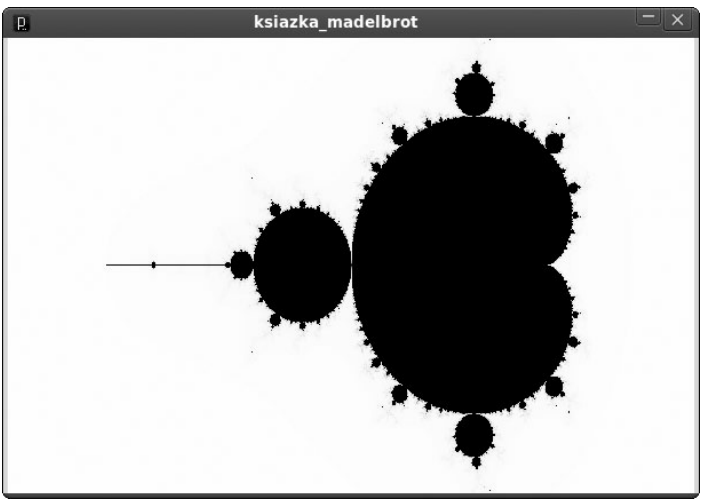
```
int i = 0;
while(i < 10){
        println(i);
        //i++;
}
```

Kolejny przykład pseudokodu, to algorytm rysujący zbiór Madelbrota – zdaje się najbardziej znanego fraktalu.

```
Algorytm rysujący zbiór Madelbrota jako grafikę
rastrową z użyciem liczb rzeczywistych:
Dla każdego piksela na ekranie zrób:
        x0 = przeskalowana współrzędna x piksela
(musi się znajdować w zakresie (-2,5, 1))
        y0 = przeskalowana współrzędna y piksela
(musi się znajdować w zakresie (-1, 1))
        x = 0
        y = 0

        iteracja = 0
        maks_iteracja = 1000

        jeśli ( x*x + y*y < (2*2)
I iteracja < maks_iteracja ):
                x_tymczasowe = x*x – y*y + x0
                y = 2*x*y + y0

                x = x_tymczasowe

                iteracja = iteracja + 1

        jeśli ( x*x + y*y < (2*2) ):
                kolor = czarny
        w przeciwnym razie:
                kolor = (iteracja+1)/
maks_iteracja * 255

                narysuj_piksel(x0,y0,kolor)
```

Jak w przypadku **NWD**, powyższy pseudokod możemy przetłumaczyć na kod działający w Processingu.

```
size(320,240);
for (int x = 0; x < width; x++) {
        for (int y =0; y<height; y++) {
                float x0 = map(x, 0, width, -2.5, 1);
                float y0 = map(y, 0, height, -1, 1);
                float _x = 0;
                float _y = 0;
                int iteracja = 0;
                int maks_iteracja = 1000;
                while (_x*_x + _y*_y <
                        4 && iteracja < maks_iteracja) {
                        float _x tmp = _x*_x - _y*_y + x0;
                        _y = 2*_x*_y + y0;
                        _x = _x_tmp;
                        iteracja++;
                }
                color k = (_x*_x + _y*_y
                        < 4) ? color(0) : color(map(iteracja,
                        0, maks_iteracja, 255,0)) ;
                        set(x,y,k);
                }
}
```



WYNIK DZIAŁANIA NASZEJ IMPLEMENTACJI
ALGORYTMU RYSUJĄCEGO ZBIÓR MADELBROTA

8 GENERATYWNOŚĆ

8.1 ANALOGIA DO ŚWIATA OŻYWIONEGO

Struktury danych, których funkcją jest przechowywanie wielu instancji identycznego typu, nazywamy ogólnie kolekcjami. Używane przez nas tablice, bez względu na typ, charakteryzuje stała pojemność. Zmiana ilości elementów przechowywanych w tablicy jest niemożliwa. W wielu okolicznościach więc konieczne okazuje się użycie struktury danych, do której można dodać obiekt, lub go z niej usunąć. Taką dynamiczną tablicą jest **`ArrayList`**. Jest to klasa, której instancję tworzymy tak jak każdą inną – poprzez wywołanie konstruktora komendą **`new`**.

```
ArrayList <NAZWA> = new ArrayList();
```

W odróżnieniu od zwykłej tablicy, nie określamy długości kolekcji w trakcie tworzenia obiektu **`ArrayList`**. Również nie określamy typu przechowywanych danych. Nie ma takiej potrzeby, gdyż **`ArrayList`** traktuje dane domyślnie jakby należały do jednej, najbardziej ogólnej klasy **`Object`**. W Processingu każde dane, które nie są wartością zmiennej prymitywnej (tj. **`int`**, **`float`**, **`boolean`** itd.), są przede wszystkim instancją klasy **`Object`**. Nie możemy umieścić w **`ArrayList`** danych prymitywnych w postaci, do której przywykliśmy. Musielibyśmy użyć wersji obiektowych typów prymitywnych, tj. klas „opakowujących" wartość prymitywną w obiekt. Nie będzie to nam jednak obecnie potrzebne, bowiem **`ArrayList`** użyjemy jedynie do przechowywania obiektów.

Warto przyjrzeć się klasie w programowaniu obiektowym poprzez analogię do przyrodniczej systematyki organizmów. Poniższa tabela przedstawia miejsce *Homo sapiens* w świecie zwierząt.

Regnum	***Animalia***
Phylum	***Chordata***
Classis	***Mammalia***
Ordo	***Primates***
Familia	***Hominidae***
Genus	***Homo***
Species	***H. sapiens***

Jak widać, nasz gatunek należy do gromady (łac. *Classis*, ang. *Class*) ssaków (łac. *Mammalia)*, co oznacza, że posiadamy wszystkie cechy, które wyróżniają ssaki spośród innych gromad, tzn. ptaków, gadów, itd. Nie zachodzi w powyższym twierdzeniu symetria, tzn. fałszem jest, że wszystkie ssaki posiadają cechy należne człowiekowi. Jest to bowiem charakterystyka hierarchiczna – człowiek jest ssakiem w ogólności a *Homo sapiens* w szczególności. Dziedziczymy wraz z innymi gatunkami ssaków zespół cech wspólnych, będących spadkiem po naszych najdawniejszych ewolucyjnych przodkach, natomiast jako osobny gatunek wykształciliśmy dodatkowo zbiór cech charakterystycznych już tylko dla człowieka. Rzecz ma się identycznie z klasami w Processingu – „wspólnym przodkiem" jest dla nich klasa **`Object`**.

Do utworzonej **ArrayList** dodajemy obiekty używając metody **add**,z dodawanym obiektem jako argumentem. Podobnie jak w zwykłych tablicach, indeksowanie rozpoczyna się od liczby **0**. Indeks będzie się zwiększać wraz z każdym dodanym obiektem. Rozmiar tablicy możemy sprawdzić funkcją **size**. Dostęp do przechowywanych obiektów jest możliwy za pomocą funkcji **get**, z indeksem jako argumentem. Funkcja ta zawsze zwraca instancję ogólnej klasy **Object**. To sprawia, że przed użyciem pobranego obiektu najczęściej konieczne jest dokonanie konwersji na obiekt klasy, której reprezentantem był on przed umieszczeniem go w tablicy. Konwersji dokonujemy za pomocą operatora rzutowania, poznanego przy okazji konwersji zmiennych liczbowych. Polega to na umieszczeniu w nawiasie przed nazwą konwertowanej zmiennej nazwy docelowego typu. Musimy o rzutowaniu pamiętać, gdyż **ArrayList** nie troszczy się o typ obiektu w niej umieszczanego i zrównuje go do instancji **Object**.

Sposobem na uniknięcie konieczności rzutowania jest podanie nazwy klasy wewnątrz operatora diamentowego '**<>**' w momencie utworzenia instancji tablicy. Przyjrzymy się temu za chwilę na przykładzie działającego skryptu.

8.2 DZIAŁANIA NA WEKTORACH

Ponieważ będziemy od tej pory intensywnie korzystać z działań na wektorach, konieczne jest poznanie czym wektory są i zasady ich arytmetyki. Przyjrzymy się klasie **PVector**, która w Processingu reprezentuje ten matematyczny obiekt. Oto podstawowe informacje.

Wektor jest odcinkiem łączącym dwa punkty. Posiada kierunek (nachylenie względem początku układu współrzędnych), zwrot (który decyduje o tym, który punkt uznamy za początek, a który za koniec) oraz długość. W Processingu wektory są osadzone w środku układu współrzędnych, czyli tworząc obiekt klasy **PVector** przekazujemy do konstruktora jedynie współrzędne punktu końcowego. Ponadto obowiązuje nas Kartezjański układ współrzędnych (o osiach x,y,z), tak więc punkt może mieć **2** lub **3** współrzędne, w zależności, czy rozpatrujemy punkt na płaszczyźnie czy w przestrzeni. Faktycznie punkt na płaszczyźnie posiada również w domyśle przypisany trzeci parametr równy **0**. Wektor o długości **1** nazywamy wersorem lub wektorem jednostkowym.

W wyniku dodania do siebie dwóch lub więcej wektorów otrzymamy nowy wektor, którego współrzędne punktu końcowego równać się będą sumom koresponduących współrzędnych. Inaczej mówiąc, współrzędna **x** nowego wektora będzie równa sumie współrzędnych **x** dodawanych do siebie wektorów. Tak samo postępujemy z pozostałymi współrzędnymi. Na przykład dane są dwa wektory

```
a=(1,3,5) i b=(2,6,8)
```

to wektor ich sumy równy będzie wektorowi

```
c=(1+2, 3+6, 5+8)=(3, 9, 13).
```

Klasa **PVector** posiada dwie funkcje o nazwach **add** i **sub**, które odpowiednio dodają i odejmują od siebie dwa wektory. Również każda instancja posiada analogiczne metody, które akceptują jako argument drugi obiekt **PVector**. Należy wówczas pamiętać,

że działania wpływają na obiekt z poziomu którego wywołujemy metodę.

Iloczynem skalarnym nazywamy procedurę przemnożenia przez jedną wartość wszystkich składowych współrzędnych wektora. Na przykład, mając wektor

```
a=(2,3,4)
```

i skalar

```
s = 5,
```

po zastosowaniu iloczynu skalarnego otrzymamy wektor

```
b=(5*2, 5*3, 5*4)=(10, 15, 20).
```

W przypadku dzielenia przez skalar, iloczyn zastępujemy ilorazem. Podobnie jak przy dodawaniu i odejmowaniu, funkcje klasy **`PVector`** , (**`mult`** dla mnożenia i **`div`** dla dzielenia), wykonują te działania za nas.

Obliczenie odległości pomiędzy wektorami jest nieco trudniejsze. W gruncie rzeczy obliczamy odległość pomiędzy punktami końcowymi wektorów. Przykładowo mając dwa wektory

```
a = (1,2,3) oraz b= (4,5,6),
```

korzystając tylko z funkcji prostych obliczymy odległość o w następujący sposób.

```
o = sqrt( sq(4-1) + sq(5-2) + sq(6-3) ) = sqrt(9 +
9 + 9) = sqrt(27) = 5.196152 (wynik w przybliżeniu)
```

W wyniku działania nie otrzymujemy wektora, lecz wartość liczbową. I znów – klasa **`PVector`** posiada funkcję o nazwie **`dist`**, która wyręcza nas z tej arytmetyki.

Długość wektora jest odległością w linii prostej pomiędzy początkiem wektora a jego końcem. Każdy obiekt **`PVector`** posiada funkcję **`mag`**, (od ang. *magnitude*), która podaje nam tą wartość.

Ponadto dwie inne operacje związane z długością wektora mogą być dla nas istotne – normalizacja wektora i ograniczenie długości wektora do określonej długości. Oba działania są różnymi wersjami tej samej operacji. Normalizacja i ograniczenie długości są więc iloczynem skalarnym wektora. W pierwszej skalujemy wektor do długości równej **1**, w drugiej skalujemy wektor tak, aby jego długość nie przekroczyła żądanego rozmiaru. W przypadku normalizacji skalar równy jest odwrotności długości wektora. Skalar przy ograniczaniu długości wektora równa się natomiast stosunkowi pożądanej długości do aktualnej długości. Oba działania są reprezentowane przez funkcje **`normalize`** (pl. *unormuj*) oraz **`limit`**.

8.3 ZACHOWANIE GRUPOWE

W kolejnych przykładach zbudujemy symulację zachowania się ławicy ryb jako ilustrację działania algorytmu dla grupy obiektów organizującej się samodzielnie. W naturze często fascynują nas widoki wielkich stad ptaków, rojów owadów czy ławic ryb, które zdolne są do zsynchronizowanych zachowań w grupie. Proces ten wydaje się tak złożony, że zaskakujący może wydać się pomysł sprowadzenia go do kilku, konkretnie trzech reguł. Niemniej w 1986 roku Craig Reynolds opracował program o nazwie „Boids", który sugestywnie symulował zachowanie gromad indywidualnych osobników. Kompleksowość zachowań opiera się na trzech prostych regułach przestrzeganych przez każdego osobnika grupy, zwanego też w literaturze przedmiotu *agentem*[23].

23 Szczególnie wartym polecenia jest pozycja autorstwa Gary'ego Williama Flake'a, „The Computational Beauty of Nature : Computer Explorations of Fractals, Chaos, Complex Systems, and Adaptation", MIT Press, 1998, w której bardzo przystępnie jest omówiony nie tylko algorytm tłumu (rozdział 16. pozycji) , ale również większość z poruszanych w niniejszym podręczniku zagadnień.

Rozdzielność – unikaj innych osobników
Dopasowanie – dopasuj kierunek ruchu do średniej kierunków najbliższego otoczenia
Zwartość – skieruj się do centrum lokalnej grupy

Oczywiście wszelkie symulacje komputerowe należy traktować jako przybliżenie naturalnego fenomenu. Jak się jednak zaraz przekonamy, realizm symulacji jest uderzający, nawet przy naszej szkicowej implementacji. Poniżej znajduje się definicja klasy o nazwie **Ryba**, która będzie pełnić rolę superklasy w kolejnych przykładach. Współrzędne, prędkość oraz cel poszczególnych agentów będą reprezentowane przez wektory. Bardzo ułatwi to operacje obliczeniowe. W klasie **Ryba** zdefiniujemy podstawowe cechy i zachowania naszej wirtualnej ryby. Nie uwzględnimy w niej natomiast zachowań grupowych. Zrobimy to w definicji klasy rozszerzającej.

```
class Ryba {
        PVector xy;
        PVector v;
        PVector act_v;
        PVector cel;
        float pred = 7;
        PImage cialo;
        float dystansDoCelu = 0;
        Ryba(PVector _xy) {
                xy = _xy.get();
                v = new PVector(random(1), random(1));
                kolor = color(random(255), random(255),
                random(255), 200);
        }
```

```
void rysuj() {
        pushMatrix();
        noStroke();
        fill(kolor);
        translate(xy.x, xy.y);
        rotate(atan2(v.y, v.x));
        if (cialo!=null) {
                image(cialo, -cialo.width/4,
                -cialo.height/4, cialo.width/2,
                cialo.height/2);
        }
        else {
                ellipse(0, 0, 10, 10);
                strokeWeight(1);
                stroke(100);
                line(0, 0, 15, 0);
                line(15, 0, 10, 3);
                line(15, 0, 10, -3);
        }
        popMatrix();
}

void poruszaj() {
        v = znajdzCel();
        if (dystansDoCelu > 10) {
                act_v = v.get();
                act_v.mult(pred);
                xy.add(act_v);
        }
}
PVector znajdzCel() {
        PVector _cel = new PVector(0, 0, 0);
        if (cel!=null) {
```

```
                    _cel = plynDo(cel);
            }
            return _cel;
        }

        PVector plynDo(PVector cel) {
            PVector _cel = cel.get();
            dystansDoCelu = _cel.dist(xy);
            _cel.sub(xy);
            _cel.normalize();
            return _cel;
        }
        void ustawCel(PVector cel) {
            this.cel = cel.get();
        }
}
```

Jedyny jak do tej pory konstruktor klasy wymaga argumentu klasy **`Pvector`**, określającego początkowe położenie agenta. Metoda **`rysuj`** jest podobna do tej znanej z **`Pilka`**. Również procedura uaktualnienia pozycji obiektu w metodzie **`poruszaj`** jest niemal identyczna, z tym zastrzeżeniem, że teraz współrzędne oraz prędkość są określane przez wektory. Pierwszym wyrażeniem w **`poruszaj`** jest przypisanie do zmiennej **`v`** wartości zwróconej przez funkcję **`znajdzCel`**. Jeśli rozpatrzymy funkcję pod względem znaczeń, to funkcja **`znajdzCel`** mówi nam, że obiekt klasy **`Ryba`** skieruje się do celu tylko wówczas, gdy zmienna **`cel`** jest określona. Funkcja ta na wyjściu zawraca wektor prędkości skierowany w stronę punktu reprezentującego cel. Skierowanie wektora w odpowiednią stronę wymaga tylko elementarnych działań na wektorach – odjęcia od wektora celu wektora aktualnego położenia obiektu oraz znormali-

zowania otrzymanej różnicy. Warunek **if** w predykacie **poruszaj** sprawdza czy cel został osiągnięty. Informacja ta jest konieczna, aby obiekt zatrzymał się w odpowiednim momencie.

Możemy sprawdzić klasę pisząc prosty program w głównej zakładce edytora.

```
Ryba ryba;
void setup() {
        size(800, 400);
        ryba = new Ryba(new PVector(width/2,height/2));
}

void draw() {
        background(#0B0321, 200);
        ryba.poruszaj();
        ryba.rysuj();
}

void mouseMoved() {
        ryba.ustawCel(newPVector(mouseX,mouseY));
}
```

Otrzymaliśmy interakcję w której obiekt podąża za ruchem myszki. Oczywiście umowność jest tu bardzo wyraźna – strzałka z przyczepionym kółkiem nie przypomina żadnego z żyjących gatunków ryb. Pamiętajmy jednak, że klasa posłużyć ma nam jedynie za bazę. W następnym kroku rozszerzymy ją o niezbędne właściwości i stany.

8.4 DZIEDZICZENIE

Dziedziczenie jest to hierarchiczna relacja pomiędzy dwiema klasami, gdzie klasa nadrzędna (nazywana superklasą) jest rozszerzana przez klasę podrzędna (podklasę). Pierwsza z nich jest z zasady mniej szczegółowa niż druga. Na przykład – wszystkie ssaki poruszają się, używając kończyn, niemniej tylko *Homo sapiens* ma chód wyprostowany. Ponadto zachowanie człowieka jest rozszerzone o wiele zachowań wykształconych stosunkowo niedawno, które są nieobecne nawet w szczątkowym stopniu u innych gatunków tej samej gromady.

Pokażemy teraz jak w praktyczny sposób wykorzystać ideę dziedziczenia cech i stanów klasy. Definicję podklasy rozpoczynamy jak definicję każdej innej klasy, z tą różnicą, że po nazwie klasy umieszczamy słowo kluczowe **extends** wraz z nazwą klasy rozszerzanej.

```
class <NAZWA> extends
<NAZWA_SUPERKLASY> {
<WYRAŻENIA>
}
```

Klasa rozszerzająca nazywana jest podklasą (ang. *subclass*). Przed pisaniem definicji podklasy należy poznać klasę, którą się chce rozszerzyć – jej konstruktory oraz członków (zmienne, metody, funkcje). Podklasa jest bowiem w istocie superklasą z dodatkowymi właściwościami i stanami. Możemy mówić też o zakresie – wszystko to, co zostało zdefiniowane w superklasie jest obecne w podklasie. Rozszerzenie klasy **Ryba** przez klasę **Swietlik** oznacza więc, że klasa **Swietlik** jest rozbudowaną wersją klasy **Ryba**. Posiada zarówno wszystkie zmienne zdefiniowane lub utworzone w superklasie jak i wszystkie jej metody i funkcje. Mówiąc inaczej, właści-

wości i stany superklasy są zawsze odziedziczane przez klasę ją rozszerzającą. To właśnie nazywamy dziedziczeniem w programowaniu obiektowym.

```
class Swietlik extends Ryba{
        Swietlik(PVector _xy, PImage cialo){
                super(_xy);
                this.cialo = cialo;
        }
}
```

Mechanizm dziedziczenia obejmuje również konstruktory, stąd też instancję klasy **Swietlik** możemy „powołać do życia" konstruktorem pochodzącym z **Ryba**. Jednak w listingu powyżej definicję **Swietlik** rozpoczęliśmy od napisania konstruktora posiadającego różną sygnaturę od konstruktora superklasy. Mamy więc obecnie alternatywę co do sposobu utworzenia obiektu **Swietlik** – wywołując konstruktor z superklasy lub też nowy, określony w podklasie. Pamiętajmy jednak, że w konstruktorze superklasy zawarliśmy operacje, polegające między innymi na przypisaniu wartości do ważnych zmiennych instancji klasy. Jeżeli w ciele naszego świeżo definiowanego konstruktora chcielibyśmy zachować operacje konstruktora superklasy, to możemy to uzyskać na dwa sposoby. Pierwszy polega na przepisaniu zawartości starego konstruktora. Drugi, znacznie wygodniejszy, na wywołaniu konstruktora superklasy z wnętrza konstruktora podklasy komendą **super**. Jest to funkcja reprezentująca konstruktory superklasy i wymaga przekazania parametrów zgodnych z sygnaturą konstruktora, który chcemy wywołać. Możemy więc **super** wywołać na jeden z wielu sposobów zdefiniowanych w superklasie. W wypadku klasy **Ryba** nie mamy

żadnego wyboru, ponieważ w superklasie zdefiniowany został tylko jeden konstruktor. Dalej możemy przekazać konstruktorowi superklasy obiekt **_xy. Gdybyśmy na tym poprzestali, konstruktory oby**dwu klas dublowałyby swoją funkcjonalność. Niemniej konstruktor zdefiniowany w **Swietlik** posiada dodaną linijkę, w której program tworzy instancję typu **Pimage**. Nadmienię tu, że jeśli zmienna wskaźnikowa nie ma przypisanego obiektu, to wówczas referencja takiej zmiennej (czyli jej wskazanie) określana jest jako **null** (ang. *pusta, nieważna, niebyła).* Jak pamiętamy, w superklasie zmienna **cialo** była jedynie zadeklarowana a więc tak jakby jej nie było **(null)**. Przypomnijmy, że zmienną zadeklarowaną w superklasie traktujemy tak, jakby należała już do podklasy.

Metoda **rysuj** w **Ryba** posiada warunek, który przekierowuje program do jednego z alternatywnych bloków kodu. W przypadku instancji klasy **Ryba**, nieważność zmiennej **cialo** sprawia, że rysowana jest schematyczna strzałka z przyczepionym kółkiem. Jest to uproszczona reprezentacja graficzna agenta. Konstruktora zdefiniowanego w podklasie użyjemy do stworzenia instancji w sytuacji, gdy zechcemy kierować przebiegiem programu tak, by na współrzędnych obiektu zamiast schematu został wyświetlony obrazek. Oczywiście nawet w przypadku użycia konstruktora z superklasy, w dalszym ciągu będziemy mogli do zmiennej **cialo** przypisać dowolny obiekt typu **PImage**, np. poprzez użycie notacji kropki. Niemniej wydaje się praktyczne posiadanie konstruktora, który w chwili utworzenia instancji przypisuje obrazek do zmiennej **cialo**, jeśli jako drugi argument przekażemy grafikę. W przykładzie użyjemy obrazka przedstawiającego świetlika o wymiarach **50x10** pikseli.

Zachęcam jednak do eksperymentowania z różnymi obrazkami, niekoniecznie ryb. Nazwa klasy nie powinna zawęzić naszej kreatywności.

W gruncie rzeczy może się wydawać nieuzasadnione wprowadzenie w superklasie zmiennej, która nie pełni żadnej konstruktywnej roli (przez cały czas pozostanie **`null`**). Tak jest w przypadku zmiennej **`cialo`**. Na domiar tego nie stworzyliśmy w superklasie żadnej metody przypisującej obiekt do wspomnianej zmiennej. Pominęliśmy też stworzenie funkcji pobierającej (tzw. *getter*) i metody ustalającej (tzw. *setter*). W tym przypadku możemy jednak naszą „niedbałość" usprawiedliwić w następujący sposób. Już w momencie projektowania klasy ogólnej zakładaliśmy, że zostanie ona rozszerzona w przyszłości przez bardziej szczegółowe podklasy, w których zmienna **`cialo`** spełni istotną rolę. Oczywiście możemy ten wypadek uznać za wyjątkowy, bowiem najczęściej superklasy definiowane są mniej precyzyjnie.

OBRAZEK, BĘDĄCY REPREZENTACJĄ WIZUALNĄ INSTANCJI KLASY `SWIETLIK`

Metody **`rysuj`** i **`poruszaj`** są zdefiniowane w superklasie, więc możemy się skupić na funkcjach związanych bezpośrednio z zachowaniem się instancji klasy **`Swietlik`** w relacjach do innych obiektów tego samego typu. Zaczniemy od zadeklarowania zmiennej reprezentującej ławicę rybek, do której będzie należeć instancja. Dodajmy nad konstruktorem **`Swietlik`** następująca linijkę.

```
ArrayList<Swietlik> lawica;
```

Umieszczenie po słowie **ArrayList** nazwy klasy, wpisanej w operator diamentowy <> sprawi, że nie będziemy zmuszeni używać operatora rzutowania po każdym pobraniu obiektu z tablicy, żeby przypomnieć programowi do jakiej klasy należą instancje w niej przechowywane. Dla przypomnienia, po umieszczeniu obiektu w **ArrayList**, program traktuje go jakby był instancją najogólniejszej klasy **Object**. Użycie operatora diamentowego sprawi, że program zapamięta typ znajdujący się w tablicy. Dodatkowo zawęzi to możliwość dodania nowych obiektów do instancji klasy, której nazwę wpisaliśmy w operatorze diamentowym. Tym samym będziemy mieli pewność, że nie znajdzie się w niej żaden „intruz". Ogólna składnia tego sposobu tworzenia obiektu **ArrayList** przedstawiona jest poniżej. Zwróćmy uwagę na część kodu po operatorze przypisania – powtórzenie operatora diamentowego i następujący po nim nawias.

```
ArrayList< <KLASA> > <NAZWA> =
new ArrayList< <KLASA> >();
```

Symbol **<KLASA>** zamieniamy na nazwę klasy, której instancje będą przechowywane w strukturze.

W parze ze zmienną **lawica** występuje metoda **ustawLawice**.

```
void ustawLawice(ArrayList lawica) {
        this.lawica = lawica;
        cel = null;
}
```

Co do właściwych funkcji, odpowiedzialnych za „zachowanie stadne" obiektu, to każda z nich oblicza jeden z wektorów wchodzących następnie w sumę decydującą o finalnym wektorze prędkości obiektu.

```
PVector unikaj(ArrayList lawica) {
        PVector unik = new PVector(0, 0);
        float kolidujeZ = 1;
        for (Swietlik sw: lawica) {
                float dystans = xy.dist(sw.xy);
                if (this.equals(sw)
                        == false && dystans < 15) {
                        PVector delta
                              = PVector.sub(xy, sw.xy);
                        delta.normalize();
                        float waga =
                             1.0 – norm(dystans, 0, 15);
                        delta.mult(waga);
                        unik.add(delta);
                        kolidujeZ++;
                }
        }
        unik.normalize();
        unik.sub(v);
        return unik;
}
PVector nasladuj(ArrayList lawica) {
        PVector zestrojenie = new PVector(0, 0);
        float nasladuje = 1;
        for (Swietlik sw: lawica) {
                float dystans = xy.dist(sw.xy);
                if (this.equals(sw)
```

```
                        == false && dystans < 100) {
                        PVector zwv = sw.v.get();
                        float waga =
                          1.0 – norm(dystans, 0, 100);
                        zwv.mult(waga);
                        zestrojenie.add(zwv);
                        nasladuje++;
                }
        }
        zestrojenie.div(nasladuje);
        zestrojenie.normalize();
        zestrojenie.sub(v);
        return zestrojenie;
}

PVector znajdzCel(ArrayList lawica) {
        PVector cel = new PVector(0, 0);
        for (Swietlik sw: lawica) {
                if(this.equals(sw)) continue;
                cel.add(sw.xy);
        }
        cel.div(lawica.size());
        poczDystans = dystansDoCelu = cel.dist(xy);
        cel.sub(xy);
        cel.normalize();
        return cel;
}
```

Te trzy funkcje implementują odpowiednio rozdzielność, dopasowanie i zwartość, zgodnie z algorytmem zachowania stadnego. Warunkiem niezbędnym do zaistnienia relacji pomiędzy niezależnymi obiektami, jest przypisanie instancji do wirtualnej ławicy. Każda instancja będzie korygować swoje własne zachowanie w rela-

cji do obiektów znajdujących się w **`ArrayList`**. Pokrótce omówimy operacje składające się na każdą z funkcji.

Funkcja **`uwazaj`** zapobiega kolizji obiektu z obiektami znajdującymi się w jego najbliższym otoczeniu. Najbliższe otoczenie określamy jako obszar o promieniu **`50`** pikseli. Jest to arbitralna wielkość, którą możemy zastąpić zmienną, co wpłynie na ruchy agenta. Program iteruje przez wszystkie obiekty w **`lawica`** i sprawdza, czy któreś z nich nie są niebezpiecznie blisko siebie. Ponieważ przedmiotem iteracji są wszystkie obiekty umieszczone w **`lawica`**, to podczas niej obiekt iterujący, będący elementem tablic, napotyka siebie samego. Stąd też konieczność umieszczenia w kodzie warunku **`if`**, wykluczającego sytuację porównywania się w takim przypadku.

Działania na wektorach są tu elementarne i powtarzają schemat z funkcji **`plynDo`**, z kilkoma drobnymi, ale istotnymi różnicami. Po pierwsze wektor nie powinien wskazywać w stronę napotkanego obiektu, lecz do niej przeciwną. Żeby tak się stało, zmieniamy kolejność odejmowania – zamiast odjąć współrzędne obiektu od współrzędnych celu, robimy na odwrót. Alternatywnie możemy pozostawić kolejność niezmienioną, lecz przemnożyć wektor różnicy przez skalar równy **`-1`**. Po drugie, ostateczny wektor odciągający, powinien być średnią ważoną wektorów skojarzonych z relacją obiektu wobec każdego z najbliższych sąsiadów z osobna. Wagę ustalamy w wyniku znormalizowania odległości między obiektem i sąsiadem wedle prawidłowości, że obiekt najbliższy oddziałuje najsilniej. Na końcu normalizujemy uzyskany wektor i odejmujemy od niego aktualną prędkość instancji. Wynik zostaje podany na wyjściu funkcji.

Druga funkcja o nazwie **nasladuj** ma podobną strukturę – również w niej iterujemy `lawica`, tym razem jednak interesuje nas średnia ważona prędkości pobliskich ryb odjęta od aktualnej prędkość instancji, z której poziomu wywołaliśmy funkcję.

Trzecia funkcja **znajdzCel** jest wersją innej funkcji o tej samej nazwie, napisanej przez nas w superklasie. Ponieważ jednak teraz celem jest środek ławicy, zmieniliśmy sygnaturę tak, że w miejscu pojedynczego **PVector** znajduje się **ArrayList**. W ciele funkcji sumujemy współrzędne wszystkich obiektów, znajdujących się w **lawica**. Po podzieleniu sumy przez rozmiar **ArrayList**, otrzymujemy współrzędne celu. Ustalamy zwrot wektora obiektu tak, jak to robiliśmy wcześniej – odejmując jego aktualne położenie od położenia celu.

Napisaliśmy więc trzy funkcje implementujące warunki interakcji pojedynczego agenta ze stadem. Ostatnią rzeczą jest zastosowanie wyników funkcji do zmiany aktualnego wektora prędkości obiektu. Będzie to suma tych pojedynczych czynników – konieczności uniknięcia kolizji, adaptacji do stada i podążania do jego środka. Każde z tych zachowań może mieć inny priorytet, np. unikanie kolizji będzie ważniejsze od podążania za grupą, itp. Innymi słowy – każdy ze składników sumy możemy inaczej wyważyć. Ponadto musimy uwzględnić aktualną prędkość obiektu, bowiem jeżeli zmiana kierunku następować będzie zbyt gwałtownie, czyli tak jakby ryba zapominała o swoim zachowaniu sprzed chwili, ruch będzie rwany i niespokojny. Każde uaktualnienie powinno więc następować w relacji do prędkości aktualizowanej. Mamy więc cztery współczynniki, inaczej wagi, przez które przemnożymy składniki sumy. Założymy też, że poszczególne jednostki ławicy nie będą

się wyróżniać priorytetami. Konfiguracja wag decyduje o całościowym charakterze zachowania grupy – jak mocno grupa będzie pilnować swojej integralności, ile autonomii zachowa każdy z agentów, itd. Zanim napiszemy definicję funkcji, która łączy składniki zachowania w jeden wektor, zdefiniujemy tablicę przechowującą wagi w następującej kolejności – „pamięć" o poprzedniej prędkości, istotności uników i dostosowania się do kierunków ruchów sąsiadów oraz siłę przyciągania do środka ławicy. Mogą one przyjąć dowolne wartości z zakresu **(0,1)**. Warto z nimi trochę poeksperymentować i sprawdzić, jak poszczególne parametry oddziałują na los symulacji.

```
float [] d = {
        0.8, 1, 0.7, 0.3
};
PVector dolaczDo(ArrayList<Swietlik> lawica) {
        PVector v_temp = v.get();
        PVector unik = unikaj(lawica);
        PVector zestrojenie = nasladuj(lawica);
        PVector koherencja = znajdzCel(lawica);
        v_temp.mult(d[0]);
        unik.mult(d[1]);
        zestrojenie.mult(d[2]);
        koherencja.mult(d[3]);
        unik.add(zestrojenie);
        unik.add(koherencja);
        unik.mult(1.0-d[0]);
        v_temp.add(unik);
        v_temp.normalize();
        return v_temp.get();
}
```

Ostatnią rzeczą w definicji naszej podklasy będzie przeciążenie kilku funkcji z superklasy. Z przeciążeniem mamy do czynienia jeśli w definicji podklasy definiujemy metodę lub funkcję o tej samej nazwie i sygnaturze jak ta, istniejąca w superklasie. Dzięki temu możemy dokonywać modyfikacji części dziedziczonego po superklasie kodu z poziomu podklasy i bez ingerencji w definicję superklasy. W przypadku klasy **Swietlik** musimy nanieść małą poprawkę do dwóch dziedziczonych funkcji – **znajdzCel** oraz **ustawCel.** Pierwsza z nich jest wywoływana w metodzie **poruszaj** superklasy, pozostawionej przez nas w niezmienionej formie. Niemniej **znajdzCel** powinna zawierać następującą alternatywę. Jeśli instancja klasy **Swietlik** należy do jakiejś ławicy, wektor prędkości instancji powinien zostać uaktualniony zgodnie z obliczeniami w **dolaczDo.** W przeciwnym wypadku instancja powinna zachowywać się identycznie jak instancja superklasy – nie podążać za grupą obiektów, lecz kierować się w stronę indywidualnego celu.

```
PVector znajdzCel() {
        PVector cel = new PVector(0, 0, 0);
        if (lawica!=null) {
                cel = dolaczDo(lawica);
        }
        else if (this.cel!=null) {
                cel = plynDo(this.cel);
        }
        return cel;
}
```

Drugie przeciążenie dotyczy metody **ustawCel**, która w odróżnieniu od tej z superklasy przekształci ponadto zmienną **lawica** na **null**, wykluczając

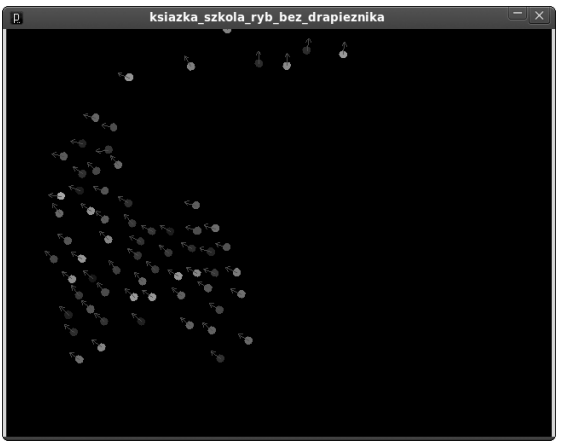
sytuację, w której instancja będzie posiadała równocześnie dwie sprzeczne przesłanki – do zachowywania się indywidualistycznie (tzn. z utworzoną zmienną **cel**) oraz taką, która skłoni ją do zajęcia miejsca w grupie obiektów tej samej klasy (tzn. ze zmienną **lawica**).

```
void ustawCel(PVector cel) {
        poczDystans = xy.dist(cel);
        this.cel = cel.get();
        lawica = null;
}
```

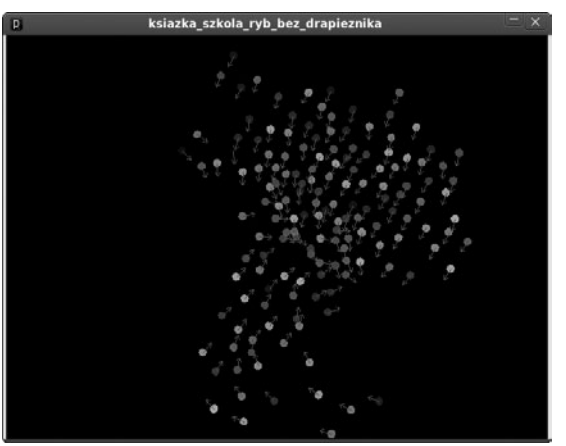
Tak zakończyliśmy pisać definicję klasy **Swietlik**. Możemy teraz w głównej zakładce umieścić metody **setup**, **draw** i obsługę zdarzeń myszki. Powinniśmy więc mieć program składający się z trzech zakładek. W pierwszej kod animacji, w drugiej definicję superklasy **Ryba**, oraz w trzeciej definicję podklasy o nazwie **Swietlik**.

```
ArrayList<Swietlik> swietliki;
void setup() {
        size(1024, 700);
        swietliki = new ArrayList<Swietlik>();
        for (int i = 0; i <10; i++) {
        /* Jeśli chcemy użyć obrazka */
        PImage c = loadImage("swietlik.png");
        Swietlik sw = new Swietlik(newPVector
                (random(width), random(height)) , c);
        /* W przeciwnym razie po prostu:
        Swietlik sw = new Swietlik(new PVector
                (random(width), random(height))); */
        swietliki.add(sw);
        sw.ustawLawice(swietliki);
        sw.rysuj();
        }
}
```

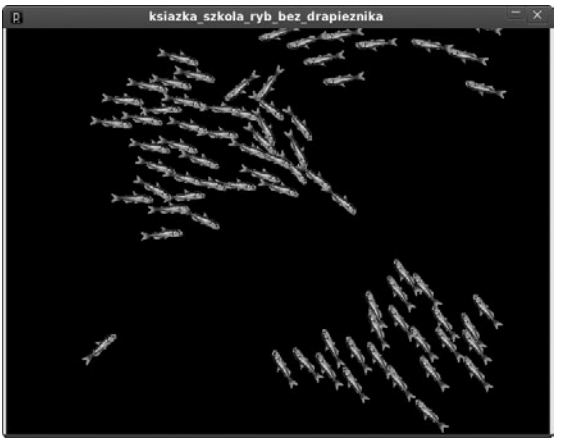
```
void draw() {
        background(0);
        println(swietliki.size());
        for (int i = 0; i < swietliki.size(); i++) {
                Swietlik sw = swietliki.get(i);
                sw.poruszaj();
                if (sw.xy.x > width) {
                        sw.xy.set(sw.xy.x-width, sw.xy.y, 0);
                }
                else if (sw.xy.x < 0) {
                sw.xy.set(sw.xy.x+width, sw.xy.y, 0);
                }
                if (sw.xy.y > height) {
                sw.xy.set(sw.xy.x, sw.xy.y-height, 0);
                }
                else if (sw.xy.y < 0) {
                sw.xy.set(sw.xy.x, sw.xy.y+height, 0);
                }
                /*
                if (sw.xy.x > width || sw.xy.x < 0) {
                        sw.xy.set(width%sw.xy.x, sw.xy.y, 0);
                        }
                        if (sw.xy.y > height || sw.xy.y < 0) {
                        sw.xy.set(sw.xy.x, height%sw.xy.y, 0);
                        }
                 */
                sw.rysuj();
        }
}
void mouseReleased() {
        Swietlik sw = new Swietlik(new
                PVector(mouseX, mouseY));
        swietliki.add(sw);
        sw.ustawLawice(swietliki);
}
```



JEDEN ZE STANÓW PROGRAMU SYMULUJĄCEGO ŁAWICĘ RYB



MOŻNA ZAOBSERWOWAĆ, ŻE POMIMO WZGLĘDNEJ AUTONOMII, POJEDYNCZY AGENCI TWORZĄ SPÓJNĄ ZBIOROWOŚĆ – ŁAWICĘ RYB



TEN SAM PROGRAM W WERSJI Z OBRAZKIEM
PRZEDSTAWIAJĄCYM MAŁĄ RYBKĘ

9 KOMUNIKACJA

9.1 BIBLIOTEKI

Na słownik Processingu składa się kilkadziesiąt starannie dobranych komend, które pozwalają na łatwą pracę z podstawowymi mediami, takimi jak obraz czy tekst. Dzięki zastosowaniu bibliotek, możemy rozszerzyć liczbę poleceń o takie, które powstały z myślą o wideo, dźwięku, interfejsie użytkownika lub też do rozwiązywania innych, niezwiązanych bezpośrednio z zawartością multimedialną zadań programistycznych. Biblioteki umożliwią nam również łatwiejsze programowanie wielu procesów, np. sieciowych typu klient-serwer, korzystanie z baz danych lub z obrazu pochodzącego z kamery internetowej.

Biblioteki rozszerzają więc leksykon języka. W istocie nie są niczym innym jak zbiorem dodatkowych klas przystosowanych do rozwiązywania specyficznych problemów. W następnych paragrafach poznamy dwie biblioteki dołączone do dystrybucji Processingu oraz jedną, którą pobierzemy z internetu i zainstalujemy w naszym systemie.

9.2 PRACA Z WIDEO

Zaczniemy od zastosowania biblioteki służącej do pracy z obrazem wideo. Wraz z Processingiem jest dostarczona biblioteka, dzięki której jest możliwa współpraca naszych programów z programem QuickTime. Ponieważ ta biblioteka jest częścią dystrybucji programu, nie wymaga ona instalacji. Możemy włączyć ją do programu w dwójnasób. Pierwszy sposób polega na wybra-

niu nazwy biblioteki z pozycji w menu: *Sketch->Import Library...-> Video*. W efekcie zostanie dodana następująca linijka tekstu do źródła naszego programu.

```
import processing.video.*;
```

Drugi sposób polega na ręcznym wpisaniu powyższego kodu. Nie ma najmniejszego znaczenia który sposób wybierzemy. Ogólnie, dodanie klas zawartych w bibliotekach odbywa się poprzez podanie pełnej nazwy biblioteki po słowie kluczowym **import**.

```
import <NAZWA_BIBLIOTEKI>.*;
```

<NAZWA_BIBLIOTEKI> nie oddaje do końca istoty sprawy, bowiem to, co napisaliśmy po słowie **import** jest adresem miejsca, w którym znajdują się skompilowane klasy biblioteki. Ciąg processing.video oznacza, że pliki klas znajdują się w katalogu *video*, który z kolei znajduje się w katalogu *processing*. Kropki oddzielają foldery, natomiast gwiazdka (*) oznacza, że chcemy mieć dostęp do wszystkich definicji klas zgromadzonych w folderze *video*. Klasy są pogrupowane w tzw. pakiety (ang. *package*). Możemy używać biblioteki tylko wtedy, gdy kompilator wie, w którym miejscu na dysku może odnaleźć pliki pożądanych klas. Każde przeszukiwanie rozpoczyna się od określonych miejsc na dysku. Biblioteki dystrybuowane z Processingiem znajdują się w folderze o nazwie *libraries*, w tym samym, w którym znajduje się plik *processing.exe* (w przypadku Windowsa, pod Mac OS X wewnątrz archiwum *Processing.app*, a pod Linuxem w miejscu rozszerzenia *exe* będziemy mieli *sh*). Jeżeli kompilator napotka w źródle polecenie importu biblioteki, to będzie to dla niego sy-

gnał do przejrzenia tego właśnie katalogu w poszukiwaniu odpowiednich klas.

Od tego momentu będziemy mogli używać klas umożliwiających nie tylko pracę z gotowymi plikami filmowymi, ale również zapisywać zawartość płótna uruchomionego programu w pliku filmowym oraz korzystać z obrazu dostarczanego przez kamerę internetową. Tą trzecią możliwość będziemy teraz stopniowo eksplorować.

Jak wyświetlić na ekranie obraz z kamery? Poniższy kod przedstawia jak najprościej to zrobić.

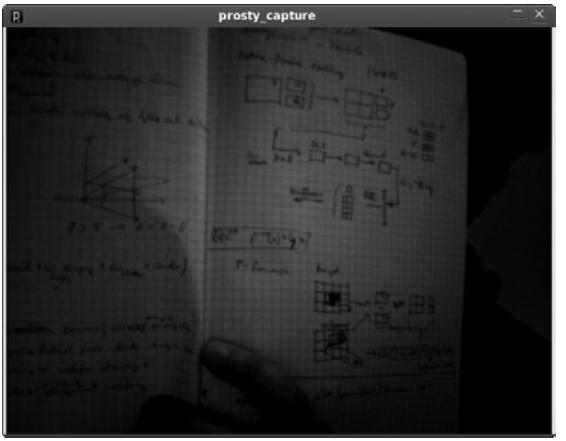
```
import processing.video.*;

Capture kamera;

void setup(){
        size(320,240);
        kamera = new Capture(this,width,height,15);
        frameRate(15);
}
void draw(){
        if(kamera.available()){
                kamera.read();
                image(kamera,0,0);
        }
}
```

Musimy utworzyć instancję klasy **`Capture`**, która reprezentuje urządzenie podłączone (lub w przypadku laptopa wbudowane) do komputera. Konstruktor wymaga kilku argumentów. Przede wszystkim musimy zacząć od słowa **`this`**, następnie po przecinku wpisu-

jemy wysokość, szerokość obrazu z kamery oraz ilość klatek na sekundę, z jaką urządzenie rejestruje obraz. Kamery różnią się budową, niemniej podane powyżej wartości powinny współpracować z większością wyprodukowanych urządzeń. W ostatniej linijce **setup** ustaliłem prędkość odświeżania na **15**. Choć nie jest to konieczne, chciałem zsynchronizować ją z częstotliwością z jaką kamera pobiera obraz. W **draw** program sprawdza, czy kamera jest dostępna. Jeśli tak, zostaje odczytany aktualny obraz. Następnie **kamera** (będąca instancją **Capture**) zostaje przekazana jako argument dobrze znanej metodzie **image**.

Obiekt **Capture** jest podobny do pod wieloma względami do **PImage**. Wspólna właściwość, która będzie dla nas szczególnie istotna, to fakt posiadania przez obiekty **Capture** i **PImage** zmiennej **pixels**. Alternatywny sposób na wyświetlenie obrazu z kamery opiera się na procedurze przekopiowania zawartości **pixels** obiektu **Capture** do tablicy **pixels** głównego okna programu, tak jak to robiliśmy wcześniej z **pixels** obiektu **PImage**. Używamy w tym celu metody **arrayCopy**, która kopiuje zawartość jednej tablicy do innej o tym samym typie i długości.

```
void draw(){
        if(kamera.available()){
                kamera.read();
                kamera.loadPixels();
                loadPixels();
                arrayCopy(kamera.pixels, pixels);
                updatePixels();
        }
}
```



9.3 ŚLEDZENIE RUCHU Wrócimy teraz do zagadnień procesowania bitmapy. Obraz z kamery możemy poddać takim samym manipulacjom jak każdą bitmapę **`PImage`**, tj. zmienić wartość piksela poprzez działanie arytmetyczne lub konwolucje. Odbywa się to identycznie jak przy statycznym obrazie, z tej prostej przyczyny, że animacja jest następstwem nieruchomych klatek. Obraz z kamery możemy też zamienić w urządzenie sterujące – w interfejs, dzięki któremu będziemy się komunikować z napisanym przez nas programem. Konieczne jest znalezienie sposobu mapowania obrazu rejestrowanego przez kamerę w dane o charakterze tych, które uzyskujemy z tradycyjnego urządzenia wskazującego (np. myszki). Skupimy się na śledzeniu zmian zachodzących w rejestrowanym obrazie i ich zamianie na współrzędne, które wykorzystamy do sterowania obiektem graficznym. Współrzęd-

nymi uzyskanymi w wyniku opisanego w następnym akapicie algorytmu posłużymy się identycznie jak pobranymi z tradycyjnego urządzenia wskazującego.

Zastosowany przez nas algorytm śledzenia ruchu określa, czy dwie kolejne klatki filmu różnią się od siebie poprzez obliczenie różnicy w wartościach pikseli przechowywanych na tych samym indeksach w tablicy **`pixels`** bieżącej i poprzedniej klatki. Jeśli różnica wynosi **`0`**, a dzieje się tak tylko wówczas, gdy odejmujemy od siebie dwie identyczne wartości, to żadna zmiana w obrazie nie zaszła. W przeciwnym razie indeksy identyfikują współrzędne miejsc w obrazie, gdzie mógł nastąpić ruch. Algorytm jest więc bardzo prosty – podczas iteracji tablicy **`pixels`** należącej do bieżącej klatki, program sumuje składowe koloru (czerwony, zielony, niebieski) i odejmuje je od analogicznej sumy z poprzedniej klatki. Indeks następnie zamieniony zostaje na współrzędne x i y. Zamiana indeksu na współrzędną jest możliwa dzięki wiadomym wartościom szerokość i wysokość obrazu. W programie, którego kod jest podany poniżej, istotne ze względu na ruch piksele obrazu wyróżnione zostaną jednolitym (czerwonym) kolorem. W celu uniknięcia zbyt wielkiej czułości, określiliśmy wartość progową, poniżej której program będzie ignorował zmiany w obrazie spowodowane ruchem. Współrzędne z indeksów, których wartość przekroczyła próg, przekazywane są jako argument konstruktorowi **`Pvector`**. Następnie dodawane są do siebie wszystkie utworzone w procedurze detekcji wektory. Suma składowych piksela na aktualnym indeksie kopiowana jest do tablicy, która w następnej klatce posłuży za punkt odniesienia. Jest to rodzaj pamięci o poprzedniej klatce. Po zakończeniu iteracji wektor będący sumą wszystkich wektorów zostanie

podzielony przez ilość składników tej sumy. Dodaliśmy jeszcze jeden wektor o nazwie **pamięć**, przechowujący współrzędne „celownika" w poprzedniej klatce.

```
import processing.video.*;

Capture kamera;
float [] poprzednia;
PVector sredni = new PVector(0, 0);
PVector pamiec = new PVector(0, 0);
float prog = 255;
int licznik = 0;

void setup() {
        size(320, 240);
        kamera = new Capture(this, width, height, 15);
        poprzednia = new float [width*height];
        Arrays.fill(poprzednia, 0);
}

void draw() {
        if (kamera.available()) {
                kamera.read();
                kamera.loadPixels();
                loadPixels();
                arrayCopy(kamera.pixels, pixels);
                licznik = 0;
                for (int x = 0; x < width; x++) {
                        for (int y = 0; y < height; y++) {
                                int i = y*width + x;
                                color teraz = kamera.pixels[i];
                                float t = red(teraz);
                                t+= green(teraz);
                                t+= blue(teraz);
```

```
                        float w = poprzednia[i];
                float roznica = abs(t - w);
                if (roznica > prog) {
                        licznik++;
                        sredni.add(new PVector(x, y));
                        pixels[i] = color(255, 0, 0);
                }
                        poprzednia[i] = t;
                }
        }
        updatePixels();
        println(frameRate);
        if (licznik > 0) {
                sredni.div(licznik+1);
                pamiec = sredni.get();
        }
    }

    stroke(255);
    strokeWeight(4);
    line(pamiec.x, pamiec.y+25, pamiec.x, pamiec.y-25);
    line(pamiec.x+25, pamiec.y, pamiec.x-25, pamiec.y);
}
```


ksiazka_detekcja_wideo


ksiazka_detekcja_wideo

Powyższy algorytm ma oczywiste wady. Na podstawie takiej analizy ruchu nie jesteśmy w stanie faktycznie zorientować się, co dzieje się w świecie zewnętrznym. Ta analiza nie dostarcza nam podstawowej wiedzy o charakterystyce obiektów, znajdujących się w zasięgu ka-

mery. Znaczy to, że nie wiemy na przykład, czy zmiany w obrazie zostały wywołane przez pojedynczy obiekt czy przez grupę. Nie wiemy również w jakiej odległości od kamery obiekt się znajduje, jakiego jest koloru, jaką posiada masę, itd. To wszystko wymaga skomplikowanych obliczeń i jest przedmiotem prężnie rozwijanej gałęzi inżynierii zwanej widzeniem maszynowym.

9.4 AMORTYZACJA RUCHU

Zauważyć możemy, że „celownik" zmienia stany skokowo – przechodzi z jednego miejsca w drugie bez stanów pośrednich, w mało elegancki sposób. Pamiętamy, że instancja klasy **Ryba** zbliżała się do celu ruchem jednostajnym, poprzez regularne dodawanie stałej wielkości do współrzędnych swojej bieżącej pozycji. Wyobraźmy sobie jednak, że obiekt zwalnia zbliżając się do celu . Osiągniemy to przesuwając poruszający się obiekt o wektor będący wynikiem przeskalowania jego aktualnej pozycji przez współczynnik z zakresu **{0,1}**. Im krótszy dystans do celu, tym składnik sumy po prawej stronie operatora przypisania w poniższej definicji będzie mniejszy. Ta technika amortyzacji ruchu nazywa się po angielsku *easing*.

```
<WSPÓŁRZĘDNA> = <WSPÓŁRZĘDNA> +
(<ODLEGŁOŚĆ> * <WSPÓŁCZYNNIK>);
```

Możemy powyższy zapis skrócić do:

```
<WSPÓŁRZĘDNA> += (<ODLEGŁOŚĆ>
* <WSPÓŁCZYNNIK>);
```

Możemy dodać do kodu amortyzację ruchu jako następującą funkcję.

```
PVector easing(PVector xy, PVector cel){
        PVector tmp = PVector.sub(cel, xy);
        tmp.mult(amortyzacja);
        tmp.add(xy);
        return tmp;
}
```

Celownik z przykładu ze śledzeniem ruchu możemy rozpatrzyć jako osobny obiekt. Przeniesiemy wówczas metody rysujące oraz powyższą funkcję do definicji klasy.

```
class Celownik {
        PVector cel;
        PVector xy;
        float amortyzacja;
        Celownik(PVector xy, float amortyzacja) {
                this.xy = xy.get();
                this.amortyzacja = amortyzacja;
        }
        void ustawCel(PVector cel) {
                this.cel = cel.get();
        }
        void ustawAmortyzacje(float amortyzacja) {
                this.amortyzacja = amortyzacja;
        }
        void idz() {
                xy = easing();
        }
        PVector easing(){
                PVector tmp = PVector.sub(cel, xy);
                tmp.mult(amortyzacja);
                tmp.add(xy);
```

```
                return tmp;
        }
        void rysuj() {
                stroke(#00FF00);
                strokeWeight(4);
                line(xy.x, xy.y+25, xy.x, xy.y-25);
                line(xy.x+25, xy.y, xy.x-25, xy.y);
         }
}
```

Funkcja **easing** w definicji klasy ma pustą sygnaturę, gdyż zarówno pozycja jak i cel są jednoznacznie określone przez zmienne instancji. Możemy teraz nieznacznie zmodyfikować **draw** w przykładzie ze śledzeniem ruchu.

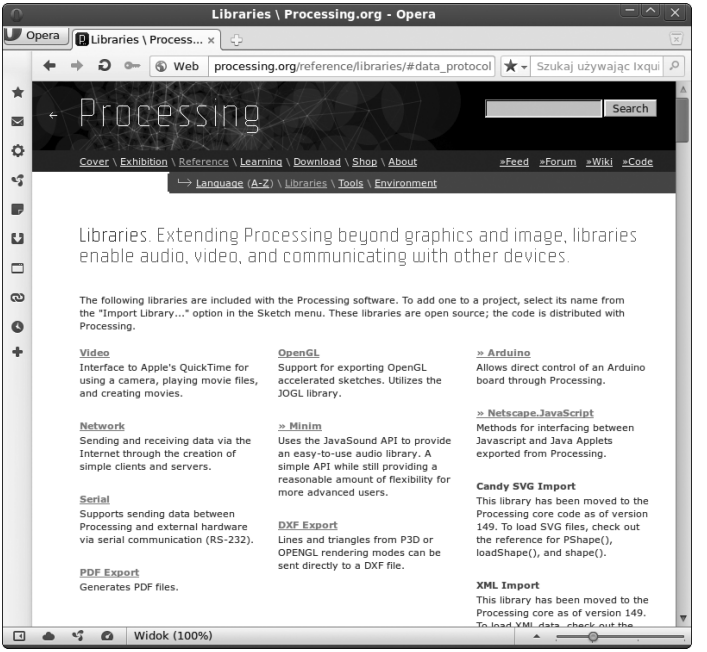
```
// zadeklaruj powyżej setup
Celownik celownik;
/* w setup poeksperymentuj z drugim parametrem
konstruktora – w zakresie (0,1) */
celownik = new Celownik(pamiec, 0.4);
celownik.ustawCel(pamiec);
/* w draw
        stroke(255);
        strokeWeight(4);
        line(pamiec.x, pamiec.y+25, pamiec.x, pamiec.y-25);
        line(pamiec.x+25, pamiec.y, pamiec.x-25, pamiec.y);
zamien na */

if (frameCount%5==0) {
        celownik.ustawCel(pamiec);
}
celownik.idz();
celownik.rysuj();
```

9.5 INSTALOWANIE BIBLIOTEK

Dla przypomnienia – w ustawieniach Processingu określamy folder, w którym zapisujemy nasze projekty. Każdy projekt jest zapisywany w folderze o tej samej nazwie. Folder *sketchbook* jest ważny jeszcze z jednej przyczyny, o której teraz będzie mowa. W ostatnim przykładzie zaimportowaliśmy jedną z bibliotek dostarczonych z główną dystrybucją Processingu. Oprócz biblioteki do pracy z plikami wideo, mamy do dyspozycji również takie biblioteki, które pozwalają na zapisanie klatek animacji do pliku *pdf* o dużej rozdzielczości (bardzo przydatne, jeżeli docelowym medium jest druk), czy do komunikacji z urządzeniami peryferyjnymi podłączonymi do portu seryjnego.

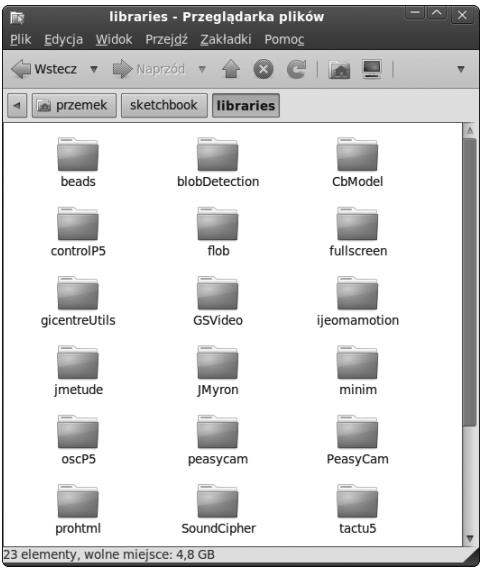
Jak wspomniałem we wstępie, popularność Processingu wynika z tego, że użytkownicy programu udostępniają innym efekty swojej pracy. W efekcie oprócz oficjalnych bibliotek, możemy rozszerzać Processing o biblioteki napisane przez innych programistów. Liczba bibliotek ciągle rośnie. Na stronie internetowej Processingu, w zakładce *Reference->Libraries,* możemy znaleźć biblioteki dotyczące niemal wszystkich zagadnień współczesnej informatyki.

Biblioteki użytkowników znajdują się w dziale *Contributions*, co można tłumaczyć jako darowizny. Pogrupowane są w **15** działów, od grafiki **3D** rozpoczynając, przez graficzny interfejs użytkownika, na typografi kończąc. Skupimy się teraz na bibliotece, znajdującej się w grupie *Data* / *Protocols*. Na jej przykładzie pokażę jak przebiega instalacja dowolnej biblioteki. Natomiast w następnym przykładzie posłużymy się nią do skomunikowania ze sobą dwóch, napisanych w tym rozdziale programów.



STRONA INTERNETOWA Z ROZSZERZENIAMI PROCESSINGU

Znajdźmy na stronie odnośnik do biblioteki o nazwie *oscP5*, autorstwa Andreasa Schlegera. Kliknijmy odnośnik, żeby przejść na stronę domową biblioteki. Ściągnijmy stamtąd plik o nazwie *oscP5-0.9.6.zip*. Instalacja wygląda następująco. W folderze *sketchbook* (*Processing* pod Windows) utwórz folder o nazwie *libraries*. Następnie rozpakuj ściągnięte archiwum i przekopiuj folder *oscP5* do folderu *libraries*. To właściwie wszystko. Procedura instalacji będzie taka sama w przypadku niemal każdej innej biblioteki. Poniżej pokazuję przykładowy folder *libraries*.



ZAWARTOŚĆ FOLDERU *LIBRARIES*, DO KTÓREGO INSTALUJEMY BIBLIOTEKI

Wewnątrz folderu każdej biblioteki powinniśmy zobaczyć podobną strukturę. Właściwy kod zawsze znajduje się w folderze *library*. Często również dołączone są przykłady, dokumentacja oraz pliki źródłowe.

BIBLIOTEKA CZĘSTO JEST UDOSTĘPNIANA WRAZ Z PRZYKŁADAMI, SZCZEGÓŁOWĄ DOKUMENTACJĄ ORAZ ŹRÓDŁEM.

Od tego momentu do każdego projektu będziemy mogli zaimportować zainstalowaną bibliotekę. Po ponownym uruchomieniu Processingu powinniśmy zobaczyć nową pozycję w menu *Sketch->Import library....* Po kliknięciu w nazwę biblioteki, do kodu automatycznie zostanie dodana komenda jej importu.

Trzy podstawowe pojęcia z teorii komunikacji to – nadawca, odbiorca oraz komunikat. Komunikat jest poprawnie przekazany odbiorcy tylko wówczas, gdy treść komunikatu po odebraniu przez adresata jest identyczna z treścią u nadawcy przed jej nadaniem. Protokoły komunikacyjne to w ogólności zespół reguł, według których dwa urządzenia mogą się ze sobą komunikować. Na klasyczny protokół składa się procedura powitalna (ang. *handshake*), w czasie której urządzenia nawiązują komunikację, przekazanie informacji oraz ostatecznie analiza poprawności przekazu, która skutkuje jednym z trzech – zakończeniem komunikacji, powrotem do procedury powitalnej lub żądaniem powtórnego przekazania informacji. Przykładem klasycznego protokołu jest faks.

9.6 PROTOKÓŁ OSC (OPENSOUND CONTROL)

Protokół OpenSound Control (OSC) został opracowany w The Center for New Music and Audio Technologies na Uniwersytecie Kalifornijskim w Berkeley w Stanach Zjednoczonych. Jego twórcami są Adrian Freed oraz Matt Wrigh, którzy po raz pierwszy zaprezentowali go w 1997 roku na Międzynarodowej Konferencji Muzy-

ki Komputerowej jako alternatywę dla standardu MIDI, służącego do przekazywania informacji pomiędzy instrumentami muzycznymi. Ze względu na fakt, że OSC jest protokołem sieciowym, może przekazywać informację nie tylko pomiędzy instrumentami, ale również między wszystkimi urządzeniami korzystającymi z infrastruktury telekomunikacyjnej (przykładowo sieci lokalnej lub internetu).

OSC wyróżnia kilka cech. Przede wszystkim sposób adresowania komunikatu bazuje na URL (ang. *Uniform Resource Locator*). W taki sposób adresowane są strony internetowe, ale tyczy się to wszelkich zasobów sieciowych. Składnia URL wygląda następująco.

```
<PROTOKÓŁ>://<ADRES_SERWERA>:<PORT>
/<HIERARCHICZNA_ŚCIEŻKA_DO_ZASOBU>?
<ZAPYTANIE>#<FRAGMENT_ZASOBU>
```

Każdy adres URL składa się z nazwy protokołu, domeny, numeru portu i hierarchicznej ścieżki do zasobu (np. pliku) na który wskazuje. Ścieżki do zasobów zdalnych są konstruowane podobnie jak ich wersje lokalne (tj. ścieżki do plików, folderów na dysku komputera) w systemach Uniksowych, z ukośnikiem '/' oddzielającym nazwy katalogów. Najczęściej spotykanym przez nas obecnie protokołem jest oczywiście http, służący do przesyłu danych pomiędzy serwerem stron internetowych, a przeglądarką tychże stron. Adres serwera to najczęściej nazwa strony, lecz możliwe jest też podanie bezpośredniego adresu IP serwera. Adres IP jest to 32 bitowa liczba przyporządkowana każdemu urządzeniu podłączonemu do internetu. Dla wygody zapisuje się ją jako ciąg czterech liczb, oddzielonych od siebie kropką. Wpisanie w pole adresu przeglądarki internetowej

ciągu http://213.144.235.204 jest równoznaczne z wpisaniem http://www.ixquick.com – oba adresy wskazują na tę samą stronę wyszukiwarki internetowej. **<PORT>** (często pomijany w adresie http) wskazuje na wirtualny slot zdalnego urządzenia z którym chcemy nawiązać komunikację.

Poniżej podaję dwa przykłady **URL.** Pierwszy jest adresem strony z wpisanym hasłem „url" w wyszukiwarce (używając przeglądarki Opera). Drugi to adres fragmentu artykuły na stronie w Wikipedii.

```
http://www.google.com/search?client=opera &rls=pl&q=
url&sourceid=opera&ie=utf-8&oe=utf-8&channel=suggest
http://pl.wikipedia.org/wiki/Uniform_Resource_
Locator#Elementy_adresu_i_przyk.C5.82ad
```

Powyższe informacje nie są niezbędne do posługiwania się zainstalowaną w poprzednim podrozdziale biblioteką. Przydają się jednak w zrozumieniu roli poszczególnych klas i metod biblioteki oraz parametrów do nich przekazywanych. Jeżeli chcemy wysyłać komunikat, potrzebujemy dwa obiekty – jeden określający adres i port adresata komunikatu, oraz drugi reprezentujących przekazywany komunikat. Będziemy też potrzebowali statycznej metody klasy **OscP5** o nazwie `flush`, która wyśle komunikat pod wskazany adres. Co do metod statycznych – są to takie metody, których możemy użyć po kropce zaraz po nazwie klasy, bez konieczności tworzenia instancji tej klasy – inaczej mówiąc są to metody należące do klasy, a nie do instancji klasy. Kolejny przykład zaczniemy od napisania kodu aplikacji nadającej komunikaty, inaczej nazywanej serwerem.

Na początku zadeklarujemy zmienną klasy **NetAddress**, która określać będzie odbiorcę komunikatów. Konstruktor tej klasy wymaga dwóch argumen-

tów – adresu sieciowego oraz numeru portu. W tym przykładzie adresujemy komunikaty do tego samego urządzenia, z którego nadajemy, więc jako adres IP wpiszemy **127.0.0.1.** Możemy zamiennie wpisać **localhost** i efekt będzie taki sam. Należy pamiętać, że numer IP wpisujemy jako **String.** Jako następny argument podajemy numer portu. Raczej może to być dowolny port, należy jednak wcześniej sprawdzić, czy nie służy on już jakiemuś innemu serwisowi, np. port nr 80 jest najczęściej zarezerwowany dla serwera http. Komunikat tworzymy jako obiekt klasy **OscMessage**. Do konstruktora przekazujemy etykietę komunikatu jako **String**, rozpoczynający się od ukośnika '/.' Pozwoli ona odbiorcy poszeregować przysłane komunikaty. Obiekt **OscMessage** zachowuje się podobnie do **ArrayList**, tzn. jest listą, która rośnie w miarę dodawania do niej danych metodą **add**. Dane muszą należeć do typów prymitywnych (**float**, **int**, **char** lub **String**). Umieścić je możemy w **OscMessage**, zarówno pod postacią pojedynczych literałów czy zmiennych jak i tablic. Jeżeli chcemy skorzystać z jednej instancji **OscMessage** do wysłania różnych paczek danych, to musimy pamiętać o wywołaniu metody **clearArguments** po każdym nadaniu komunikatu. Wysłanie komunikatu odbywa się poprzez statyczną metodę **flush** klasy **OscP5** z dwoma argumentami – komunikatem i adresem.

```
import oscP5.*;
import netP5.*;

NetAddress zdalnaLokacja;
OscMessage komunikat;

void setup() {
```

```
        size(100,100);
        zdalnaLokacja = new
                NetAddress("127.0.0.1", 12000);
        komunikat = new OscMessage("/wspolrzedne");
        frameRate(30);
}

void draw() {
}

void mousePressed() {
        komunikat.clearArguments();
        komunikat.add((int)mouseX);
        komunikat.add((int)mouseY);
        OscP5.flush(komunikat, zdalnaLokacja);
}
```

Aplikacja nasłuchująca komunikatów, czyli klient, jest jeszcze prostsza. Musimy w niej stworzyć obiekt klasy **`OscP5`** z argumentem wskazującym, na którym porcie powinien spodziewać się nadchodzących danych. **`OscP5`** posiada własną obsługę zdarzeń, reagującą na każdy nowy komunikat. Jest to po prostu metoda, analogiczna do tych obsługujących zdarzenia myszki lub klawiatury, o nazwie **`oscEvent`**, przechwytująca komunikaty **`OscMessage`**, które pojawiają się na porcie o numerze określonym przy tworzeniu obiektu **`OscP5`**. Zanim skorzystamy z przybywających danych, w definicji **`oscEvent`** musimy przeanalizować dostarczony komunikat. Interesują nas przy tym dwie rzeczy – czy nazwa etykiety zgadza się z oczekiwaną oraz czy przesłane dane są prawidłowego typu. Jeśli oba testy się powiodą, program wykona ustalony przez nas rozbiór

komunikatu na pojedyncze dane (tzw. sparsuje komunikat) oraz przekaże je w odpowiednie miejsca programu.

```
import oscP5.*;
import netP5.*;

OscP5 oscP5;

void setup() {
        oscP5 = new OscP5(this, 12000);
}

void draw() {
}

void oscEvent(OscMessage komunikat) {
        if (komunikat.checkAddrPattern("/wspolrzedne")==true) {
                if (komunikat.checkTypetag("ii")) {
                    float x = komunikat.get(0).intValue();
                    float y = komunikat.get(1).intValue();
                   println("odebrano: x=" + x + " , "+ y="+y);
                }
         }
}
```

Gdy uruchomimy jednocześnie oba programy i kilkakrotnie klikniemy w okno programu nadającego, to powinniśmy otrzymać w konsoli następujący wydruk.

```
OscP5 0.9.6 infos, comments, qu
estions at
### [2011/9/25 17:26:16] PROCESS @ OscP5 stopped.
### [2011/9/25 17:26:16] PROCESS @ UdpClient.
openSocket udp socket initialized.
### [2011/9/25 17:26:17] PROCESS @                ()
new Unicast DatagramSocket created @ port 12000
### [2011/9/25 17:26:17] PROCESS @ UdpServer.
run() UdpServer is running @ 12000
### [2011/9/25 17:26:17] INFO @ OscP5 is
running. you (127.0.1.1) are listening @ port 12000
odebrano: x=71.0 , y=69.0
odebrano: x=14.0 , y=14.0
odebrano: x=94.0 , y=15.0
odebrano: x=94.0 , y=95.0
```

9.7 PRZESYŁANIE DANYCH PRZEZ SIEĆ

Połączymy teraz dwie, wcześniej napisane aplikacje. Współrzędne obiektów śledzonych przez kamerę internetową zostaną przekazane do programu z ławicą wirtualnych rybek. W efekcie będziemy mogli wpływać na skupisko sztucznych żyjątek poprzez ruch przed obiektywem kamery. Wystarczy, że do przykładu ze śledzeniem ruchu dodamy kod serwera wysyłającego metodą **`flush`** znormalizowane współrzędne instancji klasy **`Celownik`**.

```
import oscP5.*;
import netP5.*;

NetAddress zdalnaLokacja;
```

```
OscMessage komunikat;
void setup(){
        //(...)
        zdalnaLokacja = new
                NetAddress("127.0.0.1", 12000);
        komunikat = new OscMessage("/wspolrzedne");
}

void draw(){
        //(...)
        float x_norm = norm(celownik.xy.x, 0, width);
        float y_norm = norm(celownik.xy.y, 0, height);
        komunikat.clearArguments();
        komunikat.add(x_norm);
        komunikat.add(y_norm);
        OscP5.flush(komunikat, zdalnaLokacja);
}
```

W kodzie programu z ławicą, oprócz dodania obsługi zdarzeń OSC, zmodyfikujemy odrobinę samą klasę **`Swietlik`**. Dodamy czwarte zachowanie – ucieczkę przed drapieżnikiem. Obliczenia w funkcji **`wypatrzDrapieznika`** są podobne do tych z funkcji **`unikaj`** – obie obliczają wektor odwodzący obiekt od przeszkody. Musimy też dodać kilka linijek w metodzie **`dolaczDo`**. Są one niezbędne do tego, by czwarty wektor wszedł w sumę ostatecznego wektora zmiany położenia obiektu. Uaktualnijmy definicję klasy **`Swietlik`** o następujący kod.

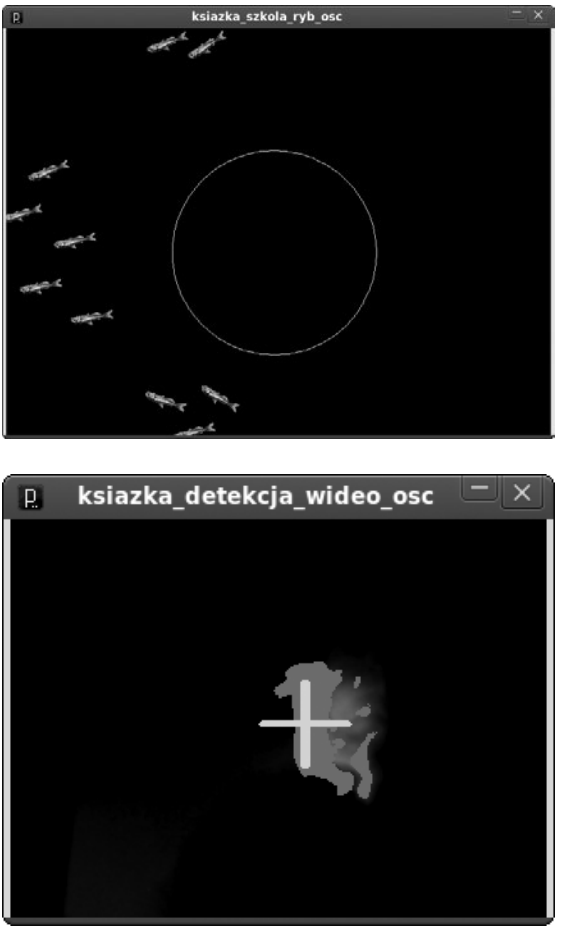
```
class Swietlik extends Ryba {
        //(...)
        PVector drapieznik = new PVector(0,0);
        // (...)
        PVector wypatrzDrapieznika(PVector
```

```
                drapieznik) {
                float dystans = xy.dist(drapieznik);
                float srodowisko = height/2;
                if (dystans < srodowisko) {
                        drapieznik.sub(xy);
                        drapieznik.normalize();
                        float waga = 1.0
                                - norm(dystans, 0, srodowisko);
                        drapieznik.mult(-waga);
                }
                else {
                        drapieznik.set(0, 0, 0);
                }
                this.drapieznik = drapieznik;
                return drapieznik;
        }

        PVector dolaczDo(ArrayList<Swietlik> lawica) {
                PVector v_temp = v.get();
                PVector unik = unikaj(lawica);
                PVector zestrojenie = nasladuj(lawica);
                PVector koherencja = znajdzCel(lawica);
                PVector drapieznik = wypatrzDrapieznika(this.
                        drapieznik.get());
                v_temp.mult(d[0]);
                unik.mult(d[1]);
                zestrojenie.mult(d[2]);
                koherencja.mult(d[3]);
                drapieznik.mult(d[4]);
                unik.add(zestrojenie);
                unik.add(koherencja);
                unik.add(drapieznik);
                //(...)
        }
}
```

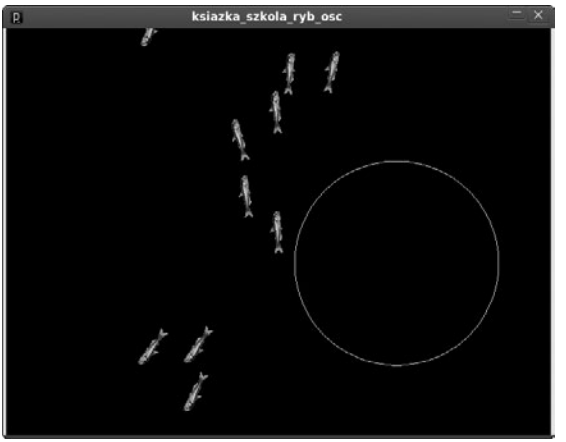
W głównej zakładce umieścimy obsługę zdarzeń **oscEvent**, w definicji której odebrane liczby staną się współrzędnymi drapieżnika. Tym sposobem będziemy mogli sterować ruchem ławicy z poziomu innego programu. Ponieważ OSC jest protokołem sieciowym, możemy obie aplikacje umieścić na osobnych urządzeniach skomunikowanych ze sobą za pomocą połączenia sieci lokalnej lub połączenia internetowego. Oba urządzenia połączymy ze sobą podając w aplikacji nadającej zamiast **127.0.0.1** adres IP maszyny, na której działa aplikacja nasłuchująca.

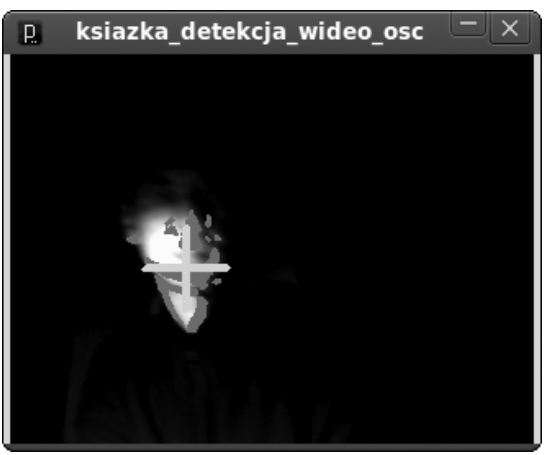
```
import oscP5.*;
import netP5.*;

OscP5 oscP5;
PVector zagrozenie;

void setup(){
        //(...)
        oscP5 = new OscP5(this, 12000);
        zagrozenie = new PVector(0,0);
}
void draw(){
        //(...)
        noFill();
        stroke(200);
        ellipse(zagrozenie.x,
        zagrozenie.y, height/2, height/2);
}
void oscEvent(OscMessage komunikat) {
        if (komunikat.checkAddrPattern("wspolrzedne")==true) {
                if (komunikat.checkTypetag("ff")) {
```

```
                float x = width - komunikat.
                        get(0).float Value() * width;
                float y = komunikat.
                        get(1).float Value() * height;
                zagrozenie.set(x, y, 0);
                for (Swietlik sw: swietliki) {
                        sw.drapieznik = zagrozenie.get();
                        }
                }
        }
}
```



PRZYKŁAD NA UŻYCIE KAMERY JAKO INTERFEJSU POŁĄCZONEGO
Z INNĄ APLIKACJĄ PRZEZ PROTOKÓŁ SIECIOWY





OBA PROGRAMY MOGĄ ZOSTAĆ URUCHOMIONE NA RÓŻNYCH KOMPUTERACH, ZNAJDUJĄCYCH SIĘ W DOWOLNYCH MIEJSCACH POŁĄCZONYCH Z SOBĄ PRZEZ INTERNET

9.8 PUBLIKACJA SKRYPTU W INTERNECIE I EKSPORT DO SAMODZIELNEJ APLIKACJI

Do tej pory uruchomialiśmy napisane programy bezpośrednio z Processingu. Ma to zaletę – możemy dowolnie i szybko modyfikować program, zmieniać wartości zmiennych, itp. Jeśli jednak chcielibyśmy uruchomić naszą aplikację na innym komputerze, musielibyśmy się upewnić, że zawiera IDE Processing. Nie jest to problemem – Processing można pobrać z sieci i działa on na niemal wszystkich platformach. Są również takie okoliczności, gdy czujemy, że nasza praca nad projektem definitywnie się skończyła i chcielibyśmy zachować nasz program w formie, która umożliwi uruchomienie go, lecz uniemożliwi jego modyfikacją. W Processingu uzyskujemy to bardzo łatwo. Zapisać końcowy program pod postacią pliku wykonywalnego możemy na dwa sposoby – jako aplet bądź jako samodzielny program.

Aplet to mały program przeznaczony do uruchomienia w przeglądarce internetowej. W przypadku Processingu aplet przyjmuje postać pliku o rozszerzeniu *jar*, który osadzamy w kodzie html jako element strony internetowej. Program możemy wyeksportować do takiej postaci na trzy sposoby:

- poprzez kliknięcie w ikonę u góry edytora o nazwie *Export Applet,*
- poprzez skrót klawiaturowy *Ctrl + e*
- poprzez pozycję w menu *File->Export Applet.*

W efekcie powstanie katalog, w którym znajdzie się plik *html* wraz ze wszystkimi plikami niezbędnymi do uruchomienia programu. Jeśli otworzymy wygenerowaną stronę w przeglądarce internetowej, zo-

baczymy okno naszego programu osadzone na charakterystycznym szarym tle. Zawartość tego folderu możemy umieścić na serwerze stron www. W ten sposób uczynimy efekt naszej pracy dostępny dla szerszej publiczności. Możemy zobaczyć też, że domyślny szablon takiej strony umożliwia odczytanie źródeł programu bezpośrednio z poziomu przeglądarki. Jak to zostało wspomniane we wstępie podręcznika, Processing związany jest z ideą społeczności twórców wzajemnie dzielących się wiedzą i doświadczeniem. Internet sprzyja temu celowi doskonale.

Aplety obok zalet mają też wady. Przede wszystkim ze względów bezpieczeństwa informatycznego, programy w przeglądarkach internetowych są uruchamiane w tzw. piaskownicach (ang. *sandbox*). W efekcie nie możemy zapisać ich jako plik na twardym dysku. Nie mają też one dostępu do urządzeń peryferyjnych (np. kamery internetowej).

Drugim sposobem jest zapisanie projektu jako pliku wykonywalnego – takiego, który po dwukrotnym kliknięciu myszką uruchomi się jako pełnoprawna aplikacja. Programu takiego nie ogranicza polityka bezpieczeństwa skojarzona z apletami. Tak uruchomiony program zachowuje dostęp do zasobów komputera oraz urządzeń peryferyjnych. W Processingu możemy wyeksportować nasz skrypt do takiej postaci posługując się skrótem klawiszowym *Shift+Ctrl+e*. Możemy też wybrać pozycję z menu *File->Export Application*. Podczas czynności zapisu pojawi się okno dialogowe z wyborem docelowego systemu operacyjnego – możemy zawęzić eksport do jednej platformy lub zaznaczyć wszystkie trzy: Windows, Mac OS X oraz Linux. Dodatkowo dostępna jest funkcja uruchomienia aplikacji w trybie pełnoekranowym, co jest pożądane w sytuacjach wysta-

wienniczych. Po eksporcie otworzy się okno systemowej przeglądarki, w którym zobaczymy pliki aplikacji.

W momencie eksportu skryptu do apletu lub aplikacji, Processing przestaje być językiem skryptowym, a staje się językiem kompilowanym – program przechowywany jest w postaci kodu maszyny wirtualnej Java (JVM – Java Virtual Machine). Gwarantuje to „przenośność" programów, (czyli np. program napisany pod Mac OS X będziemy mogli uruchomić pod Windows lub Linux i na odwrót). Od wersji Processingu z numerem **1.5**, stało się również możliwe programowanie w trybie przeznaczonym docelowo na urządzenia przenośnie z systemem operacyjnym Android. To otwiera przed przed osobą programującą w Processingu całkiem nowe perspektywy.

10 Summary

The aim of this publication is to provide students with scientific assistance for an introductory subject, led by me, aimed at 1st year BA students at the Intermedia Department of Arts University in Poznań. I assumed, taking into consideration the character of the course, that all students possess basic computer skills. This handbook will contain minimal information on IT. By taking such a decision, I hoped to give more prominence to programming heuristic[24]. The handbook does not cover the entire material done in the lessons, however it encompasses a significant part of it. It concentrates on basic programming practice in multimedia. It was not my aim to create a self-study coursebook for programming beginners. Therefore, the material covered in this handbook may seem dense at first glance. The content provides a basis for lectures and practical application workshops run by me during individual sessions or study groups. It is important to note here, that all the discussed topics in the handbook may seem to be treated briefly, however the delivery of topics in full would generally exceed the capacity of this handbook. Finally, for students to fully internalise the material covered by this handbook it is indispensable to carry out all exercises as well as to commit themselves to the process of writing programmes.

In the first chapters of this handbook, I concentrate mainly on the programming language syntax. The chapters are inspired by course books belonging to the se-

24 heuristic: involving or serving as an aid to learning, discovery, or problem-solving by experimental and especially trial-and-error methods (heuristic techniques), (a heuristic assumption); also : of or relating to exploratory problem-solving techniques that utilize self-educating techniques (Merriam-Webster Dictionary)

ries of 'How to think like a computer artist'[25]. There are several versions published for various languages so far (Python, C++, Java[26]). It must be stressed here, that this handbook does not resemble 'How to think like a computer artist' since I have doubts as to whether there is anything like 'computer art'. IT has been shaping for years now not only our perception of reality, but also the reality itself in many of its dimensions (ecological, political, cultural etc.). This process is ever growing stronger. For a contemporary artist who embarks on challenging the wider art perspective, the act of ignoring the technological background of the world we are living in cannot be done. Without it, no adequate theory or practice can be suggested. The term 'computer art' is misleading because it may suggest a practice narrowed down to some electronic device.

There are several issues crucially important for art practice. I refer here to such concepts as interactivity, productivity or communication. Each of them requires a deeper consideration, or at least justification of the used generalisation. I avoid unequivocal definitions in the handbook for two reasons—I leave the difficulty of formulating definitions to students. I also choose a direct approach —examples in the handbook point towards concepts, but they leave their direct articulation for personal conjecture. The prime teaching method assumed here is transcription of thoughts and intentions into language, which can be used to control a machine. Therefore there is such a significant emphasis on the programming language syntax.

Artists seem to be deterred from using algorithmic languages in the art-practice for the following difficulty. The artistic value of the programme is not determined

25 See http://www.openbookproject.net/thinkcs/archive

26 All coursebooks are open source

by using inaccuracies of natural language to evoke particular poetic effects. The value is determined by something completely opposite—the use of unequivocal instructions which in turn brings the user to non-algorithmic behaviour. We do not plan, therefore, a control of feelings, but through the control we create an attitude of freedom. This is a magical aspect of culture, which may evolve in a civilisation far more computerised than ours.

In the case of programming, the understanding is achieved through use. It is the leap of faith that is often helpful. Seeing programming as an activity only fit for the minds of engineers may come from lack of knowledge, or depreciating own skills. Whereas the truth is that technology should be approached without prejudice. To support this thesis, I shall recall the words of Joseph Weizenbaum, an author of the legendary bot[27] called ELIZA[28] —'It happens that programming is a relatively easy craft to learn. Almost anyone with a reasonably orderly mind can become a fairly good programmer with just a little instruction and practice.'[29] I realise, however, that programming is not for all artists an appropriate tool of artistic growth. Still, I hope that the knowledge included in this handbook, will allow to readers to reduce, in at least a minimal way, the feeling of alienation which often accompanies representatives of the arts when coming in contact with technology.

Let us have a look at some examples. At the core of programming activities are to be found algorithms—defined as effective and repetitive procedures of action. There is a book by Niklas Wirth entitled 'Algorithms + Data

27 A program which performed natural language processing

28 ELIZA—a friend you could never have before, http://www-ai.ijs.si/eliza-cgi-bin/eliza_script

29 See http://www.smeed.org/1735

Structures = Programs'[30]. Although computer science may be a fascinating branch of science, it may not be interesting for an art student. The problem in this context is as follows—how to resolve the question of style and creativity in programming. This question is far from having a plain solution. One's own implementation of an algorithm used by thousands of programmers is not destined to become reproductive. I faced this problem while writing programmes for the handbook. In the selection of examples attached to PDE *Processing*, one can find a great variety of applications of different techniques present in the handbook.

The code is highlighted in text with technical lettering. These fragments, I may wholeheartedly say, are more essential that the text written in regular type. At any rate, they should not be omitted while reading the handbook. I attempted to be concise with them semantically so that a commentary is reduced to a minimum. I also decided to embark on a bold step towards the use of Polish terminology in examples listed in Polish. Sometimes, it may seem a bit strange, considering programming languages are based on English. It is not my aim, however, to educate a corporate programmer, but acquainting a future artist with technology which can be used for other means then utilitarian IT. Since *Processing* is a dialect of *Java*, I shall obey the naming standards of *JavaBeans;* overall. Almost every listing represents a ready script which after coping to *Processing* can be executed.

Although there are many programming languages, certain concepts such as variable, function, etc., stay the same or very similar irrespectively of the language we are working in. The similarity of syntax found

30 Wirth Niklaus *Algorithms + Data Structures = Programs*, first publication 1975

in the majority of the most popular languages originates from a procedural language called C, which was created in 1972. It gained its popularity thanks to the UNIX operating system. Despite its age, C is still the most widely used programming language. It means that the most difficult task is to learn the first language from that family. Latter use of following ones shall boil down to getting acquainted with specific features of each of them. My choice of *Processing* comes down to several reasons. It was created in 2001 at the Massachusetts Institute of Technology (MIT) in the USA thanks to an initiative of two Media Lab students, Ben Fry and Casey Reas. Since then, it has been developed as an open source project by developers from around the world. It was given an award at *Ars Electronica* in 2005[8]. Over the past ten years, it has become one of the most commonly chosen tools of new media artists. What is more, *Processing* has been created by artists for artists. The number of fields, which we can explore using the language, is innumerable—from graphic animation, through digital video processing, to artificial neural network and many more. This would not have been achieved without mutual support of artists and programmers, who have been sharing their work and experience. As I mentioned, *Processing* is a dialect of *Java*, which means that we will have to learn both languages mutually. What I would like to emphasise at the end is the fact that by the end of the course, a lot of languages will seem strangely familiar to the participants.

(tł. Ela Wysakowska-Walters)

Bibliografia

LITERATURA ROZSZERZAJĄCA:

Gary William Flake *The Computational Beauty of Nature: Computer Explorations of Fractals, Chaos, Complex Systems, and Adaptation*, MIT Press, 1998

James Gleick *Chaos*, Zysk i S-ka, Poznań 1996

Roger Penrose *Nowy Umysł Cesarza*, PWN, Warszawa 2000

PODRĘCZNIKI PROGRAMOWANIA:

Allen B. Downey *How to Think Like a Computer Scientist* Java Version, 2003, do ściągnięcia z http://openbookproject.net/thinkcs/archive/java.php

Casey Reas, Ben Fry, *Processing: A Programming Handbook for Visual Designers and Artists*, MIT Press, 2007

Casey Reas and Ben Fry *Getting Started with Processing* O'Reilly Media, 2010

Daniel Shiffman, *Learning Processing: A Beginner's Guide to Programming Images, Animation, and Interaction*, Morgan Kaufmann, 2008

Keith Peters, *Foundation Actionscript 3.0 Animation Making Things Move!*, Apress, 2007

WYDAWCA:
Katedra Intermediów
Wydziału Komunikacji Multimedialnej
Uniwersytetu Artystycznego w Poznaniu

POZNAŃ 2011

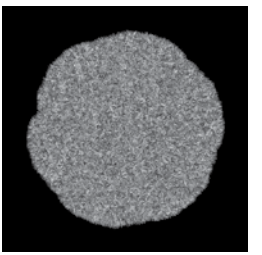
WYDANO Z DOTACJI CELOWEJ DLA MŁODYCH PRACOWNIKÓW NAUKI

ISBN 978-83-88400-94-0

NAKŁAD 300 EGZ.

REDAKCJA:
DR PIOTR BOSACKI
MGR JAKUB JASIUKIEWICZ

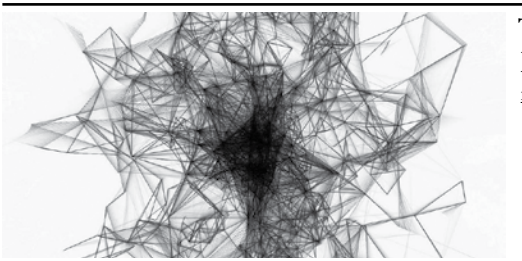
PROJEKT ORAZ OPIEKA GRAFICZNA:
NOVIKI.NET



Piotr Welk

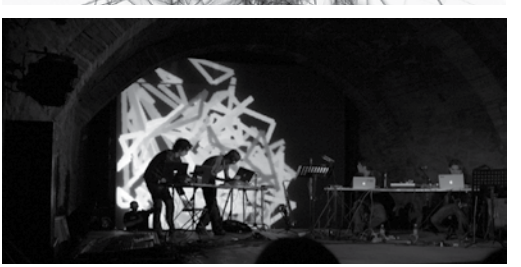
http://vimeo.com/38190897/

http://vimeo.com/38190897/

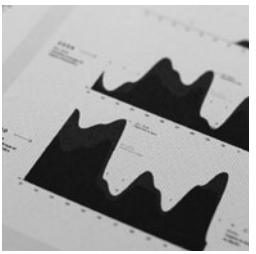
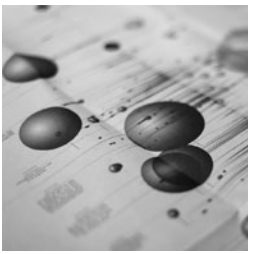


Thomas Sanchez lengeling

flickr.com/photos/70021357@N07/

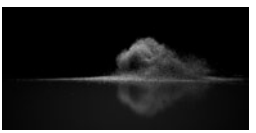


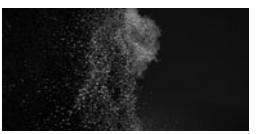
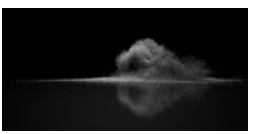
https://vimeo.com/user6662125

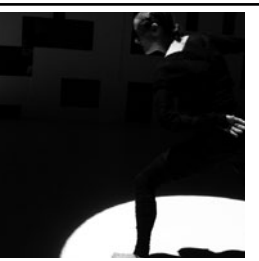
Onformative Studio

http://www.onformative.com



unnamed soundsculpture projekt:
http://prix2012.aec.at/prixwinner/6835/

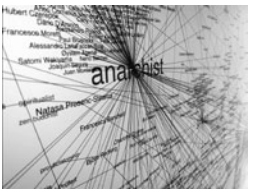
Ole Kristensen

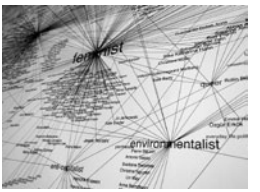
http://3xw.ole.kristensen.name


body-navigation
https://vimeo.com/2449078

PROCESSING

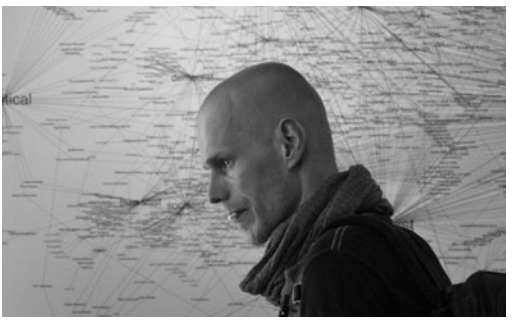
PRZYKŁADY REALIZACJI I STRONY PROJEKTÓW



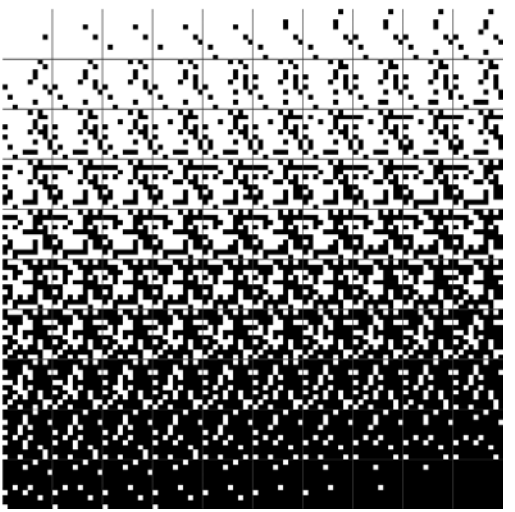


Burak Arikan

http://burak-arikan.com



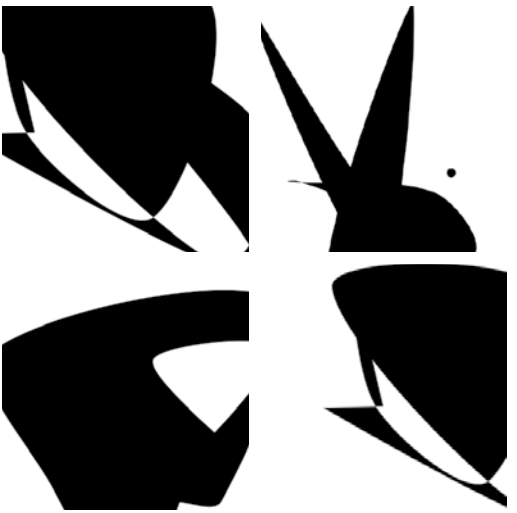
Network Map of Artists and Political Inclinations



Krzysztof Goliński

http://www.golinski.org/generative/winiarski/

Tribute to Ryszard Winiarski



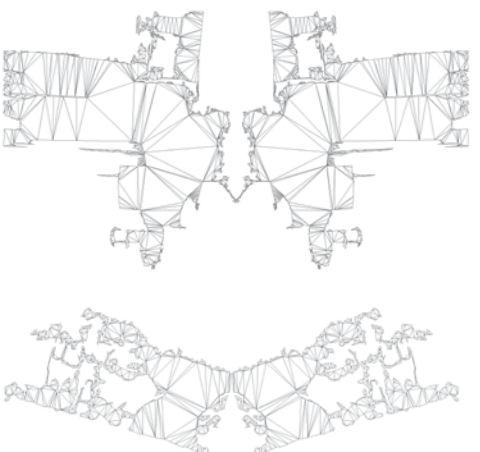
noquery

http://noquery.tumblr.com/

playing around a bug

Przemysław Sanecki

Test Rorschacha #1
Test Rorschacha #2/



ISBN 978-83-88400-94-0

UAP | POZNAŃ